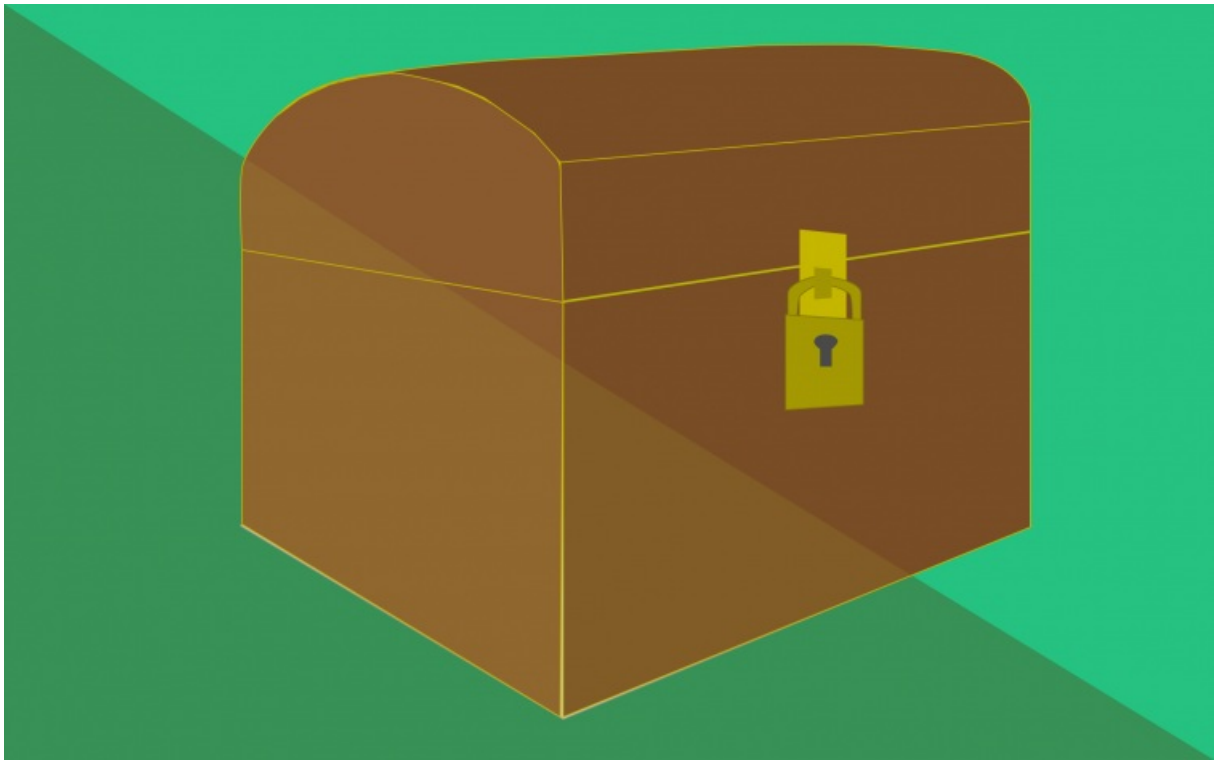
Symfony 3 Security: Beautiful Authentication, Powerful Authorization



With <3 from SymfonyCasts

Chapter 1: The All-Important User Class

Yo guys! You finally made it to the security course: you brave, brave souls. Whatever, these days, security isn't scary - it's super fun! You've got traditional login forms, Facebook authentication, GitHub authentication, and API authentication with JSON web tokens, just to name a few. When you're doing something with security these days, it's usually pretty darn fun. And we're going to learn enough so that you can implement whatever crazy, insane security system you want.

But, the only way to secure your new security skills it to feel secure in my recommendation that you *code along with me*. You guys know the drill: download the course code unzip it and look for the start/ directory. That'll have the same code that I have here. Open up the README file to find all the setup details. At the end, you'll open up a new tab and run:

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

to start the built-in web server.

Authentication vs Authorization (Fight!)

Aquanauts assemble... to talk about security!

Security has two big parts. The first is *authentication* - this is all about *who you are* - and we'll cover it first. Authentication is the tough stuff, and there's *a lot* of variation - login forms, social media auth, API stuff - you get it.

The second big piece is *authorization*, and this doesn't care about who you are, just whether or not you have fins. I mean, whether or not you have permission to take an action. Authorization comes later.

What about FOSUserBundle?

Before swim there, let's talk about a super-famous bundle: FOSUserBundle. You might be wondering, should I use this? What does it do? Or, did I turn the oven off?

First, we will *not* use this bundle, but it *is* great and you *did* turn the oven off... probably.

It gives you a lot of free features that we will build by hand. But FOSUserBundle does *not* give you any special "security" system - it's much less interesting than that, in a good way! The bundle gives you just *two* things:

1. A User entity in case you need to store users in the database
2. A bunch of routes and controllers

for things like your login form, registration and reset password. Those are all things you can easily build yourself... but if you need them, why not use the bundle?

Anyways, we *won't* use it, and that'll be the best path to learn how the security system works. But when you finish, you might save yourself some time using FOSUserBundle.

Create that User Class

Now, back to authentication. Here's our first goal: create a login form where the user can sign in with their email and password. In this app, we'll load user info from the database.

No matter *how* your users will authenticate, the first step is always the same: create a User class.

In your Entity directory - create a new class called User. The *only* rule is that this must implement a UserInterface. Add that:

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

```
... lines 1 - 2
namespace AppBundle\Entity;
... lines 4 - 5
use Symfony\Component\Security\Core\User\UserInterface;

class User implements UserInterface
{
... lines 10 - 28
}
```

I'll use Command+N - or the "Code"->"Generate" menu - and select "Implement Methods". Select all the new methods:

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

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

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

class User implements UserInterface
{
    public function getUsername()
    {
    }

    public function getRoles()
    {
    }

    public function getPassword()
    {
    }

    public function getSalt()
    {
    }

    public function eraseCredentials()
    {
    }
}
```

Oh, and let's move getUsername() to the top: it makes more sense up there.

Is User an Entity?

Notice I *did* put my User class inside of the Entity directory because eventually we *will* store users in the database. But, that's not required: sometimes user details are stored somewhere else - like a central authentication server. In those cases, you *will* still have a User class, you just won't store it with Doctrine.

More on that as we go along.

Ok, we've got the empty user class: let's fill it in!

Chapter 2: The UserInterface Methods (Keep some Blank!)

Our job: fill in the *5* methods from UserInterface. But check this out - I'm going rogue - I'm only going to implement *2* of them: getUsername() and getRoles().

getUsername()

First getUsername()... is super-unimportant. Just return *any* unique user string you want - a username, an email, a uuid, a funny, but unique joke - whatever. This is only used to show you *who* is logged in when you're debugging.

In our app, our users won't have a username - but they *will* have an email. Add a private $email property then just use that in getUsername(): return $this->email:

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

```
... lines 1 - 7
class User implements UserInterface
{
    private $email;

    // needed by the security system
    public function getUsername()
    {
        return $this->email;
    }
... lines 17 - 46
}
```

And add a setter for that method:

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

```
... lines 1 - 7
class User implements UserInterface
{
... lines 10 - 42
    public function setEmail($email)
    {
        $this->email = $email;
    }
}
```

We'll eventually need that to create users.

getRoles()

Cool: half-way done! Now: getRoles(). We'll talk roles later with authorization, but these are basically permissions we want to give the user. For now, give every user the same, one role: ROLE_USER:

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

```
... lines 1 - 7
class User implements UserInterface
{
... lines 10 - 16
    public function getRoles()
    {
        return ['ROLE_USER'];
    }
... lines 21 - 40
}
```

What about getPassword(), getSalt() and eraseCredentials()?

So what about getPassword(), getSalt() and eraseCredentials()? Keep them blank:

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

```
... lines 1 - 7
class User implements UserInterface
{
... lines 10 - 21
    public function getPassword()
    {
        // leaving blank - I don't need/have a password!
    }

    public function getSalt()
    {
        // leaving blank - I don't need/have a password!
    }

    public function eraseCredentials()
    {
        // leaving blank - I don't need/have a password!
    }
... lines 36 - 40
}
```

Whaat? It turns out, you *only* need these if your app is personally responsible for storing and checking user passwords. In our app - to start - we're *not* going to have passwords: we're just going to let anyone login with a single, central, hardcoded password. But there are also real-world situations where your app isn't responsible for managing and checking passwords.

If you have one of these, feel ok leaving these blank.

Setting up the User Entity

So in our application, we want to store users in the database. So let's set this class up with Doctrine. Copy the use statement from SubFamily and paste it here:

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

```
... lines 1 - 6
use Doctrine\ORM\Mapping as ORM;
... lines 8 - 63
```

Next, I'll put my cursor inside the class, press Command+N - or use the "Code"->"Generate" menu - and select "ORM class":

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

```
... lines 1 - 8
/**
 * @ORM\Entity
 * @ORM\Table(name="user")
 */
class User implements UserInterface
... lines 14 - 63
```

Next, add a private $id property and press Command+N one more time. This time, choose "ORM annotation" and highlight both fields:

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

```
... lines 1 - 12
class User implements UserInterface
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", unique=true)
     */
    private $email;
... lines 26 - 61
}
```

Oh, and while we're here, let's add unique=true for the email field:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 21
    /**
     * @ORM\Column(type="string", unique=true)
     */
    private $email;
... lines 26 - 61
}
```

Perfect: we have a fully functional User class. Sure, it only has an id and email, but that's enough!

Since we just added a new entity, let's generate a migration:

```
$ ./bin/console doctrine:migrations:diff
```

I'll copy the class and open it up quickly, just to make sure it looks right:

35 lines | app/DoctrineMigrations/Version20160524105319.php

```
... lines 1 - 10
class Version20160524105319 extends AbstractMigration
{
... lines 13 - 15
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysc

        $this->addSql('CREATE TABLE user (id INT AUTO_INCREMENT NOT NULL, email VARCHAR(255) NOT NULL, UNIQUE INDEX
    }
... lines 23 - 26
    public function down(Schema $schema)
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysc

        $this->addSql('DROP TABLE user');
    }
}
```

Looks great!

Adding User Fixtures

Finally, let's add some users to the database. Open our trusty fixtures.yml. I'll copy the SubFamily section, change the class to User and give these keys user_1..10:

26 lines | src/AppBundle/DataFixtures/ORM/fixtures.yml

```
... lines 1 - 22
AppBundle\Entity\User:
    user_{1..10}:
... lines 25 - 26
```

Then, the only field is email. Set it to weaverryan+<current()>@gmail.com:

26 lines | src/AppBundle/DataFixtures/ORM/fixtures.yml

```
... lines 1 - 22
AppBundle\Entity\User:
    user_{1..10}:
        email: weaverryan+<current()>@gmail.com
```

The current() function will return 1 through 10 as Alice loops through our set. That'll give us weaverryan+1@gmail.com up to weaverryan+10@gmail.com. And if you didn't know, Gmail ignores everything after a + sign, so these will all be delivered to weaverryan@gmail.com. There's your Internet hack for the day.

Ok, let's roll! Don't forget to run the migration first:

```
$ ./bin/console doctrine:migrations:migrate
```

And then load the fixtures:

```
$ ./bin/console doctrine:fixtures:load
```

Hey hey: you've got users in the database. Let's let 'em login.

Chapter 3: Rendering that Login Form

Time to build a login form. And guess what? This page is *no* different than every other page: we'll create a route, a controller and render a template.

For organization, create a new class called SecurityController. Extend the normal Symfony base Controller and add a public function loginAction():

17 lines | src/AppBundle/Controller/SecurityController.php

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

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
... lines 6 - 7
class SecurityController extends Controller
{
... lines 10 - 12
    public function loginAction()
    {
    }
}
```

Setup the URL to be /login and call the route security_login:

17 lines | src/AppBundle/Controller/SecurityController.php

```
... lines 1 - 5
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class SecurityController extends Controller
{
    /**
     * @Route("/login", name="security_login")
     */
    public function loginAction()
    {
    }
}
```

Make sure to auto-complete the @Route annotation so you get the use statement up top.

Cool!

Every login form looks about the same, so let's go steal some code. Google for "Symfony security form login" and Find a page called How to Build a Traditional Login Form.

Adding the Login Controller

Find their loginAction(), copy its code and paste it into ours:

33 lines | src/AppBundle/Controller/SecurityController.php

```
... lines 1 - 7
class SecurityController extends Controller
{
... lines 10 - 12
    public function loginAction()
    {
        $authenticationUtils = $this->get('security.authentication_utils');

        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();

        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();

        return $this->render(
            'security/login.html.twig',
            array(
                // last username entered by the user
                'last_username' => $lastUsername,
                'error'         => $error,
            )
        );
    }
}
```

Notice, one thing is immediately weird: there's no form processing code inside of here. Welcome to the strangest part of Symfony's security. We will build the login form here, but some *other* magic layer will actually handle the form submit. We'll build that layer next.

But thanks to this handy security.authentication_utils service, we can at least grab any authentication error that may have just happened in that magic layer as well as the last username that was typed in, which will actually be an email address for us.

The Login Controller

To create the template, hit Option+enter on a Mac and select the option to create the template. Or you can go create this by hand.

You guys know what to do: add {% extends 'base.html.twig' %}. Then, override {% block body %} and add {% endblock %}. I'll setup some markup to get us started:

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

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

{% block body %}
<div class="container">
    <div class="row">
        <div class="col-xs-12">
            <h1>Login!</h1>
... lines 8 - 27
        </div>
    </div>
</div>
{% endblock %}
```

Great! This template *also* has a bunch of boilerplate code, so copy that from the docs too. Paste it here. Update the form action route to security_login:

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

```
... lines 1 - 2
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-xs-12">
            <h1>Login!</h1>

            {% if error %}
                <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
            {% endif %}

            <form action="{{ path('security_login') }}" 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>
        </div>
    </div>
</div>
{% endblock %}
```

Well, it ain't fancy, but let's try it out: go to /login. There it is, in all its ugly glory.

What, No Form Class?

Now, I bet you've noticed something else weird: we are *not* using the form system: we're building the HTML form by hand. And this is *totally* ok. Security is strange because we will *not* handle the form submit in the normal way. Because of that, most people simply build the form by hand: you can do it either way.

But... our form is *ugly*. And *I* know from our forms course, that the form system is already setup to render using Bootstrap-friendly markup. So if we *did* use a real form... this would instantly be less ugly.

Ok, Ok: Let's add a Form Class

So let's do that: in the Form directory, create a new form class called LoginForm. Remove getName() - that's not needed in Symfony 3 - and configureOptions():

20 lines | src/AppBundle/Form/LoginForm.php

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

use Symfony\Component\Form\AbstractType;
... line 6
use Symfony\Component\Form\FormBuilderInterface;
... lines 8 - 9
class LoginForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 14 - 17
    }
}
```

This is a rare time when I *won't* bother binding my form to a class.

Tip

If you're building a login form that will be used with Symfony's native form_login system, override getBlockPrefix() and make it return an empty string. This will put the POST data in the proper place so the form_login system can find it.

In buildForm(), let's add two things, _username and _password, which should be a PasswordType:

20 lines | src/AppBundle/Form/LoginForm.php

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

You can name these fields anything, but _username and _password are common in the Symfony world. Again, we're calling this _username, but for us, it's an email.

Next, open SecurityController and add $form = $this->createForm(LoginForm::class):

36 lines | src/AppBundle/Controller/SecurityController.php

```
... lines 1 - 6
use AppBundle\Form\LoginForm;

class SecurityController extends Controller
{
... lines 11 - 13
    public function loginAction()
    {
... lines 16 - 23
        $form = $this->createForm(LoginForm::class, [
... line 25
        ]);
... lines 27 - 34
    }
}
```

And, if the user *just* failed login, we need to pre-populate their _username field. To pass the form default data, add a second argument: an array with _username set to $lastUsername:

36 lines | src/AppBundle/Controller/SecurityController.php

```
... lines 1 - 23
        $form = $this->createForm(LoginForm::class, [
            '_username' => $lastUsername,
        ]);
... lines 27 - 36
```

Finally, skip the form processing: that will live somewhere else. Pass the form into the template, replacing $lastUsername with 'form' => $form->createView():

36 lines | src/AppBundle/Controller/SecurityController.php

```
... lines 1 - 8
class SecurityController extends Controller
{
... lines 11 - 13
    public function loginAction()
    {
... lines 16 - 27
        return $this->render(
            'security/login.html.twig',
            array(
                'form' => $form->createView(),
                'error' => $error,
            )
        );
    }
}
```

Rendering the Form in the Template

Open up the template, Before we get to rendering, make sure our eventual error message looks nice. Add alert alert-danger:

24 lines | app/Resources/views/security/login.html.twig

```
... lines 1 - 2
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-xs-12">
            <h1>Login!</h1>

            {% if error %}
                <div class="alert alert-danger">
                    {{ error.messageKey|trans(error.messageData, 'security') }}
                </div>
            {% endif %}
... lines 14 - 19
        </div>
    </div>
</div>
{% endblock %}
```

Now, kill the *entire* form and replace it with our normal form stuff: form_start(form), from_end(form), form_row(form._username) and form_row(form._password):

24 lines | app/Resources/views/security/login.html.twig

```
... lines 1 - 2
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-xs-12">
            <h1>Login!</h1>

            {% if error %}
                <div class="alert alert-danger">
                    {{ error.messageKey|trans(error.messageData, 'security') }}
                </div>
            {% endif %}

            {{ form_start(form) }}
                {{ form_row(form._username) }}
                {{ form_row(form._password) }}
... line 18
            {{ form_end(form) }}
        </div>
    </div>
</div>
{% endblock %}
```

Don't forget your button! type="submit", add a few classes, say Login and get fancy with a lock icon:

24 lines | app/Resources/views/security/login.html.twig

```
... lines 1 - 2
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-xs-12">
... lines 7 - 14
            {{ form_start(form) }}
... lines 16 - 17
                <button type="submit" class="btn btn-success">Login <span class="fa fa-lock"></span></button>
            {{ form_end(form) }}
        </div>
    </div>
</div>
{% endblock %}
```

We did this *purely* so that Ryan could get his form looking less ugly. Let's see if it worked. So much better!

Oh, while we're here, let's hook up the Login button on the upper right. This lives in base.html.twig. The login form is just a normal route, so add path('security_login'):

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

```
<!DOCTYPE html>
<html>
... lines 3 - 13
    <body>
... lines 15 - 19
        <header class="header">
... lines 21 - 22
            <ul class="navi">
... line 24
                <li><a href="{{ path('security_login') }}">Login</a></li>
            </ul>
        </header>
... lines 28 - 46
    </body>
</html>
```

Refresh, click that link, and here we are.

Login form complete. It's finally time for the *meat* of authentication: it's time to build an *authenticator*.

Chapter 4: All About Firewalls

Open up app/config/security.yml. Security - especially *authentication* - is all configured here. We'll look at this piece-by-piece, but there's one section that's more important than all the rest: firewalls:

25 lines | app/config/security.yml

```
# To get started with security, check out the documentation:
# http://symfony.com/doc/current/book/security.html
security:
... lines 4 - 9
    firewalls:
        # disables authentication for assets and the profiler, adapt it according to your needs
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: ~
            # activate different ways to authenticate

            # http_basic: ~
            # http://symfony.com/doc/current/book/security.html#a-configuring-how-your-users-will-authenticate

            # form_login: ~
            # http://symfony.com/doc/current/cookbook/security/form_login_setup.html
```

All About Firewalls

Your firewall *is* your authentication system: it's like the security desk you pass when going into a building. Now, there's always only *one* firewall that's active on any request. You see, if you go to a URL that starts with /_profiler, /_wdt or /css, you hit the dev firewall *only*:

25 lines | app/config/security.yml

```
... lines 1 - 2
security:
... lines 4 - 9
    firewalls:
        # disables authentication for assets and the profiler, adapt it according to your needs
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
... lines 15 - 25
```

This basically turns security off: it's like sneaking through the side door of a building that has no security desk. This is here to prevent us from getting over-excited with security and accidentally securing our debugging tools.

In reality, every *real* request will activate the main firewall:

25 lines | app/config/security.yml

```
... lines 1 - 2
security:
... lines 4 - 9
    firewalls:
... lines 11 - 15
        main:
            anonymous: ~
            # activate different ways to authenticate

            # http_basic: ~
            # http://symfony.com/doc/current/book/security.html#a-configuring-how-your-users-will-authenticate

            # form_login: ~
            # http://symfony.com/doc/current/cookbook/security/form_login_setup.html
```

Because it has no pattern key, it matches *all* URLs. Oh, and these keys - main and dev, are meaningless.

Our job is to activate different ways to authenticate under this *one* firewall. We might allow the user to authenticate via a form login, HTTP basic, an API token, Facebook login or *all* of these.

So - if you ignore the dev firewall, we really only have *one* firewall, and I want yours to look like mine. There *are* use-cases for having multiple firewalls, but you probably don't need it. If you're curious, we *do* set this up on our Symfony REST API course.

We won't use form_login

Ok, *we* want to activate a system that allows the user to submit their email and password to login. If you look at the official documentation about this, you'll notice they add a key called form_login under their firewall. Then, everything just magically works. I mean, literally: you submit your login form, Symfony intercepts the request and takes care of everything else.

It's really cool because it's quick to set up! But it's super magical and hard to extend and control. If you're using FOSUserBundle, they also recommend that you use this.

But, you have a choice. We *won't* use this. Instead, we'll use a system that's new in Symfony 2.8 called Guard. It *is* more work to setup, but you'll have control over *everything* from day 1.

Chapter 5: The LoginFormAuthenticator

To use Guard - no matter *what* crazy authentication system you have - the first step is always to create an authenticator class. Create a new directory called Security and inside, a new class: how about LoginFormAuthenticator:

32 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 2
namespace AppBundle\Security;
... lines 4 - 7
use Symfony\Component\Security\Guard\Authenticator\AbstractFormLoginAuthenticator;

class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 12 - 30
}
```

The only rule about an authenticator is that it needs to extend AbstractGuardAuthenticator. Well, not totally true - if you're building some sort of login form, you can extend a different class instead: AbstractFormLoginAuthenticator - it extends that other class, but fills in some details for us.

Hit Command+N - or go to the "Code"->"Generate" menu - choose "Implement Methods" and select the first three:

32 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 4
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
... lines 8 - 9
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
    public function getCredentials(Request $request)
    {
    }

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

    public function checkCredentials($credentials, UserInterface $user)
    {
    }
... lines 23 - 30
}
```

Then, do it again, and choose the other two:

32 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 9
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 12 - 23
    protected function getLoginUrl()
    {
    }

    protected function getDefaultSuccessRedirectUrl()
    {
    }
}
```

Tip

Starting in Symfony 3.1, you won't see getDefaultSuccessRedirectUrl() in this list anymore. Don't worry! We'll tell you how to handle this later.

That was just my way to get these methods in the order I want, but it doesn't matter.

How Authenticators Work

When we're finished, Symfony will call our authenticator on *every single request*. Our job is to:

1. See if the user is submitting the login form, or if this is just some random request for some random page.
2. Read the username and password from the request.
3. Load the User object from the database.

getCredentials()

That all starts in getCredentials(). Since this method is called on *every* request, we *first* need to see if the request is a login form submit. We setup our form so that it POSTs right back to /login. So if the URL is /login and the HTTP method is POST, our authenticator should spring into action. Otherwise, it should do nothing: this is just a normal page.

Create a new variable called $isLoginSubmit Set that to $request->getPathInfo() - that's the URL -
== '/login' && $request->isMethod('POST'):

53 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 11
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 14 - 20
    public function getCredentials(Request $request)
    {
        $isLoginSubmit = $request->getPathInfo() == '/login' && $request->isMethod('POST');
... lines 24 - 34
    }
... lines 36 - 51
}
```

Tip

Instead of hardcoding the /login URL, you could instead check for the current page's route name:
if ($request->attributes->get('_route') === 'security_login' && $request->isMethod('POST'))

If both of those are true, the user has just submitted the login form.

So, if (!$isLoginSubmit), just return null:

53 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 22
        $isLoginSubmit = $request->getPathInfo() == '/login' && $request->isMethod('POST');
        if (!$isLoginSubmit) {
            // skip authentication
            return;
        }
... lines 28 - 53
```

If you return null from getCredentials(), Symfony skips trying to authenticate the user and the request continues on like normal.

getCredentials(): Build the Form

If the user *is* trying to login, our *new* task is to fetch the username & password and return them.

Since we built a form, let's let the form do the work for us.

Normally in a controller, we call $this->createForm() to build the form:

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

```
... lines 1 - 38
abstract class Controller implements ContainerAwareInterface
{
... lines 41 - 274
    /**
     * Creates and returns a Form instance from the type of the form.
     *
     * @param string|FormTypeInterface $type    The built type of the form
     * @param mixed                    $data    The initial data for the form
     * @param array                    $options Options for the form
     *
     * @return Form
     */
    protected function createForm($type, $data = null, array $options = array())
    {
        return $this->container->get('form.factory')->create($type, $data, $options);
    }
... lines 288 - 396
}
```

In reality, this grabs the form.factory service and calls create() on it.

Dependency Inject form.factory (FormFactory)

So how can we create a form in the authenticator? Use dependency injection to inject the form.factory service.

Add a __construct() method with a $formFactory argument:

53 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 11
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 14 - 15
    public function __construct(FormFactoryInterface $formFactory)
    {
... line 18
    }
... lines 20 - 51
}
```

Now, I like to type-hint my arguments, so let's just guess at the service's class name and see if there's one called FormFactory. Yep, there's even a FormFactoryInterface!

53 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 5
use Symfony\Component\Form\FormFactoryInterface;
... lines 7 - 11
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 14 - 15
    public function __construct(FormFactoryInterface $formFactory)
    {
... line 18
    }
... lines 20 - 51
}
```

That's probably what we want. I'll press Option+Enter and select "Initialize Fields" to set that property for me:

53 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 11
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
    private $formFactory;

    public function __construct(FormFactoryInterface $formFactory)
    {
        $this->formFactory = $formFactory;
    }
... lines 20 - 51
}
```

If you're still getting used to dependency injection and that all happened too fast, don't worry. We know we want to inject the form.factory service, so I guessed its class for the type-hint, which is optional. You can always find your terminal and run:

```
$ ./bin/console debug:container form.factory
```

to find out the *exact* class to use for the type-hint. We *will* also register this as a service in services.yml in a minute.

Return the Credentials

Back in getCredentials(), add $form = $this->formFactory->create() and pass it LoginForm::class:

53 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 4
use AppBundle\Form\LoginForm;
... lines 6 - 11
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 14 - 20
    public function getCredentials(Request $request)
    {
        $isLoginSubmit = $request->getPathInfo() == '/login' && $request->isMethod('POST');
        if (!$isLoginSubmit) {
            // skip authentication
            return;
        }

        $form = $this->formFactory->create(LoginForm::class);
... lines 30 - 34
    }
... lines 36 - 51
}
```

Then - just like always - use $form->handleRequest($request):

53 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 22
        $isLoginSubmit = $request->getPathInfo() == '/login' && $request->isMethod('POST');
        if (!$isLoginSubmit) {
            // skip authentication
            return;
        }

        $form = $this->formFactory->create(LoginForm::class);
        $form->handleRequest($request);
... lines 31 - 53
```

Normally, we would check if $form->isValid(), but we'll do any password checking or other validation manually in a moment. Instead, just skip to $data = $form->getData() and return $data:

53 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 22
        $isLoginSubmit = $request->getPathInfo() == '/login' && $request->isMethod('POST');
        if (!$isLoginSubmit) {
            // skip authentication
            return;
        }

        $form = $this->formFactory->create(LoginForm::class);
        $form->handleRequest($request);

        $data = $form->getData();

        return $data;
... lines 35 - 53
```

Since our form is not bound to a class, this returns an associative array with _username and _password. And that's it for getCredentials(). If you return *any* non-null value, authentication continues to the next step.

Chapter 6: Authenticator: getUser, checkCredentials & Success/Failure

Here's the deal: if you return null from getCredentials(), authentication is skipped. But if you return *anything* else, Symfony calls getUser():

53 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 8
use Symfony\Component\Security\Core\User\UserProviderInterface;
... lines 10 - 11
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 14 - 36
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
    }
... lines 40 - 51
}
```

And see that $credentials argument? That's equal to what we return in getCredentials(). In other words, add $username = $credentials['_username']:

60 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 12
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 15 - 39
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $username = $credentials['_username'];
... lines 43 - 45
    }
... lines 47 - 58
}
```

I *do* continue to call this *username*, but in our case, it's an email address. And for you, it could be anything - don't let that throw you off.

Hello getUser()

Our job in getUser() is... surprise! To get the user! What I mean is - to *somehow* return a User object. Since our Users are stored in the database, we'll query for them via the entity manager. To get that, add a second constructor argument: EntityManager $em:

60 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 12
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 15 - 17
    public function __construct(FormFactoryInterface $formFactory, EntityManager $em)
    {
... lines 20 - 21
    }
... lines 23 - 58
}
```

And once again, I'll use my Option+Enter shortcut to create and set that property:

60 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 12
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... line 15
    private $em;

    public function __construct(FormFactoryInterface $formFactory, EntityManager $em)
    {
... line 20
        $this->em = $em;
    }
... lines 23 - 58
}
```

Now, it's real simple: return $this->em->getRepository('AppBundle:User')->findOneBy() with email => $email:

60 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 12
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 15 - 39
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $username = $credentials['_username'];

        return $this->em->getRepository('AppBundle:User')
            ->findOneBy(['email' => $username]);
    }
... lines 47 - 58
}
```

Easy. If this returns null, guard authentication will fail and the user will see an error. But if we *do* return a User object, then on we march! Guard calls checkCredentials():

60 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 12
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 15 - 47
    public function checkCredentials($credentials, UserInterface $user)
    {
    }
... lines 51 - 58
}
```

Enter checkCredentials()

This is our chance to verify the user's password if they have one or do any other last-second validation. Return true if you're happy and the user should be logged in.

For us, add $password = $credentials['_password']:

67 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 12
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 15 - 47
    public function checkCredentials($credentials, UserInterface $user)
    {
        $password = $credentials['_password'];
... lines 51 - 56
    }
... lines 58 - 65
}
```

Our users don't have a password yet, but let's add something simple: pretend every user shares a global password. So, if ($password == 'iliketurtles'), then return true:

67 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 49
        $password = $credentials['_password'];

        if ($password == 'iliketurtles') {
            return true;
        }

        return false;
... lines 57 - 67
```

Otherwise, return false: authentication will fail.

When Authentication Fails? getLoginUrl()

That's it! Authenticators are always these three methods.

But, what happens if authentication fails? Where should we send the user? And what about when the login is successful?

When authentication fails, we need to redirect the user back to the login form. That will happen automatically - we just need to fill in getLoginUrl() so the system knows where that is:

67 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 12
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 15 - 58
    protected function getLoginUrl()
    {
    }
... lines 62 - 65
}
```

But to do that, we'll need the router service. Once again, go back to the top and add another constructor argument for the router. To be super cool, you can type-hint with the RouterInterface:

72 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 8
use Symfony\Component\Routing\RouterInterface;
... lines 10 - 13
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 19
    public function __construct(FormFactoryInterface $formFactory, EntityManager $em, RouterInterface $router)
    {
... lines 22 - 24
    }
... lines 26 - 70
}
```

Use the Option+Enter shortcut again to set up that property:

72 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 13
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 17
    private $router;

    public function __construct(FormFactoryInterface $formFactory, EntityManager $em, RouterInterface $router)
    {
... lines 22 - 23
        $this->router = $router;
    }
... lines 26 - 70
}
```

Down in getLoginUrl(), return $this->router->generate('security_login'):

72 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 13
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 61
    protected function getLoginUrl()
    {
        return $this->router->generate('security_login');
    }
... lines 66 - 70
}
```

When Authentication is Successful?

Tip

Due to a change in Symfony 3.1, you *can* still fill in getDefaultSuccessRedirectUrl() like we do here, but it's deprecated. Instead, you'll add a different method - onAuthenticationSuccess() - we have the code in a comment: http://bit.ly/guard-success-change

So what happens when authentication is successful? It's awesome: the user is automatically redirected back to the last page they tried to visit before being forced to login. In other words, if the user tried to go to /checkout and was redirected to /login, then they'll automatically be sent back to /checkout so they can continue buying your awesome stuff.

But, in case they go *directly* to /login and there is no previous URL to send them to, we need a backup plan. That's the purpose of getDefaultSuccessRedirectUrl():

72 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 13
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 66
    protected function getDefaultSuccessRedirectUrl()
    {
... line 69
    }
}
```

Send them to the homepage: return $this->router->generate('homepage'):

72 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 13
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 16 - 66
    protected function getDefaultSuccessRedirectUrl()
    {
        return $this->router->generate('homepage');
    }
}
```

The authenticator is *done*. If you need even more control over what happens on error or success, there are a few other methods you can override. Or check out our Guard tutorial. Let's finally hook this thing up.

Registering the Service

To do that, open up app/config/services.yml and register the authenticator as a service.

Tip

If you're using Symfony 3.3, your app/config/services.yml contains some extra code that may break things when following this tutorial! To keep things working - and learn about what this code does - see https://knpuniversity.com/symfony-3.3-changes

Let's call it app.security.login_form_authenticator. Set the class to LoginFormAuthenticator and because I'm feeling super lazy, autowire the arguments:

21 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 17
    app.security.login_form_authenticator:
        class: AppBundle\Security\LoginFormAuthenticator
        autowire: true
```

We can do that because we type-hinted all the constructor arguments.

Configuring in security.yml

Finally, copy the service name and open security.yml. To activate the authenticator, add a new key under your firewall called guard. Add authenticators below that, new line, dash and paste the service name:

28 lines | app/config/security.yml

```
... lines 1 - 2
security:
... lines 4 - 9
    firewalls:
... lines 11 - 15
        main:
... line 17
            guard:
                authenticators:
                    - app.security.login_form_authenticator
            # activate different ways to authenticate
... lines 22 - 28
```

As *soon* as we do that, getCredentials() will be called on every request and our whole system should start singing.

Let's try it! Try logging in with weaverryan+1@gmail.com, but with the wrong password.

Beautiful! Now try the right password: iliketurtles.

Debugging with intercept_redirects

Ah! Woh! It *did* redirect to the homepage as *if* it worked, but with a *nasty* error. In fact, authentication *did* work, but there's a problem with fetching the User from the session. Let me prove it by showing you an awesome, hidden debugging tool.

Open up config_dev.yml and set intercept_redirects to true:

49 lines | app/config/config_dev.yml

```
... lines 1 - 12
web_profiler:
... line 14
    intercept_redirects: true
... lines 16 - 49
```

Now, whenever the app is about to redirect us, Symfony will stop instead, and show us the web debug toolbar for that request.

Go to /login again and login in with weaverryan+1@gmail.com and iliketurtles. Check this out: we're *still* at /login: the request finished, but it did *not* redirect us yet. And in the web debug toolbar, we *are* logged in as weaverryan+1@gmail.com.

So authentication works, but there's some issue with storing our User in the session. Fortunately, that's going to be really easy to fix.

Chapter 7: The Mysterious "User Provider"

Let's see that error again: change intercept_redirects back to false:

49 lines | app/config/config_dev.yml

```
... lines 1 - 12
web_profiler:
... line 14
    intercept_redirects: false
... lines 16 - 49
```

Refresh and re-post the form. Oof, there it is again:

> There is no user provider for user AppBundle\Entity\User.

What the heck is a user provider and why do we need one?

What is a User Provider?

A user provider is one of the most misunderstood parts of Symfony's security. It's an object that does just a *few* small jobs for you. For example, the user provider is responsible for loading the User from the session and making sure that it's up to date. In Doctrine, we'll want our's to re-query for a fresh User object to make sure all the data is still up-to-date.

The user provider is also responsible for a few other minor things, like handling "remember me" functionality and a really cool feature we'll talk about later called "impersonation".

Long story short: you need a user provider, but it's not all that important. And if you're using Doctrine, it's super easy to setup.

Setting up the Entity User Provider

In security.yml, you already have a providers section - as in "user providers". Delete the in_memory stuff and replace it with our_users: that's a totally meaningless machine name - it could be anything. But below that, say entity and set it to { class: AppBundle\Entity\User, property: email }:

28 lines | app/config/security.yml

```
... lines 1 - 2
security:

    # http://symfony.com/doc/current/book/security.html#where-do-users-come-from-user-providers
    providers:
        our_users:
            entity: { class: AppBundle\Entity\User, property: email }
... lines 9 - 28
```

The property part is *not* something we care about right now, but we will use and talk about it later.

But yea, that's it! Go back to /login. Right now, I am *not* logged in. But try logging in again.

It's alive!!! We can finally surf around the site and stay logged in. Cool.

Custom User Provider

In your app, if you're *not* loading users from the database, then you'll need to create a custom user provider class that implements UserProviderInterface. Check out the official docs in this case. But if you have any questions, let me know.

Chapter 8: Logging out & Pre-filling the Email on Failure

Check this out: let's fail authentication with a bad password.

Ok: I noticed two things. First, we have an error:

> Invalid credentials.

Great! But second, the form is *not* pre-filled with the email address I just used. Hmm.

Behind the scenes, the authenticator communicates to your SecurityController by storing things in the session. That's what the security.authentication_utils helps us with:

36 lines | src/AppBundle/Controller/SecurityController.php

```
... lines 1 - 8
class SecurityController extends Controller
{
... lines 11 - 13
    public function loginAction()
    {
        $authenticationUtils = $this->get('security.authentication_utils');

        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();
... lines 20 - 34
    }
}
```

Hold command and open getLastAuthenticationError(). Ultimately, this reads a Security::AUTHENTICATION_ERROR string key from the session.

And the same is true for fetching the last username, or email in our case: it reads a key from the session.

Here's the deal: the login form is automatically setting the authentication error to the session for us. But, it is *not* setting the last username on the session... because it doesn't really know where to look for it.

No worries, fix this with $request->getSession()->set() and pass it the constant - Security::LAST_USERNAME - and $data['_username']:

77 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 9
use Symfony\Component\Security\Core\Security;
... lines 11 - 14
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 17 - 27
    public function getCredentials(Request $request)
    {
... lines 30 - 38
        $data = $form->getData();
        $request->getSession()->set(
            Security::LAST_USERNAME,
            $data['_username']
        );
... lines 44 - 45
    }
... lines 47 - 75
}
```

Now, try it again. Good-to-go!

Can I Logout?

Next challenge! Can we logout? Um... right now? Nope! But that seems important! So, let's do it.

Start like normal: In SecurityController, create a logoutAction, set its route to /logout and call the route security_logout:

44 lines | src/AppBundle/Controller/SecurityController.php

```
... lines 1 - 8
class SecurityController extends Controller
{
... lines 11 - 36
    /**
     * @Route("/logout", name="security_logout")
     */
    public function logoutAction()
    {
... line 42
    }
}
```

Now, here's the fun part. Don't put *any* code in the method. In fact, throw a new \Exception that says, "this should not be reached":

44 lines | src/AppBundle/Controller/SecurityController.php

```
... lines 1 - 8
class SecurityController extends Controller
{
... lines 11 - 39
    public function logoutAction()
    {
        throw new \Exception('this should not be reached!');
    }
}
```

Adding the logout Key

Whaaaat? Yep, our controller will do nothing. Instead, Symfony will intercept any requests to /logout and take care of everything for us. To activate it, open security.yml and add a new key under your firewall: logout. Below that, add path: /logout:

31 lines | app/config/security.yml

```
... lines 1 - 2
3   security:
... lines 4 - 9
10      firewalls:
... lines 11 - 15
16          main:
... lines 17 - 21
22              logout:
23                  path: /logout
... lines 24 - 31
```

Now, if the user goes to /logout, Symfony will automatically take care of logging them out. That's super magical, almost creepy - but it works pretty darn well.

So, why did I make you create a route and controller if Symfony wasn't going to use it? Am I trying to drive you crazy!

Come on, I'm looking out for you! It turns out, if you don't have a route that matches /logout, then the 404 page is triggered *before* Symfony has a chance to log the user out. That's why you need this.

It should work already, but let's add a friendly logout link. In base.html.twig, how can we figure out if the user is logged in? We're *about* to talk about that... but what the heck - let's get a preview. Use {% if is_granted('ROLE_USER') %}:

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

```
1   <!DOCTYPE html>
2   <html>
... lines 3 - 13
14      <body>
... lines 15 - 19
20          <header class="header">
... lines 21 - 22
23              <ul class="navi">
... line 24
25                  {% if is_granted('ROLE_USER') %}
... lines 26 - 27
28                      <li><a href="{{ path('security_login') }}">Login</a></li>
29                  {% endif %}
30              </ul>
31          </header>
... lines 32 - 50
51      </body>
52  </html>
```

Remember this role? We returned it from getRoles() in User - so *all* authenticated users have this.

If they don't have this, show the login link. But if they do, show the logout link: path('security_logout'):

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

```
<!DOCTYPE html>
<html>
... lines 3 - 13
    <body>
... lines 15 - 19
        <header class="header">
... lines 21 - 22
            <ul class="navi">
... line 24
                {% if is_granted('ROLE_USER') %}
                    <li><a href="{{ path('security_logout') }}">Logout</a></li>
                {% else %}
                    <li><a href="{{ path('security_login') }}">Login</a></li>
                {% endif %}
            </ul>
        </header>
... lines 32 - 50
    </body>
</html>
```

Perfect!

Try the *whole* thing out: head to the homepage. We're anonymous right now.. so let's login! Cool! And there's the logout link. Click it! Ok, back to anonymous. If you need to control what happens after logging out, check the official docs on the logout stuff.

Alright. Now, as much as I like turtles, we should *probably* give our users a real password.

Chapter 9: Users Need Passwords (plainPassword)

I just found out that giving everyone the same password - iliketurtles - is apparently *not* a great security system. Let's give each user their own password. Again, in *your* security setup, you might not be responsible for storing and checking passwords. Skip this if it doesn't apply to you.

In User, add a private $password that will eventually store the encoded password. Give it the @ORM\Column annotation:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 26
    /**
     * The encoded password
     *
     * @ORM\Column(type="string")
     */
    private $password;
... lines 33 - 73
}
```

Now, remember the three methods from UserInterface that we left blank?

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 37
    public function getPassword()
    {
        // leaving blank - I don't need/have a password!
    }

    public function getSalt()
    {
        // leaving blank - I don't need/have a password!
    }

    public function eraseCredentials()
    {
        // leaving blank - I don't need/have a password!
    }
... lines 52 - 61
}
```

It's finally their time to shine. In getPassword(), return $this->password:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 44
    public function getPassword()
    {
        return $this->password;
    }
... lines 49 - 73
}
```

But keep getSalt() blank: we're going to use the bcrypt algorithm, which has a built-in mechanism to salt passwords.

Use the "Code"->"Generate" menu to generate the setter for password:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 69
    public function setPassword($password)
    {
        $this->password = $password;
    }
}
```

And next, go make that migration:

```
$ ./bin/console doctrine:migrations:diff
```

I *should* check that file, but let's go for it:

```
$ ./bin/console doctrine:migrations:migrate
```

Perfect.

Handling the Plain Password

Here's the plan: we'll start with a plain text password, encrypt it through the bcrypt algorithm and store that on the password property.

How? The best way is to set the plain-text password on the User and encode it automatically via a Doctrine listener when it saves.

To do that, add a new property on User called plainPassword. But wait! *Don't* persist this with Doctrine: we will of course *never* store plain-text passwords:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 33
    /**
     * A non-persisted field that's used to create the encoded password.
     *
     * @var string
     */
    private $plainPassword;
... lines 40 - 90
}
```

This is just a temporary-storage place during a single request.

Next, at the bottom, use Command+N or the "Code"->"Generate" menu to generate the getters and setters for plainPassword:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 81
    public function getPlainPassword()
    {
        return $this->plainPassword;
    }

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

Forcing User to look Dirty?

Inside setPlainPassword(), do one more thing: $this->password = null:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 86
    public function setPlainPassword($plainPassword)
    {
        $this->plainPassword = $plainPassword;
        // forces the object to look "dirty" to Doctrine. Avoids
        // Doctrine *not* saving this entity, if only plainPassword changes
        $this->password = null;
    }
}
```

What?! Yep, this is important. Soon, we'll use a Doctrine listener to read the plainPassword property, encode it, and update password. That means that password *will* be set to a value before it actually saves: it won't remain null.

So why add this weird line if it basically does nothing? Because Doctrine listeners are *not* called if Doctrine thinks that an object has *not* been updated. If you eventually create a "change password" form, then the *only* property that will be updated

is plainPassword. Since this is *not* persisted, Doctrine will think the object is "un-changed", or "clean". In that case, the listeners will *not* be called, and the password will not be changed.

But by adding this line, the object will *always* look like it has been changed, and life will go on like normal.

Anyways, it's a necessary little evil.

Finally, in eraseCredentials(), add $this->plainPassword = null:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 61
    public function eraseCredentials()
    {
        $this->plainPassword = null;
    }
... lines 66 - 93
}
```

Symfony calls this after logging in, and it's just a minor security measure to prevent the plain-text password from being accidentally saved anywhere.

The User object is perfect. Let's add the listener.

Chapter 10: Doctrine Listener: Encode the User's Password

In AppBundle, create a new directory called Doctrine and a new class called HashPasswordListener:

15 lines | src/AppBundle/Doctrine/HashPasswordListener.php

```
... lines 1 - 2
namespace AppBundle\Doctrine;
... lines 4 - 7
class HashPasswordListener implements EventSubscriber
{
... lines 10 - 13
}
```

If this is your first Doctrine listener, welcome! They're pretty friendly. Here's the idea: we'll create a function that Doctrine will call whenever *any* entity is inserted or updated. That'll let us to do some work before that happens.

Implement an EventSubscriber interface and then use Command+N or the "Code"->"Generate" menu, select "Implement Methods" and choose the one method: getSubscribedEvents():

15 lines | src/AppBundle/Doctrine/HashPasswordListener.php

```
... lines 1 - 4
use Doctrine\Common\EventSubscriber;
... lines 6 - 7
class HashPasswordListener implements EventSubscriber
{
    public function getSubscribedEvents()
    {
... line 12
    }
}
```

In here, return an array with prePersist and preUpdate:

15 lines | src/AppBundle/Doctrine/HashPasswordListener.php

```
... lines 1 - 7
class HashPasswordListener implements EventSubscriber
{
    public function getSubscribedEvents()
    {
        return ['prePersist', 'preUpdate'];
    }
}
```

These are two event names that Doctrine makes available. prePersist is called right before an entity is originally *inserted*. preUpdate is called right before an entity is updated.

Next, add public function prePersist():

38 lines | src/AppBundle/Doctrine/HashPasswordListener.php

```
... lines 1 - 6
use Doctrine\ORM\Event\LifecycleEventArgs;
... lines 8 - 9
class HashPasswordListener implements EventSubscriber
{
... lines 12 - 18
    public function prePersist(LifecycleEventArgs $args)
    {
... lines 21 - 30
    }
... lines 32 - 36
}
```

When Doctrine calls this, it will pass you an object called LifecycleEventArgs, from the ORM namespace.

This method will be called before *any* entity is inserted. How do we know *what* entity is being saved? With $entity = $args->getEntity(). Now, if this is *not* an instanceof User, just return and do nothing:

38 lines | src/AppBundle/Doctrine/HashPasswordListener.php

```
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 9
class HashPasswordListener implements EventSubscriber
{
... lines 12 - 18
    public function prePersist(LifecycleEventArgs $args)
    {
        $entity = $args->getEntity();
        if (!$entity instanceof User) {
            return;
        }
... lines 25 - 30
    }
... lines 32 - 36
}
```

Encoding the Password

Now, on to encoding that password.

Symfony comes with a built-in service that's really good at encoding passwords. It's called security.password_encoder and if you looked it up on debug:container, its class is UserPasswordEncoder. We'll need that, so add a __construct() function and type-hint a single argument with UserPasswordEncoder $passwordEncoder. I'll hit Option+Enter and select "Initialize Fields" to save me some time:

38 lines | src/AppBundle/Doctrine/HashPasswordListener.php

```
... lines 1 - 7
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoder;

class HashPasswordListener implements EventSubscriber
{
    private $passwordEncoder;

    public function __construct(UserPasswordEncoder $passwordEncoder)
    {
        $this->passwordEncoder = $passwordEncoder;
    }
... lines 18 - 36
}
```

In a minute, we'll register this as a service.

Down below, add $encoded = $this->passwordEncoder->encodePassword() and pass it the User - which is $entity - and the plain-text password: $entity->getPlainPassword(). Finish it with $entity->setPassword($encoded):

38 lines | src/AppBundle/Doctrine/HashPasswordListener.php

```
... lines 1 - 9
class HashPasswordListener implements EventSubscriber
{
... lines 12 - 18
    public function prePersist(LifecycleEventArgs $args)
    {
        $entity = $args->getEntity();
        if (!$entity instanceof User) {
            return;
        }

        $encoded = $this->passwordEncoder->encodePassword(
            $entity,
            $entity->getPlainPassword()
        );
        $entity->setPassword($encoded);
    }
... lines 32 - 36
}
```

That's it: we are encoded!

Encoding on Update

So now also handle update, in case a User's password is changed! The two lines that actually do the encoding can be re-used, so let's refactor those into a private method. To shortcut that, highlight them, press Command+T - or go to the "Refactor"->"Refactor this" menu - and select "Method". Call it encodePassword() with one argument that's a User object:

65 lines | src/AppBundle/Doctrine/HashPasswordListener.php

```
... lines 1 - 9
class HashPasswordListener implements EventSubscriber
{
... lines 12 - 18
    public function prePersist(LifecycleEventArgs $args)
    {
... lines 21 - 25
        $this->encodePassword($entity);
    }
... lines 28 - 48
    /**
     * @param User $entity
     */
    private function encodePassword(User $entity)
    {
        if (!$entity->getPlainPassword()) {
            return;
        }

        $encoded = $this->passwordEncoder->encodePassword(
            $entity,
            $entity->getPlainPassword()
        );
        $entity->setPassword($encoded);
    }
}
```

Tip

I didn't mention it, but you also need to prevent the user's password from being encoded if plainPassword is blank. This would mean that the User is being updated, but their password isn't being changed.

Super nice!

Now that we have that, copy prePersist, paste it, and call it preUpdate. You might *think* that these methods would be identical... but not quite. Due to a quirk in Doctrine, you have to tell it that you just updated the password field, or it won't save.

The way you do this is a little nuts, and not that important: so I'll paste it in:

65 lines | src/AppBundle/Doctrine/HashPasswordListener.php

```
... lines 1 - 6
use Doctrine\ORM\Event\LifecycleEventArgs;
... lines 8 - 9
class HashPasswordListener implements EventSubscriber
{
... lines 12 - 28
    public function preUpdate(LifecycleEventArgs $args)
    {
        $entity = $args->getEntity();
        if (!$entity instanceof User) {
            return;
        }

        $this->encodePassword($entity);

        // necessary to force the update to see the change
        $em = $args->getEntityManager();
        $meta = $em->getClassMetadata(get_class($entity));
        $em->getUnitOfWork()->recomputeSingleEntityChangeSet($meta, $entity);
    }
... lines 43 - 63
}
```

Registering the Subscriber as a Service

Ok, the event subscriber is perfect! To hook it up - you guessed it - we'll register it as a service. Open app/config/services.yml and add a new service called app.doctrine.hash_password_listener. Set the class. And you guys know by now that I love to autowire things. It doesn't always work, but it's great when it does:

27 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 21
    app.doctrine.hash_password_listener:
        class: AppBundle\Doctrine\HashPasswordListener
        autowire: true
... lines 25 - 27
```

Finally, to tell Doctrine about our event subscriber, we'll add a tag. This is something we talked about in our services course: it's a way to tell the system that your service should be used for some special purpose. Set the tag to doctrine.event_subscriber:

27 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 21
    app.doctrine.hash_password_listener:
        class: AppBundle\Doctrine\HashPasswordListener
        autowire: true
        tags:
            - { name: doctrine.event_subscriber }
```

The system is complete. Before creating or updating any entities, Doctrine will call our listener.

Let's update our fixtures to try it.

Chapter 11: Configuring the Encoder in security.yml

Let's set some plain passwords! Where? In our fixtures! Open up fixtures.yml and scroll down. In theory, *all* we need to do is set the plainPassword property. The rest should happen auto-magically.

Add plainPassword: and we'll keep with iliketurtles, because I've gotten good at typing that. And turtles are cool:

```
27 lines | src/AppBundle/DataFixtures/ORM/fixtures.yml
... lines 1 - 22
AppBundle\Entity\User:
    user_{1..10}:
... line 25
        plainPassword: iliketurtles
```

Change over to your terminal and load your fixtures:

```
$ ./bin/console doctrine:fixtures:load
```

Explosion!

> No encoder has been configured for AppBundle\Entity\User

This is basically Symfony's way of saying:

> Ryan, you didn't tell me *how* you want to encode the passwords. I can't read your mind - I'm just a PHP framework.

Remember how I kept saying we would encrypt the passwords with bcrypt? Do you remember actually configuring that anywhere? Nope! We need to do that.

Open security.yml. Add an encoders key, then AppBundle\Entity\User: bcrypt:

```
33 lines | app/config/security.yml
... lines 1 - 2
security:
    encoders:
        AppBundle\Entity\User: bcrypt
... lines 6 - 33
```

If you want, you can configure a few other options, but this is good enough.

Now Symfony knows how to encrypt our passwords. Try the fixtures again:

```
$ ./bin/console doctrine:fixtures:load
```

No errors! So, that *probably* worked. Let's use it!

Checking the Encoded Password

In LoginFormAuthenticator, we can *finally* add some real password checking! How? By using that same UserPasswordEncoder object.

Head back up to __construct and add a new UserPasswordEncoder argument. I'll use my Option+Enter shortcut to setup that property for me:

80 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 10
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoder;
... lines 12 - 15
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 18 - 20
    private $passwordEncoder;

    public function __construct(FormFactoryInterface $formFactory, EntityManager $em, RouterInterface $router, UserPasswordEncod
    {
... lines 25 - 27
        $this->passwordEncoder = $passwordEncoder;
    }
... lines 30 - 78
}
```

And because we're using autowiring for this service, we don't need to change anything in services.yml.

Now, in checkCredentials(), replace the if statement with if ($this->passwordEncoder->isPasswordValid()) and pass that the User object and the plain-text password:

80 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```
... lines 1 - 15
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 18 - 58
    public function checkCredentials($credentials, UserInterface $user)
    {
... lines 61 - 62
        if ($this->passwordEncoder->isPasswordValid($user, $password)) {
            return true;
        }
... lines 66 - 67
    }
... lines 69 - 78
}
```

That'll take care of securely checking things.

Let's try it out: head to /login. Use weaverryan+1@gmail.com and iliketurtles. We're in! Password system check.

Ok team, that's *it* for authentication. You can build your authenticator to behave however you want, and you can *even* have multiple authenticators. Oh, and if you *do* want to use any of the built-in authentication systems, like the form_login key I mentioned earlier - that's totally fine. Guard authentication takes more work, but has more flexibility. If you want another example, we created a cool JSON web token authenticator in our Symfony REST series.

Now, let's start locking down the system.

Chapter 12: Authorization: access_control and Roles

Authentication is *done*. So how about we tackle the second half of security: authorization. This is all about figuring out whether or not the user has access to do something. For example, right now we have a fancy admin section, but probably not *everyone* should have access to it.

Denying with access_control

There are 2 main ways to deny access, and the simplest is right inside of security.yml. It's called "access control". Move in 4 spaces - so that you're at the same level as the firewalls key, but not inside of it. Add access_control:, new line, go out 4 more spaces and add - { path: ^/admin, roles: ROLE_USER }:

35 lines | app/config/security.yml

```
... lines 1 - 2
3   security:
... lines 4 - 33
34      access_control:
35          - { path: ^/admin, roles: ROLE_USER }
```

That path is a regular expression. So, if anyone goes to a URL that starts with /admin, the system will kick them out *unless* they have ROLE_USER.

Let see it in action. First, make sure you're logged out. Now, go to /admin/genus. Boom! That was it! Anonymous users don't have *any* roles, so the system kicked us to the login page.

Tip

Our FormLoginAuthenticator is actually responsible for sending us to /login. You can customize and override this behavior if you need to. If you use a built-in authentication system, like form_login, then *it* may be responsible for this. This functionality is called an "entry point".

Now, login. It redirects us back to /admin/genus and we *do* have access. Our user *does* have ROLE_USER - you can see that if you click the security icon in the web debug toolbar. Remember, that's happening because - in our User class - we've hardcoded the roles: *every* user has a role that I made up: ROLE_USER:

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

```
... lines 1 - 12
13  class User implements UserInterface
14  {
... lines 15 - 46
47      public function getRoles()
48      {
49          return ['ROLE_USER'];
50      }
... lines 51 - 93
94  }
```

Many access_control

And at first, that's as complex as Symfony's authorization system gets: you give each user some roles, then check to see if they have those roles. In a minute, we'll make it so each user can have different roles.

But we're not *quite* done yet with access_control. We only have one rule, but you can have *many*: just create another line below this and secure a different section of your site. For example, maybe ^/checkout requires ROLE_ALLOWED_TO_BUY.

There is one gotcha: Symfony looks for a matching access_control from top to bottom, and stops as soon as it finds the *first*

match. We won't talk about it here, but you can use that fact to lock down *every* page with an access_control, and then white-list the few public pages with access_control entries *above* that.

You can also do a few other cool things, like force the user to visit a part of your site via https. If they come via http, they'll be redirected to https.

When you Don't Have Access :(

Change the role to something we don't have, how about ROLE_ADMIN:

35 lines | app/config/security.yml

```
... lines 1 - 2
3   security:
... lines 4 - 33
34      access_control:
35          - { path: ^/admin, roles: ROLE_ADMIN }
```

Head back and refresh!

Access denied! Ok, two important things.

First, roles can be anything: I didn't have to configure ROLE_ADMIN before using it - I just made that up. The only rule about roles is that they must start with ROLE_. There's a reason for that, and I'll mention it later.

Second, notice this is an access denied screen: 403, forbidden. *We* see this because we're in development mode. But your users will see a different error page, which you can customize. In fact, you can have a different error page for 403 errors, 404 errors and 500 errors. It's easy to setup - so just check the docs.

Access controls are *super* easy to use... but they're a bit inflexible, unless you *love* writing complex, unreadable regular expressions. Next, let's look at a more precise way to control access: in your controller.

Chapter 13: Denying Access in a Controller

I *do* use access controls to lock down big sections, but, mostly, I handle authorization inside my controllers.

Deny Access (the long way)!

Let's play around: comment out the access_control:

36 lines | app/config/security.yml

```
... lines 1 - 2
security:
... lines 4 - 33
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
```

And open up GenusAdminController. To check if the current user has a role, you'll always use one service: the authorization checker. It looks like this: if (!$this->get('security.authorization_checker')->isGranted('ROLE_ADMIN'). So, if we do *not* have ROLE_ADMIN, then throw $this->createAccessDeniedException():

85 lines | src/AppBundle/Controller/Admin/GenusAdminController.php

```
... lines 1 - 13
class GenusAdminController extends Controller
{
... lines 16 - 18
    public function indexAction()
    {
        if (!$this->get('security.authorization_checker')->isGranted('ROLE_ADMIN')) {
            throw $this->createAccessDeniedException('GET OUT!');
        }
... lines 24 - 31
    }
... lines 33 - 84
}
```

That message is just for us developers.

Head back and refresh. Access denied!

So what's the magic behind that createAccessDeniedException() method? Find out: hold Command and click to open it. Ah, it *literally* just throws a special exception called AccessDeniedException:

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

```
... lines 1 - 21
22  use Symfony\Component\Security\Core\Exception\AccessDeniedException;
... lines 23 - 38
39  abstract class Controller implements ContainerAwareInterface
40  {
... lines 41 - 257
258     /**
259      * Returns an AccessDeniedException.
260      *
261      * This will result in a 403 response code. Usage example:
262      *
263      *     throw $this->createAccessDeniedException('Unable to access this page!');
264      *
265      * @param string          $message  A message
266      * @param \Exception|null $previous The previous exception
267      *
268      * @return AccessDeniedException
269      */
270     protected function createAccessDeniedException($message = 'Access Denied.', \Exception $previous = null)
271     {
272         return new AccessDeniedException($message, $previous);
273     }
... lines 274 - 396
397 }
```

It turns out - no matter *where* you are - if you need to deny access for any reason, just throw this exception. Symfony handles everything else.

Deny Access (the short way)!

Simple, but that was too much work. So, you'll probably just do this instead: $this->denyAccessUnlessGranted('ROLE_ADMIN'):

83 lines | src/AppBundle/Controller/Admin/GenusAdminController.php

```
... lines 1 - 13
14  class GenusAdminController extends Controller
15  {
... lines 16 - 18
19      public function indexAction()
20      {
21          $this->denyAccessUnlessGranted('ROLE_ADMIN');
... lines 22 - 29
30      }
... lines 31 - 82
83  }
```

Much better: that does the same thing as before.

Denying Access with Annotations

And, I have another idea! If you love annotations, you can use *those* to deny access. Above the controller, add @Security() then type a little expression: is_granted('ROLE_ADMIN'):

83 lines | src/AppBundle/Controller/Admin/GenusAdminController.php

```
... lines 1 - 7
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
... lines 9 - 14
class GenusAdminController extends Controller
{
    /**
     * @Route("/genus", name="admin_genus_list")
     * @Security("is_granted('ROLE_ADMIN')")
     */
    public function indexAction()
    {
... lines 23 - 29
    }
... lines 31 - 82
}
```

This has the *exact* same effect - it just shows us a different message.

Locking down an Entire Controller

But no matter how easy we make it, what we *really* want to do is lock down this *entire* controller. Right now, we could still go to /admin/genus/new and have access. We *could* repeat the security check in every controller... or we could do something cooler.

Add the annotation *above* the class itself:

83 lines | src/AppBundle/Controller/Admin/GenusAdminController.php

```
... lines 1 - 7
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
... lines 9 - 11
/**
 * @Security("is_granted('ROLE_ADMIN')")
 * @Route("/admin")
 */
class GenusAdminController extends Controller
{
... lines 18 - 82
}
```

As soon as you do that, all of these endpoints are locked down.

Sweet!

Chapter 14: Dynamic Roles

Denying access is great... but we still have a User class that gives *every* user the same, hardcoded role: ROLE_USER:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 46
    public function getRoles()
    {
        return ['ROLE_USER'];
    }
... lines 51 - 93
}
```

And maybe that's enough for you. But, if you *do* need the ability to assign different permissions to different users, then we've gotta go a little further.

Let's say that in *our* system, we're going to give different users different roles. How do we do that? Simple! Just create a private $roles property that's an array. Give it the @ORM\Column annotation and set its type to json_array:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 40
    /**
     * @ORM\Column(type="json_array")
     */
    private $roles = [];
... lines 45 - 110
}
```

Tip

json_array type is deprecated since Doctrine 2.6, you should use json instead.

This is *really* cool because the $roles property will hold an *array* of roles, but when we save, Doctrine will automatically json_encode that array and store it in a *single* field. When we query, it'll json_decode that back to the array. What this means is that we can store an array inside a single column, without ever worrying about the JSON encode stuff.

Returning the Dynamic Roles

In getRoles(), we can get dynamic. First, set $roles = $this->roles:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 51
    public function getRoles()
    {
        $roles = $this->roles;
... lines 55 - 61
    }
... lines 63 - 110
}
```

Second, there's just *one* rule that we need to follow about roles: every user must have at least *one* role. Otherwise, weird stuff happens.

That's no problem - just make sure that *everyone* at least has ROLE_USER by saying: if (!in_array('ROLE_USER', $roles)), then add that to $roles. Finally, return $roles:

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 51
    public function getRoles()
    {
        $roles = $this->roles;

        // give everyone ROLE_USER!
        if (!in_array('ROLE_USER', $roles)) {
            $roles[] = 'ROLE_USER';
        }

        return $roles;
    }
... lines 63 - 110
}
```

Oh, and don't forget to add a setRoles() method!

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 63
    public function setRoles(array $roles)
    {
        $this->roles = $roles;
    }
... lines 68 - 110
}
```

Migration & Fixtures

Generate the migration for the new field:

```
$ ./bin/console doctrine:migrations:diff
```

We should double-check that migration, but let's just run it:

```
$ ./bin/console doctrine:migrations:migrate
```

Finally, give some roles to our fixture users! For now, we'll give everyone the same role: ROLE_ADMIN:

28 lines | src/AppBundle/DataFixtures/ORM/fixtures.yml

```
... lines 1 - 22
AppBundle\Entity\User:
    user_{1..10}:
... lines 25 - 26
        roles: ['ROLE_ADMIN']
```

Reload the fixtures!

```
$ ./bin/console doctrine:fixtures:load
```

Ok, let's go see if we have access! Ah, we got logged out! Don't panic: that's because our user - identified by its id - was just deleted from the database. Just log back in.

So nice - it sends us *back* to the original URL, we have *two* roles and we have access. Oh, and in a few minutes - we'll talk about another tool to really make your system flexible: role hierarchy.

So, how do I Set the Roles?

But now, you might be asking me?

> How would I actually change the roles of a user?

I'm not sure though... because I can't actually hear you. But if you *are* asking me this, here's what I would say:

$roles is just a field on your User, and so you'll edit it like *any* other field: via a form. This will probably live in some "user admin area", and you'll use the ChoiceType field to allow the admin to choose the roles for every user:

```
class EditUserFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder)
    {
        $builder
            ->add('roles', ChoiceType::class, [
                'multiple' => true,
                'expanded' => true, // render check-boxes
                'choices' => [
                    'Admin' => 'ROLE_ADMIN',
                    'Manager' => 'ROLE_MANAGER',
                    // ...
                ],
            ])
            // other fields...
        ;
    }
}
```

If you have trouble, let me know.

What about Groups?

Oh, and I think I just heard one of you ask me:

> What about groups? Can you create something where Users belong to Groups, and those groups have roles?

Totally! And FOSUserBundle has code for this - so check it out. But really, it's nothing crazy: Symfony just calls getRoles(), and you can create that array however you want: like by looping over a relation:

```
class User extends UserInterface
{
    public function getRoles()
    {
        $roles = [];

        // loop over some ManyToMany relation to a Group entity
        foreach ($this->groups as $group) {
            $roles = array_merge($roles, $group->getRoles());
        }

        return $roles;
    }
}
```

Or just giving people roles at random.

Chapter 15: Fetch me a User Object!

There's really only 2 things you can do with security:

1. Deny access
2. Find out *who* is logged in

To show that off, find newAction(). Let's update the flash message to include the email address of the current user.

Surround the string with sprintf and add a %s placeholder right in the middle. How can you find out who's logged in? Yep, it's $this->getUser(). And *that* returns - wait for it - our User object. Which allows us to use *any* methods on it, like getEmail():

86 lines | src/AppBundle/Controller/Admin/GenusAdminController.php

```
... lines 1 - 15
class GenusAdminController extends Controller
{
... lines 18 - 34
    public function newAction(Request $request)
    {
... lines 37 - 40
        if ($form->isSubmitted() && $form->isValid()) {
... lines 42 - 47
            $this->addFlash(
                'success',
                sprintf('Genus created by you: %s!', $this->getUser()->getEmail())
            );
... lines 52 - 53
        }
... lines 55 - 58
    }
... lines 60 - 85
}
```

But wait! Do we have a getEmail() method on User - because I didn't see auto-completion?! Check it out. Whoops - we don't!

My bad - head to the bottom and add it!

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

```
... lines 1 - 12
class User implements UserInterface
{
... lines 15 - 83
    public function getEmail()
    {
        return $this->email;
    }
... lines 88 - 110
}
```

Nice! Now go and create a sea monster. Fill out all the fields... and... eureka!

The Secret Behind getUser()

But you know I hate secrets: I want to know what that getUser() method *really* does. So hold Command and click to see that method.

The important piece is that the user comes from a service called security.token_storage:

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

```
... lines 1 - 38
abstract class Controller implements ContainerAwareInterface
{
... lines 41 - 317
    /**
     * Get a user from the Security Token Storage.
     *
     * @return mixed
     *
     * @throws \LogicException If SecurityBundle is not available
     *
     * @see TokenInterface::getUser()
     */
    protected function getUser()
    {
        if (!$this->container->has('security.token_storage')) {
            throw new \LogicException('The SecurityBundle is not registered in your application.');
        }

        if (null === $token = $this->container->get('security.token_storage')->getToken()) {
            return;
        }

        if (!is_object($user = $token->getUser())) {
            // e.g. anonymous authentication
            return;
        }

        return $user;
    }
... lines 344 - 396
}
```

So if you ever need the User object in a service, this is how to get it.

But, it takes a couple of steps: getToken() gives you a pretty unimportant object, except for the fact that you can call getUser() on it.

The Creepy anon.

But, there's a pitfall: if you are anonymous - so, not logged in - getUser() does *not* return null, as you might expect: it returns a string: anon.. I know - it's super weird. So, if you ever *do* fetch the user object directly via this service, check to make sure getUser() returns an object. If it doesn't, the user isn't logged in.

Getting the User in Twig

The *one* other place where you'll need to fetch the User is inside Twig. In fact, let's talk about security in general in Twig. Open up base.html.twig.

Earlier, we already showed how to check for a role in Twig: it's via the is_granted() function:

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

```
<!DOCTYPE html>
<html>
... lines 3 - 13
    <body>
... lines 15 - 19
        <header class="header">
... lines 21 - 22
            <ul class="navi">
... line 24
                {% if is_granted('ROLE_USER') %}
                    <li><a href="{{ path('security_logout') }}">Logout</a></li>
                {% else %}
                    <li><a href="{{ path('security_login') }}">Login</a></li>
                {% endif %}
            </ul>
        </header>
... lines 32 - 50
    </body>
</html>
```

It's easy: it works exactly the same as in the controller.

So, how do we get the user object? To find out - open up the homepage template. If the User is logged in, let's welcome them by email.

Open the print tag and say app.user ? app.user.email : 'Aquanauts':

9 lines | app/Resources/views/main/homepage.html.twig

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

{% block body %}
    <h1 class="page-header text-center">
        Welcome
        {{ app.user ? app.user.email : 'Aquanauts' }}!
    </h1>
{% endblock %}
```

That app variable is the *one* global variable you get in Symfony. And one of its super-powers is to give you the current User object if there is one, or null if the user isn't logged in.

Head to the home page and... celebrate.

Chapter 16: Role Hierarchy

Our site will eventually have many different sections that will need to be accessed by many different *types* of aquanaut users.

Maybe the genus admin section should only be visible to marine biologists and super admins, while only the "management aquanauts" can see some future edit user section. Oh, and and of course, we programmers should be able to see everything... because let's face it - we always give ourselves access.

What's the best way to organize this? When you protect a section, instead of checking for something like ROLE_ADMIN or ROLE_MANAGEMENT - which describes the *type* of person that will have access, you might instead use a role that describes *what* is being accessed. In this case, we could use something like ROLE_MANAGE_GENUS:

86 lines | src/AppBundle/Controller/Admin/GenusAdminController.php

```
... lines 1 - 11
/**
 * @Security("is_granted('ROLE_MANAGE_GENUS')")
 * @Route("/admin")
 */
class GenusAdminController extends Controller
... lines 17 - 86
```

Why? The advantage is that you can very quickly give this role to *any* user in the database in order to allow them access to *this* section, but no others. If you plan ahead and do this, it'll give you more flexibility in the future.

A lot of Roles: A lot of Management

It's the perfect setup! Until you put it into practice. Because now, when you launch a *new* section that requires a *new* role - ROLE_OCTOPUS_PHOTO_MANAGE - the super admins and programmers *won't* have access to it until you manually add this role to all of those users. That's lame.

Of course, you *can* solve this with that group system we talked about earlier, but that's usually overkill. And, there's a simpler way.

Role Hierarchy

In security.yml, let's take advantage of something called role hierarchies. It's simple, it's awesome!. Add a new key called role_hierarchy and, below that, set ROLE_ADMIN: [ROLE_MANAGE_GENUS]:

39 lines | app/config/security.yml

```
... lines 1 - 2
security:
... lines 4 - 6
    role_hierarchy:
        ROLE_ADMIN: [ROLE_MANAGE_GENUS]
... lines 9 - 39
```

In other words, if anybody has ROLE_ADMIN, automatically give them ROLE_MANAGE_GENUS. Later, when you launch the new Octopus photo admin area, just add ROLE_OCTOPUS_PHOTO_MANAGE here and be done with it.

To see it in action, comment it out temporarily. Now, head to /admin/genus. Access denied! No surprise. Uncomment the role hierarchy and try it again. Access granted!

The strategy is this: first: lock down different sections using role names that describe *what* it's protecting - like ROLE_OCTOPUS_PHOTO_MANAGE. Second, in security.yml, create group-based roles here - like ROLE_MARINE_BIOLOGIST or ROLE_MANAGEMENT - and assign each the permissions they should have. With this setup, you should be able to give most users just *one* role in the database.

Of course, don't bother doing anything of this if your app is simple and will have just one or two different types of users.

Ok, now that we know how to give each user the *exact* access they need, let's find out how to *impersonate* them.

Chapter 17: Impersonation (Login as Someone Else)

Have you ever had a bug you couldn't reproduce? Nope, me either. Of course we have! The worst is when a user reports a bug in their account... which would be *really* easy to verify and debug... if *only* we could log and see what they're seeing.

Look, we've got too much work to do to debug this the hard way. We need to be able switch to that user's account: we need to impersonate them.

Activating switch_user

Setting up impersonation is super easy. In security.yml, under your firewall, add a new key called switch_user set to ~ to activate the system:

40 lines | app/config/security.yml

```
... lines 1 - 2
3   security:
... lines 4 - 14
15      firewalls:
... lines 16 - 20
21          main:
... lines 22 - 28
29              switch_user: ~
... lines 30 - 40
```

Back to the User Provider

Now, on any URL, you can add ?_switch_user= and then the username of whoever you want to log in as. In our case, we want to login as weaverryan+5@gmail.com. Why is that the email address in our case? This actually goes back to your user provider.

We're using the built-in entity user provider, *and* we told it that we want to identify by the email. Before now, this setting was meaningless. If we changed that to some other field, then we would actually identify ourselves by that up in the URL.

Not Everyone Can Impersonate!

Hit enter to try switching users. OMG - access denied!

That makes sense - we can't just let *anybody* do this impersonation trick. Internally, this feature checks for a very specific role called ROLE_ALLOWED_TO_SWITCH. But, we don't have that.

Hey, no problem! Let's give this role to ROLE_ADMIN under role_hierarchy:

40 lines | app/config/security.yml

```
... lines 1 - 2
3   security:
... lines 4 - 6
7       role_hierarchy:
8           ROLE_ADMIN: [ROLE_MANAGE_GENUS, ROLE_ALLOWED_TO_SWITCH]
... lines 9 - 40
```

Cool! Try it out. It works! I mean, it doesn't work! Hmm, but it's *not* that security error anymore - it just can't find the user: weaverryan 5@gmail.com You know what? This is the because of the + sign in the email addresses - which represents a space in URLs. Change that to the url-encoded plus sign: %2B.

Boom! Now we're surfing as weaverryan+5@gmail.com. Pretty awesome. Once you're done, go back by adding ?_switch_user=_exit to any URL. That's it! We're back as our original user.

Switching users... yea, it's my favorite.

Chapter 18: Registration Form

That's it for security! We covered authentication and authorization. So, I'm not really sure why I'm still recording.

Oh yea, I remember: let's create a registration form! Actually, this has nothing to do with security: registration is all about creating and saving a User entity. But, there are a few interesting things - call it a bonus round.

Controller, Form, Check!

Start like normal: create a new controller class called UserController - for stuff like registration and maybe future things like reset password:

19 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 2
3  namespace AppBundle\Controller;
... lines 4 - 5
6  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
... lines 7 - 8
9  class UserController extends Controller
10 {
... lines 11 - 17
18 }
```

Inside, add registerAction() with the URL /register. Let's call the route user_register:

19 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 4
5  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
... line 6
7  use Symfony\Component\HttpFoundation\Request;
8
9  class UserController extends Controller
10 {
11     /**
12      * @Route("/register", name="user_register")
13      */
14     public function registerAction(Request $request)
15     {
16
17     }
18 }
```

Make sure you have your use statement for @Route.

Next, this will be a nice, normal form situation. So click the Form directory, open the new menu, and create a new Symfony Form. Call it UserRegistrationForm:

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

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

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

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

    }

    public function configureOptions(OptionsResolver $resolver)
    {

    }
}
```

Brilliant! Delete the extra getName() method that's *super* not needed in Symfony 3.

Now, bind the form to User, with $resolver->setDefaults() and a data_class option to set User::class:

31 lines | src/AppBundle/Form/UserRegistrationForm.php

```
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 12
class UserRegistrationForm extends AbstractType
{
... lines 15 - 23
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => User::class
        ]);
    }
}
```

Next, the fields! And we need two: first an email field set to EmailType::class:

31 lines | src/AppBundle/Form/UserRegistrationForm.php

```
... lines 1 - 6
use Symfony\Component\Form\Extension\Core\Type\EmailType;
... lines 8 - 12
class UserRegistrationForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('email', EmailType::class)
... lines 19 - 21
    }
... lines 23 - 29
}
```

Then, we *do* need a password field, but think about it: the property we want to set on User is *not* actually the password property. We need to set plainPassword. Add this. It'll be a password type. But, if you want the user to type the password twice, use a RepeatedType. Then, in the third argument, pass the real type with type set to PasswordType::class:

31 lines | src/AppBundle/Form/UserRegistrationForm.php

```
... lines 1 - 7
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
... lines 10 - 12
class UserRegistrationForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('email', EmailType::class)
            ->add('plainPassword', RepeatedType::class, [
                'type' => PasswordType::class
            ]);
    }
... lines 23 - 29
}
```

That'll render *two* password boxes. And if the values don't match, validation will automatically fail.

Rendering the Form

Form done! In the controller, start with $form = $this->createForm(). And of course, make sure you're extending the Symfony base Controller! Then, pass this UserRegistrationForm::class:

24 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 4
use AppBundle\Form\UserRegistrationForm;
... line 6
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
... lines 8 - 9
class UserController extends Controller
{
... lines 12 - 14
    public function registerAction(Request $request)
    {
        $form = $this->createForm(UserRegistrationForm::class);
... lines 18 - 21
    }
}
```

Go straight to the template: return $this->render('user/register.html.twig') and pass it $form->createView():

24 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 16
        $form = $this->createForm(UserRegistrationForm::class);

        return $this->render('user/register.html.twig', [
            'form' => $form->createView()
        ]);
... lines 22 - 24
```

Ok, all standard!

As a short cut, I'll hover over the template, press Option+Enter and select "Create Template".

You guys know the drill: extends 'base.html.twig' then override the block body. I'll give us just a little bit of markup to get things rolling:

19 lines | app/Resources/views/user/register.html.twig

```
1  {% extends 'base.html.twig' %}
2
3  {% block body %}
4      <div class="container">
5          <div class="row">
6              <div class="col-xs-12">
7                  <h1>Register!</h1>
   ... lines 8 - 15
16             </div>
17         </div>
18     </div>
19 {% endblock %}
```

Rendering the form is exactly how it always is: form_start(form), form_end(form), and inside, form_row(form.email):

19 lines | app/Resources/views/user/register.html.twig

```
   ... lines 1 - 2
3  {% block body %}
4      <div class="container">
5          <div class="row">
6              <div class="col-xs-12">
7                  <h1>Register!</h1>
8
9                  {{ form_start(form) }}
10                     {{ form_row(form.email) }}
   ... lines 11 - 14
15                 {{ form_end(form) }}
16             </div>
17         </div>
18     </div>
19 {% endblock %}
```

Then form_row(form.plainPassword) - but because we used the RepeatedType, this will render as *two* fields - so use form.plainPassword.first and form_row(form.plainPassword.second):

19 lines | app/Resources/views/user/register.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
            <div class="col-xs-12">
                <h1>Register!</h1>

                {{ form_start(form) }}
                    {{ form_row(form.email) }}
                    {{ form_row(form.plainPassword.first) }}
                    {{ form_row(form.plainPassword.second) }}
... lines 13 - 14
                {{ form_end(form) }}
            </div>
        </div>
    </div>
{% endblock %}
```

Cool, right?

Finally show off your styling skills by adding a <button type="submit"> with some fancy Bootstrap classes. Don't forget the formnovalidate to disable HTML5 validation. And finally say, register:

19 lines | app/Resources/views/user/register.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
            <div class="col-xs-12">
                <h1>Register!</h1>

                {{ form_start(form) }}
                    {{ form_row(form.email) }}
                    {{ form_row(form.plainPassword.first) }}
                    {{ form_row(form.plainPassword.second) }}

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

That oughta do it! Finish things by adding a link to this from the login page. After the button, add a link to path('user_register'):

26 lines | app/Resources/views/security/login.html.twig

```
... lines 1 - 2
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-xs-12">
... lines 7 - 14
            {{ form_start(form) }}
... lines 16 - 17
                <button type="submit" class="btn btn-success">Login <span class="fa fa-lock"></span></button>
                 
                <a href="{{ path('user_register') }}">Register</a>
            {{ form_end(form) }}
        </div>
    </div>
</div>
{% endblock %}
```

Done! Refresh. Click "Register", and we're rendered.

Fixing the Password fields

Ooh - except for the labels: "First" and "Second": those are terrible! We can fix those real quick: pass a variables array to first with label set to Password. For the second one: Repeat Password:

23 lines | app/Resources/views/user/register.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
            <div class="col-xs-12">
                <h1>Register!</h1>

                {{ form_start(form) }}
... line 10
                    {{ form_row(form.plainPassword.first, {
                        'label': 'Password'
                    }) }}
                    {{ form_row(form.plainPassword.second, {
                        'label': 'Repeat Password'
                    }) }}
... lines 17 - 18
                {{ form_end(form) }}
            </div>
        </div>
    </div>
{% endblock %}
```

Refresh. Looking good.

Saving the User

Since the registration form has nothing to do with security, let's just finish this! Type-hint the Request argument, and then do the normal $form->handleRequest($request):

38 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 8
use Symfony\Component\HttpFoundation\Request;

class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
        $form = $this->createForm(UserRegistrationForm::class);

        $form->handleRequest($request);
... lines 21 - 35
    }
}
```

Then, if($form->isValid()) - to make sure that validation is passed:

38 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 10
class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
        $form = $this->createForm(UserRegistrationForm::class);

        $form->handleRequest($request);
        if ($form->isValid()) {
... lines 22 - 30
        }
... lines 32 - 35
    }
}
```

In the forms tutorial, we also added $form->isSubmitted() in the if statement, but you technically don't need that: isValid() checks that internally.

Inside the isValid(), set $user = $form->getData():

38 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 10
class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
... lines 18 - 20
        if ($form->isValid()) {
            /** @var User $user */
            $user = $form->getData();
... lines 24 - 30
        }
... lines 32 - 35
    }
}
```

We know this will be a User object, so I'll plan ahead and add some inline PHP documentation so I get auto-completion later. Add the $em = $this->getDoctrine()->getManager(), $em->persist($user), $em->flush():

38 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 10
class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
... lines 18 - 20
        if ($form->isValid()) {
            /** @var User $user */
            $user = $form->getData();
            $em = $this->getDoctrine()->getManager();
            $em->persist($user);
            $em->flush();
... lines 27 - 30
        }
... lines 32 - 35
    }
}
```

Now, what do we *always* do after a successful form submit? We set a flash: $this->addFlash('success') with 'Welcome '.$user->getEmail():

38 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 10
class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
... lines 18 - 20
        if ($form->isValid()) {
            /** @var User $user */
            $user = $form->getData();
            $em = $this->getDoctrine()->getManager();
            $em->persist($user);
            $em->flush();

            $this->addFlash('success', 'Welcome '.$user->getEmail());
... lines 29 - 30
        }
... lines 32 - 35
    }
}
```

Finally, redirect - at least for right now - to the homepage route:

38 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 10
class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
... lines 18 - 20
        if ($form->isValid()) {
            /** @var User $user */
            $user = $form->getData();
            $em = $this->getDoctrine()->getManager();
            $em->persist($user);
            $em->flush();

            $this->addFlash('success', 'Welcome '.$user->getEmail());

            return $this->redirectToRoute('homepage');
        }
... lines 32 - 35
    }
}
```

That's it.

Try the whole thing out: weaverryan+15@gmail.com, Password foo. Whoops, and if we just fix my typo, and refresh again:

38 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 10
class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
... lines 18 - 20
        if ($form->isValid()) {
... lines 22 - 24
            $em->persist($user);
... lines 26 - 30
        }
... lines 32 - 35
    }
}
```

It's alive!

But notice it did *not* automatically log me in. That's something we'll fix in a second. But hey, registration. It's a form. It's easy! It's done.

Chapter 19: Validation with the UniqueEntity Constraint

Registration is working, but it's missing validation.

Since the form is bound to the User class, that is where our annotation rules should live. First, you need the use statement for the annotations. We added validation earlier in Genus. So, you can either copy this use statement, grab it from the documentation, or do what I do: cheat by saying use, auto-completing an annotation I know exists - like NotBlank, deleting the last part, and adding the normal as Assert alias:

```
116 lines | src/AppBundle/Entity/User.php
... lines 1 - 7
8  use Symfony\Component\Validator\Constraints as Assert;
... lines 9 - 116
```

We obviously want email to be NotBlank. We also want email to be a valid email address. For plainPassword, that should also not be blank:

```
116 lines | src/AppBundle/Entity/User.php
... lines 1 - 13
14  class User implements UserInterface
15  {
... lines 16 - 22
23      /**
24       * @Assert\NotBlank()
25       * @Assert\Email()
... line 26
27       */
28      private $email;
... lines 29 - 36
37      /**
... line 38
39       * @Assert\NotBlank()
... lines 40 - 41
42       */
43      private $plainPassword;
... lines 44 - 114
115 }
```

Pretty simple.

Ok, go back, keep the form blank, and submit. Nice validation errors.

Forcing a Unique Email

But check this out: type weaverryan+1@gmail.com. That email is already taken, so I should *not* be able to do this. But since there aren't any validation rules checking this, the request goes through and the email looks totally valid.

How can we add a validation rule to prevent that? By using a special validation constraint made *just* for this occasion.

The UniqueEntity Constraint

This constraint's annotation doesn't go above a specific property: it lives above the entire class. Add @UniqueEntity. Notice, this lives in a different namespace than the other annotations, so PhpStorm added its own use statement.

Next configure it. You can always go to the reference section of the docs, or, if you hold command, you can click the annotation to open its class. The public properties are always the options that you can pass to the annotation.

The options we need are fields - which tell it which field needs to be unique in the database - and message so we can say something awesome when it happens.

So add fields={"email"}. This is called *fields* because you *could* make this validation be unique across several columns. Then add message="It looks like you already have an account!":

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

```
... lines 1 - 4
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
... lines 6 - 10
/**
... lines 12 - 13
 * @UniqueEntity(fields={"email"}, message="It looks like your already have an account!")
 */
class User implements UserInterface
... lines 17 - 118
```

Cool! Go back and hit register again. This just makes me happy!

We're good, right? Well, almost. There's one last gotcha with validation and registration.

Chapter 20: Validation Groups: Conditional Validation

Ready for the problem? Right now, we need the plainPassword to be required. But later when we create an edit profile page, we *don't* want to make plainPassword required. Remember, this is *not* saved to the database. So if the user leaves it blank on the edit form, it just means they don't want to change their password. And that should be allowed.

So, we need this annotation to only work on the registration form.

Validation Groups to the Rescue!

Here's how you do it: take advantage of something called validation groups. On the NotBlank constraint, add groups={"Registration"}:

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

```
... lines 1 - 15
16  class User implements UserInterface
17  {
... lines 18 - 38
39      /**
... line 40
41       * @Assert\NotBlank(groups={"Registration"})
... lines 42 - 43
44       */
45      private $plainPassword;
... lines 46 - 116
117 }
```

This "Registration" is a string I just invented: there's no significance to it.

Without doing anything else, go back, hit register, and check it out! The error went away. Here's what's happening: by default, all constraints live in a group called Default. And when your form is validated, it validates all constraints in this Default group. So now that we've put this into a different group called Registration, when the form validates, it doesn't validate using this constraint.

To use this annotation *only* on the registration form, we need to make that form validate everything in the Default group *and* the Registration group. Open up UserRegistrationForm and add a second option to setDefaults(): validation_groups set to Default - with a capital D and then Registration:

32 lines | src/AppBundle/Form/UserRegistrationForm.php

```
... lines 1 - 12
13  class UserRegistrationForm extends AbstractType
14  {
... lines 15 - 23
24      public function configureOptions(OptionsResolver $resolver)
25      {
26          $resolver->setDefaults([
... line 27
28              'validation_groups' => ['Default', 'Registration']
29          ]);
30      }
31  }
```

That should do it. Refresh: validation is back.

Ok team: one final mission: automatically authenticate the user after registration. Because really, that's what our users want.

Chapter 21: Automatically Login after Registration!

If I submitted this form right now, it would register me, but it would *not* actually log me in... which is lame. Let's fix that.

This is *always* pretty easy, but it's *especially* easy because we're using Guard authentication. Inside of UserController, instead of redirecting to the home page: do this: return $this->get() to find a service called security.authentication.guard_handler. It has a method on it called authenticateUserAndHandleSuccess(). I'll clear the arguments and use multiple lines:

```
44 lines | src/AppBundle/Controller/UserController.php
... lines 1 - 10
class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
... lines 18 - 20
        if ($form->isValid()) {
... lines 22 - 29
            return $this->get('security.authentication.guard_handler')
                ->authenticateUserAndHandleSuccess(
... lines 32 - 35
                );
        }
... lines 38 - 41
    }
}
```

The first argument is the $user and the second is $request:

```
44 lines | src/AppBundle/Controller/UserController.php
... lines 1 - 10
class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
... lines 18 - 20
        if ($form->isValid()) {
... lines 22 - 29
            return $this->get('security.authentication.guard_handler')
                ->authenticateUserAndHandleSuccess(
                    $user,
                    $request,
... lines 34 - 35
                );
        }
... lines 38 - 41
    }
}
```

The third argument is the authenticator whose success behavior we want to mimic. Open up service.yml and copy the service name for our authenticator:

27 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 17
    app.security.login_form_authenticator:
        class: AppBundle\Security\LoginFormAuthenticator
        autowire: true
... lines 21 - 27
```

In the controller, use $this->get() and paste the service ID:

44 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 10
class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
... lines 18 - 20
        if ($form->isValid()) {
... lines 22 - 29
            return $this->get('security.authentication.guard_handler')
                ->authenticateUserAndHandleSuccess(
                    $user,
                    $request,
                    $this->get('app.security.login_form_authenticator'),
... line 35
                );
        }
... lines 38 - 41
    }
}
```

Finally, the last argument is something called a "provider key". You'll see that occasionally - it's a fancy term for the name of your firewall:

40 lines | app/config/security.yml

```
... lines 1 - 2
security:
... lines 4 - 14
    firewalls:
... lines 16 - 20
        main:
            anonymous: ~
            guard:
                authenticators:
                    - app.security.login_form_authenticator
... lines 26 - 40
```

This name is almost *never* important, but it actually is in this case. We'll say main:

44 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 10
class UserController extends Controller
{
... lines 13 - 15
    public function registerAction(Request $request)
    {
... lines 18 - 20
        if ($form->isValid()) {
... lines 22 - 29
            return $this->get('security.authentication.guard_handler')
                ->authenticateUserAndHandleSuccess(
                    $user,
                    $request,
                    $this->get('app.security.login_form_authenticator'),
                    'main'
                );
        }
... lines 38 - 41
    }
}
```

And we're done!

The Bonus Superpower

Now, this will log us in, but it *also* has a bonus super-power. Right now, we're anonymous. So let's try to go to /admin/genus. Of course, it bounces us to the login page. That's fine - click to register. Use weaverryan+20@gmail.com, add a password and hit enter.

Check this out. Well, ignore the access denied screen.

First, it *did* log us in. And *second*, it redirected us back to the URL we were trying to visit before we were sent to the login page. This is *really* great because it means that if your user tries to access a secure page - like your checkout form - they'll end up *back* on the checkout form after registration. Then they can keep buying your cool stuff.

Of course, we see the access denied page because this user only has ROLE_USER and the section requires ROLE_MANAGE_GENUS.

Signing Off

Ok guys, that's it. Yes, you *can* get more complicated with security, especially authentication. And if you need to check permissions that are *object* specific - like I can edit only genuses that I created - then check out Symfony's awesome voter system. Hey, we have a course on it!

But for the most part, you guys have the tools to do really incredible things. So go out there, build something awesome and tell me about it.

Seeya next time!