



Table des matières

ToDo & Co	1
Introduction	2
Contexte	2
Migration de Symfony 3.1 à 5.4	2
L'Authentification	3
Liaison des tâches à l'utilisateur	4
Ajout des différents rôles à l'utilisateur	5
Gestion des utilisateurs uniquement par l'administrateur	6
Intégration des tests	7
Tests unitaires	7
Tests fonctionnels	7
Tests coverage	8
Les DataFixtures	9

Introduction

Cette documentation technique a pour but de vous présenter les évolutions liées à l'amélioration continue de l'application PHP/Symfony Todo & Co.

Celle-ci vous détaillera les améliorations apportées au système d'authentification de l'utilisateur ainsi qu'envers la gestion de la tâche par l'utilisateur authentifié.

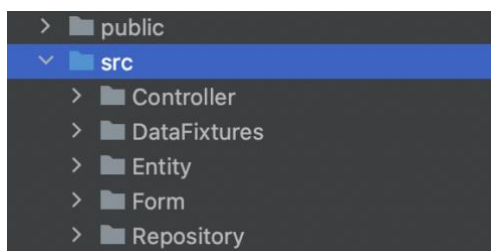
Contexte

A la prise de connaissance du cahier des charges nous avons convenu d'opérer plusieurs modifications :

- _ Migration de Symfony 3.1 à 5.4
- _ L'authentification
- _ Liaison des tâches à l'utilisateur
- _ Ajout des différents rôles à l'utilisateur (user & admin)
- _ Gestion des utilisateurs uniquement par l'administrateur
- _ Intégration des tests unitaires

Migration de Symfony 3.1 à 5.4

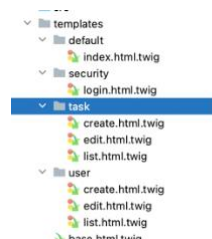
Afin de maintenir l'application à jour ainsi que dans le but d'optimiser la sécurité. Nous avons dû opérer un changement majeur en migrant la version de Symfony vers la version 5.4.



Pour cela, nous avons repris un web-skeleton de base symfony 5.4 tout en y adaptant le dossier « src » de l'ancien projet dans le but de pouvoir commencer les modifications. Le sous dossier AppBundle n'existe plus. Celui-ci comporte les Controllers les Entité ainsi que les Formulaire.

Le contenu du dossier « web » regroupe les ressources accessibles (html, css, js, images) nécessaires au bon fonctionnement navigateur a été renommé « public ». Votre server devra pointer sur ce dossier car il contient le fichier index.php.

Les dossier app/Resources/view a été remplacé par le dossier templates qui est accessible directement depuis la racine du projet.



L'Authentification

```
/**
 * @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');
}
```

L'authentification est gérée par le controller :

⇒ src/Controller/SecurityController.php

Celui-ci regroupe 2 fonctions l'une « index » et l'autre « logout »

La fonction index :

SecurityController est un controller spécial de Symfony qui gère l'authentification via la page index du formulaire de login. Celui vérifie lui-même sans avoir à ajouter du code si la soumission du formulaire du login est correcte afin d'authentifier le bon utilisateur.

La fonction logout :

Ne peut être appelée qu'en cas d'authentification reconnue de l'application Symfony. Elle permet à l'utilisateur qui le souhaite d'arrêter sa session.

Ici, non plus c'est Symfony on ne surcharge pas cette fonctionnalité.

On a juste à activer Logout dans fichier de config suivant :

config/packages/security.yaml

```
login_path: app_login
check_path: app_login

logout:
  path: logout
  target: /tasks
```

Liaison des tâches à l'utilisateur

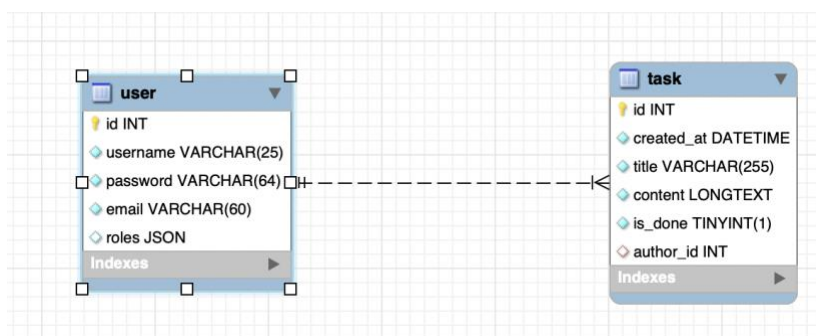
Ajout de la liaison « ManyToOne » entre le « User » et « Task » dans l'entité Task dont le chemin est

```
/**
 * @ORM\ManyToOne(targetEntity=User::class)
 */
private $author;
```

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

Nous pouvons alors lancer la migration pour mettre à jour le schéma de base données :
Avec les commandes :

- ⇒ php bin/console make:migration
- ⇒ php bin/console doctrine:migrations:migrate



Après la migration nous observons bien que la migration a bien été répercutée sur le schéma.

Les Getter et les setters sont ajoutés automatiquement à la fin de la commande.

Ajout des différents rôles à l'utilisateur

⇒ src/Entity/User.php

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

Pour ce champ spécifique la donnée est enregistrée dans un tableau. Ce tableau est envoyé ensuite vers la Base de données par Doctrine et est ensuite converti au format Json.

La classe User étant une interface spécifique de Symfony « User » nous ajoutons les méthodes Getter & Setter comme exigée par la nouvelle version de Symfony 5.4 .

```
/**
 * @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;
}
```

A chaque ajout d'un nouveau type de rôle utilisateur dans Symfony il faut renseigner la variable « access_control » dans le dossier de configuration au chemin

=> config/packages/security.yaml

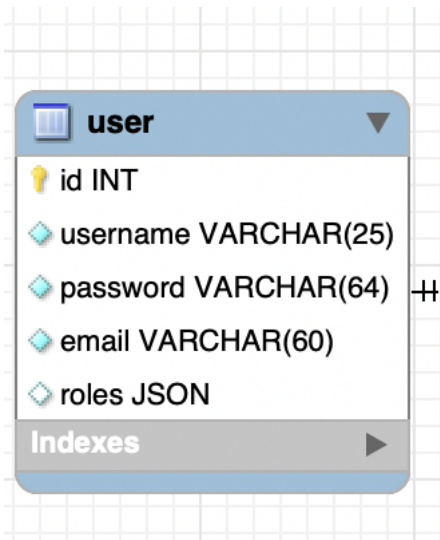
Comme suivi :

```
# 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 }
```

Nous pouvons alors lancer la migration pour mettre à jour le schéma de base donnée :

Avec les commandes :

- ⇒ php bin/console make:migration
- ⇒ php bin/console doctrine:migrations:migrate



Nous avons bien la colonne rôle qui ajoutée à notre base de données.

Gestion des utilisateurs uniquement par l'administrateur

FRONT :

```
{% 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 %}
```

Affichage des liens des routes uniquement si l'utilisateur authentifié est admin

Avec la fonction twig « is_granted »

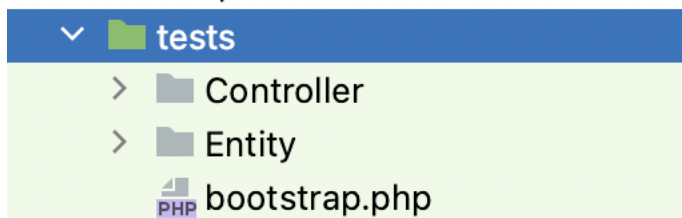
BACK :

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

Ajout dans chaque routes restreintes aux non « Admin » de la fonction
« denyAccessUnlessGranted »

Intégration des tests

Une fois les fonctionnalités développées nous devons vérifier le bon fonctionnement de celles-ci afin de maintenir le code de l'application Symfony.



Comme dans l'ancien projet nous retrouvons le dossier tests qui comporte les tests unitaires liés aux Entités (Entity) ainsi que les tests fonctionnels pour les Controllers

Nous pouvons lancer le test en configurant l'IDE mais pour être plus efficace nous pouvons lancer PHP/unit directement avec la console :

⇒ **php bin/phpunit « path »**

Tests unitaires

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

Nous retrouvons ici la façon dont il faut procéder dans le cadre d'un test unitaire destiné à tester une entité (Task.php). Nous testons ici un getter et un setter de l'attribut « createdAt ».

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

Tests fonctionnels

La classe php doit extends « WebTestCase » pour un test fonctionnel.

Nous vérifions en imitant un utilisateur qui effectue des actions via le client du site en implémentant le test des différentes fonctions du Controller. Les tests fonctionnels ont pour intérêt de faire gagner du temps car ils automatisent des tâches répétitives.

Par exemple, 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 ajoutée »






```
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

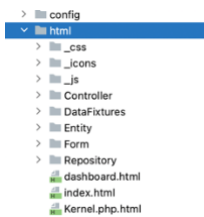
	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total	<div><div></div></div>	87.79%	115 / 131	<div><div></div></div>	83.72%	36 / 43	<div><div></div></div>	40.00%	4 / 10
 Controller	<div><div></div></div>	91.30%	63 / 69	<div><div></div></div>	66.67%	8 / 12	<div><div></div>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

En complémentarité, le test Coverage a pour but d'exécuter l'intégralité des tests pour générer un Dashboard indiquant les taux de couverture des tests unitaires et fonctionnels à l'intérieur d'un projet.

Pour obtenir la génération du Dashboard en HTML on lance une commande via la console qui va analyser la couverture du projet :

php bin/phpunit --coverage-html « nom-du-dossier-souhaité »

Vous pouvez trouver le contenu du coverage actuel dans : `/coverage/index.html`



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

Les DataFixtures

- ▼ DataFixtures
 - TaskFixtures.php
 - UserFixtures.php

Dans le cadre du bon déroulement des test j'ai dû créer des classes fixtures (voir doc : [DoctrineFixtureBundle](#)). Ce Bundle permet d'insérer des fausses données en base de données de test afin de pouvoir initialiser correctement les tests fonctionnels avec les bonnes valeurs.

Ci-dessous, nous créons 2 objets task en bdd au lancement de la commande fixture :

php bin/console doctrine:fixtures:load --env=test

```
const TITLE = ['faire de la pizza', 'manger une glace', '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();
}
```

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