

Authentication **Documentation Technique**

Résumé :

Ce document décrit comment l'implémentation de l'authentification du projet Todo & Co à été faite. Cette documentation se destine aux prochains développeurs juniors qui rejoindront l'équipe.

Tableau des Informations :

Nom du projet	ToDo & Co
Type de document	Documentation technique
Date	26/02/2021
Version	1.0
Auteur	CIRPAN Dogukan

Tableau de Révision :

Date	Auteur	Commentaire
26/02/2021	CIRPAN Dogukan	Rédaction du support

Table des matières :

Les technologies utilisées :	5
1.1. Extensions PHP requises	6
1.2. Bibliothèques Symfony utilisées	6
1.3. Gestionnaire de dépendances	6
Fonctionnement de l'application :	6
2.1. Description	6
2.2. Architecture	7
2.2.1. Configuration globale	7
2.2.2. Structure MVC	7
2.2.3. Formulaires	8
2.2.4. Fichiers publics	8
2.2.5. Configuration	8
2.2.6. Tests	9
2.2.7. Fixtures	9
2.3. Annotations	9
2.3.1. Contrôleurs	9
2.3.2. Entités	10
2.4. Modèle de données	11
2.5. Console de Symfony	11
2.6. Mises à jour	12
Authentification :	13
3.1. Fichiers concernés	13
3.2. Configuration	13
3.2.1. Cryptage des mots de passe	13
3.2.2. Passphrase	13
3.2.3. Provider	14
3.2.4. Pare-feu	14
3.2.5. Contrôle des accès	15
3.3. Stockage des utilisateurs	15
3.4. Processus d'authentification	16
3.4.1. Affichage formulaire	16
3.4.2. Soumission du formulaire	17
3.5. Gestion des rôles	19

1. Les technologies utilisées :

Le projet est basé sur le framework PHP Symfony 4.

Documentation officielle : <https://symfony.com/doc/>

La version de PHP requise est la version 7.1.3 ou supérieure.

1.1. Extensions PHP requises

- Ctype
- iconv
- JSON
- PCRE
- Session
- SimpleXML
- Tokenizer
- Xdebug

1.2. Bibliothèques Symfony utilisées

- php
- doctrine/doctrine-bundle
- doctrine/doctrine-cache-bundle
- doctrine/doctrine-migrations-bundle
- doctrine/orm
- symfony/asset
- symfony/console
- symfony/dotenv
- symfony/form
- symfony/framework-bundle
- symfony/maker-bundle
- symfony/monolog-bundle
- symfony/security-bundle
- symfony/twig-bundle
- symfony/validator

1.3. Gestionnaire de dépendances

Les dépendances sont gérées par Composer.

Documentation officielle : <https://getcomposer.org/doc/>

2. Fonctionnement de l'application :

2.1. Description

L'application a pour objet la gestion des tâches d'un ensemble d'utilisateurs. Celles-ci sont présentées sous forme de pense-bête, et comportent :

- un titre
- un auteur
- une date de création
- une description
- un statut : tâche réalisée ou à effectuer

Les utilisateurs sont gérés par un administrateur, et peuvent ajouter, modifier, supprimer des tâches ou changer leur statut.

2.2. Architecture

2.2.1. Configuration globale

La configuration globale du projet est définie dans le fichier **.env** à la racine du projet.

Ce fichier contient notamment :

- la variable définissant la bascule du passage de développement en production
- les paramètres de connexion au serveur de base de données
- la passphrase permettant le cryptage des mots de passe
- les paramètres du serveur d'envoi d'emails

Les utilisateurs sont gérés par un administrateur, et peuvent ajouter, modifier, supprimer des tâches ou changer leur statut.

2.2.2. Structure MVC

La configuration globale du projet est définie dans le fichier **.env** à la racine du projet.

Catégorie	Fichier	Description
Modèle	src/Entity/User.php	Entité utilisateur
	src/Repository/UserRepository.php	Dépôt utilisateur
	src/Entity/Task.php	Entité utilisateur
	src/Repository/TaskRepository.php	Dépôt utilisateur

Contrôleurs	src/Controller/DefaultController.php	Contrôleur page d'accueil
	src/Controller/SecurityController.php	Contrôleur authentification
	src/Controller/UserController.php	Contrôleur gestion des utilisateurs
	src/Controller/TaskController.php	Contrôleur gestion des tâches
Vues	templates/default/*	Template page d'accueil
	templates/security/*	Template authentification
	templates/task/*	Template gestion des tâches
	templates/user/*	Template gestion des utilisateurs
	templates/error/*	Template gestion des erreurs
	templates/base.html.twig	Template de base (head, body, nav)

2.2.3. Formulaires

Les formulaires définissent les champs à afficher dans les vues.

Catégorie	Fichier	Description
Utilisateur	src/Form/UserType.php	Formulaire ajout/modification d'un utilisateur
Tâches	src/Form/TaskType.php	Formulaire ajout/modification d'une tâche

2.2.4. Fichiers publics

Les fichiers accessibles depuis un navigateur sont stockés dans le dossier public.

Catégorie	Fichier	Description
Racine	public/index.php	Routeur principal
Styles	public/css/*	Feuilles de style CSS, librairies CSS des framework JQuery et Bootstrap
Javascripts	public/js/*	Scripts Javascripts, librairies JS des framework JQuery et Bootstrap
Polices	public/fonts/*	Polices utilisées dans le thème
Images	public/img/*	Images utilisées dans le thème
Code Coverage	public/code-coverage/*	Le rapport de couverture de code

2.2.5. Configuration

Les fichiers de configuration de Symfony sont stockés dans le dossier config.

Catégorie	Fichier	Description
Routes principales	config/routes.yaml	Configuration du routeur principal
Services	config/services.yaml	Configuration des services de l'application
Routes secondaires	config/routes/*	Configuration des routes des librairies
Librairies	config/package/*	Configuration des librairies

2.2.6. Tests

Les tests automatisés unitaires et fonctionnels sont stockés dans le dossier tests.
Les résultats de la couverture des tests est stockée dans le dossier test-coverage.

Catégorie	Fichier	Description
Tests unitaires	tests/Form/*	Tests unitaires des Forms
Tests fonctionnels	tests/Controller/*	Tests fonctionnels des Contrôleurs
Configuration	phpunit.xml.dist	Configuration de la librairie PHP Unit
	http://env.test/	

2.2.7. Fixtures

Les fixtures sont stockées dans le dossier src/DataFixtures.

Catégorie	Fichier	Description
Fixtures utilisateurs	src/DataFixtures/UserFixtures.php	Fixtures pour les utilisateurs
Fixtures tâches	src/DataFixtures/TaskFixtures.php	Fixtures pour les tâches

2.3. Annotations

2.3.1. Contrôleurs

La configuration des routes des contrôleurs est réalisée dans les annotations des méthodes des contrôleurs concernés.

Exemple

```
class UserController extends AbstractController
{
    /**
     * @Route("/users", name="user_list")
     */
    public function listAction()
    { ...
    }

    /**
     * @Route("/users/create", name="user_create")
     * @param Request $request
     * @param UserPasswordEncoderInterface $userPasswordEncoder
     * @return \Symfony\Component\HttpFoundation\RedirectResponse|void
     */
    public function createAction(Request $request, UserPasswordEncoderInterface $encoder)
    { ...
    }

    /**
     * @Route("/users/{id}/edit", name="user_edit", requirements={"id"="\d+"})
     */
    public function editAction(UserRepository $userRepository, $id, $request)
    { ...
    }
}
```

2.3.2. Entités

La configuration des champs utilisés par l'ORM est effectuée en annotations des attributs des entités concernées.

Exemple

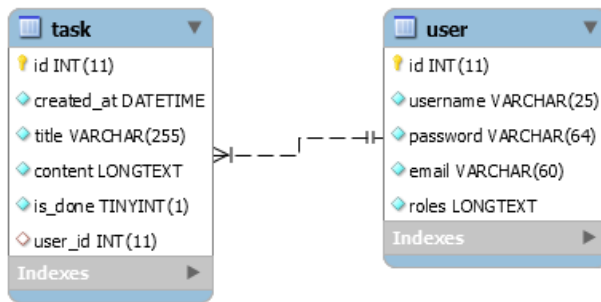
```
class User implements UserInterface
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=25, unique=true)
     * @Assert\NotBlank(message="Vous devez saisir un nom d'utilisateur")
     */
    private $username;

    /**
     * @ORM\Column(type="string", length=64)
     */
    private $password;

    /**
     * @ORM\Column(type="string", length=60, unique=true)
     * @Assert\NotBlank(message="Vous devez saisir une adresse email")
     * @Assert\Email(message="Le format de l'adresse n'est pas valide")
     */
}
```

2.4. Modèle de données



2.5. Console de Symfony

La gestion de l'application est facilitée par l'utilisation des commandes de la console du framework Symfony.

Ci-dessous un récapitulatif des commandes les plus utilisées.

Les commandes doivent être lancées dans un interpréteur de commandes depuis la racine du projet.

Lancement du serveur

```
$ php bin/console -S 127.0.0.1:(port) -t public
```

Création / mise à jour d'une entité

```
$ php bin/console make:entity
```

Création des fichiers de migration

```
$ php bin/console make:migrations
```

Mise à jour de la structure de la base de données selon les migrations

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

Création d'un formulaire

```
$ php bin/console make:form
```

Réinitialisation du cache

```
$ php bin/console cache:clear
```

Création des fixtures

```
$ php bin/console doctrine:fixtures:load
```

Lancement des tests

```
$ php bin/phpunit
```

Consultation du conteneur de services

```
$ php bin/console debug:container
```

2.6. Mises à jour

Les mises à jour de Symfony et des librairie se fait par l'intermédiaire du gestionnaire de dépendances Composer.

Mise à jour de Symfony et de ses dépendances

```
$ composer update
```

Installation d'une nouvelle librairie

La liste officielle des librairies est consultable sur le site <https://packagist.org/> .

```
$ composer require [vendeur/librairie]
```

3. Authentification :

3.1. Fichier concernés

Le processus d'authentification est réalisé grâce au composant de sécurité du framework Symfony.

Documentation officielle : <https://symfony.com/doc/current/security.html>

Catégorie	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
Vue	templates/security/login.html.twig	Template du formulaire de connexion

3.2. Configuration

La configuration du composant de sécurité s'effectue dans le fichier :
config/packages/security.yaml

3.2.1. Cryptage des mots de passe

Les mots de passe sont cryptés par le composant de sécurité de Symfony ou vous pouvez vous-mêmes choisir le type d'encodage.

Afin de modifier l'algorithme de cryptage, il faut modifier la section **encoders** du fichier de configuration.

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

3.2.2. Passphrase

La passphrase permettant le cryptage des mots de passe est définie dans le fichier **.env** situé à la racine du projet (clé APP_SECRET).

Attention : le contenu de ce fichier doit rester strictement confidentiel. La passphrase doit être **impérativement** modifiée lors de la mise en production de l'application.

```
APP_ENV=dev
```

```
APP_SECRET=660601e50f800735550f220274d177
```

3.2.3. Provider

Il est possible de configurer plusieurs fournisseurs d'authentification. L'entité User étant utilisée ici, elle est configurée dans la section **providers**. On définit également l'attribut de la classe User qui sera utilisé comme identifiant de connexion.

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

3.2.4. Pare-feu

La section **firewalls** permet de définir les parties de l'application doivent être gérées par le composant de sécurité (**pattern**). Elle définit également :

- la route permettant la connexion (**form_login**)
- la route permettant la déconnexion (**logout**)

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

  main:
    anonymous: ~
    pattern: ^/
    form_login:
      login_path: login
      check_path: login_check
      always_use_default_target_path: true
      default_target_path: /
    logout: ~
```

Attention : le pare-feu indique quelle partie du site doit être gérée par le composant de sécurité, mais ne protège pas l'application. Les zones à protéger sont configurées dans la partie **access_control** (voir ci-dessous).

3.2.5. Contrôle des accès

La partie **access_control** permet de définir quelles parties de l'application doivent être protégées par le processus d'authentification. Elle permet également de définir les rôles autorisés pour chaque partie.

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

3.3. Stockage des utilisateurs

Les utilisateurs sont stockés dans la table user.

	#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
<input type="checkbox"/>	1	id 🔑	int(11)			Non	Aucun(e)		AUTO_INCREMENT
<input type="checkbox"/>	2	username 🔑	varchar(25)	utf8mb4_unicode_ci		Non	Aucun(e)		
<input type="checkbox"/>	3	password	varchar(64)	utf8mb4_unicode_ci		Non	Aucun(e)		
<input type="checkbox"/>	4	email 🔑	varchar(60)	utf8mb4_unicode_ci		Non	Aucun(e)		
<input type="checkbox"/>	5	roles	longtext	utf8mb4_unicode_ci		Non	Aucun(e)	(DC2Type:json)	

Les champs username et email sont des champs uniques.

L'accès aux données se fait via l'instanciation d'un objet de la classe **src/Entity/User.php** grâce à l'utilisation de la librairie Doctrine.

Exemple


```
$user = $this->getDoctrine()
->getRepository(User::class)
->findOneBy(['username' => 'test']);
```

3.4. Processus d'authentification

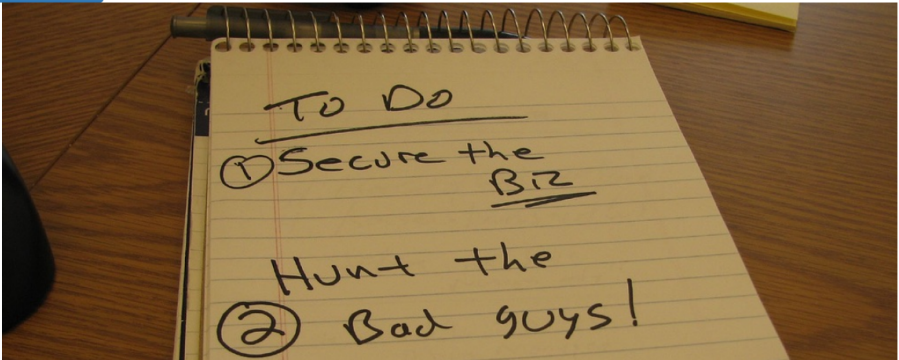
L'authentification est effectuée par le composant de sécurité de Symfony par l'intermédiaire de la classe **src/Security/LoginFormAuthenticator.php**.

3.4.1. Affichage formulaire

L'accès au formulaire se fait par l'url **/login**.

To Do List app 

[Créer un utilisateur](#)



Nom d'utilisateur : Mot de passe : [Se connecter](#)

Copyright © OpenClassrooms

La route correspondant à cette url est configurée dans l'annotation de la méthode **login()** du contrôleur **src/Controller/SecurityController.php**.

```
class SecurityController extends Controller
{
    /**
     * @Route("/login", name="login")
     */
    public function loginAction()
    {
        $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,
        ));
    }
}
```

Le contrôleur a pour rôle de lancer la création de la vue grâce au template Twig `templates/security/login.html.twig`.

```
templates > security > login.html.twig
4
5 {% block body %}
6 <form method="post">
7     {% if error %}
8         <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
9     {% endif %}
10
11     {% if app.user %}
12         <div class="mb-3">
13             You are logged in as {{ app.user.username }}, <a href="{{ path('logout') }}">Logout</a>
14         </div>
15     {% endif %}
16
17 <h1 class="h3 mb-3 font-weight-normal">Se connecter</h1>
18 <label for="inputUsername">Nom d'utilisateur :</label>
19 <input type="username" value="{{ last_username }}" name="username" id="inputUsername" class="form-control" required autofocus>
20 <label for="inputPassword">Mot de passe :</label>
21 <input type="password" name="password" id="inputPassword" class="form-control" required>
22
23 <input type="hidden" name="_csrf_token"
24     value="{{ csrf_token('authenticate') }}"
25 >
26
27 <button class="btn btn-lg btn-primary" type="submit">
28     Se connecter
29 </button>
30 </form>
31 {% endblock %}
```

3.4.2. Soumission du formulaire

Étape 1 - Soumission

L'utilisateur remplit le formulaire et le valide.

Étape 2 - Interception

La soumission du formulaire est interceptée automatiquement par le composant de sécurité de Symfony et traitée par la classe `src/Security/LoginFormAuthenticator.php`, définie en tant que classe d'authentification du firewall.


```
src > Security > LoginFormAuthenticator.php > PHP IntelliSense > LoginFormAuthenticator
19
20 class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
21 {
22     use TargetPathTrait;
23
24     public const LOGIN_ROUTE = 'login';
25
26     private $urlGenerator;
27     private $csrfTokenManager;
28     private $passwordEncoder;
29
30     public function __construct(UrlGeneratorInterface $urlGenerator, CsrfTokenManager $csrfTokenManager, PasswordEncoderInterface $passwordEncoder)
31     {
32         $this->urlGenerator = $urlGenerator;
33         $this->csrfTokenManager = $csrfTokenManager;
34         $this->passwordEncoder = $passwordEncoder;
35     }
36
37     public function supports(Request $request)
38     {
39         return self::LOGIN_ROUTE === $request->attributes->get('_route')
40             && $request->isMethod('POST');
41     }
42 }
```

Étape 3 - Récupération des champs du formulaire

Une fois la soumission interceptée, les données envoyées sont stockées dans un tableau **\$credentials** et la valeur du champ username est conservée en variable de session.

```
src > Security > LoginFormAuthenticator.php > PHP IntelliSense > LoginFormAuthenticator
42
43 public function getCredentials(Request $request)
44 {
45     $credentials = [
46         'username' => $request->request->get('username'),
47         'password' => $request->request->get('password'),
48         'csrf_token' => $request->request->get('_csrf_token'),
49     ];
50     $request->getSession()->set(
51         Security::LAST_USERNAME,
52         $credentials['username']
53     );
54
55     return $credentials;
56 }
```

Étape 4 - Récupération des champs du formulaire

Après vérification de la validité du token CSRF, l'utilisateur correspondant à la valeur du champ username envoyée va être recherché dans la base de données par l'utilisation de l'entité User et de la librairie Doctrine.

Si l'utilisateur est trouvé, il est stocké dans un objet **\$user** . Sinon une exception est levée.

```
src > Security > LoginFormAuthenticator.php > PHP IntelliSense > LoginFormAuthenticator
58 public function getUser($credentials, UserProviderInterface $userProvider)
59 {
60     $token = new CsrfToken('authenticate', $credentials['csrf_token']);
61     if (!$this->csrfTokenManager->isTokenValid($token)) {
62         throw new InvalidCsrfTokenException();
63     }
64
65     // Load / create our user however you need.
66     // You can do this by calling the user provider, or with custom logic here.
67     $user = $userProvider->loadUserByUsername($credentials['username']);
68
69     if (!$user) {
70         // fail authentication with a custom error
71         throw new CustomUserMessageAuthenticationException('Username could not be found.');
```

Étape 5 - Connexion et redirection

En cas de succès l'utilisateur est connecté et une redirection est effectuée :
- vers la page d'accueil

```
src > Security > LoginFormAuthenticator.php > PHP IntelliSense > LoginFormAuthenticator
82 public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
83 {
84     if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
85         return new RedirectResponse($targetPath);
86     }
87
88     return new RedirectResponse($this->urlGenerator->generate('homepage'));
89 }
```

En cas d'échec, la méthode login() du contrôleur est appelée à nouveau, et le formulaire est rechargé. L'erreur est affichée.

3.5. Gestion des rôles

Les rôles sont définis dans le champ rôles de la table user. Ils sont accessibles après avoir instancié l'entité User par la méthode **getRoles()**.

Pour autoriser ou refuser l'accès à une route ou une fonctionnalité, deux solutions principales sont possibles.

Exemple

```
/**
 * @Route("/users", name="user_list")
 */
public function listAction()
{
    if ($this->isGranted('ROLE_ADMIN')) {
        return $this->render('user/list.html.twig', [
            'users' => $this->getDoctrine()->getRepository(User::class)->findAll()
        ]);
    } else {
        return $this->render('error/error.html.twig');
    }
}
```

Dans cet exemple, si l'utilisateur possède le rôle "admin" il sera redirigé vers la page de la liste des utilisateurs sinon il sera redirigé vers une page d'erreur qui lui affichera qu'il ne possède pas le rôle administrateur.