Documentation technique

I. Implémentation de l'Authentification dans Symfony

1. Introduction

Cette documentation a pour but d'expliquer le fonctionnement de l'authentification dans l'application. Elle précise également quels fichiers modifier en cas de changement nécessaire et pourquoi. L'objectif est de comprendre le processus d'authentification ainsi que la manière dont les utilisateurs sont stockés et gérés.

2. Fichiers Clés de l'Authentification

Le fichier `security.yaml` est crucial pour configurer l'authentification de l'application. Il se trouve à l'emplacement suivant : `./config/packages/security.yaml`.

L'entité **User**, définie dans `./src/Entity/User.php`, contient plusieurs informations :

- **ID** : un entier auto-incrémenté non nul ;
- **email** : une chaîne de caractères non nulle et unique ;
- **role** : un tableau JSON non nul ;
- **password** : une chaîne de caractères non nulle, stockée sous forme hachée pour des raisons de sécurité ;
- **username** : une chaîne de caractères non nulle et unique.

Cette entité permet de stocker les informations des utilisateurs en base de données.

Les méthodes pour la connexion et la déconnexion sont implémentées dans le fichier `./src/Controller/SecurityController.php`.

3. Processus d'Authentification

Dans Symfony 6.4, le processus d'authentification repose sur le système des "Passeports" et des "Authenticators". Lorsqu'un utilisateur non authentifié tente d'accéder à une ressource protégée, il est redirigé vers la page de connexion. Si la requête est valide, l'authenticator crée un "Passport" contenant les identifiants de l'utilisateur. Ces informations sont vérifiées avant de générer un

token de connexion, assurant ainsi une gestion sécurisée et adaptable des utilisateurs.

4. Ajout ou Modification des Permissions

Pour gérer les rôles et les permissions, il est nécessaire de modifier le fichier `./config/packages/security.yaml`. Les modifications se feront sous l'option **'access_control'**, qui permet de gérer l'accès aux différentes routes en fonction des rôles attribués aux utilisateurs. Cette configuration peut également varier selon le contexte de l'utilisateur, par exemple, certaines routes peuvent être accessibles uniquement aux utilisateurs connectés ou ayant des rôles spécifiques.

II. Guide de Collaboration pour le Développement

1. Introduction

Ce guide a pour objectif de fournir des directives claires pour le développement collaboratif au sein de l'équipe. Il souligne l'importance de suivre des bonnes pratiques pour assurer une collaboration efficace, maintenir la qualité du code et faciliter l'intégration des contributions de chaque membre.

2. Mise en Place de l'Environnement de Développement

- **2.1. Cloner le projet**
  - Utilisez les commandes Git suivantes pour récupérer le projet :

```
git clone <url-du-repository>
cd <nom-du-projet>
```

- **2.2. Installation des dépendances**
  - Installez les dépendances du projet en utilisant Composer :

```
composer install
```

  - Configurez l'environnement en créant un fichier `.env.local` basé sur `.env` et en ajoutant les variables nécessaires, ainsi qu'un fichier `.env.test.local` avec les informations de la base de données de test

- **2.3. Configuration de la base de données**

- Créez la base de données en utilisant la commande suivante :

```
php bin/console doctrine:database:create
```

- Créez les tables :

```
php bin/console doctrine:schema:update
```

- Charger les fixtures

```
php bin/console doctrine:fixtures:load
```

- **2.4 Configuration de la base de données de test**
  - Créez la base de données en utilisant la commande suivante :

```
php bin/console doctrine:schema:create --env=test
```

  - Créez les tables :

```
php bin/console doctrine:schema:update --env=test
```

  - Charger les fixtures

```
php bin/console doctrine:fixtures:load --env=test
```

- **2.5. Lancement du serveur de développement**
  - Démarrez le serveur de développement Symfony avec la commande :

```
symfony server:start
```

  - Vous pouvez aussi utiliser :

```
php bin/console server:run
```

3. Processus de Collaboration

- **3.1. Utilisation de Git**

- Suivez un modèle de branching comme Gitflow pour structurer les branches de développement :
  - `main` : Branche principale contenant le code en production.
  - `develop` : Branche de développement pour les nouvelles fonctionnalités.
  - `feature/<nom>` : Branches pour les nouvelles fonctionnalités.
  - `hotfix/<nom>` : Branches pour les corrections urgentes.

- **3.2. Revue de Code**
  - Utilisez le processus de pull request (PR) pour intégrer les modifications :
    - Créez une PR depuis votre branche de fonctionnalité vers `develop` ou `main`.
    - Assurez-vous que la PR passe toutes les vérifications automatisées et demandez une revue de code à un ou plusieurs membres de l'équipe.

4. Règles de Qualité du Code

- **4.1. Standards de Codage**
  - Adoptez les conventions de codage recommandées, telles que les PSR (PHP Standard Recommendations) et les conventions de nommage propres au projet. Veillez à maintenir la lisibilité et la cohérence du code.

- **4.2. Tests**
  - Écrivez des tests pour valider le comportement du code :
    - **Tests unitaires** : Vérifient le fonctionnement des petites unités de code.
    - **Tests fonctionnels** : Vérifient le comportement de l'application dans son ensemble.
  - Utilisez PHPUnit pour exécuter les tests :

```
./vendor/bin/phpunit
```

- Utilisez cette commande pour générer le code coverage de l’application :

```
./vendor/bin/phpunit --coverage-html public/test-cove
rage
```