Documentation technique : Implémentation de l’authentification

Sommaire

1 – Implémentation de l'authentification 3
1.1 – Installation du Security bundle 3
1.2 – Création de l'entité utilisateur 3
1.3 – Création du contrôleur gérant l'authentification 6
1.4 – Création du template login.html.twig 7
1.5 – Configuration du pare-feu 8
1.6 – Configuration du contrôle des accès 9

1 – Implémentation de l'authentification

1.1 – Installation du Security bundle

Pour pouvoir implémenter l'authentification, le module symfony/security-bundle doit être installé sur l'application. Ce module est installé par défaut lors de la création d'une application Symfony. Mais si ce n'est pas le cas, lancer la commande suivante :

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

Le fichier de configuration « config/packages/security.yaml » est alors généré automatiquement par Symfony Flex :

```yaml
security:
    # 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#loading-the-user-the-user-provider
    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#the-firewall

            # 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 }
```

1.2 – Création de l'entité utilisateur

Si aucune entité User existe dans l'application, il est possible de la créer simplement avec la commande suivante :

```
php bin/console make:user
```

Saisir le nom que l'on veut donner à l'entité User.
Par défaut, on la nommera User.

```
The name of the security user class (e.g. User) [User]:
> User
```

Il nous est ensuite demandé si on veut que les données de l'entité User soient stockées dans la base de données.
On répondra oui, car on veut pouvoir conserver les utilisateurs créés.

```
Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
> yes
```

Une propriété unique doit être utilisée pour identifier l'utilisateur.
On utilisera la propriété « username » car l'application doit utiliser la combinaison nom d'utilisateur / mot de passe pour l'identification.

```
Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:
> username
```

Pour terminer, on nous demande si on veut utiliser un mot de passe crypté. On répondra bien évidemment oui pour des questions de sécurité car on ne veut pas que les mots de passes apparaissent en clair.

```
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]:
>
```

Une fois toutes ces informations saisies, une entité User est créée automatiquement dans le dossier Entity. La classe UserRepository est également créée dans le dossier Repository.

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

 Success!

Next Steps:
  - Review your new App\Entity\User class.
  - Use make:entity to add more fields to your User entity and then run make:migration.
  - Create a way to authenticate! See https://symfony.com/doc/current/security.html
```

Le fichier « config/packages/security.yaml » a également été modifié automatiquement et les lignes suivantes ont été rajoutées à la configuration :

```yaml
providers:
    # used to reload user from session & other features (e.g. switch_user)
    app_user_provider:
        entity:
            class: App\Entity\User
            property: username
```

Cela va indiquer à Symfony quel classe et propriété utiliser pour connecter un utilisateur.

Lorsqu'on ouvre le fichier Entity/User.php, on voit que la classe implémente 2 interfaces. Elles sont nécessaires pour que l'authentification puisse fonctionner.

```php
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
```

La classe User doit alors obligatoirement implémenter les méthodes suivantes :
- getUserIdentifier(), getRoles(), et eraseCredentials() de UserInterface.
- getPassword() de PasswordAuthenticatedUserInterface.

Grâce à la commande « php bin/console make:user » ces méthodes sont générées automatiquement.

Implémente UserInterface

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

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

    return array_unique($roles);
}
```

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

Implémente PasswordAuthenticatedUserInterface

```php
public function getPassword(): string
{
    return $this->password;
}
```

Ces méthodes peuvent être personnalisées en fonction des besoins de l'application.

1.3 – Création du contrôleur gérant l'authentification

```php
<?php

namespace App\Controller;

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

 Saro0h +1
class SecurityController extends AbstractController
{
     Florian Pohu +1
    #[Route(path: '/login', name: 'login', methods: ['GET', 'POST'])]
    public function loginAction(AuthenticationUtils $authenticationUtils): Response|RedirectResponse
    {
        if ($this->isGranted( attribute: 'IS_AUTHENTICATED_FULLY')) {
            return $this->redirectToRoute( route: 'homepage');
        }

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

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

     Saro0h +1
    #[Route(path: '/logout', name: 'logout', methods: ['GET'])]
    public function logoutCheck()
    {
        // This code is never executed.
    }
}
```

La route « login » permet à l'utilisateur de se connecter à l'application en renseignant ses identifiants dans le formulaire de connexion.
Grâce au composant « AuthenticationUtils » injecté en paramètre de la méthode loginAction, on peut récupérer les erreurs lorsque l'authentification échoue ainsi que le nom de l'utilisateur pour les afficher dans la page « login.html.twig ».

La route « logout » est utilisée pour déconnecter un utilisateur. La redirection est paramétrée dans le fichier de configuration « config/packages/security.yaml ».

1.4 – Création du template login.html.twig

Créer un template nommé « login.html.twig » dans le dossier templates/security.

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

{% block body %}
    {% if error %}
        <div class="alert alert-danger" role="alert">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
    {% endif %}

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

        <label for="password">Mot de passe :</label>
        <input type="password" id="password" name="_password" />

        <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">
        <button class="btn btn-success" type="submit">Se connecter</button>
    </form>
{% endblock %}
```

Quand l'utilisateur soumet le formulaire, une requête http utilisant la méthode POST est envoyé sur la page « /login ». Le pare-feu de symfony vérifie si le couple l'identifiant / mot de passe est correct.

Dans le cas où l'utilisateur s'est connecté avec succès, il est redirigé vers la « homepage ».

Lorsque l'authentification échoue, on affiche le message d'erreur que l'on récupère grâce à la variable passée en paramètre du template dans le contrôleur. On affiche également le nom d'utilisateur utilisé lors de la dernière tentative d'authentification.

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

1.5 – Configuration du pare-feu

Pour que la vérification des identifiants se fasse correctement, il faut ajouter les lignes suivantes dans le fichier « config/packages/security.yaml »

```yaml
security:
    # 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 }
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
            form_login:
                login_path: login
                check_path: login
                enable_csrf: true
            logout:
                path: logout
                # where to redirect after logout
                target: login
```

Form_login : utilisation du FormLoginAuthenticator qui est un composant intégré à Symfony et qui permet d'authentifier un utilisateur en utilisant un identifiant et un mot de passe.

Login_path : chemin ou route du formulaire de connexion.

Check_path : chemin ou route qui vérifie les informations de connexion.

Enable_crsf : active la protection CSRF sur le formulaire de connexion.

Logout : configuration de la déconnexion d'un utilisateur.

Path : route qui déconnecte l'utilisateur.

Target : route vers laquelle l'utilisateur est redirigé après s'être déconnecté.

1.6 – Configuration du contrôle des accès

Pour restreindre les accès en fonction du type d'utilisateur il faut ajouter des lignes supplémentaires dans le bloc security du fichier « config/packages/security.yaml »

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

```yaml
    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        - { route: 'login', roles: PUBLIC_ACCESS }
        - { route: 'homepage', roles: ROLE_USER }
        - { path: '^/tasks', roles: ROLE_USER }
        - { path: '^/users', roles: ROLE_ADMIN }
    role_hierarchy:
        ROLE_ADMIN: ROLE_USER
```

Access_control : restriction des accès en fonction de routes / pattern de chemins.

Description des restrictions mises en place :

- La route login est accessible pour tout le monde, que l'utilisateur soit connecté ou non.
- La page d'accueil et les routes concernant la gestion des tâches sont accessibles uniquement pour les utilisateurs connectés.
- Seuls les administrateurs peuvent accéder aux pages de gestion des utilisateurs.

Role_hierarchy : défini une hiérarchie entre les rôles

Ici ROLE_ADMIN hérite du ROLE_USER.