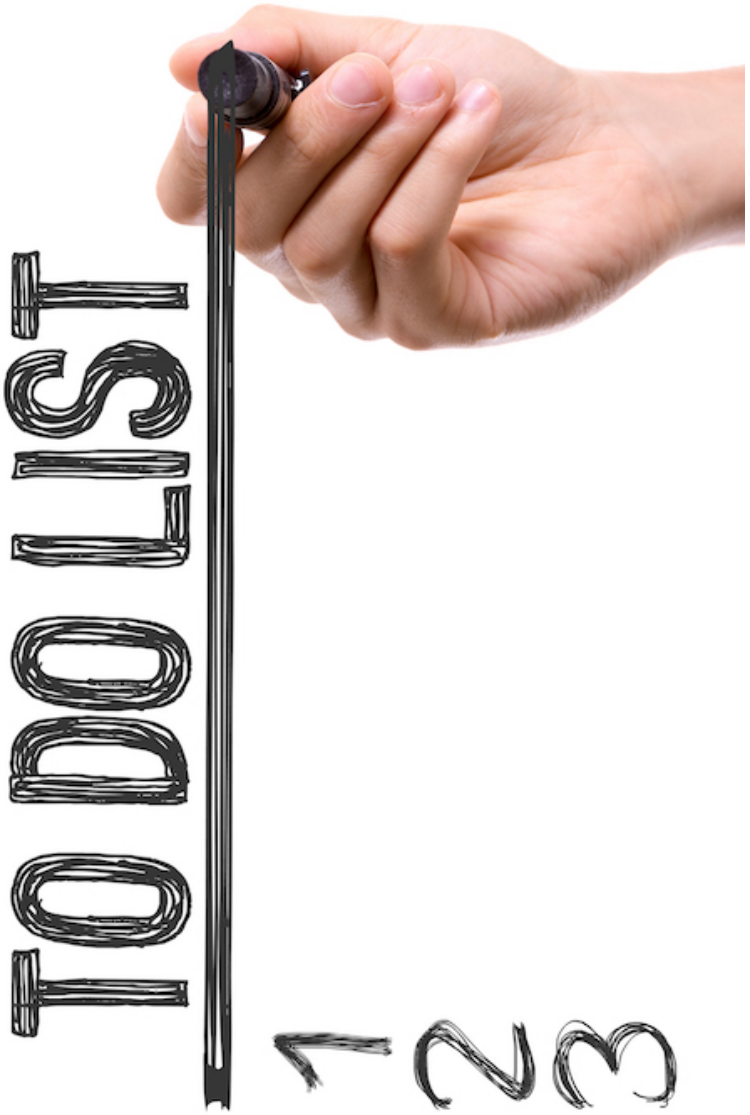


Documentation technique

- Authentification

- Autorisation

SOMMAIRE

Contexte **1**

Implémentation de l'authentification **2**

User Entity *2*

Le provider ou "fournisseur d'utilisateur" *3*

Mot de passe *3*

Le firewall *4*

CSRF Token 5

Loggin out 5

Synthèse de l'authentification *5*

Mise en place de l'autorisation **6**

Attribution du rôle *6*

Gestion des rôles *6*

Gestion des contraintes *7*

Restrictions au sein des contrôleurs 7

Restrictions au sein des templates 8

Contexte

L'application TodoList a pour vocation de proposer un système de gestion de tâches, nécessitant une inscription et une connexion avant toute utilisation.

Les cas d'utilisations détaillés ci-desous sont répartis entre deux profils d'utilisateurs que nous retrouverons dans la partie autorisation de ce document : les **utilisateurs enregistrés** ROLE_USER et les **administrateurs** ROLE_ADMIN.
Ces acteurs intéragissent avec l'application et doivent respecter des règles strictes dans leur possibilité d'action et d'utilisation.

	Utilisateur enregistré	Administrateur
Création d'une tâche	✓	✓
Accès à l'ensemble de ses tâches	✓	✓
Marquer une de ses tâches comme terminée ou non	✓	✓
Modification d'une de ses tâches	✓	✓
Suppression d'une de ses tâches	✓	✓
Accès à l'ensemble des tâches de l'application	✗	✓
Accès à l'ensemble des utilisateurs de l'application	✗	✓
Modification d'un utilisateur	✗	✓
Suppression d'un utilisateur	✗	✓

Pour ce faire, un système d'authentification a été mis en place ainsi que des restrictions d'usage.

Cette documentation a donc pour but d'expliquer :
- comment l'implémentation de l'authentification a été faite ;
- quels ont été les choix pour la partie autorisation.

Vous découvrirez :
- les fichiers utiles pour le paramétrage ;
- la façon dont sont stockées les données des utilisateurs ;
- la procédure d'authentification ;
- la mise en place des restrictions d'accès.

Avant toute chose il est important de bien saisir la distinction entre l'authentification et l'autorisation :

=> **L'authentification** est fondée sur une opération à partir de laquelle nous pouvons déterminer si un utilisateur qui souhaite se connecter à une application, est bien celui qu'il prétend être.

=> **L'autorisation** est un système qui permet de restreindre des éléments de l'application (pages, formulaires, appel de méthode...) à un utilisateur - préalablement connecté - dépendamment de son statut (utilisateur enregistré, administrateur).

Implémentation de l'authentification

C'est le bundle security de Symfony qui fournit les éléments nécessaires à la mise en place de la sécurité de l'application : les fonctionnalités liées à **l'authentification** et à **l'autorisation**.

Documentation : https://symfony.com/doc/current/security.html

Voici les fichiers utiles, liés au système d'authentification :

Fichier	Description
config/packages/security.yaml	Contient les paramétrages
src/Controller/SecurityController.php	Contrôleur contenant les routes de connexion et déconnexion
templates/security/login.html.twig	Template du formulaire de connexion
src/Entity/User.php	Entité représentant un utilisateur

User Entity

Cette entité correspondant à un utilisateur. Elle est indispensable car elle est liée au fonctionnement des systèmes d'authentification et d'autorisation de l'application.
Sa création s'est faite via la commande make:user

Ceci a permis de générer cette classe dans laquelle on retrouve toutes les propriétés qui caractérisent un utilisateur stocké dans la base de données.

Le nom et le mot de passe sont uniques et propres à chaque utilisateur afin de les différencier les uns des autres.

```php
namespace App\Entity;

use ...

#[ORM\Entity(repositoryClass: UserRepository::class)]
#[UniqueEntity('username', message: 'Ce nom d\'utilisateur est déjà utilisé.')]
#[UniqueEntity('email', message: 'Cet email est déjà associé à un compte.')]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private int $id;

    #[Assert\NotBlank(message: "Vous devez saisir un nom d'utilisateur.")]
    #[ORM\Column(type: 'string', length: 180)]
    private string $username;

    #[ORM\Column(type: 'json')]
    private array $roles = [];

    /**
     * @var string The hashed password
     */
    #[ORM\Column(type: 'string')]
    private string $password;

    #[Assert\NotBlank(message: 'Vous devez saisir une adresse email.')]
    #[Assert\Email(message: "Le format de l'adresse n'est pas correcte.")]
    #[ORM\Column(type: 'string', length: 60)]
    private string $email;

    #[ORM\OneToMany(
        mappedBy: 'user',
        targetEntity: Task::class,
        cascade: ["persist"],
        orphanRemoval: true
    )]
    private Collection $tasks;
```

src/Entity/User.php

On retrouve donc logiquement une table User dans la base de donnnées à partir de laquelle les futures requêtes seront liées.

id	username	roles	password	email
1	lolo	["ROLE_USER"]	$2y$13$RqXclFdna62CkrjBKh3wl.8N26YIVkLtBz9bq7LYUqeGWyCe89wra	lolo@gmail.com
2	jane	["ROLE_ADMIN"]	$2y$13$PG2WJN2MQBCftRQEuLuvGOJXMjjNAu8aCSUYhA4HBgFRI7f0x7C.O	jane@gmail.com

Résultat d'une requête récupérant tout les utilisateurs de la base de données

Le provider ou "fournisseur d'utilisateur"

Son rôle est de charger / recharger les utilisateurs à partir de la base de données, basé sur un "identifiant utilisateur".

La configuration actuelle permet de récupérer les utilisateurs via Doctrine en utilisant **l'entité User** et en utilisant la propriété **username** comme "identifiant utilisateur".

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

config/packages/security.yaml

Mot de passe

Afin de sécuriser les données de mot de passe, des fonctionnalités de hachage et de validation nous sont fournies par le SecurityBundle.
Ceci necessite d'avoir implémenté le PasswordAuthenticateUserInterface dans notre classe User.

```php
class User implements UserInterface, PasswordAuthenticatedUserInterface
```

Implémentation de PasswordAuthenticateUserInterface dans la classe User

Les paramètres de hachage du mot de passe sont prédéfinis à l'issue de la commande make:user, dans le fichier security.yaml

Dans notre cas, on retrouve le type de hachage choisi : 'auto'
C'est l'algorithme le plus sécurisé disponible sur notre système qui est donc choisi de manière automatique. Il est possible d'en choisir un autre suivant les besoins.

```yaml
security:
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
```

config/packages/security.yaml

Exemple :

Au traitement d'un formulaire de création d'utilisateur, on peut ainsi hacher le mot de passe saisi par l'utilisateur avant qu'il ne soit intégré à la base de données.

```php
if ($form->isSubmitted() && $form->isValid()) {
    $password = $this->userPasswordHasher->hashPassword(
        $user,
        $user->getPassword()
    );
    $user->setPassword($password);

    $this->entityManager->persist($user);
    $this->entityManager->flush();
```

src/Controller/UserController.php
Hachage d'un mot de passe avant de le persister en base de données

Le firewall

Il est le système d'authentification à lui seul, puisqu'il définit les parties sécurisées de l'application et la façon dont les utilisateurs vont pouvoir s'authentifier.

Les url de l'application sont gérées par le main firewall. On va retrouver les paramétrages correspondant au mode d'authentification choisi.
Pour notre cas c'est par un **formulaire de connexion** (form_login) que les utilisateurs devront se connecter.

A la mise en place de ce mode d'authentification, le système de sécurité redirigera alors les visiteurs non authentifiés vers le login_path lorsqu'ils tenteront d'accèder à un espace sécurisé.

```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false

    main:
        lazy: true
        pattern: ^/
        provider: app_user_provider
        form_login:
            login_path: login
            check_path: login_check
            default_target_path:  /
            enable_csrf: true
        logout:
            path: logout
```

config/packages/security.yaml

Le formulaire de connexion est appelé via le SecurityController.

La fonction **loginAction** du contrôleur retourne le rendu du formulaire. L'authentification est réalisée de façon automatique à la soumission du formulaire. Si les identifiants d'un utilisateur s'avèrent erronés, celui-ci en est informé par un message d'erreur s'affichant sur la page du formulaire de connexion.

```php
#[Route('/login', name: 'login')]
public function loginAction(AuthenticationUtils $authenticationUtils): Response
{
    $error = $authenticationUtils->getLastAuthenticationError();
    $lastUsername = $authenticationUtils->getLastUsername();

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

Fonction loginAction de SecurityController

La vue de l'espace de connexion peut être bien entendu modifiée dans le fichier templates/login/index.html.twig tout en respectant tout de même les conventions suivantes :

- L'élément form doit envoyer une requête **POST** à la route **login** choisi en tant que check_path dans security.yaml
- Les champs nom et mot de passe de l'utilisateur doivent avoir comme nom : _username et _password

```twig
<form action="{{ path('login_check') }}" method="post">
    <div class="mt-3 mb-3 d-grid col-6 mx-auto text-center">
        <label for="username">Nom d'utilisateur</label>
        <input type="text" id="username" name="_username" value="{{ last_username }}" />
    </div>
    <div class="mb-3 d-grid col-6 mx-auto text-center">
        <label for="password">Mot de passe</label>
        <input type="password" id="password" name="_password" />

    </div>
```

templates/security/login.html.twig

CSRF Token

Afin de renforcer la sécurité contre les attaques de type CSRF, un jeton CSRF caché a été ajouté au formulaire de connexion.

Activé d'abord dans security.yaml `enable_csrf: true` il est intégré au formulaire dans login.html.twig

```
<input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">
```

templates/security/login.html.twig => Intégration du jeton CSRF dans le formulaire de connexion.

Loggin out

La déconnexion est activée via le paramètre logout du firewall.
Il est lié à la route logout présente dans le SecurityController.

Lorsqu'un utilisateur sera dirigé vers la route logout, Symfony le **désauthentifiera** et le **redirigera** vers la page d'accueil de l'application.

Synthèse de l'authentification

Pour reprendre et synthétiser l'ensemble du processus d'authentification détaillé dans la documentation de symfony :

1. L'utilisateur tente d'accéder à un espace sécurisé (exemple : tasks/create)
2. Le firewall initie le processus d'authentification en redirigeant l'utilisateur vers le formulaire de connexion. (/login)
3. La page /login affiche le formulaire de connexion via sa route dans le SecurityController et le template login.html.twig.
4. L'utilisateur soumet le formulaire de connexion.
5. Le système de sécurité intercepte la demande, vérifie les informations d'identifications soumises par l'utilisateur et l'authentifie si elles sont correctes ou le renvoie vers le formulaire et lui affiche un message d'erreur si elles ne le sont pas.

Mise en place de l'autorisation

Le processus d'autorisation se fait en deux étapes :

1. Un rôle spécifique est attribué à un utilisateur lors de la création de son compte (ROLE_USER ou ROLE_ADMIN), qui lui sera lié lors de ses futures connexions ;
2. Du code est ajouté pour restreindre l'accès à une URL, un contrôleur... suivant le rôle de l'utilisateur actuellement connecté à l'application.

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

Attribution du rôle

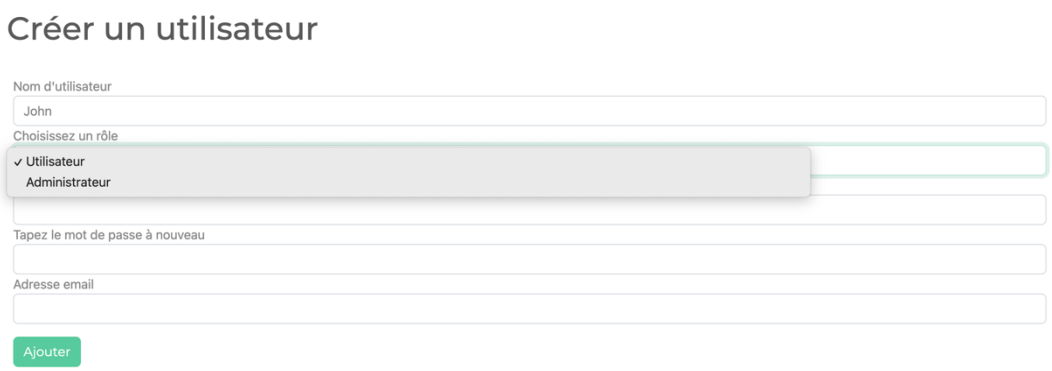
Lorsqu'un utilisateur se connecte, Symfony appelle la méthode getRoles() présente dans l'entité **User** (src/Entity/USer.php) pour déterminer les rôles de cet utilisateur.

```php
public function getRoles(): array
{
    $roles = $this->roles;
    // guarantee every user at least has ROLE_USER
    // $roles[] = 'ROLE_USER';

    return array_unique($roles);
}
```

src/Entity/User => Méthode getRoles

Dans l'application ToDoList, un utilisateur se verra attribué automatiquement le rôle préalablement sélectionné lors de la création de son compte. A titre informatif, celui-ci pourra être modifié ultérieurement par un Administrateur.



Formulaire de création d'un utilisateur

Gestion des rôles

Une hiérarchisation des rôles peut être définie dans le fichier security.yaml

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

Pour notre cas, celle-ci permet à un utilisateur ROLE_ADMIN d'hériter de celui de ROLE_USER. Ainsi il aura accès également aux éléments de contraintes concernant les ROLE_USER.

Gestion des contraintes

Dans l'application TodoList les contraintes ont été appliquées au sein du code, dans les contrôleurs et templates. Les fonctions des contrôleurs n'ayant pas de restriction sont accessibles à tout visiteur sans obligation de connexion.

Restrictions au sein des contrôleurs

Il est possible d'appliquer une restriction sur l'ensemble des fonctions du contrôleur.

```php
#[IsGranted('ROLE_USER')]
class TaskController extends AbstractController
{
```

Restriction globale sur la classe TaskController

Sur une fonction en particulier.

```php
#[Route('/users/{id}/delete', name: 'user_delete', methods: ['GET', 'POST'])]
#[IsGranted('ROLE_ADMIN', message: "Espace réservé aux administrateurs.")]
public function deleteAction(
```

Accès restreint aux ROLE_ADMIN pour la fonction deleteAction

Voici un récapitulatif des restrictions d'accès présentes dans les contrôlleurs :

Contrôlleur	Fonction	ROLE_USER	ROLE_ADMIN
DefaultController	indexAction	✓	✓
SecurityController	loginAction	Aucune restriction	
	logoutCheck		
TaskController	listTasks	✓	✓
	listTasksDone	✓	
	listTaskTodo	✓	
	listTaskManage	✗	
	createAction	✓	
	editAction	✓	
	toggleTaskAction	✓	
	deleteTaskAction	✓	
UserController	listAction	✗	✓
	createAction	Aucune restriction	
	editAction	✗	✓
	deleteAction	✗	✓

Il est également possible de restreindre l'accès à un traitement au sein du code en vérifiant le rôle attribué à l'utilisateur actuellement connecté. Il suffit alors d'injecter le service **Security** et d'appeler la méthode **isGranted** avec le **Role** en argument d'entrée.

```php
if ($security->isGranted( attributes: 'ROLE_USER') && !$security->isGranted( attributes: 'ROLE_ADMIN')) {
    if ($task->getUser() !== $this->getUser()) {
        $this->addFlash( type: 'error', message: 'Vous ne pouvez pas supprimer cette tâche.');
        return $this->redirectToRoute( route: 'homepage', [], status: Response::HTTP_SEE_OTHER);
    }
```

Exécution conditionnelle avec necessité d'avoir un utilisateur ayant pour rôle ROLE_USER et non ROLE_ADMIN

Restrictions au sein des templates

Des vérifications de rôle sont établies aussi au sein des templates en utilisant la fonction **is_granted** ce qui permet de maitriser l'affichage d'éléments visuels suivant l'utilisateur actuellement connecté.

```twig
{% if is_granted('ROLE_ADMIN') %}
    <div class="row m-3">
        <div class="col text-center">
            <a href="{{ path('user_list') }}" class="btn btn-dark ">Consulter la liste des utilisateurs de l'application</a>
        </div>
        <div class="col text-center">
            <a href="{{ path('task_manage') }}" class="btn btn-dark ">Consulter la liste des tâches de l'application</a>
        </div>
    </div>
{% endif %}
```

Contrainte d'affichage avec utilisation de la fonction is_granted dans le template default/index.html.twig