Part I

Security and Authentication

1

Quickstart Symfony security

1.1 Learn about Symfony security

There are several key Symfony reference pages to read when starting with security. These include:

- Introduction to security
- How to build a traditional login form
- Using CSRF protection

1.2 New project with open and secured routes (project `security01`)

We are going to quickly create a 2-page website, with an open home page (url `/`) and a secured admin page (at url `/admin`).

1.3 Create new project and add the security bundle library

Create a new project:

```
symfony new --full security01
```

Add the security bundle:

```
composer req symfony/security-bundle
```

Add the fixtures bundle (we'll need this later):

```
composer require orm-fixtures --dev
```

1.4 Make a Default controller

Let's make a Default controller `/src/Controller/DefaultController.php`:

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

Edit the route to be `/` and the internal name to be `homepage`:

```
/**
 * @Route("/", name="homepage")
 */
public function indexAction()
{
    $template = 'default/index.html.twig';
    $args = [];
    return $this->render($template, $args);
}
```

Change the template `/templates/default/index.html.twig` to be something like:

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

{% block body %}
    welcome to the home page
{% endblock %}
```

This will be accessible to everyone.

1.5 Make a secured Admin controller

Let's make a Admin controller:

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

This will be accessible to only to users logged in with `ROLE_ADMIN` security.

Edit the new `AdminController` in `/src/Controller/AdminController.php`. Add a `use` statement, to let us use the `@IsGranted` annotation:

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

Now we'll restrict access to the index action of our Admin controller using the `@IsGranted` annotation. Symfony security expects logged-in users to have one or more 'roles', these are simple text Strings in the form `ROLE_xxxx`. The default is to have all logged-in users having `ROLE_USER`, and they can have additional roles as well. So let's restrict our admin home page to only logged-in users that have the authentication `ROLE_ADMIN`:

```
/**
 * @Route("/admin", name="admin")
 * @IsGranted("ROLE_ADMIN")
 */
public function index()
{
    $template = 'admin/index.html.twig';
    $args = [];
    return $this->render($template, $args);
}
```

NOTE: We can **make up** whatever roles are appropriate for our application, e.g.:

```
ROLE_ADMIN
ROLE_STUDENT
ROLE_PRESIDENT
ROLE_TECHNICIAN
... etc.
```

Change the template `/templates/admin/index.html.twig` to be something like the following - a secret code we can only see if logged in:

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

{% block body %}
    <h1>Admin home</h1>

    here is the secret code to the safe:
    007123
{% endblock %}
```

That's it!

Run the web sever:

- visiting the Default page at `/` is fine, even though we have not logged in ag all
- however, visiting the the `/admin` page should result in an HTTP 401 error (Unauthorized) due to insufficient authentication. See Figure 1.1.

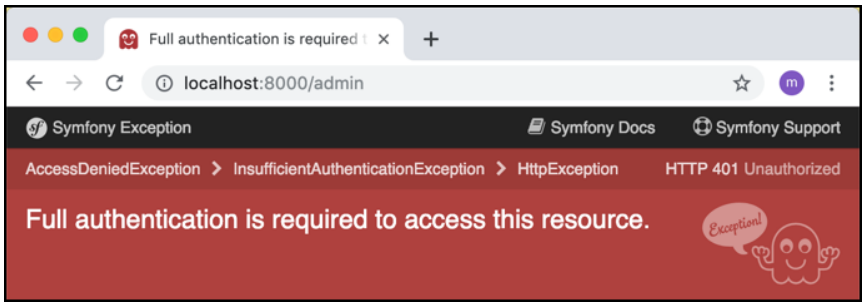

Figure 1.1: Screenshot of error attempting to visit `/admin`.

Of course, we now need to add a way to login and define different user credentials etc…

1.6 Core features about Symfony security

There are several related features and files that need to be understood when using the Symnfony security system. These include:

- **firewalls**
- **providers** and **encoders**
- **route protection** (we met this with `@IsGranded` controller method annotation comment above…)
- user **roles** (we met this as part of `@IsGranded` above `("ROLE_ADMIN")` …)

Core to Symfony security are the **firewalls** defined in `/config/packages/security.yml`. Symfony firewalls declare how route patterns are protected (or not) by the security system. Here is its default contents (less comments - lines starting with hash `#` character):

```
security:
    providers:
        users_in_memory: { memory: null }

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: lazy
            provider: users_in_memory

    access_control:
```

Symfony considers **every** request to have been authenticated, so if no login action has taken place then the request is considered to have been authenticated to be **anonymous** user `anon`. We can see in this `anon` user in Figure 1.2 this looking at the user information from the Symfony debug bar when visiting the default home page.

A Symfony **provider** is where the security system can access a set of defined users of the web application. The default for a new project is simply `in_memory` - although non-trivial applications have users in a database or from a separate API. We see that the `main` firewall simply states that users are permitted (at present) any request route pattern, and anonymous authenticated users (i.e. ones who have not logged in) are permitted.

The `dev` firewall allows Symfony development tools (like the profiler) to work without any authentication required. Leave it in `security.yml` and just ignore the `dev` firewall from this point onwards.

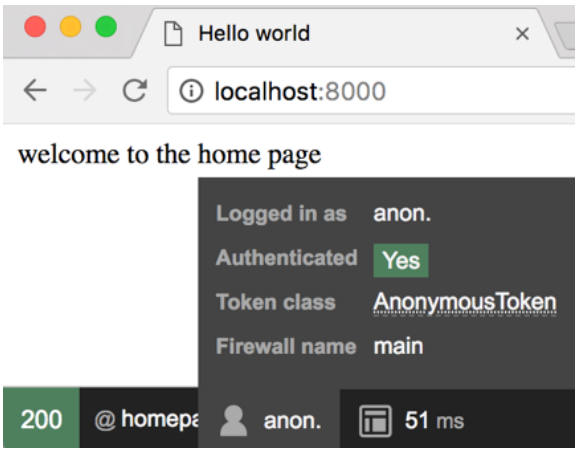

Figure 1.2: Symfony profiler showing anonymous user authentication.

1.7 Generating the special `User` Entity class (project `security02`)

Let's use the special `make:user` console command to create a `User` entity class that meets the requirements of providing user objects for the Symfony security system.

Enter the following at the command line, then just keep pressing `<RETURN>` to accept all the defaults:

```
$ php bin/console make:user

The name of the security user class (e.g. User) [User]:
>          // press <RETURN> to accept default

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
>          // press <RETURN> to accept default

Enter a property name that will be the unique "display" name for the user (e.g. email, user
>          // press <RETURN> to accept default

Will this app need to hash/check user passwords? Choose No if passwords are not needed or w

Does this app need to hash/check user passwords? (yes/no) [yes]:
>          // press <RETURN> to accept default

created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml
```

```
Success!
```

1.8 Review the changes to the `/config/packages/security.yml` file

If we look at `security.yml` it now begins as follows, taking into account our new `User` class:

```
security:
    encoders:
        App\Entity\User:
            algorithm: auto

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

1.9 Migrate new `User` class to your database

Since we've changed our Entity classes, we should migrate these changes to the database (and, of course, first create your database if you havce not already done so):

```
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

1.10 Make some `User` fixtures

Let's make some users with the `make:fixture` command:

```
php bin/console make:fixture UserFixtures
```

We'll use the Symfony sample code so that the plain-text passwords can be encoded (hashed) when stored in the database, see:

- https://symfony.com/doc/current/security.html#c-encoding-passwords

Edit your class `UserFixtures` to make use of the `PasswordEncoder`:

```php
<?php
namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;

use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
use App\Entity\User;

class UserFixtures extends Fixture
{
     private $passwordEncoder;

     public function __construct(UserPasswordEncoderInterface $passwordEncoder)
     {
          $this->passwordEncoder = $passwordEncoder;
     }

    public function load(ObjectManager $manager)
    {
        // (1) create object
        $user = new User();
        $user->setEmail('matt.smith@smith.com');
        $user->setRoles(['ROLE_ADMIN', 'ROLE_TEACHER']);

        $plainPassword = 'smith';
        $encodedPassword = $this->passwordEncoder->encodePassword($user, $plainPassword);

        $user->setPassword($encodedPassword);

        //(2) queue up object to be inserted into DB
        $manager->persist($user);

        // (3) insert objects into database
        $manager->flush();
    }
}
```

From the template class geneated for us, the first thing we need to do is add 2 `use` statements, to allow us to make use of the `User` entity class, and the `UserPasswordEncoderInterface` class:

```php
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
```

```
use App\Entity\User;
```

Next, to make it easy to encode passwords we'll add a new private instance variable `$passwordEncoder`, and a constructor method to initialise this object:

```
private $passwordEncoder;

public function __construct(UserPasswordEncoderInterface $passwordEncoder)
{
    $this->passwordEncoder = $passwordEncoder;
}
```

Finally, we can write the code to create a new `User` object, set its `email` and `roles` properties, encode a plain text password and set the encoded value to the object. This `$user` object needs to then be added to the queue of objects for the database (`persist(...)`), and then finally inserted into the database (`flush()`):

```
public function load(ObjectManager $manager)
{
    // (1) create object
    $user = new User();
    $user->setEmail('matt.smith@smith.com');
    $user->setRoles(['ROLE_ADMIN', 'ROLE_TEACHER']);

    $plainPassword = 'smith';
    $encodedPassword = $this->passwordEncoder->encodePassword($user, $plainPassword);

    $user->setPassword($encodedPassword);

    //(2) queue up object to be inserted into DB
    $manager->persist($user);

    // (3) insert objects into database
    $manager->flush();
}
```

NOTE: The `roles` property expects to be given an array of String roles, in the form `['ROLE_ADMIN', 'ROLE_SOMETHINGELSE', ...]`. These roles can be whatever we want for user:

```
$user->setRoles(['ROLE_ADMIN', 'ROLE_TEACHER']);
```

1.11 Run and check your fixtures

Load the fixtures into the database (with `doctrine:fixtures:load`), and check them with a simple SQL query `select * from user`:

```
php bin/console doctrine:query:sql "select * from user"
Cannot load Xdebug - it was already loaded

/php-symfony-5-book-codes-security-02-user/vendor/doctrine/dbal/lib/Doctrine/DBAL/Tools/Dump
array (size=1)
  0 =>
    array (size=4)
      'id' => string '1' (length=1)
      'email' => string 'matt.smith@smith.com' (length=20)
      'roles' => string '["ROLE_USER", "ROLE_ADMIN"]' (length=27)
      'password' => string '$2y$13$BInaG05FUpAHqcEBtGG05.G.qDbT5SNHoCI1nBHb58FILxJxFUmPu' (l
```

We can see the encoded password and roles `ROLE_USER` and `ROLE_ADMIN`

1.12 Creating a Login form

One new additional to the maker tool in Symfony 5 is automatic generation of a login form. Enter the following at the commadn line:

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

When prompted choose option `1`, a Login Form Authenticator:

```
What style of authentication do you want? [Empty authenticator]:
[0] Empty authenticator
[1] Login form authenticator
> 1
```

Next, give the name `LoginFormAuthenticator` for this new authenticator:

```
The class name of the authenticator to create (e.g. AppCustomAuthenticator):
> LoginFormAuthenticator
```

Accept the default (press `<RETURN>`) for the name of your controller class (`SecurityController`):

```
 Choose a name for the controller class (e.g. SecurityController) [SecurityController]:
 >
```

Accept the default (press `<RETURN>`) for creating a **logout** route (`yes`):

```
Do you want to generate a '/logout' URL? (yes/no) [yes]:
>
```

You should now have a new controller `SecurityController`, a login form `templates/security/login.html.twig`, an authenticator class `LoginFormAuthenticator`, and an updated set of security settings `config/packages/security.yaml`:

```
created: src/Security/LoginFormAuthenticator.php
updated: config/packages/security.yaml
created: src/Controller/SecurityController.php
created: templates/security/login.html.twig

 Success!
```

1.13 Check the new routes

We can check we have new login/logout routes from with the `debug:router` command:

```
 php bin/console debug:router
Cannot load Xdebug - it was already loaded
 ------------------------ -------- -------- ------ ----------------------------------
  Name                     Method   Scheme   Host   Path
 ------------------------ -------- -------- ------ ----------------------------------
  _preview_error           ANY      ANY      ANY    /_error/{code}.{_format}
    .... other _profiler debug routes here ...
  admin                    ANY      ANY      ANY    /admin
  homepage                 ANY      ANY      ANY    /
  app_login                ANY      ANY      ANY    /login
  app_logout               ANY      ANY      ANY    /logout
```

1.14 Allow any user to view the login form

Finally, we now have to edit our security firewall to allow **all** users, especially those not yet logged-in!, to access the `/login` route. Add the following line to the end of your `/config/packages/security.yml` configuration file:

```
        - { path: ^/login$, roles: IS_AUTHENTICATED_ANONYMOUSLY }
```

So the full `security.yml` file should look as follows (with comments removed):

```
security:
    encoders:
        App\Entity\User:
```

```
        algorithm: auto

providers:
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email

firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        anonymous: lazy
        provider: app_user_provider
        guard:
            authenticators:
                - App\Security\LoginFormAuthenticator
        logout:
            path: app_logout

access_control:
  - { path: ^/login$, roles: IS_AUTHENTICATED_ANONYMOUSLY }
```

1.15 Clear cache & visit `/admin`

Clear the cache (e.g. delete `/var/cache`), and open your browser to `/admin`. Since you are not currently logged-in, you should now be presented with a login form.

After we login with `matt.smith@smith.com` password = `smith`, we should now be able to see in the Symfony Profiler footer that we are logged in, and if we click this profiler footer, and then the `Security` link, we see this user has roles `ROLE_USER` and `ROLE_ADMIN`.

See Figure 1.2 this looking at the user information from the Symfony debug bar when visiting the default home page.

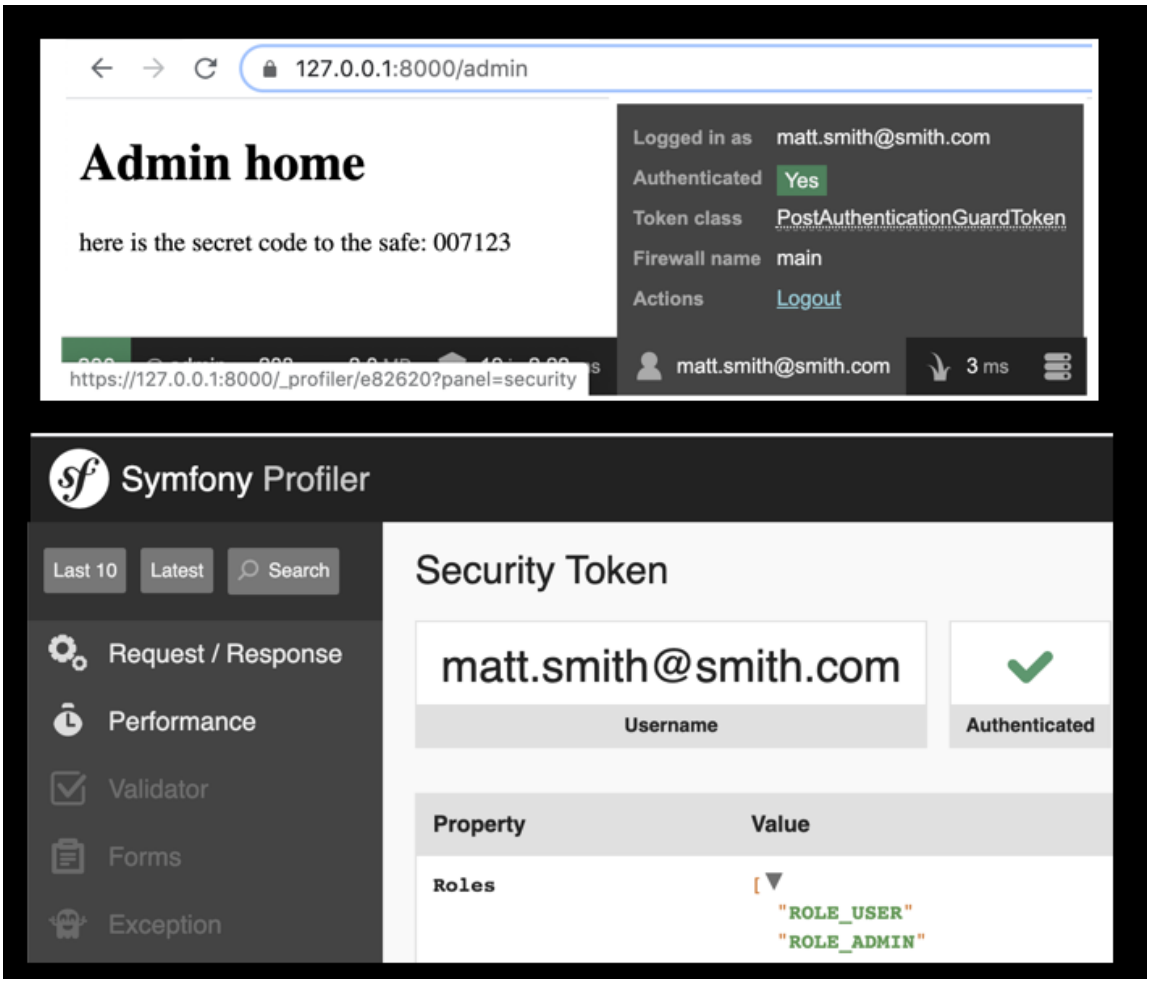

Figure 1.3: Symfony profiler showing ROLE_USER and ROLE_ADMIN authenticationm.

1.16 Using the `/logout` route

A logout route `/logout` was automatically added when we used the `make:auth` tool. So we can now use this route to logout the current user in several ways:

1. We can enter the route directly in the browser address bar, e.g. via URL:

   `http://localhost:8000/logout`

2. We can also logout via the Symfony profile toolbar. See Figure 1.4.

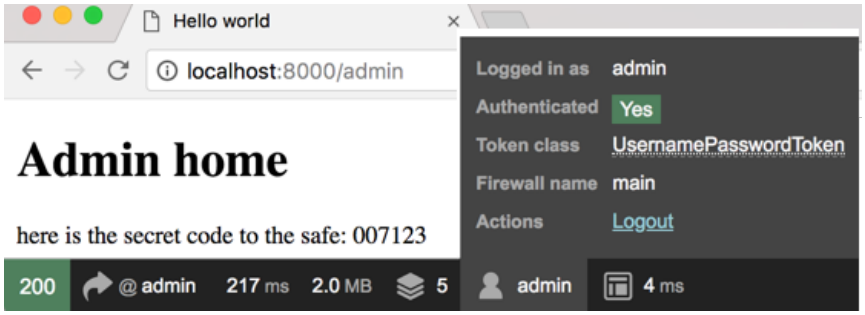

Figure 1.4: Symfony profiler user logout action.

In either case we'll logout any currently logged-in user, and return the anonymously authenticated user `anon` with no defined authentication roles.

1.17 Finding and using the internal login/logout route names in `SecurityController`

Look inside the generated `/src/controller/SecurityController.php` file to see the annotation route comments for our login/lgout routes:

```
...

class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        ...
    }
```

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

We can add links for the user to login/logout on any page in a Twig template, by using the Twig `url(...)` function and passing it the internal route name for our logout route `app_logout`, e.g.

```
<a href="{{ url('app_logout') }}">
    logout
</a>
```

2

Security users from database

2.1 Improving UserFixtures with a `createUser(...)` method (project `security03`)

Since making users in our `UserFixtures` class is very important, let's add a **helper** method to make it very clear what the properties of each new `User` object will be. See how clear the following is, if we have an exrta method `createUser(...)`:

We need a `load(...)` method, that gets invoked when we are loading fixtures from the CLI. This method creates objects for the entities we want in our database, and the saves (persists) them to the database:

```
public function load(ObjectManager $manager)
{
    // create objects
    $userUser = $this->createUser('user@user.com', 'user');
    $userAdmin = $this->createUser('admin@admin.com', 'admin', ['ROLE_ADMIN']);
    $userMatt = $this->createUser('matt.smith@smith.com', 'smith', ['ROLE_ADMIN', 'ROLE_SUPER_ADMIN

    // add to DB queue
    $manager->persist($userUser);
    $manager->persist($userAdmin);
    $manager->persist($userMatt);
```

```
        // send query to DB
        $manager->flush();
    }
```

Rather than put all the work in the `load(...)` method, we can create a helper method to create each new object. Method `createUser(...)` creates and returns a reference to a new `User` object given some parameters:

```
    private function createUser($username, $plainPassword, $roles = ['ROLE_USER']):User
    {
        $user = new User();
        $user->setUsername($username);
        $user->setRoles($roles);

        // password - and encoding
        $encodedPassword = $this->encodePassword($user, $plainPassword);
        $user->setPassword($encodedPassword);

        return $user;
    }
```

NOTE: The default role is `ROLE_USER` if none is provided.

2.2 Loading the fixtures

Loading fixtures involves deleting all existing database contents and then creating the data from the fixture classes - so you'll get a warning when loading fixtures. At the CLI type:

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

That's it!

You should now be able to access `/admin` with either the `matt.smith@smith.com/smith` or `admin@admin.com/admin` users. You will get an Access Denied exception if you login with `user@user.com/user`, since that only has `ROLE_USER` privileges, and `ROLE_ADMIN` is required to visit `/admin`.

See Figure 4.1 to see the default Symfony (dev mode) Access Denied exception page.

The next chapter will show you how to deal with (and log) access denied exceptions ...

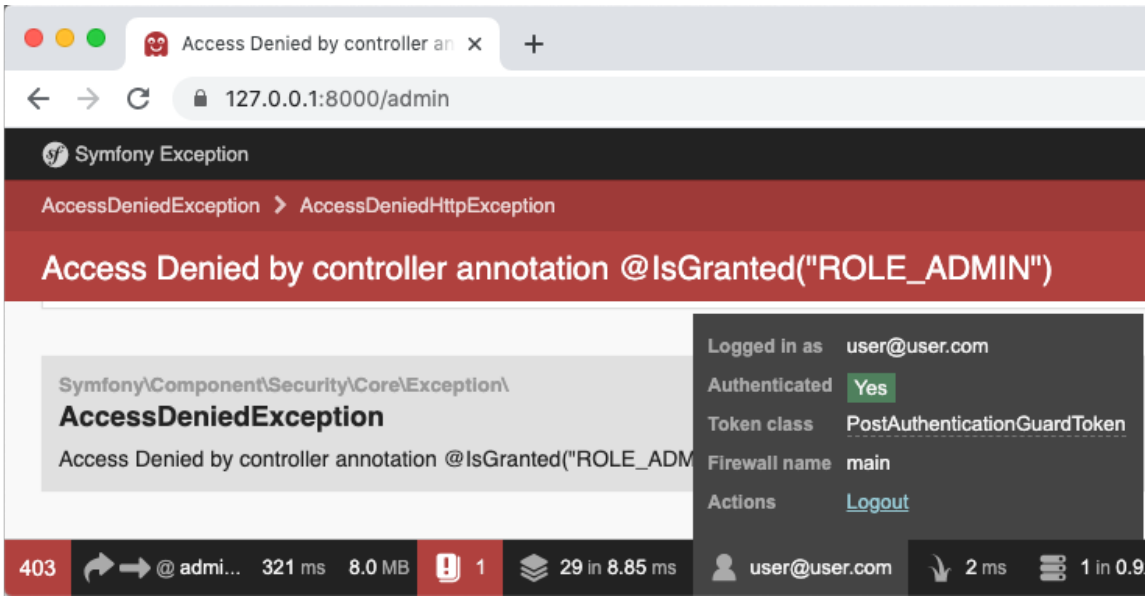

Figure 2.1: Screenshot of Default Symfony access denied page.

2.3 Using SQL from CLI to see users in DB

To double check your fixtures have been created correctly in the database, you could run an SQL query from the CLI:

```
$ php bin/console doctrine:query:sql "SELECT * FROM user"
Cannot load Xdebug - it was already loaded

/php-symfony-5-book-codes-security-03-create-user/vendor/doctrine/dbal/lib/Doctrine/DBAL/Tools/Dump
array (size=3)
  0 =>
    array (size=4)
      'id' => string '2' (length=1)
      'email' => string 'user@user.com' (length=13)
      'roles' => string '["ROLE_USER"]' (length=13)
      'password' => string '$2y$13$yfMogZlZfDQ3cJeib6Q2kOqXemYBs.4/AnyK/RbAFp69.360N60ai' (length=6
  1 =>
    array (size=4)
      'id' => string '3' (length=1)
      'email' => string 'admin@admin.com' (length=15)
      'roles' => string '["ROLE_ADMIN"]' (length=14)
      'password' => string '$2y$13$9UyVwrOluOkxLaH57IJM7uPF/NN7iKdBby.z9im2vx4531elfT80a' (length=6
  2 =>
    array (size=4)
```

```
'id' => string '4' (length=1)
'email' => string 'matt.smith@smith.com' (length=14)
'roles' => string '["ROLE_ADMIN", "ROLE_SUPER_ADMIN"]' (length=34)
'password' => string '$2y$13$4/yo6pKgUgECygZHbawemOSeANK78Cu6bGtKKbSgByFLFxASSlC3u' (l
```

3

Custom login page

3.1 A D.I.Y. (customisable) login form (project `security04`)

When we created the Authenticator it created a login form Twig template for us:

```
$ php bin/console make:auth

...

created: src/Controller/SecurityController.php
created: templates/security/login.html.twig
```

This is just a Twig template, and we should feel free to look inside and edit it ourselves …

3.2 Simplifying the generated login Twig template

The generated Twig login page is fine - but you should become confident in making it your own.

Start by replacing it with this simple, standard HTML login form:

```
<form method="post">
    <h1>Login</h1>

    Username:
```

```
        <input value="{{ last_username }}" name="email" id="inputEmail" autofocus>

        <p>
        Password:
        <input type="password" name="password" id="inputPassword">

        <input type="submit" value="Login">

    </form>
```

The form is shown when the `/login` URL is visited, or Symfony is redirected to internal route `app_login`, with the HTTP `GET` method. There is not `action` attribute for the `<form>` element, so the form is submitted to the same router, but using the `post` method.

Two name/value form variables are submitted:

- `email` - the email address being used as the unique username
- `password` - the password

3.3 CSRF (Cross Site Request Forgery) protection

Although this Twig template will present a login form to the user, it will **not** be accepted by the Symfony security system, due to an esxposre to CSRF security vulnerability.

NOTE: For any public **production** site you should always implement CSRF protection. This is implemented using CSRF 'tokens' created on the server and exchanged with the web client and form submissions. CSRF tokens help protect web applications against cross-site scripting request forgery attacks and forged login attacks.

Symfony expects forms to submit a special form variable `_csrf_token`. In Symfony this token can be generated using Twig function `csrf_token('authenticate')`. So we need to add this as a hidden form variable for our D.I.Y. form to work:

```
    <form method="post">
        <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">

        ... as before
    </form>
```

Learn more about CSRF threats and security:

- Symfony CSRF protection
- Wikipedia

When using the Symfony generated login form (as we created in this chapter) the CSRF token protection is built-in automatically.

3.4 Display any errors

We are only missing one more important set of data from Symfony - any errors to be displayed due to a previous invalid form submission. We should always check for an `error` object, and if present display its `messageData` values as follows (here I've added some CSS to add some padding and a pink background colour):

```
<form method="post">
    <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">

    {% if error %}
        <div style="background-color: pink; padding: 1rem;">
            {{ error.messageKey|trans(error.messageData, 'security') }}
        </div>
    {% endif %}

   <h1>Login</h1>

    Username:
    <input value="{{ last_username }}" name="email" id="inputEmail" autofocus>

    <p>
    Password:
    <input type="password" name="password" id="inputPassword">

    <input type="submit" value="Login">
</form>
```

Above we can see the following in our Login Twig template:

```
- the HTML `<form>` open tag, which we see submits via HTTP `POST` method

    - no action is given, so the form will submit to the same URL as displayed the form (`/login`), but

- add the security CSRF token as a hidden form variable


- display of any Twig `error` variable received
```

```
- the `username` label and text input field

    - with 'sticky' form last username value (`last_username`) if any found in the Twig variable

- the `password` label and password input field

- the submit button named `Login`
```

3.5 Custom login form when attempting to access `/admin`

See Figure 3.1 to see our custom login form in action.

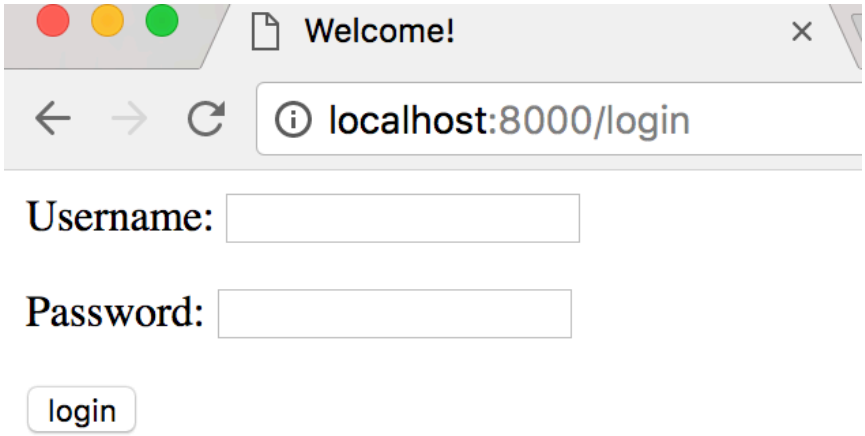

Figure 3.1: Screenshot of custom login form.

3.6 Path for successful login

If the user visits the path `/login` directly in the browser, Symfony needs to know where to direct the user if login is successful. This is defined in method `onAuthenticationSuccess` in class `Security/LoginFormAuthenticator`. If no redirect is defined, then the `TODO` Exception will be thrown:

```
        throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
```

Since we have a secure **admin** page, then let's redirect to route `admin`:

```
    public function onAuthenticationSuccess(Request $requesṢt, TokenInterface $token, $providerK
    {
        if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
            return new RedirectResponse($targetPath);
```

```
        }

        return new RedirectResponse($this->urlGenerator->generate('admin'));
    }
```

If you want to redirect to different pages, depending on the **role** of the newly logged-in user, then do the following:

- get the array of string roles from `$token` with `$token->getRoles()`
- add `IF`-statement(s) returning a different named route depending on their role, e.g. something like:

```
if(in_array('ROLE_ADMIN', $roles){
    return new RedirectResponse($this->urlGenerator->generate('index_admin'));
}

// else direct to basic staff homme page - or whatever ...
return new RedirectResponse($this->urlGenerator->generate('index_staff'));
```

4

Custom AccessDeniedException handler

4.1 Symfony documentation for 403 access denied exception

For details about this topic visit the Symfony documentation:

- https://symfony.com/doc/current/security/access_denied_handler.html

4.2 Declaring our handler (project `security05`)

In `/config/packages/security.yml` we need to declare that the class we'll write below will handle access denied exceptions.

So we add this line to the end of our `main` firewall in `security.yml`:

```
    access_denied_handler: App\Security\AccessDeniedHandler
```

So the full listing for our `security.yml` is now:

```
security:
    encoders:
        App\Entity\User:
            algorithm: bcrypt

    providers:
        our_db_provider:
```

```
            entity:
                class: App\Entity\User
                property: username

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true
            provider: our_db_provider
            form_login:
                login_path: login
                check_path: login
            logout:
                path:   /logout
                target: /
            access_denied_handler: App\Security\AccessDeniedHandler
```

4.3 The exception handler class

Now we needs to write our exception handler class in `/src/Security`.

Create new class `AccessDeniedHandler` in file `/src/Security/AccessDeniedHandler.php`:

```
namespace App\Security;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Http\Authorization\AccessDeniedHandlerInterface;

class AccessDeniedHandler implements AccessDeniedHandlerInterface
{
    public function handle(Request $request, AccessDeniedException $accessDeniedException)
    {
        return new Response('sorry - you have been denied access', 403);
    }
}
```

That's it!

Now if you try to access `/admin` with `user/user` you'll see the message 'sorry - you have been denied access' on screen. See Figure 4.1.

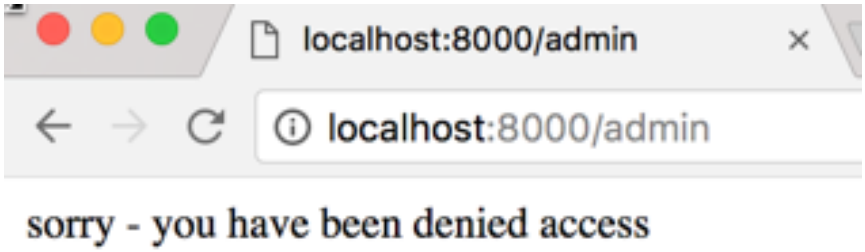

Figure 4.1: Screenshot of Custom Twig access denied page.

Although it won't be generated through the Twig templating system - we'll learn how to do that next ...

5

Twig and logging

5.1 Getting reference to Twig and Logger objects

There are many useful service objects available in the Symfony system via the 'Service Container'. This is a design pattern known as **Dependency Injection**. In Symfony we get access to a servce object by **Type Hinting** with the server or interface class name, in the parameter parentheses of the method or constructor of the class.

In this chapter we'll use this technique to get a reference to the Twig and Logger service objects.

Learn more in the Symfony documentation:

- https://symfony.com/doc/current/service_container.html
- https://symfony.com/doc/current/components/dependency_injection.html

5.2 Using Twig for access denied message (project `security06`)

Let's improved our Access Denied exception handler in 2 ways:

- display a nice Twig template
- log the exception using the standard Monolog logging system

First add Monolog to our project with Composer:

```
$ composer req logger
```

Now we will refactor class `AccessDeniedHandler` to

```
namespace App\Security;

use Psr\Log\LoggerInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Http\Authorization\AccessDeniedHandlerInterface;

class AccessDeniedHandler implements AccessDeniedHandlerInterface
{
    private $twig;
    private $logger;

    public function __construct(ContainerInterface $container, LoggerInterface $logger)
    {
        $this->twig = $container->get('twig');
        $this->logger = $logger;
    }

}
```

Now we can re-write method `handle(...)` to log an error message, and

```
public function handle(Request $request, AccessDeniedException $accessDeniedException)
{
    $this->logger->error('access denied exception');

    $template = 'error/accessDenied.html.twig';
    $args = [];
    $html = $this->twig->render($template, $args);
    return new Response($html);
}
```

5.3 The Twig page

Create a new folder `error` in our `/templates` folder, and in that create new Twig template `accessDenied.html.twig` for our nicer looking error page:

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

```
{% block title %}error{% endblock %}

{% block body %}
    sorry - access is denied for your request
    <p>
        <a href="{{ url('homepage') }}">home</a>
    </p>
{% endblock %}
```

Now, login in as `user@user.com` and try to visit `/admin`. We should get that access denied exception again, since this user does not have the required `ROLE_ADMIN` role privilege. See Figure 5.1 to see the error log register in the Symfony profiler footer, at the bottom of our custom error page.

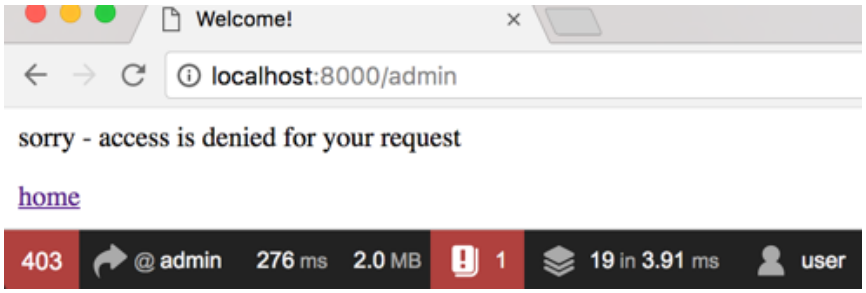

Figure 5.1: Screenshot of Custom Twig access denied page.

If you click on the red error you'll see details of all logged messages during the processing of this request. See Figure 5.2.

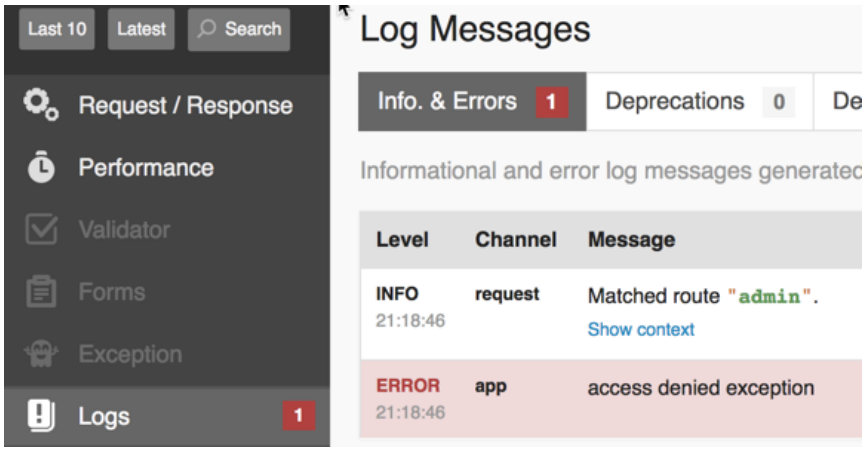

Figure 5.2: Screenshot of Profiler log entries.

5.4 Terminal log

You'll also see a red highlighted error appear in the terminal window if you are serving this website project with the Symfony web server:

```
[OK] Web server listening on https://127.0.0.1:8000 (PHP FPM 7.3.8)

    Mar 10 17:11:55 |WARN | SERVER GET  (403) /admin ip="127.0.0.1"
    Mar 10 18:11:54 |INFO | REQUES Matched route "admin". method="GET" request_uri="https://127.
    Mar 10 18:11:55 |DEBUG| SECURI Checking for guard authentication credentials. authenticators
        ... a bunch more DEBUG logs ....
    Mar 10 18:11:55 |DEBUG| SECURI Access denied, the user is neither anonymous, nor remember-me

    Mar 10 18:11:55 |ERROR| APP    access denied exception  <<<<<< here is our acess denied logg
```

5.5 Learn more about logger and exceptions

Learn more about Symfony and the Monolog logger:

- Logging with Monolog

Learn more about custom exception handlers and error pages:

- Access Denied Handler
- Custom Error pages

6

User roles and role hierarchies

6.1 Simplifying roles with a hierarchy (project `security07`)

Let's avoid repeating roles in our program logic (e.g. IF `ROLE_USER` OR `ROLE_ADMIN`) by creating a hierarchy, so we can give `ROLE_ADMIN` all properties of `ROLE_USER` as well. We can easily create a role hierarchy in `/config/packages/security.yml`:

```
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER

    ... rest of 'security.yml' as before ...
```

In fact let's go one further - let's create a 3rd user role (`ROLE_SUPER_ADMIN`) and define that as having all `ROLE_ADMIN` privileges plus the `ROLE_USER` privileges that were inherited by `ROLE_ADMIN`:

```
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: ROLE_ADMIN

    ... rest of 'security.yml' as before ...
```

Now if we log in as a user with `ROLE_SUPER_ADMIN` we also get `ROLE_ADMIN` and `ROLE_USER` too!

6.2 Modify fixtures

Now we can modify our fixtures to make user `matt` have just `ROLE_SUPER_ADMIN` - the other roles should be inherited through the hierarchy:

Change `/src/DataFixtures/UserFixtures.php` as follows:

```
    public function load(ObjectManager $manager)
    {
        ...

        $userMatt = $this->createUser('matt.smith@smith.com', 'smith', ['ROLE_SUPER_ADMIN'])

        ...
```

6.3 Removing default adding of `ROLE_USER` if using a hierarchy

If we are using a hierarchy, we don't need always add `ROLE_USER` in code, so we can simplify our getter in our `User` Entity in `/src/Entity/User.php`:

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

We'll still see `ROLE_USER` for admin and super users, but in the list of **inherited** roles from the hierarchy. This is show in Figure 6.1.

Learn about user role hierarchies at:

- Symfony hierarchical roles

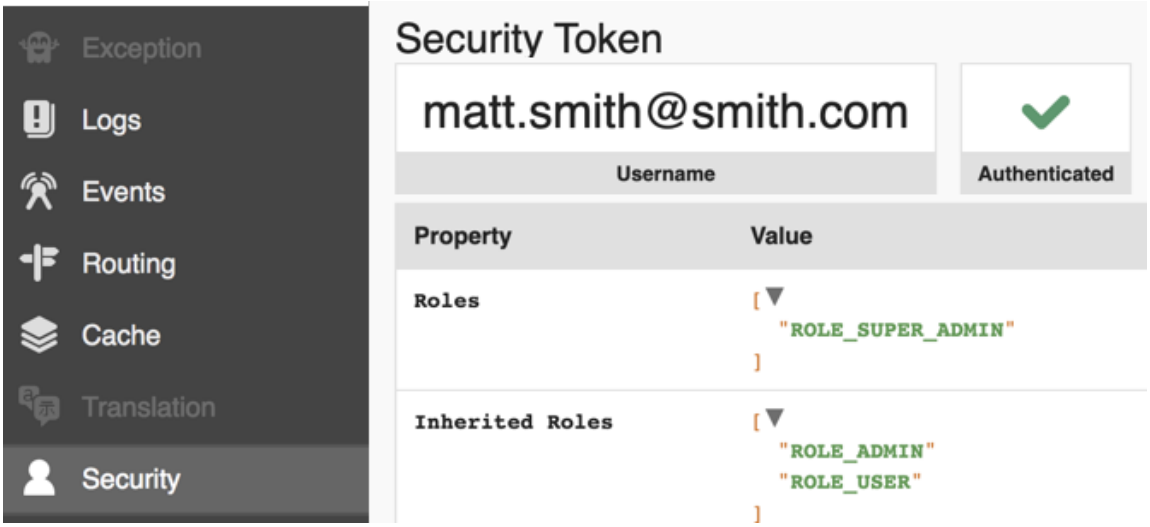

Figure 6.1: Super admin user inheriting `ROLE_USER`.

6.4 Allowing easy switching of users when debugging

If you wish to speed up testing, you can allow easy switching between users just by adding a but at the end of your request URL, **if** you add the following to your firewall:

```
switch_user: true
```

Now you can switch users bu adding the following at the end of the URL:

```
?_switch_user=<username>
```

You stop impersonating users by adding `?_switch_user=_exit` to the end of a URL.

For example to visit the home page as user `user` you would write this URL:

```
http://localhost:8000/?_switch_user=user
```

In your Twig you can allow this user to see special content (e.g. a link to exit impersonation) by testing for the special (automatically created role) `ROLE_PREVIOUS_ADMIN`:

```
{% if is_granted('ROLE_PREVIOUS_ADMIN') %}
    <a href="{{ path('admin_index', {'_switch_user': '_exit'}) }}">Exit impersonation & return to a
{% endif %}
```

Learn more at:

- Impersonating users

7

Customising view based on logged-in user

7.1 Twig nav links when logged in (project `security08`)

The Symfony security docs give us the Twig code for a conditional statement for when the current user has logged in:

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}
    <p>Username: {{ app.user.username }}</p>
{% endif %}
```

We can also test for which **role** a user may have granted when logged-in, e.g.:

```
{% if is_granted('ROLE_ADMIN') %}
      Welcome to the Admin home page ...
{% endif %}
```

We can use such conditionals in 2 useful and common ways:

1. Confirm the login username and offer a `logout` link for users who are logged in
2. Have navbar links revealed only for logged-in users (of particular roles)

So let's add such code to our `base.html.twig` master template (in `/templates`).

First, let's add a `<header>` element to either show the username and a logout link, or a link to login if the user is not logged-in yet:

```
<header>
```

```
    {% if is_granted('IS_AUTHENTICATED_FULLY') %}
        Username:
        <strong>{{ app.user.username }}</strong>
        <br>
        <a href="{{ url('app_logout') }}">logout</a>
    {% else %}
        <a href="{{ url('app_login') }}">login</a>
    {% endif %}
</header>
```

We can right align it and have a black bottom border with a little style in the `<head>`:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>

        <style>
            header {
                text-align: right;
                border-bottom: 0.5rem solid black;
                padding: 1rem;
            }
        </style>
```

Next, let's define a `<nav>` element, so that **all** users see a link to the homepage on every page on the website (at least those that extend `base.html.twig`). We will also add a conditional navigation link - to that users logged-in with `ROLE_ADMIN` can also see a link to the admin home page:

```
<nav>
    <ul>
        <li>
            <a href="{{ url('homepage') }}">home</a>
        </li>

        {% if is_granted('ROLE_ADMIN') %}
            <li>
                <a href="{{ url('admin') }}">admin home</a>
            </li>
        {% endif %}
    </ul>
</nav>
```

So when a user first visits our website homepage, they are not logged-in, so will see a `login` link in the header, and the navigation bar will only show a link to this homepage. See Figure 7.1.

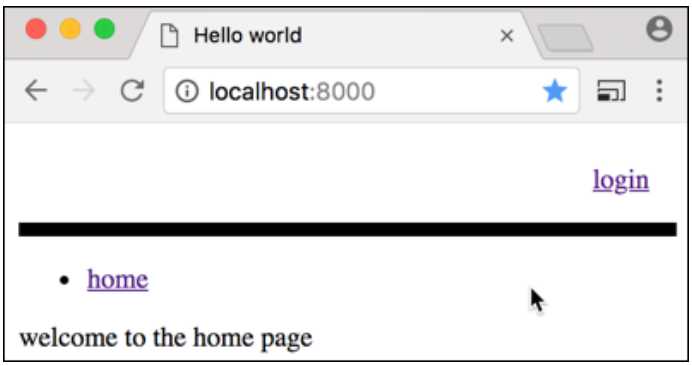

Figure 7.1: Screenshot of homepage before logging-in.

If the user has successfully logged-in with a `ROLE_ADMIN` privilege account, they will now see their userame and a `logout` link in the header, and they will also see revealed a link to the admin home page. See Figure 7.2.

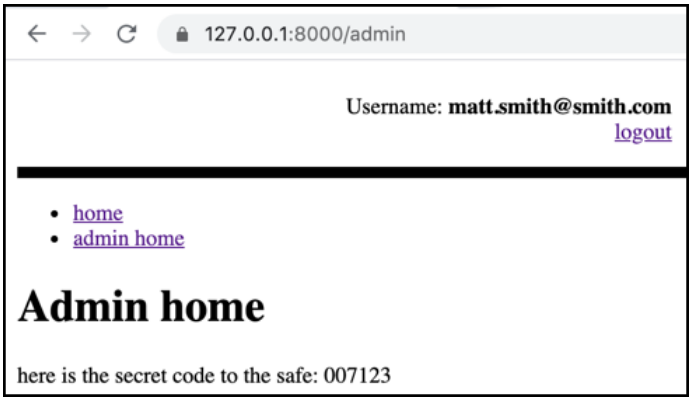

Figure 7.2: Screenshot of homepage after `ROLE_ADMIN` has logged-in.

7.2 Getting reference to the current user in a Controller

in PHP (e.g. a controller) you can get the user object as follows:

```
$user = $this->getUser();
```

or you can type-hint in a controller method declaration, and the param converter will provide the $security object for your to interrogate:

```
use Symfony\Component\Security\Core\Security;

public function indexAction(Security $security)
```

```
    {
        $user = $security->getUser();
    }
```

see:

- https://symfony.com/doc/current/security.html#a-fetching-the-user-object