TODO LIST

Système d'authentification

Introduction

L'utilisation de l'application est strictement réservée aux utilisateurs enregistrés et authentifiés. Pour s'authentifier, un système de formulaire de connexion a été implémenté. Les utilisateurs doivent saisir leur identifiant et leur mot de passe.

Seules les pages concernant la création d'un compte ainsi que la page de « login » sont accessibles sans contrainte.

Au-delà de ce système de connexion, certaines actions de l'application sont réglementées en fonction des droits dont disposent les utilisateurs. Ces droits dépendent d'une part du rôle dont ils disposent et d'autre part de liens qu'ils disposent avec les différentes tâches.

Les utilisateurs : La classe *User*

Les données utilisateurs sont stockées en base de données dans une table *user*. Chaque entrée de la table *user* représente donc un utilisateur. Pour manipuler les données, Symfony utilise la bibliothèque Doctrine qui ajoute une dose d'abstraction dans les opérations de gestion des bases de données. En effet, plutôt que de manipuler directement les données avec requêtes SQL, Doctrine utilise la représentation objet des données. La classe *User* est la représentation objet de la table *user* en base de données. Chaque entrée, chaque utilisateur est représenté par une instance de la classe *User*. Symfony se sert donc d'objets, d'instances de la classe *User* pour procéder à l'authentification. Néanmoins, pour utiliser cette classe en tant que représentation des utilisateurs des données de connexion, il est nécessaire qu'elle implémente une interface spécifique : *UserInterface*.

Parmi les méthodes à implémenter, il y a notamment :

- **getUsername()**

```
1. public function getUsername()
2. {
3.     return $this->username;
4. }
```

Cette méthode renvoie l'identifiant qui sert lors de l'authentification. En l'espèce, notre identifiant est la propriété *username*. Toutefois il est possible de choisir très simplement un autre identifiant. Par exemple, l'adresse email. Dans ce cas, la méthode doit retourner la propriété *email*.

- **getRoles()**

```php
public function getRoles()
{
    $roles = $this->roles;
    $roles[] = "ROLE_USER";

    return array_unique($roles);
}
```

Renvoi un tableau contenant les rôles dont dispose l'utilisateur. Par défaut, tout utilisateur aura au moins le rôle « User ». Pour chaque rôle, des droits sont associés. C'est le système d'autorisation qui utilise les rôles pour gérer les différents droits dont dispose chaque utilisateur. 2 Rôles sont disponibles dans notre application : *ROLE_USER* et *ROLE_ADMIN* .

- **getPassword()**

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

Cette méthode retourne le mot de passe utilisé pour l'authentification.

- **getSalt()**

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

Retourne l'encodeur utilisé pour le cryptage du password. Si l'encodeur est défini sur « auto » dans le fichier *security.yaml,* il faut alors que la méthode retourne *null*. Dans le cas contraire, il faudra retourner le bon encodeur.

- **eraseCredentials()**

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

Pour supprimer des données temporaires sensibles qui pourraient être contenues dans l'objet *User*. En l'espèce, cette fonctionnalité n'étant pas utilisé, son contenu est donc *null*.

Par ailleurs, il reste nécessaire de déclarer cette classe au système de sécurité de symfony. Pour se faire, il convient de paramétrer le fichier *security.yaml*.

Le fichier security.yaml

Ce fichier centralise toute la configuration du composant Security de Symfony. Il se trouve dans le dossier : */config/packages*.

Ce fichier comporte plusieurs sections :

providers

Sert à indiquer à Symfony où trouver les informations utilisateurs pour l'authentification :

```
1.      providers:
2.          doctrine:
3.              entity:
4.                  class: App\Entity\User
5.                  property: username
```

On y trouve :

- Le nom du provider : *doctrine*
- La classe qui représente les utilisateurs : *App\Entity\User*
- La propriété de la classe qui représente l'identifiant : *property : username*

encoders

On indique l'algorithme utilisé pour les mots de passe ainsi que la classe sur laquelle il doit travailler. En mettant *algorithm : auto*, on laisse le système choisir le meilleur algorithme disponible.

```
1.      encoders:
2.          App\Entity\User:
3.              algorithm: auto
4.
```

firewalls

L'authentification c'est le mécanisme qui permet de savoir qui est l'utilisateur. C'est le rôle du firewall. Il est choisi et opère à chaque requête. Il permet de restreindre l'accès à certaines parties de l'application sous condition d'authentification (et non la gestion des droits). Plusieurs firewalls peuvent coexister, en l'espèce il y en 2 :

```
1.   firewalls:
2.          dev:
3.              pattern: ^/(_(profiler|wdt)|css|images|js)/
4.              security: false
5.          main:
6.              anonymous: lazy
7.              provider: doctrine
8.              pattern: ^/
9.
10.             form_login:
11.                 login_path: login
```

```
                check_path: login_check
                always_use_default_target_path:  false
                default_target_path:  /

            logout:
                path: logout
                target: login
```

- *dev* : sert uniquement pour éviter de restreindre l'accès à des fichiers de Symfony. On ne doit pas le toucher.
    - *Security : false* signifie que l'accès est libre.
- *main* : C'est le firewall utilisé par l'application.
- *provider : doctrine :* On indique le nom du provider utilisé.
- *anonymous : lazy* signifie que les utilisateurs non authentifiés seront identifiés comme anonymes et que l'accès à ce niveau ne leur est pas empêché.

Le firewall indique la route pour le formulaire d'authentification dans la section form_login :

- *Form_path : login* : La route du formulaire de login.
- *check_path : login_check* : La route de l'authenticator.

Le firewall indique aussi pour le logout :

- *path : logout :* Le nom de la route pour le logout.
- *target : login :* La redirection après déconnexion.

access_control

C'est le système d'autorisation. Il permet de déterminer si un utilisateur a les droits. Il gère l'accès aux pages ou actions du site en fonction des rôles possédés par l'utilisateur authentifié. Toutefois, il est possible de se passer de cette rubrique ou d'utiliser en complément les annotations. C'est ce système d'annotations qui a été utilisé pour l'application car il est plus pratique.

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

En face de chaque action on indique l'annotation @IsGranted et les droits requis :

```
/**
 * @Route("/tasks", name="task_list")
 * @IsGranted("ROLE_USER")
 */
```

Pour un contrôle encore plus précis, on va créer un système de *Voters*.

Role hierarchy

Pour définir une hiérarchie dans les rôles :

```
role_hierarchy:
        ROLE_ADMIN: ROLE_USER
```

ROLE_ADMIN : ROLE_USER signifie que l'administrateur possède ses propres droits ainsi que ceux d'un utilisateur *User*.

Le controller security

On a le controller *SecurityController* pour l'authentification et la déconnexion. On le trouve dans */src/Controller*. 3 routes :

```
<?php

namespace App\Controller;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="login")
     */
    public function loginAction(Request $request, AuthenticationUtils $authenticatio
nUtils)
    {
        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUsername = $authenticationUtils->getLastUsername();

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

    /**
     * @Route("/login_check", name="login_check")
     */
    public function loginCheck()
    {
        // This code is never executed.
    }

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

- *Login* : pour aller au formulaire de connexion
- *Login_check :* l'action jamais exécutée, récupérée par les listeners, pour traiter le formulaire. Des listeners vont traiter les informations de connexion et vont permettre l'authentification grâce à l'authenticator de base de Symfony. Il est possible de créer son propre authenticator.
- *Logout :* l'action qui sert à se déconnecter. Fonctionne à l'aide de listeners.

Les Voteurs

Pour un contrôle plus précis des autorisations, on utilise les *Voteurs* : Classes qui étendent la classe *Voter*. En effet, grâce aux *Voteurs*, on peut gérer des autorisations qui dépendent d'autres facteurs, on crée des droits différents (indépendants de *ROLE_ADMIN* et *ROLE_USER)*.

En l'espèce, les autorisations ne dépendent plus seulement de tel ou tel rôle mais, par exemple, de l'utilisateur qui est lié à une tâche :

```php
<?php

namespace App\Security\Voter;

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

class TaskRemoveVoter extends Voter
{
    private $security;

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

    protected function supports($attribute, $subject)
    {
        return $attribute === 'REMOVE'
            && $subject instanceof \App\Entity\Task;
    }

    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)

    {
        $user = $token->getUser();
        if (!$user instanceof UserInterface) {
            return false;
        }
        // To restrict tasks removal to only their creator
        if ($user === $subject->getUser()) {
            return true;
        }
        // To allow admins to remove anonymous tasks
```

```
36.         if (null === $subject->getUser() && $this->security-
    >isGranted('ROLE_ADMIN')) {
37.             return true;
38.         }
39.
40.         return false;
41.     }
42. }
```

La méthode *supports* définit si pour cette demande d'autorisation le *Voteur* doit être exécuté. En l'espèce, si l'action est protégée par le rôle *Remove* et si le sujet du vote et la classe *Task*, alors la méthode retourne *True* et la méthode *voteOnAttribute* sera exécutée.

```
43.     protected function supports($attribute, $subject)
44.     {
45.         return $attribute === 'REMOVE'
46.             && $subject instanceof \App\Entity\Task;
47.     }
```

On déclare le rôle dans l'annotation de l'action :

```
1.  /**
2.   * @Route("/tasks/{id}/delete", name="task_delete")
3.   * @IsGranted("REMOVE", subject="task")
4.   */
```

Si la méthode *voteOnAttribute* est exécutée, les conditions sont vérifiées dans des instructions conditionnelles :

Vérifie que l'utilisateur connecté est le même que celui qui est lié à la tâche :

```
-           // To restrict tasks removal to only their creator
-           if ($user === $subject->getUser()) {
-               return true;
-           }
```

Vérifie que l'utilisateur connecté est un administrateur et que la tâche est anonyme (non liée à un utilisateur) :

```
1.          // To allow admins to remove anonymous tasks
2.          if (null === $subject->getUser() && $this->security-
    >isGranted('ROLE_ADMIN')) {
3.              return true;
4.          }
```

En retournant *true,* l'action est autorisée et en l'espèce, la tâche est supprimée.