Documentation technique

Système d’authentification

Table des matières

I. Projet....2
1) Introduction....2
2) Librairies....2
3) Initialisation et contribution....3
II. Authentification....3
1) Fichiers....3
2) L'entité User....4
3) Configurer le composant Security....5
3.1) Système d'encryptage de mot de passe....5
3.2) User Provider....6
3.3) Le firewall....6
3.4) Autorisation....8
3.4.1) La sécurité access_control....8
3.4.2) Mise en place des Voters....9
III. Stockage des utilisateurs....11
1) Base de données....11

I. Projet

1) Introduction

L'application a été développée avec le framework Symfony, en version **5.4**.

Une version de PHP **7.2** ou plus élevée est nécessaire pour le bon fonctionnement de l'application.

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

Elle va vous montrer quels fichiers il faut modifier et pourquoi. Elle vous expliquera aussi comment s'opère l'authentification au travers du composant « Security » de Symfony et ou seront stocker les utilisareurs.

Vous trouverez plus d'informations concernant ce fichier et ses différentes parties dans la documentation officielle de Symfony.

2) Librairies

L'ensemble des librairies installées via Composer sont consultables dans le fichier « composer.json » à la racine du projet. Ces librairies sont également visibles sur le site packagist.org.

Il est recommander de mettre à jour régulièrement ces librairies et de consulter leurs documentations afin d'assurer la sécurité et le bon fonctionnement de l'application.

3) Initialisation et contribution

Afin d'installer ToDoList, il vous sera nécessaire de consulter le repository de celle-ci et de suivre les étapes indiquer dans le fichier « README.md » à la racine de celle-ci. Pour contribuer à l'amélioration de l'application, il vous faudra consulter le fichier « CONTRIBUTING.md » situé également à la racine du projet.

Projet : https://github.com/mdoutreluingne/todoandco

Readme :
https://github.com/mdoutreluingne/todoandco/blob/master/README.md

Contributing :
https://github.com/mdoutreluingne/todoandco/blob/master/CONTRIBUTING.md

II. Authentification

1) Fichiers

Type	Chemin du fichier	Description
Configuration	config/packages/ security.yaml	Configuration du composant « Security » qui ce charge du processus d'authentification
Entité	src/Entity/User.php	Entité utilisateur
Contrôleur	src/Controller/ SecurityController.php	Contrôleur qui ce 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 du formulaire de connexion

2) L’entité User

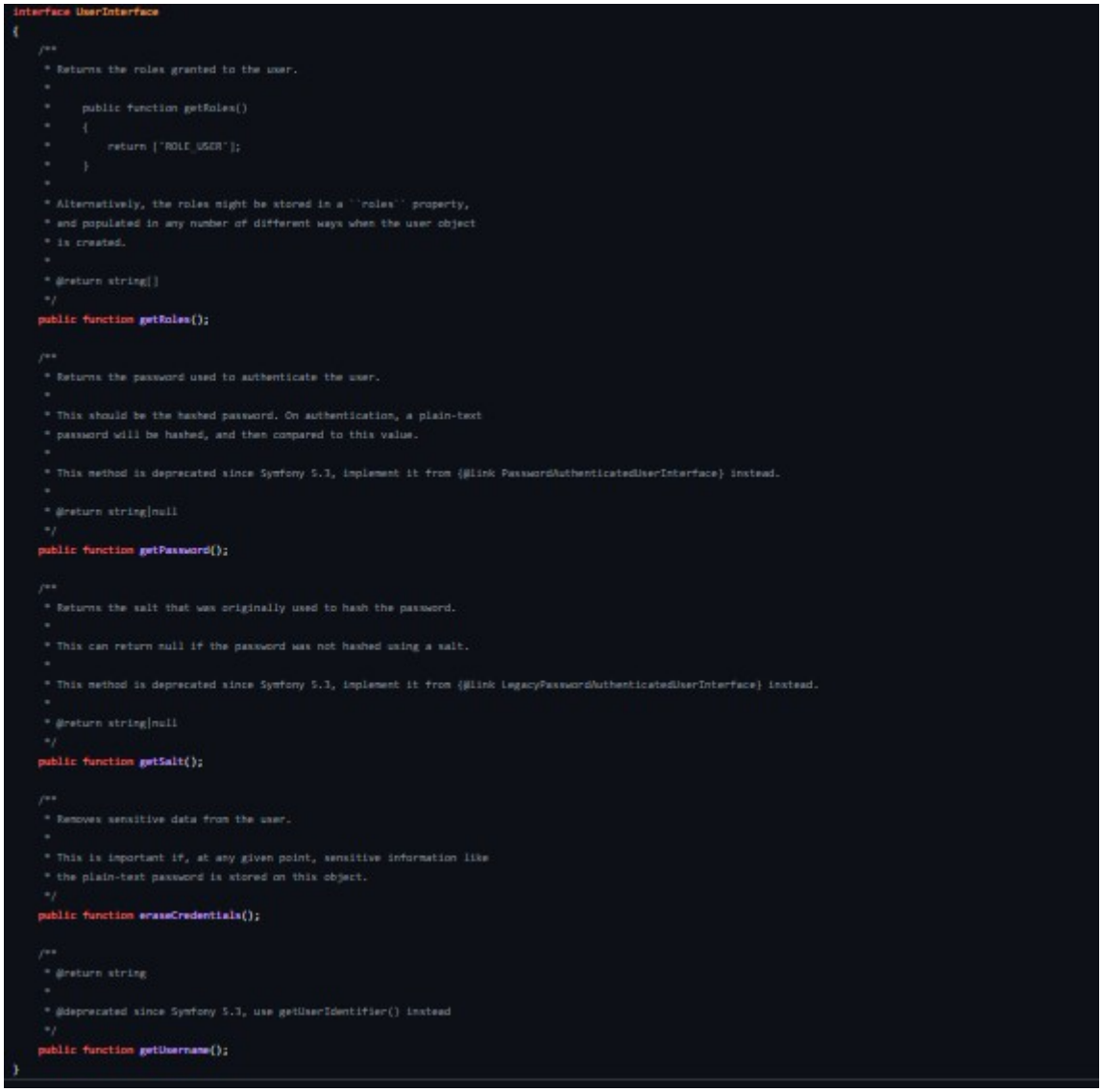
Avant toute de chose, il est nécessaire d'avoir défini une entité qui représentera l'utilisateur connecté.

La construction de l'entité User est la première étape à prendre en compte dans la création d'une authentification. L'entité User de Symfony définit la structure de notre utilisateur (username, password, role etc...).

C'est à partir de cette entité que l'on construira le controller ainsi que le formulaire de connexion.

Néanmoins, l'entité **User** n'est pas une entité comme une autre, elle doit implémenter une interface : la UserInterface qui permet d'inclure certaines méthodes que le composant « Security » de Symfony utilise pour mettre en place le pare feu et le système d'authentification. Voici les méthodes à mettre en place obligatoirement :

Code source du fichier :
https://github.com/symfony/symfony/blob/5.4/src/Symfony/Component/Security/Core/User/UserInterface.php



```php
interface UserInterface
{
    /**
     * Returns the roles granted to the user.
     *
     *     public function getRoles()
     *     {
     *         return ['ROLE_USER'];
     *     }
     *
     * Alternatively, the roles might be stored in a ``roles`` property,
     * and populated in any number of different ways when the user object
     * is created.
     *
     * @return string[]
     */
    public function getRoles();

    /**
     * Returns the password used to authenticate the user.
     *
     * This should be the hashed password. On authentication, a plain-text
     * password will be hashed, and then compared to this value.
     *
     * This method is deprecated since Symfony 5.3, implement it from {@link PasswordAuthenticatedUserInterface} instead.
     *
     * @return string|null
     */
    public function getPassword();

    /**
     * Returns the salt that was originally used to hash the password.
     *
     * This can return null if the password was not hashed using a salt.
     *
     * This method is deprecated since Symfony 5.3, implement it from {@link LegacyPasswordAuthenticatedUserInterface} instead.
     *
     * @return string|null
     */
    public function getSalt();

    /**
     * Removes sensitive data from the user.
     *
     * This is important if, at any given point, sensitive information like
     * the plain-text password is stored on this object.
     */
    public function eraseCredentials();

    /**
     * @return string
     *
     * @deprecated since Symfony 5.3, use getUserIdentifier() instead
     */
    public function getUsername();
}
```

3) Configurer le 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.

3.1) Système d'encryptage de mot de passe

Le mot de passe est encrypté dans la basse de donnée, pour plus de sécurité. Comme ci-dessous :

←T→	id	username	password	email	roles (DC2Type:json)
Éditer Copier Supprimer	10	admin	$2y$13$rhuWLvbFsVfqoeT9r0WJd.8evDCK8cJw1L7v9xVno6a...	admin@todoandco.com	["ROLE_ADMIN"]
Éditer Copier Supprimer	11	mdoutreluingne	$2y$13$D4FJM.rzQjiQe5WnPdyN4.PiAyI5nJCLG2PmMNbwFup...	maxime@todoandco.com	["ROLE_USER"]
Éditer Copier Supprimer	12	anonyme	$2y$13$azPkpNL6UYd0VoqxbP.1e.r5qb7Rhl5tJSjsO4B/GF0...	anonyme@todoandco.com	["ROLE_USER"]

L’encryptage est en mode automatique pour le moment, ce qui correspond au meilleur choix pour la version de Symfony du site. Il est bien entendu possible de changer le mode dans le ficher de configuration « security.yaml » dans le bloc « password_hashers »

```
password_hashers:
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
```

Plus de détails sur les types d’encryptage : https://symfony.com/doc/5.4/security.html#registering-the-user-hashing-passwords

3.2) User Provider

Le provider indique où aller chercher les informations nécessaires à l’authentification :

```
    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username
```

Ici c’est la classe User que l’on utilise avec comme propriété le username (nom d’utilisateur) pour la connexion suivit du mot de passe.

Le fournisseur d'utilisateur (**User Provider**) (re)chargent les utilisateurs à partir d'un stockage (par exemple une base de données) sur la base d'un "identifiant d'utilisateur" (ici c’est le username qu’on utilise).

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

3.3) Le firewall

Le système de sécurité de Symfony est configuré dans le fichier **config/packages/security.yaml.**

```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        provider: app_user_provider
        custom_authenticator: App\Security\UserAuthenticator
        logout:
            path: logout
            target: homepage

        # activate different ways to authenticate
        # https://symfony.com/doc/current/security.html#the-firewall

        # https://symfony.com/doc/current/security/impersonating_user.html
        # switch_user: true
```

Un **Firewall** est un système d'authentification. Il définit la manière dont les utilisateurs pourront s'authentifier.

Il est possible de configurer plusieurs firewalls pour une même application. Par contre il n'y a qu'un seul firewall actif à chaque requête. Symfony utilise le paramètre **pattern** pour déterminer quel firewall doit être utilisé pour la requête. Si aucun attribut **pattern** n'est déclaré dans la configuration, le firewall sera utilisé pour toutes les urls.

Le **lazy** mode anonyme empêche le démarrage de la session s'il n'y a pas besoin d'autorisation (c'est-à-dire vérification explicite d'un privilège utilisateur). Ceci est important pour garder les requêtes en cache.

La 3ème clé de configuration de la partie **main** est le fournisseur d'authentification. Il permet de configurer la route utilisé pour se connecter. Il existe plusieurs types de fournisseur d'authentification que vous pouvez consulter ici.

Le dernière clé de configuration de la partie **main** est la clé **logout** qui définit les règles de déconnexion lié au pare-feu Symfony.

On définit, comme le système de connexion, une route associée à la déconnexion de l'utilisateur, via la clé **path**. On doit aussi configuré la clé **target** qui va permettre de rediriger l'utilisateur déconnecté vers une route spécifique. Nous avons choisi de rediriger vers la page d'accueil.

3.4) Autorisation

3.4.1) La sécurité access_control

Le système d'autorisation d'accès aux ressources a deux aspects. Le premier indique quel rôle on doit avoir pour accéder à une ressource, et l'autre indique quels rôles l'utilisateur authentifié possède et donc à quelles ressources il a accès.

On distinguera trois type d'utilisateurs qui sont les suivant :

• **Utilisateur non authentifier** : Lorsqu'un utilisateur n'est pas authentifié il sera considéré comme **anonyme** et se verra automatiquement redirigé vers la page d'authentification s'il souhaite parcourir l'application.

• **Utilisateur normal** : Celui-ci dispose du rôle utilisateur simple (ROLE_USER) et ne pourra que créer ou modifier ses propres tâches. Il pourra également consulter les différentes listes de tâches.

• **Administrateur** : Cet utilisateur se voit attribué les droits les plus importants de par son rôle (ROLE_ADMIN). Il pourra créer et modifier un utilisateur, mais aussi modifier ses droits d'accès à l'application. Créer, modifier et supprimer ses propres tâches ou celles ayant un auteur anonyme.

Le système de sécurité de Symfony est configuré dans le fichier **config/packages/security.yaml.**

Chaque rôle doit contenir le préfix **ROLE_**. Exemple **ROLE_USER** ou **ROLE_ADMIN**. Il est possible de paramétrer une hiérarchie des rôles dans le fichier de configuration.

```yaml
security:
    role_hierarchy:
        ROLE_ADMIN: ROLE_USER
```

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...). Pour cela, il faut renseigner :

- **path :** définie la route accessible par l'utilisateur.
- **roles** : représente le niveau d'authentification nécessaire pour accéder à cette route.

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

On notera qu'un utilisateur non identifier n'aura accès que à la page de connexion (login) avant de se connecter.

Documentation : https://symfony.com/doc/5.4/security.html#access-control-authorization

3.4.2) Mise en place des Voters

Les **Voters** sont des classes qui étendent la classe Voter. Elles permettent d'avoir un contrôle très précis sur l'accès à une ressource. Les Voters sont le moyen le plus puissant de Symfony pour gérer les autorisations. Ils vous permettent de centraliser toute la logique d'autorisation, puis de les réutiliser à de nombreux endroits.

Un Voter permet d'autoriser un utilisateur à effectuer une action particulière (ajouter une tâche, voir la liste des utilisateurs etc...), de manière beaucoup plus "flexible" qu'avec la configuration **access_control** (comme vue ci-dessus).

Par exemple, dans notre application, nous avons mis en place un système de **Voter** pour gerer l'accès aux différentes tâches des utilisateurs. Nous avons donc crée une classe : App\Security\Voter\TaskVoter.

Pour mettre en place un Voter Symfony, il faut respecter une certaine configuration :

- Mettre en place des actions sous la forme de constante PHP. Par ex, dans notre cas, nous voulons créer 2 accès différents sur notre application :

```php
class TaskVoter extends Voter
{
    const EDIT = 'TASK_EDIT';
    const DELETE = 'TASK_DELETE';
```

Ensuite, deux méthodes sont obligatoires dans l'utilisation d'un Voter Symfony :

- supports() : Cette méthode vérifie que le "sujet" passé en paramètre est bien une instance de la classe App\Entity\Task. De plus, elle vérifie que l'attribut passé en paramètre correspond bien à une constante de la classe de notre classe Voter (dans notre cas : EDIT, DELETE).
- voteOnAttribute() : en fonction de l'attribut utilisé, on décidera de retourner une classe en particulier qui vérifiera que l'utilisateur a bien les accès demandés. Par exemple, si l'attribut est EDIT, on appellera une méthode canEdit() (qui peut s'appeller comme vous le voulez) qui vérifie les droits pour modifier cette tâche.

```php
private function canEdit($subject, User $user): bool
{
    return $user === $subject->getUser();
}
```

Maintenant que nous avons notre Voter Symfony configuré, il faut l'utiliser dans notre controller, **App\Controller\TaskController**.

```php
/**
 * @Route("/tasks/{id}/edit", name="task_edit")
 */
public function editAction(Task $task, Request $request)
{
    // check for "edit" access: calls all voters
    $this->denyAccessUnlessGranted('TASK_EDIT', $task);
```

Si l'accès n'est pas accordé, une **AccessDeniedException** est lancée et plus aucun code de votre contrôleur n'est appelé.

Ensuite, l'une des deux choses suivantes se produira :

- Si l'utilisateur n'est pas encore connecté, il lui sera demandé de se connecter (par exemple redirigé vers la page de connexion).
- Si l'utilisateur est connecté, mais n'est pas le propriétaire de cette tâche, il verra la page 403 accès refusé (que vous pouvez personnaliser).

Documentation : https://symfony.com/doc/5.4/security/voters.html

III. Stockage des utilisateurs

1) Base de données

Les utilisateurs sont persisté en base de données via l'ORM Doctrine.

L'ORM Doctrine n'est pas le sujet de cette documentation, pour plus d'informations, se référer à la documentation officielle.

Les utilisateurs sont stockés dans la table « user ».

	#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra	Action
☐	1	id	int(11)			Non	*Aucun(e)*		AUTO_INCREMENT	Modifier Supprimer Plus
☐	2	username	varchar(25)	utf8mb4_unicode_ci		Non	*Aucun(e)*			Modifier Supprimer Plus
☐	3	password	varchar(255)	utf8mb4_unicode_ci		Non	*Aucun(e)*			Modifier Supprimer Plus
☐	4	email	varchar(60)	utf8mb4_unicode_ci		Non	*Aucun(e)*			Modifier Supprimer Plus
☐	5	roles	longtext	utf8mb4_unicode_ci		Non	*Aucun(e)*	(DC2Type:json)		Modifier Supprimer Plus

Tout cocher *Avec la sélection :* Parcourir Modifier Supprimer Primaire Unique Index Texte entier

Ajouter à la liste centrale Supprimer de la liste centrale de colonnes

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.

```php
public function listAction(UserRepository $userRepository)
{
    return $this->render('user/list.html.twig', ['users' => $userRepository->findAll()]);
}
```