Documentation technique
L'authentification
Projet ToDoList

I. Introduction

L'objectif de cette documentation est d'expliquer le fonctionnement global du système d'authentification de l'application ToDoList.

Ce projet a été initialement développé sous le framework Symfony 3.1 puis migrer sous la version 5.4. Cette documentation technique s'inscrit donc dans ce contexte.

Pour rappel l'application est accessible au public (sans être « connecté ») uniquement pour la page « /login » et ce afin de permettre aux utilisateurs qui souhaitent accéder au reste de l'application de s'identifier via un formulaire (nom d'utilisateur / mot de passe). Une fois authentifié l'utilisateur peut accéder au reste de l'application en fonction de son rôle (utilisateur ou administrateur).

Le système d'authentification de l'application et la gestion des rôles des utilisateurs se fondent sur le système de sécurité du framework Symfony. Pour rappel ce dernier repose sur deux principes : l'authentification et l'autorisation.

II. L'authentification

L'Authentification permet à l'utilisateur de s'identifier. Sous Symfony, c'est le **firewall** qui prend en charge l'authentification.

A. Les utilisateurs : classe User

Le framework Symfony pour effectuer l'authentification se base sur les instances de la classe « User ». Cette classe « User » représente donc les différents utilisateurs de l'application qui seront au final enregistrés dans la base de données via la table « user ». Le lien entre cette table et cette classe étant réalisé par l'ORM Doctrine qu'utilise le framework Symfony.

Il est à noter que la classe « User » afin de pouvoir correctement gérer les utilisateurs implémente 2 interfaces « UserInterface » et « PasswordAuthenticatedUserInterface »

```
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
```

Une fois qu'un utilisateur est authentifié, vous pouvez y accéder dans un contrôleur via la méthode « getUser () » ou dans une vue Twig via la variable globale « app.user ».

B. Le fichier security.yaml

Ce fichier permet de centraliser toute la configuration du composant Security de Symfony. Il se trouve dans le dossier « config/packages ».

1. Providers

Pour permettre que chaque utilisateur enregistré dans l'application soit unique la classe « User » utilise la propriété « username » dont l'information est référencée dans le fichier « security.yaml »

```yaml
providers:
    # used to reload user from session & other features (e.g. switch_user)
    app_user_provider:
        entity:
            class: App\Entity\User
            property: username
```

On peut également choisir une autre propriété pour déterminer l'unicité d'un utilisateur comme son adresse mail par exemple.

2. Password hashers

L'utilisateur pour s'identifier utilise un mot de passe qui sera stocké dans la base de données. De ce fait il faut que celui-ci soit crypter grâce à un algorithme d'encodage.

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

En définissant « algorithm » sur « auto », nous laissons le framework symfony choisir le meilleur algorithme disponible.

3. Firewalls

Le firewall permet de vérifier l'identité de l'utilisateur.

```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        provider: app_user_provider
        custom_authenticator: App\Security\FormLoginAuthenticator
        # form_login:
            # "login" is the name of the route created previously
            # login_path: account_login
            # check_path: account_login
        logout:
            path: account_logout

        # 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

        access_denied_handler: App\Security\AccessDeniedHandler
```

Dans notre cas 2 firewalls ont été défini ;

- ✓ dev : sert uniquement pour éviter de restreindre, en développement, l'accès à des fichiers de Symfony. C'est pour ça que « security » peut rester à « false »
- ✓ main : C'est le firewall utilisé par l'application.
    - lazy : cela permet d'améliorer les performances de chargement des pages publiques en autorisant l'utilisation du cache. L'utilisateur ne sera chargé que si l'application en a besoin comme pour savoir si l'utilisateur loggé a le droit d'accéder au contenu qu'il demande.
    - provider : définit la configuration du provider. Ici l'entité User.
    - custom_authenticator : indique le chemin où se trouve notre authentificateur, en l'occurrence ici la classe « FormLoginAuthenticator » qui doit absolument extends la classe « AbstractLoginFormAuthenticator » .
    - Logout : le nom de la route pour la deconnexion.
    - access_denied_handler : indique le chemin d'une classe « AccessDeniedHandler » qui nous permet de définir les actions lorsqu'une exception est levée dans le cas où un utilisateur essaye d'accéder à un contenu qu'il lui est interdit.

```php
class AccessDeniedHandler extends AbstractController implements AccessDeniedHandlerInterface
{
    public function handle(Request $request, AccessDeniedException $accessDeniedException): ?Response
    {
        $this->addFlash('accesDenied', 'VOUS AVEZ ETE REDIRIGE SUR CETTE PAGE CAR : ' . $accessDeniedException->getMessage());
        return $this->redirectToRoute('homepage');
    }
}
```

Il y aura ici une redirection sur la page d'accueil avec un message d'alerte indiquant pourquoi l'utilisateur a été redirigé.

C. Le controller security

Pour manipuler un formulaire de connexion, il nous faut un contrôleur qui se trouve dans le dossier « src/controller » ! Ce contrôleur a deux fonctions : une pour la connexion et une pour la déconnexion.

1. login

```php
public function loginAction(AuthenticationUtils $authenticationUtils)
{
    //$authenticationUtils = $this->get('security.authentication_utils');

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

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

Dans cette fonction nous utilisons un service « ***AuthenticationUtils*** » dont la responsabilité est de valider l'authentification.

2. logout

```php
public function logoutCheck()
{
    throw new \LogicException('This method can be blank - it will be intercepted by the logout key on your firewall.');
}
```

Cette fonction est particulière, elle ne sera en effet jamais exécutée. Le composant **Security** a seulement besoin d'une *URL* pour effectuer la déconnexion.

III. L'autorisation

L'autorisation intervient après l'authentification et permet à un utilisateur d'accéder à certaine ressource de l'application en fonction de son rôle. Sous Symfony, c'est l'**access control** qui prend en charge l'autorisation.

A. Le Rôle

Pour permettre de définir les droits d'utilisation deux rôles ont été définie dans notre application :

- ✓ Le ROLE_USER
- ✓ Le ROLE_ADMIN

Chacun de ces rôles ayant des droits différents et/ou communs afin de répondre aux demandes du client qui ont été définies dans le cahier des charges.
Dans ce cadre-là il est à noter qu'une hiérarchie des rôles est établie dans le fichier « security.yaml» se trouvant dans le dossier « config/packages ».

```
role_hierarchy:
    ROLE_ADMIN: ROLE_USER
```

Concrètement, cela veut dire qu'un utilisateur qui a le rôle **ROLE_ADMIN** peut accéder à des pages qui nécessitent seulement le rôle **ROLE_USER**.

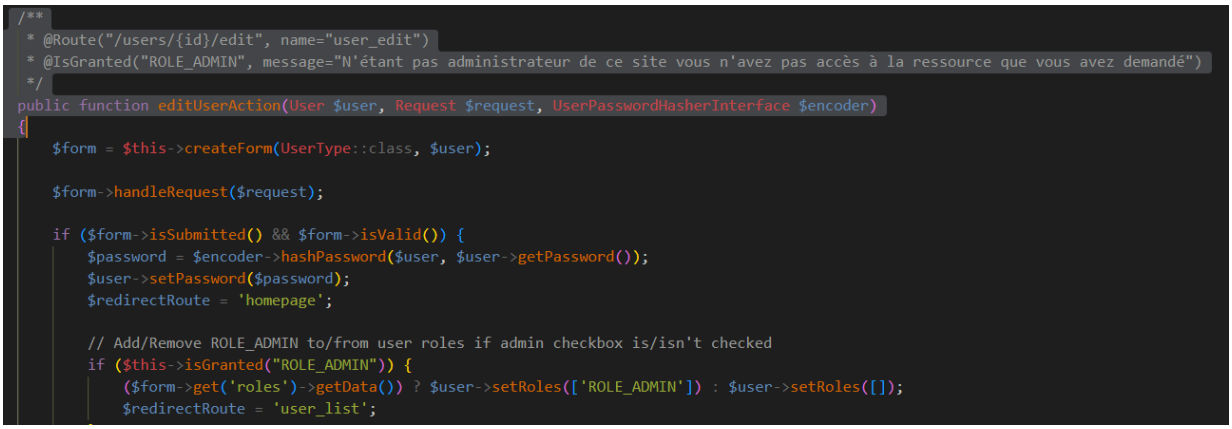
Pour les besoins futurs il sera possible de créer de nouveaux rôles si besoin. Vous pourrez nommer vos rôles comme vous le souhaitez, l'important est qu'ils commencent par « *ROLE_* ».

B. L'Access control

Pour permettre l'accès à tel ou tel ressources en fonction des rôles des utilisateurs, nous définissons principalement ces droits dans la section « acces_control » du fichier « security.yaml »

```
access_control:
    # - { path: ^/admin, roles: ROLE_ADMIN }
    # - { path: ^/profile, roles: ROLE_USER }
    - { path: ^/login, roles: PUBLIC_ACCESS }
    # - { path: ^/users, roles: ROLE_ADMIN }
    - { path: ^/, roles: ROLE_USER }
```

Cependant il est aussi possible de définir certains contrôles directement dans nos « controller » avec l'annotation commençant par « @IsGranted » comme dans la classe « UserController ».



```php
/**
 * @Route("/users/{id}/edit", name="user_edit")
 * @IsGranted("ROLE_ADMIN", message="N'étant pas administrateur de ce site vous n'avez pas accès à la ressource que vous avez demandé")
 */
public function editUserAction(User $user, Request $request, UserPasswordHasherInterface $encoder)
{
    $form = $this->createForm(UserType::class, $user);

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $password = $encoder->hashPassword($user, $user->getPassword());
        $user->setPassword($password);
        $redirectRoute = 'homepage';

        // Add/Remove ROLE_ADMIN to/from user roles if admin checkbox is/isn't checked
        if ($this->isGranted("ROLE_ADMIN")) {
            ($form->get('roles')->getData()) ? $user->setRoles(['ROLE_ADMIN']) : $user->setRoles([]);
            $redirectRoute = 'user_list';
```

Je l'utilise ici à la fois pour sécurisé l'accès à la route mais également pour afficher un champ particulier du formulaire.

IV. Documentation

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