ToDo-Co

Authentication

This documentation is about the authentication : how did we set up, how does it work, which files are concerned. We used Symfony 6.3, so if you need any more details about authentication, users or security, you can check out the official documentation : https://symfony.com/doc/current/security.html

Basic Set up

Install

- First, we installed the SecurityBundle :

  ```
  composer require symfony/security-bundle
  ```

Creating the User class

- Then, since we need some parts of the application to be accessible only to users, with specific security access, we created a User class, using the MakerBundle that comes with Symfony. We wanted a User class named User, to store the data in the database, to have the identification made through the username, and we want to check and hash our passwords, to make the authentication more secure. Knowing all those specs, here is the complete command :

  ```
  php bin/console make:user
  The name of the security user class (e.g. User) [User]:
  > User
  Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
  > yes
  Enter a property name that will be the unique "display" name for the user (e.g.
  email, username, uuid) [email]:
  > username
  Will this app need to hash/check user passwords? Choose No if passwords are not
  needed or will be checked/hashed by some other system (e.g. a single sign-on
  server). Does this app need to hash/check user passwords? (yes/no) [yes]:
  > yes

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

- Once we're done creating our User, or if we do any changes on the entity, we don't forget to migrate :

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

Loading the user : the provider, and how to change it

- Be careful, we've been setting up the user so that the username is the "display" name. The make:user command added a line in the security.yaml file accordingly, to set up the provider. (The provider loads and reloads datas about the user from the database, it can be used during the login, the Remember Me type of interactions, etc ...) Originally, we asked make:user to use the username as the display name, but if we want to use the email instead, for exemple, we will need to change two things :
  1. In the User entity file : We first had :

     ```
         public function getUserIdentifier(): string
         { return (string) $this->username;}
     ```

     We will want to make this change :

     ```
         public function getUserIdentifier(): string
         { return (string) $this->email;}
     ```

  2. In the security.yaml file : We first had :

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

     We will want to make this change :

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

Registering users : password hash and registering maker command

Symfony provides with a component taking care of the hashing (PasswordAuthenticatedUserInterface), but it must be set up properly :

- Make sure you use the component in the User.php file :

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

- Make sure your security.yaml file is set up properly :

  ```
  security:
   password_hashers:
       Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
  'auto'
  ```

  Initially, the password_hashers will be set up on 'auto', because Symfony apply the current best algorith automatically. During 6.3, it's bcrypt. If you want to specify any other hasher, you'll need to change it here.

Once you've set up your password hasher you can set up your registration :

- The form : Here, we used the make:form command to set back up the user form of the app, but later it would be better to set up the registration form with make:registration-form.
- The methods : We want to hash the password when we create a user and we update it, so in UserController.php we will need to set up the hashing like this :

  ```
    public function createUser(Request $request, UserPasswordHasherInterface
  $userPasswordHasher): Response
    {
        // ...
            $user->setPassword(
                $userPasswordHasher->hashPassword(
                    $user,
                    $form->get('password')->getData()
                );
        // ...
    }
  ```

  !! If you want to make any changes, be careful to modify the createUser function, and the editUser function.

The firewall : security rules and how to authenticate

The firewall part is where we set up how we want to authenticate (Login form, API token, ...). For this app, we chose to use the login form way to authenticate, and so we needed to create one. For this, we used the previous UserType.php form, already created, but for future improvements, we could use the make:auth command and rework the login form.

In that part, the informations has been set by the SecurityBundle, but we still need to set or change some parts. In the file security.yaml, we will need to enable the form_login. The important parts to check are the login_path and check_path : you have to set them up so they correspond to the login route. The same logic applies to the logout route, but you also need to set up the target route.

```
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
            form_login:
                # "app_login" is the name of the route created previously
                login_path: app_login
                check_path: app_login
            logout:
                path: app_logout
                target: app_home
```

Authenticating users

We needed a controller to set up the login and logout methods. For the, we used the make:controller command.

```
php bin/console make:controller Login

created: src/Controller/LoginController.php
created: templates/login/index.html.twig
```

Once created, we edited the LoginController.php according to the official documentation :

```
// ...
+ use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

  class LoginController extends AbstractController
  {
      #[Route('/login', name: 'app_login')]
-     public function index(): Response
+     public function index(AuthenticationUtils $authenticationUtils): Response
      {
+         // get the login error if there is one
+         $error = $authenticationUtils->getLastAuthenticationError();
+
+         // last username entered by the user
+         $lastUsername = $authenticationUtils->getLastUsername();
+
          return $this->render('login/index.html.twig', [
-             'controller_name' => 'LoginController',
+             'last_username' => $lastUsername,
+             'error'         => $error,
          ]);
      }
  }
```

The command also created a Twig template, that we named login.html.twig. If you need to make any changes on the login page, concerning the front or the form, you must modify that file.

We also enabled CSRF on the form login. To do this, we added it in the security.yaml file, like so :

```
security:
    # ...

    firewalls:
        secured_area:
            # ...
            form_login:
                # ...
                enable_csrf: true
```

According to the official documentation, then, in the login.html.twig file, we used the csrf_token() function in the Twig template to generate a CSRF token and store it as a hidden field of the form. By default, the HTML field must be called _csrf_token and the string used to generate the value must be authenticate.

```
    <form action="{{ path('app_login') }}" method="post">
        <input type="text" id="username" name="_username" value="{{ last_username }}"
placeholder="Nom d'utilisateur">
        <input type="password" id="password" name="_password" placeholder="Mot de
passe">
        <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate')
}}">

        <button type="submit" class="btn-primary btn-large" id="se_connecter">Se
connecter</button>
    </form>
```

Access Control

With everything we've been setting up until now, we can have users, who can authenticate on the app. We have two different types of users, the basic user and the admin user. We have to make a few verifications / modifications to be able to control the routes each category of user can access or not :

1. The getRoles() method : Symfony uses the getRoles method on the User to determine which roles the logged in user has. This function can be found in the User.php file. Every user will always be given at least one role :

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

       return array_unique($roles);
   }
   ```

   You can apply the role you want, just make sure you start the naming with "ROLE_", otherwise it won't work.

2. The access_control : We now have an automated role for every users, but we want to specify which role can access where. We want our users with the role user to be able to access everything but the pages concerning the users. So, only the users with the role admin would access all routes starting with /users. For this, we go in the security.yaml file, and here is how we've been setting up the access :

   ```
   access_control:
       - { path: ^/users, roles: ROLE_ADMIN }
   ```

3. Add the roles in the form Until now, every user only had the ROLE_USER, we want now to be able to add or modify the ROLE_ADMIN. For this, we will go in the UserType.php file, the form used to create / modify the users. To be able to add the ROLE_ADMIN, we added a checkbox, telling if the user is an admin or not :

```
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        // ....
        ->add('isAdmin', CheckboxType::class, [
            'mapped' => false,
            ])
    ;
}
```

If a user tries to access a route, and access is not granted, a special AccessDeniedException is thrown and no more code in the controller is called. Then, one of two things will happen:

- If the user isn't logged in yet, they will be asked to log in (redirected to the login page).
- If the user is logged in, but does not have the ROLE_ADMIN role, they'll be shown the 403 access denied page.