Documentation technique sur l'implémentation de l'authentification

Réalisation : Jean-Eudes Nouaille-Degorce

Symfony
5.2

PLAN

Comment s’opère l’authentification ?

Quels fichiers ont été modifiés et pourquoi ?

1

Comment s’opère l’authentification ?

Fichier security.yaml

Le cœur du système d'authentification est globalement configuré dans le fichier security.yaml :

- La section « **encoders** » définit le type d'algorithme utilisé pour l'encodage des mots de passe.
- La section « **providers** » (fournisseur d'utilisateurs) définit l'entité utilisée pour l'authentification des utilisateurs. Le paramètre « property » correspond à la propriété d'identification d'un utilisateur qui est stockée en sessions.
- La section « **firewalls** » (pare-feu) définit la manière dont les utilisateurs peuvent s'authentifier. Elle spécifie les routes de connexion et de déconnexion des utilisateurs (« **form_login** »). Un pare-feu est activé lorsqu'il n'y a pas d'utilisateur connecté (« **anonymous** ») : cela permet de distinguer entre routes publiques et routes protégées.
- La section « **role_hierarchy** » définit la hiérarchie et l'héritage de rôles en fonction de rôles prédéfinis. Elle permet par la suite de régler les droits accordés à certains utilisateurs pour des opérations spécifiques.
- La section « **access_control** » permet de restreindre globalement ou d'autoriser l'accès à certaines pages du site en fonction des rôles accordés aux utilisateurs.

config/packages/security.yaml

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

    providers:
        in_memory: { memory: ~ }

        in_database:
            entity:
                class: App\Entity\Usertodo
                property: username

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

        main:
            anonymous: true
            lazy: true
            pattern: ^/
            provider: in_database

            remember_me:
                secret:   '%kernel.secret%'
                lifetime: 604800
                path:     /

            form_login:
                login_path: login
                check_path: login
                always_use_default_target_path:  true
                default_target_path:  /

            logout:
                path: logout
                target: login

    role_hierarchy:
        ROLE_ADMIN: ROLE_USER
        ROLE_SUPER_ADMIN: ROLE_ADMIN

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

Configuration de l'entité utilisateur

L'authentification dépend ensuite du paramétrage de l'entité utilisateur. Cette classe comporte nécessairement la mention « **implements UserInterface** ». Elle appelle plusieurs méthodes de sécurisation conformes aux règles d'authentification exigées par Symfony. Ces méthodes permettent, lors d'une connexion, de stocker en session les informations relatives à l'utilisateur et de définir ensuite ses droits d'accès à certaines pages ou à certaines actions.

- La méthode « **getRoles** » permet de déterminer le rôle attribué à l'utilisateur. Pour être reconnu, un rôle retourné doit contenir le préfixe « **ROLE_** ».
- La méthode « **getPassword** » permet récupérer le mot de passe codé de l'utilisateur.
- La méthode « **getSalt** » définit les informations sur le type d'encodage du mot de passe de l'utilisateur.
- La méthode « **getUsername** » renvoie le nom de l'utilisateur qui sert à son authentification.
- La méthode « **eraseCredentials** » spécifie certaines informations clés utilisées pour l'authentification de l'utilisateur.

src/Entity/Usertodo.php

```php
public function getRole(): ?string
{
    return $this->role;
}

public function setRole(string $role): self
{
    $this->role = $role;

    return $this;
}

public function getRoles()
{
    return [$this->getRole()];
}

public function eraseCredentials()
{
    return null;
}
```

Contrôleur et formulaire de connexion

- Le fichier SecurityController définit la route de connexion (« **/login** »), les messages d'erreurs en cas d'échec de connexion et il renvoie le template Twig qui comporte le formulaire de connexion. L'appel de la classe « **AuthenticationUtils** » permet de générer les messages des éventuelles erreurs rencontrées lors de tentatives de connexions ; lorsque, par exemple, les noms d'utilisateurs ou les mots de passe saisis ne sont pas conformes.

- Le formulaire de connexion du template Twig comporte deux champs que l'internaute doit renseigner pour parvenir à s'authentifier : « **_username** » et « **_password** ». A chaque nouvelle connexion, ces informations sont vérifiées par Symfony via les méthodes « getCredentials » et « checkCredentials ».

- Ce formulaire est protégé par la présence d'un **jeton CSRF** vérifié automatiquement par Symfony à chaque nouvelle connexion (via le service « **token_manager** »).

src/Controller/SecurityController.php

```php
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="login", methods={"GET","POST"})
     * @return Response
     */
    public function loginAction(Request $request, AuthenticationUtils
    $authenticationUtils): Response
    {
        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUsername = $authenticationUtils->getLastUsername();

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

templates/security/login.html.twig

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

    <input placeholder="NOM" type="text" id="username" name="_username"
    value="{{ last_username }}" required />

    <input placeholder="MOT DE PASSE" type="password" id="password"
    name="_password" required />
```

Où sont stockés les utilisateurs ?

- Les utilisateurs sont stockés en base de données : la classe « Usertoto » étant reliée au composant de mapping géré par Doctrine. Les différents attributs de l'entité utilisateur ($id, $username, $password, $email…) correspondent aux différentes colonnes générées en base de données par Doctrine lors du développement de l'application. Les annotations de ces attributs qui commencent par la mention « ***@ORM*** » définissent les types de champs créés en base de données, et d'autres contraintes telles que la présence d'index ou de relations avec d'autres tables. L'application enregistre et actualise dans les sessions ces informations de l'objet utilisateur en les sérialisant ou en les dé-sérialisant à chaque requête.

```php
class Usertodo implements UserInterface
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue()
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=25, unique=true)
     * @Assert\NotBlank(message="Vous devez saisir un nom d'utilisateur.")
     * @Assert\Length(
     *      min=4,
     *      max=50,
     *      minMessage="Nom trop court",
     *      maxMessage = "Nom trop long"
     * )
     */
    private $username;
```

2

Quels fichiers ont été modifiés et pourquoi ?

Modifications de l’entité utilisateur

- Dans le projet de départ, la méthode « **getRoles** » de l’entité utilisateur n’attribuait qu’un seul rôle : le « **ROLE_USER** ». L’absence de rôles hiérarchiques empêchait de contrôler les droits des utilisateurs pour l’accès à certaines pages ou pour la réalisation d’opérations (prévues dans le cahier des charges des améliorations de l’application) : créations, modifications ou suppressions de comptes utilisateurs réservées aux administrateurs et suppressions restreintes de tâches par auteurs pour les simples utilisateurs.

- Afin de pouvoir attribuer de nouveaux rôles, un attribut privé « **$role** » a été ajouté à l’entité utilisateur afin d’y définir trois rôles hiérarchiques différents : ROLE_SUPER_ADMIN, ROLE_ADMIN et ROLE_USER.

```php
public function getRoles()
{
    return [$this->getRole()];
}
```

Modifications du fichier security.yaml

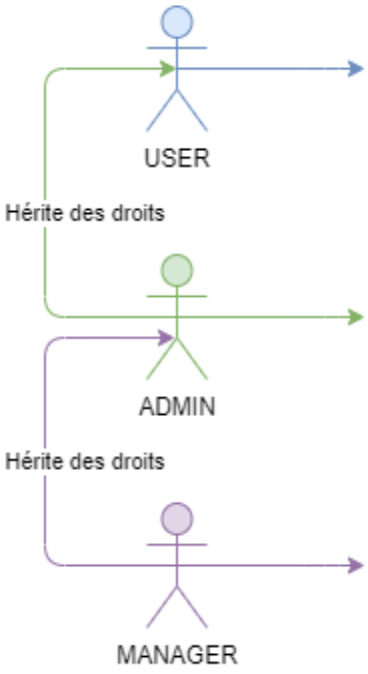
- Dans le projet initial, la partie dédiée à la gestion des droits d'accès au site (« **access_control** ») s'appuyait uniquement sur une authentification de type anonyme. Suite à la modification de l'entité utilisateur, il a été possible de limiter l'accès aux pages de gestion des utilisateurs aux utilisateurs définis avec le ROLE_ADMIN ou le ROLE_SUPER_ADMIN.

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

- Une section sur la hiérarchie des rôles a aussi été ajoutée afin de préciser que les administrateurs disposent de tous les droits d'accès des utilisateurs. Dans ce cas, les utilisateurs avec le ROLE_ADMIN héritent des droits accordés aux ROLE_USER. À un niveau supérieur, les managers disposant du ROLE_SUPER_ADMIN disposent des droits d'accès accordés aux administrateurs ou aux simples utilisateurs.

```
role_hierarchy:
    ROLE_ADMIN: ROLE_USER
    ROLE_SUPER_ADMIN: ROLE_ADMIN
```

Opérations autorisées par rôles



- Lire ou modifier des tâches
- Supprimer ses tâches
- Créer des tâches

- Créer des utilisateurs
- Modifier des utilisateurs
- Supprimer des utilisateurs

- Dispose de tous les droits d'accès
- Compte non modifiable par les autres utilisateurs
- Peut modifier le compte « anonyme »

- Les administrateurs ou managers ne peuvent pas modifier ou supprimer leurs propres comptes. Des vérifications ont été ajoutées avec l'envoi de messages d'erreurs dans le fichier « **UserVoter** » du dossier /src/Security.

```php
private function checkAuthorization(Usertodo $user)
{
    if (($this->security->getUser()->getId() === $user->getId()) ||
        (!$this->security->isGranted('ROLE_SUPER_ADMIN') && $user->getRole() === 'ROLE_ANONYMOUS') ||
        (!$this->security->isGranted('ROLE_SUPER_ADMIN') && $user->getRole() === 'ROLE_SUPER_ADMIN'))
    {
        return false;
```

Modification de la fonction de déconnexion

- Dans le projet de départ, la fonction « vide » de déconnexion « **logout** » présente dans le fichier SecurityController a été supprimée et remplacée dans le fichier « **routes.yaml** » situé dans le dossier /config. La méthode GET utilisée pour cette opération a également été spécifiée conformément aux bonnes pratiques de développement Symfony.

Modifications des templates

- Le contrôle des droits pour effectuer certaines opérations ou accéder à certaines informations a aussi été paramétré dans les templates Twig avec l'annotation « **is_granted( )** ».

```
{% if app.user and is_granted('ROLE_ADMIN') %}
```