TODO & CO

TODOLIST

DOCUMENTATION TECHNIQUE

L'AUTHENTIFICATION

Parcours OPENCLASSROOMS

Développeur d'Application PHP / Symfony

Projet d'étude 8

Développeur : Mathieu Bonhommeau

Le système d'authentification de l'application TODOLIST est basé sur le composant Security de Symfony

SOMMAIRE :

1. L'ENTITE USER ET LA BASE DE DONNEES :

Pour utiliser le composant Security de Symfony, il faut que la classe User *(App\Entity\User)* implémente UserInterface *(Symfony\Component\Security\Core\User\UserInterface)*.

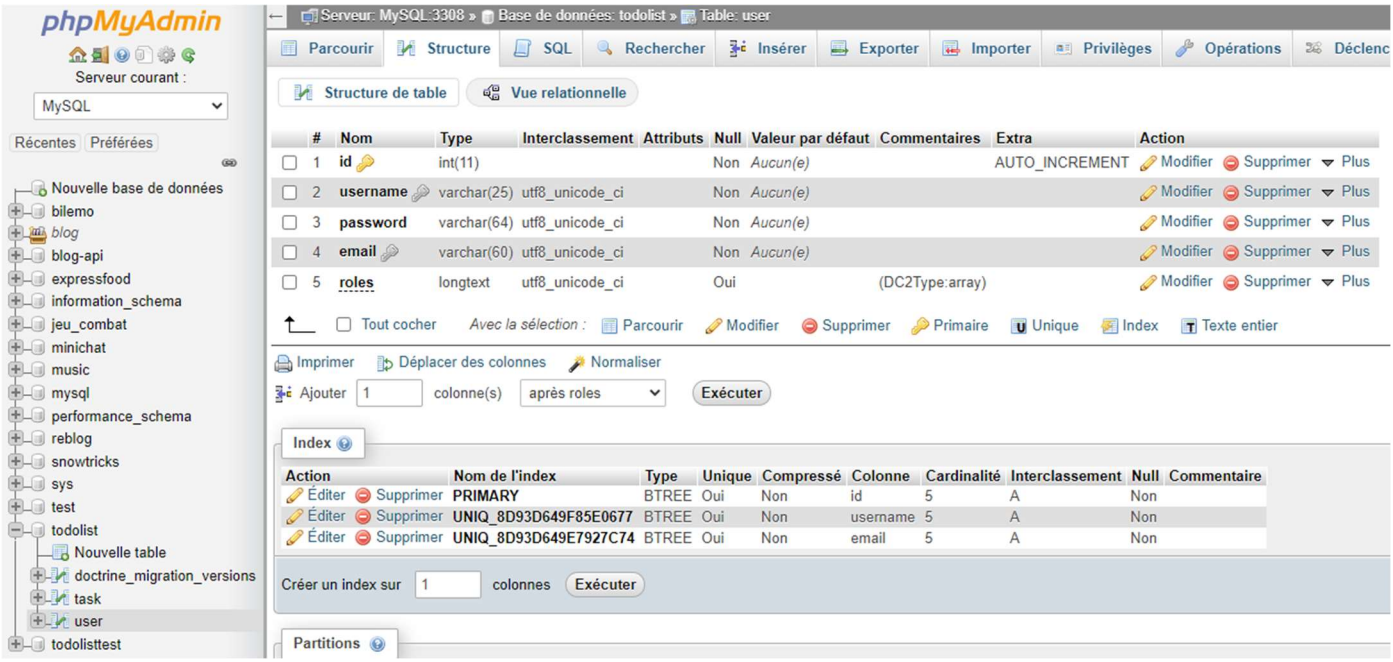
UserInterface force à intégrer dans notre classe les méthodes suivantes :

- getUsername()
- getPassword()
- getRoles()
- getSalt()
- eraseCredentials()

Nous avons donc un nom d'utilisateur ($username), un mot de passe ($password) et un rôle ($roles). Nous avons ajouté un également un email ($email) et un id unique ($id).

(Les méthodes getSalt() et eraseCredentials() sont présentes mais ne sont pas utilisées dans ce projet)

Cette entité User est synchronisée avec la base de données via le composant Doctrine. On retrouve donc une BDD todolist avec une table user et ses colonnes correspondantes dans phpMyAdmin :



Si besoin, l'ajout de colonnes est possible pour pouvoir stocker des informations supplémentaires (ex : num de tél, adresse, ...).

Pour ce faire, il faudra, au préalable ajouter les propriétés correspondantes dans l'entité **User** ainsi que leurs getters et setters (ex : $phone, $address, ...) et effectuer une migration via Symfony à l'aide de a CLI :

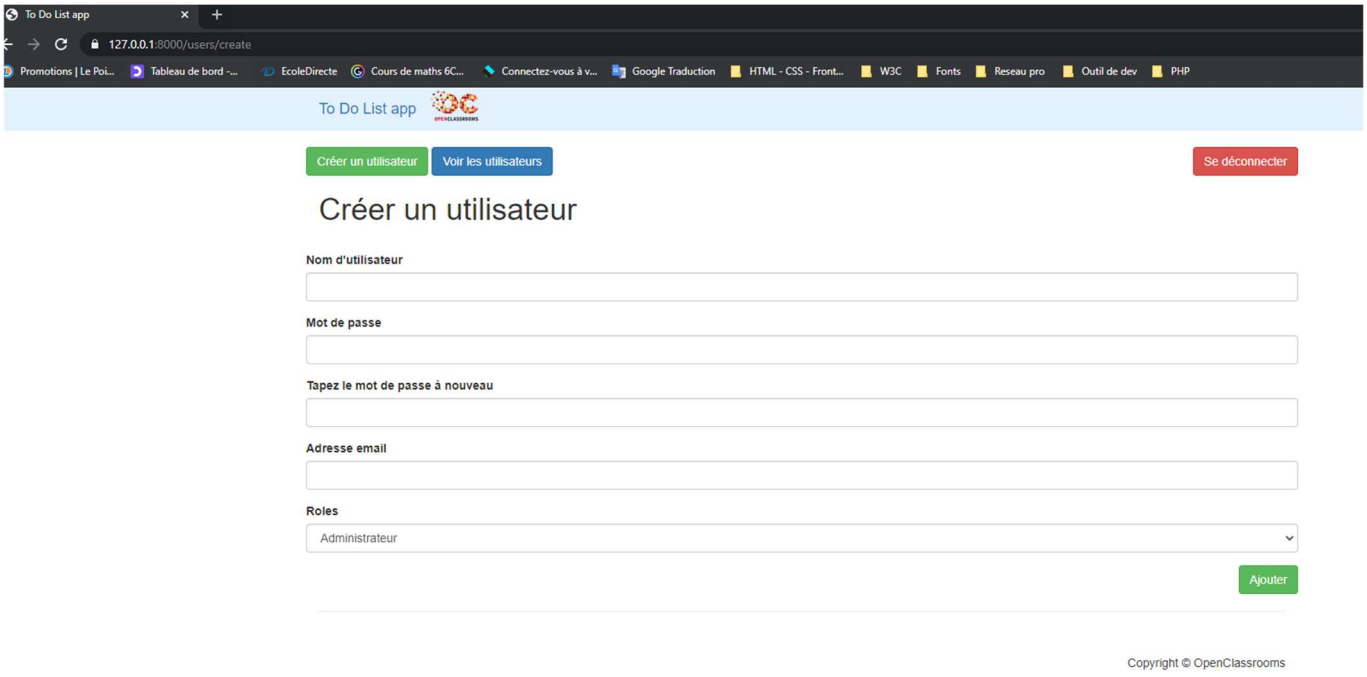
```
php bin/console create:migration

php bin/console doctrine:migrations:migrate
```

ATTENTION : Il ne faut jamais modifier la structure de la base de données directement dans phpMyAdmin sous peine de créer des conflits avec Symfony. Il faut toujours passer par le framework avec des migrations ou autre.

2. L'ENREGISTREMENT D'UN UTILISATEUR

Pour enregistrer un utilisateur, il faut passer par le formulaire correspondant présent sur le template twig : user/create.html.twig accessible via la route /users/create définie dans le UserController avec la méthode createAction().



createAction() prend en paramètre un objet de type UserPasswordEncoderInterface (Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface) qui va permettre d'encoder le mot de passe avant de l'enregistrer en base de données.

Voici la ligne de code qui gère cet aspect dans la méthode createAction() :

```
$password = $encoder->encodePassword($user, $user->getPassword());
```

(Le type d'encodage est défini dans la partie 4 de ce document)

	id	username	password	email
primer	3	matanna	$2y$13$zl4HnvVeY.ywEvOZ6JpWmeqiRFvLp36eAr9Nvlr0etw...	matanna@orange.fr
primer	4	bibiche23	$2y$13$IxmGqQrHfXu9KPBhhT7XuuydbaD5w01DU8H01WrCLKd...	bibiche23@free.fr
primer	5	matanna23	$2y$13$EFyWuW3KCOyR7SuD5m49uOOKMf/4D/xCpF1INi1t0Vr...	matanna23@orange.fr

Dans ce formulaire, on retrouve un 2ème champ password. Celui-ci n'est pas enregistré en base de données et n'est utilisé que pour valider le premier mot de passe renseigné.

On retrouve également un champ rôles qui va nous permettre de gérer les autorisations de l'utilisateur créé.
(voir partie 4 de ce document)

On peut, si besoin, ajouter des champs supplémentaires en fonction de ce que l'on souhaite enregistrer en base de données. (ex : num de tèl, ...).
(conf. Partie 1)

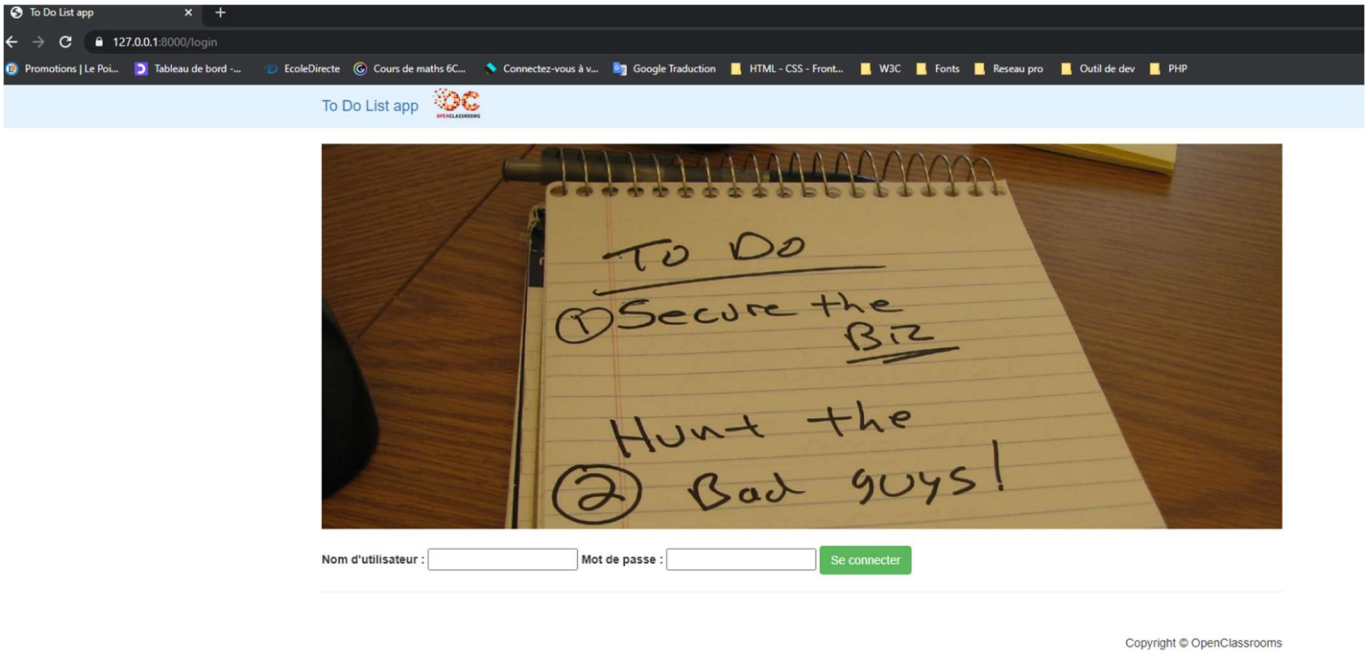
Dans ce cas, il faudra modifier le fichier UserType.php (App\Form\UserType) qui gère les champs du formulaire.

```
class UserType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('username', TextType::class, ['label' => "Nom d'utilisateur"])
            ->add('password', RepeatedType::class, [
                'type' => PasswordType::class,
                'invalid_message' => 'Les deux mots de passe doivent
correspondre.',
                'required' => true,
                'first_options'  => ['label' => 'Mot de passe'],
                'second_options' => ['label' => 'Tapez le mot de passe à
nouveau'],
            ])
            ->add('email', EmailType::class, ['label' => 'Adresse email'])
            ->add('roles', ChoiceType::class, [
                'required' => true, 'multiple' => false, 'expanded' => false,
                'choices' => ['Administrateur' => 'ROLE_ADMIN','Utilisateur' =>
'ROLE_USER']
            ])
        ;
```

3. LE SECURITYCONTROLLER ET LE FORMULAIRE DE LOGIN

Pour implémenter l'authentification, il faut créer un Controller qui est appelé SecurityController pour gérer les routes correspondantes :

- /login
  La méthode utilisée pour cette route est loginAction() et renvoie sur le formulaire de Login via le template twig : security/login.html.twig.



- /login_check
  La méthode utilisée pour cette route est loginCheck().
  Cette route est appelée lors de l'envoie du formulaire de login et est traitée directement par le composant Security de Symfony pour valider ou non la connexion (vérifiation du username et du password).
  Aucun code n'est à ajouter dans cette méthode.

Dans le template twig : security/login.html.twig, on retrouve cette route au niveau de l'action du formulaire de login :

```
<form action="{{ path('login_check') }}" method="post">
        <label for="username">Nom d'utilisateur :</label>
        <input type="text" id="username" name="_username" value="{{ las
        t_username }}" />

        <label for="password">Mot de passe :</label>
        <input type="password" id="password" name="_password" />

        <button class="btn btn-
        success" type="submit">Se connecter</button>
    </form>
```

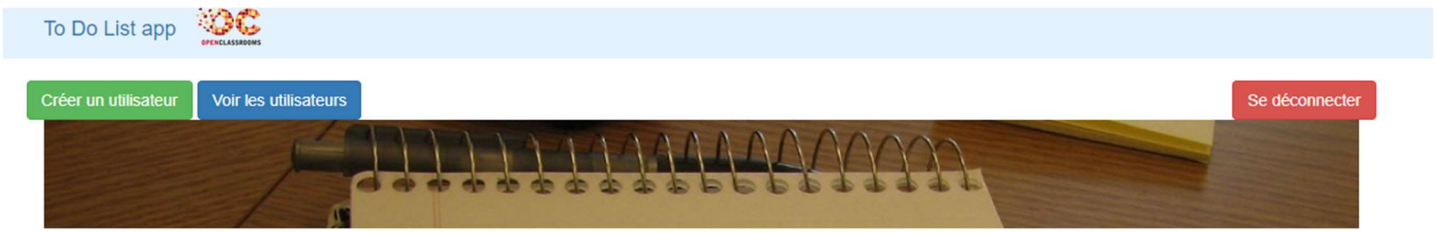
- /logout

La méthode utilisée pour cette route est logoutCheck().
Cette route est appelée lors de la déconnexion de l'utilisateur.
Comme pour /login_check, elle est gérée directement par Symfony.
Aucun code n'est à ajouter dans cette méthode.

Dans le template de base twig : base.html.twig, on retrouve cette route au niveau du lien de déconnexion :

```
{% if app.user %}
        <a href="{{ path('logout') }}" class="pull-right btn btn-
        danger">Se déconnecter</a>
{% endif %}
```

Ce lien s'affiche uniquement si l'utilisateur est connecté (*if app.user*)



4.LE FICHIER SECURITY.YAML :

Le fichier security.yaml est le fichier de configuration du composant Security de Symfony.

Voici les différents éléments qui le compose :

- Encoders :
  Permet de définir quel encodage est utilisé pour les mots de passe. Nous avons choisi bcrypt pour le projet. Ceci peut être modifié si besoin.

```
encoders:
        App\Entity\User: bcrypt
```

  On spécifie bien que l'encodage est à faire sur l'entité User.

  Lors de l'authentification, Symfony encodera le mot de passe renseigné par l'utilisateur et le comparera à celui enregistré en base de données.

- Providers :
  Permet de définir ou sont les données des utilisateurs et comment le système doit les reconnaitre.

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

  Dans notre cas, le système doit aller chercher les utilisateurs via Doctrine, au niveau de l'entité User et peut les reconnaitre grâce à la propriété username.

Si l'on souhaite que le système les reconnaisse grâce à leur email, on peut modifier le fichier de la manière suivante :

```
providers:
        doctrine:
            entity:
                class: App:User
                property: email
```

Il faudra cependant s'assurer que la propriété email possède une contrainte d'unicité.

- Firewalls :
  Permet de définir quelle partie de l'application est protègée et comment.
  Dans notre cas, l'authentification est nécessaire pour toute l'application :

```
firewalls :
        ...
        main:
            ...
            pattern: ^/
```

Si besoin, ce paramètre peut être modifié :

```
pattern : ^users/
```

Ici l'application sera protégée uniquement pour les routes /users (users/create, users/{id}/edit, ...)

On définit ensuite les paramètre correspondant à l'authentification :

```
firewalls:
        ...
        main:
            ...
            form_login:
                login_path: login
                check_path: login_check
                ...
```

login_path correspond à la route du formulaire de login.
login_check correspond à la route servant à vérifier les identifiants et à autoriser ou non la connexion

Enfin, on retrouve les paramètres correspondants à la déconnexion :

```
firewalls:
        ...
        main:
            ...
            logout:
                path: /logout
                target: /
```

path définit la route correspondant à la déconnexion.
target définit la redirection après la déconnexion.
Ici, l'utilisateur déconnecté est renvoyé sur la page d'accueil qui redirigera automatiquement sur la page de login étant donné que toute notre application est protégée.

- roles_hierarchy :
  Permet de définir les rôles pour les utilisateurs ainsi que leur hiérarchie :

```
role_hierarchy:
        ROLE_ADMIN: [ROLE_USER, IS_AUTHENTICATED_ANONYMOUSLY]
        ROLE_USER: [IS_AUTHENTICATED_ANONYMOUSLY]
```

Dans notre cas, ROLE_ADMIN a tous les droits.
IS_AUTHENTICATED_ANONYMOUSLY représente l'utilisateur non identifié.

L'ajout de rôles est possible :

```
role_hierarchy:
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_USER,
IS_AUTHENTICATED_ANONYMOUSLY]
        ...
```

Dans ce cas, nous pourrons restreindre les droits de ROLE_ADMIN si besoin.

- access_control :

  Permet de définir qui à accès à quoi en fonction du rôle qui lui est attribué.

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

  Ici, la seule page qui est accessible par tout le monde est la page de login.
  Ensuite, pour accéder au reste de l'application, il faut se connecter.

  Les pages correspondants à la gestion des utilisateurs sont cependant restreintes aux utilisateurs enregistrés avec un rôle Administrateur.

  Evidemment, ceci est modifiable si nécessaire.