المعهد العالي للدراسات التكنولوجية بمدنين
Institut Supérieur des Etudes Technologiques de Médenine

Atelier Framework Coté Serveur

TP6

Gestion des utilisateurs (Security Bundle)

(Symfony 6)

Enseignante : K.MECHLOUCH	Classe : L2DSI

1. Introduction

Symfony fournit plusieurs outils pour sécuriser les applications. Certains outils de sécurité liés à HTTP, tels que les cookies de session sécurisée et la protection CSRF, sont fournis par défaut. *SecurityBundle* fournit toutes les fonctionnalités d'authentification et d'autorisation nécessaires.

2. Travail à faire

1. Installation

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

Après l'installation un fichier de configuration (security.yaml) sera créé :

```
# config/packages/security.yaml
security:
    enable_authenticator_manager: true
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
    providers:
        users_in_memory: { memory: null }
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
```

```
        lazy: true
        provider: users_in_memory

        # activate different ways to authenticate
        # https://symfony.com/doc/current/security.html#firewalls-authentication

        # https://symfony.com/doc/current/security/impersonating_user.html
        # switch_user: true

    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }
```

Pour assurer la sécurité au sein d'une application, on doit configurer les éléments suivants :

- **L'utilisateur**

Toute section sécurisée d'une application nécessite un certain concept d'utilisateur.

- **Le pare-feu**

Le pare-feu est au cœur de la sécurisation de l'application. Chaque demande est vérifiée si elle nécessite un utilisateur authentifié. Le pare-feu se charge également d'authentifier cet utilisateur (par exemple à l'aide d'un formulaire de connexion) ;

- **Contrôle d'accès (autorisation)**

À l'aide du contrôle d'accès et du vérificateur d'autorisation, on peut contrôler les autorisations requises pour effectuer une action spécifique ou visiter une URL spécifique.

2. Les utilisateurs

Les autorisations dans Symfony sont toujours liées à un objet utilisateur. On doit avoir une classe qui implémente UserInterface.

Le moyen le plus simple pour générer une classe d'utilisateurs consiste à utiliser la commande make:user du MakerBundle :

```
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]:
 > email

 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
```

```php
// src/Entity/User.php
namespace App\Entity;

use App\Repository\UserRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
use Symfony\Component\Security\Core\User\UserInterface;

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

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

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

    /**
     * @var string The hashed password
     * @ORM\Column(type="string")
     */
    private $password;

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

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

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

    return $this;
}

/**
 * The public representation of the user (e.g. a username, an email address, etc.)
 *
 * @see UserInterface
 */
public function getUserIdentifier(): string
{
    return (string) $this->email;
}

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

    return array_unique($roles);
}

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

    return $this;
}

/**
 * @see PasswordAuthenticatedUserInterface
 */
public function getPassword(): string
{
    return $this->password;
}

public function setPassword(string $password): self
```

```
    {
        $this->password = $password;

        return $this;
    }

    /**
     * Returning a salt is only needed, if you are not using a modern
     * hashing algorithm (e.g. bcrypt or sodium) in your security.yaml.
     *
     * @see UserInterface
     */
    public function getSalt(): ?string
    {
        return null;
    }

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

Exécutez la commande *make:registration-form* pour configurer le contrôleur d'inscription et ajouter des fonctionnalités telles que la vérification de l'adresse e-mail à l'aide de *SymfonyCastsVerifyEmailBundle* :

```
composer require symfonycasts/verify-email-bundle
php bin/console make:registration-form
```

3. Le pare-feu

La section pare-feu de *config/packages/security.yaml* est la section la plus importante. Un "pare-feu" est votre système d'authentification : le pare-feu définit quelles parties de votre application sont sécurisées et comment vos utilisateurs pourront s'authentifier (par exemple, formulaire de connexion, jeton API, etc.) :

```
# config/packages/security.yaml
security:
    # ...
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
```

```
        lazy: true
        provider: users_in_memory

        # activate different ways to authenticate
        # https://symfony.com/doc/current/security.html#firewalls-authentication

        # https://symfony.com/doc/current/security/impersonating_user.html
        # switch_user: true
```

4. Authentification

Lors de l'authentification, le système essaie de trouver un utilisateur correspondant au visiteur de la page Web. En général, cela se faisait à l'aide d'un formulaire de connexion ou d'une boîte de dialogue de base HTTP dans le navigateur.

La plupart des sites Web ont un formulaire de connexion où les utilisateurs s'authentifient à l'aide d'un identifiant (par exemple, une adresse e-mail ou un nom d'utilisateur) et un mot de passe.

- Créez un contrôleur User :

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

```
// src/Controller/UserController.php
namespace App\Controller;

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

class UserController extends AbstractController
{
   #[Route('/user', name: 'user')]
   public function index(): Response
   {
      return $this->render('user/index.html.twig', [
         'controller_name' => 'UserController',
      ]);
   }
}
```

- Activez l'authentificateur de connexion par formulaire à l'aide du paramètre *form_login* :

```
# config/packages/security.yaml
security:
   # ...

   firewalls:
      main:
```

```
        # ...
            form_login:
            # "login" is the name of the route
                login_path: login
                check_path: login
```

Une fois activé, le système de sécurité redirige les visiteurs non authentifiés vers le *login_path* lorsqu'ils tentent d'accéder à un lieu sécurisé.

- Modifiez le contrôleur de connexion pour afficher le formulaire de connexion :

```
// ...
 use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;
 class UserController extends AbstractController
 {
    #[Route('/login', name: 'login')]

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

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

       return $this->render('user/login.html.twig', [

         'last_username' => $lastUsername,
         'error'         => $error,
       ]);
    }
 }
```

- Finalement mettez à jour le template :

```
{# templates/user/login.html.twig #}
{% extends 'base.html.twig' %}

{# ... #}

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

   <form action="{{ path('login') }}" method="post">
      <label for="username">Email:</label>
      <input type="text" id="username" name="_username" value="{{ last_username
}}"/>
```

```
    <label for="password">Password:</label>
    <input type="password" id="password" name="_password"/>

    {# If you want to control the URL the user is redirected to on success
    <input type="hidden" name="_target_path" value="/account"/> #}

    <button type="submit">Login</button>
  </form>
{% endblock %}
```

N.b : pour traduire les messages d'erreur en français :
- composer require symfony/translation
- dans le fichier : config/packages/translation.yaml :

```
framework:
  default_locale: fr
  translator:
    default_path: '%kernel.project_dir%/translations'
```

- **Protection CSRF dans les formulaires de connexion**

Les attaques CSRF peuvent être évitées en ajoutant des jetons CSRF cachés dans les formulaires de connexion. Le fichier *Security* fournit déjà une protection CSRF, mais vous devez configurer certaines options avant de l'utiliser.

- Installez **security-csrf** :

```
composer require symfony/security-csrf
```

- Activer CSRF dans le formulaire de connexion :

```
# config/packages/security.yaml
security:
  # ...

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

- Ensuite, utilisez la fonction *csrf_token()* dans le modèle Twig pour générer un jeton CSRF et le stocker en tant que champ masqué du formulaire. Par défaut, le champ HTML doit s'appeler *_csrf_token* :

```
{# templates/security/login.html.twig #}

{# ... #}
<form action="{{ path('login') }}" method="post">
```

```
    {# ... the login fields #}

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

    <button type="submit">login</button>
</form>
```

- **Limitation des tentatives de connexion**

Symfony fournit une protection de base contre les attaques de connexion par force brute. Vous devez l'activer à l'aide du paramètre *login_throttling* :

```
# config/packages/security.yaml
security:
    # you must use the authenticator manager
    enable_authenticator_manager: true

    firewalls:
        # ...

        main:
            # ...

            # by default, the feature allows 5 login attempts per minute
            login_throttling: null

            # configure the maximum login attempts (per minute)
            login_throttling:
                max_attempts: 3

            # configure the maximum login attempts in a custom period of time
            login_throttling:
                max_attempts: 3
                interval: '15 minutes'
```

- **Déconnexion**

- Activez le paramètre de configuration **logout** sous votre pare-feu :

```
# config/packages/security.yaml
security:
    # ...

    firewalls:
        main:
            # ...
            logout:
                path: app_logout

                # where to redirect after logout
                # target: app_any_route
```

- Définir une route pour la déconnexion :

```
// src/Controller/UserController.php
namespace App\Controller;

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

class UserController extends AbstractController
{
 #[Route('/logout', name:'app_logout')]
  public function logout(): void
   {
      // controller can be blank: it will never be called!
      throw new \Exception('Don\'t forget to activate logout in security.yaml');
   }
}
```

5. Les autorisations

L'autorisation peut être géré de plusieurs façons :

- **Depuis le fichier security.yml :**

➢ **Les rôles**

Les rôles permettent à symfony de tagger des utilisateurs sur ce qu'ils peuvent ou non accéder. Ces rôles peuvent être hiérarchisés :

```
security:
   # ...

   role_hierarchy:
      ROLE_ADMIN:       ROLE_USER
      ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

Les *access controls* permettent de sécuriser un pattern d'url

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

- **Directement depuis les contrôleurs :**

```
public function test()
{
   // Le second paramètre permet de préciser sur quel objet le rôle est testé
   $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'Impossible d\'accéder à
cette page!');
```

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

- **Via condition:**

```
if(!$this->isGranted('ROLE_USER') && !$this->isGranted('ROLE_ADMIN')){
        throw $this->createAccessDeniedException('Impossible d\'accéder à cette
page!');
    }
```

- **Via *IsGranted* :**

```
use Symfony\Component\Security\Http\Attribute\IsGranted;
……..
#[IsGranted('ROLE_ADMIN', message: 'Accès impossible')]
 public function test()
……
```

- **Depuis les templates :**

```
{% if is_granted('ROLE_ADMIN') %} {% endif %}
```