

Symfony 5 Security: Authenticators



Chapter 1: composer require security

Welcome back friends! I'm so happy that you've stumbled into my Symfony 5 security tutorial for a bunch of reasons. The first is that well... uh... the site that we've been building has NO security... and the raptors are starting to jiggle the door handles.

The *other* reason is that, once we make it to the maintenance shed on other side of the compound, we're going to explore Symfony's new security system, called the "authenticator" system. Ooh. If you've used the old system, you'll feel right at home. If you're new to Symfony security, you chose a great time to start. The new system is easier to learn and understand... but it's also more powerful.

Project Setup


And because the security system isn't going to come online by itself, let's get to work. To learn how to authenticate & authorize & do other cool security stuff at a *pro* level, you should definitely download the course code from this page and code along with me. Making real-world mistakes.... yeah, it's the best way to remember this stuff.

After unzipping the file, you'll find a `start/` directory with the same code that you see here. Pop open the `README.md` file for all the setup instructions. The last step will be to find a terminal, move into the project and start a web server. I'm going to use the `symfony` binary for this:



```
symfony serve -d
```

This starts up a new server at <https://127.0.0.1:8000>. Open that in your browser... or be lazy and run:



```
symfony open:local
```

to... "delegate" the work to someone else. Say hello to Cauldron Overflow! A question and answer site for witches and wizards, who... unfortunately... keep casting their spells live on production *first* without testing them... and usually on a Friday afternoon. Sheesh. Then they come here to ask how to undo the damage.

Installing Security.

Because Symfony's philosophy is to start small and then allow you to install the stuff you need later, right now our app... literally does *not* have a security system.

That's no fun, so let's install one! Go back to your terminal and run:

A terminal window with a dark blue header bar containing three white dots. The main area is light gray and contains the text 'composer require security' in a monospaced font.

```
composer require security
```

This installs Symfony's security bundle. After it finishes... run:

A terminal window with a dark blue header bar containing three white dots. The main area is light gray and contains the text 'git status' in a monospaced font.

```
git status
```

to see what its recipe did. In addition to the normal stuff, it added one new configuration file: `security.yaml`. Let's go check that out: `config/packages/security.yaml`:

config/packages/security.yaml

```
1 security:
2     # https://symfony.com/doc/current/security/authenticator_manager.html
3     enable_authenticator_manager: true
4     # https://symfony.com/doc/current/security.html#c-hashing-passwords
5     password_hashers:
6
7     Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
8         'auto'
9     # https://symfony.com/doc/current/security.html#where-do-users-come-
10    from-user-providers
11    providers:
12        users_in_memory: { memory: null }
13    firewalls:
14        dev:
15            pattern: ^/(_(profiler|wdt)|css|images|js)/
16            security: false
17        main:
18            lazy: true
19            provider: users_in_memory
20
21            # activate different ways to authenticate
22            # https://symfony.com/doc/current/security.html#firewalls-
23    authentication
24
25            #
26    https://symfony.com/doc/current/security/impersonating_user.html
27            # switch_user: true
28
29    # Easy way to control access for large sections of your site
30    # Note: Only the *first* access control that matches will be used
31    access_control:
32        # - { path: ^/admin, roles: ROLE_ADMIN }
33        # - { path: ^/profile, roles: ROLE_USER }
```

As you hopefully guessed by its name, this file powers the security system! By the time we're done, each section in here will be simple and boring to you. I love when programming stuff is boring.

enable_authenticator_manager

Oh, but see this `enable_authenticator_manager` key?

```
config/packages/security.yaml
```

```
1 security:
2     # https://symfony.com/doc/current/security/authenticator_manager.html
3     enable_authenticator_manager: true
↕ // ... lines 4 - 29
```

In Symfony 5.3 - the version I'm using - the old and new security systems live side-by-side and you get to *choose* which one you want! When you set `enable_authenticator_manager` to `true`, you are activating the *new* system. Yay! Shiny! If you're working on a legacy project and need to learn the *old* system, check out our [Symfony 4 Security](#) tutorial. It's pretty cool too!

Authentication & Authorization

Anyways, when you talk about security, there are two big parts: authentication and authorization. Authentication asks the question, "who are you"? And "can you prove it?" Users, login forms, remember me cookies, passwords, API keys... all of that stuff is related to authentication.

Authorization asks a different question: "Should you have access to this resource?" Authorization doesn't care much about *who* you are... it's all about allowing or denying access to different things, like different URLs or controllers.

In Symfony, or really in *any* security system, authentication is the tricky part. I mean, just think about how many *ways* there are to authenticate! Login forms, API token authentication, social authentication with OAuth, SSO's, LDAP, putting on a fake mustache and walking confidently passed a security guard. I mean... the possibilities are endless. But I *also* think that authentication is *super* fun.

So next: let's start on our journey into the new shiny authenticator system by creating the most basic part of authentication: a user class.

Chapter 2: make:user

No matter how your users authenticate - a login form, social authentication, or an API key - your security system needs some concept of a user: some class that describes the "thing" that is logged in.

Yup, step 1 of authentication is to create a `User` class. And there's a command that can help us! Find your terminal and run:



```
symfony console make:user
```

As a reminder, `symfony console` is just a shortcut for `bin/console`... but because I'm using the Docker integration with the Symfony web server, calling `symfony console` allows the `symfony` binary to inject some environment variables that point to the Docker database. It won't matter for this command, but it *will* matter for any command that talks to the database.

Ok, question one:

"The name of the user class"

Typically, this will be `User`... though it *would* be cooler to use something like `HumanoidEntity`. If the "thing" that logs into your site would be better called a `Company` or `University` or `Machine`, use that name here.

"Do you want to store user data in the database via Doctrine?"

For us: that's a definite yes... but it's not a requirement. Your user data might be stored on some other server... though even in that case, it's often convenient to store some *extra* data in your local database... in which case you would *also* say yes here.

Next:

"Enter a property name that will be the unique display name for the user."

I'm going to use `email`. This is not *that* important, and I'll explain how it's used in a few minutes. Finally:

“Will this app need to hash and check user passwords?”

You only need to say yes if it will be *your* application's responsibility to check the user's password when they log in. We *are* going to do this... but I'm going to say "no". We'll add it manually a bit later.

Hit enter and... done!

The User Class & Entity

Okay. What did this do? First, it created a `User` entity and a `UserRepository`... the exact same stuff you normally get from running `make:entity`. Let's go check out that new `User` class: `src/Entity/User.php`:

src/Entity/User.php

```
↕ // ... lines 1 - 2
3 namespace App\Entity;
4
5 use App\Repository\UserRepository;
6 use Doctrine\ORM\Mapping as ORM;
7 use
8     Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
9 use Symfony\Component\Security\Core\User\UserInterface;
10
11 /**
12  * @ORM\Entity(repositoryClass=UserRepository::class)
13  */
14 class User implements UserInterface
15 {
16     /**
17      * @ORM\Id
18      * @ORM\GeneratedValue
19      * @ORM\Column(type="integer")
20      */
21     private $id;
22
23     /**
24      * @ORM\Column(type="string", length=180, unique=true)
25      */
26     private $email;
27
28     /**
29      * @ORM\Column(type="json")
30      */
31     private $roles = [];
32
33     // ... lines 31 - 113
34 }
114
```

First and foremost, this is a normal boring Doctrine entity: it has annotations - or maybe PHP 8 attributes for you - and an id. It is... just an entity: there is nothing special about it.

UserInterface & Deprecated Methods

The *only* thing that Symfony cares about is that your user class implements `UserInterface`. Hold `Command` or `Ctrl` and click to jump way into the core code to see this.

This interface *really* has just 3 methods: `getUserIdentifier()`, which you see documented above the interface, `getRoles()` ... and another one way down here called

`eraseCredentials()`. If you're confused about why I'm skipping all of these *other* methods, it's because they're *deprecated*. In Symfony 6, this interface will *only* have those 3: `getUserIdentifier()`, `getRoles()` and `eraseCredentials()`.

In our `User` class, if you scroll down, the `make:user` command implemented *all* of this for us. Thanks to how we answered one of its questions, `getUserIdentifier()` returns the email:

```
src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface
14 {
↕ // ... lines 15 - 48
49     /**
50      * A visual identifier that represents this user.
51      *
52      * @see UserInterface
53      */
54     public function getUserIdentifier(): string
55     {
56         return (string) $this->email;
57     }
↕ // ... lines 58 - 113
114 }
```

This... isn't *too* important: it's mostly just a visual representation of your User object... it's used in the web debug toolbar... and in a few optional systems, like the "remember me" system.

If you're using Symfony 5 like I am, you'll notice that the deprecated methods *are* still generated. They're needed *just* for backwards compatibility, and you can delete them once you're on Symfony 6.

The `getRoles()` method deals with permissions:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface
14 {
↕ // ... lines 15 - 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 - 113
114 }
```

more on that later. And then `getPassword()` and `getSalt()` are both deprecated:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface
14 {
↕ // ... lines 15 - 85
86     /**
87      * This method can be removed in Symfony 6.0 - is not needed for apps
      that do not check user passwords.
88      *
89      * @see PasswordAuthenticatedUserInterface
90      */
91     public function getPassword(): ?string
92     {
93         return null;
94     }
95
96     /**
97      * This method can be removed in Symfony 6.0 - is not needed for apps
      that do not check user passwords.
98      *
99      * @see UserInterface
100     */
101     public function getSalt(): ?string
102     {
103         return null;
104     }
↕ // ... lines 105 - 113
114 }
```

You *will* still need a `getPassword()` method if you check passwords on your site - but we'll learn about that later. Finally, `eraseCredentials()` is part of `UserInterface`:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface
14 {
↕ // ... lines 15 - 105
106     /**
107      * @see UserInterface
108      */
109     public function eraseCredentials()
110     {
111         // If you store any temporary, sensitive data on the user, clear
        it here
112         // $this->plainPassword = null;
113     }
114 }
```

but it's not very important and we'll *also* talk about it later.

So at a high level... if you ignore the deprecated methods... and the not-so-important `eraseCredentials()`, the only thing that our `User` class needs to have is an identifier and a method that returns the array of roles that this user should have. Yup... it's mostly just a `User` entity.

"providers": The User Provider

The `make:user` command also made one tweak to our `security.yaml` file: you can see it right here:

```
config/packages/security.yaml
1  security:
2  // ... lines 2 - 7
8  providers:
9      # used to reload user from session & other features (e.g.
    switch_user)
10     app_user_provider:
11         entity:
12             class: App\Entity\User
13             property: email
14 // ... lines 14 - 33
```

It added what's called a "user provider", which is an object that knows how to load your user objects... whether you're loading that data from an API or from a database. Because we're using Doctrine, we get to use the built-in `entity` provider: it knows how to fetch our users from the database using the `email` property.

I wanted you to see this change... but the user provider isn't important yet. I'll show you *exactly* how and where it's used as we go along.

Next: we have *total* control over how our `User` class looks. The power! So let's add a custom field to it and then load up our database with a nice set of dummy users.

Chapter 3: Customizing the User Class

What's cool about the `User` class is that... it's our class! As long as we implement `UserInterface`, we can add *whatever* else we want:

```
src/Entity/User.php
↕ // ... lines 1 - 7
8 use Symfony\Component\Security\Core\User\UserInterface;
↕ // ... lines 9 - 12
13 class User implements UserInterface
14 {
↕ // ... lines 15 - 113
114 }
```

For example, I'd like to store the first name of my users. So let's go add a property for that. At your terminal, run:

```
symfony console make:entity
```

We'll edit the `User` entity, add a `firstName` property, have it be a string, 255 length... and say "yes" to nullable. Let's make this property optional in the database.

Done! Back over in the `User` class, no surprises! We have a new property... and new getter and setter methods:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface
14 {
↕ // ... lines 15 - 31
32     /**
33      * @ORM\Column(type="string", length=255, nullable=true)
34      */
35     private $firstName;
↕ // ... lines 36 - 119
120     public function getFirstName(): ?string
121     {
122         return $this->firstName;
123     }
124
125     public function setFirstName(string $firstName): self
126     {
127         $this->firstName = $firstName;
128
129         return $this;
130     }
131 }
```

Go generate a migration for our new `User`. At the terminal, run

```
symfony console make:migration
```

Cool! Spin over and open that up to make sure it's not hiding any surprises:

migrations/Version20211001172813.php

```
↕ // ... lines 1 - 2
3 declare(strict_types=1);
4
5 namespace Doctrine\Migrations;
6
7 use Doctrine\DBAL\Schema\Schema;
8 use Doctrine\Migrations\AbstractMigration;
9
10 /**
11  * Auto-generated Migration: Please modify to your needs!
12  */
13 final class Version20211001172813 extends AbstractMigration
14 {
15     public function getDescription(): string
16     {
17         return '';
18     }
19
20     public function up(Schema $schema): void
21     {
22         // this up() migration is auto-generated, please modify it to your
needs
23         $this->addSql('CREATE TABLE user (id INT AUTO_INCREMENT NOT NULL,
email VARCHAR(180) NOT NULL, roles JSON NOT NULL, first_name VARCHAR(255)
DEFAULT NULL, UNIQUE INDEX UNIQ_8D93D649E7927C74 (email), PRIMARY KEY(id))
DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE =
InnoDB');
24     }
25
26     public function down(Schema $schema): void
27     {
28         // this down() migration is auto-generated, please modify it to
your needs
29         $this->addSql('DROP TABLE user');
30     }
31 }
```

Awesome: `CREATE TABLE user` with `id`, `email`, `roles` and `first_name` columns. Close this... and run it:

```
symfony console doctrine:migrations:migrate
```

Success!

Adding User Fixtures

And because the `User` entity is... just a normal Doctrine entity, we can *also* add dummy users to our database using the fixtures system.

Open up `src/DataFixtures/AppFixtures.php`. We're using Foundry to help us load data. So let's create a new Foundry factory for `User`. Since we're having SO much fun running commands in this video, let's sneak in one... or three more:



```
symfony console make:factory
```

Yup! We want one for `User`. Go open it up: `src/Factory/UserFactory.php`:

src/Factory/UserFactory.php

```
↕ // ... lines 1 - 2
3 namespace App\Factory;
4
5 use App\Entity\User;
6 use App\Repository\UserRepository;
7 use Zenstruck\Foundry\RepositoryProxy;
8 use Zenstruck\Foundry\ModelFactory;
9 use Zenstruck\Foundry\Proxy;
↕ // ... lines 10 - 28
29 final class UserFactory extends ModelFactory
30 {
31     public function __construct()
32     {
33         parent::__construct();
34
35         // TODO inject services if required
36         (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#factory-as-services)
37     }
38
39     protected function getDefaults(): array
40     {
41         return [
42             // TODO add your default values here
43             (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#model-factories)
44             'email' => self::faker()->text(),
45             'roles' => [],
46             'firstName' => self::faker()->text(),
47         ];
48     }
49
50     protected function initialize(): self
51     {
52         // see
53         (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#initialize)
54         return $this
55             // ->afterInstantiate(function(User $user) {})
56             ;
57     }
58
59     protected static function getClass(): string
60     {
61         return User::class;
62     }
63 }
```

Our job in `getDefaults()` is to make sure that all of the required properties have good default values. Set `email` to `self::faker()->email()`, I won't set any roles right now and set `firstName` to `self::faker()->firstName()`:

```
src/Factory/UserFactory.php
↕ // ... lines 1 - 28
29 final class UserFactory extends ModelFactory
30 {
↕ // ... lines 31 - 37
38     protected function getDefaults(): array
39     {
40         return [
41             'email' => self::faker()->email(),
42             'firstName' => self::faker()->firstName(),
43         ];
44     }
↕ // ... lines 45 - 57
58 }
```

Cool! Over in `AppFixtures`, at the bottom, create a user: `UserFactory::createOne()`. But use a specific email so we can log in using this later. How about, `abraca_admin@example.com`:

```
src/DataFixtures/AppFixtures.php
↕ // ... lines 1 - 11
12 use App\Factory\UserFactory;
↕ // ... lines 13 - 15
16 class AppFixtures extends Fixture
17 {
18     public function load(ObjectManager $manager)
19     {
↕ // ... lines 20 - 41
42         AnswerFactory::new(function() use ($questions) {
↕ // ... lines 43 - 45
46         })->needsApproval()->many(20)->create();
47
48         UserFactory::createOne(['email' => 'abraca_admin@example.com']);
↕ // ... lines 49 - 51
52     }
53 }
```

Then, to fill out the system a bit, add `UserFactory::createMany(10)`:

src/DataFixtures/AppFixtures.php

```
↕ // ... lines 1 - 11
12 use App\Factory\UserFactory;
↕ // ... lines 13 - 15
16 class AppFixtures extends Fixture
17 {
18     public function load(ObjectManager $manager)
19     {
↕ // ... lines 20 - 47
48         UserFactory::createOne(['email' => 'abraca_admin@example.com']);
49         UserFactory::createMany(10);
↕ // ... lines 50 - 51
52     }
53 }
```

Let's try it! Back at the terminal, run:

```
symfony console doctrine:fixtures:load
```

No errors! Check out the new table:

```
symfony console doctrine:query:sql 'SELECT * FROM user'
```


And... there they are! Now that we have users in the database, we need to add one or more ways for them to authenticate. It's time to build a login form!

Chapter 4: Building a Login Form

There are a lot of ways that you can allow your users to log in... one way being a login form that loads users from the database. That's what we're going to build first.

The easiest way to build a login form system is by running a `symfony console make:auth` command. That will generate everything you need. But since we want to *really* learn security, let's do this step-by-step... mostly by hand.

Before we start thinking about authenticating the user, we *first* need to build a login page, which... if you think about it... has nothing to do with security! It's just a normal Symfony route, controller & template that renders a form. Let's cheat a little to make this. Run:



```
symfony console make:controller
```

Answer `SecurityController`. Cool! Go open up the new class:
`src/Controller/SecurityController.php`:

src/Controller/SecurityController.php

```
↕ // ... lines 1 - 2
3 namespace App\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6 use Symfony\Component\HttpFoundation\Response;
7 use Symfony\Component\Routing\Annotation\Route;
8
9 class SecurityController extends AbstractController
10 {
11     /**
12      * @Route("/security", name="security")
13      */
14     public function index(): Response
15     {
16         return $this->render('security/index.html.twig', [
17             'controller_name' => 'SecurityController',
18         ]);
19     }
20 }
```

Nothing too fancy here. Let's customize this to be a login page: set the URL to `/login`, call the route `app_login` and rename the *method* to `login()`:

src/Controller/SecurityController.php

```
↕ // ... lines 1 - 8
9 class SecurityController extends AbstractController
10 {
11     /**
12      * @Route("/login", name="app_login")
13      */
14     public function login(): Response
15     {
16         // ... line 16
17     }
18 }
```

For the template, call it `security/login.html.twig`... and don't pass any variables right now:

```
src/Controller/SecurityController.php
```

```
↕ // ... lines 1 - 8
9 class SecurityController extends AbstractController
10 {
11     /**
12      * @Route("/login", name="app_login")
13      */
14     public function login(): Response
15     {
16         return $this->render('security/login.html.twig');
17     }
18 }
```

Down in the `templates/` directory, open `templates/security/...` and rename the template to `login.html.twig`:

```
templates/security/login.html.twig
```

```
1 {% extends 'base.html.twig' %}
2
3 {% block title %}Hello SecurityController!{% endblock %}
4
5 {% block body %}
6 <style>
7     .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%;
8     font: 18px/1.5 sans-serif; }
9     .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
10 </style>
11 <div class="example-wrapper">
12     <h1>Hello {{ controller_name }}! ✓</h1>
13
14     This friendly message is coming from:
15     <ul>
16         <li>Your controller at <code><a href="{{
17         '/Users/weaverryan/Sites/knp/knpu-
18         repos/symfony5/src/Controller/SecurityController.php'|file_link(0)
19         }}">src/Controller/SecurityController.php</a></code></li>
20         <li>Your template at <code><a href="{{
21         '/Users/weaverryan/Sites/knp/knpu-
22         repos/symfony5/templates/security/index.html.twig'|file_link(0)
23         }}">templates/security/index.html.twig</a></code></li>
24     </ul>
25 </div>
26 {% endblock %}
```

To get started, I'm going to completely replace this template and paste in a new structure: you can copy this from the code block on this page:

templates/security/login.html.twig

```
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Log In!{% endblock %}
4
5  {% block body %}
6  <div class="container">
7      <div class="row">
8          <div class="login-form bg-light mt-4 p-4">
9              <form method="post" class="row g-3">
10                 <h1 class="h3 mb-3 font-weight-normal">Please sign in</h1>
11
12                 <div class="col-12">
13                     <label for="inputEmail">Email</label>
14                     <input type="email" name="email" id="inputEmail"
class="form-control" required autofocus>
15                 </div>
16                 <div class="col-12">
17                     <label for="inputPassword">Password</label>
18                     <input type="password" name="password"
id="inputPassword" class="form-control" required>
19                 </div>
20
21                 <div class="col-12">
22                     <button class="btn btn-lg btn-primary float-end"
type="submit">
23                         Sign in
24                     </button>
25                 </div>
26             </form>
27         </div>
28     </div>
29 </div>
30 {% endblock %}
```

There's nothing fancy here: we extend `base.html.twig`, override the `title` block... then we have a form that submits a POST right back to `/login`. It doesn't have an `action` attribute, which means it submits back to this same URL. The form has two fields - input `name="email"` and input `name="password"` - and a submit button... all with Bootstrap 5 classes to look nice.

Let's add a link to this page from `base.html.twig`. Search for sign up. Cool. Right *before* this, add a link with `{{ path('app_login') }}`, say "Log In"... and give *this* some classes

to make it look nice:

```
templates/base.html.twig
1  <!DOCTYPE html>
2  <html>
3  // ... lines 3 - 14
15  <body>
16      <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17          <div class="container-fluid">
18  // ... lines 18 - 26
27      <div class="collapse navbar-collapse" id="navbar-
collapsible">
28  // ... lines 28 - 33
34          <a class="nav-link text-black-50" href="{{
path('app_login') }}">Log In</a>
35          <a href="#" class="btn btn-dark">Sign up</a>
36      </div>
37  </div>
38  </nav>
39  // ... lines 39 - 43
44  </body>
45  </html>
```

Let's check it out! Refresh the home page... and click the link. Hello log in page!

And of course, if we fill out the form and submit... absolutely nothing happens! That makes sense. This submits right back to `/login`... but because we don't have any form-processing logic yet... the page just re-renders.

So next: let's write that processing code. But... surprise! It won't live in the controller. It's time to create an authenticator and learn all about Symfony firewalls.

Chapter 5: Firewalls & Authenticators

We built this log in form by making a route, controller and rendering a template:

src/Controller/SecurityController.php

```
↕ // ... lines 1 - 8
9 class SecurityController extends AbstractController
10 {
11     /**
12      * @Route("/login", name="app_login")
13      */
14     public function login(): Response
15     {
16         return $this->render('security/login.html.twig');
17     }
18 }
```

Dead simple. When we submit the form, it POSTs right back to `/login`. So, to authenticate the user, you might expect us to put some logic right here: like if this is a POST request, read the POSTed email & password, query for the `User` object... and eventually check the password. That makes perfect sense! And that is *completely* not what we're going to do.

Hello Firewalls

Symfony's authentication system works in a... bit of a magic way, which I guess is fitting for our site. At the start of every request, before Symfony calls the controller, the security system executes a set of "authenticators". The job of each authenticator is to look at the request, see if there is any authentication information that it understands - like a submitted email and password, or an API key that's stored on a header - and if there *is*, use that to query the user and check the password. If all that happens successfully then... boom! Authentication complete.

Our job is to write and *activate* these authenticators. Open up `config/packages/security.yaml`. Remember the two parts of security: authentication (who you are) and authorization (what you can do).

The most important part of this file is `firewalls`:

```
config/packages/security.yaml
```

```
1 security:
↕ // ... lines 2 - 13
14     firewalls:
15         dev:
16             pattern: ^/(_(profiler|wdt)|css|images|js)/
17             security: false
18         main:
19             lazy: true
20             provider: app_user_provider
21
22             # activate different ways to authenticate
23             # https://symfony.com/doc/current/security.html#firewalls-
authentication
24
25             #
https://symfony.com/doc/current/security/impersonating_user.html
26             # switch_user: true
↕ // ... lines 27 - 33
```

A firewall is *all* about authentication: its job is to figure out *who* you are. And, it usually makes sense to have only *one* firewall in your app... even if there are multiple different ways to authenticate, like a login form *and* an API key *and* OAuth.

The "dev" Firewall

But... woh woh woh. If we almost always want only *one* firewall... why are there already two? Here's how this works: at the start of each request, Symfony goes down the list of firewalls, reads the `pattern` key - which is a regular expression - and finds the *first* firewall whose pattern matches the current URL. So there's only ever *one* firewall active per request.

If you look closely, this first firewall is a fake! It basically matches if the URL starts with `/_profiler` or `/_wdt`... and then sets security to `false`:

```
config/packages/security.yaml
```

```
1 security:
↕ // ... lines 2 - 13
14     firewalls:
15         dev:
16             pattern: ^/(_(profiler|wdt)|css|images|js)/
17             security: false
↕ // ... lines 18 - 33
```

In other words, it's basically making sure that you don't create a security system that is so epically awesome that... you block the web debug toolbar and profiler.

So... in reality, we only have *one* real firewall called `main`. It has no `pattern` key, which means that it will match all requests that don't match the `dev` firewall. Oh, and the names of these firewalls - `main` and `dev`? They're totally meaningless.

Activating Authenticators

Most of the config that we're going to put *beneath* the firewall relates to *activating* authenticators: those things that execute early in each request and try to authenticate the user. We'll add some of that config soon. But these two top keys do something different. `lazy` allows the authentication system to not authenticate the user until it needs to and `provider` ties this firewall to the user provider we talked about earlier. You should have both of these lines... but neither are terribly important:

```
config/packages/security.yaml
1  security:
  ↕ // ... lines 2 - 13
14  firewalls:
  ↕ // ... lines 15 - 17
18      main:
19          lazy: true
20          provider: app_user_provider
  ↕ // ... lines 21 - 33
```

Creating a Custom Authenticator Class

Anyways, anytime that we want to authenticate the user - like when we submit a login form - we need an authenticator. There *are* some core authenticator classes that we can use, including one for login forms.... and I'll show you some of those later. But to start, let's build our *own* authenticator class from scratch.

To do that, go to terminal and run:

```
symfony console make:auth
```

As you can see, you can select "Login form authenticator" to cheat and generate a bunch of code for a login form. But since we're building things from scratch, select "Empty authenticator" and call it `LoginFormAuthenticator`.

Awesome. This did two things: it created a new authenticator class and *also* updated `security.yaml`. Open the class first: `src/Security/LoginFormAuthenticator.php`:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractAuthenticator
13 {
14     public function supports(Request $request): ?bool
15     {
16         // TODO: Implement supports() method.
17     }
18
19     public function authenticate(Request $request): PassportInterface
20     {
21         // TODO: Implement authenticate() method.
22     }
23
24     public function onAuthenticationSuccess(Request $request,
25     TokenInterface $token, string $firewallName): ?Response
26     {
27         // TODO: Implement onAuthenticationSuccess() method.
28     }
29
30     public function onAuthenticationFailure(Request $request,
31     AuthenticationException $exception): ?Response
32     {
33         // TODO: Implement onAuthenticationFailure() method.
34     }
35
36     // ... lines 33 - 43
37 }
44 }
```

The only rule about an authenticator is that it needs to implement `AuthenticatorInterface`... though usually you'll extend `AbstractAuthenticator`... which implements `AuthenticatorInterface` for you:

```
src/Security/LoginFormAuthenticator.php
```

```
↕ // ... lines 1 - 8
9 use Symfony\Component\Security\Http\Authenticator\AbstractAuthenticator;
↕ // ... lines 10 - 11
12 class LoginFormAuthenticator extends AbstractAuthenticator
13 {
↕ // ... lines 14 - 43
44 }
```

We'll talk about what these methods do one-by-one. Anyways, `AbstractAuthenticator` is nice because it implements a *super boring* method for you.

Once we *activate* this new class in the security system, at the beginning of every request, Symfony will call this `supports()` method and basically ask:

"Do you see authentication information on this request that you understand?"

To prove that Symfony will call this, let's just `dd('supports')`:

```
src/Security/LoginFormAuthenticator.php
```

```
↕ // ... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractAuthenticator
13 {
14     public function supports(Request $request): ?bool
15     {
16         dd('supports!');
17     }
↕ // ... lines 18 - 43
44 }
```

Activating Authenticators with custom_authenticators

Okay, so how *do* we activate this authenticator? How do we tell our firewall that it should use our new class? Back in `security.yaml`, we *already* have the code that does that! This `custom_authenticator` line was added by the `make:auth` command:

```
config/packages/security.yaml
```

```
1 security:
  // ... lines 2 - 13
14   firewalls:
  // ... lines 15 - 17
18     main:
  // ... lines 19 - 20
21     custom_authenticator: App\Security\LoginFormAuthenticator
  // ... lines 22 - 34
```

So if you have a custom authenticator class, *this* is how you activate it. Later, we'll see that you can have *multiple* custom authenticators if you want.

Anyways, this means that our authenticator *is* already active! So let's try it. Refresh the login page. It hits the `supports()` method! In fact, if you go to *any* URL it will hit our `dd()`. On every request, before the controller, Symfony now asks our authenticator if it *supports* authentication on this request.

Next let's fill in the authenticator logic and get our user logged in!

Chapter 6: Authenticator & The Passport

On a basic level, authenticating a user when we submit the login form is... pretty simple. We need to read the submitted `email`, query the database for that `User` object... and eventually check the user's password.

Symfony's Security Doesn't Happen in a Controller

The *weird* thing about Symfony's security system is that... we're *not* going to write this logic in the controller. Nope. When we POST to `/login`, our authenticator is going to *intercept* that request and do all the work itself. Yup, when we submit the login form, our controller will actually *never* be executed.

The supports() Method

Now that our authenticator is activated, at the start of each request, Symfony will call the `supports()` method on our class. Our job is to return `true` if this request "contains authentication info that we know how to process". If not, we return `false`. If we return `false`, we don't *fail* authentication: it just means that our authenticator doesn't know how to authenticate this request... and the request continues processing like normal... executing whatever controller it matches.

So let's think: when do we want our authenticator to "do its work"? Which requests will "contains authentication info that we know how to process"? The answer to that is: whenever the user submits the login form.

Inside of `supports()` return true if `$request->getPathInfo()` - that's a fancy method to get the current URL - equals `/login` and if `$request->isMethod('POST')`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractAuthenticator
13 {
14     public function supports(Request $request): ?bool
15     {
16         return ($request->getPathInfo() === '/login' && $request-
>isMethod('POST'));
17     }
↕ // ... lines 18 - 43
44 }
```

So *if* the current request is a POST to `/login`, we want to try to authenticate the user. If not, we want to allow the request to continue like normal.

To see what happens next, down in `authenticate()`, `dd('authenticate')`:

💡 Tip

`PassportInterface` was deprecated since Symfony 5.4: use `Passport` as a return type instead.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 11
12 class LoginFormAuthenticator extends AbstractAuthenticator
13 {
↕ // ... lines 14 - 18
19     public function authenticate(Request $request): PassportInterface
20     {
21         dd('authenticate!');
22     }
↕ // ... lines 23 - 43
44 }
```

Testing time! Go refresh the homepage. Yup! The `supports()` method returned `false`... and the page kept loading like normal. In the web debug toolbar, we have a new security icon that says "Authenticated: no". But now go to the login form. This page *still* loads like normal. Enter `abraca_admin@example.com` - that's the email of a *real* user in the database - and any password - I'll use `foobar`. Submit and... got it! It hit our `dd('authenticate')`!

The authenticate() Method

So if `supports()` returns true, Symfony *then* calls `authenticate()`. This is the *heart* of our authenticator... and its job is to communicate two important things. First, *who* the user is that's trying to log in - specifically, which `User` object they are - and second, some *proof* that they are this user. In the case of a login form, that would be a password. Since our users don't actually *have* passwords yet... we'll fake it temporarily.

The Passport Object: UserBadge & Credentials

We communicate these two things by returning a `Passport` object: return new `Passport()`:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 12
13 use Symfony\Component\Security\Http\Authenticator\Passport\Passport;
↕ // ... lines 14 - 15
16 class LoginFormAuthenticator extends AbstractAuthenticator
17 {
↕ // ... lines 18 - 22
23     public function authenticate(Request $request): PassportInterface
24     {
↕ // ... lines 25 - 27
28         return new Passport(
↕ // ... lines 29 - 32
33         );
34     }
↕ // ... lines 35 - 55
56 }
```

This simple object is basically just a container for things called "badges"... where a badge is a little piece of information that goes into the passport. The two most important badges are `UserBadge` and some sort of "credentials badge" that helps prove that this user *is* who they say they are.

Start by grabbing the POSTed email and password:

`$email = $request->request->get('email')`. If you haven't seen it before, `$request->request->get()` is how you read `POST` data in Symfony. In the login template, the name of the field is `email`... so we read the `email` POST field. Copy and paste this line to create a `$password` variable that reads the `password` field from the form:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 15
16 class LoginFormAuthenticator extends AbstractAuthenticator
17 {
↕ // ... lines 18 - 22
23     public function authenticate(Request $request): PassportInterface
24     {
25         $email = $request->request->get('email');
26         $password = $request->request->get('password');
27
28         return new Passport(
↕ // ... lines 29 - 32
33         );
34     }
↕ // ... lines 35 - 55
56 }
```

Next, inside of the `Passport`, the first argument is always the `UserBadge`. Say `new UserBadge()` and pass this our "user identifier". For us, that's the `$email`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 10
11 use
    Symfony\Component\Security\Http\Authenticator\Passport\Badge\UserBadge;
↕ // ... lines 12 - 15
16 class LoginFormAuthenticator extends AbstractAuthenticator
17 {
↕ // ... lines 18 - 22
23     public function authenticate(Request $request): PassportInterface
24     {
25         $email = $request->request->get('email');
26         $password = $request->request->get('password');
27
28         return new Passport(
29             new UserBadge($email),
↕ // ... lines 30 - 32
33         );
34     }
↕ // ... lines 35 - 55
56 }
```

We'll talk very soon about how this is used.

The second argument to `Passport` is some sort of "credentials". Eventually we're going to pass it a `PasswordCredentials()`.... but since our users don't have passwords yet, use a

new `CustomCredentials()`. Pass this a callback with a `$credentials` arguments and a `$user` argument type-hinted with our `User` class:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 11
12 use
    Symfony\Component\Security\Http\Authenticator\Passport\Credentials\CustomCre
↕ // ... lines 13 - 15
16 class LoginFormAuthenticator extends AbstractAuthenticator
17 {
↕ // ... lines 18 - 22
23     public function authenticate(Request $request): PassportInterface
24     {
↕ // ... lines 25 - 27
28         return new Passport(
29             new UserBadge($email),
30             new CustomCredentials(function($credentials, User $user) {
↕ // ... lines 31 - 32
33             });
34         }
↕ // ... lines 35 - 55
56 }
```

Symfony will execute our callback and allow us to manually "check the credentials" for this user... whatever that means in our app. To start, `dd($credentials, $user)`. Oh, and `CustomCredentials` needs a second argument - which is whatever our "credentials" are. For us, that's `$password`:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 15
16 class LoginFormAuthenticator extends AbstractAuthenticator
17 {
↕ // ... lines 18 - 22
23     public function authenticate(Request $request): PassportInterface
24     {
↕ // ... lines 25 - 27
28         return new Passport(
29             new UserBadge($email),
30             new CustomCredentials(function($credentials, User $user) {
31                 dd($credentials, $user);
32             }, $password)
33         );
34     }
↕ // ... lines 35 - 55
56 }
```

If this `CustomCredentials` thing is a little fuzzy, don't worry: we really need to see this in action.

But on a high level... it's kind of cool. We return a `Passport` object, which says *who* the user is - identified by their `email` - and some sort of a "credentials process" that will *prove* that the user is who they say they are.

Ok: with *just* this, let's try it. Go back to the login form and re-submit. Remember: we filled in the form using an email address that *does* exist in our database.

And... awesome! `foobar` is what I submitted for my password and it's also dumping the correct `User` entity object from the database! So... woh! Somehow it knew to query for the `User` object *using* that email. How does that work?

The answer is the user provider! Let's dive into that next, learn how we can make a *custom* query for our user and finish the authentication process.

Chapter 7: Custom User Query & Credentials

On the screen, we see a `dd()` of the password I entered into the login form and the `User` entity *object* for the *email* I entered. Something, *somehow* knew to take the submitted email and query for the User!

UserBadge & The User Provider

Here's how this works. After we return the `Passport` object, the security system tries to find the `User` object from the `UserBadge`. If you just pass *one* argument to `UserBadge` - like we are - then it does this by leveraging our user provider. Remember that thing in `security.yaml` called `providers`?

```
config/packages/security.yaml
```

```
1 security:
2 // ... lines 2 - 7
8 providers:
9     # used to reload user from session & other features (e.g.
    switch_user)
10     app_user_provider:
11         entity:
12             class: App\Entity\User
13             property: email
14 // ... lines 14 - 34
```

Because our `User` class is an entity, we're using the `entity` provider that knows how to load users using the `email` property. So basically this is an object that's *really* good at querying the user table via the `email` property. So when we pass just the email to the `UserBadge`, the user provider uses that to query for the `User`.

If a `User` object *is* found, Symfony *then* tries to "check the credentials" on our passport. Because we're using `CustomCredentials`, this means that it executes this callback... where we're dumping some data. If a `User` could *not* be found - because we entered an email that isn't in the database - authentication fails. More on both of these situations soon.

Custom User Query

Anyways, the point is this: if you *just* pass one argument to `UserBadge`, the user provider loads the user automatically. That's the easiest thing to do. And you can even customize this query a bit if you need to - search for "[Using a Custom Query to Load the User](#)" on the Symfony docs to see how.

Or... you can write your *own* custom logic to load the user right here. To do that, we're going to need the `UserRepository`. At the top of the class, add `public function __construct()`... and autowire a `UserRepository` argument. I'll hit `Alt+Enter` and select "Initialize properties" to create that property and set it:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 5
6 use App\Repository\UserRepository;
↕ // ... lines 7 - 17
18 class LoginFormAuthenticator extends AbstractAuthenticator
19 {
20     private UserRepository $userRepository;
21
22     public function __construct(UserRepository $userRepository)
23     {
24         $this->userRepository = $userRepository;
25     }
↕ // ... lines 26 - 73
74 }
```

Down in `authenticate()`, `UserBadge` has an optional *second* argument called a user loader. Pass it a callback with one argument: `$userIdentifier`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 17
18 class LoginFormAuthenticator extends AbstractAuthenticator
19 {
↕ // ... lines 20 - 31
32     public function authenticate(Request $request): PassportInterface
33     {
↕ // ... lines 34 - 36
37         return new Passport(
38             new UserBadge($email, function($userIdentifier) {
↕ // ... lines 39 - 46
47                 })),
↕ // ... lines 48 - 50
51         );
52     }
↕ // ... lines 53 - 73
74 }
```

It's pretty simple: if you pass a callable, then when Symfony loads your `User`, it will call *this* function *instead* of your user provider. Our job here is to *load the user* and return it. The `$userIdentifier` will be whatever we passed to the first argument of `UserBadge`... so the `email` in our case.

Say `$user = $this->userRepository->findOneBy()` to query for `email` set to `$userIdentifier`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 17
18 class LoginFormAuthenticator extends AbstractAuthenticator
19 {
↕ // ... lines 20 - 31
32     public function authenticate(Request $request): PassportInterface
33     {
↕ // ... lines 34 - 36
37         return new Passport(
38             new UserBadge($email, function($userIdentifier) {
39                 // optionally pass a callback to load the User manually
40                 $user = $this->userRepository->findOneBy(['email' =>
                    $userIdentifier]);
↕ // ... lines 41 - 46
47             })),
↕ // ... lines 48 - 50
51         );
52     }
↕ // ... lines 53 - 73
74 }
```

This is where you could use whatever custom query you want. If we *can't* find the user, we need to throw a special exception. So if not `$user`, `throw new UserNotFoundException()`. That will cause authentication to fail. At the bottom, return `$user`:


```

src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 17
18 class LoginFormAuthenticator extends AbstractAuthenticator
19 {
↕ // ... lines 20 - 31
32     public function authenticate(Request $request): PassportInterface
33     {
↕ // ... lines 34 - 36
37         return new Passport(
38             new UserBadge($email, function($userIdentifier) {
39                 // optionally pass a callback to load the User manually
40                 $user = $this->userRepository->findOneBy(['email' =>
41                     $userIdentifier]);
42                 if (!$user) {
43                     throw new UserNotFoundException();
44                 }
45
46                 return $user;
47             })),
↕ // ... lines 48 - 50
51         );
52     }
↕ // ... lines 53 - 73
74 }

```

This... is basically identical to what our user provider was doing a minute ago... so it won't change anything. But you can see how we have the power to load the `User` however we want to.

Let's refresh. Yup! The same dump as before.

Validating the Credentials

Ok, so if a `User` object is found - either from our custom callback or the user provider - Symfony *next* checks our credentials, which means something different depending on *which* credentials object you pass. There are 3 main ones: `PasswordCredentials` - we'll see that later, a `SelfValidatingPassport` which is good for API authentication and doesn't need any credentials - and `CustomCredentials`.

If you use `CustomCredentials`, Symfony executes the callback... and our job is to "check their credentials"... whatever that means in our app. The `$credentials` argument will match

whatever we passed to the 2nd argument to `CustomCredentials`. For us, that's the submitted password:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 17
18 class LoginFormAuthenticator extends AbstractAuthenticator
19 {
↕ // ... lines 20 - 31
32     public function authenticate(Request $request): PassportInterface
33     {
↕ // ... lines 34 - 36
37         return new Passport(
↕ // ... lines 38 - 47
48             new CustomCredentials(function($credentials, User $user) {
↕ // ... line 49
50                 }, $password)
51         );
52     }
↕ // ... lines 53 - 73
74 }
```

Let's pretend that all users have the same password `tada!` To validate that, return true if `$credentials === 'tada'`:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 17
18 class LoginFormAuthenticator extends AbstractAuthenticator
19 {
↕ // ... lines 20 - 31
32     public function authenticate(Request $request): PassportInterface
33     {
↕ // ... lines 34 - 36
37         return new Passport(
↕ // ... lines 38 - 47
48             new CustomCredentials(function($credentials, User $user) {
49                 return $credentials === 'tada';
50             }, $password)
51         );
52     }
↕ // ... lines 53 - 73
74 }
```

Air-tight security!

Authentication Failure and Success

If we return `true` from this function, authentication is successful! Woo! If we return `false`, authentication fails. To prove this, go down to `onAuthenticationSuccess()` and `dd('success')`. Do the same thing inside `onAuthenticationFailure()`:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 17
18 class LoginFormAuthenticator extends AbstractAuthenticator
19 {
↕ // ... lines 20 - 53
54     public function onAuthenticationSuccess(Request $request,
    TokenInterface $token, string $firewallName): ?Response
55     {
56         dd('success');
57     }
58
59     public function onAuthenticationFailure(Request $request,
    AuthenticationException $exception): ?Response
60     {
61         dd('failure');
62     }
↕ // ... lines 63 - 73
74 }
```

We'll put *real* code into these methods soon... but their purpose is pretty self-explanatory: if authentication is successful, Symfony will call `onAuthenticationSuccess()`. If authentication fails for *any* reason - like an invalid email or password - Symfony will call `onAuthenticationFailure()`.

Let's try it! Go directly back to `/login`. Use the real email again - `abraca_admin@example.com` with the correct password: `tada`. Submit and... yes! It hit `onAuthenticationSuccess()`. Authentication is complete!

I know, it doesn't *look* like much yet... so next, let's *do* something on success, like redirect to another page. We're also going to learn about the *other* critical job of a user provider: refreshing the user from the session at the beginning of each request to keep us logged in.

Chapter 8: Authentication Success & Refreshing the User

Let's do a quick review of how our authenticator works. After activating it in `security.yaml`:

```
config/packages/security.yaml
1  security:
  ↕ // ... lines 2 - 13
14  firewalls:
  ↕ // ... lines 15 - 17
18  main:
  ↕ // ... lines 19 - 20
21      custom_authenticator: App\Security\LoginFormAuthenticator
  ↕ // ... lines 22 - 34
```

Symfony calls our `supports()` method on every request before the controller:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 17
18 class LoginFormAuthenticator extends AbstractAuthenticator
19 {
  ↕ // ... lines 20 - 26
27     public function supports(Request $request): ?bool
28     {
29         return ($request->getPathInfo() === '/login' && $request-
>isMethod('POST'));
30     }
  ↕ // ... lines 31 - 73
74 }
```

Since our authenticator knows how to handle the login form submit, we return true if the current request is a `POST` to `/login`. Once we return true, Symfony then calls `authenticate()` and basically asks:

“Okay, tell me who is trying to log in and what proof they have.”

We answer these questions by returning a `Passport`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 17
18 class LoginFormAuthenticator extends AbstractAuthenticator
19 {
↕ // ... lines 20 - 31
32     public function authenticate(Request $request): PassportInterface
33     {
↕ // ... lines 34 - 36
37         return new Passport(
38             new UserBadge($email, function($userIdentifier) {
39                 // optionally pass a callback to load the User manually
40                 $user = $this->userRepository->findOneBy(['email' =>
41                     $userIdentifier]);
42                 if (!$user) {
43                     throw new UserNotFoundException();
44                 }
45
46                 return $user;
47             }),
48             new CustomCredentials(function($credentials, User $user) {
49                 return $credentials === 'tada';
50             }, $password)
51         );
52     }
↕ // ... lines 53 - 73
74 }
```

The first argument identifies the user and the second argument identifies some *proof*... in this case, just a callback that checks that the submitted password is `tada`. If we are able to find a user *and* the credentials are correct... then we are authenticated!

We saw this at the end of the last video! When we logged in using the email of a real user in our database and password `tada`... we hit this `dd()` statement:

```
src/Security/LoginFormAuthenticator.php
```

```
↕ // ... lines 1 - 17
18 class LoginFormAuthenticator extends AbstractAuthenticator
19 {
↕ // ... lines 20 - 53
54     public function onAuthenticationSuccess(Request $request,
        TokenInterface $token, string $firewallName): ?Response
55     {
56         dd('success');
57     }
↕ // ... lines 58 - 73
74 }
```

onAuthenticationSuccess

Yep! If authentication is successful Symfony calls `onAuthenticationSuccess()` and asks:

“Congrats on authenticating! We're super proud! But... what should we do now?”

In our situation, after success, we probably want to redirect the user to some other page. But for other types of authentication you might do something different. For example, if you're authenticating via an API token, you would return `null` from this method to allow the request to continue to the normal controller.

Anyways, that's our job here: to decide what to do "next"... which will either be "do nothing" - `null` - or return some sort of `Response` object. We're going to redirect.

Head up to the top of this class. Add a second argument - `RouterInterface $router` - use the `Alt+Enter` trick and select "Initialize properties" to create that property and set it:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 9
10 use Symfony\Component\Routing\RouterInterface;
↕ // ... lines 11 - 19
20 class LoginFormAuthenticator extends AbstractAuthenticator
21 {
↕ // ... line 22
23     private RouterInterface $router;
24
25     public function __construct(UserRepository $userRepository,
    RouterInterface $router)
26     {
↕ // ... line 27
28         $this->router = $router;
29     }
↕ // ... lines 30 - 79
80 }
```

Back down in `onAuthenticationSuccess()`, we need to return `null` or a `Response`.

Return a new `RedirectResponse()` and, for the URL, say

`$this->router->generate()` and pass `app_homepage`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 6
7 use Symfony\Component\HttpFoundation\RedirectResponse;
↕ // ... lines 8 - 19
20 class LoginFormAuthenticator extends AbstractAuthenticator
21 {
↕ // ... lines 22 - 57
58     public function onAuthenticationSuccess(Request $request,
    TokenInterface $token, string $firewallName): ?Response
59     {
60         return new RedirectResponse(
61             $this->router->generate('app_homepage')
62         );
63     }
↕ // ... lines 64 - 79
80 }
```

Let me go... double-check that route name.... it should be inside of `QuestionController`.

Yup! `app_homepage` is correct:

```

src/Controller/QuestionController.php
↕ // ... lines 1 - 17
18 class QuestionController extends AbstractController
19 {
↕ // ... lines 20 - 29
30     /**
31      * @Route("/{page<\d+>}", name="app_homepage")
32      */
33     public function homepage(QuestionRepository $repository, int $page =
34     1)
35     {
↕ // ... lines 35 - 43
44     }
↕ // ... lines 45 - 86
87 }

```

I'm not sure why PhpStorm thinks this route is missing... it's definitely there.

Anyways, let's log in from scratch. Go directly to `/login`, enter

`abraca_admin@example.com` - because that's a *real* email in our database - and password "tada". When we submit... it works! We're redirected! And we're logged in! I know because of the web debug toolbar: logged in as `abraca_admin@example.com`, authenticated: Yes.

If you click this icon to jump into the profiler, there is a *ton* of juicy info about security. We're going to talk about the most important parts of this as we go along.

Authentication Info & The Session

Click back to the homepage. Notice that, if we surf around the site, we *stay* logged in... which is what we want. This works because Symfony firewalls are, by default, "stateful". That's a fancy way of saying that, at the end of each request, the `User` object is saved to the session. Then at the start of the next request, that `User` object is *loaded* from the session... and we stay logged in.

Refreshing the User

This works great! But... there is one potential problem. Imagine we log in at our work computer. Then, we go home, log in on a totally *different* computer, and change some of our user data - like maybe we change our `firstName` in the database via an "edit profile" section. When we

come back to work the next day and refresh the site, Symfony will, of course, load the `User` object from the session. But... that `User` object will now have the wrong `firstName`! Its data will no longer match what's in the database... because we're reloading a "stale" object from the session.

Fortunately... this is *not* a real problem. Why? Because at the beginning of every request, Symfony also *refreshes* the user. Well, actually our "user provider" does this. Back in `security.yaml`, remember that user provider thingy?

```
config/packages/security.yaml
1  security:
2  // ... lines 2 - 7
8    providers:
9      # used to reload user from session & other features (e.g.
    switch_user)
10     app_user_provider:
11         entity:
12             class: App\Entity\User
13             property: email
14     firewalls:
15 // ... lines 15 - 17
18     main:
19 // ... line 19
20         provider: app_user_provider
21 // ... lines 21 - 34
```

Yep it has *two* jobs. First, if we give it an email, it knows how to find that user. If we only pass a single argument to `UserBadge` then the user provider does the hard work of loading the `User` from the database:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractAuthenticator
21 {
↕ // ... lines 22 - 35
36     public function authenticate(Request $request): PassportInterface
37     {
↕ // ... lines 38 - 40
41         return new Passport(
42             new UserBadge($email, function($userIdentifier) {
↕ // ... lines 43 - 50
51                 })),
↕ // ... lines 52 - 54
55         );
56     }
↕ // ... lines 57 - 79
80 }
```

But the user provider also has a *second* job. At the start of every request, it refreshes the **User** by querying the database for fresh data. This all happens automatically in the background... which is great! It's a boring, but critical process that you, at least, should be aware of.

User Changed === Logged Out

Oh, and by the way: after querying for the fresh **User** data, if some important data on the user *changed* - like the **email**, **password** or **roles** - you'll actually get logged out. This is a security feature: it allows a user to, for example, change their password and cause any "bad" users who may have gotten access to their account to get logged out. If you want to learn more about this, search for **EquatableInterface**: that's an interface that allows you to control this process.

Let's find out what happens when we *fail* authentication. Where does the user go? How are errors displayed? How will we deal with the emotional burden of failure? *Most* of that is next.

Chapter 9: When Authentication Fails

Go back to the login form. What happens if we *fail* login? Right now, there are two ways to fail: if we can't find a `User` for the email or if the password is incorrect. Let's try a wrong password first.

onAuthenticationFailure & AuthenticationException

Enter a real email from the database... and then any password that *isn't* "tada". And... yep! We hit the `dd()`... that comes from `onAuthenticationFailure()`:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractAuthenticator
21 {
↕ // ... lines 22 - 64
65     public function onAuthenticationFailure(Request $request,
        AuthenticationException $exception): ?Response
66     {
67         dd('failure');
68     }
↕ // ... lines 69 - 79
80 }
```

So no matter *how* we fail authentication, we end up here, and we're passed an `$exception` argument. Let's also dump that:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractAuthenticator
21 {
↕ // ... lines 22 - 64
65     public function onAuthenticationFailure(Request $request,
        AuthenticationException $exception): ?Response
66     {
67         dd('failure', $exception);
68     }
↕ // ... lines 69 - 79
80 }
```

Head back... and refresh. Cool! It's a `BadCredentialsException`.

This is cool. If authentication fails - no matter *how* it fails - we're going to end up here with some sort of `AuthenticationException`. `BadCredentialsException` is a *subclass* of that... as is the `UserNotFoundException` that we're throwing from our user loader callback:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 19
20 class LoginFormAuthenticator extends AbstractAuthenticator
21 {
↕ // ... lines 22 - 35
36     public function authenticate(Request $request): PassportInterface
37     {
↕ // ... lines 38 - 40
41         return new Passport(
42             new UserBadge($email, function($userIdentifier) {
↕ // ... lines 43 - 45
46                 if (!$user) {
47                     throw new UserNotFoundException();
48                 }
↕ // ... lines 49 - 50
51             })),
↕ // ... lines 52 - 54
55         );
56     }
↕ // ... lines 57 - 79
80 }
```

All of these exception classes have one important thing in common. Hold `Command` or `Ctrl` to open up `UserNotFoundException` to see it. All of these authentication exceptions have a special `getMessageKey()` method that contains a *safe* explanation of why authentication failed. We can use this to tell the user what went wrong.

hide_user_not_found: Showing Invalid Username/Email Errors

So here's the big picture: when authentication fails, it's because *something* threw an `AuthenticationException` or one of its sub-classes. And so, since we're throwing a `UserNotFoundException` when an unknown email is entered... if we try to log in with a bad email, *that* exception should be passed to `onAuthenticationFailure()`.

Let's test that theory. At the login form, enter some invented email... and... submit. Oh! We *still* get a `BadCredentialsException`! I was expecting this to be the *actual* exception that was thrown: the `UserNotFoundException`.

For the most part... that *is* how this works. If you throw an `AuthenticationException` during the authenticator process, that exception *is* passed to you down in

`onAuthenticationFailure()`. Then you can use it to figure out what went wrong.

However, `UserNotFoundException` is a special case. On some sites, when the user enters a *valid* email address but a wrong password, you might *not* want to tell the user that email *was* in fact found. So you say "Invalid credentials" both if the email wasn't found *or* if the password was incorrect.

This problem is called user enumeration: it's where someone can test emails on your login form to figure out which people have accounts and which don't. For some sites, you definitely do *not* want to expose that information.

And so, to be safe, Symfony converts `UserNotFoundException` to a `BadCredentialsException` so that entering an invalid email or invalid password both give the same error message. However, if you *do* want to be able to say "Invalid email" - which is much more helpful to your users - you *can* do this.

Open up `config/packages/security.yaml`. And, anywhere under the root `security` key, add a `hide_user_not_found` option set to `false`:

```
config/packages/security.yaml
1 security:
  ⬆ // ... lines 2 - 4
5   hide_user_not_found: false
  ⬆ // ... lines 6 - 37
```

This tells Symfony to *not* convert `UserNotFoundException` to a `BadCredentialsException`.

If we refresh now... boom! Our `UserNotFoundException` is now being passed directly to `onAuthenticationFailure()`.

Storing the Authentication Error in the Session

Ok, so let's think. Down in `onAuthenticationFailure()`... what do we want to do? Our job in this method is, as you can see, to return a `Response` object. For a login form, what we probably want to do is redirect the user *back* to the login page but show an error.

To be able to do that, let's stash this exception - which holds the error message - into the session. Say `$request->getSession()->set()`. We can really use whatever key we want... but there's a standard key that's used to store authentication errors. You can read it from a constant: `Security` - the one from the Symfony Security component - `::AUTHENTICATION_ERROR`. Pass `$exception` to the second argument:

💡 Tip

In Symfony 6.2 and higher, use the `SecurityRequestAttributes` class instead: `Symfony\Component\Security\Http\SecurityRequestAttributes`, then `SecurityRequestAttributes::AUTHENTICATION_ERROR`.

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 20
21 class LoginFormAuthenticator extends AbstractAuthenticator
22 {
↕ // ... lines 23 - 65
66     public function onAuthenticationFailure(Request $request,
        AuthenticationException $exception): ?Response
67     {
68         $request->getSession()->set(Security::AUTHENTICATION_ERROR,
            $exception);
↕ // ... lines 69 - 72
73     }
↕ // ... lines 74 - 84
85 }
```

Now that the error is in the session, let's redirect back to the login page. I'll cheat and copy the `RedirectResponse` from earlier... and change the route to `app_login`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 20
21 class LoginFormAuthenticator extends AbstractAuthenticator
22 {
↕ // ... lines 23 - 65
66     public function onAuthenticationFailure(Request $request,
        AuthenticationException $exception): ?Response
67     {
68         $request->getSession()->set(Security::AUTHENTICATION_ERROR,
        $exception);
69
70         return new RedirectResponse(
71             $this->router->generate('app_login')
72         );
73     }
↕ // ... lines 74 - 84
85 }
```

AuthenticationUtils: Rendering the Error

Cool! Next, inside `login()` controller, we need to read that error and render it. The most straightforward way to do that would be to grab the session and read out this key. But... it's even easier than that! Symfony provides a service that will grab the key from the session automatically. Add a new argument type-hinted with `AuthenticationUtils`:

src/Controller/SecurityController.php

```
↕ // ... lines 1 - 7
8 use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;
9
10 class SecurityController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function login(AuthenticationUtils $authenticationUtils):
        Response
16     {
↕ // ... lines 17 - 19
20     }
21 }
```

And then give `render()` a second argument. Let's pass an `error` variable to Twig set to `$authenticationUtils->getLastAuthenticationError()`:

```
src/Controller/SecurityController.php
```

```
↕ // ... lines 1 - 9
10 class SecurityController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function login(AuthenticationUtils $authenticationUtils):
        Response
16     {
17         return $this->render('security/login.html.twig', [
18             'error' => $authenticationUtils->getLastAuthenticationError(),
19         ]);
20     }
21 }
```

That's just a shortcut to read that key off of the session.

This means that the `error` variable is literally going to be an `AuthenticationException` object. And remember, to figure out what went wrong, all `AuthenticationException` objects have a `getMessageKey()` method that returns an explanation.

In `templates/security/login.html.twig`, let's render that. Right after the `h1`, say if `error`, then add a `div` with `alert alert-danger`. Inside render `error.messageKey`:

```
templates/security/login.html.twig
```

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <form method="post" class="row g-3">
10                 <h1 class="h3 mb-3 font-weight-normal">Please sign in</h1>
11
12                 {% if error %}
13                     <div class="alert alert-danger">{{ error.messageKey }}
14                 {% endif %}
15             </form>
16         </div>
17     </div>
18 </div>
19 {% endblock %}
```

You don't want to use `error.message` because if you had some sort of internal error - like a database connection error - that message might contain sensitive details. But

`error.messageKey` is guaranteed to be safe.

Testing time! Refresh! Yes! We're redirected back to `/login` and we see:

"Username could not be found."

That's the message if the `User` object can't be loaded: the error that comes from `UserNotFoundException`. It's... not a great message... since our users are logging in with an email, not a username.

So next, let's learn how to customize these error messages *and* add a way to log out.

Chapter 10: Customize Error Messages & Adding Logout

When we fail login, we store the `AuthenticationException` in the session - which explains what went wrong - and then redirect to the login page:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 20
21 class LoginFormAuthenticator extends AbstractAuthenticator
22 {
↕ // ... lines 23 - 65
66     public function onAuthenticationFailure(Request $request,
        AuthenticationException $exception): ?Response
67     {
68         $request->getSession()->set(Security::AUTHENTICATION_ERROR,
        $exception);
69
70         return new RedirectResponse(
71             $this->router->generate('app_login')
72         );
73     }
↕ // ... lines 74 - 84
85 }
```

On *that* page, we read that exception *out* of the session using this nice `AuthenticationUtils` service:

```
src/Controller/SecurityController.php
↕ // ... lines 1 - 9
10 class SecurityController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function login(AuthenticationUtils $authenticationUtils):
        Response
16     {
17         return $this->render('security/login.html.twig', [
18             'error' => $authenticationUtils->getLastAuthenticationError(),
19         ]);
20     }
21 }
```

And ultimately, in the template, call the `getMessageKey()` method to render a safe message that describes *why* authentication failed:

```
templates/security/login.html.twig
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="container">
7      <div class="row">
8          <div class="login-form bg-light mt-4 p-4">
9              <form method="post" class="row g-3">
↕ // ... lines 10 - 11
12                 {% if error %}
13                     <div class="alert alert-danger">{{ error.messageKey }}
14                 </div>
15                 {% endif %}
↕ // ... lines 15 - 29
30             </form>
31         </div>
32     </div>
33 </div>
34 {% endblock %}
```

For example, if we enter an email that doesn't exist, we see:

"Username could not be found."

On a technical level, this means that the `User` object could not be found. Cool... but for us, this isn't a great message because we're logging in via an *email*. Also, if we enter a *valid* user - `abraca_admin@example.com` - with an invalid password, we see:

"Invalid credentials."

That's a better message... but it's not super friendly.

Translating the Error Messages?

So how can we customize these? The answer is both simple and... maybe a bit surprising. We *translate* them. Check it out: over in the template, after `messageKey`, add `|trans` to translate it. Pass this two arguments. The first is `error.messageData`. This isn't too important... but in the translation world, sometimes your translations can have "wildcard" values in them... and you

pass in the values for those wildcards here. The second argument is called a "translation domain"... which is almost like a translation category. Pass `security`:

```
templates/security/login.html.twig
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <form method="post" class="row g-3">
↕ // ... lines 10 - 11
12                 {% if error %}
13                     <div class="alert alert-danger">{{
error.messageKey|trans(error.messageData, 'security') }}</div>
14                 {% endif %}
↕ // ... lines 15 - 29
30             </form>
31         </div>
32     </div>
33 </div>
34 {% endblock %}
```

If you *do* have a multi-lingual site, all of the core authentication messages have *already* been translated to other languages... and those translations are available in a domain called `security`. So by using the `security` domain here, if we switched the site to Spanish, we would instantly get Spanish authentication messages.

If we stopped now... absolutely nothing would change! But because we're going through the translator, we have the *opportunity* to "translate" these strings from English to... *different* English!

In the `translations/` directory - which you should automatically have because the translation component is already installed - create a new file called `security.en.yaml`: `security` because we're using the `security` translation domain and `en` for English. You can also create `.xlf` translation files - YAML is just easier for what we need to do.

Now, copy the exact error message including the period, paste - I'll wrap it in quotes to be safe - and set it to something different like:

```
"Invalid password entered!"
```

```
translations/security.en.yaml
```

```
1 "Invalid credentials.": "Invalid password entered!"
```

Cool! Let's try it again. Log in as `abraca_admin@example.com` with an invalid password and... much better! Let's try with a bad email.

Ok, repeat the process: copy the message, go over to the translation file, paste... and change it to something a bit more user-friendly like:

```
"Email not found!"
```

```
translations/security.en.yaml
```

```
↕ // ... line 1
```

```
2 "Username could not be found.": "Email not found!"
```

Let's try it again: same email, any password and... got it!

```
"Email not found."
```

Okay! Our authenticator is done! We load the `User` from the email, check their password and handle both success and failure. Booya! We are going to add more stuff to this later - including checking real user passwords - but this *is* fully functional.

Logging Out

Let's add a way to log out. So... like if the user goes to `/logout`, they get... logged it! This starts exactly like you expect: we need a route & controller.

Inside of `SecurityController`, I'll copy the `login()` method, paste, change it to `/logout`, `app_logout` and call the *method* `logout`:

src/Controller/SecurityController.php

```
↕ // ... lines 1 - 9
10 class SecurityController extends AbstractController
11 {
↕ // ... lines 12 - 21
22     /**
23      * @Route("/logout")
24      */
25     public function logout()
26     {
↕ // ... line 27
28     }
29 }
```

To *perform* the logging out itself... we're going to put absolutely no code in this method. Actually, I'll throw a new `\Exception()` that says "logout() should never be reached":

src/Controller/SecurityController.php

```
↕ // ... lines 1 - 9
10 class SecurityController extends AbstractController
11 {
↕ // ... lines 12 - 21
22     /**
23      * @Route("/logout")
24      */
25     public function logout()
26     {
27         throw new \Exception('logout() should never be reached');
28     }
29 }
```

Let me explain. Logging out works a bit like logging in. Instead of putting some logic in the controller, we're going to activate something on our firewall that says:

"Hey! If the user goes to `/logout`, please intercept that request, log out the user, and redirect them somewhere else."

To activate that magic, open up `config/packages/security.yaml`. Anywhere under our firewall, add `logout: true`:

```
config/packages/security.yaml
```

```
1 security:
  // ... lines 2 - 16
17   firewalls:
  // ... lines 18 - 20
21     main:
  // ... lines 22 - 25
26       logout: true
  // ... lines 27 - 39
```

Internally, this activates a "listener" that looks for any requests to `/logout`.

Configuring logout

And actually, instead of just saying `logout: true`, you can customize how this works. Find your terminal and run:

```
symfony console debug:config security
```

As a reminder, this command shows you all of your *current* configuration under the `security` key. So all of our config *plus* any default values.

If we run this... and find the `main` firewall... check out the `logout` section. All of these keys are the *default* values. Notice there's one called `path: /logout`. *This* is why it's listening to the URL `/logout`. If you wanted to log out via another URL, you would just tweak this key here.

But since we have `/logout` here... and that matches our `/logout` right here, this *should* work. By the way, you might be wondering why we needed to create a route and controller at all! Great question! We actually *don't* need a controller, it will never be called. But we *do* need a route. If we *didn't* have one, the routing system would trigger a 404 error *before* the logout system could work its magic. Plus, it's nice to have a route, so we can generate a URL to it.

Ok: let's test this thing! Log in first: `abraca_admin@example.com` and password `tada`. Awesome: we *are* authenticated. Manually go to `/logout` and... we are now logged out! The default behavior of the system is to log us out and redirect back to the homepage. If you need to customize that, there are a few options. First, under the `logout` key, you can change `target` to some other URL or route name.

But we can also hook into the logout process via an event listener, a topic that we'll talk about towards the end of the tutorial.

Next: let's give each user a *real* password. This will involve hashing passwords, so we can securely store them in the database and then *checking* those hashed passwords during authentication. Symfony makes both of these easy.

Chapter 11: Giving Users Passwords

Symfony doesn't really care if the users in your system have passwords or not. If you're building a login system that reads API keys from a header, then there are no passwords. The same is true if you have some sort of SSO system. Your users might have passwords... but they enter them on some *other* site.

But for us, we *do* want each user to have a password. When we used the `make:user` command earlier, it actually asked us if we wanted our users to have passwords. We answered no... so that we could do all of this manually. But in a real project, I *would* answer "yes" to save time.

PasswordAuthenticatedUserInterface

We know that all User classes must implement `UserInterface`:

```
src/Entity/User.php
↕ // ... lines 1 - 7
8 use Symfony\Component\Security\Core\User\UserInterface;
↕ // ... lines 9 - 12
13 class User implements UserInterface
14 {
↕ // ... lines 15 - 130
131 }
```

Then, *if* you need to check user passwords in your application, you *also* need to implement a second interface called `PasswordAuthenticatedUserInterface`:

```
src/Entity/User.php
↕ // ... lines 1 - 6
7 use
  Symfony\Component\Security\Core\User>PasswordAuthenticatedUserInterface;
↕ // ... lines 8 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 128
129 }
```

This requires you to have one new method: `getPassword()`.

If you're using Symfony 6, you won't have this yet, so add it:

```
src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface
14 {
↕ // ... lines 15 - 90
91     /**
92      * This method can be removed in Symfony 6.0 - is not needed for apps
    that do not check user passwords.
93      *
94      * @see PasswordAuthenticatedUserInterface
95      */
96     public function getPassword(): ?string
97     {
98         return null;
99     }
↕ // ... lines 100 - 130
131 }
```

I *do* have it because I'm using Symfony 5 and the `getPassword()` method is needed for backwards compatibility: it *used* to be part of `UserInterface`.

Now that our users *will* have a password, and we're implementing `PasswordAuthenticatedUserInterface`, I'm going to remove this comment above the method:

```
src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 90
91     /**
92      * @see PasswordAuthenticatedUserInterface
93      */
94     public function getPassword(): ?string
95     {
96         return null;
97     }
↕ // ... lines 98 - 128
129 }
```

Storing a Hashed Password for each User

Ok, let's forget about security for a minute. Instead, focus on the fact that we need to be able to store a unique password for each user in the database. This means that our user entity needs a new field! Find your terminal and run:

```
symfony console make:entity
```

Let's update the `User` entity, to add a new field call `password`... which is a string, 255 length is overkill but fine... and then say "no" to nullable. Hit enter to finish.

Back over in the `User` class, it's... mostly not surprising. We have a new `$password` property... and at the bottom, a new `setPassword()` method:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 36
37     /**
38      * @ORM\Column(type="string", length=255)
39      */
40     private $password;
↕ // ... lines 41 - 134
135     public function setPassword(string $password): self
136     {
137         $this->password = $password;
138
139         return $this;
140     }
141 }
```

Notice that it did *not* generate a `getPassword()` method... because we already had one. But we *do* need to update this to return `$this->password`:

```

src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 98
99     public function getPassword(): ?string
100     {
101         return $this->password;
102     }
↕ // ... lines 103 - 140
141 }

```

Very important thing about this `$password` property: it is *not* going to store the *plaintext* password. Never ever store the plaintext password! That's the fastest way to have a security breach... and lose friends.

Instead, we're going to store a *hashed* version of the password... and we'll see how to generate that hashed password in a minute. But first, let's make the migration for the new property:

```

○ ○ ○
symfony console make:migration

```

Go peek at that file to make sure everything looks good:

migrations/Version20211001185505.php

```
↕ // ... lines 1 - 12
13 final class Version20211001185505 extends AbstractMigration
14 {
15     public function getDescription(): string
16     {
17         return '';
18     }
19
20     public function up(Schema $schema): void
21     {
22         // this up() migration is auto-generated, please modify it to your
needs
23         $this->addSql('ALTER TABLE user ADD password VARCHAR(255) NOT
NULL');
24     }
25
26     public function down(Schema $schema): void
27     {
28         // this down() migration is auto-generated, please modify it to
your needs
29         $this->addSql('ALTER TABLE user DROP password');
30     }
31 }
```

💡 Tip

If you are using PostgreSQL, you should modify your migration. Add `DEFAULT ''` at the end so that the new column can be added without an error:

```
$this->addSql('ALTER TABLE product ADD description VARCHAR(255) NOT NULL DEF
```

And... it does! Close it... and run it:

```
symfony console doctrine:migrations:migrate
```

The password hashers Config

Perfect! Now that our users have a new password column in the database, let's *populate* that in our fixtures. Open up `src/Factory/UserFactory.php` and find `getDefaults()`.

Again, what we are *not* going to do is set `password` to the plain-text password. Nope, that `password` property needs to store the *hashed* version of the password.

Open up `config/packages/security.yaml`. This has a little bit of config on top called `password_hashers`, which tells Symfony which hashing *algorithm* it should use for hashing user passwords:

```
config/packages/security.yaml
1 security:
2 // ... lines 2 - 6
3
4 # https://symfony.com/doc/current/security.html#c-hashing-passwords
5 password_hashers:
6
7     Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
8         'auto'
9
10 // ... lines 10 - 39
```

This config says that any `User` classes that implement `PasswordAuthenticatedUserInterface` - which our class, of course, does - will use the `auto` algorithm where Symfony chooses the latest and greatest algorithm automatically.

The Password Hasher Service

Thanks to this config, we have access to a "hasher" service that's able to convert a plaintext password into a *hashed* version using this `auto` algorithm. Back inside `UserFactory`, we can use that to set the `password` property:

```
src/Factory/UserFactory.php
1 // ... lines 1 - 28
2
3 final class UserFactory extends ModelFactory
4 {
5     // ... lines 31 - 37
6
7     protected function getDefaults(): array
8     {
9         return [
10             // ... lines 41 - 42
11             'plainPassword' => 'tada',
12         ];
13     }
14
15     // ... lines 46 - 58
16
17 }
```

In the constructor, add a new argument:

`UserPasswordHasherInterface $passwordHasher`. I'll hit `Alt+Enter` and go to "Initialize properties" to create that property and set it:

```
src/Factory/UserFactory.php
↕ // ... lines 1 - 6
7 use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
↕ // ... lines 8 - 29
30 final class UserFactory extends ModelFactory
31 {
32     private UserPasswordHasherInterface $passwordHasher;
33
34     public function __construct(UserPasswordHasherInterface
35     $passwordHasher)
36     {
37         parent::__construct();
38
39         $this->passwordHasher = $passwordHasher;
40     }
41
42     // ... lines 40 - 67
68 }
```

Below, we can set `password` to `$this->passwordHasher->hashPassword()` and then pass it some plain-text string.

Well... to be honest... while I hope this makes sense on a *high* level... this won't *quite* work because the first argument to `hashPassword()` is the `User` object... which we don't have yet inside `getDefaults()`.

That's ok because I like to create a `plainPassword` property on `User` to help make all of this easier anyways. Let's add that next, finish the fixtures and update our authenticator to validate the password. Oh, but don't worry: that new `plainPassword` property *won't* be stored in the database.

Chapter 12: Hashing Plain Passwords & PasswordCredentials

The process of saving a user's password always looks like this: start with a plain-text password, hash that, *then* save the hashed version onto the `User`. This is something we're going to do in the fixtures... but we'll also do this on a registration form later... and you would also need it on a change password form.

Adding a plainPassword Field

To make this easier, I'm going to do something optional. In `User`, up on top, add a new `private $plainPassword` property:

```
src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 41
42     private $plainPassword;
↕ // ... lines 43 - 154
155 }
```

The *key* thing is that this property will *not* be persisted to the database: it's just a temporary property that we can use during, for example, registration, to store the plain password.

Below, I'll go to "Code"-"Generate" - or `Command+N` on a Mac - to generate the getter and setter for this. The getter will return a nullable `string`:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 143
144     public function getPlainPassword(): ?string
145     {
146         return $this->plainPassword;
147     }
148
149     public function setPlainPassword(string $plainPassword): self
150     {
151         $this->plainPassword = $plainPassword;
152
153         return $this;
154     }
155 }
```

Now, if you *do* have a `plainPassword` property, you'll want to find `eraseCredentials()` and set `$this->plainPassword` to null:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 118
119     public function eraseCredentials()
120     {
121         // If you store any temporary, sensitive data on the user, clear
        it here
122         $this->plainPassword = null;
123     }
↕ // ... lines 124 - 154
155 }
```

This... is not really *that* important. After authentication is successful, Symfony calls `eraseCredentials()`. It's... just a way for you to "clear out any sensitive information" on your `User` object once authentication is done. Technically we will never set `plainPassword` during authentication... so it doesn't matter. But, again, it's a safe thing to do.

Hashing the Password in the Fixtures

Back inside `UserFactory`, instead of setting the `password` property, set `plainPassword` to "tada":

```
src/Factory/UserFactory.php
↕ // ... lines 1 - 28
29 final class UserFactory extends ModelFactory
30 {
↕ // ... lines 31 - 37
38     protected function getDefaults(): array
39     {
40         return [
↕ // ... lines 41 - 42
43             'plainPassword' => 'tada',
44         ];
45     }
↕ // ... lines 46 - 58
59 }
```

If we just stopped now, it would set this property... but then the `password` property would stay `null`... and it would explode in the database because that column is required.

So after Foundry has finished instantiating the object, we need to run some extra code that reads the `plainPassword` and hashes it. We can do that down here in the `initialize()` method... via an "after instantiation" hook:

```
src/Factory/UserFactory.php
↕ // ... lines 1 - 28
29 final class UserFactory extends ModelFactory
30 {
↕ // ... lines 31 - 46
47     protected function initialize(): self
48     {
49         // see
https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#initializ
50         return $this
51             // ->afterInstantiate(function(User $user) {})
52         ;
53     }
↕ // ... lines 54 - 58
59 }
```

This is pretty cool: call `$this->afterInstantiate()`, pass it a callback and, inside say if `$user->getPlainPassword()` - just in case we override that to `null` - then `$user->setPassword()`. Generate the hash with

`$this->passwordHasher->hashPassword()` passing the user that we're trying to hash - so `$user` - and then whatever the plain password is: `$user->getPlainPassword()`:

```
src/Factory/UserFactory.php
↕ // ... lines 1 - 29
30 final class UserFactory extends ModelFactory
31 {
↕ // ... lines 32 - 49
50     protected function initialize(): self
51     {
52         // see
53         https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#initialia
54         return $this
55             ->afterInstantiate(function(User $user) {
56                 if ($user->getPlainPassword()) {
57                     $user->setPassword(
58                         $this->passwordHasher->hashPassword($user, $user->
59                         >getPlainPassword())
60                     );
61                 }
62             })
63         ;
64     }
↕ // ... lines 63 - 67
68 }
```

Done! Let's try this. Find your terminal and run:

```
symfony console doctrine:fixtures:load
```

This will take a bit longer than before because hashing passwords is actually CPU intensive. But... it works! Check the `user` table:

```
symfony console doctrine:query:sql 'SELECT * FROM user'
```

And... got it! Every user has a hashed version of the password!

Validating the Password: PasswordCredentials

Finally we're ready to *check* the user's password inside our authenticator. To do this, we need to hash the submitted plain password then safely *compare* that with the hash in the database.

Well we don't need to do this... because Symfony is going to do it automatically. Check it out: replace `CustomCredentials` with a new `PasswordCredentials` and pass it the plain-text submitted password:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 17
18 use
   Symfony\Component\Security\Http\Authenticator\Passport\Credentials\PasswordC
↕ // ... lines 19 - 21
22 class LoginFormAuthenticator extends AbstractAuthenticator
23 {
↕ // ... lines 24 - 37
38     public function authenticate(Request $request): PassportInterface
39     {
↕ // ... lines 40 - 42
43         return new Passport(
↕ // ... lines 44 - 53
54             new PasswordCredentials($password)
55         );
56     }
↕ // ... lines 57 - 83
84 }
```

That's it! Try it. Log in using our real user - `abraca_admin@example.com` - I'll copy that, then some *wrong* password. Nice! Invalid password! Now enter the *real* password `tada`. It works!

That's awesome! When you put a `PasswordCredentials` inside your `Passport`, Symfony automatically uses that to compare the submitted password to the hashed password of the user in the database. I *love* that.

This is all possible thanks to a powerful event listener system inside of security. Let's learn more about that next and see how we can leverage it to add CSRF protection to our login form... with about two lines of code.

Chapter 13: Security Listener System & Csrp Protection

After we return the `Passport` object, we know that two things happen. First, the `UserBadge` is used to get the `User` object:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 21
22 class LoginFormAuthenticator extends AbstractAuthenticator
23 {
↕ // ... lines 24 - 37
38     public function authenticate(Request $request): PassportInterface
39     {
↕ // ... lines 40 - 42
43         return new Passport(
44             new UserBadge($email, function($userIdentifier) {
45                 // optionally pass a callback to load the User manually
46                 $user = $this->userRepository->findOneBy(['email' =>
47                     $userIdentifier]);
48                 if (!$user) {
49                     throw new UserNotFoundException();
50                 }
51                 return $user;
52             })),
↕ // ... line 54
55         );
56     }
↕ // ... lines 57 - 83
84 }
```

In our case, because we passed this a second argument, it just calls our function, and we do the work. But if you only pass one argument, then the user provider does the work.

The second thing that happens is that the "credentials badge" is "resolved":

```

src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 21
22 class LoginFormAuthenticator extends AbstractAuthenticator
23 {
↕ // ... lines 24 - 37
38     public function authenticate(Request $request): PassportInterface
39     {
↕ // ... lines 40 - 42
43         return new Passport(
↕ // ... lines 44 - 53
54             new PasswordCredentials($password)
55         );
56     }
↕ // ... lines 57 - 83
84 }

```

Originally it did this by executing our callback. Now it checks the user's password in the database.

The Event System in Action

All of this is powered by a really cool event system. After our `authenticate()` method, the security system dispatches several events... and there are a set of *listeners* to these events that do different work. We're going to see a full list of these listeners later... and even add our *own* listeners to the system.

UserProviderListener

But let's look at a few of them. Hit `Shift+Shift` so we can load some core files from Symfony. The first is called `UserProviderListener`. Make sure to "Include non-project items"... and open it up.

This is called after we return our `Passport`. It first checks to make sure the `Passport` has a `UserBadge` - it always will in any normal situation - and then *grabs* that object. It *then* checks to see if the badge has a "user loader": that's the function that we're passing to the second argument of our `UserBadge`. If the badge *already* has a user loader, like in our case, it does nothing. But if it does *not*, it *sets* the user loader to the `loadUserByIdentifier()` method on our user provider.

It's... a little technical... but *this* is what causes our user provider in `security.yaml` to be responsible for loading the user if we only pass one argument to `UserBadge`.

CheckCredentialsListener

Let's check one other class. Close this one and hit `Shift+Shift` to open `CheckCredentialsListener`. As the name suggests, *this* is responsible for checking the user's "credentials". It first checks to see if the `Passport` has a `PasswordCredentials` badge. Even though its name doesn't sound like it, the "credentials" objects are just badges... like any other badge. So this checks to see if the `Passport` has that badge and if it *does*, it grabs the badge, reads the plain-text password off of it, and, eventually way down here, uses the password hasher to *verify* that the password is correct. So this contains *all* of that password hashing logic. Below, this listener also handles the `CustomCredentials` badge.

Badges Must be Resolved

So your `Passport` always has at least these two badges: the `UserBadge` and also some sort of "credentials badge". One important property of badges is that each one must be "resolved". You can see this in `CheckCredentialsListener`. After it finishes checking the password, it calls `$badge->markResolved()`. If, for some reason, this `CheckCredentialsListener` was never called due to some misconfiguration... the badge would *remain* "unresolved" and that would actually cause authentication to *fail*. Yup, after calling the listeners, Symfony checks to make sure that all badges have been resolved. This means that you can confidently return `PasswordCredentials` and not have to wonder if something *did* actually verify that password.

Adding CSRF Protection

And here's where things start to get more interesting. In addition to these two badges, we can also add *more* badges to our `Passport` to activate more super powers. For example, one good thing to have on a login form is CSRF protection. Basically you add a hidden field to your form that contains a CSRF token... then, on submit, you *validate* that token.

Let's do this. Anywhere inside your form, add an input `type="hidden"`, `name="_csrf_token"` - this name could be anything, but this is a standard name - then

`value="{{ csrf_token() }}"`. Pass this the string `authenticate`:

```
templates/security/login.html.twig
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <form method="post" class="row g-3">
↕ // ... lines 10 - 24
25                 <input type="hidden" name="_csrf_token"
26                     value="{{ csrf_token('authenticate') }}"
27                 >
↕ // ... lines 28 - 33
34             </form>
35         </div>
36     </div>
37 </div>
38 {% endblock %}
```

That `authenticate` could *also* be anything... it's like a unique name for this form.

Now that we have the field, copy its name and head over to `LoginFormAuthenticator`. Here, we need to read that field from the POST data and then ask Symfony:

"Is this CSRF token valid?"

Well, in reality, that second part will happen automatically.

How? The `Passport` object has a *third* argument: an array of any *other* badges that we want to add. Add one: a new `CsrfTokenBadge()`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 15
16 use
    Symfony\Component\Security\Http\Authenticator\Passport\Badge\CsrfTokenBadge;
↕ // ... lines 17 - 22
23 class LoginFormAuthenticator extends AbstractAuthenticator
24 {
↕ // ... lines 25 - 38
39     public function authenticate(Request $request): PassportInterface
40     {
↕ // ... lines 41 - 43
44         return new Passport(
↕ // ... lines 45 - 55
56             [
57                 new CsrfTokenBadge(
↕ // ... lines 58 - 59
60                     )
61             ]
62         );
63     }
↕ // ... lines 64 - 90
91 }
```

This needs two things. The first is the CSRF token ID. Say `authenticate`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 22
23 class LoginFormAuthenticator extends AbstractAuthenticator
24 {
↕ // ... lines 25 - 38
39     public function authenticate(Request $request): PassportInterface
40     {
↕ // ... lines 41 - 43
44         return new Passport(
↕ // ... lines 45 - 55
56             [
57                 new CsrfTokenBadge(
58                     'authenticate',
↕ // ... line 59
60                 )
61             ]
62         );
63     }
↕ // ... lines 64 - 90
91 }
```

this just needs to match whatever we used in the form. The second argument is the submitted value, which is `$request->request->get()` and the name of our field: `_csrf_token`:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 22
23 class LoginFormAuthenticator extends AbstractAuthenticator
24 {
↕ // ... lines 25 - 38
39     public function authenticate(Request $request): PassportInterface
40     {
↕ // ... lines 41 - 43
44         return new Passport(
↕ // ... lines 45 - 55
56             [
57                 new CsrfTokenBadge(
58                     'authenticate',
59                     $request->request->get('_csrf_token')
60                 )
61             ]
62         );
63     }
↕ // ... lines 64 - 90
91 }
```

And... we're done! Internally, a listener will notice this badge, validate the CSRF token and *resolve* the badge.

Let's try it! Go to `/login`, inspect the form... and find the hidden field. There it is. Enter any email, any password... but mess with the CSRF token value. Hit "Sign in" and... yes! Invalid CSRF token! Now if we *don't* mess with the token... and use any email and password... beautiful! The CSRF token was valid... so it continued to the email error.

Next: let's leverage Symfony's "remember me" system to allow users to stay logged in for a long time. This feature *also* leverages the listener system and a badge.

Chapter 14: Remember Me System

Another nice feature of a login form is a "remember me" checkbox. This is where we store a long-lived "remember me" cookie in the user's browser so that when they *close* their browser - and so, lose their session - that cookie will keep them logged in... for a week... or a year... or whatever we configure. Let's add this.

Enabling the remember_me System

The first step is to go to `config/packages/security.yaml` and activate the system. We do this by saying `remember_me:` and then, below, setting one required piece of config: `secret:` set to `%kernel.secret%`:

```
config/packages/security.yaml
1  security:
  ↕ // ... lines 2 - 16
17  firewalls:
  ↕ // ... lines 18 - 20
21  main:
  ↕ // ... lines 22 - 27
28      remember_me:
29          secret: '%kernel.secret%'
  ↕ // ... lines 30 - 42
```

This is used to "sign" the remember me cookie value... and the `kernel.secret` parameter actually comes from our `.env` file:

```
.env
  ↕ // ... lines 1 - 15
16  ###> symfony/framework-bundle ###
  ↕ // ... line 17
18  APP_SECRET=c28f3d37eba278748f3c0427b313e86a
19  ###
  ↕ // ... lines 20 - 28
```

Yup, this `APP_SECRET` ends up becoming the `kernel.secret` parameter... which we can reference here.

Like normal, there are a bunch of other options that you can put under `remember_me`... and you can see many of them by running:

```
symfony console debug:config security
```

Look for the `remember_me:` section. One important one is `lifetime:`, which is how long the remember me cookie will be valid for.

Earlier, I said that most of the configuration that we put under our firewall serves to *activate* different authenticators. For example, `custom_authenticator:` activates our `LoginFormAuthenticator`:

```
config/packages/security.yaml
```

```
1 security:
  // ... lines 2 - 16
17   firewalls:
  // ... lines 18 - 20
21     main:
  // ... lines 22 - 23
24       custom_authenticator: App\Security\LoginFormAuthenticator
  // ... lines 25 - 42
```

Which means that our class is now called at the start of every request and looks for a login form submit. The `remember_me` config *also* activates an authenticator: a core authenticator called `RememberMeAuthenticator`. On every request, this looks for a "remember me" cookie - that we'll create in a second - and, if it's there, uses it to authenticate the user.

Adding the Remember Me Checkbox

Now that this is in place, our *next* job is to *set* that cookie on the user's browser after they log in. Open up `login.html.twig`. Instead of *always* adding the cookie, let's let the user choose. Right after the password, add a div with some classes, a label and an input `type="checkbox"`, `name="_remember_me"`:

templates/security/login.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <form method="post" class="row g-3">
↕ // ... lines 10 - 24
25                 <div class="form-check mb-3">
26                     <label>
27                         <input type="checkbox" name="_remember_me"
class="form-check-input"> Remember me
28                     </label>
29                 </div>
↕ // ... lines 30 - 39
40             </form>
41         </div>
42     </div>
43 </div>
44 {% endblock %}
```

The name - `_remember_me` - is important and *needs* to be that value. As we'll see in a minute, the system *looks* for a checkbox with this exact name.

Ok, refresh the form. Cool, we have a checkbox! Though... it's a little ugly... I think messed something up. Use `form-check` and let's give our checkbox `form-check-input`:

templates/security/login.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <form method="post" class="row g-3">
↕ // ... lines 10 - 24
25                 <div class="form-check mb-3">
26                     <label>
27                         <input type="checkbox" name="_remember_me"
class="form-check-input"> Remember me
28                     </label>
29                 </div>
↕ // ... lines 30 - 39
40             </form>
41         </div>
42     </div>
43 </div>
44 {% endblock %}
```

Now... better!

Opting into the Remember Me Cookie

If we checked the box and submitted... absolutely *nothing* different would happen: Symfony would *not* set a remember me cookie.

That's because our authenticator needs to *advertise* that it supports remember me cookies being set. This is a little weird, but think about it: just because we activated the `remember_me` system in `security.yaml` doesn't mean that we ALWAYS want remember me cookies to be set. In a login form, definitely. But if we had some sort of API token authentication... then we wouldn't want Symfony to try to set a remember me cookie on that API request.

Anyways, all we need to add is a little flag that says that this authentication mechanism *does* support adding remember me cookies. Do this with a badge: new `RememberMeBadge()`:

```

src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 16
17 use
    Symfony\Component\Security\Http\Authenticator\Passport\Badge\RememberMeBadge
↕ // ... lines 18 - 23
24 class LoginFormAuthenticator extends AbstractAuthenticator
25 {
↕ // ... lines 26 - 39
40     public function authenticate(Request $request): PassportInterface
41     {
↕ // ... lines 42 - 44
45         return new Passport(
46             new UserBadge($email, function($userIdentifier) {
↕ // ... lines 47 - 55
56                 new PasswordCredentials($password),
57                 [
↕ // ... lines 58 - 61
62                     new RememberMeBadge(),
63                 ]
64             );
65         }
↕ // ... lines 66 - 92
93     }

```

That's it! But there's one kind of odd thing. With the `CsrfTokenBadge`, we *read* the POSTed token and passed it to the badge. But with `RememberMeBadge`... we *don't* do that. Instead, internally, the remember me system knows to look for a check box called, exactly, `_remember_me`.

The entire process works like this. After we successfully authenticate, the remember me system will look for this badge *and* look to see if this checkbox is checked. If both are true, it will add the remember me cookie.

Let's see this in action. Refresh the page... and enter our normal email, password "tada", click the remember me checkbox... and hit "Sign in". Authentication successful! No surprise. But now open your browser tools, go to "Application", find "Cookies" and... yes! We have a new `REMEMBERME` cookie... which expires a *long* time from now: that's in 1 year!

Watching the RememberMe Cookie Authenticate Us

To prove the system works, delete the session cookie that normally keeps us logged in. Watch what happens when we refresh. We're still logged in! *That* is thanks to the `remember_me`

authenticator.

In the web debug toolbar, you can see a slight difference: it's this token class. When you authenticate, internally, your `User` object is wrapped in a "token" object... which usually isn't too important. But that token shows *how* you were authenticated. Now it says `RememberMeToken`... which proves that the remember me cookie was what authenticated us.

Oh, and if you're wondering why Symfony didn't add a new session cookie... that's only because Symfony's session is lazy. You won't see it until you go to a page that uses the session - like the login page. Now it's back.

And... that's really it! In addition to our `LoginFormAuthenticator`, there is now a *second* authenticator that looks for authentication information on a `REMEMBERME` cookie.

Though, we *can* make all of this a bit fancier. Next, let's see how we could add a remember me cookie for *all* users when they log in, *without* needing a checkbox. We're also going to explore a brand-new option on the remember me system that allows you to *invalidate* all existing remember me cookies if the user changes their password.

Chapter 15: Always Remember Me & "signature_properties"

Now that we've got the remember me system working, let's play with it! Instead of giving the user the *option* to enable "remember me", could we... just enable it always?

Sure! In this case, we no longer need a remember me checkbox... so we delete that entirely.

always_remember_me: true

There are two ways that you can "force" the remember me system to *always* set a cookie even though the checkbox isn't there. The first is in `security.yaml`: set `always_remember_me:` to `true`:

```
config/packages/security.yaml
1  security:
  ⬆ ⬆ // ... lines 2 - 16
17  firewalls:
  ⬆ ⬆ // ... lines 18 - 20
21  main:
  ⬆ ⬆ // ... lines 22 - 27
28      remember_me:
  ⬆ ⬆ // ... line 29
30      always_remember_me: true
  ⬆ ⬆ // ... lines 31 - 43
```

Yes, I *totally* just misspelled `remember` ... so don't do that!

With this, our authenticator *still* needs to add a `RememberMeBadge`:

```

src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 23
24 class LoginFormAuthenticator extends AbstractAuthenticator
25 {
↕ // ... lines 26 - 39
40     public function authenticate(Request $request): PassportInterface
41     {
↕ // ... lines 42 - 44
45         return new Passport(
↕ // ... lines 46 - 55
56             new PasswordCredentials($password),
57             [
↕ // ... lines 58 - 61
62                 new RememberMeBadge(),
63             ]
64         );
65     }
↕ // ... lines 66 - 92
93 }

```

But the system will *no* longer look for that checkbox. As long as it sees this badge, it will add the cookie.

Enabling on the RememberMeBadge

The *other* way that you can enable the remember me cookie in all situations is via the badge itself. Comment-out the new option. Well... let me fix my typo and *then* comment it out:

```

config/packages/security.yaml
1 security:
↕ // ... lines 2 - 16
17     firewalls:
↕ // ... lines 18 - 20
21         main:
↕ // ... lines 22 - 27
28             remember_me:
↕ // ... line 29
30                 #always_remember_me: true
↕ // ... lines 31 - 43

```

Inside of `LoginFormAuthenticator`, on the badge itself, you can call `->enable()`... which returns the badge instance:

```

src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 23
24 class LoginFormAuthenticator extends AbstractAuthenticator
25 {
↕ // ... lines 26 - 39
40     public function authenticate(Request $request): PassportInterface
41     {
↕ // ... lines 42 - 44
45         return new Passport(
↕ // ... lines 46 - 55
56             new PasswordCredentials($password),
57             [
↕ // ... lines 58 - 61
62                 (new RememberMeBadge())->enable(),
63             ]
64         );
65     }
↕ // ... lines 66 - 92
93 }

```

This says:

"I don't care about any other settings or the checkbox: I definitely want the remember me system to add a cookie."

Let's try it! Clear the session *and* REMEMBERME cookie. This time when we login... oh, invalid CSRF token! That's because I just killed my session without refreshing - silly Ryan! Refresh and try again.

Beautiful! We have the REMEMBERME cookie!

Securing Remember Me Cookies: Invalidate on User Data Change

There *is* one thing that you need to be careful with when it comes to remember me cookies. If a bad user somehow got access to my account - like they stole my password - then they could, of course, log in. Normally, that sucks... but as soon as I find out, I could change my password, which will log them out.

But... if that bad user has a REMEMBERME cookie... then even if I change my password, they will *stay* logged in until that cookie expires... which could be a long time from now. These cookies

are almost as good as the real thing: they act like "free authentication tickets". And they keep working - no matter what we do - until they expire.

💡 Tip

You no longer need to add `signature_properties: [password]`: Symfony now adds it automatically!

Fortunately, in the new authenticator system, there's a really cool way to avoid this. In `security.yaml`, below `remember_me`, add a new option called `signature_properties` set to an array with `password` inside:

```
config/packages/security.yaml
1  security:
  // ... lines 2 - 16
17  firewalls:
  // ... lines 18 - 20
21    main:
  // ... lines 22 - 27
28      remember_me:
  // ... line 29
30          signature_properties: [password]
  // ... lines 31 - 44
```

Let me explain. When Symfony creates the remember me cookie, it creates a "signature" that proves that this cookie is valid. Thanks to this config, it will now fetch the `password` property off of our `User` and include that in the signature. Then, when that cookie is used to *authenticate*, Symfony will re-create the signature using the `password` of the `User` that's currently in the *database* and make sure the two signatures match. So if the `password` in the database is different than the password that was used to originally create the cookie... the signature match will *fail*!

In other words, for any properties in this list, if even *one* of these changes in the database on that `User`, *all* remember me cookies for that user will instantly be invalidated.

So if a bad user steals my account, all I need to do is change my password and that bad user will get kicked out.

This is *super* cool to see in action. Refresh the page. If you tweak the `signature_properties` config, that will invalidate *all* `REMEMBERME` cookies on your entire system: so make sure to get the config right when you first set things up. Watch: if I delete the

session cookie and refresh... yup! I'm *not* authenticated: the `REMEMBERME` cookie didn't work. It's still there... but it's non-functional.

Let's log in - with our normal email address... and password... so that we get a new remember me cookie that's created with the hashed password.

Cool! And now, under normal conditions, things will work just like normal. I can delete the session cookie, refresh, and I'm still logged in.

But *now*, let's change the user's password in the database. We can cheat and do this on the command line:

```
symfony console doctrine:query:sql 'UPDATE user SET password="foo" WHERE email
```

Setting the password to `foo` is utter nonsense... since this column needs to hold a hashed password... but it'll be ok for our purposes. Hit it and... awesome! This imitated what would happen if I changed the password on my account.

Now, if we are the *bad* user, the next time we come back to the site... suddenly we're logged out! Blast! And I would've gotten away with it, too, if it weren't for you meddling kids! The remember me cookie is there... but it's not working. I *love* this feature.

Let's go back... and reload our fixtures to fix my password:

```
symfony console doctrine:fixtures:load
```

And... once that's done, go log in again as `abraca_admin@example.com`, password `tada`.

Next: it's time to have a power trip and start *denying* access! Let's look at `access_control`: the simplest way to block access to entire sections of your site.

Chapter 16: Denying Access, access_control & Roles

We've now talked a *lot* about authentication: the process of logging in. And... we're even logged in right now. So let's get our first look at *authorization*. That's the fun part where we get to run around and deny access to different parts of our site.

Hello access_control

The easiest way to kick someone out of your party is actually right inside of `config/packages/security.yaml`. It's via `access_control`:

config/packages/security.yaml

```
1 security:
↕ // ... lines 2 - 38
39     # Easy way to control access for large sections of your site
40     # Note: Only the *first* access control that matches will be used
41     access_control:
42         # - { path: ^/admin, roles: ROLE_ADMIN }
43         # - { path: ^/profile, roles: ROLE_USER }
```

Un-comment the first entry:

config/packages/security.yaml

```
1 security:
↕ // ... lines 2 - 40
41     access_control:
42         - { path: ^/admin, roles: ROLE_ADMIN }
43         # - { path: ^/profile, roles: ROLE_USER }
```

The `path` is a regular expression. So this basically says:

"If a URL starts with `/admin` - so `/admin` or `/admin*` - then I shall deny access unless the user has `ROLE_ADMIN`."

We'll talk more about roles in a minute... but I can tell you that our user does *not* have that role. So... let's try to go to a URL that matches this path. We actually *do* have a small admin section on our site. Make sure you're logged in... then go to `/admin`. Access denied! I've *never* been so happy to be rejected. We get kicked out with a 403 error.

On production, you can customize what this 403 error page looks like... in addition to customizing the 404 error page or 422.

Roles! User::getRoles()

So let's talk about these "roles" thingies. Open up the `User` class: `src/Entity/User.php`. Here's how this works. The moment we log in, Symfony calls this `getRoles()` method, which is part of `UserInterface`:

```
src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 78
79 /**
80  * @see UserInterface
81  */
82 public function getRoles(): array
83 {
84     $roles = $this->roles;
85     // guarantee every user at least has ROLE_USER
86     $roles[] = 'ROLE_USER';
87
88     return array_unique($roles);
89 }
↕ // ... lines 90 - 154
155 }
```

We return an array of *whatever* roles this user should have. The `make:user` command generated this so that we *always* have a role called `ROLE_USER`... plus any extra roles stored on the `$this->roles` property. That property holds an *array* of strings... which are stored in the database as JSON:

```
src/Entity/User.php
```

```
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 26
27     /**
28      * @ORM\Column(type="json")
29      */
30     private $roles = [];
↕ // ... lines 31 - 154
155 }
```

This means that we can give each user as *many* roles as we want. So far, when we've created our users, we haven't given them *any* roles yet... so our `roles` property is empty. But thanks to how the `getRoles()` method is written, every user at *least* has `ROLE_USER`. The `make:user` command generated the code like this because *all* users need to have a least *one* role... otherwise they wander around our site like half-dead zombie users. It's... not pretty.

So, by convention, we always give a user at *least* `ROLE_USER`. Oh, and the only *rule* about *roles* - that's a mouthful - is that they must start with `ROLE_`. Later in the tutorial, we'll learn why.

Anyways, the moment we log in, Symfony calls `getRoles()`, we return the array of roles, and it stores them. We can actually see this if we click the security icon on the web debug toolbar. Yup! Roles: `ROLE_USER`.

So then, when we go to `/admin`, this matches our first `access_control` entry, it checks to see if we have `ROLE_ADMIN`, we don't, and it denies access.

Only ONE access_control Matches

Oh, but there's one important detail to know about `access_control`: only *one* will ever be matched on a request.

For example, suppose you had two access controls like this:

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


If we went to `/admin`, that would match the *first* rule and *only* use the first rule. It works like routing: it goes down the access control list one-by-one and as soon as it finds the *first* match, it stops, and uses *only* that entry.

This will *help* us later when we deny access to *all* of a section *except* for one URL. But for now, just be aware of it!

And... that's it. Access controls give us a *really* easy way to secure entire sections of our site. But it's just *one* way to deny access. Soon we'll talk about how we can deny access on a controller-by-controller basis, which I really like.

But before we do, I know that if I try to access this page without `ROLE_ADMIN`, I get the 403 forbidden error. But what if I try to access this page as an *anonymous* user? Go to `/logout`? We're now *not* logged in.

Go back to `/admin` and... whoa! An error!

"Full authentication is required to access this resource."

Next, let's talk about the "entry point" of your firewall: the way that you help anonymous users start the login process.

Chapter 17: The Entry Point: Inviting Users to Log In

Log back in using `abraca_admin@example.com` and password `tada`. When we go to `/admin`, like we saw earlier, we get "Access Denied". This is because of the `access_control`... and the fact that our user does *not* have `ROLE_ADMIN`.

But if we change this to `ROLE_USER` - a role that we *do* have - then access is granted:

```
config/packages/security.yaml
1  security:
  ↕ // ... lines 2 - 40
41  access_control:
42      - { path: ^/admin, roles: ROLE_USER }
  ↕ // ... lines 43 - 44
```

And we get to see some *pretty* impressive graphs.

Let's try one more thing. Log out - so manually go to `/logout`. Now that we are *not* logged in, if I went directly to `/admin`: what should happen?

Well, right now, we get a *big* error page with a 401 status code. But... that's not what we want! If an anonymous user tries to access a protected page on our site, instead of an error, we want to be super friendly and *invite* them to log in. Because we have a login form, it means that we want to redirect the user to the login page.

Hello Entry Point!

To figure out what to do when an anonymous user accesses a protected page, each firewall defines something called an "entry point". The entry point of a firewall is literally a function that says:

"Here's what we should do when an anonymous user tries to access a protected page!"

Each authenticator under our firewall may or may *not* "provide" an entry point. Right now, we have two authenticators: our custom `LoginFormAuthenticator` and also the `remember_me` authenticator:

```
config/packages/security.yaml
1  security:
  ↕ // ... lines 2 - 16
17  firewalls:
  ↕ // ... lines 18 - 20
21  main:
  ↕ // ... lines 22 - 23
24      custom_authenticator: App\Security\LoginFormAuthenticator
  ↕ // ... lines 25 - 27
28      remember_me:
29          secret: '%kernel.secret%'
30          signature_properties: [password]
31          #always_remember_me: true
  ↕ // ... lines 32 - 44
```

But neither of these provides an entry point, which is why, instead of redirecting the user to a page... or something different, we get this generic 401 error. Some built-in authenticators - like `form_login`, which we'll talk about soon - *do* provide an entry point... and we'll see that.

Making our Authenticator an Entry Point

But anyways, none of our authenticators provide an entry point... so let's add one!

Open up our authenticator: `src/Security/LoginFormAuthenticator.php`. If you want your authenticator to provide an entry point, all you need to do is implement a new interface: `AuthenticationEntryPointInterface`:

```
src/Security/LoginFormAuthenticator.php
  ↕ // ... lines 1 - 22
23  use
    Symfony\Component\Security\Http\EntryPoint\AuthenticationEntryPointInterface
24
25  class LoginFormAuthenticator extends AbstractAuthenticator implements
    AuthenticationEntryPointInterface
26  {
  ↕ // ... lines 27 - 89
90  }
```

This requires the class to have one new method... which we actually already have down here. It's called `start()`. Uncomment the method. Then, inside, very simply, we're going to redirect to the login page. I'll steal the code from above:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 22
23 use
    Symfony\Component\Security\Http\EntryPoint\AuthenticationEntryPointInterface
24
25 class LoginFormAuthenticator extends AbstractAuthenticator implements
    AuthenticationEntryPointInterface
26 {
↕ // ... lines 27 - 83
84     public function start(Request $request, AuthenticationException
        $authException = null): Response
85     {
86         return new RedirectResponse(
87             $this->router->generate('app_login')
88         );
89     }
90 }
```

And done!

As *soon* as an authenticator implements this interface, the security system will notice this and start using it. Literally, if an anonymous user tries to access a protected page, it will now call our `start()` method... and we're going to redirect them to the login page.

Watch: refresh! Boom! It knocks us over to the login page.

A Firewall has Exactly ONE Entry Point

But there's *one* important thing to understand about entry points. Each firewall can only have *one* of them. Think about: at the moment we go to `/admin` as an anonymous user.... we're not trying to log in via a login form... or via an API token. We're truly anonymous. And so, if we *did* have multiple authenticators that each provided an entry point, our firewall wouldn't know which to choose. It needs *one* entry point for *all* cases.

Right now, since only *one* of our two authenticators is providing an entry point, it knows to use this one. But what if that were *not* the case? Let's actually see what would happen. Find your terminal and generate a *second* custom authenticator:



```
symfony console make:auth
```

Make an empty authenticator... and call it `DummyAuthenticator`.

Beautiful! Like This created a new class called `DummyAuthenticator`:

```

↕ // ... lines 1 - 2
3 namespace App\Security;
4
5 use Symfony\Component\HttpFoundation\Request;
6 use Symfony\Component\HttpFoundation\Response;
7 use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
8 use Symfony\Component\Security\Core\Exception\AuthenticationException;
9 use Symfony\Component\Security\Http\Authenticator\AbstractAuthenticator;
10 use
    Symfony\Component\Security\Http\Authenticator\Passport\PassportInterface;
11
12 class DummyAuthenticator extends AbstractAuthenticator
13 {
14     public function supports(Request $request): ?bool
15     {
16         // TODO: Implement supports() method.
17     }
18
19     public function authenticate(Request $request): PassportInterface
20     {
21         // TODO: Implement authenticate() method.
22     }
23
24     public function onAuthenticationSuccess(Request $request,
    TokenInterface $token, string $firewallName): ?Response
25     {
26         // TODO: Implement onAuthenticationSuccess() method.
27     }
28
29     public function onAuthenticationFailure(Request $request,
    AuthenticationException $exception): ?Response
30     {
31         // TODO: Implement onAuthenticationFailure() method.
32     }
33
34 //     public function start(Request $request, AuthenticationException
    $authException = null): Response
35 //     {
36 //         /*
37 //             * If you would like this class to control what happens when an
    anonymous user accesses a
38 //             * protected page (e.g. redirect to /login), uncomment this
    method and make this class
39 //             * implement
    Symfony\Component\Security\Http\EntryPoint\AuthenticationEntrypointInterface
40 //             *

```

```

41 //      * For more details, see
    https://symfony.com/doc/current/security/experimental_authenticators.html#co
    the-authentication-entry-point
42 //      */
43 //  }
44 }

```

And it also updated `custom_authenticator` in `security.yaml` to use *both* custom classes:

config/packages/security.yaml

```

1  security:
↕ // ... lines 2 - 16
17  firewalls:
↕ // ... lines 18 - 20
21    main:
↕ // ... lines 22 - 23
24      custom_authenticator:
25        - App\Security\LoginFormAuthenticator
26        - App\Security\DummyAuthenticator
↕ // ... lines 27 - 46

```

In the new class, inside `supports()`, return `false`:

src/Security/DummyAuthenticator.php

```

↕ // ... lines 1 - 11
12 class DummyAuthenticator extends AbstractAuthenticator
13 {
14     public function supports(Request $request): ?bool
15     {
16         return false;
17     }
↕ // ... lines 18 - 43
44 }

```

We're... not going to turn this into a *real* authenticator.

If we stopped right now... and tried to go to `/admin`, it would *still* use the entry point from `LoginFormAuthenticator`. But *now* implement `AuthenticationEntryPointInterface`:

```
src/Security/DummyAuthenticator.php
// ... lines 1 - 10
11 use
    Symfony\Component\Security\Http\EntryPoint\AuthenticationEntryPointInterface
12
13 class DummyAuthenticator extends AbstractAuthenticator implements
    AuthenticationEntryPointInterface
14 {
// ... lines 15 - 38
39 }
```

And then go down... and uncomment the `start()` method. For the body, just `dd()` a message... because this won't ever be executed:

```
src/Security/DummyAuthenticator.php
// ... lines 1 - 12
13 class DummyAuthenticator extends AbstractAuthenticator implements
    AuthenticationEntryPointInterface
14 {
// ... lines 15 - 34
35     public function start(Request $request, AuthenticationException
    $authException = null): Response
36     {
37         dd('DummyAuthenticator::start()!');
38     }
39 }
```

Now the firewall will notice that we have *two* authenticators that *each* provide an entry point. And so, when we refresh any page, it panics! The error says:

“Run for you liiiives!”

Oh wait, it actually says:

“Because you have multiple authenticators in firewall "main", you need to set the `entry_point` key to one of your authenticators.”

And it helpfully tells us the two authenticators that we have. In other words: it's making us choose.

Copy the `entry_point` key... then, anywhere under the firewall, say `entry_point:` and then point to the `LoginFormAuthenticator` service:


```
config/packages/security.yaml
```

```
1 security:
  // ... lines 2 - 16
17   firewalls:
  // ... lines 18 - 20
21     main:
  // ... lines 22 - 23
24     entry_point: App\Security\LoginFormAuthenticator
  // ... lines 25 - 47
```

Technically we can point to *any* service that implements

`AuthenticationEntryPointInterface`... but usually I put that in my authenticator.

Now... if we go back to `/admin`.... it works fine! It knows to choose the entry point from `LoginFormAuthenticator`.

Speaking of `LoginFormAuthenticator`... um... we've been doing way too much work inside of it! That's my bad - we're doing things the hard way for... ya know... "learning"! But next, let's cut that out and leverage a base class from Symfony that will let us delete a *bunch* of code. We're also going to learn about something called `TargetPathTrait`: an intelligent way to redirect the user on success.

Chapter 18: AbstractLoginFormAuthenticator & Redirecting to Previous URL

I have a confession to make: in our authenticator, we did too much work! Yep, when you build a custom authenticator for a "login form", Symfony provides a base class that can make life much easier. Instead of extending `AbstractAuthenticator` extend

`AbstractLoginFormAuthenticator`:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 15
16 use
    Symfony\Component\Security\Http\Authenticator\AbstractLoginFormAuthenticator
↕ // ... lines 17 - 25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
↕ // ... lines 28 - 95
96 }
```

Hold `Command` or `Ctrl` to open that class. Yup, *it* extends `AbstractAuthenticator` and *also* implements `AuthenticationEntryPointInterface`. Cool! That means that we can remove our redundant `AuthenticationEntryPointInterface`:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 23
24 use
    Symfony\Component\Security\Http\EntryPoint\AuthenticationEntryPointInterface
25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
↕ // ... lines 28 - 95
96 }
```

The abstract class requires us to add one new method called `getLoginUrl()`. Head to the bottom of this class and go to "Code"-"Generate" - or `Command+N` on a Mac - and then "Implement Methods" to generate `getLoginUrl()`. For the inside, steal the code from above... and return `$this->router->generate('app_login')`:

```
src/Security/LoginFormAuthenticator.php
```

```
↕ // ... lines 1 - 25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
↕ // ... lines 28 - 91
92     protected function getLoginUrl(Request $request): string
93     {
94         return $this->router->generate('app_login');
95     }
96 }
```

The usefulness of this base class is pretty easy to see: it implements three of the methods *for* us! For example, it implements `supports()` by checking to see if the method is POST *and* if the `getLoginUrl()` string matches the *current* URL. In other words, it does *exactly* what *our* `supports()` method does. It also handles `onAuthenticationFailure()` - storing the error in the session and redirecting back to the login page - and also the entry point - `start()` - by, yet again, redirecting to `/login`.

This means that... oh yea... we can remove code! Let's see: delete `supports()`, `onAuthenticationFailure()` and also `start()`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
↕ // ... lines 28 - 36
37     public function supports(Request $request): ?bool
38     {
39         return ($request->getPathInfo() === '/login' && $request-
>isMethod('POST'));
40     }
↕ // ... lines 41 - 75
76     public function onAuthenticationFailure(Request $request,
AuthenticationException $exception): ?Response
77     {
78         $request->getSession()->set(Security::AUTHENTICATION_ERROR,
$exception);
79
80         return new RedirectResponse(
81             $this->router->generate('app_login')
82         );
83     }
84
85     public function start(Request $request, AuthenticationException
$authException = null): Response
86     {
87         return new RedirectResponse(
88             $this->router->generate('app_login')
89         );
90     }
↕ // ... lines 91 - 95
96 }
```

Much nicer:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
28     private UserRepository $userRepository;
29     private RouterInterface $router;
30
31     public function __construct(UserRepository $userRepository,
    RouterInterface $router)
32     {
↕ // ... lines 33 - 34
35     }
36
37     public function authenticate(Request $request): PassportInterface
38     {
↕ // ... lines 39 - 61
62     }
63
64     public function onAuthenticationSuccess(Request $request,
    TokenInterface $token, string $firewallName): ?Response
65     {
↕ // ... lines 66 - 68
69     }
70
71     protected function getLoginUrl(Request $request): string
72     {
↕ // ... line 73
74     }
75 }
```

Let's make sure we didn't break anything: go to `/admin` and... perfect! The `start()` method *still* redirects us to `/login`. Let's log in with `abraca_admin@example.com`, password `tada` and... yes! That still works too: it redirects us to the homepage... because that's what we're doing inside of `onAuthenticationSuccess`:

```

src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 25
26 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
27 {
↕ // ... lines 28 - 63
64     public function onAuthenticationSuccess(Request $request,
        TokenInterface $token, string $firewallName): ?Response
65     {
66         return new RedirectResponse(
67             $this->router->generate('app_homepage')
68         );
69     }
↕ // ... lines 70 - 74
75 }

```

TargetPathTrait: Smart Redirecting

But... if you think about it... that's *not* ideal. Since I was originally trying to go to `/admin`... shouldn't the system be smart enough to redirect us back there after we successfully log in? Yep! And that's totally possible.

Log back out. When an anonymous user tries to access a protected page like `/admin`, right *before* calling the entry point function, Symfony stores the current URL somewhere in the session. Thanks to this, in `onAuthenticationSuccess()`, we can *read* that URL - which is called the "target path" - and redirect *there*.

To help us do this, we can leverage a trait! At the top of the class use `TargetPathTrait`:

```

src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 24
25 use Symfony\Component\Security\Http\Util\TargetPathTrait;
26
27 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
28 {
29     use TargetPathTrait;
↕ // ... lines 30 - 81
82 }

```

Then, down in `onAuthenticationSuccess()`, we can check to see *if* a "target path" was stored in the session. We do that by saying if `$target = $this->getTargetPath()` - passing the session - `$request->getSession()` - and then the name of the firewall, which we actually have as an argument. That's that key `main`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 26
27 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
28 {
↕ // ... lines 29 - 66
67     public function onAuthenticationSuccess(Request $request,
        TokenInterface $token, string $firewallName): ?Response
68     {
69         if ($target = $this->getTargetPath($request->getSession(),
            $firewallName)) {
↕ // ... line 70
71         }
72
73         return new RedirectResponse(
74             $this->router->generate('app_homepage')
75         );
76     }
↕ // ... lines 77 - 81
82 }
```

This line does two things at once: it sets a `$target` variable to the target path *and*, in the if statement, checks to see if this has something in it. Because, if the user goes *directly* to the login page, then they *won't* have a target path in the session.

So, *if* we have a target path, redirect to it: `return new RedirectResponse($target);`

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 26
27 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
28 {
↕ // ... lines 29 - 66
67     public function onAuthenticationSuccess(Request $request,
        TokenInterface $token, string $firewallName): ?Response
68     {
69         if ($target = $this->getTargetPath($request->getSession(),
            $firewallName)) {
70             return new RedirectResponse($target);
71         }
72
73         return new RedirectResponse(
74             $this->router->generate('app_homepage')
75         );
76     }
↕ // ... lines 77 - 81
82 }
```

Done and done! If you hold `Command` or `Ctrl` and click `getTargetPath()` to jump into that core method, you can see that it's super simple: it just reads a very specific key from the session. This is the key that the security system sets when an anonymous user tries to access a protected page.

Let's try this thing! We're already logged out. Head to `/admin`. Our entry point redirects us to `/login`. But *also*, behind the scenes, Symfony just set the URL `/admin` onto that key in the session. So when we log in now with our usual email and password... awesome! We get redirected back to `/admin`!

Next: um... we're *still* doing too much work in `LoginFormAuthenticator`. Dang! It turns out that, unless we need some especially custom stuff, if you're building a login form, you can skip the custom authenticator class entirely and rely on a core authenticator from Symfony.

Chapter 19: form_login: The Built-in Authenticator

Custom authenticator classes like this give us *tons* of control. Like, imagine that, in addition to email and password fields, you needed a *third* field - like a "company" dropdown menu... and you use that value - along with the `email` - to query for the `User`. Doing that in here would be... pretty darn simple! Grab the `company` POST field, use it in your custom query and celebrate with nachos.

But a login form is a pretty common thing. And so, Symfony comes with a built-in login form authenticator that we can... just use!

Checking out the Core FormLoginAuthenticator

Let's open it up and check it out. Hit `Shift+Shift` and look for `FormLoginAuthenticator`.

The first thing to notice is that this extends the *same* base class that we do. And if you look at the methods - it references a bunch of options - but ultimately... it does the same stuff that our class does: `getLoginUrl()` generates a URL to the login page... and `authenticate()` creates a `Passport` with `UserBadge`, `PasswordCredentials`, a `RememberMeBadge` and a `CsrfTokenBadge`.

Both `onAuthenticationSuccess` and `onAuthenticationFailure` offload their work to *another* object... but if you looked inside of those, you would see that they're basically doing the same thing that we are.

Using form_login

So let's use *this* instead of our custom authenticator... which I would do in a real project unless I need the flexibility of a custom authenticator.

In `security.yaml`, comment-out our customer authenticator... and also comment-out the `entry_point` config:

```
config/packages/security.yaml
```

```
1 security:
  // ... lines 2 - 16
17   firewalls:
  // ... lines 18 - 20
21     main:
  // ... lines 22 - 23
24       #entry_point: App\Security\LoginFormAuthenticator
  // ... lines 25 - 27
28       custom_authenticator:
29         # - App\Security\LoginFormAuthenticator
  // ... lines 30 - 50
```

Replace it with a new key `form_login`. This activates that authenticator. Below, this has a *ton* of options - I'll show you them in a minute. But there are two important ones we need:

`login_path`: set to the route to your login page... so for us that's `app_login`... and also the `check_path`, which is the route that the login form *submits* to... which for us is *also* `app_login`: we submit to the same URL:

```
config/packages/security.yaml
```

```
1 security:
  // ... lines 2 - 16
17   firewalls:
  // ... lines 18 - 20
21     main:
  // ... lines 22 - 24
25       form_login:
26         login_path: app_login
27         check_path: app_login
  // ... lines 28 - 50
```

Setting the entry_point to form_login

And... that's it to start! Let's go try it! Refresh any page and... error! An error that we've seen:

"Because you have multiple authenticators on firewall "main", you need to set "entry_point" to one of them: either `DummyAuthenticator`, or `form_login`."

I mentioned earlier that *some* authenticators provide an entry point and some don't. The `remember_me` authenticator does *not* provide one... but our `DummyAuthenticator` *does* and so does `form_login`. Its entry point redirects to the login page.

So since we have multiple, we need to choose one. Set `entry_point:` to `form_login`:

```
config/packages/security.yaml
1  security:
  ⬆ ⬆ // ... lines 2 - 16
17  firewalls:
  ⬆ ⬆ // ... lines 18 - 20
21    main:
  ⬆ ⬆ // ... lines 22 - 23
24      entry_point: form_login
  ⬆ ⬆ // ... lines 25 - 50
```

Customizing the Login Form Field Names

Now if we refresh... cool: no error. So let's try to log in. Actually... I'll log out first... that still works... then go log in with `abraca_admin@example.com` password `tada`. And... ah! Another error!

"The key "_username" must be a string, NULL given."

And it's coming from `FormLoginAuthenticator::getCredentials()`. Ok, so when you use the built-in `form_login`, you need to make sure a few things are lined up. Open the login template: `templates/security/login.html.twig`. Our two fields are called `email`... and `password`:

templates/security/login.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <form method="post" class="row g-3">
↕ // ... lines 10 - 15
16                 <div class="col-12">
↕ // ... line 17
18                     <input type="email" name="email" id="inputEmail"
19                     class="form-control" required autofocus>
20                     </div>
21                     <div class="col-12">
↕ // ... line 21
22                         <input type="password" name="password"
23                         id="inputPassword" class="form-control" required>
24                         </div>
↕ // ... lines 24 - 34
35                 </div>
36             </div>
37 </div>
38 {% endblock %}
```

Whelp, it turns out that Symfony expects these fields to be called `_username` and `_password`... that's why we get this error: it's looking for an `_username` POST parameter... but it's not there. *Fortunately*, this is the type of thing you can configure.

Find your favorite terminal and run:

```
symfony console debug:config security
```

to see all of our *current* security configuration. Scroll up... and look for `form_login`... here it is. There are a *bunch* of options that allow you to control the `form_login` behavior. Two of the most important ones are `username_parameter` and `password_parameter`. Let's configure these to match our field names.

So, in `security.yaml` add `username_parameter: email` and `password_parameter: password`:

```
config/packages/security.yaml
```

```
1 security:
  ↕ // ... lines 2 - 16
17   firewalls:
  ↕ // ... lines 18 - 20
21     main:
  ↕ // ... lines 22 - 24
25     form_login:
  ↕ // ... lines 26 - 27
28         username_parameter: email
29         password_parameter: password
  ↕ // ... lines 30 - 53
```

This tells it to read the `email` POST parameter... and then it will pass that string to our user provider... which will handle querying the database.

Let's test it. Refresh to resubmit and... got it! We're logged in!

The moral of the story is this: using `form_login` lets you have a login form with less code. But while using a custom authenticator class is more work... it has infinite flexibility. So, it's your choice.

Next: let's see a few other things that we can configure on the login form *and* add a totally new-feature: pre-filling the email field when we fail login.

Chapter 20: More form_login Config

Using `form_login` isn't as flexible as a custom authenticator class... though a lot of stuff *can* be configured.

For example, right now, it's *not* checking our CSRF token. Enable that by saying `enable_csrf: true`:

```
config/packages/security.yaml
1  security:
  ↕ // ... lines 2 - 16
17  firewalls:
  ↕ // ... lines 18 - 20
21  main:
  ↕ // ... lines 22 - 24
25  form_login:
  ↕ // ... lines 26 - 29
30  enable_csrf: true
  ↕ // ... lines 31 - 54
```

That's it! Over in the options, when you enable CSRF protection, it looks for a hidden field called `_csrf_token` with the string `authenticate` used to generate it. Fortunately, in our template, we're already using *both* of those things... so this is just going to work.

Seeing the Full List of Options

And there are even *more* ways we can configure this. Remember: to get this config, I ran `debug:config security`... which shows your *current* configuration, including defaults. But not *all* options are shown here. To see a full list, run `config:dump security`.

```
symfony console config:dump security
```

Instead of showing your *actual* config, this shows a huge list of *example* config. This is a much bigger list... here's `form_login`. A lot of this we saw before... but `success_handler` and

`failure_handler` are both new. You can search the docs for these to learn how to control what happens after success or failure.

But also, later, we're going to learn about a more *global* way of hooking into the success or failure process by registering an event listener.

Rendering "last_username" On the Login Form

Anyways, we're not using our `LoginFormAuthenticator` anymore, so feel free to delete it.

And... I have good news! The *core* authenticator is doing one thing that *our* class never did! Up in `authenticate()`... this calls `getCredentials()` to read the POST data. Let me search for "session"... yup! This took me into `getCredentials()`. Anyways, after grabbing the submitted email - in this code that's stored as `$credentials['username']` - it saves that value into the session.

It's doing that so that if authentication fails, we can *read* that and pre-fill the email box on the login form.

Let's do it! Go to our controller: `src/Controller/SecurityController.php`. This `AuthenticationUtils` has one other useful method. Pass a new variable to the template called `last_username` - you can call it `last_email` if you'd like - set to `$authenticationUtils->getLastUsername()`:

```
src/Controller/SecurityController.php
↕ // ... lines 1 - 9
10 class SecurityController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function login(AuthenticationUtils $authenticationUtils):
        Response
16     {
17         return $this->render('security/login.html.twig', [
↕ // ... line 18
19             'last_username' => $authenticationUtils->getLastUsername(),
20         ]);
21     }
↕ // ... lines 22 - 29
30 }
```

Once again, this is just a helper to read a specific key off of the session.

Now, in the template - `login.html.twig` - up here on the email field, add

`value="{{ last_username }}"`:

```
templates/security/login.html.twig
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <form method="post" class="row g-3">
↕ // ... lines 10 - 15
16                 <div class="col-12">
↕ // ... line 17
18                     <input type="email" name="email" id="inputEmail"
19                     class="form-control" value="{{ last_username }}" required autofocus>
20                 </div>
↕ // ... lines 20 - 33
34             </form>
35         </div>
36     </div>
37 </div>
38 {% endblock %}
```

Cool! If we go to `/login`... it's already there from filling out the form a minute ago! If we enter a different email... yes! That sticks too.

Next: let's get back to authorization by learning how to deny access in a controller... in a number of different ways.

Chapter 21: Denying Access in a Controller

I like using access control in `security.yaml` to help me protect entire sections of my site... like everything under `/admin` requires some role:

```
config/packages/security.yaml
```

```
1 security:
  // ... lines 2 - 50
51   access_control:
52     - { path: ^/admin, roles: ROLE_USER }
  // ... lines 53 - 54
```

But most of the time, I protect my site on a controller-by-controller basis.

Open `QuestionController` and find the `new()` action:

```
src/Controller/QuestionController.php
```

```
  // ... lines 1 - 17
18 class QuestionController extends AbstractController
19 {
  // ... lines 20 - 45
46   /**
47    * @Route("/questions/new")
48    */
49   public function new()
50   {
51       return new Response('Sounds like a GREAT feature for V2!');
52   }
  // ... lines 53 - 86
87 }
```

This... obviously... is not a *real* page... but we're *totally* going to finish it someday... probably.

Let's pretend that this page *does* work and *anyone* on our site should be allowed to ask new questions... but you *do* need to be logged in to load this page. To enforce that, in the controller - on the first line - let's `$this->denyAccessUnlessGranted('ROLE_USER')`:

```

src/Controller/QuestionController.php
↕ // ... lines 1 - 17
18 class QuestionController extends AbstractController
19 {
↕ // ... lines 20 - 45
46     /**
47      * @Route("/questions/new")
48      */
49     public function new()
50     {
51         $this->denyAccessUnlessGranted( 'ROLE_USER' );
↕ // ... lines 52 - 53
54     }
↕ // ... lines 55 - 88
89 }

```

So if the user does *not* have `ROLE_USER` - which is only possible if you're *not* logged in - then deny access. Yup, denying access in a controller is just that easy.

Let's log out... then go to that page: `/questions/new`. Beautiful! Because we're anonymous, it redirected us to `/login`. Now let's log in - `abraca_admin@example.com`, password `tada` and... access granted!

If we change this to `ROLE_ADMIN`... which is *not* a role that we have, we get access denied:

```

src/Controller/QuestionController.php
↕ // ... lines 1 - 17
18 class QuestionController extends AbstractController
19 {
↕ // ... lines 20 - 45
46     /**
47      * @Route("/questions/new")
48      */
49     public function new()
50     {
51         $this->denyAccessUnlessGranted( 'ROLE_ADMIN' );
↕ // ... lines 52 - 53
54     }
↕ // ... lines 55 - 88
89 }

```

The AccessDeniedException

One cool thing about the `denyAccessUnlessGranted()` method is that we're not *returning* the value. We can just say `$this->denyAccessUnlessGranted()` and that interrupts the controller.... meaning the code down here is *never* executed.

This works because, to deny access in Symfony, you actually throw a special *exception* class: `AccessDeniedException`. This line *throws* that exception.

We can actually rewrite this code in a longer way... just for the sake of learning. This one line is identical to saying: if *not* `$this->isGranted('ROLE_ADMIN')` - `isGranted()` is *another* helper method on the base class - then throw that special exception by saying `throw $this->createAccessDeniedException()` with:

"No access for you!"

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 17
18 class QuestionController extends AbstractController
19 {
↕ // ... lines 20 - 45
46     /**
47      * @Route("/questions/new")
48      */
49     public function new()
50     {
51         if (!$this->isGranted('ROLE_ADMIN')) {
52             throw $this->createAccessDeniedException('No access for
you!');
53         }
↕ // ... lines 54 - 55
56     }
↕ // ... lines 57 - 90
91 }
```

That does the same thing as before.... and the message you pass to the exception is only going to be seen by *developers*. Hold `Command` or `Ctrl` to jump into the `createAccessDeniedException()` method... you can see that it lives in `AbstractController`. This method is so beautifully boring: it creates and returns a new `AccessDeniedException`. *This* exception is the key to denying access, and you could throw it from *anywhere* in your code.

Close that... and then go refresh. Yup, we get the same thing as before.

Denying Access with IsGranted Annotation/Attribute

There's one other interesting way to deny access in a controller... and it works if you have `sensio/framework-extra-bundle` installed, which we do. Instead of writing your security rules in PHP, you can write them as PHP annotations or attributes. Check it out: above the controller, say `@IsGranted()` - I'll hit tab to autocomplete that so I get the `use` statement - then `"ROLE_ADMIN"`:

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 12
13 use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
↕ // ... lines 14 - 18
19 class QuestionController extends AbstractController
20 {
↕ // ... lines 21 - 46
47 /**
↕ // ... line 48
49     * @IsGranted("ROLE_ADMIN")
50     */
51     public function new()
52     {
53         return new Response('Sounds like a GREAT feature for V2!');
54     }
↕ // ... lines 55 - 88
89 }
```

If we try this... access denied! We as developers see a slightly different error message, but the end user would see the same 403 error page. Oh, and if you're using PHP 8, you can use `IsGranted` as a PHP attribute instead of an annotation:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

class QuestionController extends AbstractController
{
    // ...
    /**
     * ...
     */
    #[IsGranted("ROLE_ADMIN")]
    public function new()
    {
        return new Response('Sounds like a GREAT feature for V2!');
    }
}
```

```
// ...  
}
```

Denying Access to an Entire Controller Class

One of the *coolest* things about the `IsGranted` annotation or attribute is that you can use it up on the controller *class*. So above `QuestionController`, add

`@IsGranted("ROLE_ADMIN")`:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;  
  
/**  
 * @IsGranted("ROLE_ADMIN")  
 */  
class QuestionController extends AbstractController  
{  
    // ...  
    public function new()  
    {  
        return new Response('Sounds like a GREAT feature for V2!');  
    }  
    // ...  
}
```

Suddenly, `ROLE_ADMIN` will be required to execute *any* controller in this file. I *won't* do this... because then only admin users could access my homepage, but it's a great feature.

Ok, back down in `new()`, let's change this to `ROLE_USER`... so that the page kind of works again:

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 18
19 class QuestionController extends AbstractController
20 {
↕ // ... lines 21 - 46
47     /**
↕ // ... line 48
49     * @IsGranted("ROLE_USER")
50     */
51     public function new()
52     {
↕ // ... line 53
54     }
↕ // ... lines 55 - 88
89 }
```

Right now, every user has *just* `ROLE_USER`. So next: let's start adding extra roles to some users in the database to differentiate between normal users and admins. We'll also learn how to check authorization rules in Twig so that we can conditionally render links - like "log in" or "log out" - in the right situation.

Chapter 22: Dynamic Roles

Earlier, we talked about how the moment a user logs in, Symfony calls the `getRoles()` method on the `User` object to figure out which *roles* that user will have:

```
src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 26
27     /**
28      * @ORM\Column(type="json")
29      */
30     private $roles = [];
↕ // ... lines 31 - 78
79     /**
80      * @see UserInterface
81      */
82     public function getRoles(): array
83     {
84         $roles = $this->roles;
85         // guarantee every user at least has ROLE_USER
86         $roles[] = 'ROLE_USER';
87
88         return array_unique($roles);
89     }
↕ // ... lines 90 - 154
155 }
```

This method reads a `$roles` array property that's stored in the database as JSON... then always adds `ROLE_USER` to it.

Until now, we haven't given any users any extra roles in the database... so *all* users have *just* `ROLE_USER`. You can see this in the web debug toolbar: click to jump into the profiler. Yup, we have `ROLE_USER`.

This is too boring... so let's add some true admin users! First, open `config/packages/security.yaml`... and, down under `access_control`, change this to once again require `ROLE_ADMIN`:

```
config/packages/security.yaml
```

```
1 security:
↕ // ... lines 2 - 50
51     access_control:
52         - { path: ^/admin, roles: ROLE_ADMIN }
↕ // ... lines 53 - 54
```

Remember: roles are just strings that we invent... they can be anything: `ROLE_USER`, `ROLE_ADMIN`, `ROLE_PUPPY`, `ROLE_ROLLERCOASTER`... whatever. The only rule is that they must start with `ROLE_`. Thanks to this, if we go to `/admin`... access denied!

Populating Roles in the Database

Let's add some admin users to the database. Open up the fixtures class:

`src/DataFixtures/AppFixtures.php`. Let's see... down here, we're creating one custom user and then 10 random users. Make this first user an admin: set `roles` to an array with `ROLE_ADMIN`:

```
src/DataFixtures/AppFixtures.php
```

```
↕ // ... lines 1 - 15
16 class AppFixtures extends Fixture
17 {
18     public function load(ObjectManager $manager)
19     {
↕ // ... lines 20 - 47
48         UserFactory::createOne([
49             'email' => 'abraca_admin@example.com',
50             'roles' => ['ROLE_ADMIN']
51         ]);
↕ // ... lines 52 - 57
58     }
59 }
```

Let's also create one normal user that we can use to log in. Copy the `UserFactory` code, paste, use `abraca_user@example.com`... and leave `roles` empty:


```
src/DataFixtures/AppFixtures.php
```

```
↕ // ... lines 1 - 15
16 class AppFixtures extends Fixture
17 {
18     public function load(ObjectManager $manager)
19     {
20     // ... lines 20 - 47
48         UserFactory::createOne([
49             'email' => 'abraca_admin@example.com',
50             'roles' => ['ROLE_ADMIN']
51         ]);
52         UserFactory::createOne([
53             'email' => 'abraca_user@example.com',
54         ]);
55     // ... lines 55 - 57
58     }
59 }
```

Let's do it! At your terminal, run:

```
symfony console doctrine:fixtures:load
```

When that finishes... spin over and refresh. We got logged out! That's because, when the user was loaded from the session, our user provider tried to refresh the user from the database... but the old user with its old id was gone thanks to the fixtures. Log back in.... with password `tada` and... access granted! We rock! And in the profiler, we have the two roles.

Checking for Access inside Twig

In addition to checking or enforcing roles via `access_control`... or from inside a controller, we often *also* need to check roles in Twig. For example, if the current user has `ROLE_ADMIN`, let's a link to the admin page.

Open `templates/base.html.twig`. Right after this answers link... so let me search for "answers"... there we go, add if, then use a special `is_granted()` function to check to see if the user has `ROLE_ADMIN`:

templates/base.html.twig

```
1  <!DOCTYPE html>
2  <html>
↕  // ... lines 3 - 14
15  <body>
16      <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17          <div class="container-fluid">
↕  // ... lines 18 - 26
27              <div class="collapse navbar-collapse" id="navbar-
collapsible">
28                  <ul class="navbar-nav me-auto mb-2 mb-lg-0">
↕  // ... lines 29 - 31
32                      {% if is_granted('ROLE_ADMIN') %}
↕  // ... lines 33 - 35
36                      {% endif %}
37                  </ul>
↕  // ... lines 38 - 40
41              </div>
42          </div>
43      </nav>
↕  // ... lines 44 - 48
49  </body>
50 </html>
```

It's that easy! If that's true, copy the nav link up here... paste.. send the user to `admin_dashboard` and say "Admin":

templates/base.html.twig

```
1  <!DOCTYPE html>
2  <html>
↕  // ... lines 3 - 14
15  <body>
16      <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17          <div class="container-fluid">
↕  // ... lines 18 - 26
27      <div class="collapse navbar-collapse" id="navbar-
collapsible">
28          <ul class="navbar-nav me-auto mb-2 mb-lg-0">
↕  // ... lines 29 - 31
32              {% if is_granted('ROLE_ADMIN') %}
33              <li class="nav-item">
34                  <a class="nav-link" href="{{
path('admin_dashboard') }}">Admin</a>
35              </li>
36              {% endif %}
37          </ul>
↕  // ... lines 38 - 40
41      </div>
42  </div>
43  </nav>
↕  // ... lines 44 - 48
49  </body>
50 </html>
```

When we refresh... got it!

Let's do the same with the "log in" and "sign up" links: we only need those if we are *not* logged in. Down here, to simply check if the user is logged in, use `is_granted('ROLE_USER')`... because, in our app, every user has at least that role. Add `else`, `endif`, then I'll indent. If we are logged in, we can paste to add a "Log out" link that points to the `app_logout` route:

templates/base.html.twig

```
1  <!DOCTYPE html>
2  <html>
3  // ... lines 3 - 14
15 <body>
16     <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17         <div class="container-fluid">
18         // ... lines 18 - 26
27         <div class="collapse navbar-collapse" id="navbar-
collapsible">
28         // ... lines 28 - 38
39             {% if is_granted('ROLE_USER') %}
40                 <a class="nav-link text-black-50" href="{{
path('app_logout') }}">Log Out</a>
41             {% else %}
42                 <a class="nav-link text-black-50" href="{{
path('app_login') }}">Log In</a>
43                 <a href="#" class="btn btn-dark">Sign up</a>
44             {% endif %}
45         </div>
46     </div>
47 </nav>
48 // ... lines 48 - 52
53 </body>
54 </html>
```

Cool! Refresh and... so much better. This is looking like a real site!

Next, let's learn about a few special "strings" that you can use with authorization: strings that *not* start with `ROLE_`. We'll use one of these to show how we could easily deny access to every page in a section *except* for one.

Chapter 23: The Special IS_AUTHENTICATED_Strings

If we simply need to figure out whether or not the user is currently logged in, we check for `ROLE_USER`:

```
templates/base.html.twig
1  <!DOCTYPE html>
2  <html>
3  // ... lines 3 - 14
15  <body>
16      <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17          <div class="container-fluid">
18  // ... lines 18 - 26
27      <div class="collapse navbar-collapse" id="navbar-
    collapsable">
28  // ... lines 28 - 38
39          {% if is_granted('ROLE_USER') %}
40  // ... lines 40 - 43
44          {% endif %}
45      </div>
46  </div>
47  </nav>
48  // ... lines 48 - 52
53  </body>
54  </html>
```

This works.... just because of how our app is built: it works because in `getRoles()`, we make sure that every logged in user at least has this role:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 81
82     public function getRoles(): array
83     {
84         $roles = $this->roles;
85         // guarantee every user at least has ROLE_USER
86         $roles[] = 'ROLE_USER';
87
88         return array_unique($roles);
89     }
↕ // ... lines 90 - 154
155 }
```

Checking if Logged In: IS_AUTHENTICATED_FULLY

Cool. But it *does* make me wonder: is there a more "official" way in Symfony to check if a user is logged in? It turns out, there is! Check for `is_granted('IS_AUTHENTICATED_FULLY')`:

templates/base.html.twig

```
1 <!DOCTYPE html>
2 <html>
↕ // ... lines 3 - 14
15     <body>
16         <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17             <div class="container-fluid">
↕ // ... lines 18 - 26
27                 <div class="collapse navbar-collapse" id="navbar-
collapsible">
↕ // ... lines 28 - 38
39                     {% if is_granted('IS_AUTHENTICATED_FULLY') %}
↕ // ... lines 40 - 43
44                         {% endif %}
45                 </div>
46             </div>
47         </nav>
↕ // ... lines 48 - 52
53     </body>
54 </html>
```

By the way, anything we pass to `is_granted()` in Twig - like `ROLE_USER` or `IS_AUTHENTICATED_FULLY` - we can *also* pass to the `isGranted()` method in the

controller, or `denyAccessUnlessGranted()` ... or to `access_control`. They all call the security system in the same way.

I bet you noticed that `IS_AUTHENTICATED_FULLY` does *not* start with `ROLE_`. Yup! Roles *must* start with `ROLE_` ... but this string is *not* a role: it's handled by an entirely different system: a part of the security system that simply returns `true` or `false` based on whether or not the user is logged in.

So, in practice, this should have the same effect as `ROLE_USER`. When we refresh... yup! No change.

Access Decision Log in the Profiler

Oh, but click the security link in the web debug toolbar to jump into the profiler. Scroll down to the bottom to find something called the "Access decision log". This is super cool: Symfony keeps track of *all* the times that the authorization system was called during the request and what the result was.

For example, this first check was for `ROLE_ADMIN`, which is probably coming from `access_control`: because we went to `/admin`, this rule matched and it checked for `ROLE_ADMIN`. The next check is again for `ROLE_ADMIN` - that's probably to show the admin link in Twig - and then there's the check for `IS_AUTHENTICATED_FULLY` to show the log in or log out link. Access was granted for all three of these.

Remember Me Authed: IS_AUTHENTICATED_REMEMBER

In addition to `IS_AUTHENTICATED_FULLY`, there are a couple of other special strings that you can pass into the security system. The first is `IS_AUTHENTICATED_REMEMBERED`, which is super powerful... but can be a bit confusing.

Here's how it works. If I am logged in, then I *always* have `IS_AUTHENTICATED_REMEMBERED`. That... so far should sound identical to `IS_AUTHENTICATED_FULLY`. But, there's one key difference. Suppose I log in, close my browser, open my browser, and refresh... so that I'm logged in thanks to a remember me cookie. In this situation, I *will* have `IS_AUTHENTICATED_REMEMBERED` but I *will not* have `IS_AUTHENTICATED_FULLY`. Yup, you *only* have `IS_AUTHENTICATED_FULLY` if you logged in during *this* browser session.

We can see this. Head over to your browser, open your debugging tools, go to Application and then Cookies. Oh... my remember me cookie is gone! This... was a mistake I made. Log out... then go to `security.yaml`.

Earlier, we switched from using our custom `LoginFormAuthenticator` to `form_login`. That system *totally* works with remember me cookies. But we also removed the checkbox from our login form. And, inside of our authenticator, we were relying on calling `enable()` on the `RememberMeBadge` to force the cookie to be set:

```
src/Security/LoginFormAuthenticator.php
↕ // ... lines 1 - 26
27 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
28 {
↕ // ... lines 29 - 39
40     public function authenticate(Request $request): PassportInterface
41     {
↕ // ... lines 42 - 44
45         return new Passport(
↕ // ... lines 46 - 56
57             [
↕ // ... lines 58 - 61
62                 (new RememberMeBadge())->enable(),
63             ]
64         );
65     }
↕ // ... lines 66 - 81
82 }
```

Whelp, the core `form_login` authenticator definitely adds the `RememberMeBadge`, which advertises that it opts into the "remember me" system. But it does *not* call `enable()` on it. This means that we either need to add a checkbox to the form... or, in `security.yaml`, add `always_remember_me: true`:

```
config/packages/security.yaml
1 security:
↕ // ... lines 2 - 16
17     firewalls:
↕ // ... lines 18 - 20
21         main:
↕ // ... lines 22 - 37
38             remember_me:
↕ // ... lines 39 - 40
41                 always_remember_me: true
↕ // ... lines 42 - 54
```


Let's log back in now: `abraca_admin@example.com`, password `tada` and... got it! There's my `REMEMBERME` cookie.

Ok: because we just logged in - so we "logged in during this session", we are "authenticated fully". But, if I closed my browser, which I'll imitate by deleting the session cookie - and refresh... we *do* stay logged in, but we are now logged in thanks to the remember me cookie. You can see that via the `RememberMeToken`.

And look up here! We have the "Log in" and "Sign up" links! Yup, we are now *not* `IS_AUTHENTICATED_FULLY` because we did *not* authenticate during this session.

This is a *long* way of saying that if you use remember me cookies, then most of the time you should use `IS_AUTHENTICATED_REMEMBERED` when you simply want to know whether or not the user is logged in:

```
templates/base.html.twig
1  <!DOCTYPE html>
2  <html>
3  // ... lines 3 - 14
15  <body>
16      <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17          <div class="container-fluid">
18  // ... lines 18 - 26
27      <div class="collapse navbar-collapse" id="navbar-
collapsible">
28  // ... lines 28 - 38
39          {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
40  // ... lines 40 - 43
44          {% endif %}
45      </div>
46  </div>
47  </nav>
48  // ... lines 48 - 52
53  </body>
54  </html>
```

And then, if there are a couple of parts of your site that are more sensitive - like maybe the "change password" page - then protect those with `IS_AUTHENTICATED_FULLY`. If the user tries to access this page and only has `IS_AUTHENTICATED_REMEMBERED`, Symfony will actually execute your entry point. In other words, it will redirect them to the login form.

Refresh the page and... yes! The correct links are back.

PUBLIC_ACCESS & access_control

Ok, there are a few other strings special similar to `IS_AUTHENTICATED_REMEMBERED`, but only one more that I think is useful. It's called `PUBLIC_ACCESS`... and it returns true 100% of time. Yup, *everyone* has `PUBLIC_ACCESS`, even if you're not logged in.

So... you might be thinking: how could that ever possibly be useful? Fair question!

Look again at `access_control` in `security.yaml`. To access any URL that starts with `/admin`, you need `ROLE_ADMIN`:

```
config/packages/security.yaml
```

```
1 security:
  // ... lines 2 - 50
51   access_control:
52     - { path: ^/admin, roles: ROLE_ADMIN }
  // ... lines 53 - 54
```

But pretend that we had a login page at the URL `/admin/login`.

Let's actually create a dummy controller for this. Down at the bottom of `AdminController`, add `public function adminLogin()`... with a route - `/admin/login` - and, inside, return a new `Response()` with:

"Pretend admin login page that should be public"

```
src/Controller/AdminController.php
```

```
  // ... lines 1 - 10
11 class AdminController extends AbstractController
12 {
  // ... lines 13 - 57
58   /**
59    * @Route("/admin/login")
60    */
61   public function adminLogin()
62   {
63       return new Response('Pretend admin login page, that should be
public');
64   }
65 }
```

Log out... and go to `/admin/login`. Access denied! We're redirected to `/login`. And really, if `/admin/login` were our login page, then we would get redirected to `/admin/login`... which would redirect us to `/admin/login`... which would redirect us to `/admin/login`... which would... well you get the idea: we would get stuck in a redirect loop. Yikes!

In `security.yaml`, we want to be able to require `ROLE_ADMIN` for all URLs starting with `/admin`... except for `/admin/login`. The key to do that is `PUBLIC_ACCESS`

Copy the access control and paste above. Remember: only one `access_control` matches per request and it matches from top to bottom. So we can add a new rule matching anything starting with `/admin/login` and have it require `PUBLIC_ACCESS`... which will always return true!

```
config/packages/security.yaml
1  security:
  ⬆ ⬆ // ... lines 2 - 50
51    access_control:
52      - { path: ^/admin/login, roles: PUBLIC_ACCESS }
53      - { path: ^/admin, roles: ROLE_ADMIN }
  ⬆ ⬆ // ... lines 54 - 55
```

Thanks to this, if we go to anything that starts with `/admin/login`, it will match only this one `access_control`... and access will be granted!

Try it: go to `/admin/login` and... it loads!

Next: we've talked about roles and we've talked about denying access in various different ways. So let's turn to the `User` object: how we can ask Symfony *who* is logged in.

Chapter 24: Fetching the User Object

One of the *amazing* features of our site is that you can up vote and down vote each answer. Right now, you don't even need to be logged in to do this. Let's change that.

Requiring Login to Vote

Find the controller that handles the Ajax call that's made when we vote: it's `src/Controller/AnswerController.php`... the `answerVote()` method. Ok: I want to require the user to be logged in to use this endpoint. Let's do that with an annotation... or attribute: `@IsGranted`... then select that class and hit tab so that it adds the `use` statement we need up on top. Inside, use `IS_AUTHENTICATED_REMEMBERED`:

```
src/Controller/AnswerController.php
↕ // ... lines 1 - 8
9 use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
↕ // ... lines 10 - 13
14 class AnswerController extends AbstractController
15 {
↕ // ... lines 16 - 29
30 /**
↕ // ... line 31
32 * @IsGranted("IS_AUTHENTICATED_REMEMBERED")
33 */
34 public function answerVote(Answer $answer, LoggerInterface $logger,
    Request $request, EntityManagerInterface $entityManager)
35 {
↕ // ... lines 36 - 51
52 }
53 }
```

Because we're using the remember me system, *this* is the correct way to check if the user is simply logged in.

If we stop now, because we're not logged in, we won't be able to vote. Yay! But it's going to look funny on the frontend because the vote links *are* still visible. So let's hide those.

The template for this section is `templates/answer/_answer.html.twig`. Let's see... down... here are the vote arrows. So we basically want to hide this entire `div` section if we are *not* logged in. If `is_granted('IS_AUTHENTICATED_REMEMBERED')`, find the closing `div`... here it is, and add `endif`:

```
templates/answer/_answer.html.twig
1  <li class="mb-4">
↕  // ... lines 2 - 12
13  <div class="row">
↕  // ... lines 14 - 20
21  <div class="col-2 text-end">
↕  // ... line 22
23      {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
24      <div
25          class="vote-arrows"
26          {{ stimulus_controller('answer-vote', {
27              url: path('answer_vote', {
28                  id: answer.id
29              })
30          }) }}
31      >
↕  // ... lines 32 - 44
45      </div>
46      {% endif %}
47  </div>
48  </div>
49  </li>
```

When we refresh... yes! The vote links are gone.

Fetching the User Object from a Controller

In a real app, when we save the vote to the database, we will probably also store *who* voted so we can prevent a user from voting multiple times on the same answer. We're *not* going to do that right now... but let's try something simpler: let's log a message that includes the email address of who is voting.

But wait: how do we find out *who* is logged in? In a controller, it's easy peasy: use the `$this->getUser()` shortcut. Check it out: on top, I'll say `$logger->info('')` with the message:

```
"{user} is voting on answer {answer}"
```

src/Controller/AnswerController.php

```
↕ // ... lines 1 - 13
14 class AnswerController extends AbstractController
15 {
↕ // ... lines 16 - 33
34     public function answerVote(Answer $answer, LoggerInterface $logger,
    Request $request, EntityManagerInterface $entityManager)
35     {
36         $logger->info('{user} is voting on answer {answer}!', [
↕ // ... lines 37 - 38
39         ]);
↕ // ... lines 40 - 56
57     }
58 }
```

Pass this a second argument, which is called the logger "context". This is unrelated to security... it's just kind of cool. The second argument is an array of any extra data that you want to store along with the message. For example, we can set `answer` to `$answer->getId()`:

src/Controller/AnswerController.php

```
↕ // ... lines 1 - 13
14 class AnswerController extends AbstractController
15 {
↕ // ... lines 16 - 33
34     public function answerVote(Answer $answer, LoggerInterface $logger,
    Request $request, EntityManagerInterface $entityManager)
35     {
36         $logger->info('{user} is voting on answer {answer}!', [
↕ // ... line 37
38         'answer' => $answer->getId(),
39         ]);
↕ // ... lines 40 - 56
57     }
58 }
```

And... if you use this nifty `{answer}` format, then the `answer` context will automatically be put into the message. We'll see that in a minute.

For the `user`, get the current user with `$this->getUser()`... it's that easy. This will give us the `User` object... and then we can call a method on it, like `->getUserIdentifier()`, which we know will be the email:

```
src/Controller/AnswerController.php
```

```
↕ // ... lines 1 - 13
14 class AnswerController extends AbstractController
15 {
↕ // ... lines 16 - 33
34     public function answerVote(Answer $answer, LoggerInterface $logger,
    Request $request, EntityManagerInterface $entityManager)
35     {
36         $logger->info('{user} is voting on answer {answer}!', [
37             'user' => $this->getUser()->getUserIdentifier(),
38             'answer' => $answer->getId(),
39         ]);
↕ // ... lines 40 - 56
57     }
58 }
```

Sweet! Let's test this thing! First... we need to log in - `abraca_admin@example.com`, password `tada`. And... got it! It redirected us back to `/admin/login` because, a few minutes ago, we tried to access this and were redirected to the login form. So it's technically still in the session as our "target path".

Head to the homepage, click into a question... and vote! On the web debug toolbar, we can see that Ajax call... and we can even open the *profiler* for that request by clicking the link. Head to `Logs`. Sweet!

```
"abraca_admin@example.com" is voting on answer 498"
```

Custom Base Controller Class

Back in the controller, we know that `$this->getUser()` will return *our* `User` object... which means that we can call whatever methods it has. For example, our `User` class has a `getEmail()` method:

```
src/Controller/AnswerController.php
```

```
↕ // ... lines 1 - 13
14 class AnswerController extends AbstractController
15 {
↕ // ... lines 16 - 33
34     public function answerVote(Answer $answer, LoggerInterface $logger,
    Request $request, EntityManagerInterface $entityManager)
35     {
36         $logger->info('{user} is voting on answer {answer}!', [
37             'user' => $this->getUser()->getEmail(),
↕ // ... line 38
39         ]);
↕ // ... lines 40 - 56
57     }
58 }
```

So this *will* work. But notice that my editor did *not* auto-complete that. Bummer!

Hold **Command** or **Ctrl** and click `getUser()`. This jumps us to the core `AbstractController`. Ah... the method advertises that it returns a `UserInterface`, which is true! But, more specifically, we know that this will return *our* `User` entity. Unfortunately, because this method doesn't say that, we don't get nice auto-completion.

I use `$this->getUser()` a *lot* in my controllers... so I like to "fix" this. How? By creating a custom base controller class. Inside of the `Controller/` directory, create a new class called `BaseController`.

You can make this `abstract`... because we won't ever use it directly. Make it extended `AbstractController` so that we get the normal shortcut methods:

```
src/Controller/BaseController.php
```

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

Creating a custom base controller is... just kind of a nice idea: you can add whatever extra shortcut methods you want. Then, in your *real* controllers, you extend this and... have fun! I'm *only* going to do this in `AnswerController` right now... just to save time:


```
src/Controller/AnswerController.php
```

```
↕ // ... lines 1 - 14
15 class AnswerController extends BaseController
16 {
↕ // ... lines 17 - 58
59 }
```

Anyways, if we stopped now... congratulations! This doesn't change *anything* because `BaseController` extends `AbstractController`. To solve *our* problem, we don't need to *add* a new shortcut method... we just need to give our editor a *hint* so that it knows that `getUser()` returns *our* `User` object... not just a `UserInterface`.

To do that, above the class, add `@method` then `User` then `getUser()`:

```
src/Controller/BaseController.php
```

```
↕ // ... lines 1 - 4
5 use App\Entity\User;
↕ // ... lines 6 - 7
8 /**
9  * @method User getUser()
10 */
11 abstract class BaseController extends AbstractController
12 {
13 }
```

Done! Back in `AnswerController`, re-type `getEmail()` and... yes! We get auto-completion!

Cool! So the way that you get the current user in a controller is `$this->getUser()`. But there are a few *other* places where we might need to do this, like in Twig or from a service. Let's check those out next.

Chapter 25: Custom User Methods & the User in a Service

We know how to fetch the current user object in a controller. What about from Twig? Head to `base.html.twig`. Let's see... this is where we render our "log out" and "log in" links. Let's try to render the first name of the user right here.

App.user In Twig

How? In Twig, we have access to a *single* global variable called `app`, which has lots of useful stuff on it, like `app.session` and `app.request`. It *also* has `app.user` which will be the current `User` object or `null`. So we can say `app.user.firstName`:

```
templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15 <body>
16     <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17         <div class="container-fluid">
↕ // ... lines 18 - 26
27         <div class="collapse navbar-collapse" id="navbar-
collapseable">
↕ // ... lines 28 - 38
39             {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
40                 {{ app.user.firstName }}
↕ // ... line 41
42             {% else %}
↕ // ... lines 43 - 44
45             {% endif %}
46         </div>
47     </div>
48 </nav>
↕ // ... lines 49 - 53
54 </body>
55 </html>
```

This is safe because we're *inside* of the `is_granted()` check... so we *know* there's a `User`.

Let's try it! Close the profiler, refresh the page and... perfect! Apparently my name is Tremayne!

Now that we've got this... time to make it fancier. Inside of the `is_granted()` check, I'm going to paste in a big user menu: you can get this from the code block on this page:

templates/base.html.twig

```
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15 <body>
16 <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17 <div class="container-fluid">
↕ // ... lines 18 - 26
27 <div class="collapse navbar-collapse" id="navbar-
collapsible">
↕ // ... lines 28 - 38
39 {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
40 <div class="dropdown">
41 <button
42     class="dropdown-toggle btn"
43     type="button"
44     id="user-dropdown"
45     data-bs-toggle="dropdown"
46     aria-expanded="false"
47 >
48 
51 </button>
52 <ul class="dropdown-menu dropdown-menu-end"
aria-labelledby="user-dropdown">
53 <li>
54 <a class="dropdown-item" href="#">Log
Out</a>
55 </li>
56 </ul>
57 </div>
58 {{ app.user.firstName }}
59 <a class="nav-link text-black-50" href="{{
path('app_logout') }}">Log Out</a>
60 {% else %}
↕ // ... lines 61 - 62
63 {% endif %}
64 </div>
65 </div>
66 </nav>
↕ // ... lines 67 - 71
72 </body>
73 </html>
```

This is *completely* hard-coded to start... but it renders nicely!

Let's make it dynamic... there are a few spots. For the image, I'm using an avatar API. We just need to take out the "John Doe" part and print the user's *real* first name:

`app.user.firstName`. Oh, then pipe that into `|url_encode` so it's safe to put in a URL.

Also render `app.user.firstName` inside the `alt` text:

```
templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15 <body>
16     <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17         <div class="container-fluid">
↕ // ... lines 18 - 26
27             <div class="collapse navbar-collapse" id="navbar-
collapsible">
↕ // ... lines 28 - 38
39                 {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
40                     <div class="dropdown">
41                         <button
↕ // ... lines 42 - 46
47                             >
48                             
51                         </button>
↕ // ... lines 52 - 56
57                     </div>
58                 {% else %}
↕ // ... lines 59 - 60
61                 {% endif %}
62             </div>
63         </div>
64     </nav>
↕ // ... lines 65 - 69
70 </body>
71 </html>
```

For the "log out" link, steal the `path()` function from below... and put it here:

templates/base.html.twig	
↕	// ... line 1
2	<html>
↕	// ... lines 3 - 14
15	<body>
16	<nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17	<div class="container-fluid">
↕	// ... lines 18 - 26
27	<div class="collapse navbar-collapse" id="navbar-collapsible">
↕	// ... lines 28 - 38
39	{% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
40	<div class="dropdown">
↕	// ... lines 41 - 51
52	<ul class="dropdown-menu dropdown-menu-end" aria-labelledby="user-dropdown">
53	
54	Log Out
55	
56	
57	</div>
58	{% else %}
↕	// ... lines 59 - 60
61	{% endif %}
62	</div>
63	</div>
64	</nav>
↕	// ... lines 65 - 69
70	</body>
71	</html>

Delete the old stuff at the bottom to finish this up:

templates/base.html.twig	
↕	// ... line 1
2	<html>
↕	// ... lines 3 - 14
15	<body>
16	<nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17	<div class="container-fluid">
↕	// ... lines 18 - 26
27	<div class="collapse navbar-collapse" id="navbar-collapsible">
↕	// ... lines 28 - 38
39	{% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
40	<div class="dropdown">
41	<button
42	class="dropdown-toggle btn"
43	type="button"
44	id="user-dropdown"
45	data-bs-toggle="dropdown"
46	aria-expanded="false"
47	>
48	
52	<ul class="dropdown-menu dropdown-menu-end"
53	aria-labelledby="user-dropdown">
54	
55	Log Out
56	
57	
58	</div>
59	{% else %}
↕	// ... lines 59 - 60
61	{% endif %}
62	</div>
63	</div>
64	</nav>
↕	// ... lines 65 - 69
70	</body>
71	</html>

Sweet! When we refresh.. voilà! A real user drop-down menu.

Adding Custom Methods to User

I've mentioned a few times that our `User` class is *our* class.... so we are free to add whatever methods we want to it. For example, imagine that we need to get the user's avatar URL in a few places on our site... and we don't want to repeat this long string.

Copy this and then go open the `User` class: `src/Entity/User.php`. All the way at the bottom, create a new `public function getAvatarUri()`. Give this an `int $size` argument that defaults to `32`... and a `string` return type:

```
src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 155
156     public function getAvatarUri(int $size = 32): string
157     {
↕ // ... lines 158 - 162
163     }
164 }
```

Paste the URL as an example. Let's return the first part of that... add a `?` - which I totally just forgot - then use `http_build_query()`:

```
src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 155
156     public function getAvatarUri(int $size = 32): string
157     {
158         return 'https://ui-avatars.com/api/?' . http_build_query([
↕ // ... lines 159 - 161
162         ]);
163     }
164 }
```

Pass this an array... with the first query parameter we need: `name` set to `$this->getFirstName()`.

Oh, but we can be even smarter. If you scroll up, the `firstName` property is allowed to be `null`:


```
src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 31
32     /**
33      * @ORM\Column(type="string", length=255, nullable=true)
34      */
35     private $firstName;
↕ // ... lines 36 - 163
164 }
```

It's an optional thing that a user can provide. So, back down in the method, use `getFirstName()` if it has a value... else fallback to the user's email. For `size`, which is the second query parameter, set it to `$size`... and we also need `background` set to `random` to make the images more fun:

```
src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 155
156     public function getAvatarUri(int $size = 32): string
157     {
158         return 'https://ui-avatars.com/api/?' . http_build_query([
159             'name' => $this->getFirstName() ?: $this->getEmail(),
160             'size' => $size,
161             'background' => 'random',
162         ]);
163     }
164 }
```

Thanks to this nice little method, back in `base.html.twig` we can replace all of this with `app.user.avatarUri`:

templates/base.html.twig	
↕	// ... line 1
2	<html>
↕	// ... lines 3 - 14
15	<body>
16	<nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17	<div class="container-fluid">
↕	// ... lines 18 - 26
27	<div class="collapse navbar-collapse" id="navbar-collapsible">
↕	// ... lines 28 - 38
39	{% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
40	<div class="dropdown">
41	<button
↕	// ... lines 42 - 46
47	>
48	
↕	// ... lines 52 - 56
57	</div>
58	{% else %}
↕	// ... lines 59 - 61
62	</div>
63	</div>
64	</nav>
↕	// ... lines 65 - 69
70	</body>
71	</html>

You can also say `getAvatarUri()`: both will do the same thing.

If we try it... broken image! Ryan: go add the `?` you forgot, you knucklehead.

`http_build_query` adds the `&` between the query parameters, but we still need the first `?`:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 155
156     public function getAvatarUri(int $size = 32): string
157     {
158         return 'https://ui-avatars.com/api/?' . http_build_query([
159             'name' => $this->getFirstName() ?: $this->getEmail(),
↕ // ... lines 160 - 161
162         ]);
163     }
164 }
```

Now... much better!

But we can make this even better-er! In `base.html.twig`, we're using `app.user.firstName`:

```

templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15 <body>
16 <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17 <div class="container-fluid">
↕ // ... lines 18 - 26
27 <div class="collapse navbar-collapse" id="navbar-
collapsible">
↕ // ... lines 28 - 38
39 {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
40 <div class="dropdown">
41 <button
↕ // ... lines 42 - 46
47 >
48 <img
↕ // ... line 49
50 alt="{{ app.user.firstName }} Avatar">
51 </button>
↕ // ... lines 52 - 56
57 </div>
58 {% else %}
↕ // ... lines 59 - 61
62 </div>
63 </div>
64 </nav>
↕ // ... lines 65 - 69
70 </body>
71 </html>

```

As we just saw, this *might* be empty. So let's add one more helper method to `User` called `getDisplayName()`, which will return a `string`:

```

src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 164
165 public function getDisplayName(): string
166 {
↕ // ... line 167
168 }
169 }

```

I'll steal some logic from above... and return that:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 164
165     public function getDisplayName(): string
166     {
167         return $this->getFirstName() ?: $this->getEmail();
168     }
169 }
```

So we either return the first name or the email. We can use this up in `getAvatarUri()` - `getDisplayName()`:

src/Entity/User.php

```
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 155
156     public function getAvatarUri(int $size = 32): string
157     {
158         return 'https://ui-avatars.com/api/?' . http_build_query([
159             'name' => $this->getDisplayName(),
↕ // ... lines 160 - 161
162         ]);
163     }
↕ // ... lines 164 - 168
169 }
```

And also in `base.html.twig`:

```

templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15 <body>
16 <nav class="navbar navbar-expand-lg navbar-light bg-light px-1">
17 <div class="container-fluid">
↕ // ... lines 18 - 26
27 <div class="collapse navbar-collapse" id="navbar-
  collapsable">
↕ // ... lines 28 - 38
39 {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
40 <div class="dropdown">
41 <button
↕ // ... lines 42 - 46
47 >
48 <img
↕ // ... line 49
50 alt="{{ app.user.displayName }}"
  Avatar">
51 </button>
↕ // ... lines 52 - 56
57 </div>
58 {% else %}
↕ // ... lines 59 - 61
62 </div>
63 </div>
64 </nav>
↕ // ... lines 65 - 69
70 </body>
71 </html>

```

When we refresh... yup! It still works!

Security Service: Fetching the User in a Service

Ok: we have now fetched the `User` object from a controller via `$this->getUser()`... and in Twig via `app.user`. The only other place where you'll need to fetch the `User` object is from within a *service*.

For example, a couple of tutorials ago, we created this `MarkdownHelper` service:

src/Service/MarkdownHelper.php

```
↕ // ... lines 1 - 8
9 class MarkdownHelper
10 {
↕ // ... lines 11 - 23
24     public function parse(string $source): string
25     {
26         if (stripos($source, 'cat') !== false) {
27             $this->logger->info('Meow!');
28         }
29
30         if ($this->isDebug) {
31             return $this->markdownParser->transformMarkdown($source);
32         }
33
34         return $this->cache->get('markdown_'.md5($source), function() use
($source) {
35             return $this->markdownParser->transformMarkdown($source);
36         });
37     }
38 }
```

We pass it markdown, it converts that into HTML... and then... profit... or something. Let's pretend that we need the `User` object inside of this method: we're going to use it log another message.

If you need the currently authenticated `User` object from a service, you can get it via *another* service called `Security`. Add a new argument type-hinted with `Security` - the one from `Symfony\Component` - called `$security`. Hit `Alt + Enter` and go to "Initialize properties" to create that property and set it:

```

src/Service/MarkdownHelper.php
↕ // ... lines 1 - 6
7 use Symfony\Component\Security\Core\Security;
↕ // ... lines 8 - 9
10 class MarkdownHelper
11 {
↕ // ... lines 12 - 17
18     public function __construct(MarkdownParserInterface $markdownParser,
    CacheInterface $cache, bool $isDebug, LoggerInterface $mdLogger, Security
    $security)
19     {
↕ // ... lines 20 - 23
24         $this->security = $security;
25     }
↕ // ... lines 26 - 46
47 }

```

Because I'm using PHP 7.4, this added a *type* to my property.

Down below, let's log a message *if* the user is logged in. To do this, say if

```
$this->security->getUser():
```

```

src/Service/MarkdownHelper.php
↕ // ... lines 1 - 9
10 class MarkdownHelper
11 {
↕ // ... lines 12 - 26
27     public function parse(string $source): string
28     {
29         if (stripos($source, 'cat') !== false) {
30             $this->logger->info('Meow!');
31         }
32
33         if ($this->security->getUser()) {
↕ // ... lines 34 - 36
37         }
↕ // ... lines 38 - 45
46     }
47 }

```

Really, this is the way to fetch the `User` object... but we can also use it to see if the `User` is logged in because this will return `null` if they're not. A more "official" way to do this would be to use `isGranted()` - that's another method on the `Security` class - and check for `IS_AUTHENTICATED_REMEMBERED`:


```

class MarkdownHelper
{
    // ...
    public function parse(string $source): string
    {
        // ...
        if ($this->security->isGranted('IS_AUTHENTICATED_REMEMBERED')) {
            // ...
        }
        // ...
    }
}

```

Anyways, inside say `$this->logger->info()` with:

“Rendering markdown for {user}”

Pass a context array with `user` set to `$this->security->getUser()->getEmail()`:

src/Service/MarkdownHelper.php

```

↕ // ... lines 1 - 9
10 class MarkdownHelper
11 {
↕ // ... lines 12 - 26
27     public function parse(string $source): string
28     {
↕ // ... lines 29 - 32
33         if ($this->security->getUser()) {
34             $this->logger->info('Rendering markdown for {user}', [
35                 'user' => $this->security->getUser()->getUserIdentifier()
36             ]);
37         }
↕ // ... lines 38 - 45
46     }
47 }

```

Like before, we know this will to be our `User` object... but our editor only knows that it's some `UserInterface`. So we *could* use `getEmail()`... but I'll stick with `getUserIdentifier()`:

```

src/Service/MarkdownHelper.php
↕ // ... lines 1 - 9
10 class MarkdownHelper
11 {
↕ // ... lines 12 - 26
27     public function parse(string $source): string
28     {
↕ // ... lines 29 - 32
33         if ($this->security->getUser()) {
34             $this->logger->info('Rendering markdown for {user}', [
35                 'user' => $this->security->getUser()->getUserIdentifier()
36             ]);
37         }
↕ // ... lines 38 - 45
46     }
47 }

```

Let's try it! We have markdown on this page... so refresh... then click any link on the web debug toolbar to jump into the profiler. Go to logs and... got it! There are a *bunch* of logs because we call this method a *bunch* of times.

Next, let's talk about a *super* useful feature called "role hierarchy". This gives you the power to assign *extra* roles to any user that has some *other* role.

Chapter 26: Role Hierarchy

Right now, our site has two types of users: normal users and admin users. If you're a normal user, you can vote on answers and probably do a bunch of other things once we're done. If you're an admin, you can also go to the admin section.

There's not much here yet... but in theory, an admin user might have access to edit questions, answers or manage user data. And... a lot of sites are just this simple: you're either a normal user or an admin user.

Organizing Role Names

But in a larger company, things might *not* be so simple: you might have *many* types of admin users. Some will have access to *some* sections and other access to other sections. The question is: what's the best way to organize our roles to accomplish this?

Well, there are really only two possibilities. The first is to assign roles to users that are named after the *type* of user. For example, you assign roles to users like `ROLE_HUMAN_RESOURCES` or `ROLE_IT` or `ROLE_PERSON_WHO_OWNS_THE_COMPANY`. Then, you deny access to controllers using these strings. But... I don't love this. You end up in weird situations where, in a controller, you realize that you need to allow access to `ROLE_HUMAN_RESOURCES` or `ROLE_IT`, which is just messy.

Ok, so what's the second option? To protect controllers with role names that describe what access that role gives you. For example, at the bottom of this controller, let's create a pretend admin page for moderating answers. Set the URL to `/admin/answers`... and call it `adminAnswers()`:

```
src/Controller/AdminController.php
```

```
↕ // ... lines 1 - 10
11 class AdminController extends AbstractController
12 {
↕ // ... lines 13 - 65
66     /**
67      * @Route("/admin/comments")
68      */
69     public function adminComments()
70     {
↕ // ... lines 71 - 72
73         return new Response('Pretend comments admin page');
74     }
75 }
```

Imagine that our "human resources" department *and* IT department should both have access to this. Well, as I mentioned earlier, I do *not* want to try to put logic here that allows

`ROLE_HUMAN_RESOURCES` or `ROLE_IT`.

Instead, say `$this->denyAccessUnlessGranted()` and pass this `ROLE_COMMENT_ADMIN`, a role name that I *just* invented that describes what is being protected:

```
src/Controller/AdminController.php
```

```
↕ // ... lines 1 - 10
11 class AdminController extends AbstractController
12 {
↕ // ... lines 13 - 65
66     /**
67      * @Route("/admin/comments")
68      */
69     public function adminComments()
70     {
71         $this->denyAccessUnlessGranted('ROLE_COMMENT_ADMIN');
72
73         return new Response('Pretend comments admin page');
74     }
75 }
```

Oh, dummy Ryan! I should've called this `ROLE_ANSWER_ADMIN` - I keep using "comment" when I mean "answer". This will work fine - but `ROLE_ANSWER_ADMIN` is *really* the best name.

Anyways, what I *love* about this is how clear the controller is: you can't access this unless you have a role that's *specific* to this controller. There's just one problem: if we go to

`/admin/answers`, we get access denied... because we do *not* have this role.

You can probably see the problem with this approach. Each time we create a new section and protect it with a new role name, we're going to need to add that role to every user in the database that should have access. That sounds like a pain in the butt!

Hello role hierarchy

Fortunately, Symfony has a feature *just* for this called *role hierarchy*. Open up `config/packages/security.yaml` and, anywhere inside of here... but I'll put it near the top, add `role_hierarchy`. Below this, say `ROLE_ADMIN` and set this to an array. For now, just include `ROLE_COMMENT_ADMIN`:

```
config/packages/security.yaml
1 security:
2 // ... lines 2 - 6
7     role_hierarchy:
8         ROLE_ADMIN: ['ROLE_COMMENT_ADMIN']
9 // ... lines 9 - 58
```

This looks just as simple as it is. It says:

"If you have `ROLE_ADMIN`, then you automatically also have `ROLE_COMMENT_ADMIN`."

The result? If we refresh the page, access granted!

The idea is that, for each "type" of user - like "human resources", or IT - you would create a new item in `role_hierarchy` for them, like `ROLE_HUMAN_RESOURCES` set to an array of whatever roles it should have.

For example, let's pretend that we are *also* protecting another admin controller with `ROLE_USER_ADMIN`:

```
config/packages/security.yaml
1 security:
2 // ... lines 2 - 6
7     role_hierarchy:
8 // ... line 8
9         ROLE_HUMAN_RESOURCES: ['ROLE_USER_ADMIN']
10 // ... lines 10 - 59
```

In this case, if you have `ROLE_HUMAN_RESOURCES`, then you automatically get `ROLE_USER_ADMIN`... which gives you access to modify user data. And if you have `ROLE_ADMIN`, maybe you can *also* access this section:

```
config/packages/security.yaml
1 security:
2 // ... lines 2 - 6
3
4     role_hierarchy:
5         ROLE_ADMIN: ['ROLE_COMMENT_ADMIN', 'ROLE_USER_ADMIN']
6         ROLE_HUMAN_RESOURCES: ['ROLE_USER_ADMIN']
7 // ... lines 10 - 59
```

With this setup, each time we add a new section to our site and protect it with a new role, we only need to go to `role_hierarchy` and add it to whatever groups need it. We don't need to change the roles in the database for *anyone*. And in the database, most - or all - users will only need *one* role: the one that represents the "type" of user they are, like `ROLE_HUMAN_RESOURCES`.

Speaking of admin users, when we're debugging a customer issue on our site, sometimes it would be *really* useful if we could temporarily log *into* that user's account... just to see what *they're* seeing. In Symfony, that's totally possible. Let's talk about *impersonation* next.

Chapter 27: Impersonation: switch_user

Have you ever had a situation where you're helping someone online... and it would be so much easier if you could see what *they're* seeing on their screen... or, better, if you could temporarily take over and fix the problem yourself?

"Yeah, just click the little paper clip icon to attach the file. It should be like near the bottom... a paper clip. What's "attaching a file"? Oh... it's um... like sending a "package"... but on the Internet."

Ah, memories. Symfony can't help teach your family how to attach files to an email. But! It *can* help your customer service people via a feature called impersonation. Very simply: this gives *some* users the superpower to temporarily log in as someone else.

Enabling the switch_user Authenticator

Here's how. First, we need to enable the feature. In `security.yaml`, under our firewall somewhere, add `switch_user: true`:

```
config/packages/security.yaml
1  security:
  ↕ // ... lines 2 - 20
21  firewalls:
  ↕ // ... lines 22 - 24
25      main:
  ↕ // ... lines 26 - 46
47          switch_user: true
  ↕ // ... lines 48 - 61
```

This activates a new authenticator. So we now have our `CustomAuthenticator`, `form_login`, `remember_me` and also `switch_user`.

How does it work? Well, we can now "log in" as anyone by adding `?_switch_user=` to the URL and then an email address. Head back to the fixtures file - `src/Fixtures/AppFixtures.php` - and scroll down. We have one other user whose email we know - it's `abraca_user@example.com`:

```
src/DataFixtures/AppFixtures.php
```

```
↕ // ... lines 1 - 15
16 class AppFixtures extends Fixture
17 {
18     public function load(ObjectManager $manager)
19     {
↕ // ... lines 20 - 51
52         UserFactory::createOne([
53             'email' => 'abraca_user@example.com',
54         ]);
↕ // ... lines 55 - 57
58     }
59 }
```

Copy that, paste it on the end of the URL and...

“Access Denied.”

Of course! We can't allow just *anyone* to do this. The authenticator will only allow this if we have a role called `ROLE_ALLOWED_TO_SWITCH`. Let's give this to our admin users. We can do this via `role_hierarchy`. Up here, `ROLE_ADMIN` has `ROLE_COMMENT_ADMIN` and `ROLE_USER_ADMIN`. Let's also give them `ROLE_ALLOWED_TO_SWITCH`:

```
config/packages/security.yaml
```

```
1 security:
↕ // ... lines 2 - 6
7     role_hierarchy:
8         ROLE_ADMIN: ['ROLE_COMMENT_ADMIN', 'ROLE_USER_ADMIN',
9                     'ROLE_ALLOWED_TO_SWITCH']
↕ // ... lines 9 - 61
```

And now... whoa! We switched users! That's a different user icon! And most importantly, down on the web debug toolbar, we see `abraca_user@example.com`... and it even shows who the original user is.

Behind the scenes, when we entered the email address in the URL, the `switch_user` authenticator grabbed that and then leveraged our *user provider* to load that `User` object. Remember: we have a user provider that knows how to load users from the database by querying on their `email` property:


```
config/packages/security.yaml
```

```
1 security:
↕ // ... lines 2 - 13
14     # https://symfony.com/doc/current/security.html#where-do-users-come-
    from-user-providers
15     providers:
16         # used to reload user from session & other features (e.g.
    switch_user)
17         app_user_provider:
18             entity:
19                 class: App\Entity\User
20                 property: email
↕ // ... lines 21 - 61
```

So that's why we used `email` in the URL.

To "exit" and go back to our original user, add `?_switch_user=` again with the special `_exit`.

Styling Changes During Impersonation

But before we do that, once a customer service person has switched to another account, we want to make sure they don't *forget* that they switched. So let's add a *very* obvious indicator to our page that we're currently "switched": let's make this header background red.

Open the base layout: `templates/base.html.twig`. Up on top... find the `body` and `nav...` and I'll break this onto multiple lines. How can we check to see if we are currently impersonating someone? Say `is_granted()` and pass this `ROLE_PREVIOUS_ADMIN`. If you're impersonating someone, you *will* have this role.

In that case, add `style="background-color: red"...` with `!important` to override the nav styling:

```

templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15     <body>
16         <nav
17             class="navbar navbar-expand-lg navbar-light bg-light px-1"
18             {{ is_granted('ROLE_PREVIOUS_ADMIN') ? 'style="background-
color: red !important"' }}
19         >
↕ // ... lines 20 - 66
67     </nav>
↕ // ... lines 68 - 72
73 </body>
74 </html>

```

Let's see it! Refresh and... ha! That's a *very* obvious hint that we're impersonating.

Helping the User End Impersonation

To help the user *stop* impersonation, let's add a link. Go down to the dropdown menu. Once again, check if `is_granted('ROLE_PREVIOUS_ADMIN')`. Copy the link below... paste... then send the user to - `app_homepage` but pass an extra `_switch_user` parameter set to `_exit`.

If you pass something to the second argument of `path()` that is *not* a wildcard on the route, Symfony will set it as a query parameter. So this should give us *exactly* what we want. For the text, say "Exit Impersonation":

```

templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15     <body>
16         <nav
17             class="navbar navbar-expand-lg navbar-light bg-light px-1"
18             {{ is_granted('ROLE_PREVIOUS_ADMIN') ? 'style="background-
color: red !important"' }}
19         >
20             <div class="container-fluid">
↕ // ... lines 21 - 29
30                 <div class="collapse navbar-collapse" id="navbar-
collapsible">
↕ // ... lines 31 - 41
42                     {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
43                         <div class="dropdown">
↕ // ... lines 44 - 54
55                             <ul class="dropdown-menu dropdown-menu-end"
aria-labelledby="user-dropdown">
56                                 {% if is_granted('ROLE_PREVIOUS_ADMIN') %}
57                                     <li>
58                                         <a class="dropdown-item" href="{{
path('app_homepage', {
59                                             '_switch_user': '_exit'
60                                         }) }}">Exit Impersonation</a>
61                                     </li>
62                                 {% endif %}
↕ // ... lines 63 - 65
66                             </ul>
67                         </div>
68                     {% else %}
↕ // ... lines 69 - 70
71                     {% endif %}
72                 </div>
73             </div>
74         </nav>
↕ // ... lines 75 - 79
80     </body>
81 </html>

```

Try that! Refresh. It's obvious that we're impersonating... hit "Exit Impersonation" and... we are back as `abraca_admin@example.com`. Sweet!

By the way, if you need more control over which users someone is allowed to switch to, you can listen to the `SwitchUserEvent`. To prevent switching, throw an

`AuthenticationException`. We'll talk more about event listeners later.

Next: let's take a short break to do something *totally* fun, but... kind of not related to security:
build a user API endpoint.

Chapter 28: User API & the Serializer

Most of our pages so far have been normal HTML pages. So let's create a pure API endpoint that returns JSON data about the currently-authenticated user. This might be an endpoint that we call from our own JavaScript... or maybe you're creating an API for someone *else* to consume. More on that later.

Let's create a new controller for this called `UserController`... and make it extend our `BaseController` class:

src/Controller/UserController.php

```
↕ // ... lines 1 - 2
3 namespace App\Controller;
↕ // ... lines 4 - 7
8 class UserController extends BaseController
9 {
↕ // ... lines 10 - 17
18 }
```

Inside, add a method called `apiMe()`. Give this an `@Route()` - autocomplete the one from the Symfony Component - and set the URL to `/api/me`:

src/Controller/UserController.php

```
↕ // ... lines 1 - 5
6 use Symfony\Component\Routing\Annotation\Route;
7
8 class UserController extends BaseController
9 {
10     /**
11      * @Route("/api/me")
↕ // ... line 12
13     */
14     public function apiMe()
15     {
↕ // ... line 16
17     }
18 }
```

This isn't a very *restful* endpoint, but it's often a *convenient* one to have. To require authentication to use this endpoint, add

@IsGranted("IS_AUTHENTICATED_REMEMBERED"):

```
src/Controller/UserController.php
↕ // ... lines 1 - 4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
↕ // ... lines 6 - 7
8 class UserController extends BaseController
9 {
10     /**
↕ // ... line 11
12     * @IsGranted("IS_AUTHENTICATED_REMEMBERED")
13     */
14     public function apiMe()
15     {
↕ // ... line 16
17     }
18 }
```

I'm using a mixture of annotations and PHP code to deny access in this project. Choose whichever one you like better for *your* app. Inside the method, we can just say: `return $this->json()` and pass it the current user: `$this->getUser()`:

```
src/Controller/UserController.php
↕ // ... lines 1 - 7
8 class UserController extends BaseController
9 {
10     /**
11     * @Route("/api/me")
12     * @IsGranted("IS_AUTHENTICATED_REMEMBERED")
13     */
14     public function apiMe()
15     {
16         return $this->json($this->getUser());
17     }
18 }
```

That's beautiful! Let's try it. We are logged in right now... so we can go to `/api/me` and see... absolutely nothing! Just empty braces!

By default, when you call `$this->json()`, it passes the data to Symfony's `JsonResponse` class. And then *that* class calls PHP's `json_encode()` function on our `User` object. In PHP, unless you do extra work, when you pass an object to `json_encode()`, all it does is include the *public* properties. Since our `User` class doesn't *have* any public properties:

```

src/Entity/User.php
↕ // ... lines 1 - 12
13 class User implements UserInterface, PasswordAuthenticatedUserInterface
14 {
↕ // ... lines 15 - 19
20     private $id;
↕ // ... lines 21 - 24
25     private $email;
↕ // ... lines 26 - 29
30     private $roles = [];
↕ // ... lines 31 - 34
35     private $firstName;
↕ // ... lines 36 - 39
40     private $password;
41
42     private $plainPassword;
↕ // ... lines 43 - 168
169 }

```

We get a boring response back.

Leveraging the Serializer

This... isn't good enough. So instead, let's leverage Symfony's serializer component. To get it installed, at your terminal, run:

```

composer require "serializer:1.0.4"

```

This installs the serializer pack, which includes Symfony's Serializer component as well as a few other libraries that help it work in a really smart way. But it doesn't have a recipe that does anything fancy: it just installs code.

One of the cool things about using `$this->json()` is that as soon as the Symfony serializer is installed, it will automatically start using *it* to serialize the data instead of the normal `json_encode()`. In other words, when we refresh the endpoint, it works!

Adding Serialization Groups

We're not going to talk too much about how the Symfony serializer works - we talk a lot about it in our API Platform tutorials. But let's at least get some basics.

By default, the serializer will serialize any public property or any property that has a "getter" on it. Heck, it will even serialize `displayName` - which is *not* a real property - because there is a `getDisplayName()` method.

In reality... this is too much info to include in the endpoint. So let's take more control. We can do this by telling the serializer to only serialize fields that are in a specific *group*. Pass 200 for the status code, an empty headers array - both of which are the default values - so that we can get to the fourth `$context` argument:

```
src/Controller/UserController.php
↕ // ... lines 1 - 7
8 class UserController extends BaseController
9 {
↕ // ... lines 10 - 13
14     public function apiMe()
15     {
16         return $this->json($this->getUser(), 200, [], [
↕ // ... line 17
18     });
19 }
20 }
```

This is sort of like "options" that you pass to the serializer. Pass one called `groups` set to an array. I'm going to invent a group called `user:read`... because we're "reading" from "user":

```
src/Controller/UserController.php
↕ // ... lines 1 - 7
8 class UserController extends BaseController
9 {
↕ // ... lines 10 - 13
14     public function apiMe()
15     {
16         return $this->json($this->getUser(), 200, [], [
17             'groups' => ['user:read']
18         ]);
19 }
20 }
```

Copy that group name. Now, inside the `User` entity, we need to add this group to every field that we want to include in the API. For example, let's include `id`. Above the property, add an

annotation or PHP attribute: `@Groups()`. Make sure you auto-complete the one from Symfony's serializer to get the `use` statement on top. Inside, I'll paste `user:read`:

```
src/Entity/User.php
↕ // ... lines 1 - 8
9 use Symfony\Component\Serializer\Annotation\Groups;
↕ // ... lines 10 - 13
14 class User implements UserInterface, PasswordAuthenticatedUserInterface
15 {
16     /**
↕ // ... lines 17 - 19
20     * @Groups("user:read")
21     */
22     private $id;
↕ // ... lines 23 - 175
176 }
```

Copy that and... let's expose `email`, we don't want to expose `roles`, yes to `firstName` and... that's it:

```
src/Entity/User.php
↕ // ... lines 1 - 13
14 class User implements UserInterface, PasswordAuthenticatedUserInterface
15 {
16     /**
↕ // ... lines 17 - 19
20     * @Groups("user:read")
21     */
22     private $id;
23
24     /**
↕ // ... line 25
26     * @Groups("user:read")
27     */
28     private $email;
↕ // ... lines 29 - 34
35     /**
↕ // ... line 36
37     * @Groups("user:read")
38     */
39     private $firstName;
↕ // ... lines 40 - 175
176 }
```

We could also put the group above `getDisplayName()` if we wanted to include that... or `getAvatarUri()`... actually I *will* add it there:

```
src/Entity/User.php
↕ // ... lines 1 - 13
14 class User implements UserInterface, PasswordAuthenticatedUserInterface
15 {
↕ // ... lines 16 - 159
160 /**
161  * @Groups("user:read")
162  */
163 public function getAvatarUri(int $size = 32): string
164 {
↕ // ... lines 165 - 169
170 }
↕ // ... lines 171 - 175
176 }
```

Let's try it! Refresh and... super cool! We have those 4 fields!

And notice one thing: even though this is an "API endpoint"... and our API endpoint requires us to be logged in, we can *totally* access this... even though we don't have a fancy API token authentication system. We have access thanks to our normal session cookie.

So that leads us to our next question: if you have API endpoints like this, do you need an API token authentication system or not? Let's tackle that topic next.

Chapter 29: To use API Token Authentication or Not?

Here's the million-dollar question when it comes to security and APIs: does my site need some sort of API token authentication? There's a pretty good chance that the answer is no. Even if your app has some API endpoints - like ours - if you're creating these 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.

Session-based authentication is precisely why we have access to this endpoint: we previously logged in... and our session cookie is used to authenticate us. This works *just* as well on a real page as on an API endpoint.

To prove it, before I started the tutorial, I created a Stimulus controller called `user-api_controller.js`:

```
assets/controllers/user-api_controller.js
1 import { Controller } from 'stimulus';
2 import axios from 'axios';
3
4 export default class extends Controller {
5     static values = {
6         url: String
7     }
8
9     async connect() {
10         const response = await axios.get(this.urlValue);
11
12         console.log(response.data);
13     }
14 }
```

It's dead simple: it makes an API request... and logs the result. We're going to use it to make an API request to `/api/me` to prove that Ajax calls can access the authenticated endpoints.

To activate the Stimulus controller, open `templates/base.html.twig`... and find the `body` element: that's an easy place to attach it: if

`is_granted('IS_AUTHENTICATED_REMEMBERED')`, then
`{{ stimulus_controller() }}` and the name: `user-api`:

```
templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15 <body
16     {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
17         {{ stimulus_controller('user-api', {
↕ // ... line 18
19             }) }}
20     {% endif %}
21 >
↕ // ... lines 22 - 85
86 </body>
87 </html>
```

So, our JavaScript will be called *only* if we're logged in. To pass the URL to the endpoint, add a 2nd arg with `url` set to `path('app_user_api_me')`:

```
templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15 <body
16     {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
17         {{ stimulus_controller('user-api', {
18             url: path('app_user_api_me')
19         }) }}
20     {% endif %}
21 >
↕ // ... lines 22 - 85
86 </body>
87 </html>
```

And I'm realizing that I haven't *given* our API endpoint a route name yet... so let's do that:

```
src/Controller/UserController.php
```

```
↕ // ... lines 1 - 7
8  class UserController extends BaseController
9  {
10     /**
11      * @Route("/api/me", name="app_user_api_me")
↕ // ... line 12
13     */
14     public function apiMe()
15     {
↕ // ... lines 16 - 18
19     }
20 }
```

Back in `base.html.twig`, yup! My editor looks happy now.

Ok, head back to the homepage, inspect element, go to the console and... there's my user data! The Ajax request sends the session cookie and so... authentication just works.

So if the *only* thing that needs to use your API is your own JavaScript, save yourself a lot of trouble and just use a login form. And if you *do* want to get fancy and submit your login for via Ajax, you can totally do that. In fact, if you use Turbo, that happens automatically. But if you wanted to write some custom JavaScript, it's still no problem. Just use Ajax to submit the login form and the session cookie will be automatically set like normal. If you *do* decide to do this, the only tweak you'll need is to make your login form authenticator return JSON instead of redirecting. I would probably go back to using my custom `LoginFormAuthenticator` because it would be super easy to return JSON from `onAuthenticationSuccess()`:

```
src/Security/LoginFormAuthenticator.php
```

```
↕ // ... lines 1 - 26
27 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
28 {
↕ // ... lines 29 - 66
67     public function onAuthenticationSuccess(Request $request,
        TokenInterface $token, string $firewallName): ?Response
68     {
↕ // ... lines 69 - 75
76     }
↕ // ... lines 77 - 81
82 }
```

When You Do Need API Tokens


So then, when *do* we need an API token authentication system? The answer is pretty simple: if someone *other* than your own site's JavaScript needs to access your API... including if your JavaScript lives on a completely different domain. If you have this situation, you're probably going to need some sort of API token system. Whether you need OAuth or a simpler system... depends. We won't cover API tokens in this tutorial, but we create a pretty nice system in our [Symfony 4 Security](#) tutorial, which you can check out.

Next: let's add a registration form!

Chapter 30: Registration Form

Let's add a registration form to our site. There's a funny thing about registration forms: they have basically nothing to do with security! Think about it: the point of a registration form is just to insert new users into the database. So creating a registration form is *really* not any different than creating a form to insert *any* data into your database.

And to make things *even* simpler, we're going to cheat... by generating code. Find your terminal and run:



```
symfony console make:registration-form
```

Ooh! This gives us an error! It says:

“Missing packages: run `composer require form validator`”

In this Symfony 5 series, we haven't talked about the Form component. And that's in part because it hasn't changed much since our [Symfony 4](#) tutorial. We're not going to go into too much detail about it right now, but we *do* need it to run this command. So let's install both packages:



```
composer require form validator
```

Awesome. When that finishes, run:



```
symfony console make:registration-form
```

again. Cool! So the first question asks:

"Do we want to add a `@UniqueEntity` validation annotation to our `User` class to make sure duplicate accounts aren't created."

You almost *definitely* want to say "Yes" so that the user gets a validation error if they enter an email that's already taken.

Next:

"Do you want to send an email to verify the user's email address after registration?"

We're going to add this later, but I want to do it manually. So say "No".

"Do you want to automatically authenticate the user after registration?"

That sounds awesome, but say "No", because we're *also* going to do that manually. I know, I'm making us work! The last question is:

"What route should the user be redirected to after registration?"

Let's just use our homepage route. So that's number 16 for me. And... done!

Checking out the Generated Code

This command just gave us a `RegistrationController`, a form type, and a template that renders that form. Let's... go check that stuff out!

Start with the controller: `src/Controller/RegistrationController.php`:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 12
13 class RegistrationController extends AbstractController
14 {
15     /**
16      * @Route("/register", name="app_register")
17      */
18     public function register(Request $request, UserPasswordHasherInterface
19     $userPasswordHasher): Response
20     {
21         $user = new User();
22         $form = $this->createForm(RegistrationFormType::class, $user);
23         $form->handleRequest($request);
24
25         if ($form->isSubmitted() && $form->isValid()) {
26             // encode the plain password
27             $user->setPassword(
28                 $userPasswordHasher->hashPassword(
29                     $user,
30                     $form->get('plainPassword')->getData()
31                 )
32             );
33
34             $entityManager = $this->getDoctrine()->getManager();
35             $entityManager->persist($user);
36             $entityManager->flush();
37             // do anything else you need here, like send an email
38
39             return $this->redirectToRoute('app_homepage');
40         }
41
42         return $this->render('registration/register.html.twig', [
43             'registrationForm' => $form->createView(),
44         ]);
45     }
46 }
```

Again, we're not going to talk much about the Form component. But, on a high level, this controller creates a `User` object and then, on submit, it hashes the plain password that was submitted and then saves the `User`. This is exactly the same thing that we're doing in our fixtures to create users: there's nothing special about this at all.

Fixing the Form Styling

So... let's see what this looks like! Head over to `/register` to see... the world's ugliest form! We... can do better. The template for this page is `registration/register.html.twig`. Open that up:

templates/registration/register.html.twig

```
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Register{% endblock %}
4
5  {% block body %}
6      <h1>Register</h1>
7
8      {{ form_start(registrationForm) }}
9          {{ form_row(registrationForm.email) }}
10         {{ form_row(registrationForm.plainPassword, {
11             label: 'Password'
12         }) }}
13         {{ form_row(registrationForm.agreeTerms) }}
14
15         <button type="submit" class="btn">Register</button>
16     {{ form_end(registrationForm) }}
17 {% endblock %}
```

and... I'm just going to add a couple of divs to give this more structure. Awesome... then indent all of this form stuff to be inside of those... and then we just need 3 closing divs on the bottom:

templates/registration/register.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="container">
7         <div class="row">
8             <div class="bg-light mt-4 p-4">
9                 <h1>Register</h1>
10
11                 {{ form_start(registrationForm) }}
12                     {{ form_row(registrationForm.email) }}
13                     {{ form_row(registrationForm.plainPassword, {
14                         label: 'Password'
15                     }) }}
16                     {{ form_row(registrationForm.agreeTerms) }}
17
18                     <button type="submit" class="btn">Register</button>
19                     {{ form_end(registrationForm) }}
20             </div>
21         </div>
22     </div>
23 {% endblock %}
```

Cool. That doesn't really fix the form... but at least our ugly form sort of appear in the center of the page. Oh, but let me fix my typo on the `mt-4`. And... yea, that looks better.

To fix the form itself, we can tell Symfony to output the form with markup that's Bootstrap 5-friendly. This is... kind of a topic for the form tutorial, but it's easy. Go to `config/packages/twig.yaml`. Here, add an option called `form_themes` with one new item: `bootstrap_5_layout.html.twig`:

config/packages/twig.yaml

```
1 twig:
2     default_path: '%kernel.project_dir%/templates'
3     form_themes:
4         - bootstrap_5_layout.html.twig
5
6 ↕ // ... lines 6 - 9
```

Try it now and... woh! That made a *huge* difference! Oh, but let me add one more class to that registration button... so that it's not invisible: `btn-primary`:

templates/registration/register.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <div class="container">
7         <div class="row">
8             <div class="bg-light mt-4 p-4">
↕ // ... lines 9 - 10
11                 {{ form_start(registrationForm) }}
↕ // ... lines 12 - 17
18                 <button type="submit" class="btn btn-
primary">Register</button>
19                 {{ form_end(registrationForm) }}
20             </div>
21         </div>
22     </div>
23 {% endblock %}
```

Cool.

And while we're making things look and work nicely, we can finally make the "Sign up" button.. actually go somewhere. In `base.html.twig`, search for "Sign up" - here it is - set the `href` to `path()` and target the new route, which... if we look... is called `app_register`:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 12
13 class RegistrationController extends AbstractController
14 {
15     /**
16      * @Route("/register", name="app_register")
17      */
18     public function register(Request $request, UserPasswordHasherInterface
$userPasswordHasher): Response
19     {
↕ // ... lines 20 - 43
44     }
45 }
```

So `path('app_register')`:

```

templates/base.html.twig
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15 <body>
↕ // ... lines 16 - 21
22 <nav
23     class="navbar navbar-expand-lg navbar-light bg-light px-1"
24     {{ is_granted('ROLE_PREVIOUS_ADMIN') ? 'style="background-
color: red !important"' }}
25 >
26 <div class="container-fluid">
↕ // ... lines 27 - 35
36 <div class="collapse navbar-collapse" id="navbar-
collapsible">
↕ // ... lines 37 - 47
48 {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
↕ // ... lines 49 - 73
74 {% else %}
↕ // ... line 75
76 <a href="{{ path('app_register') }}" class="btn
btn-dark">Sign up</a>
77 {% endif %}
78 </div>
79 </div>
80 </nav>
↕ // ... lines 81 - 85
86 </body>
87 </html>

```

Beautiful!

This *would* now work if we tried it. *But*, before we do, I want to add one other feature to this. After successfully submitting the registration form, I want to automatically authenticate the user. Is that possible? Of course! Let's do it next.

Chapter 31: Manual Authentication

Our registration form *would* work if we tried it. *But*, after registration, I want to *also* automatically authenticate the user... so they don't need to register and *then* immediately log in... that would be silly.

So far, all authentication has been done... kind of indirectly: the user makes a request, some authenticator handles it and... voilà! But in this case, we want to authenticate the user *directly*, by writing code *inside* of a controller.

Hello UserAuthenticatorInterface

And... this is totally possible, by autowiring a service *specifically* for this. Add a new argument up here type-hinted with `UserAuthenticatorInterface` and I'll call it

`$userAuthenticator`:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 11
12 use
   Symfony\Component\Security\Http\Authentication\UserAuthenticatorInterface;
13
14 class RegistrationController extends AbstractController
15 {
↕ // ... lines 16 - 18
19     public function register(Request $request, UserPasswordHasherInterface
   $userPasswordHasher, UserAuthenticatorInterface $userAuthenticator):
   Response
20     {
↕ // ... lines 21 - 48
49     }
50 }
```

This object allows you to just... authenticate any `User` object. Right before the redirect, let's do that: `$userAuthenticator->authenticateUser()` and we need to pass this a few arguments. The first one is the `User` we want to authenticate:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 13
14 class RegistrationController extends AbstractController
15 {
↕ // ... lines 16 - 18
19     public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, UserAuthenticatorInterface $userAuthenticator):
    Response
20     {
↕ // ... lines 21 - 24
25         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 26 - 38
39             $userAuthenticator->authenticateUser(
40                 $user,
41             );
42
43             return $this->redirectToRoute('app_homepage');
44         }
↕ // ... lines 45 - 48
49     }
50 }
```

Easy. The second is an "authenticator" that you want to use. This system works by basically taking your `User` object and... kind of "running it through" one of your authenticators.

If we were still using our custom `LoginFormAuthenticator`, passing this argument would be really easy. We could just autowire the `LoginFormAuthenticator` service up here and pass it in.

Injecting the Service for form_login

But, in our `security.yaml` file, our main way of authenticating is `form_login`:

```
config/packages/security.yaml
```

```
1 security:
  // ... lines 2 - 20
21 firewalls:
  // ... lines 22 - 24
25     main:
  // ... lines 26 - 28
29         form_login:
30             login_path: app_login
31             check_path: app_login
32             username_parameter: email
33             password_parameter: password
34             enable_csrf: true
  // ... lines 35 - 61
```

That *does* activate an authenticator service behind the scenes - just like *our* custom `LoginFormAuthenticator`. The tricky part is figuring out what that service *is* and injecting it into our controller.

So, we need to do a bit of digging. At your terminal, run

```
symfony console debug:container
```

and search for `form_login`:

```
symfony console debug:container form_login
```

In this list, I see a service called `security.authenticator.form_login.main...` and remember that "main" is the name of our firewall. *This* is the id of the service that we want. If you're wondering about the service *above* this, if you checked, you'd find that it's an "abstract" service. A, sort of "fake" service that's used as a template to create the *real* service for any firewalls that use `form_login`.

Anyways, I'll hit "1" to get more details. Ok cool: this service is an instance of `FormLoginAuthenticator`, which is the core class that we looked at earlier.

Back in our controller, add another argument type-hinted with `FormLoginAuthenticator`:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 12
13 use Symfony\Component\Security\Http\Authenticator\FormLoginAuthenticator;
14
15 class RegistrationController extends AbstractController
16 {
↕ // ... lines 17 - 19
20     public function register(Request $request, UserPasswordHasherInterface
        $userPasswordHasher, UserAuthenticatorInterface $userAuthenticator,
        FormLoginAuthenticator $formLoginAuthenticator): Response
21     {
↕ // ... lines 22 - 51
52     }
53 }
```

Then, down here, pass the new argument to `authenticateUser()`:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 14
15 class RegistrationController extends AbstractController
16 {
↕ // ... lines 17 - 19
20     public function register(Request $request, UserPasswordHasherInterface
        $userPasswordHasher, UserAuthenticatorInterface $userAuthenticator,
        FormLoginAuthenticator $formLoginAuthenticator): Response
21     {
↕ // ... lines 22 - 25
26         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 27 - 39
40             $userAuthenticator->authenticateUser(
41                 $user,
42                 $formLoginAuthenticator,
↕ // ... line 43
44             );
↕ // ... lines 45 - 46
47         }
↕ // ... lines 48 - 51
52     }
53 }
```

This *won't* work yet, but stick with me.

The *final* argument to `authenticateUser()` is the `Request` object... which we already have... it's our first controller argument:

```
src/Controller/RegistrationController.php
```

```
↕ // ... lines 1 - 14
15 class RegistrationController extends AbstractController
16 {
↕ // ... lines 17 - 19
20     public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, UserAuthenticatorInterface $userAuthenticator,
    FormLoginAuthenticator $formLoginAuthenticator): Response
21     {
↕ // ... lines 22 - 25
26         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 27 - 39
40             $userAuthenticator->authenticateUser(
41                 $user,
42                 $formLoginAuthenticator,
43                 $request
44             );
↕ // ... lines 45 - 46
47         }
↕ // ... lines 48 - 51
52     }
53 }
```

authenticateUser Returns a Response

Done! Oh, and one cool thing about `authenticateUser()` is that it returns a `Response` object! Specifically, the `Response` object from the `onAuthenticationSuccess()` method of whatever authenticator we passed in. This means that instead of redirecting to the homepage, we can return this and, wherever that authenticator *normally* redirects to, we will redirect there as well, which *could* be the "target path".

Binding the form_login Service

Let's try this thing! Refresh the registration form to be greeted with... an awesome error!

```
"Cannot autowire argument $formLoginAuthenticator."
```

Hmm. We *did* type-hint that argument with the correct class: `FormLoginAuthenticator`:

```

src/Controller/RegistrationController.php
↕ // ... lines 1 - 12
13 use Symfony\Component\Security\Http\Authenticator\FormLoginAuthenticator;
14
15 class RegistrationController extends AbstractController
16 {
↕ // ... lines 17 - 19
20     public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, UserAuthenticatorInterface $userAuthenticator,
    FormLoginAuthenticator $formLoginAuthenticator): Response
21     {
↕ // ... lines 22 - 51
52     }
53 }

```

The problem is that this is a rare example of a service that is not available for autowiring! So, we need to configure this manually.

Fortunately, if we didn't already know what service we need, the error message gives us a great hint. It says:

"... no such service exists, maybe you should alias this class to the existing security.authenticator.form_login.main service"

Yup, it gave us the id of the service that we need to wire.

Go copy the argument name - `formLoginAuthenticator` - and then open `config/services.yaml`. Beneath `_defaults`, add a new `bind` called `$formLoginAuthenticator` set to `@` then... I'll go copy that long service id... and paste it here:

```

config/services.yaml
↕ // ... lines 1 - 8
9 services:
10     # default configuration for services in *this* file
11     _defaults:
↕ // ... lines 12 - 13
14         bind:
↕ // ... line 15
16         $formLoginAuthenticator:
            '@security.authenticator.form_login.main'
↕ // ... lines 17 - 32

```

This says: whenever a service has a `$formLoginAuthenticator` argument, pass it this service.

That... if we refresh... will get rid of our error.

Ok, let's finally register a new user! I'll use my real-life email... then any password... as long as it's 6 characters: our registration form came pre-built with that validation rule. And... we got it. Down on the web debug toolbar, we are logged in as Merlin! I feel the magical power.

Next: sometimes denying access is *not* as simple as just checking a role. For example, what if you had a question edit page and it needs to only be accessible to the *creator* of that question? It's time to discover a powerful system inside of Symfony called *voters*.

Chapter 32: Making Questions owned by Users

Our site has users and these questions are created by those users. So in the database, each `Question` needs to be related to the `User` that created it via a Doctrine relationship. Right now, if you open `src/Entity/Question.php`, that is *not* the case. There's nothing that relates this back to the `User` that created it. Time to fix that. We'll need this so we can properly talk about voters!

Generating the Relationship

Find your terminal and run:



```
symfony console make:entity
```

We're going to modify the `Question` entity and add a new property called `owner`, which will be the "user" that *owns* this `Question`. We need a `ManyToOne` relationship. If you're ever not sure, just type "relation" and it will guide you through a wizard to help. This will be a relation to the `User` class... and the `owner` property will *not* be nullable: every question *must* be owned by some user.

Next it asks if we want to map the other side of the relationship so that we can say `$user->getQuestions()`. That might be handy, so let's say yes. And call that property `questions`. Finally, I'm going to say no to orphan removal. And... done!

If you went through our Doctrine relationships tutorial, then you know that there's nothing special here. This added a `ManyToOne` relationship above a new `$owner` property... and made getter and setter methods at the bottom:

src/Entity/Question.php

```
↕ // ... lines 1 - 16
17 class Question
18 {
↕ // ... lines 19 - 64
65     /**
66      * @ORM\ManyToOne(targetEntity=User::class, inversedBy="questions")
67      * @ORM\JoinColumn(nullable=false)
68      */
69     private $owner;
↕ // ... lines 70 - 227
228     public function getOwner(): ?User
229     {
230         return $this->owner;
231     }
232
233     public function setOwner(?User $owner): self
234     {
235         $this->owner = $owner;
236
237         return $this;
238     }
239 }
```

Over in the `User` class, it also mapped the *inverse* side of the relationship:

src/Entity/User.php

```
↕ // ... lines 1 - 5
6 use Doctrine\Common\Collections\ArrayCollection;
7 use Doctrine\Common\Collections\Collection;
↕ // ... lines 8 - 17
18 class User implements UserInterface, PasswordAuthenticatedUserInterface
19 {
↕ // ... lines 20 - 51
52 /**
53  * @ORM\OneToMany(targetEntity=Question::class, mappedBy="owner")
54  */
55 private $questions;
56
57 public function __construct()
58 {
59     $this->questions = new ArrayCollection();
60 }
↕ // ... lines 61 - 190
191 /**
192  * @return Collection|Question[]
193  */
194 public function getQuestions(): Collection
195 {
196     return $this->questions;
197 }
198
199 public function addQuestion(Question $question): self
200 {
201     if (!$this->questions->contains($question)) {
202         $this->questions[] = $question;
203         $question->setOwner($this);
204     }
205
206     return $this;
207 }
208
209 public function removeQuestion(Question $question): self
210 {
211     if ($this->questions->removeElement($question)) {
212         // set the owning side to null (unless already changed)
213         if ($question->getOwner() === $this) {
214             $question->setOwner(null);
215         }
216     }
217
218     return $this;
219 }
```

```
220 }
```

Let's go make a migration for this change:

```
symfony console make:migration
```

And... as usual, we'll run over to the new migration file... to make sure it contains *only* the stuff we expect. Yep: `ALTER TABLE question`, add `owner_id` and then the foreign key stuff:

```
migrations/Version20211012184326.php
```

```
↕ // ... lines 1 - 12
13 final class Version20211012184326 extends AbstractMigration
14 {
↕ // ... lines 15 - 19
20     public function up(Schema $schema): void
21     {
22         // this up() migration is auto-generated, please modify it to your
needs
23         $this->addSql('ALTER TABLE question ADD owner_id INT NOT NULL');
24         $this->addSql('ALTER TABLE question ADD CONSTRAINT
FK_B6F7494E7E3C61F9 FOREIGN KEY (owner_id) REFERENCES user (id)');
25         $this->addSql('CREATE INDEX IDX_B6F7494E7E3C61F9 ON question
(owner_id)');
26     }
27
28     public function down(Schema $schema): void
29     {
30         // this down() migration is auto-generated, please modify it to
your needs
31         $this->addSql('ALTER TABLE question DROP FOREIGN KEY
FK_B6F7494E7E3C61F9');
32         $this->addSql('DROP INDEX IDX_B6F7494E7E3C61F9 ON question');
33         $this->addSql('ALTER TABLE question DROP owner_id');
34     }
35 }
```

Fixing the Migration

Let's run that:


```
symfony console doctrine:migrations:migrate
```

And... it failed! That's okay. It fails because there are already rows in the `question` table. So adding a new `owner_id NOT NULL` makes those existing records... explode. In the Doctrine relations tutorial, we talked about how to responsibly handle, fix, and test failed migrations. Because we already talked about it there, I'm going to take the easy route here and just drop our database:

```
symfony console doctrine:database:drop --force
```

Then create a fresh database:

```
symfony console doctrine:database:create
```

And migrate again.

```
symfony console doctrine:migrations:migrate
```

Now it works. Reload the fixtures:

```
symfony console doctrine:fixtures:load
```

Assigning Owners in the Fixtures

And... that exploded too! Come on! The insert into question is failing because `owner_id` cannot be null. That makes sense: we haven't - yet - gone into our fixtures and given each question an owner.

Let's do that. Open `src/Factory/QuestionFactory.php`. Our job in `getDefaults()`, is to supply a default value for every *required* property. So I'm now going to add an `owner` key set to `UserFactory::new()`:

```
src/Factory/QuestionFactory.php
↕ // ... lines 1 - 28
29 final class QuestionFactory extends ModelFactory
30 {
↕ // ... lines 31 - 42
43     protected function getDefaults(): array
44     {
45         return [
↕ // ... lines 46 - 52
53             'owner' => UserFactory::new(),
54         ];
55     }
↕ // ... lines 56 - 68
69 }
```

Thanks to this, if we execute `QuestionFactory` *without* overriding any variables, this will create a brand new user for each new question.

But inside of *our* fixtures, that's... not exactly what we want. Head down to the bottom where we create the users. What I want to do is create these users first. And then, when we create the questions up here... oh actually right here, I want to use a random user from the ones that we already created.

To do this, we first need to move our users up to the top so that they're created first:

src/DataFixtures/AppFixtures.php

```
↕ // ... lines 1 - 15
16 class AppFixtures extends Fixture
17 {
18     public function load(ObjectManager $manager)
19     {
20         UserFactory::createOne([
21             'email' => 'abraca_admin@example.com',
22             'roles' => ['ROLE_ADMIN']
23         ]);
24         UserFactory::createOne([
25             'email' => 'abraca_user@example.com',
26         ]);
27         UserFactory::createMany(10);
28
29         TagFactory::createMany(100);
↕ // ... lines 30 - 61
62     }
63 }
```

Then, down here for our main questions, pass a function to the second argument and return an array... so that we can override the `owner` property. Set it to `UserFactory::random()`:

src/DataFixtures/AppFixtures.php

```
↕ // ... lines 1 - 15
16 class AppFixtures extends Fixture
17 {
18     public function load(ObjectManager $manager)
19     {
↕ // ... lines 20 - 30
31         $questions = QuestionFactory::createMany(20, function() {
32             return [
33                 'owner' => UserFactory::random(),
34             ];
35         });
↕ // ... lines 36 - 61
62     }
63 }
```

I'm not going to worry about also doing this for the unpublished questions down here... but we could.

Ok: let's try the fixtures again:



```
symfony console doctrine:fixtures:load
```

This time... they work!

Cool! So let's leverage the new relationship on our site to print the *real* owner of each question. We're also going to start a question edit page and then... have to figure out how to make it so that only the *owner* of each question can access it.

Chapter 33: Leveraging the Question Owner

Now that each `Question` has an `owner` - a `User` object - it's time to celebrate! On the frontend, we can start rendering *real* data... instead of always having the same cat picture and question written by the same Tisha. Those are both hard-coded, though we *do* love Tisha the cat here at SymfonyCasts.

Start on the homepage. Open up `templates/question/homepage.html.twig`. And... here's where we loop over the questions. First, for the avatar, we can use the helper method we created earlier: `{{ question.owner.avatarUri }}`:

```
templates/question/homepage.html.twig
```

```
↕ // ... lines 1 - 2
3  {% block body %}
↕ // ... lines 4 - 9
10 <div class="container">
↕ // ... lines 11 - 15
16     <div class="row">
17         {% for question in pager %}
18         <div class="col-12 mb-3">
19             <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
20                 <div class="q-container p-4">
21                     <div class="row">
22                         <div class="col-2 text-center">
23                             
↕ // ... lines 24 - 29
30                             </div>
↕ // ... lines 31 - 38
39                             </div>
40                         </div>
↕ // ... lines 41 - 45
46                     </div>
47                 </div>
48             {% endfor %}
↕ // ... lines 49 - 50
51         </div>
52     </div>
53 {% endblock %}
↕ // ... lines 54 - 55
```

Next... down towards the bottom, here's where we print the question owner's name. Let's use `question.owner.displayName`:

```
templates/question/homepage.html.twig
↕ // ... lines 1 - 2
3  {% block body %}
↕ // ... lines 4 - 9
10 <div class="container">
↕ // ... lines 11 - 15
16     <div class="row">
17         {% for question in pager %}
18         <div class="col-12 mb-3">
19             <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
20                 <div class="q-container p-4">
21                     <div class="row">
↕ // ... lines 22 - 30
31                         <div class="col">
↕ // ... line 32
33                             <div class="q-display p-3">
↕ // ... lines 34 - 35
36                                 <p class="pt-4"><strong>--{{
question.owner.displayName }}</strong></p>
37                                 </div>
38                             </div>
39                         </div>
40                     </div>
↕ // ... lines 41 - 45
46                 </div>
47             </div>
48         {% endfor %}
↕ // ... lines 49 - 50
51     </div>
52 </div>
53 {% endblock %}
↕ // ... lines 54 - 55
```

100 experience points for using *two* custom methods in a row.

And now... our page is starting to look real! Click into a question. Let's do the same thing for the show page. Open that template: `show.html.twig`.

For the avatar, use `question.owner.avatarUri`:

templates/question/show.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h2 class="my-4">Question:</h2>
10            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11                <div class="q-container-show p-4">
12                    <div class="row">
13                        <div class="col-2 text-center">
14                            
↕ // ... lines 15 - 32
33                        </div>
↕ // ... lines 34 - 41
42                    </div>
43                </div>
44            </div>
45        </div>
46    </div>
↕ // ... lines 47 - 59
60 </div>
61 {% endblock %}
```

Then... down here, for the name, `{{ question.owner.displayName }}`:

templates/question/show.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h2 class="my-4">Question:</h2>
10             <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11                 <div class="q-container-show p-4">
12                     <div class="row">
↕ // ... lines 13 - 33
34                         <div class="col">
35                             <h1 class="q-title-show">{{ question.name }}
</h1>
36                             <div class="q-display p-3">
↕ // ... lines 37 - 38
39                                     <p class="pt-4"><strong>--{{
question.owner.displayName }}</strong></p>
40                                     </div>
41                                 </div>
42                            </div>
43                        </div>
44                    </div>
45                </div>
46            </div>
↕ // ... lines 47 - 59
60 </div>
61 {% endblock %}
```

Oh, and I forgot to do one thing. Copy that, head back up to the avatar... so that we can also update the `alt` attribute:

templates/question/show.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h2 class="my-4">Question:</h2>
10            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11                <div class="q-container-show p-4">
12                    <div class="row">
13                        <div class="col-2 text-center">
14                            
↕ // ... lines 15 - 32
33                        </div>
↕ // ... lines 34 - 41
42                    </div>
43                </div>
44            </div>
45        </div>
46    </div>
↕ // ... lines 47 - 59
60 </div>
61 {% endblock %}
```

I also need to do that on the homepage... here it is:

```

templates/question/homepage.html.twig
↕ // ... lines 1 - 2
3 {% block body %}
↕ // ... lines 4 - 9
10 <div class="container">
↕ // ... lines 11 - 15
16     <div class="row">
17         {% for question in pager %}
18         <div class="col-12 mb-3">
19             <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
20                 <div class="q-container p-4">
21                     <div class="row">
22                         <div class="col-2 text-center">
23                             
↕ // ... lines 24 - 29
30                             </div>
↕ // ... lines 31 - 38
39                         </div>
40                     </div>
↕ // ... lines 41 - 45
46                 </div>
47             </div>
48         {% endfor %}
↕ // ... lines 49 - 50
51     </div>
52 </div>
53 {% endblock %}
↕ // ... lines 54 - 55

```

Let's try this! Refresh the page and... we are dynamic!

Creating the Question Edit Page

In a real site, we're probably going to need a page where the owner of this question can edit its details. We're not going to build this out all the way - I don't want to dive into the form system - but we *are* going to get it started. And this is going to lead us to a really interesting security situation.

Over in `src/Controller/QuestionController.php`... find the `show()` action. Let's cheat by copying this and pasting it. Change the URL to `/questions/edit/{slug}`, tweak the route name and update the method name. Inside, just render a template:

`question/edit.html.twig`:

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 18
19 class QuestionController extends AbstractController
20 {
↕ // ... lines 21 - 69
70     /**
71      * @Route("/questions/edit/{slug}", name="app_question_edit")
72      */
73     public function edit(Question $question)
74     {
75         return $this->render('question/edit.html.twig', [
76             'question' => $question,
77         ]);
78     }
↕ // ... lines 79 - 98
99 }
```

Cool! In `templates/question/`, create that: `edit.html.twig`.

I'll paste in a basic template:

```
templates/question/edit.html.twig
```

```
1 {% extends 'base.html.twig' %}
2
3 {% block title %}Edit Question: {{ question.name }}{% endblock %}
4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h1 class="my-4">Edit Question</h1>
10
11             <blockquote>{{ question.question }}</blockquote>
12
13             TODO
14         </div>
15     </div>
16 </div>
17 {% endblock %}
```

Nothing special here, except that I'm printing the dynamic question text. There's no actual form... since we're focusing on security... but pretend that there is.

Linking to the Edit Page

Before we try this page, head back into the question show template. Let's add an edit link to help out the owner. Actually, find the `h1`. Here we go.

Wrap this in a div with `class="d-flex justify-content-between"`... and then close and indent:

```
templates/question/show.html.twig
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h2 class="my-4">Question:</h2>
10            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11                <div class="q-container-show p-4">
12                    <div class="row">
↕ // ... lines 13 - 33
34                        <div class="col">
35                            <div class="d-flex justify-content-between">
36                                <h1 class="q-title-show">{{ question.name
}}</h1>
↕ // ... lines 37 - 40
41                            </div>
↕ // ... lines 42 - 46
47                        </div>
48                    </div>
49                </div>
50            </div>
51        </div>
52    </div>
↕ // ... lines 53 - 65
66 </div>
67 {% endblock %}
```

Now add a link with `href= path('app_question_edit')`. And, of course, we need to pass this the wildcard: `id` set to `question.id`. Oh... wait, actually, the wildcard is `slug`:

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 18
19 class QuestionController extends AbstractController
20 {
↕ // ... lines 21 - 69
70     /**
71      * @Route("/questions/edit/{slug}", name="app_question_edit")
72      */
73     public function edit(Question $question)
74     {
↕ // ... lines 75 - 77
78     }
↕ // ... lines 79 - 98
99 }
```

So use `slug` set to `question.slug`:

templates/question/show.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h2 class="my-4">Question:</h2>
10            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11                <div class="q-container-show p-4">
12                    <div class="row">
↕ // ... lines 13 - 33
34                        <div class="col">
35                            <div class="d-flex justify-content-between">
36                                <h1 class="q-title-show">{{ question.name
37                                }}</h1>
38                                <a href="{{ path('app_question_edit', {
39                                    slug: question.slug
40                                }) }}" class="btn btn-secondary btn-sm mb-
41                                2">Edit</a>
42                        </div>
↕ // ... lines 42 - 46
47                    </div>
48                </div>
49            </div>
50        </div>
51    </div>
52 </div>
↕ // ... lines 53 - 65
66 </div>
67 {% endblock %}
```

Cool. Then say "Edit"... and give this a few classes for prettiness.

Thanks to this... we have an edit button! Oh, but we need some margin! Add `mb-2`:

```

templates/question/show.html.twig
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h2 class="my-4">Question:</h2>
10            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11                <div class="q-container-show p-4">
12                    <div class="row">
↕ // ... lines 13 - 33
34                        <div class="col">
35                            <div class="d-flex justify-content-between">
↕ // ... lines 36 - 37
38                                <a href="{{ path('app_question_edit', {
39                                    slug: question.slug
40                                }}" class="btn btn-secondary btn-sm mb-
2">Edit</a>
41                            </div>
↕ // ... lines 42 - 46
47                        </div>
48                    </div>
49                </div>
50            </div>
51        </div>
52    </div>
↕ // ... lines 53 - 65
66 </div>
67 {% endblock %}

```

and... much better. Click that. This is the question edit page... which is not *really* an edit page... but pretend that it is.

Now let's circle back to the topic of security. Because... we can't just let *anyone* get to this page: only the *owner* of this question should be able to edit it.

So inside of `QuestionController`, we need a security check. We first need to make sure that the user is logged in. Do that with `$this->denyAccessUnlessGranted()` passing `IS_AUTHENTICATED_REMEMBERED`:

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 18
19 class QuestionController extends AbstractController
20 {
↕ // ... lines 21 - 72
73     public function edit(Question $question)
74     {
75         $this->denyAccessUnlessGranted('ROLE_USER');
↕ // ... lines 76 - 82
83     }
↕ // ... lines 84 - 103
104 }
```

Thanks to this, we're guaranteed to get a `User` object if we say `$this->getUser()`. We can use that: if `$question->getOwner()` does not equal `$this->getUser()`, then someone *other* than the owner is trying to access this page. Deny access with `throw $this->createAccessDeniedException()`. I'll say:

"You are not the owner!"

But, remember, these error messages are only shown to developers:

```
src/Controller/QuestionController.php
```

```
↕ // ... lines 1 - 18
19 class QuestionController extends AbstractController
20 {
↕ // ... lines 21 - 72
73     public function edit(Question $question)
74     {
75         $this->denyAccessUnlessGranted('ROLE_USER');
76         if ($question->getOwner() !== $this->getUser()) {
77             throw $this->createAccessDeniedException('You are not the
owner!');
78         }
↕ // ... lines 79 - 82
83     }
↕ // ... lines 84 - 103
104 }
```

Ok, so right now I'm *not* logged in at all. So if we refresh, it kicks us back to the login page. So... yay! We just successfully prevented anyone *other* than the owner from accessing this edit page!

But... bad news friends: I don't like this solution. I don't like putting any manual security logic inside my controller. Why? Because it means that we're going to need to repeat that logic in

Twig in order to hide or show the edit button. And what if our logic gets more complex? What if you can edit a question if you're the owner *or* if you have `ROLE_ADMIN`? Now we would need to update and maintain the duplicate logic in two places at *least*. Nope, we do *not* want to duplicate our security rules.

So next let's learn about the voter system, which is the key to centralizing all of this authorization logic in a beautiful way.

Chapter 34: Voters

When we need to deny access to something, we can do it in a couple of different places, like `access_control` in `security.yaml`:

```
config/packages/security.yaml
```

```
1 security:
  ⬆ // ... lines 2 - 54
55     # Easy way to control access for large sections of your site
56     # Note: Only the *first* access control that matches will be used
57     access_control:
58         - { path: ^/admin/login, roles: PUBLIC_ACCESS }
59         - { path: ^/admin, roles: ROLE_ADMIN }
60         # - { path: ^/profile, roles: ROLE_USER }
```

Or various ways inside of a controller. And when we deny access, we know that we can do it by checking for a role like `ROLE_ADMIN` or by checking one of the special strings like `IS_AUTHENTICATED_REMEMBERED`. It seems pretty simple, right? If we use something like `ROLE_ADMIN`, it clearly calls `getRoles()` on the `User` and denies or allows access.

Introducing: the Voter System

So all of this is... basically true. But in reality, whenever you call the authorization system - either via `access_control`, `->denyAccessUnlessGranted()`, or even the `IsGranted()` annotation/attribute, something more interesting happens internally. It activates what's called the *voter* system.

We can see this. Refresh the page and then click on the security icon in the web debug toolbar to jump into the profiler. Down near the bottom of this page, as we saw earlier, you'll find an "Access decision log" that shows all the different times that the authorization system was called during this request. Apparently it was called a *bunch* of times. Most of these represent us trying to figure out whether we should show or hide the voting links for each answer.

But check out this little "Show voter details" link. When you click, oooh. There are two "voters". The first one voted `ACCESS_DENIED` and the second voted `ACCESS_ABSTAIN`.

When you call the authorization system, it loops over these things called voters and asks each one:

"Do you know how to decide whether or not the user has `IS_AUTHENTICATED_REMEMBERED`, or `ROLE_ADMIN`... or whatever string we pass in."

In practice, exactly *one* of these voters will say that they *do* understand how to vote on that string, and they'll answer with either `ACCESS_DENIED` or `ACCESS_GRANTED`. All the other voters will return `ACCESS_ABSTAIN`... which just means that they don't want to vote one way or another.

So, for example, whenever you call the authorization system and pass it one of those `IS_AUTHENTICATED_` strings, it's this `AuthenticatedVoter` that knows how to decide whether the user has that or not.

The `RoleHierarchyVoter`, well you can probably guess. That's responsible for voting on anything that starts with `ROLE_`. Internally, that voter checks to see if the user has that role. Well technically it checks the "token"... but that's not too important. It also takes our `role_hierarchy` config into account.

And, by the way, even though this is called the "voter" system, in *all* cases, every voter *except* for one will abstain, which means they don't vote at all. You'll never have a situation where you have 5 voters and 3 vote access granted and 2 vote access denied. You *could* create voters that did that, but you won't.

Passing Custom "Attributes" into Authorization

Until now, denying access on our site has been pretty simple. We've either wanted to check to see if the user is logged in, or we've checked for a specific role.

But security isn't always that simple. For our edit question page, we can't just check for a global role. We need to check to see if the current user is the *owner* of this *question*. Yes: the security logic is specific to some *data*. In this case, the `Question` object. Putting the logic in the controller worked, but it means that we're going to have to duplicate this logic in our Twig template in order to hide or show the "edit question" link.

The way to fix this is by creating our *own* custom voter that centralizes our logic. To do this, delete all of this code and replace it with `$this->denyAccessUnlessGranted()`.

Here is where things get interesting: we're going to "invent" a new string to pass to this. These strings - which you may have thought of as "roles" until now - are actually called attributes. Say `EDIT`. I totally just made that up. You'll see how that's used in a minute.

Then, we haven't seen it yet, but you can also pass a *second* argument to `denyAccessUnlessGranted()`, which is some data related to this security check. Pass the `Question` object:

```
src/Controller/QuestionController.php
↕ // ... lines 1 - 18
19 class QuestionController extends AbstractController
20 {
↕ // ... lines 21 - 72
73     public function edit(Question $question)
74     {
75         $this->denyAccessUnlessGranted('EDIT', $question);
↕ // ... lines 76 - 79
80     }
↕ // ... lines 81 - 100
101 }
```

Ok, stop right now and click to the edit page. Ooh, we get "access denied". Well, it redirected us to the login page... but that means we didn't have access. Click any link on the web debug toolbar to jump into the profiler, click "Last 10", then find the request to the question edit page. Click to view *its* profiler info... and go down to the Security section.

At the bottom, under the "Access Decision Log", access was Denied for attribute "EDIT" and this `Question` object. If you look at the voter details... oh! They *all* abstained. So every voter said:

"I have no idea how to vote on the attribute "EDIT" and a `Question` object."

If all voters abstain, we get access denied.

Next: let's fix this by adding our own custom voter that *does* know how to vote on this situation. Once we're finished, we'll make our logic even *more* complex by *also* allowing admin users to access the edit page.

Chapter 35: Custom Voter

To make the security system understand what it means when we check for `EDIT` access on a `Question` object, we need a custom voter. And... to help us out, we can *generate* this.

Find your terminal and run:



```
symfony console make:voter
```

Let's call it `QuestionVoter`. I often have one voter class per *object* in my system that I need to check access for. And... done!

Adding the Voter Logic

Let's go check it out: `src/Security/Voter/QuestionVoter.php`:

src/Security/Voter/QuestionVoter.php

```
↕ // ... lines 1 - 2
3 namespace App\Security\Voter;
4
5 use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
6 use Symfony\Component\Security\Core\Authorization\Voter\Voter;
7 use Symfony\Component\Security\Core\User\UserInterface;
8
9 class QuestionVoter extends Voter
10 {
11     protected function supports(string $attribute, $subject): bool
12     {
13         // replace with your own logic
14         // https://symfony.com/doc/current/security/voters.html
15         return in_array($attribute, ['POST_EDIT', 'POST_VIEW'])
16             && $subject instanceof \App\Entity\Question;
17     }
18
19     protected function voteOnAttribute(string $attribute, $subject,
    TokenInterface $token): bool
20     {
21         $user = $token->getUser();
22         // if the user is anonymous, do not grant access
23         if (!$user instanceof UserInterface) {
24             return false;
25         }
26
27         // ... (check conditions and return true to grant permission) ...
28         switch ($attribute) {
29             case 'POST_EDIT':
30                 // logic to determine if the user can EDIT
31                 // return true or false
32                 break;
33             case 'POST_VIEW':
34                 // logic to determine if the user can VIEW
35                 // return true or false
36                 break;
37         }
38
39         return false;
40     }
41 }
```

As usual, the location of this class makes no difference. The important thing is that our voter implements `VoterInterface`. Well, not directly... but if you open the core class we extend, you can see that *it* implements `VoterInterface`. The point is: as soon as we create a class

that implements `VoterInterface`, each time that the authorization system is called, Symfony will now call our `supports()` method and basically ask:

“Hey! Do you understand how to vote on this `$attribute` and this `$subject`?”

For us, I'm going to say if `in_array($attribute, ['EDIT'])`. So basically, *if* the attribute that is passed is equal to `EDIT`:

```
src/Security/Voter/QuestionVoter.php
↕ // ... lines 1 - 10
11 class QuestionVoter extends Voter
12 {
13     protected function supports(string $attribute, $subject): bool
14     {
15         // https://symfony.com/doc/current/security/voters.html
16         return in_array($attribute, ['EDIT'])
↕ // ... line 17
18     }
↕ // ... lines 19 - 40
41 }
```

I'm just using an array in case we support other attributes in this voter later - like `DELETE`.

Anyways, if the `$attribute` is `EDIT` and the `$subject` is an instance of `Question`, then yes, we know how to vote on this:

```
src/Security/Voter/QuestionVoter.php
↕ // ... lines 1 - 10
11 class QuestionVoter extends Voter
12 {
13     protected function supports(string $attribute, $subject): bool
14     {
15         // https://symfony.com/doc/current/security/voters.html
16         return in_array($attribute, ['EDIT'])
17             && $subject instanceof \App\Entity\Question;
18     }
↕ // ... lines 19 - 40
41 }
```

If we return `false`, it means that our voter will "abstain" from voting. But if we return `true`, then Symfony calls `voteOnAttribute()`:

```
src/Security/Voter/QuestionVoter.php
```

```
↕ // ... lines 1 - 10
11 class QuestionVoter extends Voter
12 {
↕ // ... lines 13 - 19
20     protected function voteOnAttribute(string $attribute, $subject,
    TokenInterface $token): bool
21     {
↕ // ... lines 22 - 39
40     }
41 }
```

Very simply, we need to take the attribute - in our case `EDIT` - and the `$subject` - in our case a `Question` object - and determine whether or not the user should have access by returning `true` or `false`.

I'm going to start by adding a few things that will help my editor. First, to get the current `User` object, you use this `$token` and call `$token->getUser()`:

```
src/Security/Voter/QuestionVoter.php
```

```
↕ // ... lines 1 - 10
11 class QuestionVoter extends Voter
12 {
↕ // ... lines 13 - 19
20     protected function voteOnAttribute(string $attribute, $subject,
    TokenInterface $token): bool
21     {
↕ // ... line 22
23         $user = $token->getUser();
↕ // ... lines 24 - 39
40     }
41 }
```

The only problem is that my editor doesn't know that this is an instance of *my* specific `User` class: it only knows that it's some `UserInterface`. To help, I'll add `@var User $user` above this:

src/Security/Voter/QuestionVoter.php

```
↕ // ... lines 1 - 5
6 use App\Entity\User;
↕ // ... lines 7 - 10
11 class QuestionVoter extends Voter
12 {
↕ // ... lines 13 - 19
20     protected function voteOnAttribute(string $attribute, $subject,
    TokenInterface $token): bool
21     {
22         /** @var User $user */
23         $user = $token->getUser();
↕ // ... lines 24 - 39
40     }
41 }
```

Even better, you could add an if statement to check if `$user` is *not* an instance of `User` and throw an exception:

src/Security/Voter/QuestionVoter.php

```
↕ // ... lines 1 - 8
9 use Symfony\Component\Security\Core\User\UserInterface;
10
11 class QuestionVoter extends Voter
12 {
↕ // ... lines 13 - 19
20     protected function voteOnAttribute(string $attribute, $subject,
    TokenInterface $token): bool
21     {
22         /** @var User $user */
23         $user = $token->getUser();
24         // if the user is anonymous, do not grant access
25         if (!$user instanceof UserInterface) {
26             return false;
27         }
↕ // ... lines 28 - 39
40     }
41 }
```

I'll actually do that down here. We know that `$subject` will be an instance of our `Question` class. To help our editor know that, say if not `$subject` is an `instanceof Question`, then throw a new `Exception` and just say "wrong type somehow passed":

src/Security/Voter/QuestionVoter.php

```
↕ // ... lines 1 - 4
5 use App\Entity\Question;
↕ // ... lines 6 - 10
11 class QuestionVoter extends Voter
12 {
↕ // ... lines 13 - 19
20     protected function voteOnAttribute(string $attribute, $subject,
    TokenInterface $token): bool
21     {
22         /** @var User $user */
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         if (!$subject instanceof Question) {
30             throw new \Exception('Wrong type somehow passed');
31         }
↕ // ... lines 32 - 39
40     }
41 }
```

That should never happen, but we're coding defensively. And more importantly, my editor - or static analysis tools like PHPStan - will now know what type the `$subject` variable is.

Finally, down here, the generated code has a switch-case to handle multiple attributes. I'll remove the second case... and make the first case `EDIT`. And... I don't even need the `break` because I'm going to return true if `$user` is equal to `$subject->getOwner()`:

src/Security/Voter/QuestionVoter.php

```
↕ // ... lines 1 - 10
11 class QuestionVoter extends Voter
12 {
↕ // ... lines 13 - 19
20     protected function voteOnAttribute(string $attribute, $subject,
    TokenInterface $token): bool
21     {
22         /** @var User $user */
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         if (!$subject instanceof Question) {
30             throw new \Exception('Wrong type somehow passed');
31         }
32
33         // ... (check conditions and return true to grant permission) ...
34         switch ($attribute) {
35             case 'EDIT':
36                 return $user === $subject->getOwner();
37             }
38
39         return false;
40     }
41 }
```

Let's try it! Back at the browser, I'm not logged in. So if we go back... to a question page... and click "edit"... access is still denied. Log in with our normal user. And then... access is still denied... which makes sense. We're an admin user... but we are *not* the owner of this question.

So let's log in as the owner! Go back to the homepage and click into a question. To make it more obvious *which* user owns this, temporarily, open

`templates/question/show.html.twig` and... down here, after the display name, just to help debug, print `question.owner.email`:

templates/question/show.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h2 class="my-4">Question:</h2>
10             <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11                 <div class="q-container-show p-4">
12                     <div class="row">
↕ // ... lines 13 - 33
34                         <div class="col">
↕ // ... lines 35 - 41
42                             <div class="q-display p-3">
↕ // ... lines 43 - 44
45                                 <p class="pt-4"><strong>--{{
question.owner.displayName }} ({{ question.owner.email }})</strong></p>
46                                     </div>
47                                 </div>
48                             </div>
49                         </div>
50                     </div>
51                 </div>
52             </div>
↕ // ... lines 53 - 65
66 </div>
67 {% endblock %}
```

And... cool. Copy the email and let's use impersonation! At the end of the URL, add `?_switch_user=`, paste that email and... boom! Access is granted thanks to our voter! We can prove it. Jump into the profiler and scroll down. Here it is: access granted for `EDIT` of this `Question` object. I love that.

Using the Voter in Twig

Now that we have this cool voter system, we can intelligently hide and show the edit button. Back in `show.html.twig`, wrap the anchor tag with `if is_granted()` passing the string `EDIT` and the question object:

templates/question/show.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h2 class="my-4">Question:</h2>
10            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11                <div class="q-container-show p-4">
12                    <div class="row">
↕ // ... lines 13 - 33
34                        <div class="col">
35                            <div class="d-flex justify-content-between">
↕ // ... lines 36 - 37
38                                {% if is_granted('EDIT', question) %}
39                                    <a href="{{ path('app_question_edit',
40
41                                        slug: question.slug
42                                        )) }}" class="btn btn-secondary btn-sm
43                                mb-2">Edit</a>
44                                {% endif %}
45                            </div>
↕ // ... lines 44 - 48
49                        </div>
50                    </div>
51                </div>
52            </div>
53        </div>
54    </div>
↕ // ... lines 55 - 67
68 </div>
69 {% endblock %}
```

How cool is that? We *should* still have access, and... when we refresh, it's still there. But if I exit impersonation... and then click back to the question, it's gone!

Also Allowing Admin Users to Edit

But I have one more challenge. Could we make it so that you can edit a question if you are the owner *or* if you have `ROLE_ADMIN`. Sure! To do that, in the voter, we just need to check for that role. To do that, we need a new service.

Add a constructor and autowire the `Security` service from the Symfony component. I'll hit `Alt+Enter` and go to "Initialize properties" to set things up:

```

src/Security/Voter/QuestionVoter.php
↕ // ... lines 1 - 8
9 use Symfony\Component\Security\Core\Security;
↕ // ... lines 10 - 11
12 class QuestionVoter extends Voter
13 {
14     private Security $security;
15
16     public function __construct(Security $security)
17     {
18         $this->security = $security;
19     }
↕ // ... lines 20 - 52
53 }

```

We talked about this service earlier: we used it to get the currently-authenticated User object from inside a service. It can *also* be used to check security from within a service.

Even before the switch case, let's add: if `$this->security->isGranted('ROLE_ADMIN')` then always return `true`:

```

src/Security/Voter/QuestionVoter.php
↕ // ... lines 1 - 8
9 use Symfony\Component\Security\Core\Security;
↕ // ... lines 10 - 11
12 class QuestionVoter extends Voter
13 {
↕ // ... lines 14 - 27
28     protected function voteOnAttribute(string $attribute, $subject,
    TokenInterface $token): bool
29     {
↕ // ... lines 30 - 40
41         if ($this->security->isGranted('ROLE_ADMIN')) {
42             return true;
43         }
44
45         // ... (check conditions and return true to grant permission) ...
46         switch ($attribute) {
↕ // ... lines 47 - 48
49         }
↕ // ... lines 50 - 51
52     }
53 }

```

So admin users can do anything. Oh, but whooops, I didn't mean to add that exclamation point!

Since we *are* currently logged in as an admin user.... as soon as we refresh, we have the edit button... and it works. So cool!

Next: Let's add an email confirmation system to our registration form.

Chapter 36: Verify Email after Registration

On some sites, after registration, you need to verify your email. You're almost definitely familiar with the process: you register, they send a special link to your email, you click that link and voilà! Your email is verified. If you *don't* click that link, depending on the site, you might not have access to certain sections... or you may not be able to log in at all. *That's* what we're going to do.

When we originally ran the `make:registration-form` command, it asked us if we wanted to generate an email verification process. If we had said yes, it would have generated some code for us. We said no... so that we could build it by hand, learn a bit more about how it works *and* customize things a bit.

User.isVerified Property

But before we jump into sending the verification email, inside our `User` class, we need some way to track whether or not a user has verified their email. Let's add a new field for that. Run:



```
symfony console make:entity
```

Update `User`, add an `isVerified` property, boolean type, not nullable and... perfect! Head over to the class. Let's see... here we go: `$isVerified`:


```

src/Entity/User.php
↕ // ... lines 1 - 17
18 class User implements UserInterface, PasswordAuthenticatedUserInterface
19 {
↕ // ... lines 20 - 56
57 /**
58  * @ORM\Column(type="boolean")
59  */
60 private $isVerified;
↕ // ... lines 61 - 225
226 public function getIsVerified(): ?bool
227 {
228     return $this->isVerified;
229 }
230
231 public function setIsVerified(bool $isVerified): self
232 {
233     $this->isVerified = $isVerified;
234
235     return $this;
236 }
237 }

```

Let's default this to `false`:

```

src/Entity/User.php
↕ // ... lines 1 - 17
18 class User implements UserInterface, PasswordAuthenticatedUserInterface
19 {
↕ // ... lines 20 - 59
60 private $isVerified = false;
↕ // ... lines 61 - 236
237 }

```

Ok, time for the migration:

```

symfony console make:migration

```

Go check that out and... awesome. It looks exactly like we expect:

migrations/Version20211012235912.php

```
↕ // ... lines 1 - 4
5 namespace DoctrineMigrations;
6
7 use Doctrine\DBAL\Schema\Schema;
8 use Doctrine\Migrations\AbstractMigration;
9
10 /**
11  * Auto-generated Migration: Please modify to your needs!
12  */
13 final class Version20211012235912 extends AbstractMigration
14 {
15     public function getDescription(): string
16     {
17         return '';
18     }
19
20     public function up(Schema $schema): void
21     {
22         // this up() migration is auto-generated, please modify it to your
needs
23         $this->addSql('ALTER TABLE user ADD is_verified TINYINT(1) NOT
NULL');
24     }
25
26     public function down(Schema $schema): void
27     {
28         // this down() migration is auto-generated, please modify it to
your needs
29         $this->addSql('ALTER TABLE user DROP is_verified');
30     }
31 }
```

Run it!

```
symfony console doctrine:migrations:migrate
```

Beautiful! Let's do one more thing related to the database. Inside of `src/Factory/UserFactory.php`, to make life simpler, set `$isVerified` to `true`:

src/Factory/UserFactory.php

```
↕ // ... lines 1 - 29
30 final class UserFactory extends ModelFactory
31 {
↕ // ... lines 32 - 40
41     protected function getDefaults(): array
42     {
43         return [
↕ // ... lines 44 - 46
47             'isVerified' => true,
48         ];
49     }
↕ // ... lines 50 - 68
69 }
```

So, by default, users in our fixtures will be verified. But I won't worry about reloading my fixtures quite yet.

Hello VerifyEmailBundle!

Okay: *now* let's add the email confirmation system! How? By leveraging a bundle! At your terminal, run:

```
composer require "symfonycasts/verify-email-bundle:1.11.0"
```

Hey, I know them! This bundle gives us a couple of services that will help generate a signed URL that we will include in the email and then *validate* that signed URL when the user clicks it. To get this working, open up `RegistrationController`. We already have our working `register()` method. Now we need one other method. Add public function `verifyUserEmail()`. Above this, give it a route: `@Route("/verify")` with `name="app_verify_email"`:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 14
15 class RegistrationController extends AbstractController
16 {
↕ // ... lines 17 - 53
54     /**
55      * @Route("/verify", name="app_verify_email")
56      */
57     public function verifyUserEmail(): Response
58     {
59         // TODO
60     }
61 }
```

When the user clicks the "confirm email" link in the email that we send them, *this* is the route and controller that link will take them to. I'm going to leave it empty for now. But eventually, its job will be to validate the signed URL, which will prove that the user *did* click on the exact link that we sent them.

Sending the Confirmation Email

Up in the `register()` action, *here* is where we need to send that email. As I mentioned earlier, you can do different things on your site based on whether or not the user's email is verified. In our site, we are going to completely *prevent* the user from logging in until their email is verified. So I'm going to remove the `$userAuthenticator` stuff:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 14
15 class RegistrationController extends AbstractController
16 {
↕ // ... lines 17 - 19
20     public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, UserAuthenticatorInterface $userAuthenticator,
    FormLoginAuthenticator $formLoginAuthenticator): Response
21     {
↕ // ... lines 22 - 25
26         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 27 - 39
40             $userAuthenticator->authenticateUser(
41                 $user,
42                 $formLoginAuthenticator,
43                 $request
44             );
45
46             return $this->redirectToRoute('app_homepage');
47         }
↕ // ... lines 48 - 51
52     }
↕ // ... lines 53 - 60
61 }
```

And replace that with the original redirect to `app_homepage`:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 13
14 class RegistrationController extends AbstractController
15 {
↕ // ... lines 16 - 18
19     public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, VerifyEmailHelperInterface $verifyEmailHelper):
    Response
20     {
↕ // ... lines 21 - 24
25         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 26 - 47
48             return $this->redirectToRoute('app_homepage');
49         }
↕ // ... lines 50 - 53
54     }
↕ // ... lines 55 - 62
63 }
```

Up top, we can remove some arguments.

Cool. Now we need to generate the signed email confirmation link and send it to the user. To do that, autowire a new service type-hinted with `VerifyEmailHelperInterface`. Call it `$verifyEmailHelper`:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 11
12 use SymfonyCasts\Bundle\VerifyEmail\VerifyEmailHelperInterface;
13
14 class RegistrationController extends AbstractController
15 {
↕ // ... lines 16 - 18
19     public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, VerifyEmailHelperInterface $verifyEmailHelper):
    Response
20     {
↕ // ... lines 21 - 53
54     }
↕ // ... lines 55 - 62
63 }
```

Below, after we save the user, let's generate that signed URL. This... looks a little weird at first. Say `$signatureComponents` equals `$verifyEmailHelper->generateSignature()`:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 13
14 class RegistrationController extends AbstractController
15 {
↕ // ... lines 16 - 18
19     public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, VerifyEmailHelperInterface $verifyEmailHelper):
    Response
20     {
↕ // ... lines 21 - 24
25         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 26 - 35
36             $entityManager->flush();
37
38             $signatureComponents = $verifyEmailHelper->generateSignature(
↕ // ... lines 39 - 42
43             );
↕ // ... lines 44 - 48
49         }
↕ // ... lines 50 - 53
54     }
↕ // ... lines 55 - 62
63 }
```

The first argument is the route name to the verification route. For us, that will be

`app_verify_email`:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 13
14 class RegistrationController extends AbstractController
15 {
↕ // ... lines 16 - 55
56     /**
57      * @Route("/verify", name="app_verify_email")
58      */
59     public function verifyUserEmail(): Response
60     {
↕ // ... line 61
62     }
63 }
```

So I'll put that here. Then the user's id - `$user->getId()` - and the user's email, `$user->getEmail()`:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 13
14 class RegistrationController extends AbstractController
15 {
↕ // ... lines 16 - 18
19     public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, VerifyEmailHelperInterface $verifyEmailHelper):
    Response
20     {
↕ // ... lines 21 - 24
25         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 26 - 37
38             $signatureComponents = $verifyEmailHelper->generateSignature(
39                 'app_verify_email',
40                 $user->getId(),
41                 $user->getEmail(),
↕ // ... line 42
43             );
↕ // ... lines 44 - 48
49         }
↕ // ... lines 50 - 53
54     }
↕ // ... lines 55 - 62
63 }
```

These are both used to "sign" the URL, which will help prove that *this* user *did* click the link from the email we sent them:

Verifying the Email without Being Logged In

But now we have a decision point. There are two different ways to use the `VerifyEmailBundle`. The first one is where, when the user clicks this email confirmation link, you expect them to be logged in. In this situation, down in `verifyUserEmail()`, we can use `$this->getUser()` to figure out *who* is trying to verify their email and use that to help validate the signed URL.

The other way, which is the way that *we're* going to use, is to allow the user to *not* be logged in when they click the confirmation link in their email. With this mode, we need to pass an array as the final argument to include the user id:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 13
14 class RegistrationController extends AbstractController
15 {
↕ // ... lines 16 - 18
19     public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, VerifyEmailHelperInterface $verifyEmailHelper):
    Response
20     {
↕ // ... lines 21 - 24
25         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 26 - 37
38             $signatureComponents = $verifyEmailHelper->generateSignature(
39                 'app_verify_email',
40                 $user->getId(),
41                 $user->getEmail(),
42                 ['id' => $user->getId()]
43             );
↕ // ... lines 44 - 48
49         }
↕ // ... lines 50 - 53
54     }
↕ // ... lines 55 - 62
63 }
```

The *whole* point of this `generateSignature()` method is to generate a signed URL. And thanks to this last argument, that URL will now contain an `id` query parameter... which we can use down in the `verifyUserEmail()` method to query for the `User`. We'll see that soon.

At this point, in a real app, you would take this `$signatureComponents` thing, pass it into an email template, use it to render the link and then send the email. But this is not a tutorial about sending emails - though we *do* have that [tutorial](#). So I'm going to take a shortcut. Instead of sending an email, say `$this->addFlash('success')` with a little message that says, "Confirm your email at:" and then the signed URL. You can generate that by saying `$signatureComponents->getSignedUrl()`:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 13
14 class RegistrationController extends AbstractController
15 {
↕ // ... lines 16 - 18
19     public function register(Request $request, UserPasswordHasherInterface
    $userPasswordHasher, VerifyEmailHelperInterface $verifyEmailHelper):
    Response
20     {
↕ // ... lines 21 - 24
25         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 26 - 37
38             $signatureComponents = $verifyEmailHelper->generateSignature(
39                 'app_verify_email',
40                 $user->getId(),
41                 $user->getEmail(),
42                 ['id' => $user->getId()]
43             );
44
45             // TODO: in a real app, send this as an email!
46             $this->addFlash('success', 'Confirm your email at:
    ' . $signatureComponents->getSignedUrl());
↕ // ... lines 47 - 48
49         }
↕ // ... lines 50 - 53
54     }
↕ // ... lines 55 - 62
63 }
```

We haven't talked about flash messages. They're basically temporary messages that you can put into the session... then render them one time. I put this message into a `success` category. Thanks to this, over in `templates/base.html.twig`, right after the navigation - so it's on top of the page - we can render any `success` flash messages. Add for `flash` in `app.flashes()` and then look up that key `success`. Inside, add `div` with `alert`, `alert-success` and render the message:

templates/base.html.twig

```
1  <!DOCTYPE html>
2  <html>
↕  // ... lines 3 - 14
15 <body
↕  // ... lines 16 - 81
82     {% for flash in app.flashes('success') %}
83         <div class="alert alert-success">{{ flash }}</div>
84     {% endfor %}
85
86     {% block body %}{% endblock %}
↕  // ... lines 87 - 89
90 </body>
91 </html>
```

This flash stuff has nothing to do with email confirmation... it's just a feature of Symfony that's most commonly used when you're handling forms. But it's a nice shortcut to help us test this.

Next: let's... do that! Test out our registration form and see what this signed URL looks like. Then we'll fill in the logic to verify that URL and confirm our user.

Chapter 37: Verifying the Signed Confirm Email URL

We're now generating a signed URL that we would *normally* include in a "confirm your email address" email that we send to the user after registration. To keep things simple, we're just rendering that URL onto the page after registration.

Removing our Unused Bind

Let's... go see what it looks like. Refresh and... ah! A terrible-looking error!

"A binding is configured for an argument named `$formLoginAuthenticator` under `_defaults`, but no corresponding argument has been found."

So until a few minutes ago, we had an argument to our `register()` action that was called `$formLoginAuthenticator`. Over in `config/services.yaml`, we set up a global "bind" that said:

"Whenever an autowired service has an argument named `$formLoginAuthenticator`, please pass this service."

```
config/services.yaml
↕ // ... lines 1 - 8
9  services:
10     # default configuration for services in *this* file
11     _defaults:
↕ // ... lines 12 - 13
14         bind:
↕ // ... line 15
16             $formLoginAuthenticator:
                '@security.authenticator.form_login.main'
↕ // ... lines 17 - 32
```

One of the cool things about bind is that if there is *no* matching argument *anywhere* in our app, it throws an exception. It's trying to make sure that we're not making an accidental typo.

In our situation, we... just don't need that argument anymore. So, delete it. And now... our registration page is alive!

Checking out the Verify URL

Let's do this! Enter an email, some password, agree to the terms and hit register. Beautiful! Here is our email confirmation URL. You can see that it goes to `/verify`: that will hit our new `verifyUserEmail()` action. It also includes an expiration. That's something you can configure... it's how long the link is valid for. And it has a `signature`: that's something that will help prove that the user didn't just make up this URL: it definitely came from us.

It also includes an `id=18`: our user id.

Verifying the Signed URL

So our job now is to go into the `verifyUserEmail` controller method down here and *validate* that signed URL. To do that, we need a few arguments: the `Request` object - so we can read data from the URL - a `VerifyEmailHelperInterface` to help us validate the URL - and finally, our `UserRepository` - so we can query for the `User` object:

```
src/Controller/RegistrationController.php
```

```
↕ // ... lines 1 - 6
7  use App\Repository\UserRepository;
↕ // ... line 8
9  use Symfony\Component\HttpFoundation\Request;
↕ // ... lines 10 - 13
14 use SymfonyCasts\Bundle\VerifyEmail\VerifyEmailHelperInterface;
15
16 class RegistrationController extends AbstractController
17 {
↕ // ... lines 18 - 60
61     public function verifyUserEmail(Request $request,
        VerifyEmailHelperInterface $verifyEmailHelper, UserRepository
        $userRepository): Response
62     {
↕ // ... lines 63 - 80
81     }
82 }
```

And actually, that's our first job. Say `$user = $userRepository->find()` and find the user that this confirmation link belongs to by reading the `id` query parameter. So, `$request->query->get('id')`. And if, for some reason, we can't find the `User`, let's trigger a 404 page by throwing `$this->createNotFoundException()`:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 15
16 class RegistrationController extends AbstractController
17 {
↕ // ... lines 18 - 60
61     public function verifyUserEmail(Request $request,
    VerifyEmailHelperInterface $verifyEmailHelper, UserRepository
    $userRepository): Response
62     {
63         $user = $userRepository->find($request->query->get('id'));
64         if (!$user) {
65             throw $this->createNotFoundException();
66         }
↕ // ... lines 67 - 80
81     }
82 }
```

Now we can make sure that the signed URL hasn't been tampered with. To do that, add a try-catch block. Inside, say `$verifyEmailHelper->validateEmailConfirmation()` and pass in a couple of things. First, the signed URL, which... is the current URL. Get that with `$request->getUri()`. Next pass the user's id - `$user->getId()` then the user's email - `$user->getEmail()`:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 15
16 class RegistrationController extends AbstractController
17 {
↕ // ... lines 18 - 60
61     public function verifyUserEmail(Request $request,
    VerifyEmailHelperInterface $verifyEmailHelper, UserRepository
    $userRepository): Response
62     {
63         $user = $userRepository->find($request->query->get('id'));
64         if (!$user) {
65             throw $this->createNotFoundException();
66         }
67
68         try {
69             $verifyEmailHelper->validateEmailConfirmation(
70                 $request->getUri(),
71                 $user->getId(),
72                 $user->getEmail(),
73             );
↕ // ... lines 74 - 77
78         }
↕ // ... lines 79 - 80
81     }
82 }
```

This makes sure that the id and email haven't changed in the *database* since the verification email was sent. Well, the id *definitely* hasn't changed... since we just used it to query. This part only really applies if you rely on the user being logged in to verify their email.

Anyways, if this is successful... nothing will happen! If it fails, it will throw a special exception that implements `VerifyEmailExceptionInterface`:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 15
16 class RegistrationController extends AbstractController
17 {
↕ // ... lines 18 - 60
61     public function verifyUserEmail(Request $request,
    VerifyEmailHelperInterface $verifyEmailHelper, UserRepository
    $userRepository): Response
62     {
↕ // ... lines 63 - 67
68         try {
69             $verifyEmailHelper->validateEmailConfirmation(
70                 $request->getUri(),
71                 $user->getId(),
72                 $user->getEmail(),
73             );
74         } catch (VerifyEmailExceptionInterface $e) {
↕ // ... lines 75 - 77
78         }
↕ // ... lines 79 - 80
81     }
82 }
```

So, down here, we know that verifying the URL failed... maybe someone messed with it. Or, more likely, the link expired. Let's tell the user the reason by leveraging the flash system again. Say `$this->addFlash()`, but this time put it into a different category called `error`. Then, to say what went wrong, use `$e->getReason()`. Finally, use `redirectToRoute()` to send them somewhere. How about the registration page?

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 15
16 class RegistrationController extends AbstractController
17 {
↕ // ... lines 18 - 60
61     public function verifyUserEmail(Request $request,
    VerifyEmailHelperInterface $verifyEmailHelper, UserRepository
    $userRepository): Response
62     {
↕ // ... lines 63 - 67
68         try {
69             $verifyEmailHelper->validateEmailConfirmation(
70                 $request->getUri(),
71                 $user->getId(),
72                 $user->getEmail(),
73             );
74         } catch (VerifyEmailExceptionInterface $e) {
75             $this->addFlash('error', $e->getReason());
76
77             return $this->redirectToRoute('app_register');
78         }
79
80         dd('TODO');
81     }
82 }
```

To render the error, back in `base.html.twig`, duplicate this entire block, but look for `error` messages and use `alert-danger`:

templates/base.html.twig

```
1 <!DOCTYPE html>
2 <html>
↕ // ... lines 3 - 14
15     <body>
↕ // ... lines 16 - 81
82         {% for flash in app.flashes('success') %}
83             <div class="alert alert-success">{{ flash }}</div>
84         {% endfor %}
85         {% for flash in app.flashes('error') %}
86             <div class="alert alert-danger">{{ flash }}</div>
87         {% endfor %}
↕ // ... lines 88 - 92
93     </body>
94 </html>
```


Phew! Let's try the error case. Copy the URL then open a new tab and paste. If I go to this *real* URL... it works. Well, we still need to do some more coding, but it hits our TODO at the bottom of the controller. Now mess with the URL, like remove a few characters... or tweak the expiration or change the `id`. Now... yes! It failed because our link is invalid. If the link were expired, you would see a message about that.

So, finally, let's finish the happy case! At the bottom of our controller, now that we know that the verification link *is* valid, we are done. For our app, we can say `$user->isVerified(true)` and then store that in the database:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 16
17 class RegistrationController extends AbstractController
18 {
↕ // ... lines 19 - 61
62     public function verifyUserEmail(Request $request,
    VerifyEmailHelperInterface $verifyEmailHelper, UserRepository
    $userRepository, EntityManagerInterface $entityManager): Response
63     {
↕ // ... lines 64 - 68
69         try {
↕ // ... lines 70 - 78
79         }
80
81         $user->setIsVerified(true);
↕ // ... lines 82 - 86
87     }
88 }
```

Let's see... we need one more argument: `EntityManagerInterface $entityManager`:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 7
8 use Doctrine\ORM\EntityManagerInterface;
↕ // ... lines 9 - 16
17 class RegistrationController extends AbstractController
18 {
↕ // ... lines 19 - 61
62     public function verifyUserEmail(Request $request,
    VerifyEmailHelperInterface $verifyEmailHelper, UserRepository
    $userRepository, EntityManagerInterface $entityManager): Response
63     {
↕ // ... lines 64 - 86
87     }
88 }
```

Back down here, use `$entityManager->flush()` to save that change:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 16
17 class RegistrationController extends AbstractController
18 {
↕ // ... lines 19 - 61
62     public function verifyUserEmail(Request $request,
    VerifyEmailHelperInterface $verifyEmailHelper, UserRepository
    $userRepository, EntityManagerInterface $entityManager): Response
63     {
↕ // ... lines 64 - 80
81         $user->setIsVerified(true);
82         $entityManager->flush();
↕ // ... lines 83 - 86
87     }
88 }
```

And let's give this a happy success message:

"Account verified! You can now log in."

Well, the truth is, we're not yet preventing them from logging in before they verify their email.

But we *will* soon. Anyways, finish by redirecting to the login page: `app_login`:

```
src/Controller/RegistrationController.php
↕ // ... lines 1 - 16
17 class RegistrationController extends AbstractController
18 {
↕ // ... lines 19 - 61
62     public function verifyUserEmail(Request $request,
    VerifyEmailHelperInterface $verifyEmailHelper, UserRepository
    $userRepository, EntityManagerInterface $entityManager): Response
63     {
↕ // ... lines 64 - 80
81         $user->setIsVerified(true);
82         $entityManager->flush();
83
84         $this->addFlash('success', 'Account Verified! You can now log
    in.');
```

```
85
86         return $this->redirectToRoute('app_login');
87     }
88 }
```

If you wanted to be *even* cooler, you could manually authenticate the user in the same way that we did earlier in our registration controller. That's totally ok and up to you.

Back in my main tab... copy that link again, paste and... we are verified! Sweet!

The only thing left to do is to prevent the user from logging in until they've verified their email. To do that, we first need to learn about the events that happen inside of the security system. And to show off *those*, we'll leverage a really cool new feature: login throttling.

Chapter 38: Login Throttling & Events

Symfony's security system comes packed with a lot of cool stuff, like remember me, impersonation and voters. Heck, it even has built in support for a "login link" authenticator - also known as "magic login links". That's where you email a link to your user and they click *that* to log in.

One other really cool feature is login throttling: a way to prevent someone from a single IP address from testing passwords over and over again on your site... by trying to log in over and over and over again. And it's super easy to use.

Activating login throttling

Under your firewall, enable it with `login_throttling: true`:

```
config/packages/security.yaml
1  security:
  ⬆ // ... lines 2 - 20
21  firewalls:
  ⬆ // ... lines 22 - 24
25      main:
  ⬆ // ... lines 26 - 28
29          login_throttling: true
  ⬆ // ... lines 30 - 62
```

If you stopped right there... and refreshed any page, you're going to get an error:

"Login throttling requires the Rate Limiter component."

And then a *helpful* command to install it! Nice! Copy that, spin over to your terminal and run:

```
composer require symfony/rate-limiter
```

💡 Tip

Starting in Symfony 6.2, when you install `symfony/rate-limiter`, the `symfony/lock` package is no longer needed (so you won't see a `lock.yaml` file, and that's ok!).

This package also installs a package called `symfony/lock`, which has a recipe. Run:

```
git status
```

to see what it did. Interesting. It created a new `config/packages/lock.yaml`, and also modified our `.env` file.

To keep track of the login attempts, the throttling system needs to store that data somewhere. It uses the `symfony/lock` component to do that. Inside of our `.env` file, at the bottom, there's a new `LOCK_DSN` environment variable which is set to `semaphore`:

`.env`



// ... lines 1 - 28

```
29  ###> symfony/lock ###
30  # Choose one of the stores below
31  # postgresql+advisory://db_user:db_password@localhost/db_name
32  LOCK_DSN=semaphore
33  ###
```

A semaphore... is basically a super easy way to store this data *if* you only have a single server. If you need something more advanced, check out the `symfony/lock` documentation: it shows all the different storage options with their pros and cons. But this will work *great* for us.

So, step 1 was to add the `login_throttling` config. Step 2 was to install the Rate Limiter component. And step 3 is... to enjoy the feature! Yea, we're done!

Refresh. No more error. By default, this will only allow 5 consecutive log in attempts for the same email and IP address per minute. Let's try it. One, two, three, four, five and... the sixth one is rejected! It locks us out for 1 minute. Both the max attempts and interval can be configured. Actually, we can see that.

At your terminal, run:

```
symfony console debug:config security
```

And... look for `login_throttling`. There it is. Yup, this `max_attempts` defaults to 5 and `interval` to 1 minute. Oh, and by the way, this will *also* block the same IP address from making 5 *times* the `max_attempts` for *any* email. In other words, if the same IP address quickly tried 25 *different* emails, it would still block them. And if you want an awesome first line of defense, I would also highly recommend using something like Cloudflare, which can block bad users even before they hit your server... or enable defenses if your site is attacked from many IP addresses.

Digging into How Login Throttling Works

So... I think this feature is pretty cool. But the most interesting thing for *us* about it is how it works behind-the-scenes. It works via Symfony's listener system. After we log in, whether successfully or unsuccessfully, a number of events are dispatched throughout that process. We can hook *into* those events to do all sorts of cool things.

For example, the class that holds the login throttling logic is called `LoginThrottlingListener`. Let's... open it up! Hit `Shift+Shift` and open `LoginThrottlingListener.php`.

Awesome. The details inside of this aren't too important. You can see it's using something called a rate limiter... which does the checking of if the limit has been hit. Ultimately, if the limit *has* been hit, it throws this exception, which causes the message that we saw. For those of you watching closely, that exception extends `AuthenticationException`... and remember, you can throw an `AuthenticationException` at *any* point in the authentication process to make it fail.

Anyways, this method is listening to an event called `CheckPassportEvent`. This is dispatched after the `authenticate()` method is called from any authenticator. At this point, authentication isn't successful yet... and the job of most listeners to `CheckPassportEvent` is to do some extra checking and fail authentication if something went wrong.

This class also listens to another event called `LoginSuccessEvent`... which... well, it's kind of obvious: this is dispatched after any successful authentication. This resets the rate limiter on

success.

So this is *really* cool, and it's our first vision into how the event system works. Next, let's go deeper by discovering that almost *every* part of authentication is done by a listener. Then, we'll create our *own*.

Chapter 39: Security Events & Listeners

If you've used Symfony for a while, you probably know that Symfony dispatches events during the request-response process and that you can listen to them. To see these events and their listeners, we can run:

```
symfony console debug:event
```

I'm not going to go too deeply, but, this `kernel.request` event is dispatched on every request *before* the controller is called. This means that all of these *listeners* are executed before our controller. Listeners to this `kernel.response` event are called *after* our controller.

These two events have... nothing to do with the security system. But it turns out that our firewall *also* dispatches several events during the authentication process. And, we can also listen to those.

To see a list of all of the listeners to these events, we can run `debug:event` again, but with a special `--dispatcher=` set to `security.event_dispatcher.main`:

```
symfony console debug:event --dispatcher=security.event_dispatcher.main
```

I know, that looks a little funny... but this allows us to list the event listeners for the event dispatcher that's specific to the `main` firewall.

Looking at the Core Security Events & Listeners

And... awesome! A totally different set of events and listeners. This is so cool. Look back at our custom `LoginFormAuthenticator` class. We're not using this anymore, but it can help us understand which events are dispatched through the process.

We know that, in our `authenticate()` method, our job is to return the `Passport`:

src/Security/LoginFormAuthenticator.php

```
↕ // ... lines 1 - 26
27 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
28 {
↕ // ... lines 29 - 39
40     public function authenticate(Request $request): PassportInterface
41     {
42         $email = $request->request->get('email');
43         $password = $request->request->get('password');
44
45         return new Passport(
46             new UserBadge($email, function($userIdentifier) {
47                 // optionally pass a callback to load the User manually
48                 $user = $this->userRepository->findOneBy(['email' =>
49                     $userIdentifier]);
50
51                 if (!$user) {
52                     throw new UserNotFoundException();
53                 }
54
55                 return $user;
56             }),
57             new PasswordCredentials($password),
58             [
59                 new CsrfTokenBadge(
60                     'authenticate',
61                     $request->request->get('_csrf_token')
62                 ),
63                 (new RememberMeBadge())->enable(),
64             ]
65         );
↕ // ... lines 66 - 81
82 }
```

Then, after the `authenticate()` method is called - on *any* authenticator - Symfony dispatches `CheckPassportEvent`. There are a *bunch* of cool listeners to this.

For example, `UserProviderListener` is basically responsible for loading the `User` object, `CheckCredentialsListener` is responsible for checking the password, `CsrfProtectionListener` validates the CSRF token and `LoginThrottlingListener` checks... the login throttling.

If we fail authentication, there's a *different* event for that: `LoginFailureEvent`. Right now, our app has just one listener - `RememberMeListener` - which clears the "remember me" cookie if

the user had one.

When login is successful, Symfony dispatches `LoginSuccessEvent`. This already has 5 listeners in our app, including the listener that sets the "remember me" cookie.

There's also an event that's dispatched when you log out... so you can run code or even control what happens - like where the user is redirected to.

This next one - `TokenDeauthenticatedEvent` - is a bit more subtle. It's dispatched if the user "loses" authentication... but didn't log out. It's basically dispatched if certain data *changes* on the user. For example, imagine you're logged in on two computers and then you change your password on the first. When you refresh the site on the second computer, you will be "deauthenticated" because your password changed on another machine. In that case, this event is dispatched.

Oh, and this `security.authentication.success` isn't too important, it's very similar to `LoginSuccessEvent`.

Knowing about these events is critical because I want to make it so that if the user tries to log in using an email that has *not* been verified, we prevent that and show them a nice message.

Let's do that next by bootstrapping our very own shiny event listener that has the ability to cause authentication to fail.

Chapter 40: Creating a Security Event Subscriber

Here's our goal: if a user tries to log in but they have *not* verified their email yet, we need to cause authentication to fail.

If you want to stop authentication for some reason, then you probably want to listen to the `CheckPassportEvent`: that's called right *after* the `authenticate()` method is executed on *any* authenticator and... its job is to do stuff like this.

Creating the Event Subscriber

In your `src/` directory, it doesn't matter where, but I'm going to create a new directory called `EventSubscriber/`. Inside, add a class called `CheckVerifiedUserSubscriber`. Make this implement `EventSubscriberInterface` and then go to the "Code"-> "Generate" menu - or `Command` + `N` on a Mac - and hit "Implement Methods" to generate the *one* we need: `getSubscribedEvents()`:

```
src/EventSubscriber/CheckVerifiedUserSubscriber.php
```

```
↕ // ... lines 1 - 2
3 namespace App\EventSubscriber;
4
5 use Symfony\Component\EventDispatcher\EventSubscriberInterface;
6
7 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
8 {
9     public static function getSubscribedEvents()
10    {
11    }
12 }
```

Inside, return an array of all the events that we want to listen to, which is just one. Say `CheckPassportEvent::class` set to the method on this class that should be called when that event is dispatched. How about, `onCheckPassport`:

src/EventSubscriber/CheckVerifiedUserSubscriber.php

```
↕ // ... lines 1 - 5
6 use Symfony\Component\Security\Http\Event\CheckPassportEvent;
7
8 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
9 {
↕ // ... lines 10 - 14
15     public static function getSubscribedEvents()
16     {
17         return [
18             CheckPassportEvent::class => 'onCheckPassport',
19         ];
20     }
21 }
```

Up above, add that: `public function onCheckPassport()`... and this will receive this event object. So `CheckPassportEvent $event`. Start with `dd($event)` so we can see what it looks like:

src/EventSubscriber/CheckVerifiedUserSubscriber.php

```
↕ // ... lines 1 - 5
6 use Symfony\Component\Security\Http\Event\CheckPassportEvent;
7
8 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
9 {
10     public function onCheckPassport(CheckPassportEvent $event)
11     {
12         dd($event);
13     }
↕ // ... lines 14 - 20
21 }
```

Now, *just* by creating this class and making it implement `EventSubscriberInterface`, thanks to Symfony's "autoconfigure" feature, it will *already* be called when the `CheckPassportEvent` happens. And... if you want to get technical, our subscriber listens to the `CheckPassportEvent` on *all* firewalls. For us, we only have one *real* firewall, so it doesn't matter:

config/packages/security.yaml

```
1 security:
↕ // ... lines 2 - 20
21     firewalls:
22         dev:
↕ // ... lines 23 - 24
25             main:
26                 lazy: true
27                 provider: app_user_provider
28                 entry_point: form_login
29                 login_throttling: true
30                 form_login:
31                     login_path: app_login
32                     check_path: app_login
33                     username_parameter: email
34                     password_parameter: password
35                     enable_csrf: true
36
37                 custom_authenticator:
38                     # - App\Security\LoginFormAuthenticator
39                     - App\Security\DummyAuthenticator
40
41                 logout: true
42
43                 remember_me:
44                     secret: '%kernel.secret%'
45                     signature_properties: [password]
46                     always_remember_me: true
47
48                 switch_user: true
↕ // ... lines 49 - 62
```

But if you *did* have multiple real firewalls, our subscriber would be called whenever the event is triggered for *any* firewall. If you need to, you can add a little extra config to target just *one* of the firewalls.

Tweaking the Event Priority

Anyways, let's try this thing!. Log in as `abraca_admin@example.com`. We *did* set the `isVerified` flag in the fixtures to true for all users... but we haven't reloaded the database yet. So this user will *not* be verified.

Try typing an *invalid* password and submitting. Yes! It hit our `dd()`. So this *is* working. But if I type an invalid *email*, our listener is *not* executed. Why?

Both the loading of the user *and* the checking of the password happen via listeners to the `CheckPassportEvent`: the same event we're listening to. The inconsistency in behavior - the fact that our listener was executed with an invalid password but *not* with an invalid email - is due to the *priority* of the listeners.

Go back to your terminal. Ah, each event shows a *priority*, and the default is zero. Let me make this a bit smaller so we can read it. There we go.

Look closely: our listener is called *before* the `CheckCredentialsListener`. That's why it called our listener *before* the password check could fail.

But, that's not what we want. We don't want to do our "is verified" check until we know the password is valid: no reason to expose whether the account is verified or not until we *know* the real user is logging in.

The point is: we want our code to run *after* `CheckCredentialsListener`. To do that, we can give *our* listener a *negative* priority. Tweak the syntax: set the event name to an array with the method name as the first key and the priority as the second. How about negative 10:

```
src/EventSubscriber/CheckVerifiedUserSubscriber.php
↕ // ... lines 1 - 7
8 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
9 {
↕ // ... lines 10 - 14
15     public static function getSubscribedEvents()
16     {
17         return [
18             CheckPassportEvent::class => ['onCheckPassport', -10],
19         ];
20     }
21 }
```

Thanks to this, the user will need to enter a valid email *and* a valid password before our listener is called. Try it: go back to `abraca_admin@example.com`, password `tada` and... beautiful!

Using the Event Object

Check out the event object that we're passed: it's *full* of good stuff. It contains the authenticator that was used, in case we need to do something different based on that. It also holds the `Passport` ... which is *huge* because that contains the `User` object *and* badges... because sometimes you need to do different things based on the badges on the passport.

Inside of our subscriber, let's get to work. To get the user, we first need to get the passport:

`$passport = $event->getPassport()`. Now, add if *not* `$passport` is an `instanceof UserPassportInterface`, throw an exception:

```
src/EventSubscriber/CheckVerifiedUserSubscriber.php
↕ // ... lines 1 - 6
7  use
   Symfony\Component\Security\Http\Authenticator\Passport\UserPassportInterface
↕ // ... lines 8 - 9
10 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
11 {
12     public function onCheckPassport(CheckPassportEvent $event)
13     {
14         $passport = $event->getPassport();
15         if (!$passport instanceof UserPassportInterface) {
16             throw new \Exception('Unexpected passport type');
17         }
18     }
↕ // ... lines 19 - 25
26 }
```

This check isn't important and is *not* needed in Symfony 6 and higher. Basically, this check makes sure that our `Passport` has a `getUser()` method, which in practice, it always will. In Symfony 6, the check isn't needed at all because the `Passport` class literally *always* has this method.

This means that, down here, we can say `$user = $passport->getUser()`. And then let's add a sanity check: if `$user` is not an instance of our `User` class, throw an exception: "Unexpected user type":

src/EventSubscriber/CheckVerifiedUserSubscriber.php

```
↕ // ... lines 1 - 4
5 use App\Entity\User;
↕ // ... lines 6 - 10
11 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
12 {
13     public function onCheckPassport(CheckPassportEvent $event)
14     {
15         $passport = $event->getPassport();
16         if (!$passport instanceof UserPassportInterface) {
17             throw new \Exception('Unexpected passport type');
18         }
19
20         $user = $passport->getUser();
21         if (!$user instanceof User) {
22             throw new \Exception('Unexpected user type');
23         }
24     }
↕ // ... lines 25 - 31
32 }
```

In practice, in our app, this isn't possible. But that's a nice way to hint to my editor - or static analysis tools - that `$user` is *our* User class. Thanks to this, when we say if *not* `$user->getIsVerified()`, it auto-completes that method:

src/EventSubscriber/CheckVerifiedUserSubscriber.php

```
↕ // ... lines 1 - 11
12 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
13 {
14     public function onCheckPassport(CheckPassportEvent $event)
15     {
16         $passport = $event->getPassport();
17         if (!$passport instanceof UserPassportInterface) {
18             throw new \Exception('Unexpected passport type');
19         }
20
21         $user = $passport->getUser();
22         if (!$user instanceof User) {
23             throw new \Exception('Unexpected user type');
24         }
25
26         if (!$user->getIsVerified()) {
↕ // ... line 27
28         }
29     }
↕ // ... lines 30 - 36
37 }
```

Failing Authentication

Ok, if we are *not* verified, we need to cause authentication to fail. How do we do that? It turns out that, at any time during the authentication process, we can throw an

AuthenticationException - from Security - and that will cause authentication to fail:

src/EventSubscriber/CheckVerifiedUserSubscriber.php

```
↕ // ... lines 1 - 7
8 use Symfony\Component\Security\Core\Exception\AuthenticationException;
↕ // ... lines 9 - 11
12 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
13 {
14     public function onCheckPassport(CheckPassportEvent $event)
15     {
↕ // ... lines 16 - 25
26         if (!$user->getIsVerified()) {
27             throw new AuthenticationException();
28         }
29     }
↕ // ... lines 30 - 36
37 }
```

And there are a bunch of subclasses to this class, like `BadCredentialsException`. You can throw any of these because they all extend `AuthenticationException`.

Check it out. Let's refresh and... got it!

"An authentication exception occurred."

That's the generic error message tied to the `AuthenticationException` class... not a very good error message. But it *did* get the job done.

How can we customize that? Either by throwing a different authentication exception that matches the message you want - like `BadCredentialsException` - or by taking *complete* control by throwing the special `CustomUserMessageAuthenticationException()`. Pass this the message to show the user:

"Please verify your account before logging in."

```
src/EventSubscriber/CheckVerifiedUserSubscriber.php
// ... lines 1 - 8
9 use
  Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationException;
// ... lines 10 - 12
13 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
14 {
15     public function onCheckPassport(CheckPassportEvent $event)
// ... lines 16 - 26
27         if (!$user->getIsVerified()) {
28             throw new CustomUserMessageAuthenticationException(
29                 'Please verify your account before logging in.'
30             );
31         }
32     }
// ... lines 33 - 39
40 }
```

Let's see how this works. Hold `Cmd` or `Ctrl` and click to open this class. No surprise: it extends `AuthenticationException`. If you try to pass a custom exception message to `AuthenticationException` or one of its sub-classes, that message will normally *not* be shown to the user.

This is because every authentication exception class has a `getMessageKey()` method containing a hardcoded message... and *that* is what is shown to the user. This is done for

security so that we don't accidentally expose some internal exception message to our users. This is why different authentication exception sub-classes give us different messages.

However, there *are* some cases when you want to show a *truly* custom message. You can do that by using this class. This will fail authentication just like before, but now *we* control the message. Beautiful.

But we can do even better! Instead of just saying, "please verify your account", let's redirect the user to another page where we can better explain why they can't log in *and* give them an opportunity to re-send the email. This will require a *second* listener and some serious team work. That's next.

Chapter 41: Custom Redirect when "Email Not Verified"

It's cool that we can listen to the `CheckPassportEvent` and cause authentication to fail by throwing any authentication exception, like this

`CustomUserMessageAuthenticationException`:

```
src/EventSubscriber/CheckVerifiedUserSubscriber.php
↕ // ... lines 1 - 8
9 use
  Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationExc
↕ // ... lines 10 - 12
13 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
14 {
15     public function onCheckPassport(CheckPassportEvent $event)
↕ // ... lines 16 - 26
27         if (!$user->getIsVerified()) {
28             throw new CustomUserMessageAuthenticationException(
29                 'Please verify your account before logging in.'
30             );
31         }
32     }
↕ // ... lines 33 - 39
40 }
```

But what if, instead of the normal failure behavior - where we redirect to the login page and show the error - we want to do something different. What if, in *just* this situation, we want to redirect to a totally *different* page so we can explain that their email isn't verified... and maybe even allow them to *resend* that email.

Well, unfortunately, there is no way - on this event - to control the failure response. There's no `$event->setResponse()` or anything like that.

So we can't control the error behavior from here, but we *can* control it by listening to a *different* event. We'll "signal" from *this* event that the account wasn't verified, *look* for that signal from a different event listener, and redirect to that other page. It's ok if this doesn't make sense yet: let's see it in action.

Creating a Custom Exception Class

To start, we need to create a custom authentication exception class. This will serve as the "signal" that we're in this "account not verified" situation.

In the `Security/` directory, add a new class: how about `AccountNotVerifiedAuthenticationException`. Make it extend `AuthenticationException`. And then... do absolutely nothing else:

```
src/Security/AccountNotVerifiedAuthenticationException.php
↕ // ... lines 1 - 2
3 namespace App\Security;
4
5 use Symfony\Component\Security\Core\Exception\AuthenticationException;
6
7 class AccountNotVerifiedAuthenticationException extends
  AuthenticationException
8 {
9 }
```

This is just a marker class we'll use to hint that we're failing authentication due to an unverified email.

Back in the subscriber, replace the `CustomUserMessageAuthenticationException` with `AccountNotVerifiedAuthenticationException`. We don't need to pass it *any* message:

```
src/EventSubscriber/CheckVerifiedUserSubscriber.php
↕ // ... lines 1 - 5
6 use App\Security\AccountNotVerifiedAuthenticationException;
↕ // ... lines 7 - 13
14 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
15 {
16     public function onCheckPassport(CheckPassportEvent $event)
17     {
↕ // ... lines 18 - 27
28         if (!$user->getIsVerified()) {
29             throw new AccountNotVerifiedAuthenticationException();
30         }
31     }
↕ // ... lines 32 - 38
39 }
```

If we stopped right now, this won't be very interesting. Logging in still fails, but we're back to the generic message:

"An authentication exception occurred"

This is because our new custom class extends `AuthenticationException`... and that's the generic message you get from *that* class. So this isn't what we want yet, but step 1 *is* done!

Listening to LoginFailureEvent

For the next step, remember from the `debug:event` command that one of the listeners we have is for a `LoginFailureEvent`, which, as the name suggests, is called any time that authentication *fails*.

Let's add another listener right in this class for that. Say `LoginFailureEvent::class` set to, how about, `onLoginFailure`. In this case, the priority won't matter:

```
src/EventSubscriber/CheckVerifiedUserSubscriber.php
```

```
↕ // ... lines 1 - 12
13 use Symfony\Component\Security\Http\Event\LoginFailureEvent;
14
15 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
16 {
↕ // ... lines 17 - 38
39     public static function getSubscribedEvents()
40     {
41         return [
↕ // ... line 42
43             LoginFailureEvent::class => 'onLoginFailure',
44         ];
45     }
46 }
```

Add the new method: `public function onLoginFailure()`... and we know this will receive a `LoginFailureEvent` argument. Just like before, start with `dd($event)` to see what it looks like:

```
src/EventSubscriber/CheckVerifiedUserSubscriber.php
```

```
↕ // ... lines 1 - 12
13 use Symfony\Component\Security\Http\Event\LoginFailureEvent;
14
15 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
16 {
↕ // ... lines 17 - 33
34     public function onLoginFailure(LoginFailureEvent $event)
35     {
36         dd($event);
37     }
↕ // ... lines 38 - 45
46 }
```

So with any luck, if we fail login - for any reason - our listener will be called. For example, if I enter a bad password, yup! It gets hit. And notice that the `LoginFailureEvent` has an exception property. In this case, it holds a `BadCredentialsException`.

Now log in with the correct password and... it got hit again. But *this* time, check out the exception. It's our custom `AccountNotVerifiedAuthenticationException`! So the `LoginFailureEvent` object contains the authentication exception that *caused* the failure. We can use that to know - from this method - if authentication failed due to the account not being verified.

Redirecting when Account is Not Verified

So, if *not* `$event->getException()` is an instance of `AccountNotVerifiedAuthenticationException`, then just return and allow the default failure behavior to do its thing:

src/EventSubscriber/CheckVerifiedUserSubscriber.php

```
↕ // ... lines 1 - 14
15 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
16 {
↕ // ... lines 17 - 33
34     public function onLoginFailure(LoginFailureEvent $event)
35     {
36         if (!$event->getException() instanceof
AccountNotVerifiedAuthenticationException) {
37             return;
38         }
39     }
↕ // ... lines 40 - 47
48 }
```

Finally, down here, we know that we should redirect to that custom page. Let's... go create that page real quick. Do it in `src/Controller/RegistrationController.php`. Down at the bottom, add a new method. I'll call it `resendVerifyEmail()`. Above this, add `@Route()` with, how about `/verify/resend` and name equals `app_verify_resend_email`. Inside, I'm just going to render a template: `return $this->render()`, `registration/resend_verify_email.html.twig`:

src/Controller/RegistrationController.php

```
↕ // ... lines 1 - 16
17 class RegistrationController extends AbstractController
18 {
↕ // ... lines 19 - 88
89     /**
90      * @Route("/verify/resend", name="app_verify_resend_email")
91      */
92     public function resendVerifyEmail()
93     {
94         return $this-
>render('registration/resend_verify_email.html.twig');
95     }
96 }
```

Let's go make that! Inside of `templates/registration/`, create `resend_verify_email.html.twig`. I'll paste in the template:


```
templates/registration/resend_verify_email.html.twig
```

```
1 {% extends 'base.html.twig' %}
2
3 {% block title %}Verify Email{% endblock %}
4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <h1 class="h3 mb-3 font-weight-normal">Verify your Email</h1>
10
11             <p>
12                 A verification email was sent - please click it to enable
13                 your
14                 account before logging in.
15             </p>
16             <a href="#" class="btn btn-primary">Re-send Email</a>
17         </div>
18     </div>
19 </div>
20 {% endblock %}
```

There's nothing fancy here at all. It just explains the situation.

I *did* include a button to resend the email, but I'll leave the implementation to you. I'd probably surround it with a form that POSTs to this URL. And then, in the controller, if the method is POST, I'd use the verify email bundle to generate a new link and re-send it. Basically the same code we used after registration.

Anyways, now that we have a functional page, copy the route name and head back to our subscriber. To override the normal failure behavior, we can use a `setResponse()` method on the event.

Start with `$response = new RedirectResponse()` - we're going to generate a URL to the route in a minute - then `$event->setResponse($response)`:

```

src/EventSubscriber/CheckVerifiedUserSubscriber.php
↕ // ... lines 1 - 16
17 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
18 {
↕ // ... lines 19 - 42
43     public function onLoginFailure(LoginFailureEvent $event)
44     {
45         if (!$event->getException() instanceof
AccountNotVerifiedAuthenticationException) {
46             return;
47         }
48
49         $response = new RedirectResponse(
↕ // ... line 50
51         );
52         $event->setResponse($response);
53     }
↕ // ... lines 54 - 61
62 }

```

To generate the URL, we need a `__construct()` method - let me spell that correctly - with a `RouterInterface $router` argument. Hit `Alt+Enter` and go to "Initialize properties" to create that property and set it:

```

src/EventSubscriber/CheckVerifiedUserSubscriber.php
↕ // ... lines 1 - 8
9 use Symfony\Component\Routing\RouterInterface;
↕ // ... lines 10 - 16
17 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
18 {
19     private RouterInterface $router;
20
21     public function __construct(RouterInterface $router)
22     {
23         $this->router = $router;
24     }
↕ // ... lines 25 - 61
62 }

```

Back down here, we're in business: `$this->router->generate()` with `app_verify_resend_email`:

src/EventSubscriber/CheckVerifiedUserSubscriber.php

```
↕ // ... lines 1 - 16
17 class CheckVerifiedUserSubscriber implements EventSubscriberInterface
18 {
↕ // ... lines 19 - 42
43     public function onLoginFailure(LoginFailureEvent $event)
44     {
↕ // ... lines 45 - 48
49         $response = new RedirectResponse(
50             $this->router->generate('app_verify_resend_email')
51         );
↕ // ... line 52
53     }
↕ // ... lines 54 - 61
62 }
```

Donezo! We fail authentication, our first listener throws the custom exception, we look for that exception from the `LoginFailureEvent` listener... and set the redirect.

Testing time! Refresh and... got it! We're sent over to `/verify/resend`. I love that!

Next: let's finish this tutorial by doing something *super* cool, super fun, and... kinda nerdy. Let's add two-factor authentication, complete with fancy QR codes.

Chapter 42: 2 Factor Authentication & Authentication Tokens

For our last trick in this tutorial, we're going to do something fun: add two-factor authentication. This can take a few forms, but the basic flow looks like this, you're probably familiar. First, the user submits a valid email and password to the login form. But then, instead of that logging them in, they're redirected to a form where they need to enter a temporary code.

This code could be something that we email them or text to their phone... or it could be a code from an authenticator app like Google authenticator or Authy. Once the user fills in the code and submits, *then* they are finally logged in.

Installing The [scheb/2fa-bundle](#)

In the Symfony world, we're *super* lucky to have a *fantastic* library to help with two-factor auth. Search for Symfony 2fa to find the [scheb/2fa](#) library. Scroll down... and click into the documentation, which lives on Symfony.com. Then head to the Installation page.

Cool! Let's get this thing installed! At your terminal, run:



```
composer require "2fa:^5.13"
```

Where 2fa is a Flex alias to the actual bundle name.

Once this finishes... I'll run:



```
git status
```

to see what the bundle's recipe did. Cool: it added a new configuration file... and also a new routes file.

That routes file, which lives at `config/routes/scheb_2fa.yaml`, adds two routes to our app:

```
config/routes/scheb_2fa.yaml
1 2fa_login:
2   path: /2fa
3   defaults:
4     _controller: "scheb_two_factor.form_controller:form"
5
6 2fa_login_check:
7   path: /2fa_check
```

The first will render the "enter the code" form that we see after submitting our email and password. The second route is the URL that this form will submit to.

Bundle Configuration / Setup

Back at the docs, let's walk through this. Step 2 - enable the bundle - was done by Flex automatically... and step 3 - define the routes - was handled thanks to the recipe. Nice!

Step 4 is to configure the firewall. This part we *do* need to do.

Start by copying the `two_factor` stuff. Then open up `config/packages/security.yaml`. This new config can live anywhere under our `main` firewall. I'll paste it after `form_login`... and we can remove this comment: it highlighted that `2fa_login` should match the route name in our routes file, which it does:

```
config/packages/security.yaml
1 security:
2 // ... lines 2 - 20
21 firewalls:
22 // ... lines 22 - 24
25 main:
26 // ... lines 26 - 49
50     two_factor:
51         auth_form_path: 2fa_login
52         check_path: 2fa_login_check
53 // ... lines 53 - 71
```

Oh, and remember how the job of *most* keys under our firewall is to activate another *authenticator*? Whelp, the `two_factor` key is no exception: this activates a new authenticator

that handles the "enter your code" form submit that we'll see in a few minutes.

The README also recommends a couple of access controls, which are a good idea. Copy those... and paste them at the top of our `access_control`:

```
config/packages/security.yaml
1  security:
  ⬆ // ... lines 2 - 61
62  access_control:
63      # This makes the logout route accessible during two-factor
    authentication. Allows the user to
64      # cancel two-factor authentication, if they need to.
65      - { path: ^/logout, role: PUBLIC_ACCESS }
66      # This ensures that the form can only be accessed when two-factor
    authentication is in progress.
67      - { path: ^/2fa, role: IS_AUTHENTICATED_2FA_IN_PROGRESS }
68      - { path: ^/admin/login, roles: PUBLIC_ACCESS }
69      - { path: ^/admin, roles: ROLE_ADMIN }
70      # - { path: ^/profile, roles: ROLE_USER }
```

This second one makes sure that you can't go to `/2fa` - that's the URL that renders the "enter your code" form - unless you *have* already submitted your valid email and password. When you're in that, sort of, "in-between-login" state, the 2fa bundle makes sure that you have this `IS_AUTHENTICATED_2FA_IN_PROGRESS` attribute:

```
config/packages/security.yaml
1  security:
  ⬆ // ... lines 2 - 61
62  access_control:
  ⬆ // ... lines 63 - 65
66      # This ensures that the form can only be accessed when two-factor
    authentication is in progress.
67      - { path: ^/2fa, role: IS_AUTHENTICATED_2FA_IN_PROGRESS }
  ⬆ // ... lines 68 - 71
```

The first entry - for `/logout` - makes sure that if you *are* in that "in-between" state, you *can* still cancel the login by going to `/logout`. Oh, but change this to `PUBLIC_ACCESS`:

```
config/packages/security.yaml
```

```
1 security:
```

```
↕ // ... lines 2 - 61
```

```
62     access_control:
```

```
63         # This makes the logout route accessible during two-factor
authentication. Allows the user to
```

```
64         # cancel two-factor authentication, if they need to.
```

```
65         - { path: ^/logout, role: PUBLIC_ACCESS }
```

```
↕ // ... lines 66 - 71
```

Configuring the security_tokens

The *last* step in the README is to configure this `security_tokens` config.

Let me explain. When we submit a valid email and password into the login form, the two-factor authentication system - via a listener - is going to decide whether or not it should *interrupt* authentication and start the two-factor authentication process... where it redirects the user to the "enter the code" form.

If we think about it, we definitely *do* want this to happen when a user logs in via the login form. But... we probably *wouldn't* want this to happen if, for example, a user was authenticating via an API token. The bundle needs a way to figure out whether or not we want 2fa based on *how* the user just authenticated.

We haven't talked about it much, but whenever you log in, you're authenticated with a certain type of *token* object. This token object is... sort of a wrapper around the `User` object... and you almost *never* care about it.

But, different authentication systems - like `form_login` or `remember_me` - use different token classes... which means that you can figure out *how* the user originally logged in, by looking at the currently-authenticated token.

For example, this top token class is actually the token that you get if you log in via the `form_login` authenticator. I'll prove it. Hit `Shift+Shift` and search for `FormLoginAuthenticator`. Inside... it has a `createAuthenticatedToken()` method, a method that every authenticator has. It returns a new `UsernamePasswordToken`.

Here's the point. If we login via this authenticator... and the matching token class is listed under our `scheb_two_factor` config, the two-factor authentication process will take over and redirect the user to the "enter the code" form.

Let's go see what our file looks like: `config/packages/scheb_2fa.yaml`:

`config/packages/scheb_2fa.yaml`

```
1 # See the configuration reference at
  https://github.com/scheb/2fa/blob/master/doc/configuration.md
2 scheb_two_factor:
3     security_tokens:
4         -
5             Symfony\Component\Security\Core\Authentication\Token\UsernamePasswordToken
6             # If you're using guard-based authentication, you have to use this
7             one:
8             # -
9             Symfony\Component\Security\Guard\Token\PostAuthenticationGuardToken
10            # If you're using authenticator-based security (introduced in
11            Symfony 5.1), you have to use this one:
12            # -
13            Symfony\Component\Security\Http\Authenticator\Token\PostAuthenticationToken
```

By default, the only uncommented class is `UsernamePasswordToken`, which is *perfect* for us.

But notice the last comment. If you're authenticating via a *custom* authenticator - like we were doing earlier - then you should use this class.

Let see *exactly* why that's the case. Open our custom `LoginFormAuthenticator`. We're not using this anymore, but pretend we are. This extends `AbstractLoginFormAuthenticator`:

`src/Security/LoginFormAuthenticator.php`

```
↕ // ... lines 1 - 15
16 use
   Symfony\Component\Security\Http\Authenticator\AbstractLoginFormAuthenticator
↕ // ... lines 17 - 26
27 class LoginFormAuthenticator extends AbstractLoginFormAuthenticator
28 {
↕ // ... lines 29 - 81
82 }
```

Hold `Cmd` or `Ctrl` to open that... then open *its* base class `AbstractAuthenticator`. Scroll down a bit and... hello `createAuthenticatedToken()`! This returns a new `PostAuthenticatedToken`. And so, by default, *this* is the token class you get with a custom authenticator.

These token classes *aren't* super important... they basically all extend the same `AbstractToken`... and mostly just help to identify *how* the user logged in.

By leveraging this knowledge, along with the `scweb` configuration, you can tell the two-factor bundle *which* authenticators require two-factor authentication and which don't.

Oh, and if you're using *two* custom authenticators... and only *one* of them needs two-factor authentication, you'll need to create a *custom* token class and override the `createAuthenticatedToken()` method in your authenticator to return that. *Then* you can target *just* the custom class here.

Phew! It may not feel like we've done much yet... other than listen to me talk about tokens... but the bundle *is* now... basically set up. But next, we need to choose *how* our users will get the tokens. Will we email them? Or will they use an authenticator app with a QR code? We're going to do the second.

Chapter 43: 2fa with TOTP (Time-Based One Time Password)

It may not *feel* like it yet, but the bundle *is* now set up... except for one big missing piece: how do we want our users to *get* the temporary token they'll enter into the form?

In the docs, there are 3 choices... well kind of only 2. These first two are where you use an authenticator app - like Google authenticator or Authy. The other option is to send the code via email.

We're going to use this "totp" authentication, which is basically the same as Google authenticator and stands for "time-based one-time password".

The logic for this actually lives in a separate library. Copy the Composer require line, find your terminal, and paste:

```
composer require "scheb/2fa-totp:^5.13"
```

This time there's no recipe or anything fancy: it just installs the library. Next, if you head back to the documentation, we need to enable this as an authentication method inside the config file.

That's back in `config/packages/scheb_2fa.yaml`. Paste that at the bottom:

```
config/packages/scheb_2fa.yaml
```

```
1 # See the configuration reference at
  https://github.com/scheb/2fa/blob/master/doc/configuration.md
2 scheb_two_factor:
3 // ... lines 3 - 8
9     totp:
10         enabled: true
```

Implementing TwoFactorInterface

The last step, if you look over at the docs, is to make our `User` implements a `TwoFactorInterface`. Open up our user class: `src/Entity/User.php`, add `TwoFactorInterface`:

```
src/Entity/User.php
↕ // ... lines 1 - 9
10 use Scheb\TwoFactorBundle\Model\Totp\TwoFactorInterface;
↕ // ... lines 11 - 19
20 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
21 {
↕ // ... lines 22 - 253
254 }
```

Then head down to the bottom. Now go to the "Code"->"Generate" menu - or `Command+N` on a Mac - and choose implement methods to generate the 3 we need:

```
src/Entity/User.php
↕ // ... lines 1 - 8
9 use Scheb\TwoFactorBundle\Model\Totp\TotpConfigurationInterface;
↕ // ... lines 10 - 19
20 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
21 {
↕ // ... lines 22 - 239
240     public function isTotpAuthenticationEnabled(): bool
241     {
242         // TODO: Implement isTotpAuthenticationEnabled() method.
243     }
244
245     public function getTotpAuthenticationUsername(): string
246     {
247         // TODO: Implement getTotpAuthenticationUsername() method.
248     }
249
250     public function getTotpAuthenticationConfiguration(): ?
        TotpConfigurationInterface
251     {
252         // TODO: Implement getTotpAuthenticationConfiguration() method.
253     }
254 }
```

Beautiful. Here's how TOTP authentication works. Each user that decides to activate two-factor authentication for their account will have a TOTP secret - a random string - stored on a property.

This will be used to validate the code *and* will be used to help the user set up their authenticator app when they first activate two-factor authentication.

The methods from the interface are fairly straightforward.

`isTotpAuthenticationEnabled()` returns whether or not the user has activated two-factor auth... and we can just check to see if the property is set. The

`getTotpAuthenticationUsername()` method is used to help generate some info on the QR code. The last method - `getTotpAuthenticationConfiguration()` - is the most interesting: it determines how the codes are generated, including the number of digits and how long each will last. Usually, authenticator apps generate a new code every 30 seconds.

Copy the `$totpSecret` property, scroll up to the properties in *our* class and paste:

```
src/Entity/User.php
↕ // ... lines 1 - 19
20 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
21 {
↕ // ... lines 22 - 63
64     /**
65      * @ORM\Column(type="string", length=255, nullable=true)
66      */
67     private $totpSecret;
↕ // ... lines 68 - 270
271 }
```

Then head back to the bottom and use the "Code"->"Generate" menu to generate a getter and setter for this. But we can make this nicer: give the argument a nullable string type, a `self` return type, and return `$this`... because the rest of our setters are "fluent" like this:

```

src/Entity/User.php
↕ // ... lines 1 - 19
20 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
21 {
↕ // ... lines 22 - 259
260     public function getTotpSecret(): ?string
261     {
262         return $this->totpSecret;
263     }
264
265     public function setTotpSecret(?string $totpSecret): self
266     {
267         $this->totpSecret = $totpSecret;
268
269         return $this;
270     }
271 }

```

For the getter... let's delete this entirely. We just won't need it... and it's kind of a sensitive value.

Let's fill in the three methods. I'll steal the code for the first... and paste:

```

src/Entity/User.php
↕ // ... lines 1 - 20
21 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
22 {
↕ // ... lines 23 - 245
246     public function isTotpAuthenticationEnabled(): bool
247     {
248         return $this->totpSecret ? true : false;
249     }
↕ // ... lines 250 - 266
267 }

```

For the username, in our case, return `$this->getUserIdentifier()`, which is really just our email:

```

src/Entity/User.php
↕ // ... lines 1 - 20
21 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
22 {
↕ // ... lines 23 - 250
251     public function getTotpAuthenticationUsername(): string
252     {
253         return $this->getUserIdentifier();
254     }
↕ // ... lines 255 - 266
267 }

```

For the last method, copy the config from the docs... and paste:

```

src/Entity/User.php
↕ // ... lines 1 - 20
21 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
22 {
↕ // ... lines 23 - 255
256     public function getTotpAuthenticationConfiguration(): ?
        TotpConfigurationInterface
257     {
258         return new TotpConfiguration($this->totpSecret,
            TotpConfiguration::ALGORITHM_SHA1, 30, 6);
259     }
↕ // ... lines 260 - 266
267 }

```

I'll re-type the end of `TotpConfiguration` and hit tab so that PhpStorm adds the `use` statement on top:

```

src/Entity/User.php
↕ // ... lines 1 - 8
9 use Scheb\TwoFactorBundle\Model\Totp\TotpConfiguration;
↕ // ... lines 10 - 20
21 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
22 {
↕ // ... lines 23 - 255
256     public function getTotpAuthenticationConfiguration(): ?
        TotpConfigurationInterface
257     {
258         return new TotpConfiguration($this->totpSecret,
            TotpConfiguration::ALGORITHM_SHA1, 30, 6);
259     }
↕ // ... lines 260 - 266
267 }

```

But, be careful. Change the 20 to 30, and the 8 to 6:

```

src/Entity/User.php
↕ // ... lines 1 - 20
21 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
22 {
↕ // ... lines 23 - 255
256     public function getTotpAuthenticationConfiguration(): ?
        TotpConfigurationInterface
257     {
258         return new TotpConfiguration($this->totpSecret,
            TotpConfiguration::ALGORITHM_SHA1, 30, 6);
259     }
↕ // ... lines 260 - 266
267 }

```

This says that each code should last for 30 seconds and contain 6 digits. The reason I'm using these exact values - including the algorithm - is to support the Google Authenticator app. Other apps, apparently, allow you to tweak these, but Google Authenticator doesn't. So if you want to support Google Authenticator, stick with this config.

Okay, our user system is ready! In theory, if we set a `totpSecret` value for one of our users in the database, and then tried to log in as that user, we would be redirected to the "enter your code" form. But, we're missing a step.

Next: let's add a way for a user to *activate* two-factor authentication on their account. When they do that, we'll generate a `totpSecret` and - most importantly - use it to show a QR code the

user can scan to set up their authenticator app.

Chapter 44: Activating 2FA

Ok: here's the flow. When we submit a valid email and password, the two-factor bundle will intercept that and redirect us to an "enter the code" form. To validate the code, it will read the `totpSecret` that's stored for that `User`:

```
src/Entity/User.php
↕ // ... lines 1 - 20
21 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
22 {
↕ // ... lines 23 - 64
65     /**
66      * @ORM\Column(type="string", length=255, nullable=true)
67      */
68     private $totpSecret;
↕ // ... lines 69 - 266
267 }
```

But in order to know what code to type, the user *first* needs to *activate* two-factor authentication on their account and scan a QR code we provide with their authenticator app.

Let's build *that* side of things now: the activation and QR code.

Oh, but before I forget again, we added a new property to our `User` in the last chapter... and I forgot to make a migration for it. At your terminal, run:

```
symfony console make:migration
```

Let's go check out that file:

migrations/Version20211012201423.php

```
↕ // ... lines 1 - 4
5 namespace DoctrineMigrations;
6
7 use Doctrine\DBAL\Schema\Schema;
8 use Doctrine\Migrations\AbstractMigration;
9
10 /**
11  * Auto-generated Migration: Please modify to your needs!
12  */
13 final class Version20211012201423 extends AbstractMigration
14 {
15     public function getDescription(): string
16     {
17         return '';
18     }
19
20     public function up(Schema $schema): void
21     {
22         // this up() migration is auto-generated, please modify it to your
needs
23         $this->addSql('ALTER TABLE user ADD totp_secret VARCHAR(255)
DEFAULT NULL');
24     }
25
26     public function down(Schema $schema): void
27     {
28         // this down() migration is auto-generated, please modify it to
your needs
29         $this->addSql('ALTER TABLE user DROP totp_secret');
30     }
31 }
```

And... good. No surprises, it adds one column to our table. Run that:

```
symfony console doctrine:migrations:migrate
```

Adding a way to Activate 2fa

Here's the plan. A user will *not* have two-factor authentication enabled by default. Instead, they'll *activate* it by clicking a link. When they do that, we'll generate a `totpSecret`, set it on the user, save it to the database and show the user a QR code to scan.

Head over to `src/Controller/SecurityController.php`. Let's create the endpoint that activates two-factor authentication: `public function enable2fa()`. Give this a route: how about `/authenticate/2fa/enable` - and `name="app_2fa_enable"`:

```
src/Controller/SecurityController.php
↕ // ... lines 1 - 12
13 class SecurityController extends BaseController
14 {
↕ // ... lines 15 - 33
34     /**
35      * @Route("/authenticate/2fa/enable", name="app_2fa_enable")
↕ // ... line 36
37     */
38     public function enable2fa(TotpAuthenticatorInterface
39                               $totpAuthenticator, EntityManagerInterface $entityManager)
40     {
↕ // ... lines 40 - 47
48     }
49 }
```

Just be careful not to start the URL with `/2fa`... that's kind of reserved for the two-factor authentication process:

```
config/packages/security.yaml
1 security:
↕ // ... lines 2 - 61
62     access_control:
↕ // ... lines 63 - 65
66         # This ensures that the form can only be accessed when two-factor
67         # authentication is in progress.
68         - { path: ^/2fa, role: IS_AUTHENTICATED_2FA_IN_PROGRESS }
↕ // ... lines 68 - 71
```

Inside of the method, we need two services. The first is an autowireable service from the bundle - `TotpAuthenticatorInterface $totpAuthenticator`. That will help us generate the secret. The second is `EntityManagerInterface $entityManager`:

src/Controller/SecurityController.php

```
↕ // ... lines 1 - 4
5 use Doctrine\ORM\EntityManagerInterface;
6 use
  Scheb\TwoFactorBundle\Security\TwoFactor\Provider\Totp\TotpAuthenticatorInterface;
↕ // ... lines 7 - 12
13 class SecurityController extends BaseController
14 {
↕ // ... lines 15 - 37
38     public function enable2fa(TotpAuthenticatorInterface
    $totpAuthenticator, EntityManagerInterface $entityManager)
39     {
↕ // ... lines 40 - 47
48     }
49 }
```

Oh, and, of course, you can only use this route if you're authenticated. Add `@IsGranted("ROLE_USER")`. Let me re-type that and hit tab to get the `use` statement on top:

src/Controller/SecurityController.php

```
↕ // ... lines 1 - 6
7 use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
↕ // ... lines 8 - 12
13 class SecurityController extends BaseController
14 {
↕ // ... lines 15 - 33
34     /**
↕ // ... line 35
36     * @IsGranted("ROLE_USER")
37     */
38     public function enable2fa(TotpAuthenticatorInterface
    $totpAuthenticator, EntityManagerInterface $entityManager)
39     {
↕ // ... lines 40 - 47
48     }
49 }
```

💡 Tip

This next paragraph is... wrong! Using `ROLE_USER` will not force a user to re-enter their password if they're only authenticated via a "remember me" cookie. To do that, you should use `IS_AUTHENTICATED_FULLY`. And that's what I should have used here.

For the most part, I've been using `IS_AUTHENTICATED_REMEMBERED` for security... so that you *just* need to be logged in... even if it's via a "remember me" cookie. But I'm using `ROLE_USER` here, which is effectively identical to `IS_AUTHENTICATED_FULLY`. That's on purpose. The result is that if the user *were* authenticated... but only thanks to a "remember me" cookie, Symfony will force them to re-type their password before getting here. A little extra security before we enable two-factor authentication.

Anyways, say `$user = $this->getUser()`... and then if *not* `$user->isTotpAuthenticationEnabled()`:

```
src/Controller/SecurityController.php
↕ // ... lines 1 - 12
13 class SecurityController extends BaseController
14 {
↕ // ... lines 15 - 37
38     public function enable2fa(TotpAuthenticatorInterface
    $totpAuthenticator, EntityManagerInterface $entityManager)
39     {
40         $user = $this->getUser();
41         if (!$user->isTotpAuthenticationEnabled()) {
↕ // ... lines 42 - 44
45     }
↕ // ... lines 46 - 47
48 }
49 }
```

Hmm, I want to see if totp authentication is *not* already enabled... but I'm not getting auto-completion for this.

We know why: the `getUser()` method only knows that it returns a `UserInterface`. We fixed this earlier by making our own base controller. Let's extend that:

```
src/Controller/SecurityController.php
↕ // ... lines 1 - 12
13 class SecurityController extends BaseController
14 {
↕ // ... lines 15 - 48
49 }
```

Back down here, if not `$user->isTotpAuthenticationEnabled()` - so if the user does *not* already have a `totpSecret` - let's set one: `$user->setTotpSecret()` passing `$totpAuthenticator->generateSecret()`. Then, save with `$entityManager->flush()`.

At the bottom, for now, just `dd($user)` so we can make sure this is working:

src/Controller/SecurityController.php

```
↕ // ... lines 1 - 12
13 class SecurityController extends BaseController
14 {
↕ // ... lines 15 - 37
38     public function enable2fa(TotpAuthenticatorInterface
    $totpAuthenticator, EntityManagerInterface $entityManager)
39     {
40         $user = $this->getUser();
41         if (!$user->isTotpAuthenticationEnabled()) {
42             $user->setTotpSecret($totpAuthenticator->generateSecret());
43
44             $entityManager->flush();
45         }
46
47         dd($user);
48     }
49 }
```

Linking to the Route

Cool! Let's link to this! Copy the route name... then open `templates/base.html.twig`. Search for "Log Out". There we go. I'll paste that route name, duplicate the entire `li`, clean things up, paste the new route name, remove my temporary code and say "Enable 2FA":

templates/base.html.twig

```
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 14
15 <body>
↕ // ... lines 16 - 21
22 <nav
23     class="navbar navbar-expand-lg navbar-light bg-light px-1"
24     {{ is_granted('ROLE_PREVIOUS_ADMIN') ? 'style="background-
color: red !important"' }}
25 >
26 <div class="container-fluid">
↕ // ... lines 27 - 35
36 <div class="collapse navbar-collapse" id="navbar-
collapsible">
↕ // ... lines 37 - 47
48 {% if is_granted('IS_AUTHENTICATED_REMEMBERED') %}
49 <div class="dropdown">
↕ // ... lines 50 - 60
61 <ul class="dropdown-menu dropdown-menu-end"
aria-labelledby="user-dropdown">
↕ // ... lines 62 - 68
69 <li>
70 <a class="dropdown-item" href="{{
path('app_2fa_enable') }}">Enable 2fa</a>
71 </li>
↕ // ... lines 72 - 74
75 </ul>
76 </div>
77 {% else %}
↕ // ... lines 78 - 79
80 {% endif %}
81 </div>
82 </div>
83 </nav>
↕ // ... lines 84 - 95
96 </body>
97 </html>
```

Testing time! Oh, but first, at your terminal, reload your fixtures:

symfony console doctrine:fixtures:load

That will make sure all of the users have *verified* emails so that we can actually log in. When that finishes, log in with `abraca_admin@example.com`, password `tada`. Beautiful. Then hit "Enable 2FA" and... got it! It hits our user dump! And most importantly, we have a `totpSecret` set!

That's great! But the *final* step is to show the user a QR code that they can scan to get their authenticator app set up. Let's do that next.

Chapter 45: Rendering the QR Code

Ok, we've just added a URL the user can go to in order to *enable* two-factor authentication on their account. What this *really* means is pretty simple: we generate a `totpSecret` and save it to their user record in the database. Thanks to this, when the user tries to log in, the 2-factor bundle will notice this and send them to the "fill in the code" form.

But, in order to *know* what code to enter, the user needs to set up an authenticator app. And to do *that*, we need to render a QR code they can scan.

Dumping the QR Content

How? The `$totpAuthenticator` has a method that can help. Try dumping `$totpAuthenticator->getQRContent()` and pass it `$user`:

```
src/Controller/SecurityController.php
↕ // ... lines 1 - 12
13 class SecurityController extends BaseController
14 {
↕ // ... lines 15 - 37
38     public function enable2fa(TotpAuthenticatorInterface
    $totpAuthenticator, EntityManagerInterface $entityManager)
39     {
↕ // ... lines 40 - 46
47         dd($totpAuthenticator->getQRContent($user));
48     }
49 }
```

When we refresh we see... a super weird-looking URL! *This* is the info that we need to send to our authenticator app. It contains our email address - that's just a label that will help the app - and most importantly the totp secret, which the app will use to generate the codes.

In theory, we could enter this URL manually into an authenticator app. But, pfff. That's crazy! In the real world, we translate this string into a QR code image.

Generating the QR Code

Fortunately, this is *also* handled by the Scheb library. If you scroll down a bit, there's a spot about QR codes. If you want to generate one, you need one last library. Actually, *right* after I recorded this, the maintainer deprecated this `2fa-qr-code` library! Dang! So, you *can* still install it, but I'll also show you how to generate the QR code *without* it. The library was deprecated because, well, it's pretty darn easy to create the QR code even without it.

Anyways, I'll copy that, find my terminal, and paste.

```
composer require "scheb/2fa-qr-code:^5.12.1"
```

To use the *new* way of generating QR codes - which I recommend - skip this step and instead run:

```
composer require "endroid/qr-code:^3.0"
```

While that's working. Head back to the docs... and copy this controller from the documentation. Over in `SecurityController`, at the bottom, paste. I'll tweak the URL to be `/authentication/2fa/qr-code` and call the route `app_qr_code`:

src/Controller/SecurityController.php

```
// ... lines 1 - 13
14 class SecurityController extends BaseController
15 {
// ... lines 16 - 50
51     /**
52      * @Route("/authentication/2fa/qr-code", name="app_qr_code")
53      */
54     public function displayGoogleAuthenticatorQrCode(QrCodeGenerator
55 $qrCodeGenerator)
56     {
57         // $qrCode is provided by the endroid/qr-code library. See the
58         docs how to customize the look of the QR code:
59         // https://github.com/endroid/qr-code
60         $qrCode = $qrCodeGenerator->getTotpQrCode($this->getUser());
61         return new Response($qrCode->writeString(), 200, ['Content-Type'
62 => 'image/png']);
61     }
62 }
```

I also need to re-type the "R" on `QrCodeGenerator` to get its use statement:

```
src/Controller/SecurityController.php
↕ // ... lines 1 - 6
7 use Scheb\TwoFactorBundle\Security\TwoFactor\QrCode\QrCodeGenerator;
↕ // ... lines 8 - 13
14 class SecurityController extends BaseController
15 {
↕ // ... lines 16 - 53
54     public function displayGoogleAuthenticatorQrCode(QrCodeGenerator
    $qrCodeGenerator)
55     {
↕ // ... lines 56 - 60
61     }
62 }
```

If you're using the *new* way of generating the QR codes, then your controller should like this instead. You can copy this from the code block on this page:

```
namespace App\Controller;

use Endroid\QrCode\Builder\Builder;
use Scheb\TwoFactorBundle\Security\TwoFactor\Provider\Totp\TotpAuthenticatorInt
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class SecurityController extends BaseController
{
    // ...

    /**
     * @Route("/authentication/2fa/qr-code", name="app_qr_code")
     * @IsGranted("ROLE_USER")
     */
    public function displayGoogleAuthenticatorQrCode(TotpAuthenticatorInterface
    {
        $qrCodeContent = $totpAuthenticator->getQRContent($this->getUser());
        $result = Builder::create()
            ->data($qrCodeContent)
            ->build();

        return new Response($result->getString(), 200, ['Content-Type' => 'imag
```

```
}  
}
```

This special endpoint *literally* returns the QR code *image*, as a png. Oh, and I forgot it here, but you should add an `@IsGranted("ROLE_USER")` above this: only authenticated users should be able to load this image.

Anyways, the user won't go to this URL *directly*: we'll use it inside an `img` tag. But to see if it's working, copy the URL, paste that into your browser and... sweet! Hello QR code!

Finally, after the user enables two-factor authentication, let's render a template with an image to this URL. Return `$this->render('security/enable2fa.html.twig')`.

Copy the template name, head into `templates/security`, and create that:

`enable2fa.html.twig`. I'll paste in a basic structure... it's just an `h1` that tells you to scan the QR code... but no image yet:

```
templates/security/enable2fa.html.twig
```

```
1  {% extends 'base.html.twig' %}  
2  
3  {% block title %}2fa Activation{% endblock %}  
4  
5  {% block body %}  
6    <div class="container">  
7      <div class="row">  
8        <div class="login-form bg-light mt-4 p-4">  
9          <h1 class="h3 mb-3 font-weight-normal">Use Authy or Google  
Authenticator to Scan the QR Code</h1>  
10  
11  
12      </div>  
13    </div>  
14  </div>  
15  {% endblock %}
```

Let's add it: an `img` with `src` set to `{{ path() }}` and then the route name to the controller we just built. So `app_qr_code`. For the alt, I'll say `2FA QR code`:

```
templates/security/enable2fa.html.twig
```

```
1 {% extends 'base.html.twig' %}
2
3 {% block title %}2fa Activation{% endblock %}
4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <h1 class="h3 mb-3 font-weight-normal">Use Authy or Google
Authenticator to Scan the QR Code</h1>
10
11             
12         </div>
13     </div>
14 </div>
15 {% endblock %}
```

Sweet! Time to try the whole flow. Start on the homepage, enable two-factor authentication and... yes! We see the QR code! We are ready to scan this and try logging in.

Making the User Confirm The Scanned the QR Code

Oh, but before we do, in a real app, I would probably add an extra property on my user, called `isTotpEnabled` and use *that* in the `isTotpAuthenticationEnabled()` method on my `User` class. Why? Because it would allow us to have the following flow. First, the user clicks "Enable two-factor authentication", we generate the `totpSecret`, save it, and render the QR code. So, *exactly* what we're doing now. *But*, that new `isTotpEnabled` flag would still be *false*. So if something went wrong and the user never scanned the QR code, they would *still* be able to log in *without* us requiring the code. *Then*, at the bottom of this page, we could add a "Confirm" button. When the user clicks that, we would *finally* set that `isTotpEnabled` property to true. Heck, you could even require the user to enter a code from their authenticator app to *prove* they set things up: the `TotpAuthenticatorInterface` service has a `checkCode()` method in case you ever want to manually check a code.

Next: let's scan this QR code with an authenticator app and finally try the full two-factor authentication flow. We'll then learn how to customize the "enter the code template" to match our design.

Chapter 46: QR Data & Scanning with an Authenticator App

Okay, status check. Any user can now enable two-factor authentication on their account by clicking this link. Behind the scenes, when they do that, we populate the `totpSecret` on the `User` object, save that to the database, and then render a QR code the user can scan. This QR code is a fancy image that contains two pieces of information. The first is the email of our user. Or, more precisely, if I scroll down to the "totp methods" in `User`, it contains whatever we return from `getTotpAuthenticationUsername()`:

```
src/Entity/User.php
↕ // ... lines 1 - 20
21 class User implements UserInterface, PasswordAuthenticatedUserInterface,
    TwoFactorInterface
22 {
↕ // ... lines 23 - 250
251     public function getTotpAuthenticationUsername(): string
252     {
253         return $this->getUserIdentifier();
254     }
↕ // ... lines 255 - 266
267 }
```

The second thing the QR code image contains is the `totpSecret`. In a minute, I'm going to scan this code with an authenticator app, which will allow me to generate the correct two-factor authentication code that I'll need to log in. It does that by leveraging that secret.

Adding Extra Info to the QR Code

But first, there *is* some *extra* info that we can add to the QR code. Head over to `config/packages/scheb_2fa.yaml`. Under `totp:`, one of the most important things that you can add is called an `issuer`. I'm going to set this to `Cauldron Overflow`:

```
config/packages/scheb_2fa.yaml
```

```
1 # See the configuration reference at
  https://github.com/scheb/2fa/blob/master/doc/configuration.md
2 scheb_two_factor:
3 // ... lines 3 - 8
9     totp:
10 // ... line 10
11     issuer: 'Cauldron Overflow'
```

That *literally* just added new information to the QR code image. Watch the image when we refresh. See that? It changed!

Thanks to this, in addition to the `email` and `totpSecret`, the code *now* contains an "issuer" key. If you want to learn about *all* the extra information that you can put here, check out the documentation or read about totp authentication in general. Because, for example, "issuer" is just a "totp concept"... that helps authenticator apps generate a label for our site when we scan this code.

Scanning with our Authenticator App

At this point, I want to pretend that we're a *real* user and test the *entire* flow. If we *were* a real user we would pull out our phone, open an authenticator app - like Google authenticator or Authy - and scan this code.

I like using Authy, here's what it looks like for me. I add a new account, scan and... got it! It reads my email and the "issuer" from the QR code and suggests a name and logo. If your company is well-known, it might actually guess the correct logo, but you can also add an `image` to your QR code in the same way that we added the "issuer". When I accept this, it gives me codes!

Logging in

So we are ready! Let's try it! Log out... and then log back in with `abraca_admin@example.com`, password `tada`. Submit and... sweet! Instead of actually being logged in, we're redirected to the two-factor authentication page! This happened for two reasons. First, the user has two-factor authentication enabled on their account. Specifically, this `isTotpAuthenticationEnabled()` method returned true. Second, the security "token" - that internal thing that wraps your `User` object when you log in - well, it matches one of the

tokens in our configuration. Specifically, we get the `UsernamePasswordToken` when we log in via the `form_login` mechanism.

If we try going *anywhere* else on the site, it kicks us right back here. The only place we can go to is `/logout` if we wanted to cancel the process. This is because the two-factor bundle will now deny access to *any* page on our site unless you've explicitly allowed it via the `access_control` rules, like we did for `/logout` and for the URL showing this form. This form *is* ugly, but we'll fix that soon.

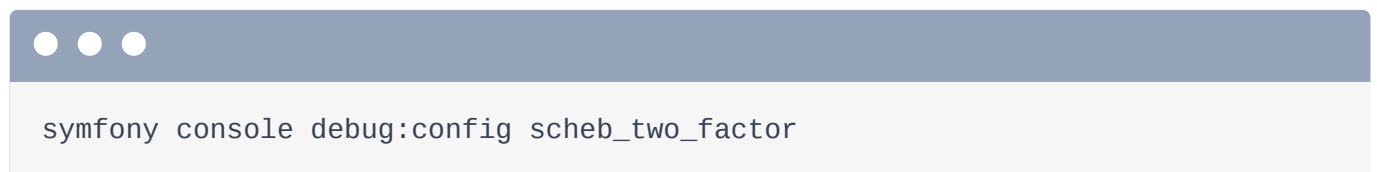
Ok, back to pretending I'm a real user. I'll open up my authenticator app, type in a valid code: 5, 3, 9, 9, 2, 2 and... got it! We're logged in! So cool!

Next, let's customize that two-factor authentication form... because it was *ugly*.

Chapter 47: Customize The 2-Factor Auth Form

We just successfully logged in using two-factor authentication. Woo! But, the form where we entered the code was *ugly*. Time to fix that! Log out... then log back in... with our usual email... and password *tada*. Here's our ugly form.

How can we customize this? Well, the wonderful documentation, of course, could tell us. But let's be tricky and see if we can figure it out for ourselves. Find your terminal and load the current configuration for this bundle: `symfony console debug:config`... and then, find the config file, copy the root key - `scheb_two_factor` - and paste.

A terminal window with a dark blue header bar containing three white circles. The main area is light gray and contains the command `symfony console debug:config scheb_two_factor` in a monospaced font.

Awesome! We see `security_tokens` with `UsernamePasswordToken`... that's no surprise because that's what we have here. But this also shows us some default values that we have not specifically configured. The one that's interesting to *us* is `template`. *This* is the template that's currently rendered to show the two-factor "enter the code" page.

Overriding the Template

Let's go check it out. Copy most of the file name, hit `Shift+Shift`, paste and... here it is! It's not too complex: we have an `authenticationError` variable that renders a message if we type an invalid code.

Then... we basically have a form with an action set to the correct submit path, an input and a button.

To customize this, go down into the `templates/security/` directory and create a new file called, how about, `2fa_form.html.twig`. I'll paste in a structure to get us started:

```
templates/security/2fa_form.html.twig
```

```
1 {% extends 'base.html.twig' %}
2
3 {% block title %}Two Factor Auth{% endblock %}
4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <h1 class="h3 mb-3 font-weight-normal">Two Factor
Authentication</h1>
10
11             <p>
12                 Open your Authenticator app and type in the number.
13             </p>
14
15             FORM TODO
16         </div>
17     </div>
18 </div>
19 {% endblock %}
```

This extends `base.html.twig`... but there's nothing dynamic yet: the form is a big TODO.

So obviously, this isn't done... but, let's try to use it anyways! Back in `config/packages/scheb_2fa.yaml`, under `totp`, add `template` set to `security/2fa_form.html.twig`:

```
config/packages/scheb_2fa.yaml
```

```
1 # See the configuration reference at
  https://github.com/scheb/2fa/blob/master/doc/configuration.md
2 scheb_two_factor:
3 // ... lines 3 - 8
9     totp:
10 // ... lines 10 - 11
12         template: security/2fa_form.html.twig
```

Back at the browser, refresh and... yes! That's our template!

Oh, and now that this renders a full HTML page, we have our web debug toolbar again. Hover over the security icon to see one interesting thing. We're, sort of, authenticated, but with this special `TwoFactorToken`. And if you look closer, we don't have any roles. So, we are *kind of* logged in, but without any roles.

And also, the two-factor bundle executes a listener at the start of each request that guarantees the user can't try to navigate the site in this half-logged-in state: it stops all requests and redirects them back to this URL. And if you scroll down, even *on* this page, all security checks return ACCESS DENIED. The two-factor bundle hooks into the security system to cause this.

Anyways, let's fill in the form TODO part. For this, copy *all* of the core template, and paste it over our TODO:

```

↕ // ... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="login-form bg-light mt-4 p-4">
9             <h1 class="h3 mb-3 font-weight-normal">Two Factor
Authentication</h1>
10
11             <p>
12                 Open your Authenticator app and type in the number.
13             </p>
14
15             {% if authenticationError %}
16             <p>{{ authenticationError|trans(authenticationErrorData,
'SchebTwoFactorBundle') }}</p>
17             {% endif %}
18
19             {# Let the user select the authentication method #}
20             <p>{{ "choose_provider"|trans({}, 'SchebTwoFactorBundle') }}:
21                 {% for provider in availableTwoFactorProviders %}
22                     <a href="{{ path("2fa_login", {"preferProvider":
provider}) }}">{{ provider }}</a>
23                 {% endfor %}
24             </p>
25
26             {# Display current two-factor provider #}
27             <p class="label"><label for="_auth_code">{{
"auth_code"|trans({}, 'SchebTwoFactorBundle') }} {{ twoFactorProvider }}:
</label></p>
28
29             <form class="form" action="{{ checkPathUrl ? checkPathUrl:
path(checkPathRoute) }}" method="post">
30                 <p class="widget">
31                     <input
32                         id="_auth_code"
33                         type="text"
34                         name="{{ authCodeParameterName }}"
35                         autocomplete="one-time-code"
36                         autofocus
37                     {#
38                         https://www.twilio.com/blog/html-attributes-two-
factor-authentication-autocomplete
39                         If your 2fa methods are using numeric codes only,
add these attributes for better user experience:
40                         inputmode="numeric"
41                         pattern="[0-9]*"
42                     #}

```

```

43         />
44     </p>
45
46     {% if displayTrustedOption %}
47     <p class="widget"><label for="_trusted"><input
id="_trusted" type="checkbox" name="{{ trustedParameterName }}" /> {{
"trusted"|trans({}, 'SchebTwoFactorBundle') }}</label></p>
48     {% endif %}
49     {% if isCsrfProtectionEnabled %}
50     <input type="hidden" name="{{ csrfParameterName }}"
value="{{ csrf_token(csrfTokenId) }}">
51     {% endif %}
52     <p class="submit"><input type="submit" value="{{
"login"|trans({}, 'SchebTwoFactorBundle') }}" /></p>
53 </form>
54
55     {# The logout link gives the user a way out if they can't
complete two-factor authentication #}
56     <p class="cancel"><a href="{{ logoutPath }}">{{
"cancel"|trans({}, 'SchebTwoFactorBundle') }}</a></p>
57
58 </div>
59 </div>
60 </div>
61 {% endblock %}

```

Now... it's just a matter of customizing this. Change the error `p` to a `div` with `class="alert alert-error"`. That should be `alert-danger`... I'll fix it in a minute. Below, I'm going to remove the links to authenticate in a different way because we're only supporting totp. For the `input` we need `class="form-control"`. Then all the way down here, I'll leave these `displayTrusted` and `isCsrfProtectionEnabled` sections... though I'm not using them. You can activate these in the config. Finally, remove the `p` around the button, change it to a `button` - I just like those better - put the text inside the tag... then add a few classes to it.

Oh, and I'm also going to move the "Log Out" link up a bit... clean it up a little... and add some extra classes:

templates/security/2fa_form.html.twig

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="container">
7      <div class="row">
8          <div class="login-form bg-light mt-4 p-4">
↕ // ... lines 9 - 14
15      {% if authenticationError %}
16          <div class="alert alert-danger">{{
authenticationError|trans(authenticationErrorData, 'SchebTwoFactorBundle')
}}</div>
17      {% endif %}
18
19      <form class="form" action="{{ checkPathUrl ? checkPathUrl:
path(checkPathRoute) }}" method="post">
20          <p class="widget">
21              <input
↕ // ... lines 22 - 25
26                  class="form-control"
↕ // ... lines 27 - 33
34              />
35          </p>
36
37          {% if displayTrustedOption %}
38              <p class="widget"><label for="_trusted"><input
id="_trusted" type="checkbox" name="{{ trustedParameterName }}" /> {{
"trusted"|trans({}, 'SchebTwoFactorBundle') }}</label></p>
39          {% endif %}
40          {% if isCsrfProtectionEnabled %}
41              <input type="hidden" name="{{ csrfParameterName }}"
value="{{ csrf_token(csrfTokenId) }}">
42          {% endif %}
43          <a class="btn btn-link" href="{{ logoutPath }}">{{
"cancel"|trans({}, 'SchebTwoFactorBundle') }}</a>
44          <button type="submit" class="btn btn-primary">{{
"login"|trans({}, 'SchebTwoFactorBundle') }}</button>
45          </form>
46
47      </div>
48  </div>
49 </div>
50 {% endblock %}
```

Phew! With any luck, that should make it look *fairly* good. Refresh and... sweet! Bah, except for a little extra quotation on my "Login". I always do that. There we go, that looks better:

```
templates/security/2fa_form.html.twig
```

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="container">
7      <div class="row">
8          <div class="login-form bg-light mt-4 p-4">
↕ // ... lines 9 - 18
19      <form class="form" action="{{ checkPathUrl ? checkPathUrl:
    path(checkPathRoute) }}" method="post">
↕ // ... lines 20 - 43
44      <button type="submit" class="btn btn-primary">{{
    "login"|trans({}, 'SchebTwoFactorBundle') }}</button>
45      </form>
46
47      </div>
48  </div>
49 </div>
50 {% endblock %}
```

If we type an invalid code... error! Oh, but it's not red... the class should be `alert-danger`. That's why we test things! And now... that's better:

```
templates/security/2fa_form.html.twig
```

```
↕ // ... lines 1 - 4
5  {% block body %}
6  <div class="container">
7      <div class="row">
8          <div class="login-form bg-light mt-4 p-4">
↕ // ... lines 9 - 14
15      {% if authenticationError %}
16      <div class="alert alert-danger">{{
    authenticationError|trans(authenticationErrorData, 'SchebTwoFactorBundle')
    }}</div>
17      {% endif %}
↕ // ... lines 18 - 46
47      </div>
48  </div>
49 </div>
50 {% endblock %}
```

If we type a *valid* code from my Authy app, we've got it! Mission accomplished!

Also, even though we won't talk about them, the two-factor bundle also supports "backup codes" and "trusted devices" where a user can choose to skip future two-factor authentication on a specific device. Check out their docs for the details.

And... we made it! Congrats on your incredibly hard work! Security is *supposed* to be a dry, boring topic, but I absolutely *love* this stuff. I hope you enjoyed the journey as much as I did. If there's something we didn't cover or you still have some questions, we're here for you down in the comments section.

All right friends, see ya next time!

With <3 from SymphonyCasts