

AMÉLIOREZ UNE APPLICATION EXISTANTE DE TODO & CO

PROJET 8

DÉVELOPPEUR D'APPLICATION - PHP / SYMFONY

DOCUMENTATION TECHNIQUE

Sommaire

1. Présentation Projet.....	3
1.1 Technologies	3
1.2 Librairies	3
2. Paramétrage du site	3
2.1 Introduction	3
2.2 Fichiers d'authentification.....	3
2.3 L'entité User	3
2.4 Les Providers	4
2.5 Les encoders.....	4
2.6 Les Firewalls	4
2.7 Les Access_Control	5
2.8 Les Role_Hierarchy	5
3. Les bases de données.....	5
3.1 Modifications de la base de données.....	5
3.2 Stockage des utilisateurs.....	5
4. Fonctionnement de l'authentification	6
4.1 Page authentification.....	6
4.2 Soumission du formulaire	6
4.3 Redirection de l'utilisateur	6

1. Présentation Projet

1.1 Technologies

Le projet a été mis à jour de la version 3 vers une version de Symfony plus récente et stable : v5.4.
Pour le bon fonctionnement du projet, la version 7.3.2 de PHP est requise.

1.2 Librairies

Les libraires sont toutes installés par Composer, et sont donc visible sur le site <https://packagist.org>.
Toutes les librairies sont listées dans le fichier composer.json à la racine du projet, il faut pas hésiter à faire des mises à jours, car cela peut corriger de potentielles failles de sécurité. Cependant, une consultation de la documentation est obligatoire pour éviter toutes complications ! Par exemple un conflit de dépendance.

2. Paramétrage du site

2.1 Introduction

La sécurité concernant l'authentification est configurée dans le fichier config/packages/security.yaml.

[Documentation officielle de Symfony](#), pour plus d'information

2.2 Fichiers d'authentification

Type	Fichier	Description
Configuration	config/packages/security.yaml	Configuration du processus d'authentification
Entité	src/Entity/User.php	Entité utilisateur
Contrôleur	src/Controller/SecurityController.php	Contrôleur connexion/déconnexion
Authentification	src/Security/LoginFormAuthenticator.php	Méthodes du processus d'authentification de l'application
Vue	templates/security/login.html.twig	Template du formulaire de connexion

2.3 L'entité User

Avant toute de chose, il est nécessaire d'avoir défini une entité qui représentera l'utilisateur connecté. Cette classe doit implémenter l'interface « UserInterface » et donc implémenter les différentes méthodes définies dans celle-ci. Dans ce cas-ci, cette classe a déjà été implémentée et se situe dans le fichier « src/Entity/User.php ».

Le moyen le plus simple de générer une classe d'utilisateurs consiste à utiliser la commande **make:user** du **MakerBundle** :

2.4 Les Providers

Un provider va nous permettre d'indiquer où se situe les informations que l'on souhaite utiliser pour authentifier l'utilisateur. Ce provider sait comment (re)charger des utilisateurs à partir d'un stockage (par exemple une base de données) sur la base d'un "identifiant d'utilisateur" (par exemple ici le « **nom d'utilisateur** »). La configuration ci-dessus utilise Doctrine pour charger l'entité Utilisateur en utilisant la propriété « **username** » comme "**identifiant d'utilisateur**".

```
providers:
  app_user_provider:
    entity:
      class: App\Entity\User
      property: username
```

2.5 Les encoders

Un encoder va simplement nous permettre de déterminer quel est l'algorithme que l'on souhaite utiliser lors de l'encodage d'une certaine information dans une certaine entité. L'encryptage est en mode automatique pour le moment, ce qui correspond au meilleur choix pour la version de Symfony du site. Il est bien entendu possible de changer le mode dans le fichier de configuration « security.yaml » dans le bloc « **password_hashers** ».

```
password_hashers:
  # Use native password hasher, which auto-selects and migrates the best
  # possible hashing algorithm (which currently is "bcrypt")
  Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
```

Maintenant que Symfony sait comment vous voulez hacher les mots de passe, vous pouvez utiliser le service **UserPasswordHasherInterface** pour le faire avant d'enregistrer vos utilisateurs dans la base de données :

```
public function createAction(Request $request, EntityManagerInterface $em, UserPasswordHasherInterface $passwordHasher)
{
    $user = new User();
    $form = $this->createForm(UserType::class, $user);

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // $em = $this->getDoctrine()->getManager();
        // $password = $this->container->get('security.password_encoder')->encodePassword($user, $user->getPassword());
        // $user->setPassword($password);

        // hash the password (based on the security.yaml config for the $user class)
        $hashedPassword = $passwordHasher->hashPassword(
            $user,
            $user->getPassword()
        );
    }
}
```

2.6 Les Firewalls

Un firewall va définir comment nos utilisateurs vont être authentifiés sur certaines parties du site. Le firewall **dev** ne concerne que le développement ainsi que le profiler et ne devra à priori pas être modifié. Le firewall **main** englobe l'entièreté du site à partir de la racine défini via **pattern**: `^/`, l'accès y est autorisé en anonyme, C'est-à-dire sans être authentifié, on y indique que c'est le provider "doctrine" qui sera utilisé. Afin de s'authentifier, on définit un formulaire de connexion via **form login**: où sont indiqués le nom des routes correspondant à ce formulaire, la route de vérification du login ainsi que la route vers laquelle l'utilisateur devra être redirigé par défaut après son authentification.

```

firewalls:
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false

  main:
    lazy: true
    provider: app_user_provider
    form_login:
      login_path: login
      check_path: login_check
    logout:
      path: /logout
      target: /

```

2.7 Les Access_Control

Un **access_control** va définir les limitations d'accès à certaines parties du site. Dans ce cas-ci, on indique que : - L'url /login est accessible sans authentification. - L'url /users n'est accessible qu'en étant authentifié avec un utilisateur ayant le rôle "ROLE_ADMIN". - Tout le reste du site n'est accessible qu'aux utilisateurs authentifiés c-à-d ayant le rôle "ROLE_USER".

```

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

```

2.8 Les Role_Hierarchy

Un **role_hierarchy** permet de s'assurer qu'un utilisateur ayant un certain rôle aura automatiquement d'autres rôles. Dans ce cas-ci, un utilisateur possédant le rôle "ROLE_ADMIN" aura automatiquement le rôle "ROLE_USER".

```

role_hierarchy:
  ROLE_ADMIN: [ROLE_ADMIN, ROLE_USER]

```

3. Les bases de données

3.1 Modifications de la base de données

Pour un éventuel changement du nom de base de données. Il faudra changer les informations de connexion dans le fichier. env à la racine du projet.

```

DATABASE_URL="mysql://root:@127.0.0.1:3306/todolistapp?serverVersion=5.7"

```

Détail configuration BDD :

<https://symfony.com/doc/current/doctrine.html#configuring-the-database>

3.2 Stockage des utilisateurs

Les données utilisateurs sont stockés dans une base de données MySQL. Plus précisément dans la table user. Il faut savoir que le champ **Username** est unique et il est possible de rajouter un autre champ unique.

Il faudra simplement aller dans l'entité user : `src/entity/user`, et de rajouter « unique = « true » à l'annotation `@Column`.

Exemple :

```
/**
 * @ORM\Column(type="string", length=180, unique=true)
 */
private $username;
```

4. Fonctionnement de l'authentification

4.1 Page authentification

Dans un premier temps, la personne va devoir rentrer ces identifiants sur la page [login.html.twig](#) à la route [/login](#). Pour accéder à cette page, la méthode login est exécutée, sur le Controller : [SecurityController](#). Cette méthode sert à générer la page, et à envoyer des données à la vue avec un render.

4.2 Soumission du formulaire

Lorsque vous soumettez le formulaire, le système de sécurité lit automatiquement les paramètres POST `_username` et `_password`, charge l'utilisateur via le fournisseur d'utilisateurs, vérifie les informations d'identification de l'utilisateur et authentifie l'utilisateur ou le renvoie au formulaire de connexion où l'erreur peut être affichée.

4.3 Redirection de l'utilisateur

Ne laissez pas ce contrôleur vous confondre. Son travail consiste uniquement à rendre le formulaire : l'authentificateur `form_login` gèrera automatiquement la soumission du formulaire. Si l'utilisateur soumet un e-mail ou un mot de passe invalide, cet authentificateur stockera l'erreur et la redirigera vers ce contrôleur, où nous lirons l'erreur (en utilisant `AuthenticationUtils`) afin qu'elle puisse être affichée à l'utilisateur.

```
/**
 * @Route("/login", name="login")
 */
public function loginAction(AuthenticationUtils $authenticationUtils): Response
{
    // $authenticationUtils = $this->container->get('security.authentication_utils');

    // get the login error if there is one
    $error = $authenticationUtils->getLastAuthenticationError();

    // last username entered by the user
    $lastUsername = $authenticationUtils->getLastUsername();

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