To Do List App

Documentation technique

Système d'authentification

Elisa KLEIN
OpenClassRooms – Projet 8

Table des matières

Contexte

L'application ToDoList est accessible uniquement aux utilisateurs authentifiés. L'inscription d'un utilisateur se fait par le biais d'un compte administrateur. Seul un administrateur peut avoir accès à la gestion des utilisateurs de l'application.

L'authentification passe par un formulaire de connexion en renseignant un identifiant et un mot de passe.

Un système de rôle a été mis en place pour restreindre l'accès à certaines pages de l'application, ainsi qu'une restriction concernant la suppression d'une tâche qui doit être faite uniquement par l'auteur.

Dans ce document, vous retrouverez comment a été implémenté le système d'authentification du projet sous Symfony avec :

- Les différents fichiers utiles pour le paramétrage
- La procédure d'authentification
- La gestion des rôles utilisateur

Système d'authentification et d'autorisation

Ci-dessous la liste des fichiers utiles :

Fichier	Description
src/Entity/User.php	Classe entité utilisateur
src/Controller/SecurityController.php	Contrôleur contenant les routes de connexion et de déconnexion
src/Security/Voter/TaskDeleteVoter.php	Vérification de l'autorisation pour la suppression d'une tâche
config/packages/security.yaml	Paramètrage d'authentification

L'entité « User » : src/Entity/User.php

L'entité correspond à un utilisateur indispensable pour le fonctionnement du système d'authentification du projet.

Dans un premier temps, l'application nécessite l'installation du bundle *SecurityBundle*. Ce bundle fournit toutes les fonctionnalités d'authentification et d'autorisation nécessaires pour sécuriser l'application :

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

La création de l'entité utilisateur a été réalisé avec la commande suivante :

```
symfony console make:user
```

Cette commande permet de générer la classe « *User* », qui implémente l'interface *UserInterface*, en configurant différents éléments (nom de l'entité, utilisation de doctrine pour le stockage en base de données, le champ correspondant au login et l'encodage du mot de passe).

Dans l'entité « *User* », on retrouve notamment toutes les propriétés qui caractérisent un utilisateur avec une gestion de l'identifiant et de l'email unique.

```php
<?php

namespace App\Entity;

use App\Repository\UserRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;

#[ORM\Entity(repositoryClass: UserRepository::class)]
#[ORM\Table('users')]
#[UniqueEntity(fields: ['email'], message: 'Un compte est déjà existant avec cette email')]
#[UniqueEntity(fields: ['username'], message: 'Un compte est déjà existant avec ce nom d\'utilisateur')]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 25, unique: true)]
    #[Assert\NotBlank(message: "Vous devez saisir un nom d'utilisateur.")]
    #[Assert\Length(
        min: 2,
        max: 25,
        minMessage: "Le nom d'utilisateur doit contenir {{ limit }} caractères minimum",
        maxMessage: "Le nom d'utilisateur doit contenir {{ limit }} caractères maximum"
    )]
    private string $username;

    #[ORM\Column(length: 180, unique: true)]
    #[Assert\NotBlank(message: 'Vous devez saisir une adresse email.')]
    #[Assert\Email(message: "Le format de l'adresse n'est pas correcte.")]
    private string $email;

    /**
     * @var array<int, string>
     */
    #[ORM\Column]
    private array $roles = [];

    /**
     * @var string The hashed password
     */
    #[ORM\Column]
    #[Assert\NotBlank(message: 'Vous devez saisir un mot de passe.')]
    private string $password;
```

Les méthodes de la classe

Sachant que la classe implémente l'interface *UserInterface*, l'entité doit comprendre les méthodes suivantes en plus des setter et getter :

- **getUserIdentifier()** permet de renvoyer l'identifiant qui représente un utilisateur. Ici ,c'est l'attribut email, néanmoins, il est possible de définir une autre propriété de l'utilisateur (ex : username).

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

```

- **getRoles()** renvoi un tableau contenant les rôles d'un utilisateur. Par défaut, un utilisateur aura au minimum le rôle « Utilisateur » (ROLE_USER).

```php
/**
 * @see UserInterface
 */
public function getRoles(): array
{
    $roles = $this->roles;
    $roles[] = 'ROLE_USER';

    return array_unique($roles);
}
```

- **eraseCredentials()** permet de supprimer les données temporaires qui sont sensibles, notamment le mot de passe, contenu dans l'objet User. Ici, la fonctionnalité n'étant pas utilisée, la méthode est donc vide.

```php
/**
 * @see UserInterface
 */
public function eraseCredentials(): void
{
}
```

Concernant le hachage du mot de passe, la classe implémente également l'interface *PasswordAuthenticatedUserInterface* fournie par *SecurityBundle*. La classe *User* doit implémenter la méthode **getPassword()** qui retourne le mot de passe haché de l'utilisateur :

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

Le fichier de configuration Security.yaml : config/packages/security.yaml

Le fichier *security.yaml* décrit les règles d'authentification et d'autorisation de l'application. On retrouve les sections suivantes :

- **providers :** indique comment (re)charger les utilisateurs à partir d'un stockage sur la base d'un « identifiant d'utilisateur ». La configuration ci-dessous utilise Doctrine pour charger l'entité *User* en utilisant la propriété *username* comme « identifiant d'utilisateur ».

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

- **password_hashers :** indique le hacheur de mot de passe à utiliser avec le format « auto », c'est-à-dire que Symfony choisira le niveau le plus élevé possible.

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

- **firewalls :** c'est la partie essentielle du processus de sécurisation. C'est ce qui permet de définir quand il faut vérifier et authentifier un utilisateur.

```yaml
firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
            form_login:
                check_path: login_check
                login_path: login
                always_use_default_target_path:  true
                default_target_path: /
            entry_point: 'form_login'
            logout:
                target: /login
```

La partie « *dev* » montre que pour les routes (assets et profiler), il n'y a pas de vérification. Pour les autres routes (« *main* »), on indique qu'il faut utiliser le provider contant nos utilisateurs et le mode d'authentification choisi. Ici, on passe par un formulaire de connexion (*form_login*) en renseignant le path auquel un visiteur sera redirigé automatiquement lorsqu'il tente d'accéder à une page sécurisée de l'application (*login_path*).

- **access_control :** permet de définir pour chaque pattern d'URL quel rôle peut y accéder. Il est possible de définir autant de modèles d'URL que l'on souhaite, mais un seul sera trouvé par requête. C'est-à-dire que Symfony démarre en haut de la liste et s'arrête lorsqu'il trouve la première correspondance.

```yaml
access_control:
    - { path: ^/login, roles: PUBLIC_ACCESS}
    - { path: ^/users, roles: ROLE_ADMIN}
    - { path: ^/, roles: ROLE_USER }
```

  Préfixer le chemin avec « ^ » (ex : ^/users) signifie que seuls les URL commençant par ce modèle sont mis en correspondances.

  Ici, seule la page *login* est accessible par tous les visiteurs. Toutes les pages avec l'URL commençant par */users* sont restreints au rôle administrateur et le reste des pages au rôle utilisateur.

Toutefois, il est possible de gérer les autorisations en omettant ou en complétant l'*access_control* en passant par les annotations. Il est possible de rétreindre l'accès pour chaque route depuis les contrôleurs :

```
#[Route('/users', name: 'user_list', methods: ['GET'])]
#[IsGranted('ROLE_ADMIN')]
```

Le contrôleur SecurityController : src/Controller/SecurityController.php

C'est le contrôleur qui comprend les routes pour l'authentification et la déconnexion.

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class SecurityController extends AbstractController
{
    #[Route('/login', name: 'login', methods: ['GET', 'POST'])]
    public function loginAction(AuthenticationUtils $authenticationUtils): Response
    {
        if ($this->getUser()) {
            return $this->redirectToRoute('homepage');
        }

        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUsername = $authenticationUtils->getLastUsername();

        return $this->render(
            'security/login.html.twig',
            [
                'last_username' => $lastUsername,
                'error' => $error,
            ]
        );
    }

    #[Route('/login_check', name: 'login_check', methods: ['POST'])]
    public function loginCheck(): void
    {
        // This code is never executed.
    }

    #[Route('/logout', name: 'logout', methods: ['GET'])]
    public function logoutCheck(): void
    {
        // This code is never executed.
    }
}
```

- /login : pour l'affichage du formulaire de connexion
- /login_check : récupérée par les listeners pour le traitement du formulaire et donc permettre l'authentification grâce à l'authenticator de base de Symfony
- Logout : pour la déconnexion

Le Voter TaskDeleteVoter.php : src/Security/Voter/TaskDeleteVoter.php

Pour gérer l'autorisation de suppression d'une tâche uniquement par son auteur et d'une tâche anonyme uniquement par un administrateur, il a été mis en place un système de Voter. Un Voter est un outil centralisant les logiques d'autorisation en dehors du système de rôle dans le but de pouvoir les réutiliser.

```php
<?php

namespace App\Security\Voter;

use App\Entity\Task;
use App\Entity\User;
use Symfony\Bundle\SecurityBundle\Security;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;
use Symfony\Component\Security\Core\User\UserInterface;

class TaskDeleteVoter extends Voter
{
    public const DELETE = 'TASK_DELETE';

    private Security $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    protected function supports(string $attribute, mixed $subject): bool
    {
        return in_array($attribute, [self::DELETE]) && $subject instanceof Task;
    }

    protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool
    {
        /** @var User $user */
        $user = $token->getUser();
        if (!$user instanceof UserInterface) {
            return false;
        }

        switch ($attribute) {
            case self::DELETE:
                /** @var Task $task */
                $task = $subject;
                return $this->canDelete($task, $user);
        }

        return false;
    }

    private function canDelete(Task $task, User $user): bool
    {
        if ($user === $task->getAuthor()) {
            return true;
        }

        if (null === $task->getAuthor() && $this->security->isGranted('ROLE_ADMIN')) {
            return true;
        }

        return false;
    }
}
```

- supports() : permet de vérifier que le voter sera bien utilisé pour une entité précise et que l'attribut reçu est bien une des permissions qui a été définies dans les constantes. Ici TASK_DELETE.
- voteOnAttribute() contient la logique de vérification des permissions. Une fois l'étape du support passée, cette méthode retournera un booléen pour accepter ou non l'autorisation. L'instruction switch permet de dispatcher la logique des autorisations selon la condition reçue.
- canDelete() : vérifie que l'utilisateur connecté est l'auteur de la tâche à supprimer ou si c'est un administrateur qui est connecté et que la tâche est anonyme. La fonction retourne un booléen.

L'utilisation du Voter se fait depuis le contrôleur concerné en ajoutant en début de méthode :

$this->denyAccessUnlessGranted('TASK_DELETE', $task) ;

TAST_DELETE = l'autorisation qu'on veut vérifier

$task = l'objet de l'entité sur laquelle effectuer la vérification

```php
#[Route('/tasks/{id}/delete', name: 'task_delete', methods: ['DELETE'])]
public function deleteTaskAction(Task $task, Request $request, TaskDeleteHandler $handler): Response
{
    $this->denyAccessUnlessGranted('TASK_DELETE', $task);

```