Documentation technique

Table des matières

1. L'architecture des fichiers

L'application ToDo & Co a été conçue à l'aide de la version 5.3 du framework Symfony.

Le projet contient les répertoires suivants :

- bin
- config
- public
- src
- templates
- tests
- var
- vendor

1.1. Le répertoire /bin

Ce répertoire contient le fichier « console » qui est le point d'entrée de toutes les commandes Symfony.

Il n'y a aucune modification à faire dans ce répertoire.

1.2. Le répertoire /config

Ce répertoire contient les fichiers de configuration nécessaires à notre application. On pourra configurer les routes, définir des services, régler la sécurité de l'application, configurer nos bundles, etc.

Les bundles installés pour l'application sont les suivants :

- DoctrineFixturesBundle (pour les fixtures)
- LiipTestFixturesBundle (pour charger les fixtures dans la base de données depuis un test)
- FidryAliceDataFixturesBundle (pour créer des fixtures au format yaml en incluant les « fake data » de FakerPHP/Faker)

1.3. Le répertoire /public

Ce répertoire contient tous les fichiers destinés aux visiteurs : images, fichiers CSS et JavaScript, etc. C'est le seul répertoire qui devrait être accessible aux visiteurs.

Il contient également le contrôleur frontal (index.php).

1.4. Le répertoire /src

Ce répertoire contient le cœur du code source de l'application. On y trouvera :

- Les controllers
- Les entités
- Les formulaires
- Le fichier Security\UserAuthenticator.php nécessaire à l'authentification.
- Le fichier Security\Voter\TaskVoter.php permettant de vérifier les droits des utilisateurs lors de l'édition ou la suppression d'une tâche.

1.5. Le répertoire /templates

Ce répertoire contient les templates Twig qui définissent le squelette visuel de l'application.

1.6. Le répertoire /tests

Ce répertoire contient les tests unitaires et fonctionnels de l'application.

1.7. Le répertoire /var

On trouvera dans ce répertoire les journaux des opérations de l'application ainsi que le cache.

1.8. Le répertoire /vendor

Ce répertoire contient toutes les bibliothèques externes à notre application.

1.9. Configuration des variables d'environnement dans les fichiers .env

Le fichier .env :

Au lieu de définir des variables d'environnement dans votre shell ou votre serveur Web, Symfony fournit un moyen pratique de les définir dans un fichier « .env » situé à la racine de votre projet.

Ce fichier sera lu et analysé à chaque requête et ses variables d'environnement seront ajoutées aux variables superglobales « $_ENV » et « $_SERVER » de PHP.

Ce fichier sera commité dans votre repository et ne devra contenir que des valeurs "**par défaut**" pour un développement local. Ce fichier ne devra donc pas contenir de valeurs de production ou tout autre valeur sensible.

Le fichier .env.local :

Si vous devez remplacer une valeur d'environnement sur votre machine locale, vous pouvez le faire en créant un fichier « .env.local ». Ce fichier sera ignoré de Git et ne sera donc pas commité.

C'est dans ce fichier qu'on indiquera par exemple l'URL de notre base de données locale :

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7"
```

Le fichier .env.test :

Remplace les variables d'environnement uniquement lors des tests. Ce fichier sera commité et ne devra donc contenir aucune valeur sensible.

Le fichier .env.test.local :

Définit les variables d'environnement locales spécifiques à votre machine uniquement pour les tests. Ce fichier sera ignoré de Git et ne sera donc pas commité.

C'est dans ce fichier qu'on indiquera par exemple l'URL de la base de données locale utilisée lors des tests unitaires ou fonctionnels :

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7"
```

Pour créer votre base de données de test, il faudra alors lancer les commandes suivantes :

php bin/console doctrine:database:create --env=test

php bin/console doctrine:schema:update --env=test –force

1.10. Intégration continue

Le projet utilise GitHub Actions pour l'intégration continue.

La configuration de l'intégration continue se trouve dans le fichier :

- .github\workflows\symfony.yml

À chaque mise à jour de la branche « main » du projet, nous effectuons les opérations suivantes :

- Une analyse avec php-cs-fixer pour vérifier la conformité du code PHP selon différentes normes (PSR)
- Une analyse statique du code PHP avec PHPStan pour capturer les erreurs avant qu'elles ne se produisent.
- L'exécution de nos tests unitaires et fonctionnels
- La transmission du rapport de couverture de nos tests à Codacy afin d'afficher le pourcentage de couverture dans notre fichier « README.md »
- Une analyse de la performance avec Blackfire lors de l'accès à la page login

2. Les entités

Les entités représentent les objets enregistrés dans la base de données. L'application contient actuellement deux entités : « User » et « Task ».

2.1. User (src/Entity/User.php)

Cette classe représente les utilisateurs de l'applications.

Champs	Types
id	integer
username	string
roles	array
password	string
email	string
tasks	Relation OneToMany avec l'entité « Task »

Les rôles possibles sont :

- **ROLE_USER**

  -> Créer une tâche
  -> Modifier une tâche appartenant à l'utilisateur
  -> Supprimer une tâche appartenant à l'utilisateur
  -> Modifier le statut d'une tâche appartenant à l'utilisateur (« à faire » ou « terminée »)
  -> Voir ses tâches

- **ROLE_ADMIN**

  -> Créer un utilisateur
  -> Supprimer un utilisateur
  -> Modifier un utilisateur

La hiérarchie des rôles est définie dans le fichier config\packages\security.yaml :

```yaml
    role_hierarchy:
        ROLE_ADMIN: ROLE_USER
```

Cela signifie que « ROLE_ADMIN » hérite de tous les droits de « ROLE_USER ».

Remarque :

Si un administrateur supprime un utilisateur, les tâches de ce dernier seront conservées et attribuées à un utilisateur anonyme.

2.2. Task (src/Entity/Task.php)

Champs	Types
id	integer
createdAt	datetime
title	string
content	text
isDone	boolean
user	Relation ManyToOne avec l'entité « User »

2.3. Mettre à jour la base de données

Si vous modifiez vos entités, il faudra alors mettra à jour votre base de données locale.

Pour cela, effectuer les commandes suivantes :

- **php bin/console make:migration**

  Cette commande va comparer l'état actuel de la base de données avec ce qu'elle devrait être en tenant compte de toutes nos entités. Puis elle va créer un fichier dans le dossier « migrations » comportant tout le SQL nécessaire pour qu'il n'y ait plus de différences entre ce qu'on souhaite avec nos entités et ce qu'il y a dans la base de données.

- **php bin/console doctrine:migrations:migrate**

  Cette commande exécute tous les fichiers de migration qui n'ont pas encore été exécutés sur votre base de données. Cela va donc mettre à jour votre base de données.

2.4. Fixtures

Pour tester l'application en local, vous pouvez initialiser les données de la base de données en chargeant les fixtures du fichier src\DataFixtures\UserFixtures.php.

Pour cela, effectuez les commandes suivantes dans votre terminal :

```
php bin/console d:d:d --if-exists --force
php bin/console d:d:c
php bin/console d:m:m
php bin/console doctrine:fixtures:load
```

Ces commandes vont respectivement :

- Effacer la base de données si elle existe
- Créer la base de données
- Créer les tables de la base de données d'après la structure de nos entités
- Charger les fixtures définies dans le fichier UserFixtures.php.

Pour effectuer ces commandes plus rapidement, vous pouvez les rajouter dans votre composer.json :

```
"scripts": {
    ...
    "prepare-database-dev": [
        "php bin/console doctrine:database:drop --if-exists -f",
```

```
        "php bin/console doctrine:database:create",
        "php bin/console doctrine:migrations:migrate",
        "php bin/console doctrine:fixtures:load -n"
    ]
},
```

Dans votre terminal, il vous suffira alors de lancer la commande suivante pour réinitialiser votre base de données locale :

```
composer prepare-database-dev
```

3. Sécurité de l'application

3.1. L'authentification

L'application est protégée par le firewall « main » défini dans le fichier config\packages\security.yaml :

```yaml
    firewalls:
        main:
            lazy: true
            provider: app_user_provider
            custom_authenticator: App\Security\UserAuthenticator
            logout:
                path: app_logout
```

L'absence d'une clé « pattern » indique que toutes les routes de l'application sont protégées.

La classe App\Security\UserAuthenticator sera chargée de l'authentification des utilisateurs. Elle contient une méthode « support » permettant de définir les conditions pour commencer une authentification :

```php
    public function supports(Request $request): bool
    {
        return $request->isMethod('POST') && $this->getLoginUrl($request) ===
        $request->getPathInfo();
    }
```

Pour que l'authentification soit vérifiée, l'utilisateur devra effectuer une requête « POST » sur la route /login.

Si c'est le cas, la méthode « support » renverra True et la méthode « authenticate » de la classe UserAuthenticator sera appelée pour vérifiera l'identité de l'utilisateur.

La méthode « authenticate » renvoie une instance de la classe « Passport » :

```
        $username = $request->request->get('username', '');

        $request->getSession()->set(Security::LAST_USERNAME, $username);

        return new Passport(
            new UserBadge($username),
            new PasswordCredentials($request->request->get('password', '')),
            [
                new CsrfTokenBadge('authenticate', $request->get('_csrf_token')),
            ]
        );
```

Ce passport contient trois badges qui devront être validés par différents listeners de Symfony appelés lorsque l'évènement « CheckPassportEvent » sera déclenché (voir méthode « executeAuthenticator » de la classe AuthenticatorManager »).

- **UserBadge :**

  Ce badge a besoin d'un identifiant (ici le nom de l'utilisateur) et d'un « user loader » pour charger l'instance User correspondante.

  Dans notre application, le « user loader » est défini dans le fichier config\packages\security.yaml :

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

  Le « user loader » sera donc la classe :

  vendor\symfony\doctrine-bridge\Security\User\EntityUserProvider.php

  Le listener « UserCheckerListener » conduira Symfony à appeler la méthode « loadUserByIdentifier » de la classe « EntityUserProvider » afin de récupérer dans la base de données l'utilisateur ayant le même username que celui fourni au UserBadge.

  Si cela réussi, le UserBadge sera automatiquement validé, sinon une UserNotFoundException sera lancée.

- **CsrfTokenBadge :**

Ce badge sera vérifié par la méthode « checkPassport » du listener CsrfProtectionListener afin de s'assurer que le token CSRF du formulaire de connexion est bien valide.

- **PasswordCredentials :**

  Ce badge sera vérifié par la méthode « checkPassport » du listener CheckCredentialsListener afin de s'assurer que le mot de passe de l'utilisateur est valide.

Si tous les badges sur passport sont résolus, l'authentification sera validée et un token d'authentification sera créé.

Dans ce cas, Symfony exécutera la méthode « onAuthenticationSuccess » de la classe « UserAuthenticator » :

```
   public function onAuthenticationSuccess(Request $request, TokenInterface
$token,string $firewallName): ?Response
    {
        if ($targetPath = $this->getTargetPath($request->getSession(),
$firewallName)) {
            return new RedirectResponse($targetPath);
        }
        return new RedirectResponse($this->urlGenerator->generate('homepage'));
    }
```

En cas d'échec de l'authentification, c'est la méthode « onAuthenticationFailure » qui sera appelée :

```
    public function onAuthenticationFailure(Request $request, AuthenticationException
$exception): Response
    {
        if ($request->hasSession()) {
            $request->getSession()->set(Security::AUTHENTICATION_ERROR, $exception);
        }
        $url = $this->getLoginUrl($request);
        return new RedirectResponse($url);
    }
```

3.2. L'autorisation

La gestion des rôles est définie dans le fichier config\packages\security.yaml :

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

- si l'utilisateur n'est pas authentifié, il ne pourra accéder qu'à la route « /login ».
- les routes commençant par «/users » ne sont accessibles qu'aux administrateurs
- les routes restantes sont accessibles aux utilisateurs authentifiés.

Exemple :

Si un utilisateur non authentifié se rend sur la route « / » :

La méthode « support » de la classe UserAuthenticator renverra False.

Comme la route « / » nécessite le rôle « ROLE_USER », la méthode « support » de la classe « AccessListener » interne à Symfony renverra « True » et la méthode « authenticate » de cette même classe sera appelée afin de vérifier que l'utilisateur a un rôle suffisant pour accéder à cette route.

L'utilisateur n'étant pas authentifié, il n'aura pas le rôle suffisant donc une exception « AccessDeniedException » sera lancée.

Symfony gérera cette exception en appelant la méthode « start » de la classe parente de « UserAuthenticator » afin de rediriger l'utilisateur vers la page de connexion :

```
    /**
     * Override to control what happens when the user hits a secure page
     * but isn't logged in yet.
     */
    public function start(Request $request, AuthenticationException
$authException = null): Response
    {
        $url = $this->getLoginUrl($request);

        return new RedirectResponse($url);
    }
```

Cette méthode est le point d'entrée de notre authentification.

Pour plus d'informations sur les « entry points » dans Symfony :

https://symfony.com/doc/current/components/security/firewall.html#entry-points

https://symfony.com/doc/current/security/authenticator_manager.html#authenticators-required-entry-point

4. Comment contribuer au projet ?

4.1. Créez une issue

Rendez-vous sur le repository GitHub du projet et vérifiez si une issue existe déjà pour ce que vous souhaitez faire et mettez à jour son contenu et/ou son statut si nécessaire. Sinon, créez une nouvelle issue que vous mettrez à jour tout au long du processus.

4.2. Installez le projet en local

Vous devez ensuite dupliquer le projet (fork) puis le cloner pour avoir une copie de la dernière version du projet sur votre ordinateur.

Plus de détails sur la documentation GitHub.

4.3. Créez une nouvelle branche

Créez une branche pour résoudre votre issue tout en prenant soin de la nommer de manière cohérente et compréhensible (en anglais de préférence).

La convention de dénomination des branches est la suivante :

type/nom ou type/nom/issue_ID

Exemples :

- feature/add-delete-user-action/17
- fix/link-tasks-to-user

Placez-vous sur cette nouvelle branche et apportez vos modifications de code, en les divisant en plusieurs commits si nécessaire. Rédigez les messages de vos commits en anglais de préférence.

4.4. Testez vos modifications

Exécutez les tests pour vérifier qu'ils réussissent toujours après vos modifications :

```
./vendor/bin/phpunit
```

Si nécessaire, mettez à jour les tests existants ou créez-en de nouveaux pour tester votre contribution.

Mettez ensuite à jour le fichier de test de couverture pour Codacy, avec la commande suivante :

```
./vendor/bin/phpunit --coverage-clover tests/coverage.xml
```

N'oubliez pas de commiter ce nouveau fichier « **tests/coverage.xml** ».

4.5. Créez une pull request

Poussez ensuite vos modifications sur votre dépôt distant et créez une pull request.

Plus de détails sur les pull requests sur la documentation GitHub .

Si votre contribution est approuvée, elle sera fusionnée dans la branche principale du projet.