SYMFONY

Authentification

I – L'ENTITÉ USER

1 – CRÉER UNE CLASSE USER

Une classe User peut être facilement créée grâce au MakerBundle et la commande

php bin/console make :user

Cette commande vous permettra de créer une classe User de base, son « Repository » associé (fichier permettant de récupérer les entités grâce à Doctrine), ainsi que de modifier le fichier

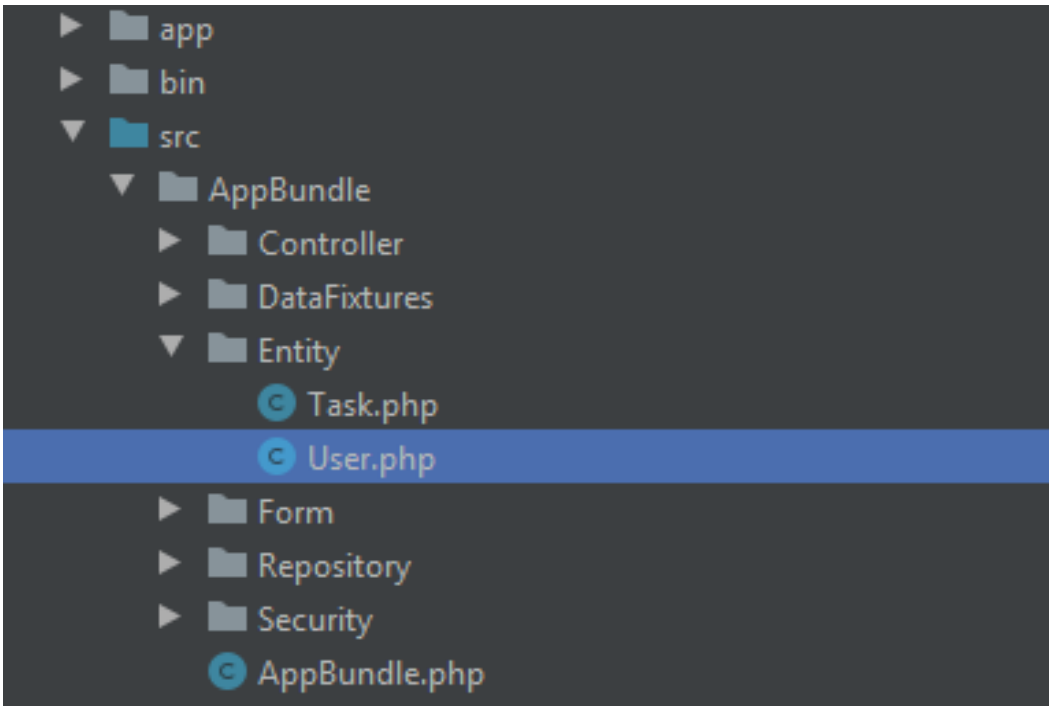
« security.yaml » ; ce que nous verrons plus tard.

Référence :

https://symfony.com/doc/current/security.html#a-create-your-user-class

2 – OÙ SE SITUE LE FICHIER ?

La classe User se situe dans src/Entity/User.php.



3 – LES SPÉCIFICITÉS DE LA CLASSE USER

A – LES ATTRIBUTS DE BASE

Les attributs de base sont :

- Id
- Username
- Email
- Password
- Rôle

Ce sont des attributs classiques pour un utilisateur qui a besoin de s'authentifier.

La seule particularité étant l'attribut rôle permettant d'associer un rôle à l'utilisateur : c'est un tableau (de type Json en BDD), un utilisateur pouvant avoir plusieurs rôles.

C'est dans cette classe que vous pourrez aussi ajouter des contraintes sur les attributs, par exemple forcer l'utilisateur à choisir un mot de passe « solide » ou s'assurer de l'unicité d'un attribut (Il vous faudra importer les classes nécessaires).

```
/**
 * @Assert\NotBlank()
 * @ORM\Column(type="string", length=64)
 */
private $password;
```

Ci-dessus, des exemples de contraintes que l'on peut apporter à l'attribut « password ».

@Assert\NotBlank	Ne peut pas être vide
@Assert\Lenght	Force une longueur minimum et maximale
@Assert\Regex	Une expression régulière permettant de forcer un mot de passe a sécurité haute ou de mettre certain caractère ou majuscule.

Référence :

https://symfony.com/doc/current/reference/constraints.html

B – USERINTERFACE

Toute classe permettant de s'authentifier sur Symfony doit implémenter l'interface UserInterface.

```
class User implements UserInterface
```

Implémenter cette interface nous force à implémenter dans la classe User des méthodes Indispensables à l'authentification.

1 – LA MÉTHODE GETUSERNAME()

```
public function getUsername()
{
    return $this->username;
}
```

La méthode getUsername() retourne un identifiant pour l'utilisateur, c'est avec celui-ci qu'il se connectera.

2 – LA MÉTHODE GETROLES()

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

    return array_unique($roles);
}
```

La méthode getRoles() permet de retourner les rôles de l'utilisateur, ci-dessus c'est la méthode

getRoles() créée avec la commande make:user .

Dans notre application pour des raisons de sécurité de cohérence et de simplicité nous utilisons la

méthode ci-dessous :

```php
public function getRoles()
{
    $roles = $this->roles ;
    return $roles;
}
```

3 – LES AUTRES MÉTHODES

Nous devons implémenter également la méthode getPassword() qui retourne le mot de passe pour authentifier l'utilisateur.

```php
public function getPassword()
{
    return $this->password;
}
```

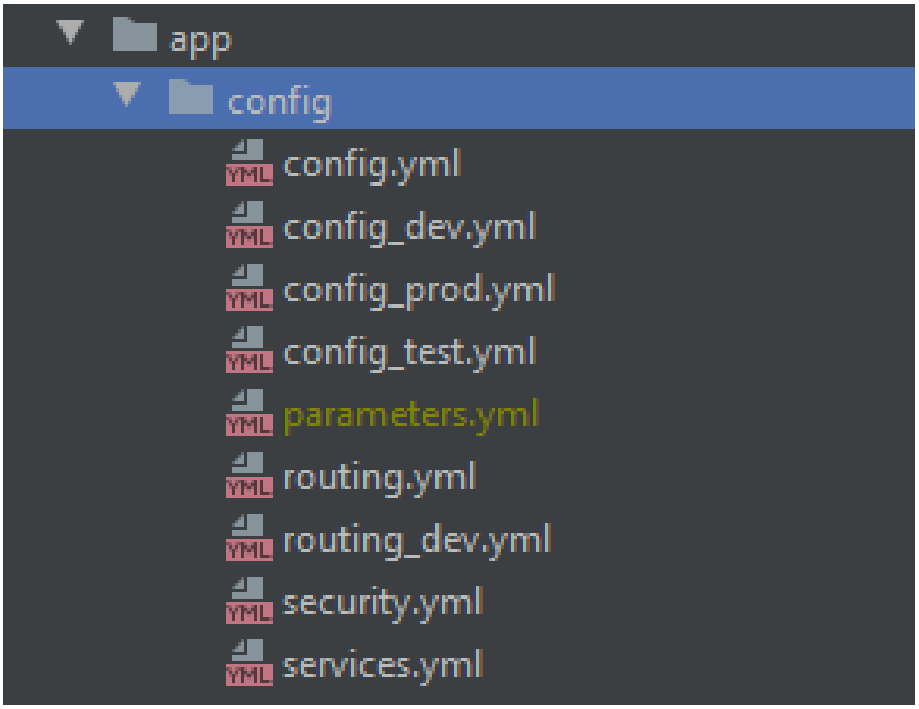
La méthode getSalt() qui retourne le « sel » permettant d'encoder le mot de passe, nous laissons dans notre cas cette méthode vide car nous utilisons les méthodes d'encodage fournies par Symfony.

```
public function getSalt()
{
    return null;
}
```

La méthode eraseCredentials() qui permet d'effacer les données sensibles de l'utilisateur, par exemple si le mot de passe non-encodé est stocké dans l'utilisateur. Nous laissons également cette méthode vide car nous ne stockons pas de données sensibles

```
public function eraseCredentials()
{
}
```

1 – OÙ SE SITUE LE FICHIER ?



2 – L'USERPROVIDER

```
providers:
    doctrine:
        entity:
            class: AppBundle:User
            property: username
```

Nous spécifions ici que le « Provider » devra utiliser une entité pour récupérer les utilisateurs. Ici c'est l'entité « User » avec la propriété « username ».

En spécifiant au « Provider » que nous utiliserons une entité, il utilisera Doctrine pour récupérer les utilisateurs stockés en base de données en utilisant la propriété « username ».

Vous pouvez ici modifier la propriété de recherche et par exemple vouloir que l'utilisateur soit récupéré en fonction de son email.

Référence : https://symfony.com/doc/current/security/user_provider.html

3 – ENCODAGE DU MOT DE PASSE

Ici nous définissons l'encodage du mot de passe de l'utilisateur.

L'encodage du mot de passe se fera selon un algorithme défini ici.

```
encoders:
    AppBundle\Entity\User: bcrypt
```

Le chemin de la nôtre classe User doit être spécifié entièrement.

Petit rappel wikipédia sur le bcrypt :

bcrypt est une fonction de hachage créée par Niels Provos et David Mazières. Elle est basée sur l'algorithme de chiffrement Blowfish et a été présentée lors de USENIX en 1999[1]. En plus de l'utilisation d'un sel pour se protéger des attaques par table arc-en-ciel (*rainbow table*), bcrypt est une fonction adaptative, c'est-à-dire que l'on peut augmenter le nombre d'itérations pour la rendre plus lente. Ainsi elle continue à être résistante aux attaques par force brute malgré l'augmentation de la puissance de calcul.

Référence :

https://symfony.com/doc/current/security.html#c-encoding-passwords

1 – CRÉER L'AUTHENTIFICATION

Vous pouvez également créer un système d'authentification en utilisant la commande php bin/console make:auth

Cela créera automatiquement un « authenticators » : un fichier que l'on utilisera pour authentifier les utilisateurs avec « guard ».

Cela mettra à jour le fichier « security.yaml ».

Il sera également créé un « controller » avec une méthode pour se connecter et un fichier « twig » contenant un formulaire d'authentification.

Référence :
https://symfony.com/doc/current/security/form_login_setup.html#generating-the-login-form

2 – PARE-FEU ET GUARD

A – PARE-FEU

Toujours dans le fichier « security.yaml » nous allons voir comment est configuré le pare-feu.

```
main:
    logout_on_user_change: true
    anonymous: ~
    pattern: ^/
    form_login:
        login_path: login
        check_path: login_check
        always_use_default_target_path:  true
        default_target_path:  /
    logout: ~
```

- La partie « dev » est automatiquement configuré lors de la création d'un projet Symfony.
- La partie « main » est notre pare-feu principal.
- La partie anonymous ~ veux dire que pour accéder à la page de connexion nous n'avons pas besoin d'être connecter.
- La partie « pattern » signifie le l'initialisation du pare-feu .Le pattern définie au début de quel expression régulière le pare-feu débute.
- La partie « logout_on_user_change » signifie que l'utilisateur est considéré comme changé si la méthode renvoie false.Si la valeur est false, Symfony ne déclenche pas de déconnexion lorsque l'utilisateur a changé.
- La partie « login_path» signifie le chemin vers laquelle l'utilisateur est redirigé s'il tente d'accéder à une ressource protégé.
- La partie « login_path» signifie la route ou le chemin que votre formulaire de connexion doit soumettre. Le pare-feu interceptera toutes

les demandes (demandes POST uniquement, par défaut) vers cette URL et traitera les informations d'identification de connexion soumises.

- La partie « always_use_default_target_path » signifie que si la valeur est à true, les utilisateurs sont toujours redirigés vers le chemin cible par défaut.
- La partie « default_target_path » signifie que vous pouvez modifier la page vers laquelle l'utilisateur est redirigé si aucune page précédente n'a été stockée dans la session.

IV – GESTIONS DES RÔLES

1 – SECURITY.YAML

Toujours dans le fichier « security.yaml » nous pouvons gérer la hiérarchie des rôles et restreindre l'accès à certaines parties du site en méthode de ceux-ci.

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

Dans cette partie « access_control » nous définissions quel partie de notre application est ouverte à quel utilisateur.

Sur cette access_control nous constatons que la partie login et home sont accessible à tout personne anonyme.

Référence : https://symfony.com/doc/3.4/security/access_control.html

2 – UTILISATION DES RÔLES

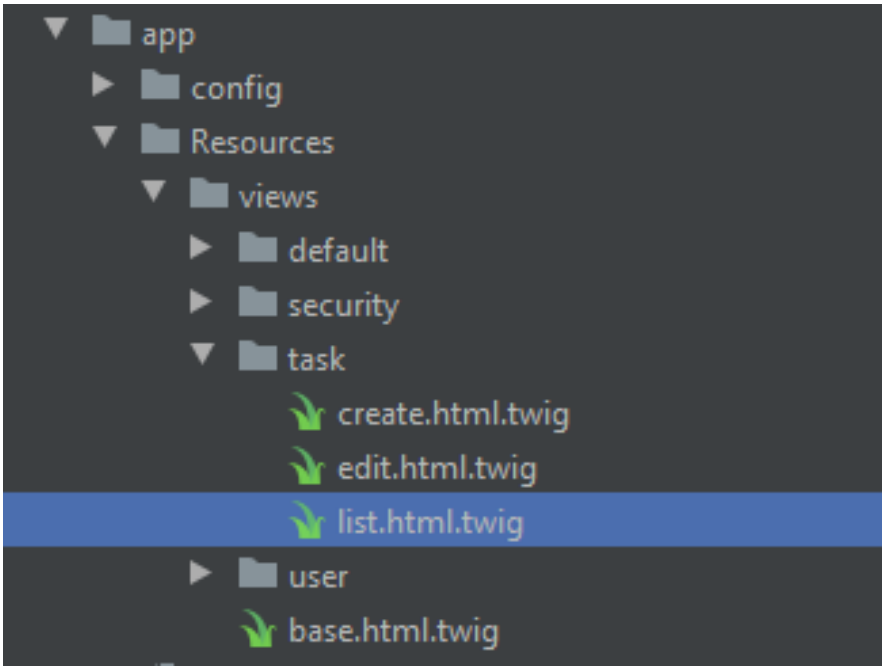
A – EXEMPLE D'UTILISATION DANS UN CONTROLLER

```php
/**
 * @Route("/{id}/edit", name="task_edit")
 * @IsGranted({"ROLE_ADMIN","ROLE_USER"})
 * @param Task $task
 * @param Request $request
 * @return RedirectResponse|Response
 */
public function editAction(Task $task, Request $request)
{
```

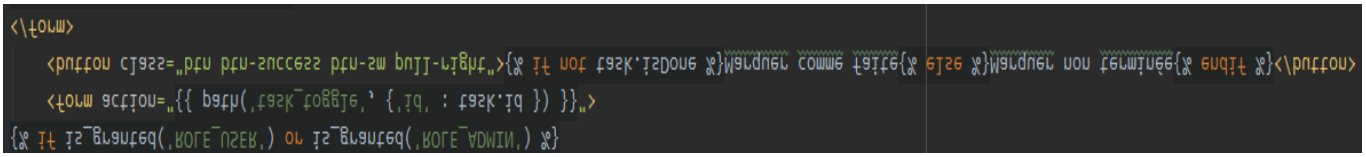
Ici l'annotation @IsGranted(« ROLE_ADMIN ») permet de s'assurer que la fonction editAction() ne sera accessible qu'aux utilisateurs ayant au minimum le rôle d'administrateur.

B – EXEMPLE D'UTILISATION DANS UN FICHIER TWIG

1 – OÙ SE SITUE LE FICHIER ?



2 – CAS D'UTILISATION



Ici le bouton «Marquer comme faite» ne sera visible que par les utilisateurs ayant au minimum le rôle d'administrateur ou le rôle User.

Cependant l'utilisation du @Security plus complexe vu précédemment n'est pas possible directement avec la fonction is_granted () ; afin d'utiliser cette fonction sur des permissions plus complexes nous allons utiliser les « voters », que nous allons voir maintenant.

Référence :
https://symfony.com/doc/current/reference/twig_reference.html#is-granted

3 – LES VOTERS

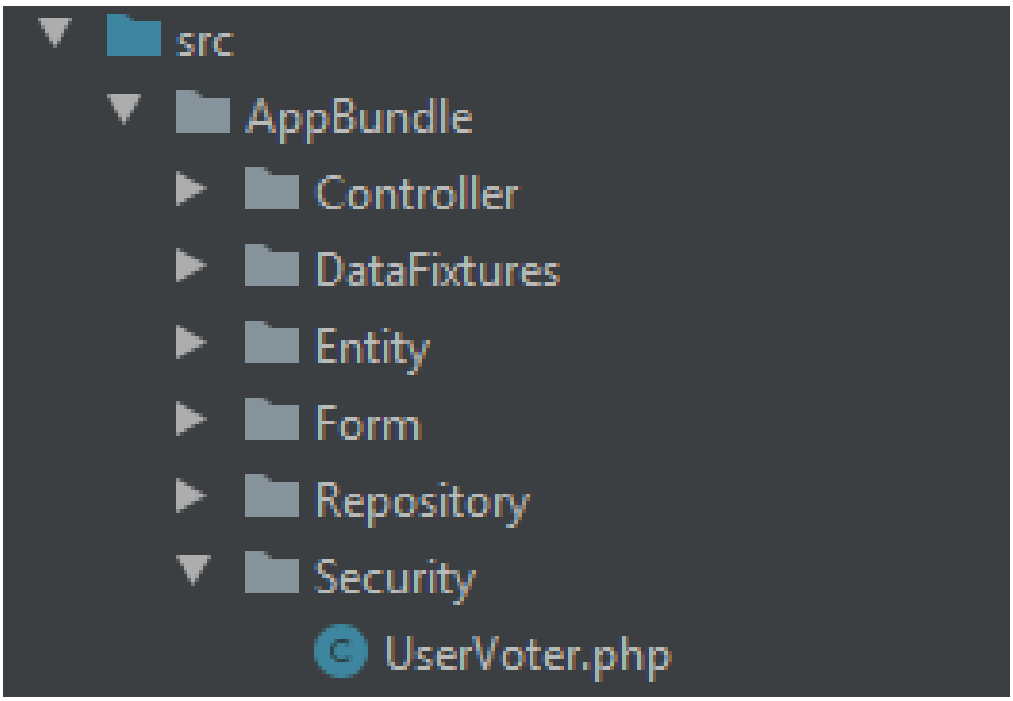
Un voter est un objet PHP qui nous permet de gérer les cas de permissions plus complexes, que la hiérarchie des rôles seule ne permet pas, mais également d'utiliser ces permissions simplement dans nos fichiers twig.

Référence :

https://symfony.com/doc/current/components/security/authorization.html#voters

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

1 – OÙ SE SITUE LE FICHIER ?



B – LA MÉTHODE SUPPORTS ()

```php
protected function supports($attribute, $subject)
{
    return in_array($attribute, ['EDIT','DELETE']) && $subject instanceof Task;
}
```

Cette fonction permet de déterminer si l'attribut (type d'action) et le sujet (l'objet à sécuriser) sont pris en charge par notre « voter ».

Nous ajoutons donc ici les attributs (les actions que l'on va sécuriser) et la classe de notre sujet, ici la classe Task.

C – LA MÉTHODE VOTEONATTRIBUTE()

```php
protected function voteOnAttribute($attribute, $task, TokenInterface  $token)
{
    $user = $token->getUser();
    if (!$user instanceof UserInterface) {
        return false;
    }
    if ($task->getUser()->getId() == null && $user->getRoles() == ['ROLE_ADMIN']) {
        return true;
    }
    switch ($attribute) {
        case 'EDIT' || 'DELETE':
            if ($task->getUser()->getId()== $user->getId()) {
                return true;
            }
            break;
    }
    return false;
}
```

Nous allons nous concentrer ici sur l'action « EDIT », l'édition d'un utilisateur, la logique est la même pour les autres actions.

Premièrement on récupère l'utilisateur courant grâce au token.

Si l'utilisateur est « anonyme », c'est-à-dire qu'il n'est pas connecté, false est renvoyé et l'action ne peut pas être effectuée.

Dans notre projet cette vérification se fait par d'autres couches de l'application, avec les « access_control » un utilisateur doit être connecté et avoir au minimum le rôle Admin pour accéder à la page d'édition des utilisateurs.

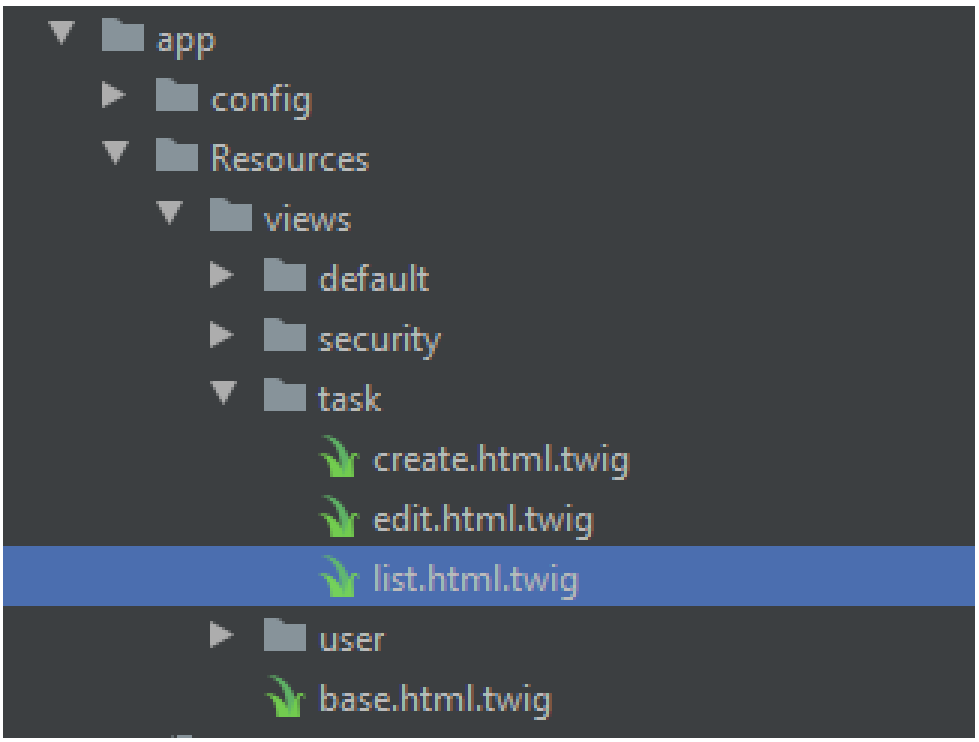
Après en fonction de l'attribut, ici « EDIT » on effectue des actions.

Si l'utilisateur connecté est le sujet, c'est-à-dire qu'il veut éditer son propre compte, on renvoie true, la permission est accordée.

En fin de compte, un utilisateur pourra éditer une Tâche, si c'est lui-même qui la créé, un administrateur ne pourra pas éditer une tâche qui n'est pas à lui, même si son rôle sera beaucoup plus élevé.

C – UTILISATION DU VOTER DANS UN FICHIER TWIG

1 – OÙ SE SITUE LE FICHIER ?



2 – CAS D'UTILISATION

```
{% if is_granted('EDIT',task)%}
    <form action="{{ path('task_edit', {'id' : task.id }) }}">
    <button class="btn btn-warning btn-sm pull-right">Modifier</button>
</form>
    {% endif %}
```

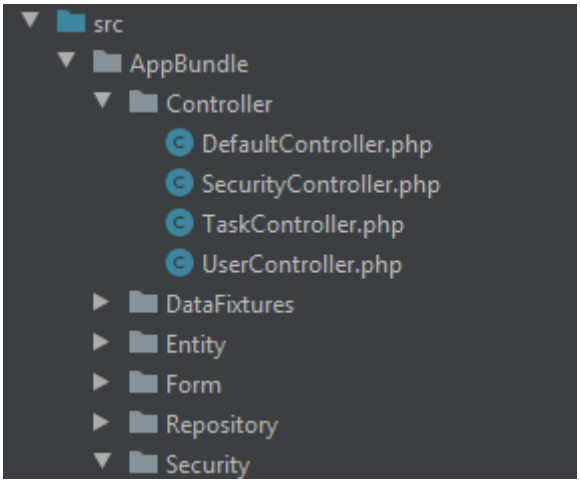
Ici on utilise très simplement notre « voter » pour afficher uniquement les utilisateurs que l'on a le droit d'éditer en fonction des permissions établies dans le « voter ».

Sans l'utilisation d'un « voter », il aurait fallu plusieurs « if » imbriqués.

De même que précédemment, le premier paramètre est le type d'action (ou attribut), le second le sujet.

V – LE SECURITYCONTROLLER

1 – OÙ SE SITUE LE FICHIER ?



2 – LA MÉTHODE LOGIN()

```php
/**
 * @Route("/login", name="login")
 */
public function loginAction(Request $request)
{
    $authenticationUtils = $this->get('security.authentication_utils');

    $error = $authenticationUtils->getLastAuthenticationError();
    $lastUsername = $authenticationUtils->getLastUsername();

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

Le système de sécurité gère automatiquement la soumission du formulaire pour vous. Si l'utilisateur soumet un nom d'utilisateur ou un mot de passe non

valide, ce contrôleur lit l'erreur de soumission du formulaire à partir du système de sécurité, afin de pouvoir l'afficher à nouveau.

En d'autres termes, votre travail consiste à afficher le formulaire de connexion et toutes les erreurs de connexion qui pourraient s'être produites, mais le système de sécurité lui-même se charge de vérifier le nom d'utilisateur et le mot de passe soumis et d'authentifier l'utilisateur.

Références : https://symfony.com/doc/3.4/security/form_login_setup.html

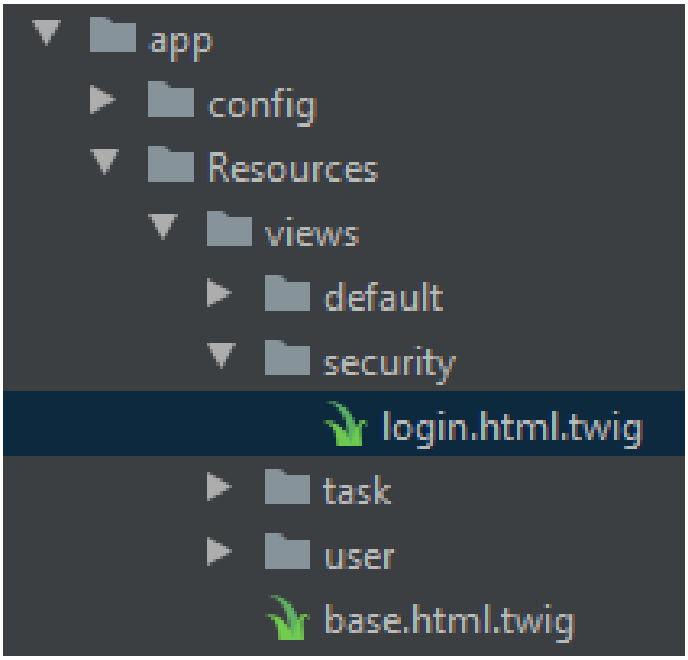
3 – LA MÉTHODE LOGOUT()

```php
/**
 * @Route("/logout", name="logout")
 */
public function logoutCheck()
{
    // This code is never executed.
}
```

La fonction logout() a simplement besoin d'être déclaré pour que Symfony puisse s'occuper de la déconnexion.

Référence : https://symfony.com/doc/current/security.html#logging-out

1 – OÙ SE SITUE LE FICHIER ?



2 – LE FICHIER TWIG

Ce formulaire est fourni dans la documentation officielle de symfony 3.4:

- L'affichage des erreurs s'il y en a.

- Les champs de notre formulaire « username » et « password » avec leurs labels respectifs.

- Le champ du token caché.

- Le bouton pour se connecter.

Référence : https://symfony.com/doc/3.4/security/form_login_setup.html