Todo&Co

Documentation technique

- Authentification
- Autorisation

Nassim Taoussi

Sommaire :

Contexte

L'application Todolist à pour vocation de mettre à disposition un système de gestions des rôles en fonction des utilisateurs des leurs création en leur attribuant un rôle.

Les rôles sont réparties en deux profil :
- **ROLE_USER** pour les utilisateurs enregistrés
- **ROLE_ADMIN** pour les administrateurs

Ces deux profils ont donc pour objectif de définir des champs d'actions stricte.

Voici un récapitulatif des actions autorisé selon le profil :

	Utilisateur	Administrateur
Accès à la liste des tâches	X	X
Création d'une tâche	X	X
Edition d'une tâche	X	X
Suppression d'une tâche	X	X
Marqué une tâche comme terminer ou non	X	X
Accès à la liste des utilisateurs		X
Modifier un utilisateur		X
Supprimer un utilisateur		X

Implémentation de la partie Authentification :

Tout d'abord qu'est ce que l'authentification ? C'est tout simplement un système à parti duquel on peut déterminer si l'utilisateur qui souhait se connecter est bien celui qu'il prétend être.

Afin de mettre en place tout les éléments de sécurité nécessaire à cette dernière ainsi qu'à l'autorisation (que nous verrons plus tard dans cette documentation) pour notre application. Nous allons nous servir du bundle security de Symfony .

Lien de la documentation du bundle :
https://symfony.com/doc/current/security.html

L'entité User :

Cette entité correspond à un utilisateur elle est indispensable à l'application et plus particulièrement au fonctionnement des systèmes d'authentification et d'autorisation. Il faudra crée une classe User qu'on implémentera par la suite avec **UserInterface** & **PasswordAuthentictatedUserInterface**. Le **UserInterface** va nous permette de géré l'identité d'un utilisateur et le **PasswordAuthentictatedUserInterface** va géré la partie mot de passe de l'utilisateur

L'adresse email de l'utilisateur est unique afin de le différencier des autres utilisateurs.

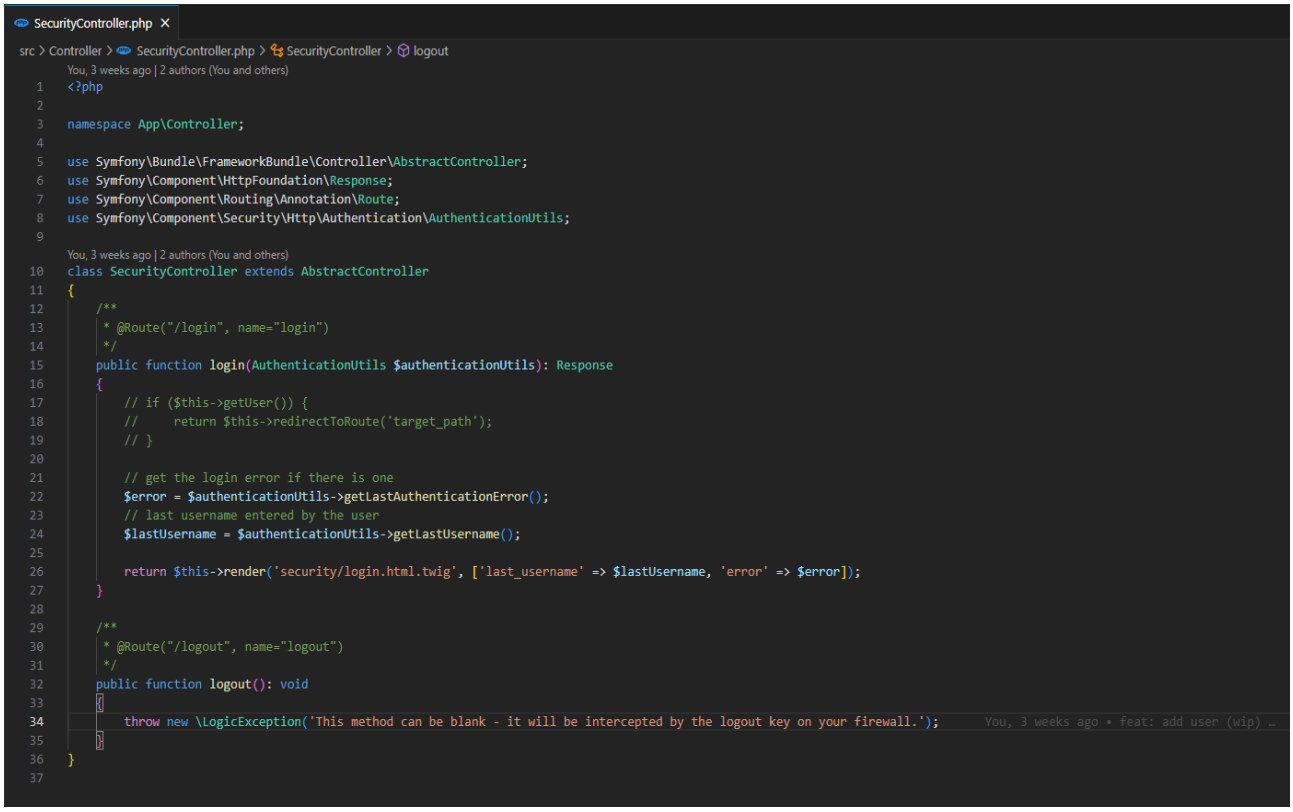
User.php

src > Entity > User.php > User

```php
namespace App\Entity;

use App\Repository\UserRepository;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
use Symfony\Component\Uid\Uuid;

You, 2 weeks ago | 2 authors (NassimTaoussi and others)
#[ORM\Entity(repositoryClass: UserRepository::class)]
#[UniqueEntity(fields: ['email'], message: 'Il y a déjà un compte crée avec cet email')]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private int $id;

    #[ORM\Column(length: 255)]
    private ?string $username = null;

    #[ORM\Column(length: 64)]
    private ?string $password = null;

    #[ORM\Column(length: 60)]
    private ?string $email = null;

    #[ORM\Column(length: 30)]
    private ?string $role = null;

    #[ORM\OneToMany(mappedBy: 'author', targetEntity: Task::class)]
    private Collection $tasks;

    public function __construct()
    {
        $this->tasks = new ArrayCollection();
    }

    public function getId(): ?int
    {
        return $this->id;
    }

    public function getUsername(): ?string
    {
        return $this->username;
    }

    public function setUsername(string $username): static
    {
        $this->username = $username;

        return $this;
    }

    public function getPassword(): ?string
    {
        return $this->password;
    }
```

Controller :

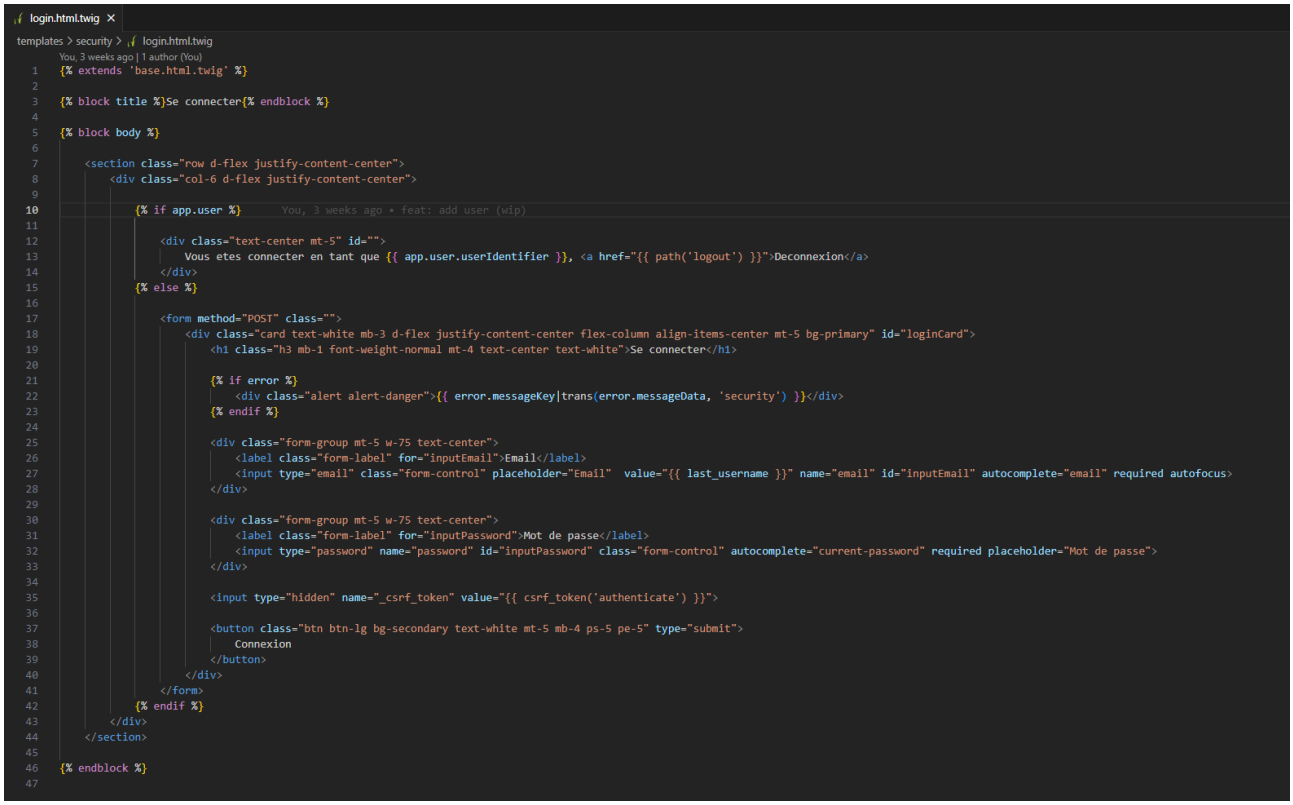
Nous avons un SecurityController, qui nous permet de géré l'authentification en se plaçant entre la vue coté client et l'entity User. Ce dernier contient une méthode « login » qui gère l'authentification lorsque la route pour se connecter est appeler, de même qu'une méthode « logout » pour la déconnexion.



Dans notre méthode « login » on lui injecte le service AuthenticationUtils qui va géré l'authentification pour nous. Puis a la fin de notre méthode nous renvoyons la vue qui lui est dédié pour que l'utilisateur renseigne son email et mot de passe afin de se connecter.

Vue :

Comme vue précédemment la vue renvoyer depuis le controlleur est le fichier « login.html.twig » contenue dans le dossier « security » qui lui même est contenue dans le dossier « templates ». Ce fichier écrit en twig contient un formulaire de connexion



```twig
{% extends 'base.html.twig' %}

{% block title %}Se connecter{% endblock %}

{% block body %}

    <section class="row d-flex justify-content-center">
        <div class="col-6 d-flex justify-content-center">

            {% if app.user %}       You, 3 weeks ago • feat: add user (wip)

                <div class="text-center mt-5" id="">
                    Vous etes connecter en tant que {{ app.user.userIdentifier }}, <a href="{{ path('logout') }}">Deconnexion</a>
                </div>
            {% else %}

                <form method="POST" class="">
                    <div class="card text-white mb-3 d-flex justify-content-center flex-column align-items-center mt-5 bg-primary" id="loginCard">
                        <h1 class="h3 mb-1 font-weight-normal mt-4 text-center text-white">Se connecter</h1>

                        {% if error %}
                            <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
                        {% endif %}

                        <div class="form-group mt-5 w-75 text-center">
                            <label class="form-label" for="inputEmail">Email</label>
                            <input type="email" class="form-control" placeholder="Email"  value="{{ last_username }}" name="email" id="inputEmail" autocomplete="email" required autofocus>
                        </div>

                        <div class="form-group mt-5 w-75 text-center">
                            <label class="form-label" for="inputPassword">Mot de passe</label>
                            <input type="password" name="password" id="inputPassword" class="form-control" autocomplete="current-password" required placeholder="Mot de passe">
                        </div>

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

                        <button class="btn btn-lg bg-secondary text-white mt-5 mb-4 ps-5 pe-5" type="submit">
                            Connexion
                        </button>
                    </div>
                </form>
            {% endif %}
        </div>
    </section>

{% endblock %}
```

Security.yaml :

Le fichier security.yaml contenue dans le dossier config sert a paramétré la sécurité de l'application comme par exemple : la classe utilisé pour fournir des utilisateurs, le hashage des mots de passes, la hierarchy des rôles ou encore « se souvenir de moi ».

1 - Le provider : Appeler plus communément en français « le fournisseur d'utilisateur ». Son rôle est de charger ou recharger les utilisateurs à partir de la « base de donnée » en se basant sur la classe « User » contenue dans notre dossier Entity. Ce dernier se base sur un identifiant utilisateur et en l'occurrence pour notre application celui ci va être l'email de l'utilisateur.

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

2 - Hasher :

Le hashage du mot de passe de l'utilisateur est prédéfini dans le fichier « security.yaml ».

Dans notre cas, on retrouve le type de hashage « auto ». C'est l'algorithme qui est choisi de manière automatique. Il est bien évidemment possible d'en avoir d'autres selon le besoin.

security.yaml

config > packages > security.yaml

You, 2 weeks ago | 2 authors (NassimTaoussi and others)

```
security:
    enable_authenticator_manager: true
    # 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
```

Par l'exemple avec cet configuration on peut depuis la création d'un utilisateur, à la soumission du formulaire hasher le mot de passe dans notre « UserController » lorqu'on appelle la méthode add() de notre « UserManagerInterface ». Ceci à l'aide du service UserPasswordHasherInterface qui va hasher le mot de passe pour nous

```php
#[Route('/users/create', name: 'create_user')]
#[IsGranted("ROLE_ADMIN")]
public function createUser(Request $request, UserManagerInterface $userManagerInterface): Response
{
    $user = new User();

    $form = $this->createForm(UserType::class, $user)->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $userManagerInterface->add($user);      You, 4 days ago • feat: interface …
        $this->addFlash('success', 'Vous venez de créé un nouvelle utilisateur');

        return $this->redirectToRoute('users');
    }


    return $this->render('user/create_user.html.twig', [
        'form' => $form->createView()
    ]);
}
```

```php
You, 4 days ago | 1 author (You)
11  final class UserManager implements UserManagerInterface
12  {
13
14      public function __construct(
15          private Security $security,
16          private EntityManagerInterface $entityManager,
17          private UserPasswordHasherInterface $passwordHasher,
18          private UserRepository $userRepository,
19      ) {
20      }
21
22      public function add(User $user) {
23          $hashedPassword = $this->passwordHasher->hashPassword(
24              $user,
25              $user->getPassword()
26          );
27
28          $user->setPassword($hashedPassword);
29          $this->entityManager->persist($user);
30          $this->entityManager->flush();
31      }
```

3 - firewall :

En français « pare-feu » il défini les parties sécurisé de l'application avec la possibilité de restreindre l'accès à certaines parties. On va y retrouver notamment la configuration choisi pour le mode d'authentification.

```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
                # where to redirect after logout
                # target: app_any_route

            remember_me:
                secret: '%kernel.secret%'
                lifetime: 604800
                path: /
                always_remember_me: true
```

Dans notre cas la gestion de l'authentification est défini et géré par notre classe « UserAuthenticator » dans le dossier « src/security » depuis la methode authenticate.

```php
You, 3 weeks ago | 1 author (You)
class UserAuthenticator extends AbstractLoginFormAuthenticator
{
    use TargetPathTrait;

    public const LOGIN_ROUTE = 'login';

    public function __construct(private UrlGeneratorInterface $urlGenerator)
    {
    }

    public function authenticate(Request $request): Passport
    {
        $email = $request->request->get('email', '');

        $request->getSession()->set(Security::LAST_USERNAME, $email);

        return new Passport(
            new UserBadge($email),
            new PasswordCredentials($request->request->get('password', '')),
            [
                new CsrfTokenBadge('authenticate', $request->request->get('_csrf_token')),
                new RememberMeBadge(),
            ]
        );
    }

    public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response
    {
        if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
            return new RedirectResponse($targetPath);
        }

        // For example:
        return new RedirectResponse($this->urlGenerator->generate('home'));
        //throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);       You, 3 weeks ago • feat: add
    }

    protected function getLoginUrl(Request $request): string
    {
        return $this->urlGenerator->generate(self::LOGIN_ROUTE);
    }
}
```

Implémentation de la partie Autorisation :

L'autorisation c'est un système qui permet de restreindre des éléments : une page, un formulaire, un bouton, des méthodes à un utilisateur en fonction du rôle qu'il détient.

Security.yaml :

1 – access_control :

Dans notre fichier security.yaml il y a une notion d'access_control qui permet de définir l'accès à des methodes pour une catégorie d'utilisateur qui détient un rôle spécifique.

```
39      access_control:
40          # - { path: ^/users, roles: ROLE_ADMIN }
41          # - { path: ^/users/create, roles: ROLE_ADMIN }
42          # - { path: ^/users/edit/{id}, roles: ROLE_ADMIN }
43          # - { path: ^/users/delete/{id}, roles: ROLE_ADMIN }
```

2 – role_hierarchy

La notion de rôle hierarchy quand a elle permet de hierarchiser les rôles

```
45      role_hierarchy:
46          ROLE_USER: [ROLE_USER]
47          ROLE_ADMIN: [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

L’attribut isGranted de Symfony et la fonction is_granted de Twig :

- L’attribut isGranted de Symfony :

Dans les différents Controller une annotation spécifique a été mis en place pour une établir restriction en fonction du rôle des utilisateurs.

Exemple sur une méthode du controller « UserController » :

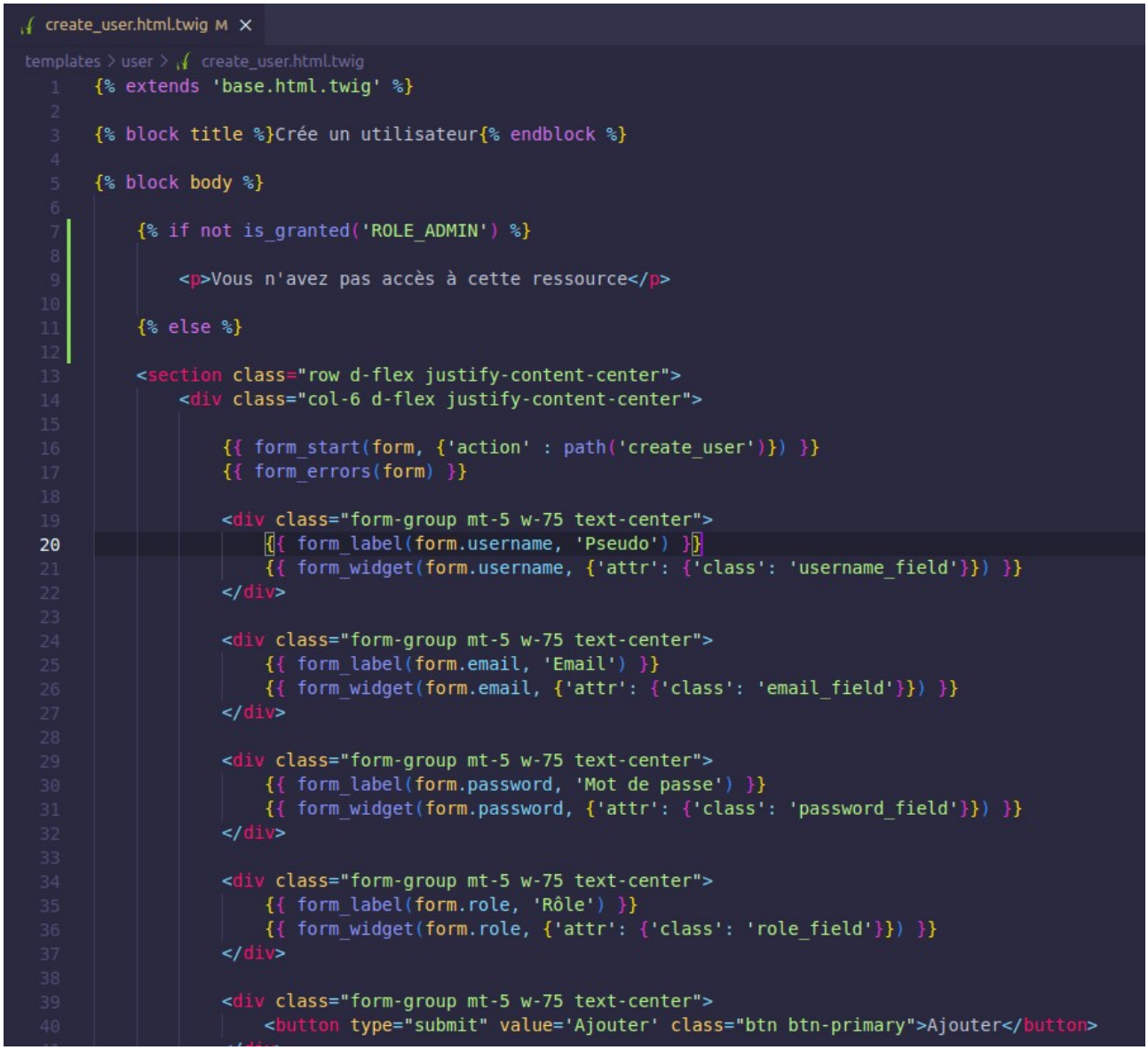
```php
class UserController extends AbstractController
{
    #[Route('/users', name: 'users')]
    #[IsGranted("ROLE_ADMIN")]
    public function index(UserRepository $userRepository): Response
    {

        $users = $userRepository->findAll();

        return $this->render('user/index.html.twig', [
            'users' => $users,
        ]);
    }
```

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

Ensuite sur cette base l’annotation isGranted établi une restriction qui permet seulement aux utilisateur ayant le rôle « ROLE_ADMIN » de pouvoir accéder à cet ressource.

- La fonction is_granted de Twig:

On à également la possibilité d’établir des restrictions à travers nos templates twig avec sa fonction **is_granted** à qui l’on passe le rôle en question sur qui l’on souhaite statuer ce qui va nous permettre d’afficher ou ne pas afficher un éléments à un utilisateur en fonction du profil qu’il a. Comme par exemple dans ce fichier twig :



Ici on demande a Twig à travers une condition que si l'utilisateur demandant cette ressource n'a pas le rôle administrateur de lui afficher seulement un message texte lui expliquant qu'il n'a pas accès à cette ressource. Dans le cas contraire on lui permet d'avoir accès au formulaire de création d'un utilisateur.

Mis en place du voter :

Afin d'établir des contraintes pour l'accès a certaines actions de la ressource « Task » notamment la suppression et l'édition, un Voter a été mis en place.

Ce dernier permet de contraindre l'accès à une ressource par un autre facteur que les rôles. Dans notre cas on y fait appel à travers l'attribut **isGranted** de Symfony en lui passant directement le service en argument avec la constante que l'on veut appeler qui détient le nom de la methode sur laquelle on souhaite établir la contrainte.

```php
#[Route('/tasks/delete/{id}', name: 'delete_task')]
#[IsGranted(TaskVoter::TASK_DELETE, subject: 'task')]
public function deleteTask(Task $task, TaskManagerInterface $taskManagerInterface): Response
{
    //$this->denyAccessUnlessGranted(TaskVoter::TASK_DELETE, $task);
    $taskManagerInterface->delete($task);
    return $this->redirectToRoute('tasks');

}
```

Par exemple, ici en l'occurence que l'utilisateur qui est lié à une tâche :

```php
<?php

namespace App\Security\Voter;

use App\Entity\User;
use App\Entity\Task;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;
use Symfony\Component\Security\Core\Security;
use Symfony\Component\Security\Core\User\UserInterface;

class TaskVoter extends Voter {

    public const TASK_EDIT = 'editTask';
    public const TASK_DELETE = 'deleteTask';
    public const TASK_TOGGLE = 'toggleTask';
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    protected function supports(string $attribute, mixed $subject): bool
    {
        return in_array($attribute, [self::TASK_EDIT, self::TASK_DELETE,self::TASK_TOGGLE])
            && $subject instanceof \App\Entity\Task;
    }

    protected function voteOnAttribute($attribute, $subject, TokenInterface $token) : bool
    {
        $user = $token->getUser();



        // On verifie si l'utilisateur est anonyme
        if (!$user instanceof UserInterface) {
            return false;
        }
```

Si la méthode voteOnAttribute est exécutée, les conditions sont vérifiées dans des instructions conditionnelles : Vérifie que l'utilisateur est déjà dans un premier temps connecté est ensuite s'il a le rôle admin et on check si la task n'a pas d'auteur.

```php
        // On verifie si l'utilisateur est Admin
        if($this->security->isGranted("ROLE_ADMIN")) {
            return true;
        }

        // On verifie si la tache a un proprietaire
        if($subject->getAuthor() === null){
            return false;
        }

        switch ($attribute) {
            case self::TASK_EDIT:

                //On verifie si l'utilisateur  peut editer
                return $this->canEditTask($subject, $user);

                break;
            case self::TASK_DELETE:

                //On verifie si l'utilisateur peut supprimer
                return $this->canDeleteTask($subject, $user);
                break;
            case self::TASK_TOGGLE:

                //On verifie si l'utilisateur peut marquer la tache
                return $this->canToggleTask($subject, $user);
                break;
        }

        return false;
    }
```

Si elle ne rentre dans aucun de ces cas, en fonction du $attribut passer dans notre switch qui contient la methode appeler dans le controller. On appelle une seconde methode. Ici **canDeleteTask** qui prend 2 arguments, le sujet (task) et le $user l'utilisateur.

```php

    private function canEditTask(Task $task, User $user)
    {
        // L'auteur de la tache peut modifier
        return $user === $task->getAuthor();
    }

    private function canDeleteTask(Task $task, User $user)
    {
        // L'auteur de la tache peut supprimer
        return $user === $task->getAuthor();
    }
    private function canToggleTask(Task $task, User $user)
    {
        // L'auteur peut marquer la tache
        return $user === $task->getAuthor();
    }
}
```

Celle ci va retourner true si l’utilisateur ayant fait la demande de suppression est bien l’auteur de la task.