Documentation technique d’authentification

Crée pour le projet « To Do List » sous Symfony 5.1

L’authentification du projet To Do List

Objectif

Cette documentation à pour but de comprendre comment a été mise en place l’authentification du projet « To Do list »

A savoir

L’authentification du projet se base sur la version 5,1, de Symfony. La version actuel utilisé pour le projet.
C’est sensiblement le même principe de configuration depuis la structure de la version 4,0 de Symfony.

Sommaire :

1. Le bundle de sécurité de Symfony

La toute première chose à faire est de vérifié la bonne mise en place du bundle de sécurité de symfony.

1-Vérification :

Utiliser la commande "symfony console debug:autowiring security" pour vérifier que le composant de sécurité de symfony soit bien installés.

Sinon il faudra l'installer.

2.1-Installation du bundle via composer

"composer require symfony/security-bundle«

Vous retrouverez dans le fichier composer.json, dans l'accolade "require " , la ligne "symfony/security-bundle " .

2.2-Activer le bundle (si vous n'avez pas symfony/flex qui le fait pour vous)

Ouvrez le fichier config/bundle.php

- Insérer la ligne suivante : Symfony\Bundle\SecurityBundle\SecurityBundle::class => ['all' => true],

2. L'entité « User »

1. Créer la classe « user »

Nous aurons un simple entité « User » sans rapport à l'authentification.
Il va comme tout autre entité simplement servir à stocker les informations et gérer les interactions avec la base de donnée.

Nous retrouverons donc nos propriétés :

« id », « name », « email », « password »

Ainsi que leurs méthodes « getter » et « setter »

2. Adapter notre class User pour l'authentification

Cette fois-ci nous allons adapter notre objets « user » pour l'authentification avec symfony

1 – Implémenter la class User avec la class UserInterface

2- Avoir obligatoirement les propriétés « username » et « password »

Nous allons donc changer la propriété « name » en « username » ainsi que ses méthodes.

3 –Ajouter les méthodes « getSalt », « eraseCredentials » qui retournerons null.
Et la méthode, « getRoles »qui doit retourner un **tableau** avec un rôle tel que ['ROLE_USER']

Dans notre cas, nous avons crée une propriété « role » ainsi que la méthode « setter ».
Celui-ci va sera donc gérer avec la base de donné pour récupérer le rôle associé, et ne sera pas systématiquement le ['ROLE_USER'].

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

use Symfony\Component\Security\Core\User\UserInterface;

class User implements UserInterface
{
    private $username;
    private $password;
    private array $roles = [];

    public function getId()
    {
        return $this->id;
    }

    public function getUsername()
    {
        return $this->username;
    }

    public function setUsername($username)
    {
        $this->username = $username;
    }

    public function getPassword()
    {
        return $this->password;
    }

    public function setPassword($password)
    {
        $this->password = $password;
    }

    public function getRoles(): ?array
    {
        $roles = $this->roles;

        return array_unique($roles);
    }

    public function eraseCredentials() {}

    public function getSalt(){}

}
```

3. Configurations des composants de sécurité

Nous allons configurer et gérer la gestion d' authentification avec le fichier config/packages/security.yaml

1 . Encoding password

Il va expliquer à l'encoder de symfony quand on va coder un mot de passe concernant l'entité « User ». Ce mot de passe suivra l'algorithme « **bcrypt** » pour le cryptage de celui-ci.

2. User provider

C'est lui qui va gérer les données de l'utilisateur, sauvegarder les sessions de l'utilisateur et autres.

On lui donne un **nom** au choix, il va à partir de l'**entité User**, récupérer les informations de l'utilisateur en se basant sur « **username** »

*On aurait pu à la place, définir l'**email** pour l'utilisation de celui-ci lors de la connexion avec le formulaire.*

3. Firewall

Nous pouvons créer plusieurs firewall.

• Le firewall **« dev »** est un faux . Il garantis simplement qu'on ne bloque pas les assets et profiler via les url indiqués.

• Sinon les firewall comme notre **« main »** on tous leurs propres politiques d'authentification.

Nous lui disons :

- A la première connexion soit identifier comme anonyme.
- Base toi sur notre provider que nous avons créer.
- D'utiliser le chemin « login » pour la connexion faisant référence à notre méthode pour la connexion.

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

    providers:
       in_database:
            entity:
                class: App\Entity\User
                property: username

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: true
            provider: in_database
            form_login:
                login_path: login
                check_path: login
```

4. Le controller et le formulaire de connexion

Création du controller de connexion

- Le path « login », soit le chemin indiqué dans le fichier « security.yaml »
- Le retour à la page où sera notre formulaire de connexion.
- L'ajout de l'affichage des erreurs pour notre formulaire

```php
/**
 * @Route("/login", name="login")
 */
public function loginAction(AuthenticationUtils $authenticationUtils)
{
    $error = $authenticationUtils->getLastAuthenticationError();

    return $this->render('security/login.html.twig');
}
```

Création du formulaire de connexion

- L'affichage des erreurs.
- Notre formulaire de connexion.

A savoir que pour les champs, les attributs « name » seront toujours les suivants.

- « _username » pour l'utilisation du login *(même si email utilisé)*
- « _password » pour le mot de passe

```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" />

        <label for="password">Mot de passe :</label>
        <input type="password" id="password" name="_password" />
        <button class="btn btn-success" type="submit">Se connecter</button>
    </form>
{% endblock %}
```

5. Les rôles

Par défaut les visiteurs sont connectés comme anonymes. Une fois connecté ils auront soit le rôle utilisateur, soit le rôle administrateur.
Nous allons gérer tout cela toujours avec le fichier « security.yaml »

1. Rôles hiérarchiques

Permet d'organiser **l'héritage des rôles**.
Dans notre cas, le rôle administrateur hérite de toutes les fonctionnalités du rôle utilisateur.

```yaml
role_hierarchy:
    ROLE_ADMIN: ROLE_USER
```

2. Accès aux pages selon leur rôles

Permet de dire qui à le droit d'accéder à certaines pages :

- Seul les personnes non connectés soit anonymes peuvent accéder à page de connexion.
- Seul les administrateurs peuvent accéder au pages commençant par « /users »
- Seuls les personnes ayant les rôles « users » ou « admin » peuvent accéder au site.

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

6. La déconnexion

Créer la méthode pour la déconnexion

Au sein de notre controller, crée notre méthode avec le path **« logout »**

```php
/**
 * @Route("/logout", name="logout")
 */
public function logoutCheck()
{
    // This code is never executed.
}
```

Configurer la déconnexion

Dans le fichier « sécurity.yaml » définir :

- Le chemin utilisé pour la déconnexion soit notre méthode crée juste avant.
- La redirection après la déconnexion.