Symfony Security: Beautiful Authentication, Powerful Authorization

With <3 from SymfonyCasts

Chapter 1: Security & the User Class

Yeaaaa! You've done it! You've made it to the tutorial where *we* get to build a security system with Symfony. This stuff is *cool*. Seriously, these days, the topic of security is gigantic! Just think about authentication: you might need to build a traditional login form, or a token-based API authentication system, or two-factor authentication or authentication across an API to a Single Sign-On server or something I've never even dreamed of before! For authorization, there are roles, access controls and more.

Woh. So we're going to write some *seriously* fun code in this tutorial. And it will be *especially* fun, because there are some new cool toys in Symfony's security system that make it nicer than ever to work with.

Coding Along!

As always, to become a *true* Symfony security geek... and to obtain the blueprint to the Deathstar, you should *definitely* code along with me. Download the course code from this page. When you unzip it, you'll find a start/ directory that has the same code that you see here. Follow the README.md file for all the important setup details.

Oh, and if you've been coding along with me in the Symfony series so far, um, you're amazing! But also, be sure to download the *new* code: I made a few changes since the last tutorial, including upgrading to Symfony 4.1 and improving our fixture system. More on that later.

Anyways, the *last* setup step will be to open a terminal, move into the project and run:

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

to start the built in web server. Ok: head back to your browser and open our app by going to http://localhost:8000.

Hello The SpaceBar! Our awesome intergalactic *real* news site that helps connect alien species across this side of the Milky Way.

Installing Security & Upgrading MakerBundle

Our *first* goal in this tutorial is to create an *authentication* system. In other words: a way for the user to login. No matter *how* you want your users to authenticate - whether it's a login form, API authentication or something crazier - the first step is always the same: brew some coffee or tea. The *second* step is *also* always the same: create a User class.

To do this, we're going to use a brand-spanking new feature! Woo! Find your terminal and run:

```
$ composer update symfony/maker-bundle
```

Version 1.7 of MakerBundle comes with a new command that will make our life *much* easier. Yep, there it is: 1.7. The new command is called make:user - try it:

```
$ php bin/console make:user
```

Ah! It explodes! Of course! Remember: in Symfony 4, our project starts *small*. If you need a feature, you need to install it. Run:

```
$ composer require security
```

Ah, check it out: this library has a recipe! When Composer finishes... find out what it did by running:

```
$ git status
```

A new config file! Check it out: config/packages/security.yaml. This file is *super* important. We'll start talking about it soon.

Creating the User Class with make:user

Before we run make:user again, add all the changed files to git and commit with a message about upgrading MakerBundle & adding security:

```
$ git add .
$ git commit -m "Upgraded MakerBundle and added security"
```

I'm doing this because I want to see *exactly* what the make:user command does.

Ok already, let's try it!

```
$ php bin/console make:user
```

Call the class User. Second question:

> Do you want to store user data in the database

For most apps, this is an easy yes... because most apps store user data in a local database table. But, what if your user data is stored on some *other* server, like an LDAP server or a single sign-on server? Well, *even* in those cases, if you want to store *any* extra information about your users in a local database table, you should still answer yes. Answer "no" *only* if you don't need to store *any* user information to your database.

So, "yes" for us! Next: choose one property on your user that will be its unique display name. This can be anything - it's usually an email or username. We'll talk about how it's used later. Choose email.

And, the last question: is our app responsible for checking the user's password? In some apps - like a pure API with only token authentication, users might not even *have* a password. And even if your users *will* be able to login with a password, only answer yes if *this* app will be responsible for directly checking the user's password. If you actually send the password to a third-party server and *it* checks if it's valid, choose no.

Remember when I mentioned how complex & different modern authentication systems can be? That's why this command exists: to help walk us through *exactly* want we need.

I'm going to choose "No" for now. We *will* add a password later, but we'll keep things extra simple to start.

And... we're done! Awesome! This created a User entity, a Doctrine UserRepository for it, and updated the security.yaml file.

Let's check out these changes next!

Chapter 2: All about the User class

Now matter *how* your users will login, the *first* step to creating an authentication system is to create a User class. And we just did that with the handy make:user command.

Go check out that class: src/Entity/User.php:

101 lines | src/Entity/User.php

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

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 */
class User implements UserInterface
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=180, unique=true)
     */
    private $email;

    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];

    public function getId(): ?int
    {
        return $this->id;
    }

    public function getEmail(): ?string
    {
        return $this->email;
    }

    public function setEmail(string $email): self
    {
        $this->email = $email;

        return $this;
    }
```

```
    /**
     * A visual identifier that represents this user.
     *
     * @see UserInterface
     */
    public function getUsername(): string
    {
        return (string) $this->email;
    }

    /**
     * @see UserInterface
     */
    public function getRoles(): array
    {
        $roles = $this->roles;
        // guarantee every user at least has ROLE_USER
        $roles[] = 'ROLE_USER';

        return array_unique($roles);
    }

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

        return $this;
    }

    /**
     * @see UserInterface
     */
    public function getPassword()
    {
        // not needed for apps that do not check user passwords
    }

    /**
     * @see UserInterface
     */
    public function getSalt()
    {
        // not needed for apps that do not check user passwords
    }

    /**
     * @see UserInterface
     */
    public function eraseCredentials()
    {
        // If you store any temporary, sensitive data on the user, clear it here
        // $this->plainPassword = null;
    }
}
```

Two important things. First, because we chose "yes" to storing user info in the database, the command created an *entity* class with the normal annotations and id property. It *also* added an email property, a roles property - that we'll talk about later - and the normal getter and setter methods. Yep, this User class is just a normal, boring entity class.

Now look back at the top of the class. Ah, it implements a UserInterface:

```
101 lines | src/Entity/User.php
... lines 1 - 5
6  use Symfony\Component\Security\Core\User\UserInterface;
... lines 7 - 10
11 class User implements UserInterface
12 {
... lines 13 - 99
100 }
```

This is the *second* important thing make:user did. Our User class can look *however* we want. The *only* rule is that it must implement this interface... which is actually pretty simple. It just means that you need a few extra methods. The first is getUsername()... which is a *bad* name... because your users do *not* need to have a username. This method should just return a visual identifier for the user. In our case: email:

```
101 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 46
47     /**
48      * A visual identifier that represents this user.
49      *
50      * @see UserInterface
51      */
52     public function getUsername(): string
53     {
54         return (string) $this->email;
55     }
... lines 56 - 99
100 }
```

And actually, this method is only used by Symfony to display who is currently logged in on the web debug toolbar. It's not important.

Next is getRoles():

101 lines | src/Entity/User.php

```
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 56
57     /**
58      * @see UserInterface
59      */
60     public function getRoles(): array
61     {
62         $roles = $this->roles;
63         // guarantee every user at least has ROLE_USER
64         $roles[] = 'ROLE_USER';
65
66         return array_unique($roles);
67     }
... lines 68 - 99
100 }
```

This is related to user permissions, and we'll talk about it later.

The last 3 are getPassword(), getSalt() and eraseCredentials(). And *all* 3 of these are *only* needed if your app is responsible for storing and checking user passwords. Because our app will *not* check user passwords - well, not *yet* - these can safely be blank:

101 lines | src/Entity/User.php

```
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 75
76     /**
77      * @see UserInterface
78      */
79     public function getPassword()
80     {
81         // not needed for apps that do not check user passwords
82     }
83
84     /**
85      * @see UserInterface
86      */
87     public function getSalt()
88     {
89         // not needed for apps that do not check user passwords
90     }
91
92     /**
93      * @see UserInterface
94      */
95     public function eraseCredentials()
96     {
97         // If you store any temporary, sensitive data on the user, clear it here
98         // $this->plainPassword = null;
99     }
100 }
```

So, for us: we basically have a normal entity class that also has a getUsername() method and a getRoles() method. It's really, pretty boring.

The *other* file that was modified was config/packages/security.yaml. Go back to your terminal and run:

```
$ git diff
```

to see what changed. Ah, it updated this providers key:

29 lines | config/packages/security.yaml

```yaml
security:
    # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
... lines 9 - 29
```

This is called a "user provider". Each User class - and you'll almost definitely only need one User class - needs a corresponding "user provider". And actually, it's not *that* important. I'll tell you what it does later.

But before we get there, forget about security and remember that our User class is a Doctrine entity. Let's add another field to it, generate a migration & add some dummy users to the database. Then, to authentication!

Chapter 3: Customizing the User Entity

The *really* neat thing about Symfony's security system is that it doesn't care at *all* about what your User class looks like. As long as it implements UserInterface, so, as long as it has these methods, you can do *anything* you want with it. Heck, it doesn't even need to be an entity!

Adding more Fields to User

For example, we already have an email field, but I also want to be able to store the first name for each user. Cool: we can just add that field! Find your terminal and run:

```
$ php bin/console make:entity
```

Update the User class and add firstName as a string, length 255 - or shorter if you want - and not nullable. Done!

Check out the User class! Yep, there's the new firstName property and... at the bottom, the getter and setter methods:

118 lines | src/Entity/User.php

```
... lines 1 - 10
class User implements UserInterface
{
... lines 13 - 29
    /**
     * @ORM\Column(type="string", length=255)
     */
    private $firstName;
... lines 34 - 105
    public function getFirstName(): ?string
    {
        return $this->firstName;
    }

    public function setFirstName(string $firstName): self
    {
        $this->firstName = $firstName;

        return $this;
    }
}
```

Awesome!

Setting Doctrine's server_version

I think we're ready to make the migration. But! A word of warning. Check out the roles field on top:

```
118 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 24
25     /**
26      * @ORM\Column(type="json")
27      */
28     private $roles = [];
... lines 29 - 116
117 }
```

It's an array and its Doctrine type is json. This is *really* cool. Newer databases - like PostgreSQL and MySQL 5.7 - have a native "JSON" column type that allows you to store an *array* of data.

But, if you're using MySQL 5.6 or lower, this column type does *not* exist. And actually, that's not a problem! In that case, Doctrine is smart enough to use a normal text field, json_encode() your array when saving, and json_decode() it automatically when we query. So, no matter *what* database you use, you *can* use this json Doctrine column type.

But, here's the catch. Open config/packages/doctrine.yaml. One of the keys here is server_version, which is set to 5.7 by default:

```
31 lines | config/packages/doctrine.yaml
... lines 1 - 7
8 doctrine:
9     dbal:
... lines 10 - 11
12         server_version: '5.7'
... lines 13 - 31
```

This tells Doctrine that when it interacts with the database, it should *expect* that our database has all the features supported by MySQL 5.7, *including* that native JSON column type. If your computer, or more importantly, if your *production* database is using MySQL 5.6, then you'll get a *huge* error when Doctrine tries to make queries using the native MySQL JSON column type.

If you're in this situation, just set this back to 5.6:

```
31 lines | config/packages/doctrine.yaml
... lines 1 - 7
8 doctrine:
9     dbal:
... lines 10 - 11
12         server_version: '5.6'
... lines 13 - 31
```

Doctrine will then create a normal text column for the JSON field.

Generating the Migration

Ok, *now* run:

```
$ php bin/console make:migration
```

Perfect! Go check that file out in src/Migrations:

```
29 lines | src/Migrations/Version20180830012659.php
... lines 1 - 10
11 final class Version20180830012659 extends AbstractMigration
12 {
13     public function up(Schema $schema) : void
14     {
... lines 15 - 17
18         $this->addSql('CREATE TABLE user (id INT AUTO_INCREMENT NOT NULL, email VARCHAR(180) NOT NULL, roles LONGTEX
19     }
... lines 20 - 27
28 }
```

And... nice! CREATE TABLE user. Look at the roles field: a LONGTEXT column. If you kept your server_version at 5.7, this would be a json column.

Let's run this:

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

Adding Fixtures

One last step: we need to add some dummy users into the database. Start with:

```
$ php bin/console make:fixtures
```

Call it UserFixture. Go check that out: src/DataFixtures/UserFixture.php:

```
18 lines | src/DataFixtures/UserFixture.php
... lines 1 - 2
3 namespace App\DataFixtures;
4
5 use Doctrine\Bundle\FixturesBundle\Fixture;
6 use Doctrine\Common\Persistence\ObjectManager;
7
8 class UserFixture extends Fixture
9 {
10     public function load(ObjectManager $manager)
11     {
12         // $product = new Product();
13         // $manager->persist($product);
14
15         $manager->flush();
16     }
17 }
```

If you watched our Doctrine tutorial, you might remember that we created a special BaseFixture with some sweet shortcut methods. Before I started recording *this* tutorial, based on some feedback from *you* nice people, I made a few improvements to that class. Go team!

92 lines | src/DataFixtures/BaseFixture.php

```
... lines 1 - 9
10 abstract class BaseFixture extends Fixture
11 {
... lines 12 - 21
22     public function load(ObjectManager $manager)
23     {
24         $this->manager = $manager;
25         $this->faker = Factory::create();
26
27         $this->loadData($manager);
28     }
... lines 29 - 90
91 }
```

The way you use this class is still the same: extend BaseFixture and update the load() method to be protected function loadData(). I'll remove the old use statement.

17 lines | src/DataFixtures/UserFixture.php

```
... lines 1 - 2
3 namespace App\DataFixtures;
4
5 use Doctrine\Common\Persistence\ObjectManager;
6
7 class UserFixture extends BaseFixture
8 {
9     protected function loadData(ObjectManager $manager)
10     {
... lines 11 - 14
15     }
16 }
```

Inside, call $this->createMany(). The arguments to this method changed a bit since the last tutorial:

92 lines | src/DataFixtures/BaseFixture.php

```
... lines 1 - 9
10 abstract class BaseFixture extends Fixture
11 {
... lines 12 - 29
30     /**
31      * Create many objects at once:
32      *
33      *     $this->createMany(10, function(int $i) {
34      *         $user = new User();
35      *         $user->setFirstName('Ryan');
36      *
37      *         return $user;
38      *     });
39      *
40      * @param int      $count
41      * @param string   $groupName Tag these created objects with this group name,
42      *                            and use this later with getRandomReference(s)
43      *                            to fetch only from this specific group.
44      * @param callable $factory
45      */
46     protected function createMany(int $count, string $groupName, callable $factory)
47     {
48         for ($i = 0; $i < $count; $i++) {
49             $entity = $factory($i);
50
51             if (null === $entity) {
52                 throw new \LogicException('Did you forget to return the entity object from your callback to BaseFixture::createMany()?');
53             }
54
55             $this->manager->persist($entity);
56
57             // store for usage later as groupName_#COUNT#
58             $this->addReference(sprintf('%s_%d', $groupName, $i), $entity);
59         }
60     }
... lines 61 - 90
91 }
```

Pass this 10 to create 10 users. Then, pass a "group name" - main_users. Right now, this key is meaningless. But later, we'll use it in a different fixture class to relate other objects to these users. Finally, pass a callback with an $i argument:

23 lines | src/DataFixtures/UserFixture.php

```
... lines 1 - 7
8 class UserFixture extends BaseFixture
9 {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(10, 'main_users', function($i) {
... lines 13 - 17
18         });
19
20         $manager->flush();
21     }
22 }
```

This will be called 10 times and our job inside is simple: create a User, put some data on it and return!

Do it! $user = new User():

23 lines | src/DataFixtures/UserFixture.php

```
... lines 1 - 4
5  use App\Entity\User;
... lines 6 - 7
8  class UserFixture extends BaseFixture
9  {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(10, 'main_users', function($i) {
13             $user = new User();
... lines 14 - 17
18         });
... lines 19 - 20
21     }
22 }
```

Then $user->setEmail() with sprintf() spacebar%d@example.com. For the %d wildcard, pass $i:

23 lines | src/DataFixtures/UserFixture.php

```
... lines 1 - 7
8  class UserFixture extends BaseFixture
9  {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(10, 'main_users', function($i) {
13             $user = new User();
14             $user->setEmail(sprintf('spacebar%d@example.com', $i));
... lines 15 - 17
18         });
... lines 19 - 20
21     }
22 }
```

Which will be one, two, three, four, five, six, seven, eight, nine, ten for the 10 calls.

The only other field is first name. To set this, we an use Faker, which we already setup inside BaseFixture: $this->faker->firstName:

23 lines | src/DataFixtures/UserFixture.php

```
... lines 1 - 7
8  class UserFixture extends BaseFixture
9  {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(10, 'main_users', function($i) {
13             $user = new User();
14             $user->setEmail(sprintf('spacebar%d@example.com', $i));
15             $user->setFirstName($this->faker->firstName);
... lines 16 - 17
18         });
... lines 19 - 20
21     }
22 }
```

Finally, at the bottom, return $user:

23 lines | src/DataFixtures/UserFixture.php

```
... lines 1 - 7
8  class UserFixture extends BaseFixture
9  {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(10, 'main_users', function($i) {
13             $user = new User();
14             $user->setEmail(sprintf('spacebar%d@example.com', $i));
15             $user->setFirstName($this->faker->firstName);
16
17             return $user;
18         });
... lines 19 - 20
21     }
22 }
```

And... we're done! This step had *nothing* to do with security: this is just boring Doctrine & PHP code inside a fancy createMany() method to make life easier.

Load 'em up:

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

Let's see what these look like:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM user'
```

Nice! Our User class is done! *Now*, it's time to add a login form and a login form *authenticator*: the *first* way that we'll allow our users to login.

Chapter 4: The Login Form

There are *two* steps to building a login form: the visual part - the HTML form itself - *and* the logic when you *submit* that form: finding the user, checking the password, and logging in. The interesting part is... if you think about it, the *first* part - the HTML form - has absolutely *nothing* to do with security. It's just... well... a boring, normal HTML form!

Let's get that built first. By the way, there are plans to add a make command to generate a login form and the security logic automatically, so that *we* only need to fill in a few details. That doesn't exist yet, so.. we'll do it manually. But, that's a bit better for learning anyways.

Creating the Login Controller & Template

To build the controller, let's at *least* use one shortcut. At your terminal, run:

```
$ php bin/console make:controller
```

to create a new class called SecurityController. Move over and open that:

20 lines | src/Controller/SecurityController.php

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

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class SecurityController extends AbstractController
{
    /**
     * @Route("/security", name="security")
     */
    public function index()
    {
        return $this->render('security/index.html.twig', [
            'controller_name' => 'SecurityController',
        ]);
    }
}
```

Ok: update the URL to /login, change the route name to app_login and the method to login():

20 lines | src/Controller/SecurityController.php

```
... lines 1 - 7
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function login()
    {
... lines 15 - 17
    }
}
```

We don't need to pass any variables yet, and we'll call the template login.html.twig:

20 lines | src/Controller/SecurityController.php

```
... lines 1 - 7
8  class SecurityController extends AbstractController
9  {
... lines 10 - 12
13     public function login()
14     {
15         return $this->render('security/login.html.twig', [
16
17         ]);
18     }
19 }
```

Next, down in templates/security, rename index.html.twig to login.html.twig. Let's try it! Move over, go to /login and... whoops!

> Variable controller_name does not exist.

Duh! I removed the variables that we *were* passing into the template:

20 lines | src/Controller/SecurityController.php

```
... lines 1 - 7
8  class SecurityController extends AbstractController
9  {
... lines 10 - 12
13     public function login()
14     {
15         return $this->render('security/login.html.twig', [
16
17         ]);
18     }
19 }
```

Empty all of the existing code from the template. Then, change the title to Login! and, for now, just add an h1 with "Login to the SpaceBar!":

8 lines | templates/security/login.html.twig

```
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Login!{% endblock %}
4
5  {% block body %}
6      <h1>Login to the SpaceBar!</h1>
7  {% endblock %}
```

Filling in the Security Logic & Login Form

Try it again: perfect! Well, not *perfect* - it looks *terrible*... and there's no login form yet. To fix *that* part, Google for "Symfony login form" to find a page on the Symfony docs that talks all about this. We're coming here so that we can steal some code!

Scroll down a bit until you see a login() method that has some logic in it. Copy the body, move back to our controller, and paste!

28 lines | src/Controller/SecurityController.php

```
... lines 1 - 8
9  class SecurityController extends AbstractController
10 {
... lines 11 - 13
14     public function login(AuthenticationUtils $authenticationUtils)
15     {
16         // get the login error if there is one
17         $error = $authenticationUtils->getLastAuthenticationError();
18
19         // last username entered by the user
20         $lastUsername = $authenticationUtils->getLastUsername();
... lines 21 - 25
26     }
27 }
```

This needs an AuthenticationUtils class as an argument. Add it: AuthenticationUtils $authenticationUtils:

28 lines | src/Controller/SecurityController.php

```
... lines 1 - 6
7  use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;
8
9  class SecurityController extends AbstractController
10 {
... lines 11 - 13
14     public function login(AuthenticationUtils $authenticationUtils)
15     {
... lines 16 - 25
26     }
27 }
```

Then, these two new variables are passed into Twig. Copy them, and also paste it:

28 lines | src/Controller/SecurityController.php

```
... lines 1 - 8
9  class SecurityController extends AbstractController
10 {
... lines 11 - 13
14     public function login(AuthenticationUtils $authenticationUtils)
15     {
... lines 16 - 21
22         return $this->render('security/login.html.twig', [
23             'last_username' => $lastUsername,
24             'error'         => $error,
25         ]);
26     }
27 }
```

In a few minutes, we're going to talk about *where* these two variables are set. They both deal with authentication.

But first, go back to the docs and find the login form. Copy this, move over and paste it into our body:

28 lines | templates/security/login.html.twig

```
... lines 1 - 4
{% block body %}
    <h1>Login to the SpaceBar!</h1>

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

    <form action="{{ path('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>
{% endblock %}
```

Notice: there is *nothing* special about this form: it has a username field, a password field and a submit button. And, we're going to customize it, so don't look too closely yet.

Move back to your browser to check things out. Bah!

> Unable to generate a URL for the named route "login"

This comes from login.html.twig. Of course! The template we copied is pointing to a route called login, but *our* route is called app_login:

28 lines | src/Controller/SecurityController.php

```
... lines 1 - 8
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function login(AuthenticationUtils $authenticationUtils)
    {
... lines 16 - 25
    }
}
```

Actually, just remove the action= entirely:

```
28 lines | templates/security/login.html.twig
... lines 1 - 4
{% block body %}
... lines 6 - 11
    <form method="post">
... lines 13 - 25
    </form>
{% endblock %}
```

If a form doesn't have an action attribute, it will submit right back to the *same* URL - /login - which is what I want anyways.

Refresh again. Perfect! Well, it still looks *awful*. Oof. To fix that, I'm going to replace the HTML form with some markup that looks nice in Bootstrap 4 - you can copy this from the code block on this page:

```
32 lines | templates/security/login.html.twig
... lines 1 - 10
{% block body %}
    <form class="form-signin" method="post">
        {% if error %}
            <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
        {% endif %}

        <h1 class="h3 mb-3 font-weight-normal">Please sign in</h1>
        <label for="inputEmail" class="sr-only">Email address</label>
        <input type="email" name="email" id="inputEmail" class="form-control" placeholder="Email address" required autofocus>
        <label for="inputPassword" class="sr-only">Password</label>
        <input type="password" name="password" id="inputPassword" class="form-control" placeholder="Password" required>
        <div class="checkbox mb-3">
            <label>
                <input type="checkbox" value="remember-me"> Remember me
            </label>
        </div>
        <button class="btn btn-lg btn-primary btn-block" type="submit">
            Sign in
        </button>
    </form>
{% endblock %}
```

Including the login.css File

Before we look at this new code, try it! Refresh! Still ugly! Dang! Oh yea, that's because we need to include a new CSS file for this markup.

If you downloaded the course code, you should have a tutorial/ directory with two CSS files inside. Copy login.css, find your public/ directory and paste the file into public/css:

39 lines | public/css/login.css

```
body {
    background-color: #fff;
}

.form-signin {
    width: 100%;
    max-width: 330px;
    padding: 15px;
    margin: auto;
    margin-top: 50px;
}

.form-signin .checkbox {
    font-weight: 400;
}

.form-signin .form-control {
    position: relative;
    box-sizing: border-box;
    height: auto;
    padding: 10px;
    font-size: 16px;
}

.form-signin .form-control:focus {
    z-index: 2;
}

.form-signin input[type="email"] {
    margin-bottom: -1px;
    border-bottom-right-radius: 0;
    border-bottom-left-radius: 0;
}

.form-signin input[type="password"] {
    margin-bottom: 10px;
    border-top-left-radius: 0;
    border-top-right-radius: 0;
}
```

So far in this series, we are *not* using Webpack Encore, which is an *awesome* tool for professionally combining and loading CSS and JS files. Instead, we're just putting CSS files into the public/ directory and pointing to them directly. If you want to learn more about Encore, go check out our Webpack Encore tutorial.

Anyways, we need to add a link tag for this new CSS file... but I *only* want to include it on *this* page, *not* on *every* page - we just *don't* need the CSS on every page. Look at base.html.twig:

```
69 lines | templates/base.html.twig
<!doctype html>
<html lang="en">

    <head>
... lines 5 - 8
        {% block stylesheets %}
            <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384
            <link rel="stylesheet" href="{{ asset('css/font-awesome.css') }}">
            <link rel="stylesheet" href="{{ asset('css/styles.css') }}">
        {% endblock %}
    </head>
... lines 15 - 66
    </body>
</html>
```

We're including three CSS files in the base layout. Ah, and they *all* live inside a block called stylesheets.

We basically want to add a *fourth* link tag right *below* these... but *only* on the login page. To do that, in login.html.twig, add block stylesheets and endblock:

```
32 lines | templates/security/login.html.twig
... lines 1 - 4
{% block stylesheets %}
... lines 6 - 8
{% endblock %}
... lines 10 - 32
```

This will *override* that block completely... which is actually *not* exactly what we want. Nope, we want to *add* to that block. To do that print parent():

```
32 lines | templates/security/login.html.twig
... lines 1 - 4
{% block stylesheets %}
    {{ parent() }}
... lines 7 - 8
{% endblock %}
... lines 10 - 32
```

This will print the content of the *parent* block - the 3 link tags - and then we can add the new link tag below: link, with href= and login.css. PhpStorm helps fill in the asset() function:

```
32 lines | templates/security/login.html.twig
... lines 1 - 4
{% block stylesheets %}
    {{ parent() }}

    <link rel="stylesheet" href="{{ asset('css/login.css') }}">
{% endblock %}
... lines 10 - 32
```

Now it should look good. Try it. Boom! Oh, but we don't need that h1 tag anymore.

The Fields of the Login Form

So even though this looks much better, it's still just a very boring HTML form. It has an email field and a password field... though, we won't add the password-checking logic until later. It also has a "remember me" checkbox that we'll learn how to

activate.

The point is: you can make your login form look *however* you want. The only special part is this error variable, which, when we're done, will be the authentication error if the user just entered a bad email or password:

32 lines | templates/security/login.html.twig

```
... lines 1 - 10
{% block body %}
    <form class="form-signin" method="post">
        {% if error %}
            <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
        {% endif %}
... lines 16 - 29
    </form>
{% endblock %}
```

I'll plan ahead and add a Bootstrap class for this:

32 lines | templates/security/login.html.twig

```
... lines 1 - 10
{% block body %}
    <form class="form-signin" method="post">
        {% if error %}
            <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
        {% endif %}
... lines 16 - 29
    </form>
{% endblock %}
```

Adding a Link to the Login Page

Ok. Login form is done! But... we probably need a *link* to this page. In the upper right corner, we have a cute user dropdown... which is *totally* hardcoded with fake data. Go back to base.html.twig and scroll down to find this. There it is! For now, let's comment-out that drop-down:

74 lines | templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
   ... lines 36 - 38
39                     {#
40                     <li class="nav-item dropdown" style="margin-right: 75px;">
41                         <a class="nav-link dropdown-toggle" href="http://example.com" id="navbarDropdownMenuLink" data-toggle="dropdown
42                           <img class="nav-profile-img rounded-circle" src="{{ asset('images/astronaut-profile.png') }}">
43                         </a>
44                         <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
45                             <a class="dropdown-item" href="#">Profile</a>
46                             <a class="dropdown-item" href="#">Create Post</a>
47                             <a class="dropdown-item" href="#">Logout</a>
48                         </div>
49                     </li>
50                     #}
51                 </ul>
52             </div>
53         </nav>
   ... lines 54 - 71
72     </body>
73 </html>
```

We'll re-add it later when we have *real* data. Then, copy a link from above, paste ithere and change it to Login with a link to app_login:

74 lines | templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
36                     <li class="nav-item">
37                         <a style="color: #fff;" class="nav-link" href="{{ path('app_login') }}">Login</a>
38                     </li>
39                     {#
40                     <li class="nav-item dropdown" style="margin-right: 75px;">
41                         <a class="nav-link dropdown-toggle" href="http://example.com" id="navbarDropdownMenuLink" data-toggle="dropdow
42                           <img class="nav-profile-img rounded-circle" src="{{ asset('images/astronaut-profile.png') }}">
43                         </a>
44                         <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
45                             <a class="dropdown-item" href="#">Profile</a>
46                             <a class="dropdown-item" href="#">Create Post</a>
47                             <a class="dropdown-item" href="#">Logout</a>
48                         </div>
49                     </li>
50                     #}
51                 </ul>
52             </div>
53         </nav>
   ... lines 54 - 71
72     </body>
73 </html>
```

Try it - refresh! We got it! HTML login form, check! We are now ready to fill in the logic of what happens when we *submit* the form. We'll do that in something called an "authenticator".

Chapter 5: Firewalls & Authenticator

We built a login form with a traditional route, controller and template. And so you *might* expect that because the form submits back to this same URL, the submit logic would live right inside this controller:

28 lines | src/Controller/SecurityController.php

```
... lines 1 - 8
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function login(AuthenticationUtils $authenticationUtils)
    {
... lines 16 - 25
    }
}
```

Like, if the request method is POST, we would grab the email, grab the password and do some magic.

What are Authentication Listeners / Authenticators?

Well... we are *not* going to do that. Symfony's security works in a bit of a "magical" way, at least, it feels like magic at first. At the beginning of every request, Symfony calls a set of "authentication listeners", or "authenticators". The job of each authenticator is to look at the request to see if there is any authentication info on it - like a submitted email & password or maybe an API token that's stored on a header. *If* an authenticator finds some info, it then tries to use it to find the user, check the password if there is one, and log in the user! *Our* job is to write these authenticators.

Understanding Firewalls

Open up config/packages/security.yaml. The *most* important section of this file is the firewalls key:

29 lines | config/packages/security.yaml

```
security:
... lines 2 - 8
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true

            # activate different ways to authenticate

            # http_basic: true
            # https://symfony.com/doc/current/security.html#a-configuring-how-your-users-will-authenticate

            # form_login: true
            # https://symfony.com/doc/current/security/form_login_setup.html
... lines 23 - 29
```

Ok, what the heck is a "firewall" in Symfony language? First, let's back up. There are *two* main parts of security: authentication and authorization. Authentication is all about finding out *who* you are and making you prove it. It's the login process. Authorization happens after authentication: it's all about determining whether or not you have access to something.

The *whole* job of the firewall is to *authenticate* you: to figure out who you are. And, it *usually* only makes sense to have *one* firewall in your app, *even* if you want your users to have *many* different ways to login - like a login form or API authentication.

But... hmm... Symfony gave us *two* firewalls by default! What the heck? Here's how it works: at the beginning of each request, Symfony determines the *one* firewall that matches the current request. It does that by comparing the URL to the regular expression pattern config. And if you look closely... the first firewall is a fake!

29 lines | config/packages/security.yaml

```
1  security:
   ... lines 2 - 8
9      firewalls:
10         dev:
11             pattern: ^/(_(profiler|wdt)|css|images|js)/
12             security: false
   ... lines 13 - 29
```

It becomes the active firewall if the URL starts with /_profiler, /_wdt, /css, /images or /js. *When* this is the active firewall, it sets security to false. Basically, this firewall exists *just* to make sure that we don't make our site *so* secure that we block the web debug toolbar or some of our static assets.

In reality, we only have *one* real firewall called main:

29 lines | config/packages/security.yaml

```
1  security:
   ... lines 2 - 8
9      firewalls:
   ... lines 10 - 12
13         main:
14             anonymous: true
15
16             # activate different ways to authenticate
17
18             # http_basic: true
19             # https://symfony.com/doc/current/security.html#a-configuring-how-your-users-will-authenticate
20
21             # form_login: true
22             # https://symfony.com/doc/current/security/form_login_setup.html
   ... lines 23 - 29
```

And because it does *not* have a pattern key, it will be the active firewall for *all* URLs, except the ones matched above. Oh, and, in case you're wondering, the names of the firewalls, dev and main are totally meaningless.

Anyways, because the job of a firewall is to authenticate the user, most of the config that goes below a firewall relates to "activating" new authentication listeners - those things that execute at the beginning of Symfony and try to log in the user. We'll add some new config here pretty soon.

Oh, and see this anonymous: true part?

29 lines | config/packages/security.yaml

```
1  security:
   ... lines 2 - 8
9      firewalls:
   ... lines 10 - 12
13         main:
14             anonymous: true
   ... lines 15 - 29
```

Tip

Starting with Symfony 4.4.1 and 5.0.1, instead of anonymous: true you will see anonymous: lazy. Both should not behave in any noticeably different way - it's basically the same.

Keep that. This allows *anonymous* requests to pass through this firewall so that users can access your public pages, without needing to login. *Even* if you want to require authentication on *every* page of your site, keep this. There's a different place - access_control - where we can do this better:

29 lines | config/packages/security.yaml

```
1  security:
   ... lines 2 - 23
24     # Easy way to control access for large sections of your site
25     # Note: Only the *first* access control that matches will be used
26     access_control:
27         # - { path: ^/admin, roles: ROLE_ADMIN }
28         # - { path: ^/profile, roles: ROLE_USER }
```

Creating the Authentication with make:auth

Ok, let's get to work! To handle the login form submit, we need to create our very first authenticator. Find your terminal and run make:auth:

```
$ php bin/console make:auth
```

Tip

Since MakerBundle v1.8.0 this command asks you to choose between an "Empty authenticator" and a "Login form authenticator". Choose the first option to follow along with the tutorial exactly. Or choose the second to get more generated code than the video!

Call the new class LoginFormAuthenticator.

Nice! This creates one new file: src/Security/LoginFormAuthenticator.php:

54 lines | src/Security/LoginFormAuthenticator.php

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

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Guard\AbstractGuardAuthenticator;

class LoginFormAuthenticator extends AbstractGuardAuthenticator
{
    public function supports(Request $request)
    {
        // todo
    }

    public function getCredentials(Request $request)
    {
        // todo
    }

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

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

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

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

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

    public function supportsRememberMe()
    {
        // todo
    }
}
```

This class is awesome: it basically has a method for each step of the authentication process. Before we walk through each

one, because this authenticator will be for a login form, there's a different base class that allows us to... well... do less work!

Instead of extends AbstractGuardAuthenticator use extends AbtractFormLoginAuthenticator:

43 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 8
9  use Symfony\Component\Security\Guard\Authenticator\AbstractFormLoginAuthenticator;
10
11 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
12 {
... lines 13 - 41
42 }
```

I'll remove the old use statement.

Thanks to this, we no longer need onAuthenticationFailure(), start() or supportsRememberMe(): they're all handled for us:

43 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 8
9  use Symfony\Component\Security\Guard\Authenticator\AbstractFormLoginAuthenticator;
10
11 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
12 {
13     public function supports(Request $request)
14     {
15         // todo
16     }
17
18     public function getCredentials(Request $request)
19     {
20         // todo
21     }
22
23     public function getUser($credentials, UserProviderInterface $userProvider)
24     {
25         // todo
26     }
27
28     public function checkCredentials($credentials, UserInterface $user)
29     {
30         // todo
31     }
32
33     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
34     {
35         // todo
36     }
... lines 37 - 41
42 }
```

But don't worry, when we create an API token authenticator later, we *will* learn about these methods. We *do* now need one *new* method. Go to the "Code"->"Generate" menu, or Command+N on a Mac, and select "Implement Methods" to generate getLoginUrl():

```
43 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 10
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 13 - 37
    protected function getLoginUrl()
    {
        // TODO: Implement getLoginUrl() method.
    }
}
```

Activating the Authenticator in security.yaml

Perfect! Unlike a lot of features in Symfony, this authenticator won't be activated automatically. To tell Symfony about it, go *back* to security.yaml. Under the main firewall, add a new guard key, a new authenticators key below that, and add one item in that array: App\Security\LoginFormAuthenticator:

```
33 lines | config/packages/security.yaml
security:
... lines 2 - 8
    firewalls:
... lines 10 - 12
        main:
... lines 14 - 15
            guard:
                authenticators:
                    - App\Security\LoginFormAuthenticator
... lines 19 - 33
```

The whole authenticator system comes from a part of the Security component called "Guard", hence the name. The important part is that, as *soon* as we add this, at the beginning of *every* request, Symfony will call the supports() method on our authenticator.

To prove it, add a die() statement with a message:

```
43 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 10
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
    public function supports(Request $request)
    {
        die('Our authenticator is alive!');
    }
... lines 17 - 41
}
```

Then, move over and, refresh! Got it! And it doesn't matter *what* URL we go to: the supports() method is always called at the start of the request.

And now, we're in business! Let's fill in these methods and get our user logged in.

Chapter 6: Login Form Authenticator

Now that we've added our authenticator under the authenticators key:

33 lines | config/packages/security.yaml

```
1  security:
   ... lines 2 - 8
9      firewalls:
   ... lines 10 - 12
13         main:
   ... lines 14 - 15
16             guard:
17                 authenticators:
18                     - App\Security\LoginFormAuthenticator
   ... lines 19 - 33
```

Symfony calls its supports() method at the beginning of *every* request, which is why we see this little die statement:

43 lines | src/Security/LoginFormAuthenticator.php

```
   ... lines 1 - 10
11 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
12 {
13     public function supports(Request $request)
14     {
15         die('Our authenticator is alive!');
16     }
   ... lines 17 - 41
42 }
```

These authenticator classes are really cool because *each* method controls just *one* small part of the authentication process.

The supports() Method

The first method - supports() - is called on every request. Our job is simple: to return true if this request contains authentication info that this authenticator knows how to process. And if not, to return false.

In this case, when we submit the login form, it POSTs to /login. So, our authenticator should *only* try to authenticate the user in that exact situation. Return $request->attributes->get('_route') === 'app_login':

45 lines | src/Security/LoginFormAuthenticator.php

```
   ... lines 1 - 10
11 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
12 {
13     public function supports(Request $request)
14     {
15         // do your work when we're POSTing to the login page
16         return $request->attributes->get('_route') === 'app_login'
   ... line 17
18     }
   ... lines 19 - 43
44 }
```

Let me... explain this. If you look in SecurityController, the *name* of our login route is app_login:

```
28 lines | src/Controller/SecurityController.php
... lines 1 - 8
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function login(AuthenticationUtils $authenticationUtils)
    {
... lines 16 - 25
    }
}
```

And, though you don't need to do it very often, if you want to find out the *name* of the currently-matched route, you can do that by reading this special _route key from the request attributes. In other words, this is checking to see if the URL is /login. We also only want our authenticator to try to login the user if this is a POST request. So, add && $request->isMethod('POST'):

```
45 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 10
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
    public function supports(Request $request)
    {
        // do your work when we're POSTing to the login page
        return $request->attributes->get('_route') === 'app_login'
            && $request->isMethod('POST');
    }
... lines 19 - 43
}
```

Here's how this works: if we return false from supports(), nothing else happens. Symfony doesn't call *any* other methods on our authenticator, and the request continues on like normal to our controller, like nothing happened. It's not an authentication *failure* - it's just that nothing happens at all.

If we return true from supports(), well, that's when the fun starts. If we return true, Symfony will immediately call getCredentials():

```
45 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 10
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 13 - 19
    public function getCredentials(Request $request)
    {
... line 22
    }
... lines 24 - 43
}
```

To see if things are working, let's just dump($request->request->all()), then die():

```
45 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 10
11 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
12 {
... lines 13 - 19
20     public function getCredentials(Request $request)
21     {
22         dump($request->request->all());die;
23     }
... lines 24 - 43
44 }
```

I know, that looks funny. *Unrelated* to security, if you want to read POST data off of the request, you use the $request->request property.

Anyways, let's try it! Go back to your browser and hit enter on the URL so that it makes a GET request to /login. Hello login page! Our supports() method just returned false. And so, the request continued *anonymously*, like normal.

Log in with one of our dummy users: spacebar1@example.com. The password doesn't matter. And... enter! Yes! *This* time, because this is a POST request to /login, supports() returns true! So, Symfony calls getCredentials() and our dump fires! As expected, we can see the email and password POST parameters, because the login form uses these names:

```
32 lines | templates/security/login.html.twig
... lines 1 - 10
11 {% block body %}
12     <form class="form-signin" method="post">
... lines 13 - 18
19         <input type="email" name="email" id="inputEmail" class="form-control" placeholder="Email address" required autofocus>
... line 20
21         <input type="password" name="password" id="inputPassword" class="form-control" placeholder="Password" required>
... lines 22 - 29
30     </form>
31 {% endblock %}
```

The Brand-New dd() Function

Oh, and I want to show you a *quick* new Easter egg in Symfony 4.1, *unrelated* to security. Instead of dump() and die, use dd() and then remove the die:

```
45 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 10
11 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
12 {
... lines 13 - 19
20     public function getCredentials(Request $request)
21     {
22         dd($request->request->all());
23     }
... lines 24 - 43
44 }
```

Refresh! Same result. This is just a nice, silly shortcut: dd() is dump() and die. We'll use it... because... why not?

The getCredentials() Method

Back to work! Our job in getCredentials() is simple: to read our authentication credentials off of the request and return them. In this case, we'll return the email and password. But, if this were an API token authenticator, we would return that token. We'll see that later.

Return an array with an email key set to $request->request->get('email') and password set to $request->request->get('password'):

```
48 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 10
11 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
12 {
... lines 13 - 19
20     public function getCredentials(Request $request)
21     {
22         return [
23             'email' => $request->request->get('email'),
24             'password' => $request->request->get('password'),
25         ];
26     }
... lines 27 - 46
47 }
```

I'm just inventing these email and password keys for the new array: we can really return *whatever* we want from this method. Because, after we return from getCredentials(), Symfony will immediately call getUser() and pass this array *back* to us as the first $credentials argument:

```
48 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 10
11 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
12 {
... lines 13 - 27
28     public function getUser($credentials, UserProviderInterface $userProvider)
29     {
... line 30
31     }
... lines 32 - 46
47 }
```

Let's see that in action: dd($credentials):

```
48 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 10
11 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
12 {
... lines 13 - 27
28     public function getUser($credentials, UserProviderInterface $userProvider)
29     {
30         dd($credentials);
31     }
... lines 32 - 46
47 }
```

Move back to your browser and, refresh! Coincidentally, it dumps the *exact* same thing as before. But, this time, it's coming from line 30 - our line in getUser().

The getUser() Method

Let's keep going! Our job in getUser() is to use these $credentials to return a User object, or null if the user isn't found. Because we're storing our users in the database, we need to *query* for the user via their email. And to do that, we need the UserRepository that was generated with our entity.

At the top of the class, add public function __construct() with a UserRepository $userRepository argument:

```
56 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 4
5  use App\Repository\UserRepository;
... lines 6 - 11
12 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
13 {
... lines 14 - 15
16     public function __construct(UserRepository $userRepository)
17     {
... line 18
19     }
... lines 20 - 54
55 }
```

I'll hit Alt+Enter and select "Initialize Fields" to add that property and set it:

```
56 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 4
5  use App\Repository\UserRepository;
... lines 6 - 11
12 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
13 {
14     private $userRepository;
15
16     public function __construct(UserRepository $userRepository)
17     {
18         $this->userRepository = $userRepository;
19     }
... lines 20 - 54
55 }
```

Back down in getUser(), just return $this->userRepository->findOneBy() to query by email, set to $credentials['email']:

```
56 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
13 {
... lines 14 - 35
36     public function getUser($credentials, UserProviderInterface $userProvider)
37     {
38         return $this->userRepository->findOneBy(['email' => $credentials['email']]);
39     }
... lines 40 - 54
55 }
```

This will return our User object, or null. The *cool* thing is that if this returns null, the whole authentication process will stop, and the user will see an error. But if we return a User object, then Symfony immediately calls checkCredentials(), and passes it the same $credentials and the User object *we* just returned:

```
56 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
13 {
... lines 14 - 40
41     public function checkCredentials($credentials, UserInterface $user)
42     {
... line 43
44     }
... lines 45 - 54
55 }
```

Inside, dd($user) so we can see if things are working:

```
56 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
13 {
... lines 14 - 40
41     public function checkCredentials($credentials, UserInterface $user)
42     {
43         dd($user);
44     }
... lines 45 - 54
55 }
```

Refresh and... got it! That's *our* User object!

The checkCredentials() Method

Ok, final step: checkCredentials(). This is your opportunity to check to see if the user's password is correct, or any other last, security checks. Right now... well... we don't have a password, so, let's return true:

```
57 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
13 {
... lines 14 - 40
41     public function checkCredentials($credentials, UserInterface $user)
42     {
43         // only needed if we need to check a password - we'll do that later!
44         return true;
45     }
... lines 46 - 55
56 }
```

And actually, in *many* systems, simply returning true is perfect! For example, if you have an API token system, there's no password.

If you *did* return false, authentication would fail and the user would see an "Invalid Credentials" message. We'll see that soon.

But, when you return *true*... authentication is successful! Woo! To figure out what to *do*, now that the user is authenticated, Symfony calls onAuthenticationSuccess():

```
57 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
13 {
... lines 14 - 46
47     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
48     {
... line 49
50     }
... lines 51 - 55
56 }
```

Put a dd() here that says "Success":

```
57 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
13 {
... lines 14 - 46
47     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
48     {
49         dd('success!');
50     }
... lines 51 - 55
56 }
```

Move over and... refresh the POST! Yes! We hit it! At this point, we have *fully* filled in *all* the authentication logic. We used supports() to tell Symfony whether or not our authenticator should be used in this request, fetched credentials off of the request, used those to find the user, and returned true in checkCredentials() because we don't have a password.

Next, let's fill in these *last* two methods and *finally* see - for *real* - that our user is logged in. We'll also learn a bit more about what happens when authentication fails and how the error message is rendered.

Chapter 7: Redirecting on Success & the User Provider

If our authenticator is able to return a User from getUser() *and* we return true from checkCredentials():

57 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
13 {
... lines 14 - 35
36     public function getUser($credentials, UserProviderInterface $userProvider)
37     {
38         return $this->userRepository->findOneBy(['email' => $credentials['email']]);
39     }
40
41     public function checkCredentials($credentials, UserInterface $user)
42     {
43         // only needed if we need to check a password - we'll do that later!
44         return true;
45     }
... lines 46 - 55
56 }
```

Then, congrats! Our user is logged in! The *last* question Symfony asks us is: now what? Now that the user is authenticated, what do you want to do?

For a form login system, the answer is: redirect to another page. For an API token system, the answer is... um... nothing! Just allow the request to continue like normal.

This is why, once authentication is successful, Symfony calls onAuthenticationSuccess():

57 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
13 {
... lines 14 - 46
47     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
48     {
49         dd('success!');
50     }
... lines 51 - 55
56 }
```

We can either return a Response object here - which will be immediately sent back to the user - *or* nothing... in which case, the request would continue to the controller.

Redirecting on Success

So, hmm, *we* want to *redirect* the user to another page. So... how do we redirect in Symfony? If you're in a controller, there's a redirectToRoute() shortcut method. Hold Command or Ctrl and click into that. I want to see what this does.

Ok, it leverages two *other* methods: redirect() and generateUrl(). Look at redirect(). Oh.... So, to redirect in Symfony, you return a RedirectResponse object, which is a sub-class of the normal Response. It just sets the status code to 301 or 302 and adds a Location header that points to where the user should go. That makes sense: a redirect is just a special type of response!

The *other* method, generateUrl(), is a shortcut to use the "router" to convert a route *name* into its URL. Go back to the

controller and clear out our dummy code.

Back in LoginFormAuthenticator, return a new RedirectResponse(). Hmm, let's just send the user to the homepage. But, *of course*, we don't ever hardcode URLs in Symfony. Instead, we need to *generate* a URL to the route named app_homepage:

```
65 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 25
26     /**
27      * @Route("/", name="app_homepage")
28      */
29     public function homepage(ArticleRepository $repository)
30     {
... lines 31 - 35
36     }
... lines 37 - 63
64 }
```

We *know* how to generate URLs in Twig - the path() function. But, how can we do it in PHP? The answer is... with Symfony's *router* service. To find out how to get it, run:

```
$ php bin/console debug:autowiring
```

Look for something related to routing... there it is! Actually, there are a few different router-related interfaces... but they're all different ways to get the *same* service. I usually use RouterInterface.

Back on top, add a *second* constructor argument: RouterInterface $router:

```
61 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 7
8 use Symfony\Component\Routing\RouterInterface;
... lines 9 - 13
14 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
15 {
... lines 16 - 18
19     public function __construct(UserRepository $userRepository, RouterInterface $router)
20     {
... lines 21 - 22
23     }
... lines 24 - 59
60 }
```

I'll hit Alt+Enter and select "Initialize Fields" to create that property and set it:

```
61 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 7
8  use Symfony\Component\Routing\RouterInterface;
... lines 9 - 13
14 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
15 {
... line 16
17     private $router;
18
19     public function __construct(UserRepository $userRepository, RouterInterface $router)
20     {
... line 21
22         $this->router = $router;
23     }
... lines 24 - 59
60 }
```

Then, back down below, use $this->router->generate() to make a URL to app_homepage:

```
61 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 13
14 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
15 {
... lines 16 - 50
51     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
52     {
53         return new RedirectResponse($this->router->generate('app_homepage'));
54     }
... lines 55 - 59
60 }
```

Ok! We still have one empty method:

```
61 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 13
14 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
15 {
... lines 16 - 55
56     protected function getLoginUrl()
57     {
58         // TODO: Implement getLoginUrl() method.
59     }
60 }
```

But, forget that! We're ready! Go back to your browser, and hit enter to show the login page again. Let's walk through the *entire* process. Use the same email, *any* password and... enter! It worked! How do I know? Check out the web debug toolbar! We are logged in as spacebar1@example.com!

Authentication & the Session: User Provider

This is even *cooler* than it looks. Think about it: we made a POST request to /login and became authenticated thanks to our authenticator. Then, we were redirected to the homepage... where our authenticator did nothing, because its supports() method returned false.

The *only* reason we're *still* logged in - even though our authenticator did nothing on this request - is that user authentication info is stored to the session. At the beginning of every request, that info is *loaded* from the session and we're logged in. Cool!

Look back at your security.yaml file. Remember this user provider thing that was setup for us?

33 lines | config/packages/security.yaml

```yaml
security:
    # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
... lines 9 - 33
```

This is a class that *helps* with the process of loading the user info from the session.

Honestly, it's a little bit confusing, but super important. Here's the deal: when you refresh the page, the User object is loaded from the session. But, we need to make sure that the object isn't out of date with the database. Think about it. Imagine we login at work. Then, we login at home and update our first name in the database. The next day, when we go back to work, we reload the page. Well... if we did *nothing* else, the User object we reloaded from the session for *that* browser would have our *old* first name. That would probably cause some weird issues.

So, that's the job of the user provider. When we refresh, the user provider takes the User object from the session and uses its id to query for a *fresh* User object. It all happens invisibly, which is *great*. But it *is* an important, background detail.

Next, I want to see what happens when we *fail* authentication. What does the user see? How are errors displayed? And how can we control them?

Chapter 8: Authentication Errors

Go back to the login page. I wonder what happens if we *fail* the login... which, is only possible right now if we use a non-existent email address. Oh!

> Cannot redirect to an empty URL

Filling in getLoginUrl()

Hmm: this is coming from AbstractFormLoginAuthenticator *our* authenticator's base class. If you dug a bit, you'd find out that, on failure, that authenticator class is calling getLoginUrl() and trying to redirect *there*. And, yea, that makes sense: if we fail login, the user should be redirected *back* to the login page. To make this actually work, all *we* need to do is fill in this method.

No problem: return $this->router->generate('app_login'):

61 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 13
14 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
15 {
... lines 16 - 55
56     protected function getLoginUrl()
57     {
58         return $this->router->generate('app_login');
59     }
60 }
```

Ok, try it again: refresh and... perfect! Hey! You can even see an error message on top:

> Username could not be found.

We get *that* exact error because of *where* the authenticator fails: we *failed* to return a user from getUser():

61 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 13
14 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
15 {
... lines 16 - 39
40     public function getUser($credentials, UserProviderInterface $userProvider)
41     {
42         return $this->userRepository->findOneBy(['email' => $credentials['email']]);
43     }
... lines 44 - 59
60 }
```

In a little while, we'll learn how to customize this message because... probably saying "Email" could not be found would make more sense.

The *other* common place where your authenticator can fail is in the checkCredentials() method:

```
61 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 13
14 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
15 {
... lines 16 - 44
45     public function checkCredentials($credentials, UserInterface $user)
46     {
47         // only needed if we need to check a password - we'll do that later!
48         return true;
49     }
... lines 50 - 59
60 }
```

Try returning false here for a second:

```
// ...
    public function checkCredentials($credentials, UserInterface $user)
    {
        return false;
    }
// ...
```

Then, login with a *legitimate* user. Nice!

Invalid credentials.

Anyways, go change that back to true:

```
61 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 13
14 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
15 {
... lines 16 - 44
45     public function checkCredentials($credentials, UserInterface $user)
46     {
47         // only needed if we need to check a password - we'll do that later!
48         return true;
49     }
... lines 50 - 59
60 }
```

How Authentication Errors are Stored

What I *really* want to find out is: where are these errors coming from? In SecurityController, we're getting the error by calling some $authenticationUtils->getLastAuthenticationError() method:

28 lines | src/Controller/SecurityController.php

```
... lines 1 - 6
7  use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;
8
9  class SecurityController extends AbstractController
10 {
... lines 11 - 13
14     public function login(AuthenticationUtils $authenticationUtils)
15     {
16         // get the login error if there is one
17         $error = $authenticationUtils->getLastAuthenticationError();
... lines 18 - 21
22         return $this->render('security/login.html.twig', [
... line 23
24             'error'         => $error,
25         ]);
26     }
27 }
```

We're passing that into the template and rendering its messageKey property... with some translation magic we'll talk about soon too:

32 lines | templates/security/login.html.twig

```
... lines 1 - 10
11 {% block body %}
12     <form class="form-signin" method="post">
13         {% if error %}
14             <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
15         {% endif %}
... lines 16 - 29
30     </form>
31 {% endblock %}
```

The point is: we magically fetch the "error" from... somewhere and render it. Let's demystify that. Go back to the top of your authenticator and hold command or control to click into AbstractFormLoginAuthenticator.

In reality, when authentication fails, this onAuthenticationFailure() method is called. It's a bit technical, but when authentication fails, internally, it's because something threw an AuthenticationException, which is passed to this method. And, ah: this method *stores* that exception onto a special key in the session! Then, back in the controller, the lastAuthenticationError() method is just a *shortcut* to read that key *off* of the session!

So, it's simple: our authenticator stores the error in the session and then we read the error *from* the session in our controller and render it:

28 lines | src/Controller/SecurityController.php

```
... lines 1 - 8
9  class SecurityController extends AbstractController
10 {
... lines 11 - 13
14     public function login(AuthenticationUtils $authenticationUtils)
15     {
16         // get the login error if there is one
17         $error = $authenticationUtils->getLastAuthenticationError();
... lines 18 - 25
26     }
27 }
```

The *last* thing onAuthenticationFailure() does is call our getLoginUrl() method and redirect there.

Filling in the Last Email

Go back to the login form and fail authentication again with a fake email. We see the error... but the email field is empty - that's not ideal. For convenience, it *should* pre-fill with the email I just entered.

Look at the controller again. Hmm: we *are* calling a getLastUsername() method and passing that into the template:

32 lines | templates/security/login.html.twig

```
... lines 1 - 10
11 {% block body %}
12     <form class="form-signin" method="post">
... lines 13 - 18
19         <input type="email" name="email" id="inputEmail" class="form-control" placeholder="Email address" required autofocus>
... lines 20 - 29
30     </form>
31 {% endblock %}
```

Oh, but I forgot to render it! Add value= and print last_username:

32 lines | templates/security/login.html.twig

```
... lines 1 - 10
11 {% block body %}
12     <form class="form-signin" method="post">
... lines 13 - 18
19         <input type="email" value="{{ last_username }}" name="email" id="inputEmail" class="form-control" placeholder="Email address"
... lines 20 - 29
30     </form>
31 {% endblock %}
```

But... we're not quite done. Unlike the error message, the last user name is *not* automatically stored to the session. This is something that *we* need to do inside of our LoginFormAuthenticator. But, it's super easy. Inside getCredentials(), instead of returning, add $credentials = :

69 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 14
15 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
16 {
... lines 17 - 32
33     public function getCredentials(Request $request)
34     {
35         $credentials = [
36             'email' => $request->request->get('email'),
37             'password' => $request->request->get('password'),
38         ];
... lines 39 - 45
46     }
... lines 47 - 67
68 }
```

Now, set the email onto the session with $request->getSession()->set(). Use a special key: Security - the one from the Security component - ::LAST_USERNAME and set this to $credentials['email']:

```
69 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 14
15 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
16 {
... lines 17 - 32
33     public function getCredentials(Request $request)
34     {
35         $credentials = [
36             'email' => $request->request->get('email'),
37             'password' => $request->request->get('password'),
38         ];
39
40         $request->getSession()->set(
41             Security::LAST_USERNAME,
42             $credentials['email']
43         );
... lines 44 - 45
46     }
... lines 47 - 67
68 }
```

Then, at the bottom, return $credentials:

```
69 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 14
15 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
16 {
... lines 17 - 32
33     public function getCredentials(Request $request)
34     {
35         $credentials = [
36             'email' => $request->request->get('email'),
37             'password' => $request->request->get('password'),
38         ];
39
40         $request->getSession()->set(
41             Security::LAST_USERNAME,
42             $credentials['email']
43         );
44
45         return $credentials;
46     }
... lines 47 - 67
68 }
```

Try it! Go back, login with that same email address and... nice! Both the error *and* the last email are read from the session and displayed.

Next: let's learn how to *customize* these error messages. And, we really need a way to logout.

Chapter 9: Customizing Errors & Logout

If we enter an email that doesn't exist, we get this

> Username could not be found

error message. And, as we saw a moment ago, if we return false from checkCredentials(), the error is something about "Invalid credentials".

The point is, depending on *where* authentication fails, the user will see one of these two messages.

The question *now* is, what if we want to customize those? Because, username could not be found? Really? In an app that doesn't use usernames!? That's... confusing.

Customizing Error Messages

There are two ways to control these error messages. The first is by throwing a very special exception class from anywhere in your authenticator. It's called CustomUserMessageAuthenticationException. When you do this, you can create your own message. We'll do this later when we build an API authenticator.

The second way is to *translate* this message. No, this isn't a tutorial about translations. But, if you look at your login template, when we print this error.messageKey thing, we are *already* running it through Symfony's translation filter:

32 lines | templates/security/login.html.twig

```
... lines 1 - 10
{% block body %}
    <form class="form-signin" method="post">
        {% if error %}
            <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
        {% endif %}
... lines 16 - 29
    </form>
{% endblock %}
```

Another way to look at this is on the web debug toolbar. See this little translation icon? Click that! Cool: you can see all the information about translations that are being processed on this page. Not surprisingly - since we're not trying to translate anything - there's only one: "Username could not be found."... which... is being translated into... um... "Username could not be found."

Internally, Symfony ships with translation files that will translate these authentication error messages into most other languages. For example, if we were using the es locale, we would see this message in Spanish.

Ok, so, why the heck do we care about all of this? *Because*, the errors are passed through the translator, we can *translate* the English into... *different* English!

Check this out: in your translations/ directory, create a security.en.yaml file:

```
32 lines | templates/security/login.html.twig
... lines 1 - 10
{% block body %}
    <form class="form-signin" method="post">
        {% if error %}
            <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
        {% endif %}
... lines 16 - 29
    </form>
{% endblock %}
```

This file is called *security* because of this security key in the translator. This is called the translation "domain" - it's kind of a translation category - a way to organize things.

Anyways, inside the file, copy the message id, paste that inside quotes, and assign it to our newer, hipper message:

> Oh no! It doesn't look like that email exists!

```
1 lines | translations/security.en.yaml
"Username could not be found.": "Oh no! It doesn't look like that email exists!"
```

That's it! If you go back to your browser and head over to the login page, in theory, if you try failing login now, this should work instantly. But... no! Same message. Today is *not* our lucky day.

This is thanks to a small, um, bug in Symfony. Yes, yes, they *do* happen sometimes, and this bug only affects our development... slightly. Here's the deal: whenever you create a *new* translation file, Symfony won't see that file until you manually clear the cache. In your terminal, run:

```
$ php bin/console cache:clear
```

When that finishes, go back and try it again: login with a bad email and... awesome!

Logging Out

Hey! Our login authentication system is... done! And... not that I want to rush our moment of victory - we did it! - but now that our friendly alien users can log *in*... they'll probably need a way to log *out*. They're just never satisfied...

Right now, I'm still logged in as spacebar1@example.com. Let's close a few files. Then, open SecurityController. Step 1 to creating a logout system is to create the route. Add public function logout():

```
36 lines | src/Controller/SecurityController.php
... lines 1 - 8
class SecurityController extends AbstractController
{
... lines 11 - 30
    public function logout()
    {
... line 33
    }
}
```

Above this, use the normal @Route("/logout") with the name app_logout:

```
36 lines | src/Controller/SecurityController.php
... lines 1 - 5
6  use Symfony\Component\Routing\Annotation\Route;
... lines 7 - 8
9  class SecurityController extends AbstractController
10 {
... lines 11 - 27
28     /**
29      * @Route("/logout", name="app_logout")
30      */
31     public function logout()
32     {
... line 33
34     }
35 }
```

And *this* is where things get interesting... We *do* need to create this route... but we *don't* need to write any *logic* to log out the user. In fact, I'm feeling so sure that I'm going to throw a new Exception():

> will be intercepted before getting here

```
36 lines | src/Controller/SecurityController.php
... lines 1 - 8
9  class SecurityController extends AbstractController
10 {
... lines 11 - 27
28     /**
29      * @Route("/logout", name="app_logout")
30      */
31     public function logout()
32     {
33         throw new \Exception('Will be intercepted before getting here');
34     }
35 }
```

Remember how "authenticators" run automatically at the beginning of every request, before the controllers? The logout process works the same way. All *we* need to do is tell Symfony what *URL* we want to use for logging out.

In security.yaml, under your firewall, add a new key: logout and, below that, path set to our logout route. So, for us, it's app_logout:

```
36 lines | config/packages/security.yaml
1  security:
... lines 2 - 8
9      firewalls:
... lines 10 - 12
13         main:
... lines 14 - 19
20             logout:
21                 path: app_logout
... lines 22 - 36
```

That's it! *Now*, whenever a user goes to the app_logout route, at the beginning of that request, Symfony will automatically log the user out and then redirect them... *all* before the controller is ever executed.

So... let's try it! Change the URL to /logout and... yes! The web debug toolbar reports that we are once again floating around the site anonymously.

By the way, there *are* a few other things that you can customize under the logout section, like *where* to redirect. You can find those options in the Symfony reference section.

But now, we need to talk about CSRF protection. We'll also add remember me functionality to our login form with almost no effort.

Chapter 10: CSRF Protection

Our login form is working perfectly. But... there's one *tiny* annoying detail that we need to talk about: the fact that *every* form on your site that performs an action - like saving something *or* logging you in - needs to be protected by a CSRF token. When you use Symfony's form system, CSRF protection is built in. But because we're *not* using it here, we need to add it manually. Fortunately, it's no big deal!

Adding the CSRF Input Field

Step one: we need to add an <input type="hidden"> field to our form. For the name... this could be anything, how about _csrf_token. For the value, use a special csrf_token() Twig function and pass it the string authenticate:

37 lines | templates/security/login.html.twig

```
... lines 1 - 10
11 {% block body %}
12     <form class="form-signin" method="post">
... lines 13 - 22
23         <input type="hidden" name="_csrf_token"
24             value="{{ csrf_token('authenticate') }}"
25         >
... lines 26 - 34
35     </form>
36 {% endblock %}
```

What's that? It's sort of a "name" that's used when creating this token, and it could be anything. We'll use that *same* name in a minute when we check to make sure the submitted token is valid.

Verifying the CSRF Token

In fact, what a great idea! Let's do that now! Step 2 happens inside of LoginFormAuthenticator. Start in getCredentials(): in addition to the email and password, let's *also* return a csrf_token key set to $request->request->get('_csrf_token'):

80 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 17
18 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
19 {
... lines 20 - 37
38     public function getCredentials(Request $request)
39     {
40         $credentials = [
... lines 41 - 42
43             'csrf_token' => $request->request->get('_csrf_token'),
44         ];
... lines 45 - 51
52     }
... lines 53 - 78
79 }
```

Next, in getUser(), *this* is where we'll check the CSRF token. We could do it down in checkCredentials(), but I'd rather make sure it's valid *before* we query for the user.

So... how do we check if a CSRF token is valid? Well... like pretty much everything in Symfony, it's done with a service. Without even reading the documentation, we can probably find the service we need by running:

```
$ php bin/console debug:autowiring
```

And searching for CSRF. Yea! There are a few: a CSRF token manager, a token generator and some sort of token storage. The second two are a bit lower-level: the CsrfTokenManagerInterface is what we want.

To get this, go back to your constructor and add a third argument: CsrfTokenManagerInterface. I'll re-type the "e" and hit tab to auto-complete that so that PhpStorm politely adds the use statement on top of the file:

```
80 lines | src/Security/LoginFormAuthenticator.php
   ... lines 1 - 14
15 use Symfony\Component\Security\Csrf\CsrfTokenManagerInterface;
   ... lines 16 - 17
18 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
19 {
   ... lines 20 - 23
24     public function __construct(UserRepository $userRepository, RouterInterface $router, CsrfTokenManagerInterface $csrfTokenMana
25     {
   ... lines 26 - 28
29     }
   ... lines 30 - 78
79 }
```

Call the argument $csrfTokenManager and hit Alt+Enter to initialize that field:

```
80 lines | src/Security/LoginFormAuthenticator.php
   ... lines 1 - 14
15 use Symfony\Component\Security\Csrf\CsrfTokenManagerInterface;
   ... lines 16 - 17
18 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
19 {
   ... lines 20 - 21
22     private $csrfTokenManager;
23
24     public function __construct(UserRepository $userRepository, RouterInterface $router, CsrfTokenManagerInterface $csrfTokenMana
25     {
   ... lines 26 - 27
28         $this->csrfTokenManager = $csrfTokenManager;
29     }
   ... lines 30 - 78
79 }
```

Perfect! To see how this interface works, hold Command or Ctrl and click into it. Ok: we have getToken(), refreshToken(), removeToken() and... yes: isTokenValid()! Apparently we need to pass this a CsrfToken object, which *itself* needs two arguments: id and value. The id is referring to that string - authenticate - or whatever string you used when you originally generated the token:

```
37 lines | templates/security/login.html.twig
... lines 1 - 10
{% block body %}
    <form class="form-signin" method="post">
... lines 13 - 22
        <input type="hidden" name="_csrf_token"
            value="{{ csrf_token('authenticate') }}"
        >
... lines 26 - 34
    </form>
{% endblock %}
```

The value is the CSRF token *value* that the user submitted.

Let's close all of this. Go back to LoginFormAuthenticator and find getUser(). First, add $token = new CsrfToken() and pass this authenticate and then the submitted token: $credentials['csrf_token']:

```
80 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 13
use Symfony\Component\Security\Csrf\CsrfToken;
... lines 15 - 17
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 20 - 53
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $token = new CsrfToken('authenticate', $credentials['csrf_token']);
... lines 57 - 61
    }
... lines 63 - 78
}
```

Because that's the key we used in getCredentials():

```
80 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 17
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 20 - 37
    public function getCredentials(Request $request)
    {
        $credentials = [
... lines 41 - 42
            'csrf_token' => $request->request->get('_csrf_token'),
        ];
... lines 45 - 51
    }
... lines 53 - 78
}
```

Then, if not $this->csrfTokenManager->isTokenValid($token), throw a special new InvalidCsrfTokenException():

80 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 9
use Symfony\Component\Security\Core\Exception\InvalidCsrfTokenException;
... lines 11 - 17
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 20 - 53
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $token = new CsrfToken('authenticate', $credentials['csrf_token']);
        if (!$this->csrfTokenManager->isTokenValid($token)) {
            throw new InvalidCsrfTokenException();
        }
... lines 60 - 61
    }
... lines 63 - 78
}
```

That's it! Let's first try logging in successfully. Refresh the login form to get the new hidden input. Use spacebar1@example.com, any password and... success!

Now, go back. Let's be shifty and mess with stuff. Inspect element on the form, find the token field, change it and queue your evil laugh. Mwahahaha. Log in! Ha! Yes! Invalid CSRF token! We rock!

Next: let's add a really convenient feature for users: a remember me checkbox!

Chapter 11: Adding Remember Me

Go back to the HTML form: it has *one* other field that we haven't talked about yet: the "remember me" checkbox:

37 lines | templates/security/login.html.twig

```
... lines 1 - 10
11 {% block body %}
12     <form class="form-signin" method="post">
... lines 13 - 26
27         <div class="checkbox mb-3">
28             <label>
29                 <input type="checkbox" value="remember-me"> Remember me
30             </label>
31         </div>
... lines 32 - 34
35     </form>
36 {% endblock %}
```

You could check & uncheck this to your heart's delight: that works great. But... checking it does... nothing. No worries: making this *actually* work is super easy - just two steps.

First, make sure that your checkbox has no value and that its name is _remember_me:

37 lines | templates/security/login.html.twig

```
... lines 1 - 10
11 {% block body %}
12     <form class="form-signin" method="post">
... lines 13 - 26
27         <div class="checkbox mb-3">
28             <label>
29                 <input type="checkbox" name="_remember_me"> Remember me
30             </label>
31         </div>
... lines 32 - 34
35     </form>
36 {% endblock %}
```

That's the magic name that Symfony will look for. Second, in security.yaml, under your firewall, add a new remember_me section. Add two *other* keys below this. The first is required: secret set to %kernel.secret%:

40 lines | config/packages/security.yaml

```
1 security:
... lines 2 - 8
9     firewalls:
... lines 10 - 12
13         main:
... lines 14 - 22
23             remember_me:
24                 secret:   '%kernel.secret%'
... lines 25 - 40
```

Second, lifetime set to 2592000, which is 30 days in seconds:

```
40 lines | config/packages/security.yaml
security:
... lines 2 - 8
    firewalls:
... lines 10 - 12
        main:
... lines 14 - 22
            remember_me:
                secret:   '%kernel.secret%'
                lifetime: 2592000 # 30 days in seconds
... lines 26 - 40
```

This option is... optional - it defaults to one year.

More about Parameters

As *soon* as you add this key, *if* the user checks a checkbox whose name is _remember_me, then a "remember me" cookie will be instantly set and used to log in the user if their session expires. This secret option is a cryptographic secret that's used to *sign* the data in that cookie. If you ever need a cryptographic secret, Symfony has a parameter called kernel.secret. Remember: anything surrounded by percent signs is a *parameter*. We never created this parameter directly: this is one of those built-in parameters that Symfony always makes available.

To see a list of *all* of the parameters, don't forget this handy command:

```
$ php bin/console debug:container --parameters
```

The most important ones start with kernel. Check out kernel.secret. Interesting, it's set to %env(APP_SECRET)%. This means that it's set to the *environment* variable APP_SECRET. That's one of the variables that's configured in our .env file.

Watch the Cookie Save Login!

Anyways, let's try this out! I'll re-open my inspector and refresh the login page. Go to Application, Cookies. Right now, there is only one: PHPSESSID.

This time, check the "remember me" box and log in. *Now* we *also* have a REMEMBERME cookie! And, check this out: I'm logged in as spacebar1@example.com. Delete the PHPSESSID - it currently starts with q3 - and refresh. Yes! We are *still* logged in!

A totally *new* session was created - with a new id. But even though this new session is empty, the remember me cookie causes us to stay logged in. You can even see that there's a new Token class called RememberMeToken. That's a low-level detail, but, it's a nice way to prove that this just worked.

Next - we've happily existed so far *without* storing or checking user passwords. Time to change that!

Chapter 12: Adding & Checking the User's Password

Until now, we've allowed users to login without *any* password. As *much* fun as it would be to deploy this to production... I think we should *probably* fix that. If you look at your User class, our users actually don't have a password field at *all*:

```
118 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 17
18     private $id;
... lines 19 - 22
23     private $email;
... lines 24 - 27
28     private $roles = [];
... lines 29 - 32
33     private $firstName;
... lines 34 - 116
117 }
```

When you originally use the make:user command, you *can* tell it to create this field for you. We told it to *not* do this... just to keep things simpler as we were learning. So, we'll do it now.

Adding the password Field to User

Find your terminal and run:

```
$ php bin/console make:entity
```

Update the User class to add a new field called password. Make it a string with length 255. It doesn't need to be *quite* that long, but that's fine. Can it be null? Say no: in our system, each user will *always* have a password.

And... done! It updated the User.php file, but it did *not* generate the normal getPassword() method because we already had that method before. We'll check that out in a minute.

Before that, run:

```
$ php bin/console make:migration
```

Move over and check out the Migrations directory. Open the new file and... yes! It looks perfect:
ALTER TABLE user ADD password:

```
29 lines | src/Migrations/Version20180831181732.php
... lines 1 - 10
11 final class Version20180831181732 extends AbstractMigration
12 {
13     public function up(Schema $schema) : void
14     {
... lines 15 - 17
18         $this->addSql('ALTER TABLE user ADD password VARCHAR(255) NOT NULL');
19     }
20
21     public function down(Schema $schema) : void
22     {
... lines 23 - 25
26         $this->addSql('ALTER TABLE user DROP password');
27     }
28 }
```

Close that, go back to your terminal, and migrate:

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

Awesome!

Updating the User Class

Go open the User class. Yep - we *now* have a password field:

```
130 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 34
35     /**
36      * @ORM\Column(type="string", length=255)
37      */
38     private $password;
... lines 39 - 128
129 }
```

And *all* the way at the bottom, a setPassword() method:

```
130 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 122
123     public function setPassword(string $password): self
124     {
125         $this->password = $password;
126
127         return $this;
128     }
129 }
```

Scroll up to find getPassword():

```
130 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 85
86     /**
87      * @see UserInterface
88      */
89     public function getPassword()
90     {
91         // not needed for apps that do not check user passwords
92     }
... lines 93 - 128
129 }
```

This already existed from back when our user had no password. Now that it does, return $this->password:

```
130 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 85
86     /**
87      * @see UserInterface
88      */
89     public function getPassword()
90     {
91         return $this->password;
92     }
... lines 93 - 128
129 }
```

Oh, and just to be clear, this password will *not* be a plain-text password. No, no, no! The string that we store in the database will always be properly salted & encoded. In fact, look at the method below this: getSalt():

```
130 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 93
94     /**
95      * @see UserInterface
96      */
97     public function getSalt()
98     {
99         // not needed for apps that do not check user passwords
100    }
... lines 101 - 128
129 }
```

In reality, there are *two* things you need to store in the database: the encoded password and the random *salt* value that was *used* to encode the password.

But, great news! Most modern encoders - including the one we will use - store the salt value as *part* of the encoded password string. In other words, we *only* need this one field. *And*, the getSalt() method can stay blank. I'll update the comment to explain why:

```
130 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 93
94     /**
95      * @see UserInterface
96      */
97     public function getSalt()
98     {
99         // not needed when using bcrypt or argon
100    }
... lines 101 - 128
129 }
```

I love doing no work!

Configuring the Encoder

Symfony will take care of *all* of this password encoding stuff *for* us. Nice! We just need to tell it which encoder algorithm to use. Go back to security.yaml. Add one new key: encoders. Below that, put the class name for your User class: App\Entity\User. And below *that*, set algorithm to bcrypt:

```
44 lines | config/packages/security.yaml
security:
    encoders:
        App\Entity\User:
            algorithm: bcrypt
... lines 5 - 44
```

Tip

In Symfony 4.3, you should use auto as your algorithm. This will use the *best* possible algorithm available on your system.

There are at least two good algorithm options here: bcrypt and argon2i. The argon2i encoder is actually a bit more secure. But, it's only available on PHP 7.2 or by installing an extension called Sodium.

If you and your production server have this available, awesome! Use it. If not, use bcrypt. Just know that once you start encoding passwords, *changing* algorithms in the future is a *pain*.

Oh, and for both encoders, there is one other option you can configure: cost. A higher cost makes passwords harder to crack... but will take more CPU. If security is really important for your app, check out this setting.

Anyways, thanks to this config, Symfony can *now* encrypt plaintext passwords *and* check whether a submitted password is valid.

Encoding Passwords

Open the UserFixture class because *first*, we need to populate the new password field in the database for our dummy users.

To encode a password - surprise! - Symfony has a service! Find your terminal and run our favorite:

```
$ php bin/console debug:autowiring
```

Search for "password". There it is! UserPasswordEncoderInterface. This service can encode *and* check passwords. Back in UserFixture, add a constructor with one argument: UserPasswordEncoderInterface. I'll re-type the "e" and hit tab to autocomplete and get the use statement I need on top. Call it $passwordEncoder:

```
36 lines | src/DataFixtures/UserFixture.php
... lines 1 - 6
7  use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
8
9  class UserFixture extends BaseFixture
10 {
... lines 11 - 12
13     public function __construct(UserPasswordEncoderInterface $passwordEncoder)
14     {
... line 15
16     }
... lines 17 - 34
35 }
```

Press Alt+Enter and select initialize fields to create that property and set it:

```
36 lines | src/DataFixtures/UserFixture.php
... lines 1 - 8
9  class UserFixture extends BaseFixture
10 {
11     private $passwordEncoder;
12
13     public function __construct(UserPasswordEncoderInterface $passwordEncoder)
14     {
15         $this->passwordEncoder = $passwordEncoder;
16     }
... lines 17 - 34
35 }
```

Now... the fun part: $user->setPassword(). But, instead of setting the plain password here - which would be *super* uncool... - say $this->passwordEncoder->encodePassword():

```
36 lines | src/DataFixtures/UserFixture.php
... lines 1 - 8
9  class UserFixture extends BaseFixture
10 {
... lines 11 - 17
18     protected function loadData(ObjectManager $manager)
19     {
20         $this->createMany(10, 'main_users', function($i) {
... lines 21 - 24
25             $user->setPassword($this->passwordEncoder->encodePassword(
... lines 26 - 27
28             ));
... lines 29 - 30
31         });
... lines 32 - 33
34     }
35 }
```

This needs two arguments: the $user object and the plain-text password we want to use. To make life easier for my brain, we'll use the same for everyone: engage:

```
36 lines | src/DataFixtures/UserFixture.php
... lines 1 - 8
class UserFixture extends BaseFixture
{
... lines 11 - 17
    protected function loadData(ObjectManager $manager)
    {
        $this->createMany(10, 'main_users', function($i) {
... lines 21 - 24
            $user->setPassword($this->passwordEncoder->encodePassword(
                $user,
                'engage'
            ));
... lines 29 - 30
        });
... lines 32 - 33
    }
}
```

That's it! The reason we need to pass the User object as the first argument is so that the password encoder knows which encoder algorithm to use. Let's try it: find your terminal and reload the fixtures:

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

You might notice that this is a bit slower now. By design, password encoding is CPU-intensive. Ok, check out the database!

```
$ php bin/console doctrine:query:sql 'SELECT * FROM user'
```

Awesome! Beautiful, encoded passwords. The bcrypt algorithm generated a unique salt for each user, which lives right inside this string.

Checking the Password

Ok, just *one* more step - and it's an easy one! We need to *check* the submitted password in LoginFormAuthenticator. *This* is the job of checkCredentials():

```
80 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 17
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 20 - 63
    public function checkCredentials($credentials, UserInterface $user)
    {
        // only needed if we need to check a password - we'll do that later!
        return true;
    }
... lines 69 - 78
}
```

We already know which service can do this. Add one more argument to your constructor: UserPasswordEncoderInterface $passwordEncoder. Hit Alt+Enter to initialize that field:

```
82 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 9
10 use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
... lines 11 - 18
19 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
20 {
... lines 21 - 23
24     private $passwordEncoder;
25
26     public function __construct(UserRepository $userRepository, RouterInterface $router, CsrfTokenManagerInterface $csrfTokenMan
27     {
... lines 28 - 30
31         $this->passwordEncoder = $passwordEncoder;
32     }
... lines 33 - 80
81 }
```

Then down in checkCredentials(), return $this->passwordEncoder->isPasswordValid() and pass this the User object and the raw, submitted password... which we're storing inside the password key of $credentials:

```
82 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 18
19 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
20 {
... lines 21 - 66
67     public function checkCredentials($credentials, UserInterface $user)
68     {
69         return $this->passwordEncoder->isPasswordValid($user, $credentials['password']);
70     }
... lines 71 - 80
81 }
```

And.. we're done! Time to celebrate by trying it! Move over, but this time put "foo" as a password. Login fails! Try engage. Yes!

Next: it's finally time to start talking about how we deny access to certain parts of our app. We'll start off that topic with a fun feature called access_control.

Chapter 13: access_control Authorization & Roles

Everything that we've done so far has been about authentication: how your user logs in. But now, our space-traveling users *can* log in! We're loading users from the database, checking their password and even protecting ourselves from the Borg Collective... with CSRF tokens.

So let's start to look at the second part of security: **authorization**. Authorization is all about deciding whether or not a user should have access to something. This is where, for example, you can require a user to log in before they see some page - or restrict some sections to admin users only.

There are two main ways to handle authorization: first, access_control and second, denying access in your controller. We'll see both, but I want to talk about access_control first, it's pretty cool.

access_control in security.yaml

At the bottom of your security.yaml file, you'll find a key called, well, access_control:

```
44 lines | config/packages/security.yaml
security:
... lines 2 - 38
    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }
```

Uncomment the first access control:

```
44 lines | config/packages/security.yaml
security:
... lines 2 - 40
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
... lines 43 - 44
```

The path is a regular expression. So, this access control says that any URL that starts with /admin should require a role called ROLE_ADMIN. We'll talk about roles in a minute.

Go to your terminal and run

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

Ah, yes, we *do* already have a few URLs that start with /admin, like /admin/comment. Well... let's see what happens when we try to go there!

Access denied! Cool! We get kicked out!

Roles!

Let's talk about how *roles* work in Symfony: it's simple and it's beautiful. Down on the web debug toolbar, click on the user icon. Cool: we're logged in as spacebar1@example.com and we have one role: ROLE_USER. Here's the idea: when a user logs in, you give them whatever "roles" you want - like ROLE_USER. Then, you run around your code and make different URLs require different roles. Because our user does *not* have ROLE_ADMIN, we are denied access.

But... why does our user have ROLE_USER? I don't remember doing *anything* with roles during the login code. Open the

User class. When we ran the make:user command, one of the methods that it generated was getRoles():

```
130 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 66
67     /**
68      * @see UserInterface
69      */
70     public function getRoles(): array
71     {
72         $roles = $this->roles;
73         // guarantee every user at least has ROLE_USER
74         $roles[] = 'ROLE_USER';
75
76         return array_unique($roles);
77     }
... lines 78 - 128
129 }
```

Look at it carefully: it reads a roles property, which is an array that's stored in the database:

```
130 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 24
25     /**
26      * @ORM\Column(type="json")
27      */
28     private $roles = [];
... lines 29 - 128
129 }
```

Right now, this property is empty for *every* user in the database: we have *not* set this to any value in the fixtures.

But, inside getRoles(), there's a little extra logic that guarantees that *every* user *at least* has this one role: ROLE_USER:

```
130 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 69
70     public function getRoles(): array
71     {
... line 72
73         // guarantee every user at least has ROLE_USER
74         $roles[] = 'ROLE_USER';
... lines 75 - 76
77     }
... lines 78 - 128
129 }
```

This is nice because we *now* know that, *if* you are logged in, you definitely have this *one* role. Also... you need to make sure that getRoles() always returns at least *one* role... otherwise weird stuff happens: the user becomes an undead zombie that is "sort of" logged in.

To prove that this roles system works like we expect, change ROLE_ADMIN to ROLE_USER in the access control:

```
security:
    # ...
    access_control:
        - { path: ^/admin, roles: ROLE_USER }
```

Then, click *back* to the admin page and... access granted!

Change that back to ROLE_ADMIN.

Only One access_control Matches per Page

As you can see in the examples down here, you're allowed to have as *many* access_control lines as you want: each has their own regular expression path. But, there is one *super* important thing to understand. Access controls work like *routes*: Symfony checks them one-by-one from top to bottom. And as soon as it finds *one* access control that matches the URL, it uses that and stops. Yep, a maximum of *one* access control is used on each page load.

Actually... this fact allows you to do some cool things if you want *most* of your pages to require login. We'll talk about that later.

Now that we can deny access... something interesting happens if you try to access a protected page as an *anonymous* user. Let's see that next.

Chapter 14: Target Path: Redirecting an Anonymous User

After changing the access_control back to ROLE_ADMIN:

```
44 lines | config/packages/security.yaml
security:
... lines 2 - 40
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
... lines 43 - 44
```

If we try to access /admin/comment again, we see that same "Access Denied" page: 403 forbidden.

Customizing the Error Page

Like with *all* the big, beautiful error pages, these are only shown to us, the developers. On production, by default, your users will see a boring, generic error page that *truly* looks like it was designed by a developer.

But, you can - and *should* - customize this. We won't go through it now, but if you Google for "Symfony error pages", you can find out how. The cool thing is that you can have a different error page per *status* code. So, a custom 404 not found page and a *different* custom 403 "Access Denied" page - with, ya know, like a mean looking alien or something to tell you to *stop* trying to hack the site.

Redirecting Anonymous Users: Entry Point

Anyways, I have a question for you. First, log out. Now that we are anonymous: what do you think will happen if we try to go to /admin/comment? Will we see that same Access Denied page? After all, we *are* anonymous... so we definitely do *not* have ROLE_ADMIN.

Well... let's find out! No! We are redirected to the login page! That's... awesome! If you think about it, that's the *exact* behavior we want: if we're not logged in and we try to access a page that requires me to be logged in, we should *totally* be sent to the login form so that we *can* login.

The logic behind this actually comes from our authenticator. Or, really, from the parent AbstractFormLoginAuthenticator. It has a method - called start() - that decides what to do when an anonymous user tries to access something. It's called an entry point, and we'll learn more about this later when we talk about API authentication.

Redirecting Back on Success

But for now, great! Our system already behaves like we want. But now... check this out. Log back in with spacebar1@example.com, password engage. When I hit enter, where do you think we'll be redirected to? The homepage? /admin/comment? Let's find out.

We're sent to the homepage! Perfect, right? No, not perfect! I originally tried to go to /admin/comment. So, after logging in, to have a great user experience, we should be redirected back *there*.

The reason that we're sent to the homepage is because of *our* code in LoginFormAuthenticator. onAuthenticationSuccess() *always* sends the user to the homepage, no matter what:

```
82 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 18
19 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
20 {
... lines 21 - 71
72     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
73     {
74         return new RedirectResponse($this->router->generate('app_homepage'));
75     }
... lines 76 - 80
81 }
```

Hmm: how could we update this method to send the user back to the *previous* page instead?

Symfony can help with this. Find your browser, log out, and then go back to /admin/comment. *Whenever* you try to access a URL as an anonymous user, *before* Symfony redirects to the login page, it saves this URL - /admin/comment - into the session on a special key. So, if we can *read* that value from the session inside onAuthenticationSuccess(), we can redirect the user back there!

To do this, at the top of your authenticator, use a *trait* TargetPathTrait:

```
89 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 17
18 use Symfony\Component\Security\Http\Util\TargetPathTrait;
19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
22     use TargetPathTrait;
... lines 23 - 87
88 }
```

Then, down in onAuthenticationSuccess(), add if $targetPath = $this->getTargetPath(). *This* method comes from our handy trait! It needs the session - $request->getSession() - and the "provider key", which is actually an argument to this method:

```
89 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
... lines 22 - 74
75     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
76     {
77         if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
... line 78
79         }
... lines 80 - 81
82     }
... lines 83 - 87
88 }
```

The provider key is just the *name* of your firewall... but that's not too important here.

Oh, and, yea, the if statement might look funny to you: I'm assigning the $targetPath variable and *then* checking to see if it's empty or not. If it's *not* empty, if there *is* something stored in the session, return new RedirectResponse($targetPath):

89 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
... lines 22 - 74
75     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
76     {
77         if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
78             return new RedirectResponse($targetPath);
79         }
... lines 80 - 81
82     }
... lines 83 - 87
88 }
```

That's it! If there is *no* target path in the session - which can happen if the user went to the login page directly - fallback to the homepage:

89 lines | src/Security/LoginFormAuthenticator.php

```
... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
... lines 22 - 74
75     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
76     {
77         if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
78             return new RedirectResponse($targetPath);
79         }
80
81         return new RedirectResponse($this->router->generate('app_homepage'));
82     }
... lines 83 - 87
88 }
```

Let's try it! Log back in... with password engage. Yea! Got it! I know, it feels weird to celebrate when you see an access denied page. But we *expected* that part. The important thing is that we *were* redirected back to the page we originally tried to access. That's *excellent* UX.

Next - as nice as access controls are, we need more *granular* control. Let's learn how to control user access from inside the controller.

Chapter 15: Deny Access in the Controller

There are two main places where you can deny access. The first we just learned about: access_control in security.yaml:

44 lines | config/packages/security.yaml

```
security:
... lines 2 - 40
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
... lines 43 - 44
```

It's simple - just a regular expression and a role. It's *the* best way to protect entire *areas* of your site - like *everything* under /admin with ROLE_ADMIN.

I *do* use access controls for things like that. But, most of the time, I prefer to control access at a more granular level. Open CommentAdminController. Most of the time, I deny access *right* inside the controller.

To test this out - let's comment-out our access control:

44 lines | config/packages/security.yaml

```
security:
... lines 2 - 40
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
... lines 43 - 44
```

Back in CommentAdminController, how can we deny access here? Simple: $this->denyAccessUnlessGranted() and pass this a role: ROLE_ADMIN:

35 lines | src/Controller/CommentAdminController.php

```
... lines 1 - 10
class CommentAdminController extends Controller
{
... lines 13 - 15
    public function index(CommentRepository $repository, Request $request, PaginatorInterface $paginator)
    {
        $this->denyAccessUnlessGranted('ROLE_ADMIN');
... lines 19 - 32
    }
}
```

That's it. Move over and refresh!

Nice! Try changing it to ROLE_USER:

```
35 lines | src/Controller/CommentAdminController.php
... lines 1 - 10
11 class CommentAdminController extends Controller
12 {
... lines 13 - 15
16     public function index(CommentRepository $repository, Request $request, PaginatorInterface $paginator)
17     {
18         $this->denyAccessUnlessGranted('ROLE_USER');
... lines 19 - 32
33     }
34 }
```

Access granted! I love it!

IsGranted Annotation

But wait, there's more! As simple as this is, I like to use annotations. Check this out: delete the denyAccessUnlessGranted() code. Instead, above the method, add @IsGranted() to use an annotation that comes from SensioFrameworkExtraBundle: a bundle that we installed a long time ago via composer require annotations. In double quotes, pass ROLE_ADMIN:

```
35 lines | src/Controller/CommentAdminController.php
... lines 1 - 6
7 use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
... lines 8 - 11
12 class CommentAdminController extends Controller
13 {
14     /**
... line 15
16      * @IsGranted("ROLE_ADMIN")
17      */
18     public function index(CommentRepository $repository, Request $request, PaginatorInterface $paginator)
19     {
... lines 20 - 32
33     }
34 }
```

Nice! Try it: refresh!

> Access Denied by controller annotation

Pretty sweet. I know not everyone will *love* using annotations for this. So, if you don't love it, use the PHP version. No problem.

Protecting an Entire Controller Class

Oh, but the annotation *does* have one superpower. In addition to putting @IsGranted above a controller method, you can *also* put it above the controller *class*. Above CommmentAdminController, add @IsGranted("ROLE_ADMIN"):

37 lines | src/Controller/CommentAdminController.php

```
... lines 1 - 6
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
... lines 8 - 11
/**
 * @IsGranted("ROLE_ADMIN")
 */
class CommentAdminController extends Controller
{
... lines 17 - 35
}
```

Now, *every* method inside of this controller... which is only one right now, will require this role. When you refresh... yep! Same error. That is an awesome way to deny access.

We know how to make sure a user has a role. But, how can we simply make sure a user is logged in, regardless of roles? Let's find out next - *and* - create our first admin users.

Chapter 16: Dynamic Roles

I want to create something new: a *new* user account page. Find your terminal and run:

```
$ php bin/console make:controller
```

Create a new AccountController. Open that up:

20 lines | src/Controller/AccountController.php

```
... lines 1 - 2
3  namespace App\Controller;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6  use Symfony\Component\Routing\Annotation\Route;
7
8  class AccountController extends AbstractController
9  {
10     /**
11      * @Route("/account", name="account")
12      */
13     public function index()
14     {
15         return $this->render('account/index.html.twig', [
16             'controller_name' => 'AccountController',
17         ]);
18     }
19 }
```

Perfect! A new /account page, which we can see instantly if we go to that URL.

Change the route name to app_account to be consistent with our other code:

20 lines | src/Controller/AccountController.php

```
... lines 1 - 7
8  class AccountController extends AbstractController
9  {
10     /**
11      * @Route("/account", name="app_account")
12      */
13     public function index()
14     {
... lines 15 - 17
18     }
19 }
```

And, I'm not going to pass any variables to the template for now:

20 lines | src/Controller/AccountController.php

```
... lines 1 - 7
class AccountController extends AbstractController
{
    /**
     * @Route("/account", name="app_account")
     */
    public function index()
    {
        return $this->render('account/index.html.twig', [

        ]);
    }
}
```

Open that: templates/account/index.html.twig. Let's customize this just a bit: Manage Account and an h1: Manage your Account:

8 lines | templates/account/index.html.twig

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

{% block title %}Manage Account!{% endblock %}

{% block body %}
    <h1>Manage Your Account</h1>
{% endblock %}
```

That's pretty boring... but it should be enough for us to get into trouble!

Check if the User Is Logged In

Ok: I *only* want this page to be accessible by users who are logged in. Log out and then go back to /account. Obviously, right now, *anybody* can access this. Hmm: we *do* already know how to require the user to have a specific *role* to access something - like in CommentAdminController where we require ROLE_ADMIN:

37 lines | src/Controller/CommentAdminController.php

```
... lines 1 - 11
/**
 * @IsGranted("ROLE_ADMIN")
 */
class CommentAdminController extends Controller
{
... lines 17 - 35
}
```

But... how can we make sure that the user is simply... logged in?

There are actually *two* different ways. I'll tell you about *one* of those ways later. But right now, I want to tell you about the easier of the two ways: just check for ROLE_USER.

Above the AccountController class - so that it applies to any future methods - add @IsGranted("ROLE_USER"):

```
24 lines | src/Controller/AccountController.php
... lines 1 - 4
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
... lines 6 - 8
/**
 * @IsGranted("ROLE_USER")
 */
class AccountController extends AbstractController
{
... lines 14 - 22
}
```

So... *why* is this a valid way to check that the user is simply logged in? Because, remember! In User, our getRoles() method is written so that *every* user always has at least this role:

```
130 lines | src/Entity/User.php
... lines 1 - 10
class User implements UserInterface
{
... lines 13 - 66
    /**
     * @see UserInterface
     */
    public function getRoles(): array
    {
        $roles = $this->roles;
        // guarantee every user at least has ROLE_USER
        $roles[] = 'ROLE_USER';

        return array_unique($roles);
    }
... lines 78 - 128
}
```

If you are logged in, you *definitely* have ROLE_USER.

Refresh the page now: it bumps us to the login page. Log in with password engage and... nice! We're sent back over to /account. Smooth.

Adding Admin Users

At this point, *even* though we're requiring ROLE_ADMIN in CommentAdminController, we... well... don't actually have any admin users! Yep, *nobody* can access this page because *nobody* has ROLE_ADMIN!

To make this page... um... actually usable, open src/DataFixtures/UserFixture.php. In addition to these "normal" users, let's *also* create some admin users. Copy the whole createMany() block and paste below. Give this set of users a different "group name" - admin_users:

```
50 lines | src/DataFixtures/UserFixture.php
... lines 1 - 8
9  class UserFixture extends BaseFixture
10 {
... lines 11 - 17
18     protected function loadData(ObjectManager $manager)
19     {
20         $this->createMany(10, 'main_users', function($i) {
... lines 21 - 30
31         });
32
33         $this->createMany(3, 'admin_users', function($i) {
... lines 34 - 44
45         });
46
47         $manager->flush();
48     }
49 }
```

Remember: this key is *not* important right now. But we can use it later in *other* fixture classes if we wanted to "fetch" these admin users and relate them to different objects. We'll see that later.

Let's create three admin users. For the email, how about admin%d@thespacebar.com:

```
50 lines | src/DataFixtures/UserFixture.php
... lines 1 - 8
9  class UserFixture extends BaseFixture
10 {
... lines 11 - 17
18     protected function loadData(ObjectManager $manager)
19     {
... lines 20 - 32
33         $this->createMany(3, 'admin_users', function($i) {
34             $user = new User();
35             $user->setEmail(sprintf('admin%d@thespacebar.com', $i));
... lines 36 - 44
45         });
... lines 46 - 47
48     }
49 }
```

The first name is fine and keep the password so that I don't get completely confused. But *now* add $user->setRoles() with ROLE_ADMIN:

```
50 lines | src/DataFixtures/UserFixture.php
... lines 1 - 8
class UserFixture extends BaseFixture
{
... lines 11 - 17
    protected function loadData(ObjectManager $manager)
    {
... lines 20 - 32
        $this->createMany(3, 'admin_users', function($i) {
            $user = new User();
            $user->setEmail(sprintf('admin%d@thespacebar.com', $i));
            $user->setFirstName($this->faker->firstName);
            $user->setRoles(['ROLE_ADMIN']);

            $user->setPassword($this->passwordEncoder->encodePassword(
                $user,
                'engage'
            ));

            return $user;
        });
... lines 46 - 47
    }
}
```

Notice that I do not *also* need to add ROLE_USER: the getRoles() method will make sure that's returned *even* if it's not stored in the database:

```
130 lines | src/Entity/User.php
... lines 1 - 10
class User implements UserInterface
{
... lines 13 - 66
    /**
     * @see UserInterface
     */
    public function getRoles(): array
    {
        $roles = $this->roles;
        // guarantee every user at least has ROLE_USER
        $roles[] = 'ROLE_USER';

        return array_unique($roles);
    }
... lines 78 - 128
}
```

Let's reload those fixtures!

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

When that finishes, move over and go back to /login. Log in as one of the new users: admin2@thespacebar.com, password engage. Then, try /admin/comment. Access granted! Woohoo! And we, of course, *also* have access to /account because our user has both ROLE_ADMIN *and* ROLE_USER.

Checking for Roles in Twig

Oh, and now that we know how to check if the user is logged in, let's fix our user drop-down: we should *not* show the login link once we're logged in.

In PhpStorm, open templates/base.html.twig and scroll down a bit. Earlier, when we added the login link, we commented out our big user drop-down:

```
74 lines | templates/base.html.twig
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
   ... lines 36 - 38
39                     {#
40                     <li class="nav-item dropdown" style="margin-right: 75px;">
41                         <a class="nav-link dropdown-toggle" href="http://example.com" id="navbarDropdownMenuLink" data-toggle="dropdown
42                           <img class="nav-profile-img rounded-circle" src="{{ asset('images/astronaut-profile.png') }}">
43                         </a>
44                         <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
45                             <a class="dropdown-item" href="#">Profile</a>
46                             <a class="dropdown-item" href="#">Create Post</a>
47                             <a class="dropdown-item" href="#">Logout</a>
48                         </div>
49                     </li>
50                     #}
51                 </ul>
52             </div>
53         </nav>
   ... lines 54 - 71
72     </body>
73 </html>
```

Now, we can be a bit smarter. Copy that entire section: we *will* show it when the user is logged in.

Oh, but how can we check if the user has a role from inside Twig? With: is_granted(): if is_granted('ROLE_USER'), else - I'll indent my logout link - and endif:

75 lines | templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
36                     {% if is_granted('ROLE_USER') %}
   ... lines 37 - 46
47                     {% else %}
48                         <li class="nav-item">
49                             <a style="color: #fff;" class="nav-link" href="{{ path('app_login') }}">Login</a>
50                         </li>
51                     {% endif %}
52                 </ul>
53             </div>
54         </nav>
   ... lines 55 - 72
73     </body>
74 </html>
```

Inside the if, paste the drop-down code:

75 lines | templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
36                     {% if is_granted('ROLE_USER') %}
37                         <li class="nav-item dropdown" style="margin-right: 75px;">
38                             <a class="nav-link dropdown-toggle" href="http://example.com" id="navbarDropdownMenuLink" data-toggle="dropdo
39                               <img class="nav-profile-img rounded-circle" src="{{ asset('images/astronaut-profile.png') }}">
40                             </a>
41                             <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
42                                 <a class="dropdown-item" href="#">Profile</a>
43                                 <a class="dropdown-item" href="#">Create Post</a>
44                                 <a class="dropdown-item" href="#">Logout</a>
45                             </div>
46                         </li>
47                     {% else %}
48                         <li class="nav-item">
49                             <a style="color: #fff;" class="nav-link" href="{{ path('app_login') }}">Login</a>
50                         </li>
51                     {% endif %}
52                 </ul>
53             </div>
54         </nav>
   ... lines 55 - 72
73     </body>
74 </html>
```

Ah! Let's go see it! Refresh! Our user drop-down is back! Oh, except all of these links go... nowhere. We can fix that!

For profile, that route is app_account: path('app_account'):

75 lines | templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
36                     {% if is_granted('ROLE_USER') %}
37                         <li class="nav-item dropdown" style="margin-right: 75px;">
   ... lines 38 - 40
41                             <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
42                                 <a class="dropdown-item" href="{{ path('app_account') }}">Profile</a>
   ... lines 43 - 44
45                             </div>
46                         </li>
   ... lines 47 - 50
51                     {% endif %}
52                 </ul>
53             </div>
54         </nav>
   ... lines 55 - 72
73     </body>
74 </html>
```

For logout, that's path('app_logout'):

75 lines | templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
36                     {% if is_granted('ROLE_USER') %}
37                         <li class="nav-item dropdown" style="margin-right: 75px;">
   ... lines 38 - 40
41                             <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
   ... lines 42 - 43
44                                 <a class="dropdown-item" href="{{ path('app_logout') }}">Logout</a>
45                             </div>
46                         </li>
   ... lines 47 - 50
51                     {% endif %}
52                 </ul>
53             </div>
54         </nav>
   ... lines 55 - 72
73     </body>
74 </html>
```

And, for "Create Post", we haven't built that yet. But, there *is* a controller called ArticleAdminController and we have at least *started* this. Give this route a name="admin_article_new":

31 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 14
15 class ArticleAdminController extends AbstractController
16 {
17     /**
18      * @Route("/admin/article/new", name="admin_article_new")
19      */
20     public function new(EntityManagerInterface $em)
21     {
... lines 22 - 28
29     }
30 }
```

Tip

Oh! And don't forget to require ROLE_ADMIN on the controller!

31 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 6
7  use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
... lines 8 - 11
12 /**
13  * @IsGranted("ROLE_ADMIN")
14  */
15 class ArticleAdminController extends AbstractController
16 {
... lines 17 - 29
30 }
```

We'll link here, even though it's not done:

77 lines | templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
36                     {% if is_granted('ROLE_USER') %}
37                         <li class="nav-item dropdown" style="margin-right: 75px;">
   ... lines 38 - 40
41                             <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
   ... lines 42 - 43
44                                 <a class="dropdown-item" href="{{ path('admin_article_new') }}">Create Post</a>
   ... lines 45 - 46
47                             </div>
48                         </li>
   ... lines 49 - 52
53                     {% endif %}
54                 </ul>
55             </div>
56         </nav>
   ... lines 57 - 74
75     </body>
76 </html>
```

Oh, but this link is only for admin users. So, surround this with is_granted("ROLE_ADMIN"):

77 lines | templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
36                     {% if is_granted('ROLE_USER') %}
37                         <li class="nav-item dropdown" style="margin-right: 75px;">
   ... lines 38 - 40
41                             <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
   ... line 42
43                                 {% if is_granted('ROLE_ADMIN') %}
44                                     <a class="dropdown-item" href="{{ path('admin_article_new') }}">Create Post</a>
45                                 {% endif %}
   ... line 46
47                             </div>
48                         </li>
   ... lines 49 - 52
53                     {% endif %}
54                 </ul>
55             </div>
56         </nav>
   ... lines 57 - 74
75     </body>
76 </html>
```

Nice! Let's make sure we didn't mess up - refresh! Woohoo! Because we *are* logged in as an admin user, we see the user drop-down *and* the Create Post link.

Next: we need to talk about a few unique roles that start with IS_AUTHENTICATED and how these can be used in access_control to easily require login for *every* page on your site.

Chapter 17: IS_AUTHENTICATED_ & Protecting All URLs

I mentioned earlier that there are *two* ways to check whether or not the user is simply logged in. The first is by checking ROLE_USER:

77 lines | templates/base.html.twig

```
1   <!doctype html>
2   <html lang="en">
    ... lines 3 - 15
16      <body>
17          <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
    ... lines 18 - 21
22              <div class="collapse navbar-collapse" id="navbarNavDropdown">
    ... lines 23 - 34
35                  <ul class="navbar-nav ml-auto">
36                      {% if is_granted('ROLE_USER') %}
    ... lines 37 - 52
53                      {% endif %}
54                  </ul>
55              </div>
56          </nav>
    ... lines 57 - 74
75      </body>
76  </html>
```

I like this one because it's simple. It works because of how our getRoles() method is written:

130 lines | src/Entity/User.php

```
    ... lines 1 - 10
11  class User implements UserInterface
12  {
    ... lines 13 - 66
67      /**
68       * @see UserInterface
69       */
70      public function getRoles(): array
71      {
72          $roles = $this->roles;
73          // guarantee every user at least has ROLE_USER
74          $roles[] = 'ROLE_USER';
75
76          return array_unique($roles);
77      }
    ... lines 78 - 128
129 }
```

The *only* reason I'm even going to *mention* the *second* way is because I want you to know what it is if you see it, *and*, it leads us towards a few other interesting things.

IS_AUTHENTICATED_FULLY

Let's *play* a little bit in security.yaml. Under access_control add a new entry with path ^/account. Yes, this will be a *totally* redundant access control because we're already requiring ROLE_USER from inside the controller:

```
24 lines | src/Controller/AccountController.php
... lines 1 - 8
/**
 * @IsGranted("ROLE_USER")
 */
class AccountController extends AbstractController
{
... lines 14 - 22
}
```

Just pretend that we don't have this controller code for a minute.

On your access_control, if you wanted to require the user to be logged in, you could use roles: ROLE_USER *or* IS_AUTHENTICATED_FULLY:

```
45 lines | config/packages/security.yaml
security:
... lines 2 - 40
    access_control:
        - { path: ^/account, roles: IS_AUTHENTICATED_FULLY }
... lines 43 - 45
```

OoooOOOoo.

Well, it's not really that fancy: it's just a special string that *simply* checks if the user is logged in or not. In our system, it's 100% identical to ROLE_USER.

Move over, go back to /account and... yep! Access is *still* granted.

Web Debug Toolbar & Access Control Checks

Oh, and I want to show you something cool! Click the little security icon on the web debug toolbar. This has some *pretty* sweet stuff in it. In addition to saying who you're logged in as and your roles, it also has a table down here with some lower-level info. But what I *really* want to show you is *all* the way at the bottom. Yes! The access decision log. This records *every* time that we checked whether or not the user had access to something on this page. The first check is for IS_AUTHENTICATED_FULLY from access_control. Granted! Then, two ROLE_USER checks and one ROLE_ADMIN check.

One of those ROLE_USER checks is from AccountController:

```
24 lines | src/Controller/AccountController.php
... lines 1 - 8
/**
 * @IsGranted("ROLE_USER")
 */
class AccountController extends AbstractController
{
... lines 14 - 22
}
```

And the other comes from is_granted() in the template. The ROLE_ADMIN check also lives here:

```
77 lines | templates/base.html.twig
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
36                     {% if is_granted('ROLE_USER') %}
37                         <li class="nav-item dropdown" style="margin-right: 75px;">
   ... lines 38 - 40
41                             <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
   ... line 42
43                                 {% if is_granted('ROLE_ADMIN') %}
   ... line 44
45                                 {% endif %}
   ... line 46
47                             </div>
48                         </li>
   ... lines 49 - 52
53                     {% endif %}
54                 </ul>
55             </div>
56         </nav>
   ... lines 57 - 74
75     </body>
76 </html>
```

So, this is just a nice way to debug all the security checks happening on your page.

Requiring Login on Every Page

Anyways, we now know IS_AUTHENTICATED_FULLY is a way to check if the user is logged in. Though... because of the way our app is written, checking ROLE_USER does the same thing and... it's shorter to write.

But! This *does* touch on another interesting topic. This is a news site, so most of the pages will be accessible to anonymous users. We'll require login on just the pages that need it. Not all sites are like this, however. On *some* sites, you want to do the opposite: you want to require authentication for *every* page, or at least, *almost* every page. In those cases, a better strategy is to require login on *all* pages and then *allow* anonymous access on just a few pages.

We can do this by being clever with access_control. Try this: change the path to just ^/:

```
46 lines | config/packages/security.yaml
security:
... lines 2 - 40
    access_control:
        # if you wanted to force EVERY URL to be protected
        - { path: ^/, roles: IS_AUTHENTICATED_FULLY }
... lines 44 - 46
```

Because this is a regular expression, it will match *every* URL and so *every* page now requires login.

If we refresh, we still have access. But now, log out!

Allowing the Login Page: IS_AUTHENTICATED_ANONYMOUSLY

Whoa! The page is broken! Like, *crazy* broken! localhost redirected too many times!? Yep, our security system is *too* awesome. Because we're now anonymous, when we try to access any page, we're redirected to /login. But guess what? /login requires authentication too! So what does Symfony do? It redirects us to /login!

We made security *so* tight that anonymous users can't even get to the login page! Here's the fix: add a new access_control - *above* the one for all URLs with path: ^/login. You can add a $ on the end to match only this URL exactly, not also /login/foo. Your call. For roles, use a *second* special string: IS_AUTHENTICATED_ANONYMOUSLY:

48 lines | config/packages/security.yaml

```
security:
... lines 2 - 40
    access_control:
        # but, definitely allow /login to be accessible anonymously
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        # if you wanted to force EVERY URL to be protected
        - { path: ^/, roles: IS_AUTHENTICATED_FULLY }
... lines 46 - 48
```

This one is *weird*. Who has IS_AUTHENTICATED_ANONYMOUSLY? Everyone! If you're anonymous, you have it. If you're logged in, you have it too! So, *why* would we *ever* want to use a role that *everyone* has? Well, go refresh.

Because it fixes our problem! Remember: Symfony goes down each access_control one-by-one. As *soon* as it finds *one* that matches, it uses that *one* and stops. So when we go to /login, *only* the first access control is used and access is granted. Every *other* page will still require login. Booya!

IS_AUTHENTICATED_REMEMBERED

We've now learned *two* special "strings" that can be used in place of the normal roles: IS_AUTHENTICATED_FULLY and IS_AUTHENTICATED_ANONYMOUSLY. But, there is *one* more. Change "fully" to IS_AUTHENTICATED_REMEMBERED:

48 lines | config/packages/security.yaml

```
security:
... lines 2 - 40
    access_control:
        # but, definitely allow /login to be accessible anonymously
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        # if you wanted to force EVERY URL to be protected
        - { path: ^/, roles: IS_AUTHENTICATED_REMEMBERED }
... lines 46 - 48
```

Go back to your site and log in. Because we *just* logged in, we have all three special strings: IS_AUTHENTICATED_FULLY, IS_AUTHENTICATED_REMEMBERED and, of course, IS_AUTHENTICATED_ANONYMOUSLY.

But now, imagine that you're using the "remember me" functionality. You close your browser, re-open it, and are *still* authenticated, but only thanks to the remember me cookie. *Now*, you would *still* have IS_AUTHENTICATED_REMEMBERED, but you would *not* have IS_AUTHENTICATED_FULLY. Fully means that you have authenticated during *this* session.

This allows you to do something really neat. If you use the remember me functionality you should protect all pages that require login with IS_AUTHENTICATED_REMEMBERED. This says that you don't care whether the user just logged in during this session or if they are logged in via the remember me cookie. *Then* you can protect more sensitive pages - like the change password page - with IS_AUTHENTICATED_FULLY:

```
50 lines | config/packages/security.yaml
security:
... lines 2 - 40
    access_control:
        # but, definitely allow /login to be accessible anonymously
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        # require the user to fully login to change password
        - { path: ^/change-password, roles: IS_AUTHENTICATED_FULLY }
        # if you wanted to force EVERY URL to be protected
        - { path: ^/, roles: IS_AUTHENTICATED_REMEMBERED }
... lines 48 - 50
```

If a user tries to access that page, but is *only* authenticated with the remember me cookie, Symfony will redirect them to the login page so that they can become "fully" authenticated. Nice, right?

By the way, I'm showing you all of these examples for the IS_AUTHENTICATED strings inside access_control. But, you absolutely can use these in your controller or inside Twig.

Ok, because our site will be mostly public, I'll comment-out these examples:

```
51 lines | config/packages/security.yaml
security:
... lines 2 - 40
    access_control:
        # but, definitely allow /login to be accessible anonymously
        #- { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        # require the user to fully login to change password
        #- { path: ^/change-password, roles: IS_AUTHENTICATED_FULLY }
        # if you wanted to force EVERY URL to be protected
        #- { path: ^/, roles: IS_AUTHENTICATED_REMEMBERED }
... lines 48 - 51
```

Next, let's learn how to find out *who* is logged in by fetching their User object.

Chapter 18: Fetch the User Object

Once you have your authentication system step, pff, life is easy! On a day-to-day basis, you'll spend most of your time in a controller where... well, there's really only *two* things you can do related to security. One, deny access, like, based on a role:

24 lines | src/Controller/AccountController.php

```
... lines 1 - 8
/**
 * @IsGranted("ROLE_USER")
 */
class AccountController extends AbstractController
{
... lines 14 - 22
}
```

Or two, figure out *who* is logged in.

That's *exactly* what we need to do in AccountController so that we can start printing out details about the user's account. So... how *can* we find out who is logged in? With $this->getUser():

25 lines | src/Controller/AccountController.php

```
... lines 1 - 11
class AccountController extends AbstractController
{
... lines 14 - 16
    public function index()
    {
        dd($this->getUser()->getFirstName());
... lines 20 - 22
    }
}
```

Using the User Object

Go back to your browser and head to /account. Nice! This gives us the User entity object! That's *awesome* because we can do all kinds of cool stuff with it. For example, let's see if we can log the email address of who is logged in.

Add a LoggerInterface $logger argument:

26 lines | src/Controller/AccountController.php

```
... lines 1 - 4
use Psr\Log\LoggerInterface;
... lines 6 - 12
class AccountController extends AbstractController
{
... lines 15 - 17
    public function index(LoggerInterface $logger)
    {
... lines 20 - 23
    }
}
```

Then say $logger->debug():

Checking account page for

And then $this->getUser(). Because we know this is *our* User entity, we know that we can call, getEmail() on it. Do that: ->getEmail():

26 lines | src/Controller/AccountController.php

```
... lines 1 - 12
13 class AccountController extends AbstractController
14 {
... lines 15 - 17
18     public function index(LoggerInterface $logger)
19     {
20         $logger->debug('Checking account page for '.$this->getUser()->getEmail());
... lines 21 - 23
24     }
25 }
```

Cool! Move over and refresh. No errors. Click anywhere down on the web debug toolbar to get into the profiler. Go to the logs tab, click "Debug" and... down a bit, there it is!

Checking account page for spacebar5@example.com.

Base Controller: Auto-complete $this->getUser()

But, hmm, something is bothering me: I do *not* get any auto-complete on this getEmail() method. Why not? Hold Command or Control and click the getUser() method. Ah: it's simple: Symfony doesn't know what our User class is. So, its PhpDoc can't really tell PhpStorm what this method will return.

To get around this, I like to create my own BaseController class. In the Controller/ directory, create a new PHP class called BaseController. I'll make it abstract because this is not going to be a real controller - just a helpful base class. Make it extend the normal AbstractController that we've been using in our existing controllers:

14 lines | src/Controller/BaseController.php

```
... lines 1 - 2
3 namespace App\Controller;
... lines 4 - 5
6 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
7
8 abstract class BaseController extends AbstractController
9 {
... lines 10 - 13
14 }
```

Tip

A simpler solution (and one that avoids a deprecation warning) is to advertise to your IDE that getUser() returns a User (or null) with some PHPDoc:

```
/**
 * @method User|null getUser()
 */
class BaseController extends AbstractController
{
}
```

Then, I'll go to the "Code"->"Generate" menu - or Command+N on a Mac, click "Override Methods" and override getUser(). We're not *actually* going to override how this method works. Just return parent::getUser(). But, add a return type User - *our* User class:

```
14 lines | src/Controller/BaseController.php
... lines 1 - 4
5  use App\Entity\User;
... lines 6 - 7
8  abstract class BaseController extends AbstractController
9  {
10     protected function getUser(): User
11     {
12         return parent::getUser();
13     }
14 }
```

From now on, instead of extending AbstractController, we should extend BaseController:

```
25 lines | src/Controller/AccountController.php
... lines 1 - 11
12 class AccountController extends BaseController
13 {
... lines 14 - 23
24 }
```

And *this* will give us the proper auto-completion on getUser():

```
25 lines | src/Controller/AccountController.php
... lines 1 - 11
12 class AccountController extends BaseController
13 {
... lines 14 - 16
17     public function index(LoggerInterface $logger)
18     {
19         $logger->debug('Checking account page for '.$this->getUser()->getEmail());
... lines 20 - 22
23     }
24 }
```

I also like to use my BaseController to add other shortcut methods specific to my app. If there's something that you do frequently, but it doesn't make sense to move that logic into a service, just add a new protected function.

I won't go and update my other controllers to extend BaseController right this second - I'll do that little-by-little when I need to.

Fetching the User in Twig

Ok: we *now* know how to fetch the User object in a controller. So, how can we fetch it inside a template? Find the templates/ directory and open our account/index.html.twig. The answer is... app.user. That's it! We can call app.user.firstName:

```
8 lines | templates/account/index.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Manage Account!{% endblock %}
4
5  {% block body %}
6      <h1>Manage Your Account {{ app.user.firstName }}</h1>
7  {% endblock %}
```

Try that out. Go back to /account and... perfect!

Symfony gives you exactly *one* global variable in Twig: app. And it just has a few helpful things on it, like app.user and app.session. And because app.user returns *our* User object, we can call firstName on it. Twig will call getFirstName() on

User.

Making the Account Page Pretty

Oh, and, oof. This page is *super* ugly. Clear out the h1. I'm going to paste in some HTML markup I prepared: you can copy this markup from the code block on this page:

49 lines | templates/account/index.html.twig

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

{% block title %}Manage Account!{% endblock %}
... lines 4 - 10
{% block body %}
    <div class="container">
        <div class="row user-menu-container square">
            <div class="col-md-12 user-details">
                <div class="row spacepurplebg white">
                    <div class="col-md-2 no-pad">
                        <div class="user-image">
                            <img src="https://robohash.org/hello@symfonycasts.com" class="img-responsive thumbnail">
                        </div>
                    </div>
                    <div class="col-md-10 no-pad">
                        <div class="user-pad">
                            <h3>Welcome back, ?????</h3>
                            <h4 class="white"><i class="fa fa-twitter"></i> ?????</h4>
                            <a class="btn btn-labeled btn-info" href="#">
                                <span class="btn-label"><i class="fa fa-pencil"></i></span>Update
                            </a>
                        </div>
                    </div>
                </div>
                <div class="row overview">
                    <div class="col-md-4 user-pad text-center">
                        <h3>COMMENTS</h3>
                        <h4>184</h4>
                    </div>
                    <div class="col-md-4 user-pad text-center">
                        <h3>ARTICLES READ</h3>
                        <h4>1,910</h4>
                    </div>
                    <div class="col-md-4 user-pad text-center">
                        <h3>LIKES</h3>
                        <h4>3,892</h4>
                    </div>
                </div>
            </div>
        </div>
    </div>
{% endblock %}
```

If you refresh right now... oof. It still looks pretty terrible. Oh, hello robot! Anyways, the page looks awful because this markup requires another CSS file. If you downloaded the course code, you should have a tutorial/ directory. We already copied this login.css file earlier. Now, copy account.css, find your public/ directory, open css/ and... paste! To include this stylesheet on this page, add block stylesheets and endblock:

```
49 lines | templates/account/index.html.twig
... lines 1 - 4
5  {% block stylesheets %}
... lines 6 - 8
9  {% endblock %}
... lines 10 - 49
```

Inside, call parent() so that we *add* to the existing stylesheets, instead of replacing them. Add link and point to css/account.css:

```
49 lines | templates/account/index.html.twig
... lines 1 - 4
5  {% block stylesheets %}
6      {{ parent() }}
7
8      <link rel="stylesheet" href="{{ asset('css/account.css') }}">
9  {% endblock %}
... lines 10 - 49
```

PhpStorm auto-completes the asset() function for me.

Now refresh again. So much better! All of this markup is 100% hardcoded. But I added friendly ? marks where we need to print some dynamic stuff. Let's do it! For the Avatar, we're using this cool RoboHash site where you give it an email, and it gives you a robot avatar. I love the Internet!

Replace this with app.user.email:

```
49 lines | templates/account/index.html.twig
... lines 1 - 10
11 {% block body %}
12     <div class="container">
13         <div class="row user-menu-container square">
14             <div class="col-md-12 user-details">
15                 <div class="row spacepurplebg white">
16                     <div class="col-md-2 no-pad">
17                         <div class="user-image">
18                             <img src="https://robohash.org/{{ app.user.email }}" class="img-responsive thumbnail">
19                         </div>
20                     </div>
... lines 21 - 29
30                 </div>
... lines 31 - 44
45             </div>
46         </div>
47     </div>
48 {% endblock %}
```

Then, down by "Welcome back", replace that with app.user.firstName:

49 lines | templates/account/index.html.twig

```
... lines 1 - 10
11 {% block body %}
12     <div class="container">
13         <div class="row user-menu-container square">
14             <div class="col-md-12 user-details">
15                 <div class="row spacepurplebg white">
16                     <div class="col-md-2 no-pad">
17                         <div class="user-image">
18                             <img src="https://robohash.org/{{ app.user.email }}" class="img-responsive thumbnail">
19                         </div>
20                     </div>
21                     <div class="col-md-10 no-pad">
22                         <div class="user-pad">
23                             <h3>Welcome back, {{ app.user.firstName }}</h3>
... lines 24 - 27
28                         </div>
29                     </div>
30                 </div>
... lines 31 - 44
45             </div>
46         </div>
47     </div>
48 {% endblock %}
```

Cool! Let's see how it looks like now.

Hey! A brand new robot avatar *and* we see the first name of the dummy user. We *are* still missing this twitter handle... because... our User class doesn't have that property yet:

49 lines | templates/account/index.html.twig

```
... lines 1 - 10
11 {% block body %}
12     <div class="container">
13         <div class="row user-menu-container square">
14             <div class="col-md-12 user-details">
15                 <div class="row spacepurplebg white">
... lines 16 - 20
21                     <div class="col-md-10 no-pad">
22                         <div class="user-pad">
... line 23
24                             <h4 class="white"><i class="fa fa-twitter"></i> ?????</h4>
... lines 25 - 27
28                         </div>
29                     </div>
30                 </div>
... lines 31 - 44
45             </div>
46         </div>
47     </div>
48 {% endblock %}
```

Let's add that next. Add a cool shortcut method to our User class *and* talk about how we can fetch the User object from the one place we haven't talked about yet - services.

Chapter 19: Custom User Method

Our fancy new account page is complete! Oh, except for that missing Twitter username part - aliens freakin' *love* Twitter. The problem is that we don't have this field in our User class yet. No problem, find your terminal and run:

```
$ php bin/console make:entity
```

to update the User entity. Add twitterUsername... and make this nullable in the database: this is an optional field:

147 lines | src/Entity/User.php

```
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 39
40     /**
41      * @ORM\Column(type="string", length=255, nullable=true)
42      */
43     private $twitterUsername;
... lines 44 - 134
135     public function getTwitterUsername(): ?string
136     {
137         return $this->twitterUsername;
138     }
139
140     public function setTwitterUsername(?string $twitterUsername): self
141     {
142         $this->twitterUsername = $twitterUsername;
143
144         return $this;
145     }
146 }
```

Cool! Now run:

```
$ php bin/console make:migration
```

Let's go check that out: look in the Migrations/ directory and open the new file:

29 lines | src/Migrations/Version20180831195803.php

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

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20180831195803 extends AbstractMigration
{
    public function up(Schema $schema) : void
    {
        // 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 \'mys

        $this->addSql('ALTER TABLE user ADD twitter_username VARCHAR(255) DEFAULT NULL');
    }

    public function down(Schema $schema) : void
    {
        // 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 \'mys

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

And... yep! It looks perfect. Move back to your terminal one more time and run:

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

Excellent! Now that we have the new field, let's *set* it on our dummy users in the database. Open UserFixture. Inside the first set of users, add if $this->faker->boolean, then $user->setTwitterUsername($this->faker->userName):

54 lines | src/DataFixtures/UserFixture.php

```
... lines 1 - 8
class UserFixture extends BaseFixture
{
... lines 11 - 17
    protected function loadData(ObjectManager $manager)
    {
        $this->createMany(10, 'main_users', function($i) {
... lines 21 - 24
            if ($this->faker->boolean) {
                $user->setTwitterUsername($this->faker->userName);
            }
... lines 28 - 34
        });
... lines 36 - 51
    }
}
```

The $faker->boolean is cool: it will return true or false randomly. So, about *half* of our users will have a twitter username.

Go reload! Run:

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

Finally! Let's get to work in account/index.html.twig. Replace the ? marks with app.user.twitterUsername:

51 lines | templates/account/index.html.twig

```
... lines 1 - 10
{% block body %}
    <div class="container">
        <div class="row user-menu-container square">
            <div class="col-md-12 user-details">
                <div class="row spacepurplebg white">
... lines 16 - 20
                    <div class="col-md-10 no-pad">
                        <div class="user-pad">
... lines 23 - 24
                            <h4 class="white"><i class="fa fa-twitter"></i> {{ app.user.twitterUsername }}</h4>
... lines 26 - 29
                        </div>
                    </div>
                </div>
... lines 33 - 46
            </div>
        </div>
    </div>
{% endblock %}
```

Hmm, but we probably don't want to show this block if they don't have a twitterUsername. No problem: surround this with an if statement:

```
51 lines | templates/account/index.html.twig
... lines 1 - 10
11 {% block body %}
12     <div class="container">
13         <div class="row user-menu-container square">
14             <div class="col-md-12 user-details">
15                 <div class="row spacepurplebg white">
... lines 16 - 20
21                     <div class="col-md-10 no-pad">
22                         <div class="user-pad">
... line 23
24                             {% if app.user.twitterUsername %}
25                                 <h4 class="white"><i class="fa fa-twitter"></i> {{ app.user.twitterUsername }}</h4>
26                             {% endif %}
... lines 27 - 29
30                         </div>
31                     </div>
32                 </div>
... lines 33 - 46
47             </div>
48         </div>
49     </div>
50 {% endblock %}
```

Perfect! Ok, let's go find a user that has their twitterUsername set! Run:

```
$ php bin/console doctrine:query:sql "SELECT * FROM user"
```

Scroll up and... cool: spacebar1@example.com. Move back to your browser and refresh. Oh! We got logged out! That's because the id of the user that we *were* logged in as was removed from the database when we reloaded the fixtures.

Login as spacebar1@example.com, password engage. Click sign in and... got it!

Custom User Method for RoboHash

Oh, and there's one other thing that we can *finally* update! See the user avatar on the drop-down? That's *totally* hardcoded. Let's roboticize that! Yea, roboticize apparently *is* a real word.

Copy the src for the RoboHash:

51 lines | templates/account/index.html.twig

```
... lines 1 - 10
{% block body %}
    <div class="container">
        <div class="row user-menu-container square">
            <div class="col-md-12 user-details">
                <div class="row spacepurplebg white">
                    <div class="col-md-2 no-pad">
                        <div class="user-image">
                            <img src="https://robohash.org/{{ app.user.email }}" class="img-responsive thumbnail">
                        </div>
                    </div>
... lines 21 - 31
                </div>
... lines 33 - 46
            </div>
        </div>
    </div>
{% endblock %}
```

Then, open up base.html.twig and, instead of pointing to the astronaut's profile image, paste it!

77 lines | templates/base.html.twig

```
... line 1
<html lang="en">
... lines 3 - 15
    <body>
        <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
... lines 18 - 21
            <div class="collapse navbar-collapse" id="navbarNavDropdown">
... lines 23 - 34
                <ul class="navbar-nav ml-auto">
                    {% if is_granted('ROLE_USER') %}
                        <li class="nav-item dropdown" style="margin-right: 75px;">
                            <a class="nav-link dropdown-toggle" href="http://example.com" id="navbarDropdownMenuLink" data-toggle="dropdo
                              <img class="nav-profile-img rounded-circle" src="https://robohash.org/{{ app.user.email }}">
                            </a>
... lines 41 - 47
                        </li>
... lines 49 - 52
                    {% endif %}
                </ul>
            </div>
        </nav>
... lines 57 - 74
    </body>
</html>
```

Try it! Move over and... refresh!

Nice! But, hmm... there is one small thing that I don't like. Right click on the image, copy the image address and paste in a new tab. Oh. That's a pretty big image: 300x300. It's not a huge deal, but our users are downloading a *pretty* big image, *just* to display this teenie-tiny thumbnail.

Fortunately, the fine people who created RoboHash added a feature to help us! By adding ?size=100x100, we can get a smaller image. Let's do that on the menu.

But, wait! Instead of just putting ?size= right here... let's get organized! I don't like duplicating the RoboHash link everywhere. Open your User class. Let's add a new custom function called public function getAvatarUrl().

We don't actually have an avatarUrl property... but that's ok! Give this an int argument that's optional and the method will return a string:

```
157 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 146
147     public function getAvatarUrl(string $size = null): string
148     {
... lines 149 - 154
155     }
156 }
```

Inside, set $url = and paste the RoboHash link. Remove the email but add $this->getEmail():

```
157 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 146
147     public function getAvatarUrl(string $size = null): string
148     {
149         $url = 'https://robohash.org/'.$this->getEmail();
... lines 150 - 154
155     }
156 }
```

Easy enough! For the size part, if a $size is passed in, use $url .= to add sprintf('?size=%dx%d'), passing $size for both of these wildcards. At the bottom, return $url:

```
157 lines | src/Entity/User.php
... lines 1 - 10
11 class User implements UserInterface
12 {
... lines 13 - 146
147     public function getAvatarUrl(string $size = null): string
148     {
149         $url = 'https://robohash.org/'.$this->getEmail();
150
151         if ($size)
152             $url .= sprintf('?size=%dx%d', $size, $size);
153
154         return $url;
155     }
156 }
```

Now that we're done with our fancy new function, go into index.html.twig, remove the long string, and just print app.user.avatarUrl:

51 lines | templates/account/index.html.twig

```
... lines 1 - 10
{% block body %}
    <div class="container">
        <div class="row user-menu-container square">
            <div class="col-md-12 user-details">
                <div class="row spacepurplebg white">
                    <div class="col-md-2 no-pad">
                        <div class="user-image">
                            <img src="{{ app.user.avatarUrl }}" class="img-responsive thumbnail">
                        </div>
                    </div>
... lines 21 - 31
                </div>
... lines 33 - 46
            </div>
        </div>
    </div>
{% endblock %}
```

We can reference avatarUrl like a property, but behind the scenes, we know that Twig is smart enough to call the getAvatarUrl() method.

Copy that, go back into base.html.twig and paste. But this time, call it like a function: pass 100:

77 lines | templates/base.html.twig

```
... line 1
<html lang="en">
... lines 3 - 15
    <body>
        <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
... lines 18 - 21
            <div class="collapse navbar-collapse" id="navbarNavDropdown">
... lines 23 - 34
                <ul class="navbar-nav ml-auto">
                    {% if is_granted('ROLE_USER') %}
                        <li class="nav-item dropdown" style="margin-right: 75px;">
                            <a class="nav-link dropdown-toggle" href="http://example.com" id="navbarDropdownMenuLink" data-toggle="dropdo
                              <img class="nav-profile-img rounded-circle" src="{{ app.user.avatarUrl(100) }}">
                            </a>
... lines 41 - 47
                        </li>
... lines 49 - 52
                    {% endif %}
                </ul>
            </div>
        </nav>
... lines 57 - 74
    </body>
</html>
```

Let's see if it works! Close a tab then, refresh! Yep! And if we copy the image address again and load it... nice! A little bit smaller.

Next, let's find out how to fetch the user object from the *one* spot we haven't talked about yet: services.

Chapter 20: Fetching the User In a Service

We know how to get the user in a template:

77 lines | templates/base.html.twig

```
... line 1
2  <html lang="en">
... lines 3 - 15
16     <body>
17         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
... lines 18 - 21
22             <div class="collapse navbar-collapse" id="navbarNavDropdown">
... lines 23 - 34
35                 <ul class="navbar-nav ml-auto">
36                     {% if is_granted('ROLE_USER') %}
37                         <li class="nav-item dropdown" style="margin-right: 75px;">
38                             <a class="nav-link dropdown-toggle" href="http://example.com" id="navbarDropdownMenuLink" data-toggle="dropdo
39                               <img class="nav-profile-img rounded-circle" src="{{ app.user.avatarUrl(100) }}">
40                             </a>
... lines 41 - 47
48                         </li>
... lines 49 - 52
53                     {% endif %}
54                 </ul>
55             </div>
56         </nav>
... lines 57 - 74
75     </body>
76 </html>
```

And... we know how to get the user from a controller with $this->getUser():

25 lines | src/Controller/AccountController.php

```
... lines 1 - 11
12 class AccountController extends BaseController
13 {
... lines 14 - 16
17     public function index(LoggerInterface $logger)
18     {
19         $logger->debug('Checking account page for '.$this->getUser()->getEmail());
... lines 20 - 22
23     }
24 }
```

But... what about from inside a service? Because this nice $this->getUser() shortcut will *only* work in controllers.

To show you what I mean, I need to remind you of a feature we built a long time ago, like 3 screencasts ago. Click on any article. Then, click anywhere on the web debug toolbar to open the profiler. Find the "Logs" section and click on "Info & Errors". There it is!

> They are talking about bacon again!

This is a super-informative log message that *we* added from inside our markdown service: src/Service/MarkdownHelper.php:

```
44 lines | src/Service/MarkdownHelper.php
... lines 1 - 8
class MarkdownHelper
{
... lines 11 - 23
    public function parse(string $source): string
    {
        if (stripos($source, 'bacon') !== false) {
            $this->logger->info('They are talking about bacon again!');
        }

        // skip caching entirely in debug
        if ($this->isDebug) {
            return $this->markdown->transform($source);
        }

        $item = $this->cache->getItem('markdown_'.md5($source));
        if (!$item->isHit()) {
            $item->set($this->markdown->transform($source));
            $this->cache->save($item);
        }

        return $item->get();
    }
}
```

This code parses the article content through markdown and caches it. But also, *if* it sees the word "bacon" in the content ... which *every* article has in our fixtures, it logs this message.

So here's our challenge: I want to add information about *who* is currently logged in to this message. To do that, we need to answer one question: how can we access the current User object from inside a service?

The Security Service

The answer is... of *course* - by using another service. The name of the service that gives you access to the User object is easy to remember. Add another argument: Security $security:

```
50 lines | src/Service/MarkdownHelper.php
... lines 1 - 7
use Symfony\Component\Security\Core\Security;

class MarkdownHelper
{
... lines 12 - 18
    public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $markdownLogger, bool $is
    {
... lines 21 - 25
    }
... lines 27 - 48
}
```

I'll hit Alt+Enter and click "Initialize Fields" to create that property and set it:

```
50 lines | src/Service/MarkdownHelper.php
... lines 1 - 7
8  use Symfony\Component\Security\Core\Security;
9
10 class MarkdownHelper
11 {
... lines 12 - 16
17     private $security;
18
19     public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $markdownLogger, bool $is
20     {
... lines 21 - 24
25         $this->security = $security;
26     }
... lines 27 - 48
49 }
```

So how can we use this service? Well... let's just look inside! Hold Command or Control and click to open the Security class. It has just two important methods: getUser() and isGranted(). Hey! That makes a lot of sense! Remember, once you set up authentication, there are only *two* things you can do with security: get the user object or figure out whether or not the user should have access to something, like a role. That's what isGranted() does.

Close that and move down to the log message. Ok, we *could* get the user object, maybe call getEmail() on it, and concatenate that onto the end of the log string. But! There's a *cooler* way. Add a 2nd argument to info: an array. Give it a user key - I'm just making that up - and set it to the user *object*: $this->security->getUser():

```
50 lines | src/Service/MarkdownHelper.php
... lines 1 - 9
10 class MarkdownHelper
11 {
... lines 12 - 27
28     public function parse(string $source): string
29     {
30         if (stripos($source, 'bacon') !== false) {
31             $this->logger->info('They are talking about bacon again!', [
32                 'user' => $this->security->getUser()
33             ]);
34         }
... lines 35 - 47
48     }
49 }
```

Unrelated to security, every method on the logger, like info(), debug() or alert(), has *two* arguments. The first is the message string. The *second* is an optional array called a "context". This is just an array of any extra info that you want to include with the log message. I invented a user key and set it to the User object.

Let's go see what it looks like! Refresh! Then, click back into the profiler, find logs, and check out "Info & Errors". The message looks the same, but now we have a "Show Context" link. Click that! Nice! There is our *entire* User object in all of its glory. That's pretty sweet. And *now*, you know how to get the User object from anywhere.

Next, we get to talk about a feature called "role hierarchy". A little feature that will make you *love* working with roles, especially if you have complex access rules.

Chapter 21: Role Hierarchy

So far, our site has two types of users. First, for some pages, like the account page, we only care that you are logged in - a "normal" user. And second, there are a few admin pages. Open up ArticleAdminController and CommentAdminController. Both of these are protected by ROLE_ADMIN:

31 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 11
/**
 * @IsGranted("ROLE_ADMIN")
 */
class ArticleAdminController extends AbstractController
{
... lines 17 - 29
}
```

37 lines | src/Controller/CommentAdminController.php

```
... lines 1 - 11
/**
 * @IsGranted("ROLE_ADMIN")
 */
class CommentAdminController extends Controller
{
... lines 17 - 35
}
```

A lot of sites are just this simple: you have normal users and admin users, who have access to *all* of the admin sections. But, if you have a more complex setup - like a bigger company where different groups of people need access to different things, this isn't good enough. The question is: what's the best way to organize that with roles?

Role Naming

Well, there are only two possibilities. First, you could use roles that are named by the *type* of user that will have them - like ROLE_EDITOR, ROLE_HUMAN_RESOURCES or ROLE_THE_PERSON_THAT_OWNS_THE_COMPANY... or something like that. But, I don't *love* this option. It's just not super clear what having ROLE_EDITOR will give me access to.

Instead, I like to use role names that *specifically* describe *what* you're protecting - like ROLE_ADMIN_ARTICLE for ArticleAdminController:

31 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 11
/**
 * @IsGranted("ROLE_ADMIN_ARTICLE")
 */
class ArticleAdminController extends AbstractController
{
... lines 17 - 29
}
```

And, for CommentAdminController: ROLE_ADMIN_COMMENT:

37 lines | src/Controller/CommentAdminController.php

```
... lines 1 - 11
/**
 * @IsGranted("ROLE_ADMIN_COMMENT")
 */
class CommentAdminController extends Controller
{
... lines 17 - 35
}
```

Oh, and also open base.html.twig. There's one other spot here where we use ROLE_ADMIN. There it is: to hide or show the "Create Post" link. *Now* that should be ROLE_ADMIN_ARTICLE:

77 lines | templates/base.html.twig

```
... line 1
<html lang="en">
... lines 3 - 15
    <body>
        <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
... lines 18 - 21
            <div class="collapse navbar-collapse" id="navbarNavDropdown">
... lines 23 - 34
                <ul class="navbar-nav ml-auto">
                    {% if is_granted('ROLE_USER') %}
                        <li class="nav-item dropdown" style="margin-right: 75px;">
... lines 38 - 40
                            <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
... line 42
                                {% if is_granted('ROLE_ADMIN_ARTICLE') %}
                                    <a class="dropdown-item" href="{{ path('admin_article_new') }}">Create Post</a>
                                {% endif %}
... line 46
                            </div>
                        </li>
... lines 49 - 52
                    {% endif %}
                </ul>
            </div>
        </nav>
... lines 57 - 74
    </body>
</html>
```

role_hierarchy

I love it! Except... for one problem. Go to /admin/comment. Access denied! Well, I'm not even logged in as an admin user. But even if I *were*, I would still not have access! Admin users do *not* have these two new roles!

And, yea, we *could* go back to UserFixture, add ROLE_ADMIN_COMMENT and ROLE_ADMIN_ARTICLE and *then* reload the fixtures. But, this highlights an annoying problem. *Each* time we add a new admin section to the site and introduce a new role, we will need to go into the database, find *all* the users who need access to that new section, and give *them* that new role. That's a bummer!

But... don't worry! Symfony has our backs with a sweet feature called role_hierarchy. Open config/packages/security.yaml. Anywhere inside, I'll do it above firewalls, add role_hierarchy. Below, put ROLE_ADMIN set to an array with ROLE_ADMIN_COMMENT and ROLE_ADMIN_ARTICLE:

55 lines | config/packages/security.yaml

```yaml
security:
... lines 2 - 13
    role_hierarchy:
        ROLE_ADMIN: [ROLE_ADMIN_COMMENT, ROLE_ADMIN_ARTICLE]
... lines 16 - 55
```

It's *just* that simple. Now, *anybody* that has ROLE_ADMIN *also* has these two roles, automatically. To prove it, go log out so that we can log in as one of our admin users: admin2@thespacebar.com, password engage.

Go back to /admin/comment and... access granted!

This is even cooler than you might think! It allows us to organize our roles into different groups of people in our company. For example, ROLE_EDITOR could be given access to all the sections that "editors" need. Then, the *only* role that you need to *assign* to an editor user is this *one* role: ROLE_EDITOR. And if all editors need access to a new section in the future, just add that new role to role_hierarchy.

We can use this new super-power to try out a *really* cool feature that allows you to *impersonate* users... and become the international spy you always knew you would.

Chapter 22: Impersonation (switch_user)

While we're inside security.yaml, I want to talk about another really cool feature called switch_user. Imagine you're an admin user and you're trying to debug an issue that a customer saw. But, dang it! The feature works perfectly for you! Is the customer wrong? Or is there something unique to their account? We'll never know! Time to find a different career! The end is nigh!

Suddenly, a super-hero swoops in to save the day! This hero's name? switch_user.

In security.yaml, under your firewall, activate our hero with a new key: switch_user set to true:

```
57 lines | config/packages/security.yaml
1  security:
   ... lines 2 - 16
17     firewalls:
   ... lines 18 - 20
21         main:
   ... lines 22 - 34
35             switch_user: true
   ... lines 36 - 57
```

As *soon* as you do this, you can go to *any* URL and add ?_switch_user= and the email address of a user that you want to impersonate. Let's try spacebar1@example.com.

And... access denied! Of course! To prevent *any* user from taking advantage of this little trick, the switch_user feature requires you to have a special role called ROLE_ALLOWED_TO_SWITCH. Go back to security.yaml and give ROLE_ADMIN users this new role under role_hierarchy:

```
57 lines | config/packages/security.yaml
1  security:
   ... lines 2 - 13
14     role_hierarchy:
15         ROLE_ADMIN: [ROLE_ADMIN_COMMENT, ROLE_ADMIN_ARTICLE, ROLE_ALLOWED_TO_SWITCH]
   ... lines 16 - 57
```

Ok, watch closely: we still have the magic ?_switch_user= in the URL. Hit enter. That's gone, yea! I'm logged in as spacebar1@example.com! You can see this down in the web debug toolbar. Of course, this normal user can't access this page. But if you go back to the homepage, you can surf around as the spacebar1 user.

User Provider & _switch_user

Oh, by the way, the reason that we use the email address with _switch_user, and not some other field like the id, is due to the user provider. Remember, this is the code inside Symfony that helps reload the user from the session at the beginning of each request. But it is *also* used by a few other features to load the user, like remember_me and switch_user. If you're using the Doctrine user provider like we are, then this property key determines which field will be used for all of this:

```
57 lines | config/packages/security.yaml
security:
... lines 2 - 6
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
... line 11
                property: email
... lines 13 - 57
```

If you changed this to id, we would need to use the id with switch user.

Adding a Banner when you are Impersonating

Anyways, to *exit* and return to your normal identity, find a phone booth, close the door, and add ?_switch_user=_exit to any URL. And... we're back to being us!

Switch one more time back to spacebar1@example.com. One of the *only* issues with _switch_user is that it's not super obvious that we're switched! Yep, you might switch to a user, go check Facebook, then come back, forget that you're *still* switched to them, and start commenting on their behalf. What? No, I've definitely never done this... I'm just saying it's *possible*.

To prevent these... awkward situations, let's put a big banner on top when we're switched. Open base.html.twig and find the body tag. Here's the key: *when* we are switched to another user, Symfony gives us a special role called ROLE_PREVIOUS_ADMIN. We can use that to our advantage: if is_granted('ROLE_PREVIOUS_ADMIN'), then print an alert block. Inside, say:

You are currently switched to this user

```
83 lines | templates/base.html.twig
... line 1
<html lang="en">
... lines 3 - 15
    <body>
        {% if is_granted('ROLE_PREVIOUS_ADMIN') %}
            <div class="alert alert-warning" style="margin-bottom: 0;">
                You are currently switched to this user.
... line 20
            </div>
        {% endif %}
... lines 23 - 80
    </body>
</html>
```

And, to maximize our fanciness, let's add a link to exit. Use the path function to point to app_homepage. For the second argument, pass an array with the necessary _switch_user set to _exit. At the end, say "Exit Impersonation":

83 lines | templates/base.html.twig

```
... line 1
<html lang="en">
... lines 3 - 15
    <body>
        {% if is_granted('ROLE_PREVIOUS_ADMIN') %}
            <div class="alert alert-warning" style="margin-bottom: 0;">
                You are currently switched to this user.
                <a href="{{ path('app_homepage', {'_switch_user': '_exit'}) }}">Exit Impersonation</a>
            </div>
        {% endif %}
... lines 23 - 80
    </body>
</html>
```

Adding Query Parameters with path()

Let's see how it looks! Move over and refresh! Nice! Even I won't forget when I'm impersonating. And, check out the URL on the link: it's perfect - ?_switch_user=_exit. But... wait... the way we just used the path() function was a bit weird.

Why? Open templates/article/homepage.html.twig and find the article list. You might remember that the second argument of the path() function is *normally* used to fill in the "wild card" values for a route:

65 lines | templates/article/homepage.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
... lines 6 - 8
            <div class="col-sm-12 col-md-8">
... lines 10 - 21
                <div class="article-container my-1">
                    <a href="{{ path('article_show', {slug: article.slug}) }}">
... lines 24 - 38
                </div>
... line 40
            </div>
... lines 42 - 61
        </div>
    </div>
{% endblock %}
```

Hold Command or Control and click article_show. Yep! This route has a {slug} wild card:

```
65 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 37
38     /**
39      * @Route("/news/{slug}", name="article_show")
40      */
41     public function show(Article $article, SlackClient $slack)
42     {
... lines 43 - 49
50     }
... lines 51 - 63
64 }
```

And so, when we link to it, we need to pass a value for that slug wildcard via the 2nd argument to path().

We *already* knew that. And *this* is the *normal* purpose of the second argument to path(). However, *if* you pass a key to the second argument, and that route does *not* have a wildcard with that name, Symfony just adds it as a query parameter.

That is why we can click this link to exit impersonation.

Next - let's build an API endpoint with Symfony's serializer! That will be our *first* step towards API authentication.

Chapter 23: Serializer & API Endpoint

In addition to our login form authentication, I *also* want to allow users to log in by sending an API token. But, before we get there, let's make a proper API endpoint first.

Creating the API Endpoint

I'll close a few files and open AccountController. To keep things simple, we'll create an API endpoint right here. Add a public function at the bottom called accountApi():

35 lines | src/Controller/AccountController.php

```
... lines 1 - 11
12 class AccountController extends BaseController
13 {
... lines 14 - 27
28     public function accountApi()
29     {
... lines 30 - 32
33     }
34 }
```

This new endpoint will return the JSON representation of whoever is logged in. Above, add @Route("/api/account") with name="api_account":

35 lines | src/Controller/AccountController.php

```
... lines 1 - 11
12 class AccountController extends BaseController
13 {
... lines 14 - 24
25     /**
26      * @Route("/api/account", name="api_account")
27      */
28     public function accountApi()
29     {
... lines 30 - 32
33     }
34 }
```

The code here is simple - excitingly simple! $user = $this->getUser() to find who's logged in:

```
35 lines | src/Controller/AccountController.php
... lines 1 - 11
12 class AccountController extends BaseController
13 {
... lines 14 - 24
25     /**
26      * @Route("/api/account", name="api_account")
27      */
28     public function accountApi()
29     {
30         $user = $this->getUser();
... lines 31 - 32
33     }
34 }
```

We can safely do this thanks to the annotation on the class: every method requires authentication. Then, to transform the User object into JSON - this is pretty cool - return $this->json() and pass $user:

```
35 lines | src/Controller/AccountController.php
... lines 1 - 11
12 class AccountController extends BaseController
13 {
... lines 14 - 24
25     /**
26      * @Route("/api/account", name="api_account")
27      */
28     public function accountApi()
29     {
30         $user = $this->getUser();
31
32         return $this->json($user);
33     }
34 }
```

Let's try it! In your browser, head over to /api/account. And! Oh! That's not what I expected! It's JSON... but it's totally empty!

Installing the Serializer

Why? Hold Command or Control and click into the json() method. This method does two different things, depending on your setup. First, it checks to see if Symfony's serializer component is installed. Right now, it is *not*. So, it falls back to passing the User object to the JsonResponse class. I won't open that class, but *all* it does internally is called json_encode() on that data we pass in: the User object in this case.

Do you know what happens when you call json_encode() on an object in PHP? It only... sorta works: it encodes only the *public* properties on that class. And because we have *no* public properties, we get back nothing!

This is actually the *entire* point of Symfony's serializer component! It's a kick butt way to turn objects into JSON, or any other format. I don't want to talk *too* much about the serializer right now: we're trying to learn security! But, I *do* want to use it. Find your terminal and run:

```
$ composer require serializer
```

This installs the serializer pack, which downloads the serializer and a few other things. As *soon* as this finishes, the json() method will start using the new serializer service. Try it - refresh! Hey! It works! That's awesome!

Serialization Groups

Except... well... we probably don't want to include *all* of these properties - especially the encoded password. I know, I said we *weren't* going to talk about the serializer, and yet, I *do* want to fix this one thing!

Open your User class. To control which fields are serialized, above each property, you can use an annotation to organize into "groups". I won't expose the id, but let's expose email by putting it into a group: @Groups("main"):

```
161 lines | src/Entity/User.php
... lines 1 - 6
7  use Symfony\Component\Serializer\Annotation\Groups;
... lines 8 - 11
12 class User implements UserInterface
13 {
... lines 14 - 20
21     /**
... line 22
23      * @Groups("main")
24      */
25     private $email;
... lines 26 - 159
160 }
```

When I auto-completed that annotation, the PHP Annotations plugin added the use statement I need to the top of the file:

```
161 lines | src/Entity/User.php
1  <?php
... lines 2 - 6
7  use Symfony\Component\Serializer\Annotation\Groups;
... lines 8 - 161
```

Oh, and I totally invented the "main" part - that's the group name, and you'll see how I use it in a minute. Copy the annotation and also add firstName and twitterUsername to that same group:

```
161 lines | src/Entity/User.php
... lines 1 - 11
12 class User implements UserInterface
13 {
... lines 14 - 20
21     /**
... line 22
23      * @Groups("main")
24      */
25     private $email;
... lines 26 - 31
32     /**
... line 33
34      * @Groups("main")
35      */
36     private $firstName;
... lines 37 - 42
43     /**
... line 44
45      * @Groups("main")
46      */
47     private $twitterUsername;
... lines 48 - 159
160 }
```

To complete this, in AccountController, we just need to tell the json() method to *only* serialize properties that are in the group called "main". To do that, pass the normal 200 status code as the second argument, we don't need any custom headers, but we *do* want to pass one item to "context". Set groups => an array with the string main:

37 lines | src/Controller/AccountController.php

```
... lines 1 - 11
12 class AccountController extends BaseController
13 {
... lines 14 - 24
25     /**
26      * @Route("/api/account", name="api_account")
27      */
28     public function accountApi()
29     {
... lines 30 - 31
32         return $this->json($user, 200, [], [
33             'groups' => ['main'],
34         ]);
35     }
36 }
```

You can include just one group name here like this, or tell the serializer to serialize the properties from multiple groups.

Let's try it! Refresh! Yes! Just these three fields.

Ok, we are *now* ready to take on a big, cool topic: API token authentication.

Chapter 24: API Auth: Do you Need it? And its Parts

Before we dive into the code, we need to have a heart-to-heart about API authentication. Because... I think there's some confusion out there that tends to make people over-complicate things. And I *never* want to over-complicate things.

Do you Need API Authentication

First, you need to ask yourself a very important question:

> Do you actually need an API token authentication system?

There's a *pretty* good chance that the answer is... no, *even* if your app has API endpoints:

37 lines | src/Controller/AccountController.php

```
... lines 1 - 11
12 class AccountController extends BaseController
13 {
... lines 14 - 24
25     /**
26      * @Route("/api/account", name="api_account")
27      */
28     public function accountApi()
29     {
... lines 30 - 31
32         return $this->json($user, 200, [], [
33             'groups' => ['main'],
34         ]);
35     }
36 }
```

If you're creating API endpoints *solely* so that your *own* JavaScript for your *own* site can use them, then, you do *not* need an API token authentication system. Nope! Your life will be much simpler if you use a normal login form and session-based authentication.

Yep! You probably already know, that, once you login via a login form, you can instantly make authenticated AJAX requests from JavaScript, because those requests send the session cookie. So, if the only thing that needs to use your API is your own JavaScript, just use LoginFormAuthenticator.

Oh, and if you need to be fancier with your login form, sure! You can *totally* use JavaScript to make the login form submit via AJAX. Nothing would need to change in your authenticator, except that you would probably want to send back JSON on success, instead of redirecting:

```
89 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
... lines 22 - 74
75     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
76     {
77         if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
78             return new RedirectResponse($targetPath);
79         }
80
81         return new RedirectResponse($this->router->generate('app_homepage'));
82     }
... lines 83 - 87
88 }
```

You would also override onAuthenticationError() and the start() method to do the same. We'll learn more about those methods soon.

Of course, even if your JavaScript will be the only thing using your API, you *can* still build an API token authentication system, if you want. And if you need *other* things to be able to access your API, then you *need* a token system.

Two Sides of API Token Authentication

If you're still here, then you've either decided that you *do* need an API token authentication system, or you just want to nerd out with us on this topic. Me too!

This brings us to important topic number 2! An API token authentication system has two, quite *unrelated* parts. The first is *how* your app processes an existing API token and logs in the user. The second is how those API tokens are *created* and *distributed*.

Part 1: Processing an API Token

For the first part, no matter how you build it, an API token is just a string that is somehow connected to a User in your system. The client that makes the request sets the token string on a header and then they become authenticated *as* that User. There are some variations on this, like, giving tokens "scopes" or "permissions" so that they can only do *some* things that a user can do, but that's the basic idea.

The *way* that the token string is related to the user can be done in a few different ways. For example, you could have an API token database table where each random API token has a relationship to a row in the user table. It's simple: our app reads the token string from a header, finds that API token in the database, finds the User it's related to, and authenticates as that user. We're going to build exactly this.

Another variation is JSON web tokens. In this case, instead of the token being a random string, the user's information - like the user id - is used to create a signed string. In that case, your app reads the header, verifies the signature on the string, and uses the id inside that string to query for the User.

Anyways, *that* is the first part of API token authentication: designing your app to be able to read API tokens from an API request, and use that information - somehow - to find the correct User and authenticate them.

Part 2: Creating & Distributing API Tokens

The *second* part of an API authentication system asks this question:

> How are these API tokens created and distributed?

It turns out that *this* is a totally separate conversation. And, once again, there are several valid answers. I'll give you 3 examples with when each should probably be used. Actually, the GitHub API is an example of a system that allows you to do all *three* of these.

First, you could allow API tokens to be created through a web interface. Like, a user logs in, they navigate to some API token page, and then they create one or more API tokens that are tied to their account. This solution is dead simple. The negative is

that there is no automated way to create an API token: you can't write a script that can create them. It must be done manually.

Second, you could write an API endpoint whose jobs is to create & return tokens. In this example, you would send your email & password to the API endpoint, it would validate them, then create & return the token. This is *still* pretty simple, but now it's programmable: you can write a script that can create tokens on its own. The *downside* is that this solution can't be used by third parties. What I mean is, it's okay for the *user* to write some code that sends their own email and password to an API endpoint in order to create a token. But, if some third-party were building an iPhone app for your site, that app should *not* use this method. Why? Because it would require the user to enter their email & password directly into the app, so that it could send the info to our API. Ideally, we *never* want users to give their password to a third-party.

This leads us to the *third* way of creating & distributing tokens: OAuth2. If you need third-parties to be able to securely create & get API tokens for your users, then you probably need OAuth. The only negative is that OAuth is more complex.

Phew! So, the *whole* second part of API token authentication... well, really has nothing to do with authentication at all! It's more about how these secrets keys are created and handed out to who needs them. So, we are *not* going to talk about that part of API authentication.

But we *are* going to build the first part: the *true* authentication part. Let's get to work!

Chapter 25: ApiToken Entity

Time to get to work on our API token authentication system! As we just learned, there are a bunch of different ways to do API auth. We're going to code through *one* way, which will make you *plenty* dangerous for whatever way *you* ultimately need.

For our API tokens, we're going to create an ApiToken entity in the database to store them. Find your terminal and run:

```
$ php bin/console make:entity
```

Call the class ApiToken. And, we need a few fields: token, a string that's not nullable, expiresAt so that we can set an expiration as a datetime, and user, which will be a relation type to our User class. In this situation, we want a ManyToOne relationship so that each ApiToken has one User and each User can have many ApiTokens. Make this *not* nullable: every API token must be related to a User. And, though it doesn't matter for authentication, let's map both sides of the relationship. That will allow us to easily fetch all of the API tokens for a specific user. For orphanRemoval, this is also not important, but choose yes. If we create a page where a user can manage their API tokens, this might make it easier to delete API tokens.

And... done!

76 lines | src/Entity/ApiToken.php

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

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\ApiTokenRepository")
 */
class ApiToken
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

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

    /**
     * @ORM\Column(type="datetime")
     */
    private $expiresAt;

    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\User", inversedBy="apiTokens")
     * @ORM\JoinColumn(nullable=false)
     */
    private $user;

```

```
35      public function getId(): ?int
36      {
37          return $this->id;
38      }
39
40      public function getToken(): ?string
41      {
42          return $this->token;
43      }
44
45      public function setToken(string $token): self
46      {
47          $this->token = $token;
48
49          return $this;
50      }
51
52      public function getExpiresAt(): ?\DateTimeInterface
53      {
54          return $this->expiresAt;
55      }
56
57      public function setExpiresAt(\DateTimeInterface $expiresAt): self
58      {
59          $this->expiresAt = $expiresAt;
60
61          return $this;
62      }
63
64      public function getUser(): ?User
65      {
66          return $this->user;
67      }
68
69      public function setUser(?User $user): self
70      {
71          $this->user = $user;
72
73          return $this;
74      }
75  }
```

Generate the migration with:

```
$ php bin/console make:migration
```

Go check it out - in the Migrations/ directory, open that file:

30 lines | src/Migrations/Version20180901171717.php

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

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20180901171717 extends AbstractMigration
{
    public function up(Schema $schema) : void
    {
        // 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 \'mys

        $this->addSql('CREATE TABLE api_token (id INT AUTO_INCREMENT NOT NULL, user_id INT NOT NULL, token VARCHAR(25
        $this->addSql('ALTER TABLE api_token ADD CONSTRAINT FK_7BA2F5EBA76ED395 FOREIGN KEY (user_id) REFERENCES
    }

    public function down(Schema $schema) : void
    {
        // 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 \'mys

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

Cool! CREATE TABLE api_token with id, user_id, token and expires_at. And, it creates the foreign key.

That looks perfect. Move back and run it!

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

How are Tokens Created?

So, the question of *how* these ApiTokens will be created is *not* something we're going to answer. As we talked about, it's either super easy... or super complicated, depending on your needs.

So, for our app, we're just going to create some ApiTokens via the fixtures.

Making the ApiToken Class Awesome

But before we do that, open the new ApiToken entity class. Yep, all the usual stuff: some properties, annotations and a getter & setter for each method. I want to change things a bit. The make:entity command always generates getter and setter methods. But, in some cases, there is a better way to design things.

Add a public function __construct() method with a User argument:

62 lines | src/Entity/ApiToken.php

```
... lines 1 - 9
10 class ApiToken
11 {
... lines 12 - 34
35     public function __construct(User $user)
36     {
... lines 37 - 39
40     }
... lines 41 - 60
61 }
```

Because every ApiToken needs a User, why not make it required when the object is instantiated? Oh, and we can *also* generate the random token string here. Use $this->token = bin2hex(random_bytes(60)). Then $this->user = $user:

62 lines | src/Entity/ApiToken.php

```
... lines 1 - 9
10 class ApiToken
11 {
... lines 12 - 34
35     public function __construct(User $user)
36     {
37         $this->token = bin2hex(random_bytes(60));
38         $this->user = $user;
... line 39
40     }
... lines 41 - 60
61 }
```

Oh, and we can also set the expires time here - $this->expiresAt = new \DateTime() with +1 hour:

62 lines | src/Entity/ApiToken.php

```
... lines 1 - 9
10 class ApiToken
11 {
... lines 12 - 34
35     public function __construct(User $user)
36     {
37         $this->token = bin2hex(random_bytes(60));
38         $this->user = $user;
39         $this->expiresAt = new \DateTime('+1 hour');
40     }
... lines 41 - 60
61 }
```

You can set the expiration time for however long you want.

Now that we are initializing everything in the constructor, we can clean up the class: remove all the setter methods:

62 lines | src/Entity/ApiToken.php

```
... lines 1 - 9
10 class ApiToken
11 {
12     /**
13      * @ORM\Id()
14      * @ORM\GeneratedValue()
15      * @ORM\Column(type="integer")
16      */
17     private $id;
18
19     /**
20      * @ORM\Column(type="string", length=255)
21      */
22     private $token;
23
24     /**
25      * @ORM\Column(type="datetime")
26      */
27     private $expiresAt;
28
29     /**
30      * @ORM\ManyToOne(targetEntity="App\Entity\User", inversedBy="apiTokens")
31      * @ORM\JoinColumn(nullable=false)
32      */
33     private $user;
34
35     public function __construct(User $user)
36     {
37         $this->token = bin2hex(random_bytes(60));
38         $this->user = $user;
39         $this->expiresAt = new \DateTime('+1 hour');
40     }
41
42     public function getId(): ?int
43     {
44         return $this->id;
45     }
46
47     public function getToken(): ?string
48     {
49         return $this->token;
50     }
51
52     public function getExpiresAt(): ?\DateTimeInterface
53     {
54         return $this->expiresAt;
55     }
56
57     public function getUser(): ?User
58     {
59         return $this->user;
60     }
61 }
```

Yep, our token class is now *immutable*, which wins us *major* hipster points. Immutable just means that, once it's instantiated, this object's data can never be changed. Some developers think that making immutable objects like this is *super* important. I don't fully agree with that. But, it *definitely* makes sense to be thoughtful about your entity classes. Sometimes having setter methods makes sense. But sometimes, it makes more sense to setup some things in the constructor and remove the setter methods if you don't need them.

Oh, and if, in the future, you want to *update* the data in this entity - maybe you need to change the expiresAt, it's totally OK to add a new public function to allow that. But, when you do, again, be thoughtful. You *could* add a public function setExpiresAt(). Or, if all you ever do is re-set the expiresAt to one hour from now, you could instead create a public function renewExpiresAt() that handles that logic for you:

```
public function renewExpiresAt()
{
    $this->expiresAt = new \DateTime('+1 hour');
}
```

That method name is more meaningful, and centralizes more control inside the class.

Ok, I'm done with my rant!

Adding ApiTokens to the Fixtures

Let's create some ApiTokens in the fixtures already! We *could* create a new ApiTokenFixture class, but, to keep things simple, I'm going to put the logic right inside UserFixture.

Use $apiToken1 = new ApiToken() and pass our User. Copy that and create $apiToken2:

```
60 lines | src/DataFixtures/UserFixture.php
    ... lines 1 - 4
5   use App\Entity\ApiToken;
    ... lines 6 - 9
10  class UserFixture extends BaseFixture
11  {
    ... lines 12 - 18
19      protected function loadData(ObjectManager $manager)
20      {
21          $this->createMany(10, 'main_users', function($i) use ($manager) {
    ... lines 22 - 34
35              $apiToken1 = new ApiToken($user);
36              $apiToken2 = new ApiToken($user);
    ... lines 37 - 39
40              return $user;
41          });
    ... lines 42 - 57
58      }
59  }
```

With our fancy createMany() method, you do *not* need to call persist() or flush() on the object that you return. That's because our base class calls persist() on the object *for* us:

```
92 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 9
10 abstract class BaseFixture extends Fixture
11 {
... lines 12 - 45
46     protected function createMany(int $count, string $groupName, callable $factory)
47     {
48         for ($i = 0; $i < $count; $i++) {
... lines 49 - 54
55             $this->manager->persist($entity);
... lines 56 - 58
59         }
60     }
... lines 61 - 90
91 }
```

But, if you create some objects manually - like this - you *do* need to call persist(). No big deal: add use ($manager) to make the variable available in the callback. Then,$manager->persist($apiToken1) and $manager->persist($apiToken2):

```
60 lines | src/DataFixtures/UserFixture.php
... lines 1 - 4
5 use App\Entity\ApiToken;
... lines 6 - 9
10 class UserFixture extends BaseFixture
11 {
... lines 12 - 18
19     protected function loadData(ObjectManager $manager)
20     {
21         $this->createMany(10, 'main_users', function($i) use ($manager) {
... lines 22 - 34
35             $apiToken1 = new ApiToken($user);
36             $apiToken2 = new ApiToken($user);
37             $manager->persist($apiToken1);
38             $manager->persist($apiToken2);
... lines 39 - 40
41         });
... lines 42 - 57
58     }
59 }
```

That should be it! Let's reload some fixtures!

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

When it's done, run:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM api_token'
```

Beautiful, long, random strings. And *each* is related to a User.

Next, let's create an authenticator that's capable of reading, processing & authenticating these API tokens.

Chapter 26: Entry Point: Helping Users Authenticate

We now have a database table full of API Tokens where each is related to a User. I can *already* feel the API power! So here's our new goal: when an API request sends a valid API token string, we'll read it and *authenticate* that request *as* the User who owns the token:

62 lines | src/Entity/ApiToken.php

```
... lines 1 - 9
10 class ApiToken
11 {
... lines 12 - 28
29     /**
30      * @ORM\ManyToOne(targetEntity="App\Entity\User", inversedBy="apiTokens")
31      * @ORM\JoinColumn(nullable=false)
32      */
33     private $user;
... lines 34 - 60
61 }
```

make:auth ApiTokenAuthenticator

This will be the *second* way that users can authenticate in our app. So, we need a *second* authenticator. Find your terminal and run:

```
$ php bin/console make:auth
```

If you see a question about choosing which *type* of authentication you want, choose an "Empty authenticator". I'm using an older version of the command, which *only* generates empty authenticators. Call it ApiTokenAuthenticator. Oh, and you may also be asked a question about an "Entry point". We'll talk about that soon, but choose the LoginFormAuthenticator option.

Ok, go check this out!

54 lines | src/Security/ApiTokenAuthenticator.php

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

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Guard\AbstractGuardAuthenticator;

class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
    public function supports(Request $request)
    {
        // todo
    }

    public function getCredentials(Request $request)
    {
        // todo
    }

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

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

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

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

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

    public function supportsRememberMe()
    {
        // todo
    }
}
```

Hey! I know this class! It's that same, big, adorable empty authenticator we saw earlier. To tell Symfony to use this, open

config/packages/security.yaml and add the new class under authenticators:

```
58 lines | config/packages/security.yaml
1  security:
   ... lines 2 - 16
17     firewalls:
   ... lines 18 - 20
21         main:
   ... lines 22 - 23
24             guard:
25                 authenticators:
   ... line 26
27                     - App\Security\ApiTokenAuthenticator
   ... lines 28 - 58
```

If you're using that newer, fancier version of this command, it already did this for you. Lucky you!

As *soon* as we do this, the supports() method will be called at the beginning of *every* request. But... refresh. Woh! Big error!

> Because you have multiple guard authenticators, you need to set the "guard.entry_point" key to one of your authenticators.

What is an Entry Point?

If you did *not* see this error, it's your lucky day! Well, really, it's because the newer make:auth command took care of this step for you! But, it *is* important to understand. Move back to security.yaml and, under guard, make sure you have key called entry_point. Your make:auth command probably added it for you. If not, add it, copy the LoginFormAuthenticator class and paste:

```
61 lines | config/packages/security.yaml
1  security:
   ... lines 2 - 16
17     firewalls:
   ... lines 18 - 20
21         main:
   ... lines 22 - 23
24             guard:
   ... lines 25 - 28
29                 # redirect anonymous users to the login page
30                 entry_point: App\Security\LoginFormAuthenticator
   ... lines 31 - 61
```

So... what the heck is an entry point anyways? Your firewall has exactly one "entry point" and its job is simple: to determine what should happen when an anonymous user tries to access a protected page. So far, if we, for example, went to /admin/comment without being logged in, our "entry point" has been redirecting users to /login.

But, where does that entry point code live? Actually, it's inside our LoginFormAuthenticator! Ok, really, it's in the parent class. Hold Command or Ctrl and click to open AbstractFormLoginAuthenticator.

Every authenticator has a method called start() and *it* is the entry point. *This* is the method that Symfony calls when an anonymous user tries to access a protected page. And, no surprises: it redirects you to the login page.

Nice! Except... there's a slight problem: while you can have as *many* authenticators as you want for a firewall, you can only have *one* entry point. Why? Think about it: when an anonymous user tries to access a protected page, well, they're not using any of our authenticators yet: it's just an anonymous user sending *no* authentication info. So, Symfony doesn't know *which* of your authenticators it should use as the entry point. That's why we need to tell it *specifically* which authenticator's start() method to use.

In our app, we will *always* redirect anonymous users to the login form. Of course, if you want to make this logic smarter, you could override the start() method in LoginFormAuthenticator and make it do different things under different conditions. Like, maybe you return an API response instead of redirecting if the URL starts with /api.

Anyways, when we refresh now, it works just like we expect: it redirects us to /login. Log back in with password engage and.... awesome! We're back!

Time to start filling in our authenticator!

Chapter 27: API Token Authenticator

Time to put some code in our ApiTokenAuthenticator! Woo! I'm going to use Postman to help make test API requests. The only thing *better* than using Postman is creating functional tests in your own app. But that's the topic for another tutorial.

Let's make a GET request to http://localhost:8000/api/account. Next, how should we send the API token? As a query parameter? As a header? Well, you can do whatever you want - but using a header is pretty standard. Great! And um... what should we call that header? Postman has a nice system to help configure common authentication types. Choose something called "Bearer token". I'll show you what that means in a minute.

But first, move over to your terminal: we need to find a valid API key! Run:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM api_token'
```

Authorization: Bearer

Copy one of these long strings, move back to Postman and paste! To see what this Auth stuff does, hit "Preview Request".

> Request headers were successfully updated.

Cool! Click back to "Headers". Ahh! This "Auth" section is just a shortcut to add a request header called Authorization. Hey! Go away tooltip! Anyways, the Authorization header is set to the word "Bearer", a space, and then our token.

Honestly, you can name this header *whatever* you want - like SEND-ME-YOUR-TOKEN, WHATS-THE-MAGIC-WORD or I-LIKE-DINOSAURS. The name Authorization is just a standard, yea, and I guess... it *does* sound a bit more professional than my other ideas. There's also nothing significant about that "Bearer" part. That's *another* standard that's commonly used when your token is what's known as a "Bearer token": a fancy term that means whoever "bears" this token - so, whoever "possesses" this token - can use it to authenticate, without needing to provide *any* other types of authentication, like a master key or a password. Most API tokens, also known as "access tokens" are "bearer" tokens. And this is a standard way of attaching them to a request.

supports()

Back to work! Open ApiTokenAuthenticator. Ok: this is our *second* authenticator, so it's time to use our existing knowledge to kick some security butt! For supports(), our authenticator should only become active if the request has an Authorization header whose value starts with the word "Bearer". No problem: return $request->headers->has('Authorization') to make sure that header is set and also check that 0 is the position inside $request->headers->get('Authorization') where the string Bearer and a space appears:

59 lines | src/Security/ApiTokenAuthenticator.php

```
... lines 1 - 11
12 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
13 {
14     public function supports(Request $request)
15     {
16         // look for header "Authorization: Bearer <token>"
17         return $request->headers->has('Authorization')
18             && 0 === strpos($request->headers->get('Authorization'), 'Bearer ');
19     }
... lines 20 - 57
58 }
```

I know: weird-looking code. But it does exactly what we need! If the Authorization Bearer header isn't there, supports() will return false and no other methods will be called.

getCredentials()

Next: getCredentials(). Our job is to read the token string and return it. Start with $authorizationHeader = $request->headers->get('Authorization'):

59 lines | src/Security/ApiTokenAuthenticator.php

```
... lines 1 - 11
12 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
13 {
... lines 14 - 20
21     public function getCredentials(Request $request)
22     {
23         $authorizationHeader = $request->headers->get('Authorization');
... lines 24 - 26
27     }
... lines 28 - 57
58 }
```

But, instead of returning that *whole* value, skip the Bearer part. So, return a sub-string of $authorizationHeader where we start at the 7th character:

59 lines | src/Security/ApiTokenAuthenticator.php

```
... lines 1 - 11
12 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
13 {
... lines 14 - 20
21     public function getCredentials(Request $request)
22     {
23         $authorizationHeader = $request->headers->get('Authorization');
24
25         // skip beyond "Bearer "
26         return substr($authorizationHeader, 7);
27     }
... lines 28 - 57
58 }
```

Ok. Deep breath: let's see if this is working so far. In getUser(), dump($credentials) and die:

59 lines | src/Security/ApiTokenAuthenticator.php

```
... lines 1 - 11
12 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
13 {
... lines 14 - 28
29     public function getUser($credentials, UserProviderInterface $userProvider)
30     {
31         dump($credentials);die;
32     }
... lines 33 - 57
58 }
```

This *should* be the API token *string*. Oh, and notice that this is different than LoginFormAuthenticator: we returned an *array* from getCredentials() there:

```
89 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
... lines 22 - 43
44     public function getCredentials(Request $request)
45     {
46         $credentials = [
47             'email' => $request->request->get('email'),
48             'password' => $request->request->get('password'),
49             'csrf_token' => $request->request->get('_csrf_token'),
50         ];
... lines 51 - 56
57         return $credentials;
58     }
... lines 59 - 87
88 }
```

But that's the beauty of the authenticators: you can return *whatever* you want from getCredentials(). The only thing we need is the token string... so, we just return that.

Try it! Find Postman and... send! Nice! I mean, it looks terrible, but go to Preview. Yes! *There* is our API token string.

getUser()

Next up: getUser(). First, we need to query for the ApiToken entity. At the top of this class, make an __construct function and give it an ApiTokenRepository $apiTokenRepo argument. I'll hit Alt+Enter to initialize that:

```
75 lines | src/Security/ApiTokenAuthenticator.php
... lines 1 - 4
5 use App\Repository\ApiTokenRepository;
... lines 6 - 12
13 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
14 {
15     private $apiTokenRepo;
16
17     public function __construct(ApiTokenRepository $apiTokenRepo)
18     {
19         $this->apiTokenRepo = $apiTokenRepo;
20     }
... lines 21 - 73
74 }
```

Then, back in getUser(), get that token: $token = $this->apiTokenRepo->findOneBy() to query where the token property is set to the $credentials string:

75 lines | src/Security/ApiTokenAuthenticator.php

```
... lines 1 - 12
13 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
14 {
... lines 15 - 36
37     public function getUser($credentials, UserProviderInterface $userProvider)
38     {
39         $token = $this->apiTokenRepo->findOneBy([
40             'token' => $credentials
41         ]);
... lines 42 - 47
48     }
... lines 49 - 73
74 }
```

If we do *not* find an ApiToken, return null. That will make authentication fail. If we *do* find one, we need to return the User, not the token. So, return $token->getUser():

75 lines | src/Security/ApiTokenAuthenticator.php

```
... lines 1 - 12
13 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
14 {
... lines 15 - 36
37     public function getUser($credentials, UserProviderInterface $userProvider)
38     {
39         $token = $this->apiTokenRepo->findOneBy([
40             'token' => $credentials
41         ]);
42
43         if (!$token) {
44             return;
45         }
46
47         return $token->getUser();
48     }
... lines 49 - 73
74 }
```

Finally, *if* you return a User object from getUser(), Symfony calls checkCredentials(). Let's dd('checking credentials') to see if we *continue* to be lucky:

75 lines | src/Security/ApiTokenAuthenticator.php

```
... lines 1 - 12
13 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
14 {
... lines 15 - 49
50     public function checkCredentials($credentials, UserInterface $user)
51     {
52         dd('checking credentials');
53     }
... lines 54 - 73
74 }
```

Move back over to Postman, Send and... yes! Checking credentials.

We're *almost* done! But before we handle success, I want to see what happens with a *bad* API key. And learn how we can

send back the *perfect* error response.

Chapter 28: API Token Authenticator Part 2!

When the request sends us a *valid* API token, our authenticator code is working! At least all the way to checkCredentials(). But before we finish that, I want to see what happens if a client sends us a *bad* key. So let's see... the last number in the token is six. Let's add a space: that will be enough to mess things up.

Hit send again. Woh! It redirects us to /login? I did *not* see that coming.

Sometimes the *hardest* part of security is figuring out what's happening when something unexpected occurs. So, let's figure out *exactly* what's going on here.

When authentication fails, this onAuthenticationFailure() method is called:

```
75 lines | src/Security/ApiTokenAuthenticator.php
... lines 1 - 12
13 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
14 {
... lines 15 - 54
55     public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
56     {
57         // todo
58     }
... lines 59 - 73
74 }
```

Our job is to return a Response that should be sent back to the client. Right now... we're doing nothing.! So, instead of sending an error back to the user, the request *continues* like normal to the controller. But, the request is still *anonymous*. So when it hits our security check in AccountController, Symfony activates the "entry point", which redirects the user to /login.

onAuthenticationFailure()

But... that's not what we want at all! If an API client sends a bad API token, we need to tell them! Bad API client! Let's return a new JsonResponse() with a message key that describes what went wrong. Earlier, I mentioned that whenever authentication fails - for any reason - it's because, internally, some sort of AuthenticationException is thrown. That's important because this exception is passed to us as an argument:

```
75 lines | src/Security/ApiTokenAuthenticator.php
... lines 1 - 12
13 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
14 {
... lines 15 - 54
55     public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
56     {
... line 57
58     }
... lines 59 - 73
74 }
```

And it has a method - getMessageKey() - that holds a message about what went wrong. Set the status code to 401:

```
78 lines | src/Security/ApiTokenAuthenticator.php
... lines 1 - 5
use Symfony\Component\HttpFoundation\JsonResponse;
... lines 7 - 13
class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 16 - 55
    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        return new JsonResponse([
            'message' => $exception->getMessageKey()
        ], 401);
    }
... lines 62 - 76
}
```

Custom Error Messages with CustomUserMessageAuthenticationException

Let's try it again! Send the request. Yes! A 401 Unauthorized response. But, oh. That message isn't right at all!

> Username could not be found?

This is because Symfony creates a different error message based on *where* authentication fails inside your authenticator. If you fail to return a User from getUser(), you get this "Username could not be found" error.

For our login form, we render this *exact* messageKey field in the template. But we *also* pass it through the translator:

```
37 lines | templates/security/login.html.twig
... lines 1 - 10
{% block body %}
    <form class="form-signin" method="post">
        {% if error %}
            <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
        {% endif %}
... lines 16 - 34
    </form>
{% endblock %}
```

That allowed us to translate that into a better message:

```
1 lines | translations/security.en.yaml
"Username could not be found.": "Oh no! It doesn't look like that email exists!"
```

We *could* do the same here: inject the translator service into ApiTokenAuthenticator and translate the message key. But... hmmm, the message *still* wouldn't be right - it would use the "It doesn't look like that email exists!" message from the translation file.

No problem: there is a *second* way to control error messages in an authenticator, and it's *super* flexible. At *any* point in your authenticator, you can throw a new CustomUserMessageAuthenticationException() that will cause authentication to fail *and* accepts *any* custom error message you want, like, "Invalid API Token":

```
81 lines | src/Security/ApiTokenAuthenticator.php
... lines 1 - 9
10 use Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationException;
... lines 11 - 14
15 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
16 {
... lines 17 - 38
39     public function getUser($credentials, UserProviderInterface $userProvider)
40     {
... lines 41 - 44
45         if (!$token) {
46             throw new CustomUserMessageAuthenticationException(
47                 'Invalid API Token'
48             );
49         }
... lines 50 - 51
52     }
... lines 53 - 79
80 }
```

That's it! This exception will be passed to onAuthenticationFailure() and its getMessageKey() method will return that message.

Go back to Postman to try it: send! We got it! So much better!

Checking Token Expiration

Oh, while we're talking about tokens *failing*, we should *definitely* check to make sure the token hasn't expired. Inside ApiToken, we created this nice expiresAt property:

```
62 lines | src/Entity/ApiToken.php
... lines 1 - 9
10 class ApiToken
11 {
... lines 12 - 23
24     /**
25      * @ORM\Column(type="datetime")
26      */
27     private $expiresAt;
... lines 28 - 60
61 }
```

Go down to the bottom of the class and add a new helper function: isExpired() that returns a bool. Return $this->getExpiresAt() is less than or equal to new \DateTime():

```
67 lines | src/Entity/ApiToken.php
... lines 1 - 9
10 class ApiToken
11 {
... lines 12 - 61
62     public function isExpired(): bool
63     {
64         return $this->getExpiresAt() <= new \DateTime();
65     }
66 }
```

Nice! Back in ApiTokenAuthenticator, in getUser(), if $token->isExpired(), then

throw new CustomUserMessageAuthenticationException() with Token Expired:

```
87 lines | src/Security/ApiTokenAuthenticator.php
... lines 1 - 9
10 use Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationException;
... lines 11 - 14
15 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
16 {
... lines 17 - 38
39     public function getUser($credentials, UserProviderInterface $userProvider)
40     {
... lines 41 - 44
45         if (!$token) {
46             throw new CustomUserMessageAuthenticationException(
47                 'Invalid API Token'
48             );
49         }
50
51         if ($token->isExpired()) {
52             throw new CustomUserMessageAuthenticationException(
53                 'Token expired'
54             );
55         }
... lines 56 - 57
58     }
... lines 59 - 85
86 }
```

We're killin' it! Oh, but, why are we putting this code *here* and not in checkCredentials()? Answer: no reason! These two methods are called one after the other and you can *really* put any code inside *either* of these methods. Actually, I chose getUser() just because we have access to the $token object there.

Head back to Postman. Let's remove that extra space so our API token is valid once again. Send! Success! Now, go back to the ApiToken class and, temporarily, return true from isExpired() so we can see the error:

```
class ApiToken
{
    // ...

    public function isExpired(): bool
    {
        return true;
        return $this->getExpiresAt() <= new \DateTime();
    }
}
```

And... send it again! Got it! Token Expired. Remove that dummy code.

onAuthenticationSuccess()

At this point... we're basically done! In checkCredentials(), there is no password to check. And so, it's perfectly ok for us to return true:

```
87 lines | src/Security/ApiTokenAuthenticator.php
... lines 1 - 14
15 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
16 {
... lines 17 - 59
60     public function checkCredentials($credentials, UserInterface $user)
61     {
62         return true;
63     }
... lines 64 - 85
86 }
```

Finally, in onAuthenticationSuccess(), hmm. What *should* we do when authentication is successful? With a login form, we redirect the user after success. But with an API token system we, well, want to do... nothing! Yep! We want to allow the request to continue so that it can hit the controller and return the JSON response:

```
87 lines | src/Security/ApiTokenAuthenticator.php
... lines 1 - 14
15 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
16 {
... lines 17 - 71
72     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
73     {
74         // allow the authentication to continue
75     }
... lines 76 - 85
86 }
```

start() & supportsRememberMe()

So what about start()? Because we chose LoginFormAuthenticator as the entry_point, this will never be called. To prove it, I'll throw an exception that says:

> Not used: entry_point from other authenticator is used:

```
87 lines | src/Security/ApiTokenAuthenticator.php
... lines 1 - 14
15 class ApiTokenAuthenticator extends AbstractGuardAuthenticator
16 {
... lines 17 - 76
77     public function start(Request $request, AuthenticationException $authException = null)
78     {
79         throw new \Exception('Not used: entry_point from other authentication is used');
80     }
... lines 81 - 85
86 }
```

And, *finally*, supportsRememberMe(). Return false:

87 lines | src/Security/ApiTokenAuthenticator.php

```
... lines 1 - 14
class ApiTokenAuthenticator extends AbstractGuardAuthenticator
{
... lines 17 - 81
    public function supportsRememberMe()
    {
        return false;
    }
}
```

If you return true from this method, it just means that the "remember me" system is activated and looking for that _remember_me checkbox to be checked. Because that makes absolutely *no* sense for an API, just turn it off.

That's it! Find a stranger to high-five! Cheers your coffee with a co-worker! And find Postman! Brace yourself... send! Yes! It executes our controller and we are *definitely* authenticated because we see the info for spacebar9@example.com.

People - we now have *two* valid ways to authenticate in our system! The *super* cool thing is that, inside of our controller, we don't care *which* method is used! We just say $this->getUser()... never caring whether the user was authenticated via the login form or with an API token.

Next: let's set up a registration form and learn how we can *manually* authenticate the user after success.

Chapter 29: Manual Authentication / Registration

Hey! You've made it through almost this *entire* tutorial! Nice work! I have just a *few* more tricks to show you before we're done - and they're good ones!

Creating the Registration Form

First, I want to create a registration form. Find your code and open SecurityController. In addition to login and logout, add a new public function register():

44 lines | src/Controller/SecurityController.php

```
... lines 1 - 8
9  class SecurityController extends AbstractController
10 {
... lines 11 - 38
39     public function register()
40     {
... line 41
42     }
43 }
```

Give it a route - /register and a name: app_register:

44 lines | src/Controller/SecurityController.php

```
... lines 1 - 8
9  class SecurityController extends AbstractController
10 {
... lines 11 - 35
36     /**
37      * @Route("/register", name="app_register")
38      */
39     public function register()
40     {
... line 41
42     }
43 }
```

Here's the interesting thing about registration. It has *nothing* to do with security! Think about it. What *is* registration? It's just a form that creates a new record in the User table. That's it! That's just database stuff.

So then... why are we even *talking* about this in a security tutorial? Well... to create the *best* user experience, there will be just a *little* bit of security right at the end. Because, after registration, I want to instantly authenticate the new user.

More on that later. Right now, render a template: $this->render('security/register.html.twig'):

44 lines | src/Controller/SecurityController.php

```
... lines 1 - 8
class SecurityController extends AbstractController
{
... lines 11 - 35
    /**
     * @Route("/register", name="app_register")
     */
    public function register()
    {
        return $this->render('security/register.html.twig');
    }
}
```

Then... I'll cheat: in security/, copy the login.html.twig template, paste and call it register.html.twig:

43 lines | templates/security/register.html.twig

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

{% block title %}Login!{% endblock %}

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

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

{% block body %}
<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <form class="form-signin" method="post">
                {% if error %}
                    <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
                {% endif %}

                <h1 class="h3 mb-3 font-weight-normal">Please sign in</h1>
                <label for="inputEmail" class="sr-only">Email address</label>
                <input type="email" value="{{ last_username }}" name="email" id="inputEmail" class="form-control" placeholder="Email addr
                <label for="inputPassword" class="sr-only">Password</label>
                <input type="password" name="password" id="inputPassword" class="form-control" placeholder="Password" required>

                <input type="hidden" name="_csrf_token"
                    value="{{ csrf_token('authenticate') }}"
                >

                <div class="checkbox mb-3">
                    <label>
                        <input type="checkbox" name="_remember_me"> Remember me
                    </label>
                </div>
                <button class="btn btn-lg btn-primary btn-block" type="submit">
                    Sign in
                </button>
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

Let's see: change the title, delete the authentication error stuff and I am going to add a little comment here that says that we should replace this with a Symfony form later:

36 lines | templates/security/register.html.twig

```
... lines 1 - 2
{% block title %}Register!{% endblock %}
... lines 4 - 10
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-sm-12">
            {# todo - replace with a Symfony form! #}
            <form class="form-signin" method="post">
... lines 17 - 30
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

We haven't talked about the form system yet, so I don't want to use it here. But, normally, I *would* use the form system because it handles validation and automatically adds CSRF protection.

But, to show off how to manually authenticate a user after registration, this HTML form will work *beautifully*. Change the h1, remove the value= on the email field so that it always starts blank and take out the CSRF token:

36 lines | templates/security/register.html.twig

```
... lines 1 - 2
{% block title %}Register!{% endblock %}
... lines 4 - 10
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-sm-12">
            {# todo - replace with a Symfony form! #}
            <form class="form-signin" method="post">
                <h1 class="h3 mb-3 font-weight-normal">Register</h1>
                <label for="inputEmail" class="sr-only">Email address</label>
                <input type="email" name="email" id="inputEmail" class="form-control" placeholder="Email address" required autofocus>
                <label for="inputPassword" class="sr-only">Password</label>
                <input type="password" name="password" id="inputPassword" class="form-control" placeholder="Password" required>
... lines 22 - 30
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

We *do* need CSRF protection on this form... but I'll skip it for now, because we'll refactor this into a Symfony form in a future tutorial.

And finally, hijack the "remember me" checkbox and turn it into a terms box. We'll say:

> Agree to terms I for sure read

36 lines | templates/security/register.html.twig

```
... lines 1 - 2
{% block title %}Register!{% endblock %}
... lines 4 - 10
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-sm-12">
            {# todo - replace with a Symfony form! #}
            <form class="form-signin" method="post">
                <h1 class="h3 mb-3 font-weight-normal">Register</h1>
                <label for="inputEmail" class="sr-only">Email address</label>
                <input type="email" name="email" id="inputEmail" class="form-control" placeholder="Email address" required autofocus>
                <label for="inputPassword" class="sr-only">Password</label>
                <input type="password" name="password" id="inputPassword" class="form-control" placeholder="Password" required>

                <div class="checkbox mb-3">
                    <label>
                        <input type="checkbox" name="_remember_me" required> Agree to terms I for sure read
                    </label>
                </div>
... lines 28 - 30
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

Oh, and update the button: Register:

36 lines | templates/security/register.html.twig

```
... lines 1 - 2
{% block title %}Register!{% endblock %}
... lines 4 - 10
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-sm-12">
            {# todo - replace with a Symfony form! #}
            <form class="form-signin" method="post">
                <h1 class="h3 mb-3 font-weight-normal">Register</h1>
                <label for="inputEmail" class="sr-only">Email address</label>
                <input type="email" name="email" id="inputEmail" class="form-control" placeholder="Email address" required autofocus>
                <label for="inputPassword" class="sr-only">Password</label>
                <input type="password" name="password" id="inputPassword" class="form-control" placeholder="Password" required>

                <div class="checkbox mb-3">
                    <label>
                        <input type="checkbox" name="_remember_me" required> Agree to terms I for sure read
                    </label>
                </div>
                <button class="btn btn-lg btn-primary btn-block" type="submit">
                    Register
                </button>
            </form>
        </div>
    </div>
</div>
{% endblock %}
```

Let's see how it looks! Move over, go to /register and... got it! Logout, then move back over and open up base.html.twig. Scroll down just a little bit to find the "Login" link. Let's create a second link that points to the new app_register route. Say, "Register":

86 lines | templates/base.html.twig

```
1  <!doctype html>
2  <html lang="en">
   ... lines 3 - 15
16     <body>
   ... lines 17 - 22
23         <nav class="navbar navbar-expand-lg navbar-dark navbar-bg mb-5">
   ... lines 24 - 27
28             <div class="collapse navbar-collapse" id="navbarNavDropdown">
   ... lines 29 - 40
41                 <ul class="navbar-nav ml-auto">
42                     {% if is_granted('ROLE_USER') %}
   ... lines 43 - 58
59                         <li class="nav-item">
60                             <a style="color: #fff;" class="nav-link" href="{{ path('app_register') }}">Register</a>
61                         </li>
62                     {% endif %}
63                 </ul>
64             </div>
65         </nav>
   ... lines 66 - 83
84     </body>
85 </html>
```

Move back and check it out. Not bad!

Handing the Registration Submit

Just like with the login form, because there is no action= on the form, this will submit right back to the same URL. But, *unlike* login, because this is just a normal page, we *are* going to handle that submit logic right inside of the controller.

First, get the Request object by adding an argument with the Request type hint: the one from HttpFoundation. Below, I'm going to add *another* reminder to use the Symfony form & validation system later:

58 lines | src/Controller/SecurityController.php

```
   ... lines 1 - 11
12 class SecurityController extends AbstractController
13 {
   ... lines 14 - 41
42     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
43     {
44         // TODO - use Symfony forms & validation
   ... lines 45 - 55
56     }
57 }
```

Then, to only process the data when the form is being submitted, add if ($request->isMethod('POST')):

```
58 lines | src/Controller/SecurityController.php
... lines 1 - 11
12 class SecurityController extends AbstractController
13 {
... lines 14 - 41
42     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
43     {
44         // TODO - use Symfony forms & validation
45         if ($request->isMethod('POST')) {
... lines 46 - 52
53         }
... lines 54 - 55
56     }
57 }
```

Inside... our job is simple! Registration is nothing more than a mechanism to create a new User object. So $user = new User(). Then set some data on it: $user->setEmail($request->request->get('email')):

```
58 lines | src/Controller/SecurityController.php
... lines 1 - 11
12 class SecurityController extends AbstractController
13 {
... lines 14 - 41
42     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
43     {
44         // TODO - use Symfony forms & validation
45         if ($request->isMethod('POST')) {
46             $user = new User();
47             $user->setEmail($request->request->get('email'));
... lines 48 - 52
53         }
... lines 54 - 55
56     }
57 }
```

Remember $request->request is the way that you get $_POST data. And, the *names* of the fields on our form are name="email" and name="password". But before we handle the password, add $user->setFirstName(). This field is required in the database... but, we don't *actually* have that field on the form. Just use Mystery for now:

58 lines | src/Controller/SecurityController.php

```
... lines 1 - 11
12 class SecurityController extends AbstractController
13 {
... lines 14 - 41
42     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
43     {
44         // TODO - use Symfony forms & validation
45         if ($request->isMethod('POST')) {
46             $user = new User();
47             $user->setEmail($request->request->get('email'));
48             $user->setFirstName('Mystery');
... lines 49 - 52
53         }
... lines 54 - 55
56     }
57 }
```

In a real app, I would either add this field to the registration form, or make it nullable in the database, so it's optional.

Finally, let's set the password. But... of course! We are never ever, ever, ever going to save the *plain* password. We need to encode it. We already did this inside of UserFixture:

60 lines | src/DataFixtures/UserFixture.php

```
... lines 1 - 7
8  use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
9
10 class UserFixture extends BaseFixture
11 {
12     private $passwordEncoder;
13
14     public function __construct(UserPasswordEncoderInterface $passwordEncoder)
15     {
16         $this->passwordEncoder = $passwordEncoder;
17     }
18
19     protected function loadData(ObjectManager $manager)
20     {
21         $this->createMany(10, 'main_users', function($i) use ($manager) {
... lines 22 - 29
30             $user->setPassword($this->passwordEncoder->encodePassword(
31                 $user,
32                 'engage'
33             ));
... lines 34 - 40
41         });
42
43         $this->createMany(3, 'admin_users', function($i) {
... lines 44 - 48
49             $user->setPassword($this->passwordEncoder->encodePassword(
50                 $user,
51                 'engage'
52             ));
... lines 53 - 54
55         });
... lines 56 - 57
58     }
59 }
```

Ah yes, the key was the UserPasswordEncoderInterface service. In our controller, add another argument: UserPasswordEncoderInterface $passwordEncoder:

58 lines | src/Controller/SecurityController.php

```
... lines 1 - 8
9  use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
... lines 10 - 11
12 class SecurityController extends AbstractController
13 {
... lines 14 - 41
42     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
43     {
... lines 44 - 55
56     }
57 }
```

Below, we can say $passwordEncoder->encodePassword(). This needs the User object and the plain password that was just submitted: $request->request->get('password'):

58 lines | src/Controller/SecurityController.php

```
... lines 1 - 11
12 class SecurityController extends AbstractController
13 {
... lines 14 - 41
42     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
43     {
44         // TODO - use Symfony forms & validation
45         if ($request->isMethod('POST')) {
... lines 46 - 48
49             $user->setPassword($passwordEncoder->encodePassword(
50                 $user,
51                 $request->request->get('password')
52             ));
53         }
... lines 54 - 55
56     }
57 }
```

We are ready to save! Get the entity manager with $em = $this->getDoctrine()->getManager(). Then, $em->persist($user) and $em->flush():

64 lines | src/Controller/SecurityController.php

```
... lines 1 - 11
12 class SecurityController extends AbstractController
13 {
... lines 14 - 41
42     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
43     {
44         // TODO - use Symfony forms & validation
45         if ($request->isMethod('POST')) {
... lines 46 - 48
49             $user->setPassword($passwordEncoder->encodePassword(
50                 $user,
51                 $request->request->get('password')
52             ));
53
54             $em = $this->getDoctrine()->getManager();
55             $em->persist($user);
56             $em->flush();
... lines 57 - 58
59         }
... lines 60 - 61
62     }
63 }
```

All delightfully boring code. This looks a lot like what we're doing in our fixtures.

Finally, after *any* successful form submit, we always redirect. Use return $this->redirectToRoute(). This is the shortcut method that we were looking at earlier. Redirect to the account page: app_account:

```
64 lines | src/Controller/SecurityController.php
... lines 1 - 11
12 class SecurityController extends AbstractController
13 {
... lines 14 - 41
42     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
43     {
44         // TODO - use Symfony forms & validation
45         if ($request->isMethod('POST')) {
... lines 46 - 48
49             $user->setPassword($passwordEncoder->encodePassword(
50                 $user,
51                 $request->request->get('password')
52             ));
53
54             $em = $this->getDoctrine()->getManager();
55             $em->persist($user);
56             $em->flush();
57
58             return $this->redirectToRoute('app_account');
59         }
... lines 60 - 61
62     }
63 }
```

Awesome! Let's give this thing a spin! I'll register as ryan@symfonycasts.com, password engage. Agree to the terms that I for sure read and... Register! Bah! That smells like a Ryan mistake! Yep! Use $this->getDoctrine()->getManager():

```
64 lines | src/Controller/SecurityController.php
... lines 1 - 11
12 class SecurityController extends AbstractController
13 {
... lines 14 - 41
42     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
43     {
44         // TODO - use Symfony forms & validation
45         if ($request->isMethod('POST')) {
... lines 46 - 53
54             $em = $this->getDoctrine()->getManager();
... lines 55 - 58
59         }
... lines 60 - 61
62     }
63 }
```

That's what I meant to do.

Move over and try this again: ryan@symfonycasts.com, password engage, agree to the terms that I read and... Register!

Authentication after Registration

Um... what? We're on the *login* form? What happened? First, according to the web debug toolbar, we are still anonymous. That makes sense: we *registered*, but we did *not* login. After registration, we were redirected to /account...

```
64 lines | src/Controller/SecurityController.php
... lines 1 - 11
12 class SecurityController extends AbstractController
13 {
... lines 14 - 41
42     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
43     {
44         // TODO - use Symfony forms & validation
45         if ($request->isMethod('POST')) {
... lines 46 - 57
58             return $this->redirectToRoute('app_account');
59         }
... lines 60 - 61
62     }
63 }
```

But because we are *not* logged in, that sent us here.

This is *not* the flow that I want my users to experience. Nope, as *soon* as the user registers, I want to log them in automatically.

Oh, and there's also *another* problem. Open LoginFormAuthenticator and find onAuthenticationSuccess():

```
89 lines | src/Security/LoginFormAuthenticator.php
... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
... lines 22 - 74
75     public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
76     {
77         if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
78             return new RedirectResponse($targetPath);
79         }
80
81         return new RedirectResponse($this->router->generate('app_homepage'));
82     }
... lines 83 - 87
88 }
```

We added some extra code here to make sure that if the user went to, for example, /admin/comment as an anonymous user, then, after they log in, they would be sent *back* to /admin/comment.

And... hey! I want that *same* behavior for my registration form! Imagine that you're building a store. As an anonymous user, I add some things to my cart and finally go to /checkout. But because /checkout requires me to be logged in, I'm sent to the login form. And because I don't have an account yet, I instead click to register and fill out that form. After submitting, where should I be taken to? That's easy! I should *definitely* be taken *back* to /checkout so I can continue what I was doing!

These two problems - the fact that we want to automatically authenticate the user after registration *and* redirect them intelligently - can be solved at the same time! After we save the User to the database, we're basically going to tell Symfony to use our LoginFormAuthenticator class to authenticate the user and redirect by using its onAuthenticationSuccess() method.

Check it out: add two arguments to our controller. First, a service called GuardAuthenticatorHandler $guardHandler. Second, the authenticator that you want to authenticate through: LoginFormAuthenticator $formAuthenticator:

```
71 lines | src/Controller/SecurityController.php
... lines 1 - 5
6  use App\Security\LoginFormAuthenticator;
... lines 7 - 10
11 use Symfony\Component\Security\Guard\GuardAuthenticatorHandler;
... lines 12 - 13
14 class SecurityController extends AbstractController
15 {
... lines 16 - 43
44     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guard
45     {
... lines 46 - 68
69     }
70 }
```

Once we have those two things, instead of redirecting to a normal route use return $guardHandler->authenticateUserAndHandleSuccess():

```
71 lines | src/Controller/SecurityController.php
... lines 1 - 13
14 class SecurityController extends AbstractController
15 {
... lines 16 - 43
44     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guard
45     {
46         // TODO - use Symfony forms & validation
47         if ($request->isMethod('POST')) {
... lines 48 - 59
60             return $guardHandler->authenticateUserAndHandleSuccess(
... lines 61 - 64
65             );
66         }
... lines 67 - 68
69     }
70 }
```

This needs a few arguments: the $user that's being logged in, the $request object, the authenticator - $formAuthenticator and the "provider key". That's just the name of your firewall: main:

```
71 lines | src/Controller/SecurityController.php
... lines 1 - 13
14 class SecurityController extends AbstractController
15 {
... lines 16 - 43
44     public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guard
45     {
46         // TODO - use Symfony forms & validation
47         if ($request->isMethod('POST')) {
... lines 48 - 59
60             return $guardHandler->authenticateUserAndHandleSuccess(
61                 $user,
62                 $request,
63                 $formAuthenticator,
64                 'main'
65             );
66         }
... lines 67 - 68
69     }
70 }
```

Cool! Let's try it! Click back to register. This time, make sure that you register as a different user, password engage, agree to the terms, submit and... nice! We're authenticated *and* sent to the correct place.

Next - we're going to start talking about a *very* important and *very* fun feature called "voters". Voters are *the* way to make more *complex* access decisions, like, determining that a User can edit *this* Article because they are its author, but not an Article created by someone else.

Chapter 30: Author ManyToOne Relation to User

Check out the homepage: every Article has an author. But, open the Article entity. Oh: the author property is just a *string*!

249 lines | src/Entity/Article.php

```
... lines 1 - 15
16 class Article
17 {
... lines 18 - 47
48     /**
49      * @ORM\Column(type="string", length=255)
50      */
51     private $author;
... lines 52 - 247
248 }
```

When we originally created this field, we hadn't learned how to handle database relationships yet.

But now that we are *way* more awesome than "past us", let's replace this author string property with a proper relation to the User entity. So every Article will be "authored" by a specific User.

Wait... why are we talking about database relationship in the security tutorial? Am I wandering off-topic again? Well, only a *little*. Setting up database relations is *always* good practice. But, I have a *real*, dubious, security-related goal: this setup will lead us to some *really* interesting access control problems - like denying access to edit an Article unless the logged in user is that Article's *author*.

Let's smash this relationship stuff so we can get to that goodness! First, remove the author property entirely. Find the getter and setter methods and remove those too. Now, find your terminal and run:

```
$ php bin/console make:migration
```

If our app were already deployed, we might need to be a little bit more careful so that we don't *lose* all this original author data. But, for us, no worries: that author data was garbage! Find the Migrations/ directory, open up the new migration file and yep! ALTER TABLE Article DROP author:

29 lines | src/Migrations/Version20180901184240.php

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

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20180901184240 extends AbstractMigration
{
    public function up(Schema $schema) : void
    {
        // 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 \'mys

        $this->addSql('ALTER TABLE article DROP author');
    }

    public function down(Schema $schema) : void
    {
        // 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 \'mys

        $this->addSql('ALTER TABLE article ADD author VARCHAR(255) NOT NULL COLLATE utf8mb4_unicode_ci');
    }
}
```

Adding the Relation

Now, lets *re-add* author as a relation:

```
$ php bin/console make:entity
```

Update the Article entity and add a new author property. This will be a "relation" to the User entity. For the type, it's another ManyToOne relation: each Article has one User and each User can have many articles. The author property will be *required*, so make it *not* nullable. We'll say "yes" to mapping the other side of the relationship and I'll say "no" to orphanRemoval, though, that's not important. Cool! Hit enter to finish:

250 lines | src/Entity/Article.php

```php
... lines 1 - 15
class Article
{
... lines 18 - 68
    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\User", inversedBy="articles")
     * @ORM\JoinColumn(nullable=false)
     */
    private $author;
... lines 74 - 237
    public function getAuthor(): ?User
    {
        return $this->author;
    }

    public function setAuthor(?User $author): self
    {
        $this->author = $author;

        return $this;
    }
}
```

Now run:

```
$ php bin/console make:migration
```

Like always, let's go check out the new migration:

33 lines | src/Migrations/Version20180901184346.php

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

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20180901184346 extends AbstractMigration
{
    public function up(Schema $schema) : void
    {
        // 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 \'mys

        $this->addSql('ALTER TABLE article ADD author_id INT NOT NULL');
        $this->addSql('ALTER TABLE article ADD CONSTRAINT FK_23A0E66F675F31B FOREIGN KEY (author_id) REFERENCES use
        $this->addSql('CREATE INDEX IDX_23A0E66F675F31B ON article (author_id)');
    }

    public function down(Schema $schema) : void
    {
        // 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 \'mys

        $this->addSql('ALTER TABLE article DROP FOREIGN KEY FK_23A0E66F675F31B');
        $this->addSql('DROP INDEX IDX_23A0E66F675F31B ON article');
        $this->addSql('ALTER TABLE article DROP author_id');
    }
}
```

Woh! I made a mistake! It *is* adding author_id but it is *also* dropping author. But that column should already be gone by now! My bad! After generating the *first* migration, I forgot to run it! This diff contains *too* many changes. Delete it. Then, execute the first migration:

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

Bye bye original author column. *Now* run:

```
$ php bin/console make:migration
```

Go check it out:

33 lines | src/Migrations/Version20180901184346.php

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

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20180901184346 extends AbstractMigration
{
    public function up(Schema $schema) : void
    {
        // 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 \'mys

        $this->addSql('ALTER TABLE article ADD author_id INT NOT NULL');
        $this->addSql('ALTER TABLE article ADD CONSTRAINT FK_23A0E66F675F31B FOREIGN KEY (author_id) REFERENCES use
        $this->addSql('CREATE INDEX IDX_23A0E66F675F31B ON article (author_id)');
    }

    public function down(Schema $schema) : void
    {
        // 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 \'mys

        $this->addSql('ALTER TABLE article DROP FOREIGN KEY FK_23A0E66F675F31B');
        $this->addSql('DROP INDEX IDX_23A0E66F675F31B ON article');
        $this->addSql('ALTER TABLE article DROP author_id');
    }
}
```

Much better: it adds the author_id column and foreign key constraint. Close that and, once again, run:

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

Failed Migration!

Woh! It explodes! Bad luck! This is one of those *tricky* migrations. We made the new column required... but that field will be *empty* for all the existing rows in the table. That's not a problem on its own... but it *does* cause a problem when the migration tries to add the foreign key! The fix depends on your situation. If our app were already deployed to production, we would need to follow a 3-step process. First, make the property nullable=true at first and generate that migration. Second, run a script or query that can somehow set the author_id for all the existing articles. And finally, change the property to nullable=false and generate one last migration.

But because our app has *not* been deployed yet... we can cheat. First, drop *all* of the tables in the database with:

```
$ php bin/console doctrine:schema:drop --full-database --force
```

Then, re-run all the migrations to make sure they're working:

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

Awesome! Because the article table is empty, no errors.

Adding Article Author Fixtures

Now that the database is ready, open ArticleFixtures. Ok: this simple setAuthor() call will *not* work anymore:

83 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 10
11 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
12 {
... lines 13 - 24
25     private static $articleAuthors = [
26         'Mike Ferengi',
27         'Amy Oort',
28     ];
... line 29
30     protected function loadData(ObjectManager $manager)
31     {
32         $this->createMany(10, 'main_articles', function($count) use ($manager) {
... lines 33 - 59
60             $article->setAuthor($this->faker->randomElement(self::$articleAuthors))
... lines 61 - 62
63             ;
... lines 64 - 70
71         });
... lines 72 - 73
74     }
... lines 75 - 81
82 }
```

Nope, we need to relate this to one of the users from UserFixture. Remember we have two groups: these main_users and these admin_users:

60 lines | src/DataFixtures/UserFixture.php

```
... lines 1 - 9
10 class UserFixture extends BaseFixture
11 {
... lines 12 - 18
19     protected function loadData(ObjectManager $manager)
20     {
21         $this->createMany(10, 'main_users', function($i) use ($manager) {
... lines 22 - 40
41         });
42
43         $this->createMany(3, 'admin_users', function($i) {
... lines 44 - 54
55         });
... lines 56 - 57
58     }
59 }
```

Let's allow normal users to be the author of an Article. In other words, use $this->getRandomReference('main_users') to get a *random* User object from that group:

```
79 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 10
11 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
12 {
... lines 13 - 24
25     protected function loadData(ObjectManager $manager)
26     {
27         $this->createMany(10, 'main_articles', function($count) use ($manager) {
... lines 28 - 54
55             $article->setAuthor($this->getRandomReference('main_users'))
... lines 56 - 57
58             ;
... lines 59 - 65
66         });
... lines 67 - 68
69     }
... lines 70 - 77
78 }
```

At the top of the class, I can remove this old static property.

Try it! Move over and run:

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

It works! But... only by chance. UserFixture was executed before ArticleFixtures... and that's important! It would *not* work the other way around. We just got lucky. To enforce this ordering, at the bottom of ArticleFixtures, in getDependencies(), add UserFixture::class:

```
79 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 10
11 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
12 {
... lines 13 - 70
71     public function getDependencies()
72     {
73         return [
... line 74
75             UserFixture::class,
76         ];
77     }
78 }
```

Now UserFixture will *definitely* run before ArticleFixtures.

If you try the fixtures again:

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

Same result. But now, it's guaranteed!

Next - let's finish our refactoring and create a new "Article Edit" page!

Chapter 31: Article Admin & Low-Level Access Controls

Each Article's author is now a proper relationship to the User entity, instead of a string. That's great... except that we haven't updated anything else yet in our code to reflect this. Refresh the homepage. Yep! A big ol' error:

> Exception thrown rendering the template Catchable Fatal Error: Object of Class Proxies__CG__\App\Entity\User cannot be converted to string.

Wow! Two important things here. First, whenever you see this "Proxies" thing, ignore it. This is an internal object that Doctrine sometimes wraps around your entity in order to enable some of its lazy-loading relation awesomeness. The object looks and works *exactly* like User.

Second, the error itself basically means that something is trying to convert our User object into a string. This makes sense: in our template, we're just rendering {{ article.author }}:

65 lines | templates/article/homepage.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">
... lines 10 - 20
                {% for article in articles %}
                <div class="article-container my-1">
                    <a href="{{ path('article_show', {slug: article.slug}) }}">
... line 24
                        <div class="article-title d-inline-block pl-3 align-middle">
... lines 26 - 34
                            <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('images/alien-pro
... line 36
                        </div>
                    </a>
                </div>
                {% endfor %}
            </div>
... lines 42 - 61
        </div>
    </div>
{% endblock %}
```

That *was* a string before, but now it's a User object.

We *could* go change this to article.author.firstName. *Or*, we can go into our User class and add a public function __toString() method. Return $this->getFirstName():

```
246 lines | src/Entity/User.php
... lines 1 - 13
14 class User implements UserInterface
15 {
... lines 16 - 240
241     public function __toString()
242     {
243         return $this->getFirstName();
244     }
245 }
```

As *soon* as we do that... we're back!

Adding the Edit Endpoint

What I *really* want to talk about is controlling access in your system on an *object-by-object* basis. Like, User A can edit *this* Article because they are the author, but not that *other* Article. Open ArticleAdminController and add a new endpoint: public function edit():

```
39 lines | src/Controller/ArticleAdminController.php
... lines 1 - 14
15 class ArticleAdminController extends AbstractController
16 {
... lines 17 - 33
34     public function edit(Article $article)
35     {
... line 36
37     }
38 }
```

Add the normal route with a URL of /admin/article/{id}/edit. I won't give it a name yet:

```
39 lines | src/Controller/ArticleAdminController.php
... lines 1 - 14
15 class ArticleAdminController extends AbstractController
16 {
... lines 17 - 30
31     /**
32      * @Route("/admin/article/{id}/edit")
33      */
34     public function edit(Article $article)
35     {
... line 36
37     }
38 }
```

Next, add an argument to the method: Article $article:

```
39 lines | src/Controller/ArticleAdminController.php
... lines 1 - 4
5  use App\Entity\Article;
... lines 6 - 14
15 class ArticleAdminController extends AbstractController
16 {
... lines 17 - 33
34     public function edit(Article $article)
35     {
... line 36
37     }
38 }
```

Because Article is an *entity*, SensioFrameworkExtraBundle - a bundle we installed a long time ago - will use the {id} route parameter to query for the correct Article.

To see if this is working, dd($article):

```
39 lines | src/Controller/ArticleAdminController.php
... lines 1 - 4
5  use App\Entity\Article;
... lines 6 - 14
15 class ArticleAdminController extends AbstractController
16 {
... lines 17 - 33
34     public function edit(Article $article)
35     {
36         dd($article);
37     }
38 }
```

Oh, and remember: this *entire* controller class is protected by ROLE_ADMIN_ARTICLE:

```
39 lines | src/Controller/ArticleAdminController.php
... lines 1 - 11
12 /**
13  * @IsGranted("ROLE_ADMIN_ARTICLE")
14  */
15 class ArticleAdminController extends AbstractController
16 {
... lines 17 - 37
38 }
```

To get a valid Article ID, find your terminal and run:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM article'
```

Ok - we'll use 20. Fly over to you browser and... hit it: /admin/article/20/edit. That bounces us to the login page. Use an admin user: admin2@thespacebar.com password engage.

Perfect! We're back on the Article edit page, access is *granted* and Doctrine queried for the Article object.

Planning the Access Controls

And *this* is where things get interesting. I want to *continue* to require ROLE_ADMIN_ARTICLE to be able to go to the *new* article page. But, down here, if you're *editing* an article, I want to allow access if you have ROLE_ADMIN_ARTICLE *or* if you

are the *author* of this Article. This is the *first* time that we've had to make an access decision that is *based* on an object - the Article.

Manually Denying Access

Start by moving @IsGranted() from above the class to above the new() method:

37 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 10

class ArticleAdminController extends AbstractController
{
    /**
... line 15
     * @IsGranted("ROLE_ADMIN_ARTICLE")
     */
    public function new(EntityManagerInterface $em)
    {
... lines 20 - 26
    }
... lines 28 - 35
}
```

Thanks to this, our edit() endpoint is temporarily open to the world.

Right now, we're looking at article id 20. Go back to your terminal. Ok, this article's author is user 18. Find out who that is:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM user WHERE id = 18'
```

Ok, cool: the author is spacebar4@example.com. Go back to the browser, go to the login page, and log in as this user: spacebar4@example.com, password engage.

Perfect! We still have access but... well... *anyone* has access to this page right now.

The *simplest* way to enforce our custom security logic is to add it *right* in the controller. Check it out: if ($article->getAuthor() !== $this->getUser()) and if !$this->isGranted('ROLE_ADMIN_ARTICLE'), then throw $this->createAccessDeniedException('No access!'):

41 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 11
class ArticleAdminController extends AbstractController
{
... lines 14 - 31
    public function edit(Article $article)
    {
        if ($article->getAuthor() != $this->getUser() && !$this->isGranted('ROLE_ADMIN_ARTICLE')) {
            throw $this->createAccessDeniedException('No access!');
        }

        dd($article);
    }
}
```

The $this->isGranted() method is new to us, but simple: it returns true or false based on whether or not the user has ROLE_ADMIN_ARTICLE. We also haven't seen this createAccessDeniedException() method yet either. Up until now, we've denied access using $this->denyAccessUnlessGranted(). It turns out, that method is just a shortcut to call $this->isGranted() and then throw $this->createAccessDeniedException() if that returned false. The cool takeaway is that, the way you

ultimately deny access in Symfony is by throwing a special exception object that this method creates. Oh, and the message - No access! - that's only shown to developers.

Let's try it! Reload the page. We *totally* get access because we *are* the author of this article. Mission accomplished, right? Well... no! This sucks! I don't want this important logic to live in my controller. Why not? What if I need to re-use this somewhere else? Duplicating security logic is a bad idea. And, what if I need to use it in Twig to hide or show an edit link? That would *really* be ugly.

Nope, there's a better way: a wonderful system called voters.

Chapter 32: Voters

We need to centralize this logic so that it can be reused in other places:

41 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 11
12 class ArticleAdminController extends AbstractController
13 {
... lines 14 - 31
32     public function edit(Article $article)
33     {
34         if ($article->getAuthor() != $this->getUser() && !$this->isGranted('ROLE_ADMIN_ARTICLE')) {
35             throw $this->createAccessDeniedException('No access!');
36         }
... lines 37 - 38
39     }
40 }
```

How? Well... it may look a bit weird at first. Remove all of this logic and replace it with:
if (!$this->isGranted('MANAGE', $article)):

41 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 11
12 class ArticleAdminController extends AbstractController
13 {
... lines 14 - 31
32     public function edit(Article $article)
33     {
34         if (!$this->isGranted('MANAGE', $article)) {
... line 35
36         }
... lines 37 - 38
39     }
40 }
```

Hmm. I'm using the same isGranted() function as before. But instead of passing a *role*, I'm just "inventing" a string: MANAGE. It *also* turns out that isGranted() has an optional *second* argument: a piece of *data* that is relevant to making this access decision.

Don't worry - this will *not* magically work somehow. If you try it... yep!

> Access denied.

Hello Voter System

Let me explain what's happening. *Whenever* you call isGranted, or one of the other functions like denyAccessUnlessGranted(), Symfony executes what's known as the "Voter system". Basically, it takes the string - MANAGE, or ROLE_ADMIN_ARTICLE - and it asks each voter:

> Hey voter! Do you know how to decide whether or not the current user has this string - ROLE_ADMIN_ARTICLE or MANAGE?

In the core of Symfony, there are basically two voters by default: RoleVoter and AuthenticatedVoter. When you pass *any* string that starts with ROLE_, the RoleVoter says:

> Ah, yea! I totally know how to determine if the user should have access!

Then, it checks to see if the User has that role and returns true or false. The other voter "abstains" - which means it doesn't vote - and so access is entirely granted or denied by that one voter.

When you pass any string that starts with IS_AUTHENTICATED_, like IS_AUTHENTICATED_FULLY, the *other* voters says:

> Oh. This is me! I know how to check this!

And it returns true or false based on *how* authenticated the user is and which of those three IS_AUTHENTICATED_ strings we passed.

Adding our Custom Voter

The *really* cool thing is that we can add our *own* custom voters. Right now, when we call isGranted() with the string MANAGE, both voters say:

> Hmm, no, we don't understand what this is

They both "abstain" from voting. And when nobody votes, access is denied by default. So our goal is clear: introduce a *new* voter that understands how to handle the string MANAGE and an Article object. By the way, up until now, I've been calling this MANAGE string a role... because it has usually started with ROLE_. But actually, it's generally called a "permission attribute". Some permission attributes are roles, but some are other strings handled by other voters.

Oh, and why did I choose the word MANAGE? I just made that up. If you need different permissions for edit, show and delete, you would use different attributes for each - like EDIT, SHOW, DELETE - and create a voter that can handle all of those. You'll see soon. My case is simpler: I'll use MANAGE for *any* operation on an Article - for example, for editing, deleting or publishing it.

Ok, let's *finally* create our voter!

Chapter 33: Adding a Custom Voter

Time to create our new Voter class! To do it... we can cheat! Find your terminal and run:

```
$ php bin/console make:voter
```

Call it ArticleVoter. It's pretty common to have *one* voter per object that you need to decide access for. Let's go check it out src/Security/Voter/ArticleVoter.php:

42 lines | src/Security/Voter/ArticleVoter.php

```
... lines 1 - 2
namespace App\Security\Voter;

use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;
use Symfony\Component\Security\Core\User\UserInterface;

class ArticleVoter extends Voter
{
    protected function supports($attribute, $subject)
    {
        // replace with your own logic
        // https://symfony.com/doc/current/security/voters.html
        return in_array($attribute, ['EDIT', 'VIEW'])
            && $subject instanceof App\Entity\BlogPost;
    }

    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
    {
        $user = $token->getUser();
        // if the user is anonymous, do not grant access
        if (!$user instanceof UserInterface) {
            return false;
        }

        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case 'EDIT':
                // logic to determine if the user can EDIT
                // return true or false
                break;
            case 'VIEW':
                // logic to determine if the user can VIEW
                // return true or false
                break;
        }

        return false;
    }
}
```

supports()

Nice! Voters are a bit simpler than authenticators: just two methods. Here's how it works: whenever anybody in the system calls isGranted() with *any* permission attribute string, the supports() method on your voter will be called:

42 lines | src/Security/Voter/ArticleVoter.php

```
... lines 1 - 8
class ArticleVoter extends Voter
{
    protected function supports($attribute, $subject)
    {
        // replace with your own logic
        // https://symfony.com/doc/current/security/voters.html
        return in_array($attribute, ['EDIT', 'VIEW'])
            && $subject instanceof App\Entity\BlogPost;
    }
... lines 18 - 40
}
```

It's *our* job to decide whether or not our voter knows how to vote.

The $attribute argument will be the *string* passed to isGranted() and $subject is the *second* argument - the Article object for us. The example in the generated code is actually pretty good. Let's say that our voter knows how to vote if the $attribute is MANAGE and if the $subject is an instanceOf Article:

43 lines | src/Security/Voter/ArticleVoter.php

```
... lines 1 - 4
use App\Entity\Article;
... lines 6 - 9
class ArticleVoter extends Voter
{
    protected function supports($attribute, $subject)
    {
        // replace with your own logic
        // https://symfony.com/doc/current/security/voters.html
        return in_array($attribute, ['MANAGE'])
            && $subject instanceof Article;
    }
... lines 19 - 41
}
```

If we return false from supports, nothing happens: Our ArticleVoter doesn't vote and it's up to some *other* voter to handle things. But if we return true, Symfony immediately calls voteOnAttribute():

```
42 lines | src/Security/Voter/ArticleVoter.php
... lines 1 - 8
class ArticleVoter extends Voter
{
... lines 11 - 18
    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
    {
        $user = $token->getUser();
        // if the user is anonymous, do not grant access
        if (!$user instanceof UserInterface) {
            return false;
        }

        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case 'EDIT':
                // logic to determine if the user can EDIT
                // return true or false
                break;
            case 'VIEW':
                // logic to determine if the user can VIEW
                // return true or false
                break;
        }

        return false;
    }
}
```

This is where our logic goes to determine access. If we return true, access will be granted. If we return false, access will be denied.

voteOnAttribute()

Symfony passes us the same $attribute and $subject, as well as something called the $token:

```
42 lines | src/Security/Voter/ArticleVoter.php
... lines 1 - 4
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
... lines 6 - 8
class ArticleVoter extends Voter
{
... lines 11 - 18
    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
    {
        $user = $token->getUser();
... lines 22 - 39
    }
}
```

The token is a lower-level object that you don't see *too* often. But, you can use it to get access to the User object:

```
42 lines | src/Security/Voter/ArticleVoter.php
... lines 1 - 4
5  use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
... line 6
7  use Symfony\Component\Security\Core\User\UserInterface;
8
9  class ArticleVoter extends Voter
10 {
... lines 11 - 18
19     protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
20     {
21         $user = $token->getUser();
22         // if the user is anonymous, do not grant access
23         if (!$user instanceof UserInterface) {
24             return false;
25         }
... lines 26 - 39
40     }
41 }
```

I'm going to start in this method by helping my editor. At the top, add /** @var Article $subject */ to say that the $subject variable is an Article object:

```
43 lines | src/Security/Voter/ArticleVoter.php
... lines 1 - 9
10 class ArticleVoter extends Voter
11 {
... lines 12 - 19
20     protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
21     {
22         /** @var Article $subject */
... lines 23 - 40
41     }
42 }
```

We can safely do this because of the supports() method:

```
43 lines | src/Security/Voter/ArticleVoter.php
... lines 1 - 9
10 class ArticleVoter extends Voter
11 {
12     protected function supports($attribute, $subject)
13     {
14         // replace with your own logic
15         // https://symfony.com/doc/current/security/voters.html
16         return in_array($attribute, ['MANAGE'])
17             && $subject instanceof Article;
18     }
... lines 19 - 41
42 }
```

$subject will *definitely* be an Article at this point.

Below this, it's pretty common to have a voter that votes on *multiple* attributes, like EDIT and DELETE. We don't need it, but I'll keep the switch case statement. Our only case is MANAGE:

```
43 lines | src/Security/Voter/ArticleVoter.php
... lines 1 - 9
10 class ArticleVoter extends Voter
11 {
... lines 12 - 19
20     protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
21     {
22         /** @var Article $subject */
23         $user = $token->getUser();
24         // if the user is anonymous, do not grant access
25         if (!$user instanceof UserInterface) {
26             return false;
27         }
28
29         // ... (check conditions and return true to grant permission) ...
30         switch ($attribute) {
31             case 'MANAGE':
... lines 32 - 36
37                 break;
38         }
... lines 39 - 40
41     }
42 }
```

Excellent! It's time to shine. First, if $subject->getAuthor() == $user then return true:

```
43 lines | src/Security/Voter/ArticleVoter.php
... lines 1 - 9
10 class ArticleVoter extends Voter
11 {
... lines 12 - 19
20     protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
21     {
... lines 22 - 28
29         // ... (check conditions and return true to grant permission) ...
30         switch ($attribute) {
31             case 'MANAGE':
32                 // this is the author!
33                 if ($subject->getAuthor() == $user) {
34                     return true;
35                 }
36
37                 break;
38         }
39
40         return false;
41     }
42 }
```

The current user is the author and so access *should* be granted.

Checking for Roles inside a Voter

If they are *not* the author, we need to check for ROLE_ADMIN_ARTICLE. But, hmm. We know how to check if a User has a role in a controller: $this->isGranted():

```
41 lines | src/Controller/ArticleAdminController.php
... lines 1 - 11
12 class ArticleAdminController extends AbstractController
13 {
... lines 14 - 31
32     public function edit(Article $article)
33     {
34         if (!$this->isGranted('MANAGE', $article)) {
... line 35
36         }
... lines 37 - 38
39     }
40 }
```

But, how can we check that from inside of a voter? Or, from inside any service?

The answer is.... with the Security service! We actually *already* know this service! Add a public function __construct() method with a new Security argument: the one from the Symfony component. I'll hit Alt+Enter and select "Initialize Fields" to create that property and set it:

```
55 lines | src/Security/Voter/ArticleVoter.php
... lines 1 - 7
8 use Symfony\Component\Security\Core\Security;
... lines 9 - 10
11 class ArticleVoter extends Voter
12 {
13     private $security;
14
15     public function __construct(Security $security)
16     {
17         $this->security = $security;
18     }
... lines 19 - 53
54 }
```

Do you remember *where* we used this service before? It was inside MarkdownHelper: it's the last argument *way* over here:

```
50 lines | src/Service/MarkdownHelper.php
... lines 1 - 9
10 class MarkdownHelper
11 {
... lines 12 - 16
17     private $security;
18
19     public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $markdownLogger, bool $is
20     {
... lines 21 - 24
25         $this->security = $security;
26     }
... lines 27 - 48
49 }
```

We used it because it gives us access to the current User object:

50 lines | src/Service/MarkdownHelper.php

```
... lines 1 - 9
class MarkdownHelper
{
... lines 12 - 27
    public function parse(string $source): string
    {
        if (stripos($source, 'bacon') !== false) {
            $this->logger->info('They are talking about bacon again!', [
                'user' => $this->security->getUser()
            ]);
        }
... lines 35 - 47
    }
}
```

But, there's one *other* thing that the Security class can do. Hold Command or Ctrl and click to open it. It has a getUser() method but it *also* has an isGranted() method! Awesome! The Security service is the *key* to get the User *or* check if the user has access for some permission attribute.

Back down in our voter logic, it's now very simple: if $this->security->isGranted('ROLE_ADMIN_ARTICLE'), then return true. At the bottom, instead of break, return false: if both of these conditions are *not* met, access denied:

55 lines | src/Security/Voter/ArticleVoter.php

```
... lines 1 - 10
class ArticleVoter extends Voter
{
... lines 13 - 27
    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
    {
... lines 30 - 36
        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case 'MANAGE':
                // this is the author!
                if ($subject->getAuthor() == $user) {
                    return true;
                }

                if ($this->security->isGranted('ROLE_ADMIN_ARTICLE')) {
                    return true;
                }

                return false;
        }

        return false;
    }
}
```

Ok, let's try this! Move over, refresh and... access granted! Symfony calls the supports() method, that returns true, and because we're logged in as the author, access is granted. Comment out the author check real quick:

```
// src/Security/Voter/ArticleVoter.php

class ArticleVoter extends Voter
{
    // ...
    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
    {
        switch ($attribute) {
            case 'MANAGE':
                // this is the author!
                if ($subject->getAuthor() == $user) {
                    //return true;
                }
                // ...
        }

        return false;
    }
}
```

Try it again. Access denied! Put that back.

@IsGranted with a Subject

Voters are *great*. And using them to centralize this kind of logic will keep your security code solid. But, there's *one* small thing that now seems *impossible* to do. First, open ArticleAdminController. We can actually shorten this to the normal $this->denyAccessUnlessGranted('MANAGE', $article):

39 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 11
12 class ArticleAdminController extends AbstractController
13 {
... lines 14 - 31
32     public function edit(Article $article)
33     {
34         $this->denyAccessUnlessGranted('MANAGE', $article);
35
36         dd($article);
37     }
38 }
```

Try it - reload the page. Access granted! This does the *exact* same thing as before. But... what about using the @IsGranted() annotation?

39 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 6
7  use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
... lines 8 - 11
12 class ArticleAdminController extends AbstractController
13 {
14     /**
... line 15
16      * @IsGranted("ROLE_ADMIN_ARTICLE")
17      */
18     public function new(EntityManagerInterface $em)
19     {
... lines 20 - 26
27     }
... lines 28 - 37
38 }
```

Hmm... now there's a problem: can we use the annotation and still, somehow, pass in the Article object? Actually, yes!

Add @IsGranted(), pass it MANAGE and then a second argument: subject="article":

38 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 11
12 class ArticleAdminController extends AbstractController
13 {
... lines 14 - 28
29     /**
... line 30
31      * @IsGranted("MANAGE", subject="article")
32      */
33     public function edit(Article $article)
34     {
... line 35
36     }
37 }
```

That's it! When you use subject=, you're allowed to pass this the same name as any of the *arguments* to your controller. This only works because we used the feature that automatically queries for the Article object and passes it as an argument. These two features combine *perfectly*. But, if you're ever in a situation where your "subject" isn't a controller argument, no worries, just use the normal denyAccessUnlessGranted() code. But, remove it in this case:

38 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 11
12 class ArticleAdminController extends AbstractController
13 {
... lines 14 - 28
29     /**
... line 30
31      * @IsGranted("MANAGE", subject="article")
32      */
33     public function edit(Article $article)
34     {
35         dd($article);
36     }
37 }
```

Let's... try it! Access granted! That was too easy. Go back to the voter and comment-out the author check again - let's *really* make sure this is working:

```
// src/Security/Voter/ArticleVoter.php

class ArticleVoter extends Voter
{
    // ...
    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
    {
        switch ($attribute) {
            case 'MANAGE':
                // this is the author!
                if ($subject->getAuthor() == $user) {
                    //return true;
                }
                // ...
        }

        return false;
    }
}
```

Now... yes! Access denied! Go put that code back.

Oh my gosh friends, we did it! We *killed* this tutorial! We have a great authentication system that allows both login form authentication *and* API authentication! We have a rich dynamic roles system and a voter system where we can control access with *any* custom rules. Oh, I love security! I hope you guys are feeling empowered to create your simple, complex, crazy, whatever authentication system you need. As always, if you have questions, ask us down in the comments.

Alright people, seeya next time!