Documentation Technique : Authentification dans l'Application Symfony

1 Introduction

L'authentification est une partie essentielle de toute application web moderne, car elle permet de vérifier l'identité des utilisateurs avant de leur accorder l'accès aux fonctionnalités ou aux données protégées. Dans une application Symfony, ce processus est bien pris en charge par les outils et les services de sécurité offerts par le framework. Ce document détaille la mise en place de l'authentification dans notre application Symfony, en couvrant les principaux composants : la classe `User`, le fichier de configuration `security.yaml`, le `SecurityController`, et enfin, le système de `Voters`.

2 Classe User

La classe `User` représente les utilisateurs de notre application. Chaque instance de cette classe correspond à un utilisateur réel avec ses informations d'identification, ses rôles, et d'autres attributs liés à la gestion de ses accès dans l'application.

```
/**
 * @ORM\Table("user")
 * @ORM\Entity
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 * @UniqueEntity(fields={"email"}, message="This value ( {{ value }} )
is already used.")
 * @UniqueEntity(fields={"username"}, message="This value ( {{ value }} )
is already used.")
 */
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    // Attributs et m thodes
}
```

2.1 Détails de la classe User

- **Implémentation des interfaces :** La classe `User` implémente deux interfaces importantes : `UserInterface` et `PasswordAuthenticatedUserInterface`. La première assure que la classe dispose des méthodes requises pour la gestion des utilisateurs dans Symfony, tandis que la seconde gère l'authentification par mot de passe.
- **Annotations Doctrine :** La classe utilise des annotations Doctrine pour la gestion de la persistance. Cela signifie que chaque instance de `User` est mappée à une table dans la base de données.
- **Contraintes de validation :** Les annotations `@UniqueEntity` garantissent l'unicité de certains champs comme l'email et le nom d'utilisateur, empêchant ainsi les doublons.
- **Gestion des rôles :** Un attribut `roles` permet de définir les différents rôles que peut avoir un utilisateur (par exemple, `ROLE_USER`, `ROLE_ADMIN`).
- **Méthodes importantes :**
  - `getRoles()` : Cette méthode garantit que chaque utilisateur a au moins le rôle de base `ROLE_USER`, même si aucun autre rôle n'est défini.
  - `getUserIdentifier()` : Elle retourne l'email comme identifiant unique, car nous avons décidé que l'email servirait de base pour l'authentification.

3 Fichier de Configuration `security.yaml`

Le fichier `security.yaml` est le cœur de la configuration de sécurité dans Symfony. Il permet de définir les paramètres relatifs à l'authentification, aux autorisations, et à la gestion des rôles.

```
security:
    enable_authenticator_manager: true
    password_hashers:
        App\Entity\User:
            algorithm: 'auto'
    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
    firewalls:
        main:
            lazy: true
            form_login:
                login_path: login
```

```
            check_path: login
            enable_csrf: true
        logout:
            path: logout
role_hierarchy:
    ROLE_ADMIN: ROLE_USER
access_control:
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/, roles: IS_AUTHENTICATED_FULLY }
    - { path: ^/users, roles: ROLE_ADMIN }
```

3.1 Détails de la configuration

- **Gestion des mots de passe :** Nous utilisons un algorithme de hachage automatique (`'auto'`) pour les mots de passe des utilisateurs, garantissant ainsi un niveau élevé de sécurité.
- **Providers :** Le `provider` d'utilisateur est défini pour se baser sur l'entité `User`, ce qui permet à Symfony de récupérer les informations d'identification de la base de données.
- **Firewalls :** La configuration du `firewall` principal inclut un formulaire de connexion, avec un chemin défini pour la connexion (`login`) et pour la déconnexion (`logout`), et une protection CSRF est activée pour sécuriser les requêtes.
- **Hiérarchie des rôles :** La hiérarchie des rôles permet de simplifier la gestion des autorisations. Par exemple, un utilisateur avec `ROLE_ADMIN` hérite automatiquement des permissions associées à `ROLE_USER`.
- **Contrôle d'accès :** Des règles d'accès sont définies pour contrôler les pages accessibles en fonction des rôles de l'utilisateur. Par exemple, seules les personnes anonymes peuvent accéder à la page de connexion, tandis que l'accès à certaines parties de l'application est restreint aux utilisateurs entièrement authentifiés.

4 SecurityController

Le `SecurityController` est chargé de gérer les actions liées à l'authentification, comme l'affichage du formulaire de connexion et la gestion de la déconnexion.

```
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="login")
     */
    public function loginAction(AuthenticationUtils $authenticationUtils)
```

```
    {
        // R cup re les ventuelles erreurs d'authentification
        $error = $authenticationUtils->getLastAuthenticationError();
        // Dernier nom d'utilisateur entr
        $lastUsername = $authenticationUtils->getLastUsername();

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

    /**
     * @Route("/logout", name="logout")
     */
    public function logoutCheck()
    {
        // Cette m thode peut rester vide car Symfony g re automatiquement la
    }
}
```

4.1 Fonctionnalités du `SecurityController`

- **Connexion :** La méthode `loginAction()` récupère les éventuelles erreurs d'authentification et affiche le dernier nom d'utilisateur saisi. Ces informations sont ensuite passées au formulaire de connexion pour être affichées à l'utilisateur en cas d'échec.
- **Déconnexion :** Symfony gère automatiquement la déconnexion, il n'est donc pas nécessaire de mettre en œuvre une logique spécifique dans la méthode `logoutCheck()`.

5 UserVoter

Les `Voters` permettent d'implémenter des règles d'autorisation spécifiques. Le `UserVoter` est responsable de gérer les autorisations pour certaines actions, telles que la modification ou la suppression d'utilisateurs.

```
class UserVoter extends Voter
{
    const USER_EDIT = 'user_edit';
    const USER_DELETE = 'user_delete';
    const USER_VIEW = 'user_view';

    // Impl mentation des m thodes pour voter
}
```

5.1 Détails des actions autorisées

- **Actions définies :** Le `UserVoter` gère les actions de visualisation, d'édition, et de suppression d'un utilisateur, grâce à des constantes comme `USER_EDIT`, `USER_DELETE`, et `USER_VIEW`.
- **Autorisation en fonction des rôles :** Seuls les administrateurs peuvent supprimer un utilisateur, mais chaque utilisateur peut voir ou modifier son propre profil.

6 Conclusion

La mise en place de l'authentification dans notre application Symfony permet une gestion efficace et sécurisée des utilisateurs. Grâce à une configuration bien pensée dans `security.yaml`, une implémentation solide de la classe `User`, et l'utilisation des `Voters` pour des contrôles fins d'autorisation, notre application assure un haut niveau de sécurité tout en restant flexible pour s'adapter à différents scénarios.