DOCUMENTATION TECHNIQUE

I. Introduction

Ce document a pour objectif d'expliquer l'implémentation de l'authentification de ToDo & Co. L'authentification nécessite l'utilisation de plusieurs fichiers avec un rôle différent. L'objectif est d'ici éclaircir le processus d'authentification.

Parmi ces fichiers :

1. *config\packages\security.yaml* pour configurer le processus d'authentification
2. *src\Entity\User.php* pour la création de notre entité utilisateur et la gestion de ces utilisateurs en base de données.
3. *src\Controller\SecurityController.php* pour gérer les routes permettant la connexion et déconnexion des utilisateurs.
4. *src\Security\AppCustomAuthenticator.php* pour gérer ces routes et leur redirection en cas de succès ou d'échec de la connexion à l'application.
5. *templates\security\login.html.twig* pour la création de la page de connexion.

Nous allons détailler le process à travers ces éléments.

II. La classe User

Les utilisateurs de l'application sont enregistrés dans la base de données de l'application dans l'entité User.

L'inscription des utilisateurs se fait par ces derniers. Ils accèdent à cette possibilité par la route « /user/create » nommée « user_create ». Cette route est définie dans le fichier « App\Controller\UserController.php » sous la méthode « createAction ».

Elle utilise le formulaire « App\Form\UserType.php » qui permet de valider certaines contraintes concernant les attributs de notre classe (entité) User. Pour valider l'inscription, Doctrine vérifie :

- L'unicité de l'identifier (ici, l'email l'utilisateur, cf. la configuration du fichier security.yaml).
- La présence d'un email, d'un username et un mot de passe suffisamment fort (via PasswordStrength).

Le formulaire d'inscription valide aussi la cohérence du mot de passe et de sa confirmation.

Afin d'assurer la protection des données utilisateurs, leurs mots de passe sont encodés grâce à l'utilisation de la classe « UserPasswordHasherInterface » et le password_hashers du bundle Security (cf. la configuration du fichier security.yaml).

Conformément à la demande initiale, à l'inscription, l'utilisateur peut également choisir le rôle qui lui sera attribué. (ROLE_ADMIN ou non en plus du ROLE_USER de base)

Après vérification de ces éléments, les données si validées sont enregistrées dans la table « user » de la base de données de l'application.

17	106	chroye@hotmail.fr	["ROLE_ADMIN"]	$2y$13$1bkBKeEt9aQj1uz2BgYcWO5dDKNchaj/Rlq5ofiaeSSHZqSIK8vJy	charleroyer
18	107	libarr@hotmail.com	[]	$2y$13$SY4QolR8VErqTGEdfvKK.uxAbUkuxxYGjW7W8g4OIX6Oofevmw5ua	lilliaba
19	108	dagera@yahoo.fr	[]	$2y$13$ukBrFXpd4uiXDzUz4yvT3OxZxOX.DXeOjyY23bM2jRcPXE/5ócXUe	dagerar

III. Le fichier security.yaml

Ce fichier présent dans le dossier config\packages a vocation à définir les règles d'authentification des utilisateurs et à définir leurs autorisations ou non à réaliser certaines actions sur l'application.

Comme dit précédemment, le bundle Security par l'intermédiaire du password_hashers fournit le hashage du mot de passe ainsi que sa vérification.

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

C'est également dans ce fichier que l'on indique la propriété (l'identifier) de l'entité User qui sera utilisée pour identifier notre utilisateur. Cela se passe sous la partie providers de ce fichier :

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

```yaml
main:
    lazy: true
    provider: app_user_provider
    custom_authenticator: App\Security\AppCustomAuthenticator
    logout:
        path: app_logout
        target: homepage
```

La classe « App\Security\AppCustomAuthenticator » présente dans le firewall main du fichier security.yaml est alors chargée de requêter la base de données pour trouver la présence dans la table user d'un utilisateur avec l'adresse mail renseignée à la connexion et vérifie ensuite le mot de passe attaché à cet utilisateur.

Ce fichier permet aussi de définir les accès de l'ensemble des rôles à certaines parties de l'application et de hiérarchiser les rôles des différents utilisateurs.

```yaml
role_hierarchy:
    ROLE_ADMIN: ROLE_USER
# Easy way to control access for large sections of your site
# Note: Only the *first* access control that matches will be used
access_control:
    - { path: ^/admin, roles: ROLE_ADMIN }
    - { path: ^/task, roles: ROLE_USER }
```

Ainsi les administrateurs du site peuvent accéder à la partie /admin, les utilisateurs peuvent eux accéder à la partie /task. Par la définition de rôle_hierarchy, on sait aussi que les admins héritent du rôle user et pourront ainsi réaliser toutes les actions de ces derniers. Cependant, nous préférerons utiliser les « voters » pour affiner les actions des différents rôles de l'application. (cf. V. les voters)

IV. Le SecurityController

Lorsqu'un utilisateur souhaite se connecter, il se rend sur la route « login » nommée « app_login » et située dans le fichier « App\Controller\SecurityController.php ».

Si l'utilisateur est déjà connecté, il sera alors redirigé vers la page de ses tâches. Pour se connecter, il renseigne dans le formulaire son email (ie le provider du fichier security.yaml) ainsi que son mot de passe.

V. Les voters

Les voters permettent d'accorder ou non la réalisation de certaines actions à certains utilisateurs sur certains objets.

La méthode supports de nos voters permettent de définir si conjointement l'action à réaliser est possible ET si l'objet est bien une instance de la classe sur laquelle on souhaite réaliser l'action. Dans le cas où cette combinaison est vraie, on appelle la seconde méthode : voteOnAttribute qui va définir si oui ou non l'utilisateur peut réaliser cette action sur cet objet.

Ainsi nous avons pu mettre en place les fonctionnalités comme demandées initialement :

```php
private function canEdit(TokenInterface $token, Task $task): Bool {
    $user = $token->getUser();

    return $user === $task->getAuthor();
}
```

1. Un utilisateur peut éditer une tâche s'il en est l'auteur.

```php
private function canDelete(TokenInterface $token, Task $task): Bool {
    $user = $token->getUser();

    return (in_array( needle: User::ROLE_ADMIN, $user->getRoles()) && $task->getAuthor() === null) ||
        $user === $task->getAuthor();
}
```

2. Un utilisateur peut supprimer une tâche s'il en est l'auteur ou si la tâche est anonyme (auteur *null)* à condition qu'il soit ADMIN.

```php
private function canEditRoles(TokenInterface $token): Bool {
    $loggedUser = $token->getUser();

    return in_array( needle: User::ROLE_ADMIN, $loggedUser->getRoles());
}
```

3. Un utilisateur peut éditer le rôle d'un utilisateur s'il est ADMIN.

```php
private function canEditUser(TokenInterface $token, User $user): Bool {
    $loggedUser = $token->getUser();

    return $loggedUser === $user;
}
```

4. Un utilisateur peut éditer le profil d'un user s'il s'agit de son propre compte.

Grâce à ces voters, on pourrait imaginer implémenter de nouvelles fonctionnalités, par exemple la possibilité de voir ou d'éditer des tâches qui seraient définies comme publique. (en adaptant l'entité Task)