ToDo & Co - Authentification

Introduction

L'application est basée sur le *Framework* Symfony, et utilise le composant Security pour la gestion de l'authentification.

Cette documentation détaille comment fonctionne le processus d'authentification dans l'application ToDo & Co, où se situent les fichiers de configuration et quels fichiers modifier.

1. Authentification

Dans Symfony, le processus d'authentification nécessite d'avoir une classe *User*, peu importe la façon dont les utilisateurs se connectent à l'application, ou comment les données utilisateurs sont stockées.

1.1. La classe User

Dans Symfony, un utilisateur est représenté par une entité *User*. Cette entité contient les propriétés nécessaires à l'identification de l'utilisateur (nom d'utilisateur, email, mot de passe, etc.).

- Cette classe doit implémenter `UserInterface`.
- La classe *User* est une entité *Doctrine*, et toutes les données des utilisateurs sont stockées dans la base de données du projet, dans la table `user`.
- Il est nécessaire de choisir un attribut de la classe *User*, qui permet d'identifier de façon unique chaque utilisateur. Pour ce projet, il s'agit de l'**email**. Il est possible de modifier cet attribut dans le fichier `config/packages/security.yaml`, dans la clé `providers`.
- Pour vérifier l'unicité de chaque utilisateur (par exemple lors de la validation des formulaires de création/modification de l'utilisateur), l'annotation `@UniqueEntity("email")` doit être présente dans la classe *User*. Avec Symfony 5.3, une nouvelle méthode `getUserIdentifier()` permet de récupérer l'attribut identifiant de façon unique l'utilisateur. Dans ce projet, cette méthode retourne l'**email** de l'utilisateur.
- Dans un souci de sécurité, le mot de passe de l'utilisateur doit être *hashé* avant d'être stocké en base de données. Pour activer le *hashage* des mots de passe, un algorithme de *hashage* doit être configuré dans la clé `password_hashers` du fichier `security.yaml`. Par défaut, l'option `auto` est définie, Symfony utilise le meilleur algorithme disponible (dans Symfony 5.3, il s'agit de *bcrypt*).
- Lorsqu'il y a besoin de *hasher* les mots de passe dans le code, par exemple lors de la création d'un nouvel utilisateur, il suffit d'utiliser `UserPasswordHasherInterface` en tant que service, en utilisant l'injection de dépendances :

```
$hashedPassword = $passwordHasher->hashPassword($user, $rawPassword);
```

1.2. Firewall

L'authentification des utilisateurs dans Symfony est gérée par un *firewall*. La configuration de celui-ci permet de modifier la méthode d'authentification de l'utilisateur (formulaire, token API, etc.), et de restreindre l'accès de l'utilisateur à certaines parties de l'application.

La configuration du *firewall* est disponible dans le fichier `security.yaml`, dans la clé `firewalls`.

Dans ce projet, le *firewall* `main` spécifie que toutes les URLs du site nécessitent une authentification (clé `pattern`) via un formulaire de connexion (clé `entry_point`), c'est-à-dire que tous les utilisateurs non authentifiés seront redirigés vers la page de connexion, pour se connecter et ainsi pouvoir accéder au site.

Le `SecurityController` est chargé de rendre le template de la page de connexion, qui contient le formulaire de connexion, ce contrôleur permet également de récupérer les erreurs liées à la connexion (dans le cas où l'utilisateur renseigne des informations de connexion invalides).

Enfin, lorsque le formulaire de connexion est soumis, les *Authenticators* de Symfony assurent l'authentification, et aucune autre action n'est requise pour authentifier les utilisateurs.

2. Authorisation

Un processus d'autorisation est utilisé dans le but de restreindre l'accès à certaines parties du site, par exemple la section admin.

Ce processus se base sur les `roles` de l'utilisateur. Dans la classe *User*, la méthode `getRoles()` retourne les rôles de l'utilisateur, cette information étant stockée en base de données. Par défaut, si l'utilisateur n'a pas de rôle, un `ROLE_USER` lui est automatiquement attribué.

Pour répondre aux besoins de ce projet, un `ROLE_ADMIN` a été ajouté. Les utilisateurs possédant ce rôle ont accès à toutes les URLs commençant par `/users` (création d'un utilisateur, modification et suppression).

Si besoin, il est possible d'ajouter des nouveaux rôles lors de la création/modification des utilisateurs, le seul impératif est que le rôle **doit** commencer par `ROLE_`. Il est ensuite possible de compléter avec n'importe quelle chaîne de caractères (par exemple `ROLE_TASK_UPDATE`).

Pour restreindre l'accès à certaines URLs du site, il y a deux façons possibles :

- la clé `access_control` dans le fichier `security.yaml`
- dans les `Controllers` ou dans un template *Twig*.

Pour ce projet, les deux options ont été utilisées, en fonction des besoins.

La clé `access_control` dans le fichier `security.yaml` est utilisée pour accorder l'accès à toutes les URLs commençant par `/users` à tous les utilisateurs ayant le rôle `ROLE_ADMIN`, en utilisant l'option `- { path: ^/users, roles: ROLE_ADMIN }`.

Dans le code, il est possible d'utiliser `$this->denyAccessUnlessGranted(<role>)` dans un contrôleur, pour restreindre l'accès pour une route en particulier.

Ou, dans un template *Twig*, il est possible d'utiliser :

```
{% if is_granted('ROLE_ADMIN') %}
  <a href="...">Supprimer</a>
{% endif %}
```

pour afficher un lien seulement si l'utilisateur possède le rôle `ROLE_ADMIN`.

Peu importe la méthode utilisée, si l'utilisateur ne possède pas les droits d'accès suffisants, une erreur HTTP 403 sera renvoyée.

En savoir plus

- La documentation Security
- Le composant Security
- La section Authentification du composant Security

Processus d'authentification