Documentation technique

Table des matières

Introduction 1
Contexte 2
Migration de Symfony 3.1 à 5.4 2
L'Authentification 2
Liaison des tâches à l'utilisateur 4
Ajout des différents rôles à l'utilisateur 4
Gestion des utilisateurs uniquement par l'administrateur 6
Intégration des tests 6
Tests unitaires 7
Tests fonctionnels 7
Tests coverage 8
Les DataFixtures 9

Introduction

Cette documentation technique a pour objectif de présenter les évolutions liées à l'amélioration continue de l'application PHP/Symfony Todo & Co. Elle détaille les améliorations apportées au système d'authentification de l'utilisateur ainsi qu'à la gestion des tâches par l'utilisateur authentifié.

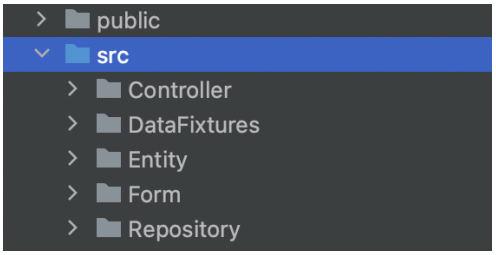
Contexte

Suite à l'examen du cahier des charges, plusieurs modifications ont été convenues :

1. Migration de Symfony 3.1 à 5.4
2. Système d'authentification
3. Liaison des tâches à l'utilisateur
4. Ajout des différents rôles à l'utilisateur (utilisateur et administrateur)
5. Gestion des utilisateurs uniquement par l'administrateur
6. Intégration des tests unitaires
7. Les DataFixtures

Migration de Symfony 3.1 à 5.4

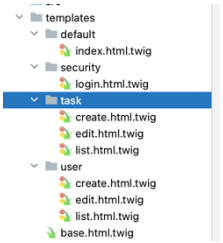
Afin de maintenir l'application à jour et d'améliorer la sécurité, une migration majeure a été effectuée vers Symfony version 5.4.



Un nouveau projet basé sur Symfony 5.4 a été créé, en adaptant le dossier "src" de l'ancien projet. Le dossier "AppBundle" a été supprimé, et ses contenus (contrôleurs, entités et formulaires) ont été intégrés directement dans le projet.

Le dossier "web", qui contenait les ressources accessibles (html, css, js, images) a été renommé en "public". Il est important de configurer le serveur pour pointer vers ce dossier, car il contient le fichier "index.php".

De plus, le dossier "app/Resources/view" a été remplacé par le dossier "templates" accessible depuis la racine du projet.



L'Authentification

L'authentification est gérée par le contrôleur "SecurityController.php" situé dans "src/Controller/SecurityController.php

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

    return $this->render( view: 'security/login', [
        'last_username' => $lastUsername,
        'error' => $error,
    ]);
}


/**
 * @Route("/logout", name="logout")
 */
public function logout()
{
    $this->redirectToRoute( route: 'task_list');
}
```

Ce contrôleur contient deux fonctions : "index" et "logout".

. La fonction "index" est responsable de l'authentification de l'utilisateur via le formulaire de connexion.

La fonction "logout" permet à un utilisateur authentifié de terminer sa session. La surcharge de cette fonctionnalité n'est pas nécessaire, car Symfony la gère automatiquement.

Pour activer la fonctionnalité "Logout", il suffit de configurer le fichier de configuration suivant : "config/packages/security.yaml".

```
            login_path: app_login
            check_path: app_login
        logout:
            path: logout
            target: /tasks
```

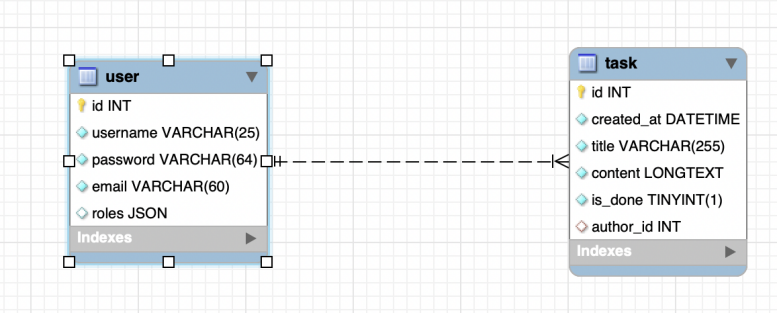
Liaison des tâches à l'utilisateur

Une relation "ManyToOne" a été ajoutée entre les entités "User" et "Task" dans l'entité "Task.php".

- src/Entity/Task.php
- Command client : php bin/console make:task
- Selectionner « relation » lors du choix du type
- Choisir ManyToOne

La migration du schéma de la base de données peut être effectuée en utilisant les commandes suivantes :

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



Après la migration, les getter et setter sont automatiquement ajoutés.

Ajout des différents rôles à l'utilisateur

- src/Entity/User.php

```php
/**
 * @var array
 * @ORM\Column(type="json", nullable=true)
 */
private $roles = [];
```

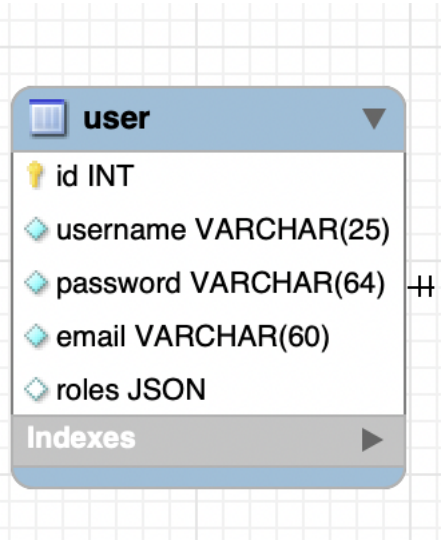
Dans l'entité "User.php", le champ spécifique des rôles est enregistré sous forme de tableau. Ce tableau est ensuite envoyé à la base de données en tant que format JSON.

Étant donné que la classe "User" est une interface spécifique de Symfony, les méthodes getter et setter doivent être ajoutées conformément à la nouvelle version de Symfony 5.4.

```php
/**
 * @see UserInterface
 */
public function getRoles(): ?array
{
    $roles = $this->roles;
    // guarantee every user at least has ROLE_USER
    $roles[] = 'ROLE_USER';

    return array_unique($roles);
}

public function setRoles(?array $roles): ?self
{
    $this->roles = $roles;

    return $this;
}
```

Pour chaque nouveau type de rôle utilisateur ajouté dans Symfony, la variable "access_control" dans le fichier de configuration "config/packages/security.yaml" doit être mise à jour.

La migration du schéma de la base de données peut être effectuée en utilisant les commandes mentionnées précédemment. « Liaison des tâches à l'utilisateur »



```yaml
# Easy way to control access for large sections of your site
# Note: Only the *first* access control that matches will be used
access_control:
    - { path: ^/admin, roles: ROLE_ADMIN }
    - { path: ^/profile, roles: ROLE_USER }
```

Après cela, nous avons bien la colonne rôle qui est ajoutée à notre base de données.

Gestion des utilisateurs uniquement par l'administrateur

FRONT :

```twig
{% if is_granted('ROLE_ADMIN') %}
    <a href="{{ path('user_create') }}" class="btn btn-primary">Créer un utilisateur</a>
    <a href="{{ path('user_list') }}" class="btn btn-primary">Liste des utilisateurs</a>
{% endif %}
```

Pour l'affichage des liens de navigation, seuls les utilisateurs authentifiés en tant qu'administrateurs peuvent les voir. Cela est réalisé en utilisant la fonction Twig "is_granted".

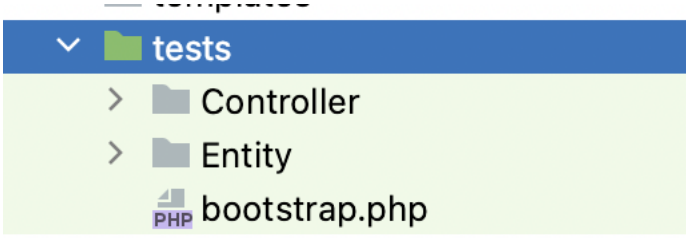
BACK :

```php
public function listAction(EntityManagerInterface $entityManager): Response
{
    $this->denyAccessUnlessGranted( attribute: 'ROLE_ADMIN');
```

Pour restreindre l'accès aux routes aux utilisateurs non administrateurs, la fonction "denyAccessUnlessGranted" est ajoutée à chaque route concernée.

Intégration des tests

Une fois les fonctionnalités développées, il est important de vérifier leur bon fonctionnement afin de maintenir la qualité du code de l'application Symfony.



Les tests sont divisés en tests unitaires pour les entités (Entity) et tests fonctionnels pour les contrôleurs (Controllers).

Les tests peuvent être exécutés en configurant l'IDE ou en utilisant la console avec la commande "php bin/phpunit 'chemin'".

Tests unitaires

La classe php doit extends « TestCase » pour un test unitaire

Un exemple de test unitaire est présenté pour tester les méthodes getter et setter de l'attribut "createdAt" de l'entité "Task.php".

```php
public function testcreatedAt()
{
    $date = new DateTime();
    $task = $this->getEntityTask();
    $task->setCreatedAt($date);
    $this->assertEquals($date, $task->getCreatedAt());
}
```

Tests fonctionnels

Les classes de test fonctionnel doivent étendre la classe "WebTestCase".

Un exemple de test fonctionnel est présenté ci-dessous pour simuler les actions d'un utilisateur via le client du site et vérifier le bon fonctionnement des différentes fonctions du contrôleur.
Les tests fonctionnels automatisent des tâches répétitives et permettent de gagner du temps.

ici, nous vérifions si dans le cadre d'un ajout d'une tâche si la page redirige bien vers une autre comportant le message « la tâche a bien été ajouté »
.

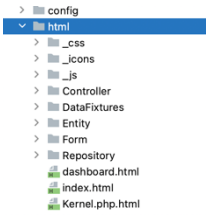
```php
public function testListPageIsUp()
{
    $this->client->request( method: Request::METHOD_GET, $this->urlGenerator->generate( name: 'task_

    $this->assertResponseIsSuccessful();
}

public function testCreateNewTaskUser()
{
    $crawler = $this->client->request( method: Request::METHOD_GET, $this->urlGenerator->generate(
    $form = $crawler->selectButton( value: 'Ajouter')->form();
    $form['task[title]'] = self::ADD_TASK_TITLE_1;
    $form['task[content]'] = self::ADD_TASK_CONTENT_1;
    $this->client->submit($form);
    $this->assertSelectorTextContains(
        selector: 'div.alert.alert-success',
        text: "La tâche a été bien été ajoutée."
    );

    $this->client->loginUser($this->user);
    $crawler = $this->client->request( method: Request::METHOD_GET, $this->urlGenerator->generate(
    $form = $crawler->selectButton( value: 'Ajouter')->form();
    $form['task[title]'] = self::ADD_TASK_TITLE_2;
    $form['task[content]'] = self::ADD_TASK_CONTENT_2;
    $this->client->submit($form);
    $this->assertSelectorTextContains(
        selector: 'div.alert.alert-success',
        text: "La tâche été ajoutée."
    );
}
```

Tests coverage

Les tests de couverture visent à exécuter tous les tests afin de générer un tableau de bord indiquant le taux de couverture des tests unitaires et fonctionnels dans le projet.

	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total		87.79%	115 / 131		83.72%	36 / 43		40.00%	4 / 10
Controller		91.30%	63 / 69		66.67%	8 / 12		25.00%	1 / 4
Entity		96.67%	29 / 30		95.83%	23 / 24		50.00%	1 / 2
Form		100.00%	14 / 14		100.00%	3 / 3		100.00%	2 / 2
Repository		50.00%	9 / 18		50.00%	2 / 4		0.00%	0 / 2
Kernel.php		n/a	0 / 0		n/a	0 / 0		n/a	0 / 0

Pour générer le tableau de bord en HTML, la commande suivante est utilisée via la console : "php bin/phpunit --coverage-html 'nom-du-dossier-souhaité'". Le résultat du test de couverture est accessible via le fichier "index.html" généré dans le dossier du projet.



Nous obtenons dans le projet un nouveau dossier qui comporte une page : index.html que nous pouvons ouvrir via un navigateur.

Les DataFixtures

Dans le cadre des tests, des classes de fixtures ont été créées en utilisant le bundle "DoctrineFixtureBundle". Ce bundle permet d'insérer des fausses données dans la base de données de test afin d'initialiser correctement les tests fonctionnels avec les bonnes valeurs.

Un exemple de création de deux objets "Task" lors de l'exécution de la commande de fixtures est présenté : "php bin/console doctrine:fixtures:load --env=test". Veuillez noter que les données enregistrées seront supprimées à chaque exécution de cette commande.

Attention à chaque lancement de commande les données enregistrées seront supprimées.

```php
const TITLE = ['faire de la pizza', 'manger une glâce', 'faire les course'];
const CONTENT = ['préparer la pizza', 'préparer la glace', 'prendre une orange'];

public function load(ObjectManager $manager)
{
    $userAdmin = $manager->getRepository(User::class)->findOneBy(['username' => "helloUser"]);
    $user = $manager->getRepository(User::class)->findOneBy(['username' => "Admin"]);
    $date = new \DateTime( datetime: 'now');

    for ($i = 0; $i < 3; $i++) {
        $task = new Task();
        if ($i === 0) {
            $task->setAuthor($userAdmin);
        } else if ($i === 1){
            $task->setAuthor($user);
        }
        $task->setCreatedAt($date);
        $task->setTitle(self::TITLE[$i]);
        $task->setContent(self::CONTENT[$i]);
        $manager->persist($task);
    }
    $manager->flush();
}
```