Documentation technique

Système d'authentification

Auteur : **Pavel KLIMOVICH**

Table des matières

1. Introduction

L'application a été créée en utilisant le framework Symfony, en utilisant la version 5.4(LTS). Pour assurer le bon fonctionnement de l'application, il est nécessaire d'avoir une version de PHP égale ou supérieure à 8.1.

Cette documentation vous explique comment l'authentification a été implémenté sur l'application.

Vous trouverez plus d'informations concernant l'Authentification dans la documentation officielle de Symfony.

2. Package « Security » (Authentification).

1. La liste des fichiers.

Type de fichier	Chemin du fichier	Description
Configuration	*config/packages/security.yaml*	Configuration du composant « Security » qui se charge du processus d'authentification
Entité	*src/Entity/User.php*	Entité utilisateur
Contrôleur	*src/Controller/SecurityController.php*	Contrôleur qui se charge de la connexion / déconnexion de l'utilisateur
Authentification	*src/Security/LoginFormAuthenticator. php*	Méthodes du processus d'authentification de l'application
Vue	*templates/security/ login.html.twig*	Template de la page de connexion

2. L'entité User

Avant toute chose, il est essentiel de définir une entité qui représentera l'utilisateur connecté.

La première étape dans la création d'une authentification consiste à construire l'entité User. Dans Symfony, l'entité User définit la structure de notre utilisateur (nom d'utilisateur, mot de passe, rôle, etc.).

À partir de cette entité, nous construirons le contrôleur et le formulaire de connexion.

Cependant, l'entité User n'est pas une entité ordinaire, elle doit implémenter une interface : UserInterface. Cette interface permet d'inclure certaines méthodes utilisées par le composant "Sécurité" de Symfony pour mettre en place le pare-feu et le système d'authentification. Voici les méthodes qu'il est obligatoire de mettre en place :

getUsername(): Cette méthode retourne le nom d'utilisateur de l'utilisateur.

getRoles(): Cette méthode retourne un tableau des rôles attribués à l'utilisateur.

getPassword(): Cette méthode retourne le mot de passe haché de l'utilisateur.

En implémentant ces méthodes dans notre entité User, nous permettons au composant de sécurité de Symfony de gérer efficacement l'authentification et la gestion des utilisateurs.

3. Configuration de composant Security.

La gestion de la sécurité et donc de nos utilisateurs s'effectue avec le composant « Security » de Symfony via le fichier **security.yaml** qui contient plusieurs paramètres importants pour le bon fonctionnement de l'application.

1/ Système d'encryptage de mot de passe

Le mot de passe est crypté dans la base de données afin d'assurer une sécurité maximale.

Le cryptage est actuellement configuré en mode automatique, ce qui est le meilleur choix pour la version actuelle de Symfony utilisée sur le site. Cependant, il est possible de modifier ce mode dans le fichier de configuration **security.yaml**, dans la section « **password_hashers** ».

2/ User Provider

Le fournisseur (provider) indique où trouver les informations nécessaires à l'authentification. Dans ce cas, la classe User est utilisée, avec la propriété "username" (nom d'utilisateur) pour la connexion, suivie du mot de passe.

Le fournisseur d'utilisateurs (User Provider) charge (ou recharge) les utilisateurs à partir d'un stockage, tel qu'une base de données, en se basant sur un "identifiant d'utilisateur" (dans ce cas, c'est le username qui est utilisé comme identifiant).

Les utilisateurs sont stockés dans une base de données, et le fournisseur d'utilisateurs utilise Doctrine pour les récupérer.

3/ Le firewall

Le fichier de configuration permet aussi de configurer les firewalls, qui sont des systèmes d'authentification déterminant comment les utilisateurs peuvent s'authentifier.

Il est possible de configurer plusieurs firewalls pour une même application, mais seul un firewall actif est utilisé à chaque requête. Symfony utilise le paramètre "**pattern**" pour déterminer quel firewall doit être utilisé pour une requête donnée. Si aucun attribut "pattern" n'est déclaré, le firewall sera utilisé pour toutes les URLs.

Le mode "**lazy**" anonyme permet de retarder le démarrage de la session tant qu'aucune autorisation n'est nécessaire (c'est-à-dire qu'il n'y a pas de vérification explicite d'un privilège utilisateur). Ceci est important pour améliorer la mise en cache des requêtes.

La troisième clé de configuration dans la section principale concerne le fournisseur d'authentification. Elle permet de configurer la route utilisée pour la connexion. Différents types de fournisseurs d'authentification sont disponibles, que vous pouvez consulter pour choisir celui qui convient le mieux à vos besoins.

La dernière clé de configuration dans la section principale est la clé "**logout**", qui définit les règles de déconnexion associées au pare-feu de Symfony. Elle permet de configurer comment les utilisateurs doivent être déconnectés et quelles actions doivent être effectuées lors de la déconnexion.

Pour le système de déconnexion de l'utilisateur, nous définissons une route associée en utilisant la clé "**path**" dans la configuration. De plus, nous devons configurer la clé "**target**" qui permettra de rediriger l'utilisateur déconnecté vers une route spécifique. Dans notre cas, nous avons choisi de rediriger vers la page d'accueil.

4/ Autorisation.

La sécurité d'accès aux ressources est gérée par le système « **access_control** ». Il comporte deux aspects : le premier aspect indique les rôles nécessaires pour accéder à une ressource, tandis que le deuxième aspect détermine les rôles possédés par l'utilisateur authentifié, déterminant ainsi les ressources auxquelles il peut accéder.

Trois types d'utilisateurs sont distingués :

Utilisateur non authentifié : Lorsqu'un utilisateur n'est pas authentifié, il est considéré comme anonyme et sera automatiquement redirigé vers la page d'authentification s'il tente d'accéder à l'application.

Utilisateur normal : Cet utilisateur possède le rôle d'utilisateur simple (ROLE_USER). Il est autorisé à créer ou modifier ses propres tâches et peut également consulter les différentes listes de tâches.

Administrateur : L'administrateur a le rôle le plus élevé (ROLE_ADMIN) et dispose de droits étendus. Il peut créer et modifier des utilisateurs, ainsi que gérer leurs droits d'accès à l'application. De plus, il peut créer, modifier et supprimer ses propres tâches ainsi que celles créées par des utilisateurs anonymes.

Chaque rôle doit contenir le préfix ROLE_. Exemple ROLE_USER ou ROLE_ADMIN.

La clé de configuration **"access_control"** permet d'autoriser l'accès à certaines pages en fonction du rôle de l'utilisateur connecté (**ROLE_USER**, **ROLE_ADMIN**, etc.). Pour ce faire, il est nécessaire de renseigner les éléments suivants :

"**path**" : spécifie la route à laquelle l'utilisateur peut accéder.

"**roles**" : représente le niveau d'authentification requis pour accéder à cette route.

3. Stockage des utilisateurs

Les utilisateurs sont persistés en base de données à l'aide de l'ORM Doctrine. La gestion de l'ORM Doctrine n'est pas le sujet de cette documentation, il est recommandé de se référer à la documentation officielle pour obtenir plus d'informations à ce sujet.

Dans la base de données, les utilisateurs sont stockés dans la table "**user**". Les champs **username** et **email** sont des champs uniques. L'accès aux données se fait via l'instanciation d'un objet de la classe « *src/Entity/User.php* », grâce à l'utilisation de l'ORM Doctrine et de l'injection de dépendance en important le repository de la classe User.