Documentation - ToDo & Co

SOMMAIRE

1. Contexte

a. Résumé

ToDo & Co est une application permettant de gérer ses tâches quotidiennes. Toutes les actions / fonctionnalités du système nécessitent que l'utilisateur soit connecté / authentifié. Pour ce fait l'application dispose d'un système d'authentification par formulaire qui permet à un utilisateur de se connecter avant d'effectuer n'importe quelle action dans le système.

b. Les acteurs / utilisateurs

Cette application regroupe principalement deux acteurs:

- Les utilisateurs
- Les administrateurs

c. Déploiement du projet

- Cloner le Repository GitHub dans le dossier de votre choix. Pour cela utilisez la commande:

  git clone https://github.com/Gui-Dev86/P8-ToDoList.git

- Installer les dépendances du projet avec la commande Composer que vous devez avoir au préalable installé:

  composer install

- Le projet utilise deux bases de données l'une pour le fonctionnement de l'application l'autre pour les tests afin de ne pas interférer avec la base de données principale.

  Pour créer la base de données principale utilisez ces commandes:

  php bin/console doctrine:database:create

- Générez le fichier de migration des tables de la base de données

  php bin/console make:migration

- Effectuez la migrations vers la base de données:

  php bin/console doctrine:migrations:migrate

- Afin de vérifier le bon fonctionnement de l'application à l'aide de données fictives, vous pouvez intégrer de fausses données (fixtures) dans la base de données en validant avec "yes" pour accepter l'intégration.

  php bin/console doctrine:fixtures:load

- Pour créer la seconde base de données rendez-vous dans votre phpMyAdmin, une fois dans votre base de données "todolist" **exportez là afin de créer un fichier "todolist.sql"**. Il ne reste plus qu'à **créer une nouvelle base de données appelée "todolist_test" et y importer le fichier "todolist.sql"** précédemment créé afin d'obtenir une copie de la base de données "todolist". Celle-ci sera la cible des tests de l'application.

- Il ne reste qu'à connecter la base de données à l'application. Pour cela rendez vous à la fin de cette documentation dans la partie Stockage des utilisateurs.

d. Environnement de tests

- Pour vérifier le niveau de couverture de test en générant la documentation il est nécessaire d'installer XDebug (Documentation d'installation).

  Voici la commande qui générera la documentation de couverture de test dans le dossier reports/ :

  vendor\bin\phpunit --coverage-html reports/

- Pour vérifier le niveau de performance de l'application vous pouvez utiliser Blackfire (Documentation d'installation).

  Vous devrez ensuite vous rendre dans votre terminal de commande en mode administrateur afin de pouvoir lancer ou stopper Blackfire à l'aide de ces commandes:

  sc.exe start Blackfire
  sc.exe stop Blackfire

  Pour pourrez ensuite utiliser l'extension Blackfire que vous aurez installé dans votre navigateur web afin de générer une documentation sur le niveau de performance de l'application.

- Si vous souhaitez lancer les tests unitaires/fonctionnels d'une entité ou d'un controller voici deux exemples de commande à utiliser:

  php bin/phpunit tests/Entity/TaskEntityTest.php
  php bin/phpunit tests/Controller/TaskControllerTest.php

  Attention, les tests influent sur la base de données de test pour les relancer il peut-être nécessaire d'importer à nouveau le fichier .sql afin de la réinitialiser.

2. Entity User

a. Rôle de l'entité user

Une entité est une classe PHP, qui peut être connectée à une table d'une base de données via **l'ORM**. Lorsqu'une entité est liée à une table, via l'ORM, il y a en général un fichier "repository" associé. Un repository permet la génération de requêtes simples ou complexes et que le développeur peut modifier à volonté.

Un **ORM (Object Relation Mapper)** permet de gérer, manipuler et de récupérer des tables de données de la même façon qu'un objet

quelconque, donc en gardant le langage PHP. Plus besoin de requête MySQL, PostgresSQL ou autre.

Symfony utilise **Doctrine comme ORM** dans son système par défaut. Nous allons utiliser Doctrine dans l'application ToDo & Co.

Le rôle de l'entité user sera donc d'accéder et manipuler les données liées aux utilisateurs; afficher les utilisateurs, créer de nouveaux utilisateurs, les modifier et ce qui nous intéresse ici accéder aux logins et mots de passe des utilisateurs afin de les identifier lors de leur connexion.

Pour cela l'entité user implémente le **UserInterface** afin d'accéder aux fonctions liées à la gestion d'un utilisateur de la classe UserInterface qui est au préalable importé dans l'entité.

```php
<?php

namespace App\Entity;

use App\Repository\UserRepository;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;
```

b. Annotations

Pour des raisons de sécurité il est nécessaire d'intégrer certaines conditions à la création des utilisateurs notamment la complexité du mot de passe. Pour cela on importe le **Constraints as Assert** qui permettra de mettre en place ces conditions dans les annotations de déclaration des variables.

```php
/**
 * @var string The hashed password
 * @ORM\Column(type="string", length=64)
 * @Assert\NotBlank(
 *      message = "Ce champ est requis."
 * )
 * @Assert\Length(
 *      min = 6,
 *      max = 64,
 *      minMessage = "Votre mot de passe doit contenir au moins 6 caractères.",
 *      maxMessage = "Votre mot de passe ne peut pas contenir plus de {{ limit }} caractères."
 * )
 * @Assert\Regex(
 *     pattern = "^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])^",
 *     match = true,
 *     message = "Le mot de passe doit contenir au moins une minuscule, une majuscule et un chiffre."
 * )
 */
private $password;
```

Si le niveau de sécurité du mot de passe vous semble trop ou pas assez élevé, il est tout à fait possible de modifier sa longueur dans **les champs min/max** ou modifier le **pattern du regex** afin de complexifier ou simplifier le choix de password par l'utilisateur.

3. Component Security

a. Password hasher

Afin d'effectuer le hachage des mots de passe il est nécessaire de configurer l'option **password_hashers**. Vous devez configurer l' **algorithme de hachage** et éventuellement certaines options d'algorithme. Dans cette application, **l'algorithme "auto"** est utilisé afin de hacher les mots de passe. Ce hacheur sélectionne automatiquement l'algorithme le plus sécurisé disponible sur le système.

```yaml
password_hashers:
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    App\Entity\User:
        algorithm: auto
```

b. Providers

Il s'agit du fournisseur d'utilisateurs le plus courant. Les utilisateurs sont stockés dans une base de données et le fournisseur d'utilisateurs

utilise **Doctrine** pour les récupérer. Il s'agit d'indiquer le chemin vers l'entité utilisée pour la récupération de l'utilisateur ici **"user"** et la propriété utilisée pour l'authentification qui est **"username"**.

```
providers:
    # used to reload user from session &
    app_user_provider:
        entity:
            class: App\Entity\User
            property: username
```

c. Firewalls

Le firewall est le système d'identification à l'application. Son rôle est donc de définir comment les utilisateurs pourront se connecter à l'application.

Le firewall **dev** est un faux pare-feu : il s'assure que l'on ne bloque pas accidentellement les outils de développement de Symfony - qui vivent sous des URL comme **/_profileret /_wdt**.
Le firewall **main** permet de gérer l'authentification d'un utilisateur.

- Le provider utilise le **app_user_provider** qui a été configuré précédemment afin d'accéder aux informations sur les utilisateurs (login et mots de passe).

- Le **form_login** indique que la connexion à l'application se fait par un formulaire d'authentification. A l'intérieur les **login_path et check_path** prennent en charge les noms de route de la fonction utilisée pour se connecter.

- Le **logout** permet de gérer la déconnexion de l'application. Le path est le nom de route utilisé dans la fonction de logout.

```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        provider: app_user_provider
        form_login:
            login_path: app_login
            check_path: app_login
        logout:
            path: logout
```

d. Acces Control

L' access control va permettre de gérer les droits d'accès aux pages de l'application. Une personne se connectant devra avoir le droit d'accéder à la page d'accueil afin de se connecter.
Une utilisateur ayant le **ROLE_USER** aura le droit d'accéder à toutes les pages en dehors de celles liées à l'administration de l'application tandis qu'un utilisateur avec le **ROLE_ADMIN** devra pouvoir accéder à l'entièreté de l'application.
On configure donc ici quels rôles auront accès à quelles routes.

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

e. Role hierarchy

Le **role_hierarchy** permet d'organiser l'héritage des rôles. Ici le ROLE_ADMIN obtiendra donc les droits d'accès aux routes étant en ROLE_USER.
De cette manière le ROLE_ADMIN aura bien accès à toutes les routes tandis que le ROLE_USER n'aura pas droit de d'accès aux routes en /users.

```
role_hierarchy:
    ROLE_ADMIN: [ROLE_USER]
```

4. Template login

Voici le template d'identification à l'application. Comme cela a été configuré dans le firewall il s'agit d'un formulaire. Afin d'appeler la fonction d'identification c'est dans le **"<form action="">"** que l'on appelle le nom de la route accédant à la fonction d'identification.

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

5. Controller Security

C'est ici que se trouve les fonctions liées à la l'identification de l'utilisateur (login et logout) on peut donc bien voir dans l'annotation que le nom de la fonction est **"app_login"** c'est ce nom qui permet au formulaire ainsi qu'au firewall de trouver cette fonction et ainsi l'appeler.

```
/**
 * Login user
 *
 * @Route("/login", name="app_login")
 *
 * @return Response
 */
public function loginAction(AuthenticationUtils $authenticationUtils): Response
{
    $error = $authenticationUtils->getLastAuthenticationError();
    $lastUsername = $authenticationUtils->getLastUsername();

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

6. Stockage des utilisateurs

Les utilisateurs de l'application sont stockés dans une base de données. Pour avoir accès à celle-ci il est nécessaire de la paramétrer dans le fichier **.env**.

Il faut paramétrer le **DATABASE_URL** avec le nom d'utilisateur, mot de passe et nom de la base de données correspondant (ne pas oublier de retirer le # devant la ligne afin qu'elle soit prise en compte).

exemple : DATABASE_URL="mysql://utilisateur(root de base):mot de passe(vide de base)@127.0.0.1:3306/(nom de la base de données)

```
#
# DATABASE_URL="sqlite:///%kernel.project_dir%/var/data.db"
DATABASE_URL=mysql://root:@127.0.0.1:3306/todolist
# DATABASE_URL="postgresql://symfony:ChangeMe@127.0.0.1:5432/app?serverVersion=13&charset=utf8"
###< doctrine/doctrine-bundle ###
```