Guide de l'authentification

La sécurité concernant l'authentification est configuré dans le fichier `config/packages/security.yaml` Vous trouverez plus d'informations dans la documentation officielle de Symfony.

L'entité User

L'entité User a été choisi pour la gestion des utilisateurs et de la sécurité. Cette classe doit implémenter l'interface `UserInterface` et donc implémenter les différentes méthodes définis dans celle-ci.
Vous trouverez plus d'informations dans la documentation officielle de Symfony
Dans notre cas, cette classe a déjà été implementée :

```
# src/Entity/User.php
/**
 * @ORM\Entity(repositoryClass=UserRepository::class)
 * @UniqueEntity("email", message="Email deja utilisé")
 */
class User implements UserInterface
{
```

L'Authentification

La gestion de l'authentification est gérée grâce à l'utilisation du MakerBundle de symfony et à la commande `php bin/console make:auth`, elle génère :

```
# src/Controller/SecurityController.php
Le controlleur de securité avec les routes login/logout
# src/Security/LoginFormAuthenticator.php
La classe Guard authenticator qui traite le login submit
# config/packages/security.yaml
Elle met à jour le fichier de sécurité pour activer le Guard authenticator
# templates/security/login.html.twig
Création du template de login qui étend de base.html.twig (personnalisable)
```

}

Le User Provider

Le provider va nous permettre d'indiquer où se situe les informations que l'on souhaite utiliser pour authentifier l'utilisateur, dans notre cas il a été généré par la commande `make:auth` :

```
# config/packages/security.yaml
providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
```

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

Les encoders

L'encoder permet de déterminer quel est l'algorithme utilisé pour l'encodage des mots de passe Utilisateur via `UserPasswordEncoderInterface`.

```
# config/packages/security.yaml
security:
    encoders:
        App\Entity\User:
            algorithm: bcrypt
```

Les Firewalls

Le pare-feu est désigné afin de restreindre les accès aux utilisateurs non authentifié.
La partie `dev` ne concerne que les accès pendant la phase de développement de l'application.
Le firewall `main` nous permet de définir les règles d'accès à l'application.
L'accès y est autorisé en anonyme, on y indique que c'est le provider "app_user_provider" qui sera utilisé ainsi que son Guard Authenticator. Ce dernier traite automatiquement les fonctions login et logout.

```
# config/packages/security.yaml
firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true
            lazy: true
            provider: app_user_provider
            guard:
                authenticators:
                    - App\Security\LoginFormAuthenticator
            logout:
                path: app_logout
```

Les Access_Control

On définit ici les routes ainsi que les accès :

- L'url /task est accessible une fois authentifié.
- L'url /admin/user n'est accessible qu'en étant authentifié avec un utilisateur ayant le rôle "ROLE_ADMIN".

- La page d'accueil du site est disponible pour tous (en 3ème position pour éviter qu'elle prenne le dessus sur les autres routes)

```
# config/packages/security.yaml
access_control:
        - { path: ^/task, roles: ROLE_USER }
        - { path: ^/admin/user, roles: ROLE_ADMIN }
        - { path: ^/, roles: IS_AUTHENTICATED_ANONYMOUSLY }
```

Les Role_Hierarchy

Cette partie permet d'établir une hiérarchie dans les rôles, dans notre cas le `ROLE_ADMIN` aura automatiquement le `ROLE_USER`.

```
# config/packages/security.yaml
role_hierarchy:
        ROLE_ADMIN: ROLE_USER
```

Le TaskVoter

Un système d'autorisation a été mis en place grâce à un Voter custom. La vérification se déclenche à l'intérieur des controlleurs, ex:

```
($this->isGranted('edit', $task))
```

Il permet de gérer les accès à certaines fonctionnalitées:

```
# src/security/TaskVoter.php
class TaskVoter extends Voter
{
    private const DELETE = 'delete';
    private const VALIDATE = 'validate';
    private const EDIT = 'edit';
```

chaque accès est définit dans cette classe:

```
switch ($attribute) {
            case self::DELETE:
                return $this->canDelete($task, $user);
            case self::VALIDATE:
                return $this->canValidate($task, $user);
            case self::EDIT:
                return $this->canEdit($task, $user);
```

Les autorisations d'accès sont définit ensuite:

`Seul l'utilisateur ayant créé la tâche peut la supprimer ou le ROLE_ADMIN si la task est anonyme:`

```
private function canDelete(Task $task, User $user)
        {
            return (
                $user === $task->getUser()
                || ($task->getUser()->getEmail() == "anonymous" && $user == $tl
            );
        }
```

`Seul l'utilisateur assigné à la tâche ou ROLE_ADMIN peut la valider:`

```
private function canValidate(Task $task, User $user)
    {
        return ($user === $task->getAssignedTo() || $this->security->isGranted(
    }
```

`Un utilisateur assigné, un ROLE_ADMIN ou l'utilisateur qui a créé la tâche peut l'éditer`

```
private function canEdit(Task $task, User $user)
    {
        return (
            $user === $task->getAssignedTo()
            || $this->security->isGranted('ROLE_ADMIN')
            || $user === $task->getUser()
        );
    }
```