Documentation technique

Améliorez une application existante de ToDo & Co

Table des matières

Pré-requis

Pour démarrer ce projet convenablement, il est nécessaire de respecter les points suivants :

1. Git doit être installé sur votre machine.
   https://git-scm.com

2. Composer doit être installé
   Si cela n'est pas le cas, vous pouvez vous rendre à l'adresse suivante et suivre le processus pour l'installer : https://getcomposer.org/doc/00-intro.md

3. Un serveur web Apache et MySQL doit être installé sur votre machine.
   Exemple de logiciel à avoir :
   - MAMP (MacOS) : https://www.mamp.info/fr/downloads/
   - WAMP (Windows) : https://www.wampserver.com

4. La version de PHP qui doit être installé est la 7.3

5. Les extensions PHP doivent être installées/activées
   a. JSON
   b. Xdebug
   c. Session

Le framework

Le framework utilisé est Symfony.
Lors de la reprise du projet, celui-ci était sous la version Symfony 3.1, puis une mise à jour vers la dernière version LTS (Symfony 4.4) a été faite

Librairies utilisées

- DEV/PROD
  - php
  - blackfire/php-sdk
  - doctrine/doctrine-migrations-bundle
  - sensio/framework-extra-bundle
  - symfony/asset
  - symfony/console
  - symfony/flex
  - symfony/form
  - symfony/framework-bundle
  - symfony/lts
  - symfony/maker-bundle
  - symfony/phpunit-bridge
  - symfony/twig-bundle
  - symfony/bundle
  - symfony/validator
  - symfony/yaml
- DEV
  - doctrine/doctrine-fixtures-bundle
  - liip/test-fixtures-bundle
  - symfony/debug-pack
  - symfony/dotenv
  - symfony/test-pack
  - symfony/web-server-bundle

Composer

Pour installer facilement de nouvelles dépendances, ou pour simplement les mettre à jour, Symfony utilise composer.
Voici certaines commandes que nous pouvons avoir régulièrement besoins :

- `macbook-pro-de-damien:ToDoList damien$ composer install`
  composer install sert à installer composer. Cela créera un fichier « vendor » à la racine du projet avec toutes les dépendances présente dans le fichier composer.json
- `macbook-pro-de-damien:ToDoList damien$ composer update`
  composer update sert à mettre à jour les dépendances présentes dans le fichier composer.json.
- `macbook-pro-de-damien:ToDoList damien$ composer require`
  Avec **composer require**, il est possible d'installer de nouvelles dépendances. Pour trouver celle qui correspond le mieux aux besoins, vous pouvez vous rendre sur le site suivant : https://packagist.org

Route

Symfony permet de configurer les routes de plusieurs façons différentes.
Mais pour ce projet, les annotations ont été utilisées.
Dans chaque méthode d'un contrôleur, il est nécessaire d'ajouter une annotation @Route afin de définir le chemin d'accès à ce rendu

Exemple :

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route as Route;

class DefaultController extends AbstractController
{
    /**
     * @Route("/", name="homepage")
     */
    public function indexAction()
    {
        return $this->render('default/index.html.twig');
    }
}
```

Pour cela, le bundle *Symfony\Component\Routing\Annotation\Route* doit être obligatoirement utilisé dans le contrôleur avec les paramètres suivant :

- Entre guillemet : « le chemin d'accès »
- **Le nom de la page**. Pour pouvoir ajouter des liens dans les templates.

Liste des fichiers de l'application

Configuration

config	packages	config/packages/*	Dossier comportant les fichiers de configuration de l'application.
	routes	config/routes/*	Dans ce dossier se trouve les fichiers de configuration des routes de l'application
	bootstrap.php	config/bootstrap.php	
	bundles.php	config/bundles.php	Fichers retournant les bundles installés dans l'application Symfony.
	routes.yaml	config/routes.yaml	Fichier de configuration des routes. Dans ce projet, il n'est pas utilisé.
	services.yaml	config/services.yaml	Fichier de configuration des services

Structures MVC

Modèles	Task.php	src/Entity/Task.php	Entité des tâches
	User.php	src/Entity/User.php	Entité des utilisateurs
Vues	Default	templates/default/*	Templates de la page d'accueil
	Registration	templates/registration/*	Templates de la création des utilisateurs
	Security	templates/security/*	Templates de la connexion utilisateur
	Task	templates/task/*	Templates de la gestion des tâches
	User	templates/user/*	Templates de la gestion des utilisateurs
	base.html.twig	templates/base.html.twig	Template du layout de l'application
Contrôleurs	DefaultController.php	src/Controller/DefaultController.php	Contrôleur affichant la page d'accueil
	SecurityController.php	src/Controller/SecurityController.php	Contrôleur affichant le formulaire de connexion, la fonction Login exécutant les appels aux fonctions pour se connecter et la fonction pour se déconnecter
	TaskController.php	src/Controller/TaskController.php	Contrôleur de gestion des tâches. Il permet d'afficher les différents formulaires de création, d'édition et suppression des tâches. Il est également possible de marquer la tâche comme "terminée" et d'afficher la liste de toutes les tâches
	UserController.php	src/Controller/UserController.php	Contrôleur permettant d'afficher le formulaire d'édition des utilisateurs et la liste de tous les utilisateurs.
	RegistrationController.php	src/Controller/RegistrationController.php	Contrôleur de création d'utilisateur. Il permet d'afficher le formulaire de création et d'exécuter les fonctions pour ajouter un nouvel utilisateur

Formulaires

Formulaires	RegistrationFormType.php	src/Form/RegistrationFormType.php	Formulaire de création et de modification d'un utilisateur
	TaskType.php	src/Form/TaskType.php	Formulaire de création et de modification d'une tâche

Autres

Repository	TaskRepository.php	src/Repository/TaskRepository.php	Dépôt des tâches. Dans ce fichier, il est possible de créer des méthodes afin de récupérer certaines données en BDD
	UserRepository.php	src/Repository/UserRepository.php	Dépôt des utilisateurs. Dans ce fichier, il est possible de créer des méthodes afin de récupérer certaines données en BDD
DataFixtures	TaskFixtures.php	src/DataFixtures/TaskFixtures.php	Fichiers permettant d'ajouter automatiquement des tâches dans la base de données de l'applications Attention : Les tâches sont toujours ajoutées après les utilisateurs
	UserFixtures.php	src/DataFixtures/UserFixtures.php	Fichiers permettant d'ajouter automatiquement des utilisateurs dans la base de données de l'applications Attention : Les utilisateurs sont toujours ajoutés avant les tâches

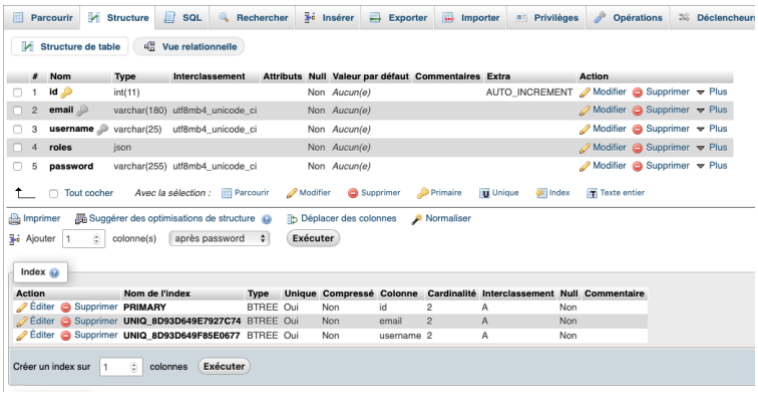
Fichiers publics

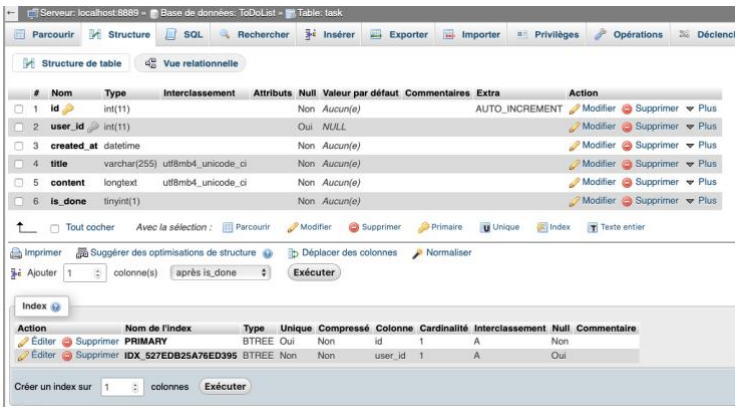
Styles	public/css	Dans ce fichier se trouvent tous les fichiers liés aux feuilles de style CSS
Polices	public/fonts	Dans ce fichier se trouvent tous les fichiers liés aux à la police
Images	public/img	Dans ce fichier se trouvent toutes les images
Javascript	public/js	Dans ce fichier se trouvent tous les fichiers liés aux script JS
Racine	public/index.php	Ce fichier va être le premier à être exécuté par le serveur. C'est lui qui va ensuite lancer les différentes fonctions du framework afin d'obtenir l'affichage des pages

Base de données

La base de données a pour nom : ToDoList
Elle est composée de deux tables :

- User
- Task



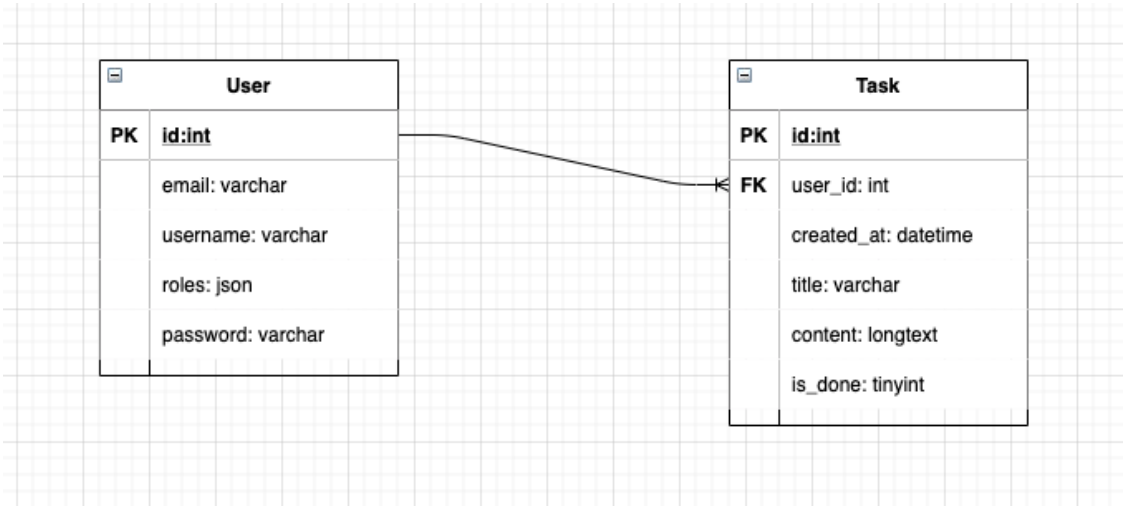


Configuration de la base de données

La configuration des champs se fait dans les fichiers des entités de chaque objet.
L'ORM utilise les annotations pour faire la configuration de la BDD.

```php
/**
 * @ORM\Entity(repositoryClass=UserRepository::class)
 * @UniqueEntity("email")
 */
class User implements UserInterface
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=180, unique=true)
     * @Assert\NotBlank(message="Vous devez saisir une adresse email.")
     * @Assert\Email(message="Le format de l'adresse n'est pas correcte.")
     */
    private $email;

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

    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];

    /**
     * @var string The hashed password
     * @ORM\Column(type="string")
     */
    private $password;

    /**
     * @ORM\OneToMany(targetEntity=Task::class, mappedBy="user")
     */
    private $tasks;
```

Modèle de données



Authentification

Fichiers

Configuration	Config/packages/security.yaml	Configuration de l'authentification
Entité	Src/Entity/User.php	Entité des utilisateurs
Contrôleur	Src/Controller/SecurityController.php	Contrôleur de connexion et déconnexion des utilisateurs
Authentification	Src/Security/LoginFormAuthentification.php	Processus d'authentification de l'application
Vue	Templates/security/login.html.twig	Template du formulaire de connexion

Configuration

La configuration du composant de sécurité s'effectue dans le fichier : *config/packages/security.yaml*
Les mots de passe sont cryptés par le composant de sécurité de Symfony. Pour modifier l'algorithme de cryptage, il faut modifier la section *encoders* du fichier de configuration.

```yaml
security:
    encoders:
        App\Entity\User:
            algorithm: auto
```

Les mots de passe sont cryptés grâce à une passphrase définie dans le fichier « *.env* » situé à la racine du projet (APP_SECRET).

Provider

Dans le fichier *config/packages/security.yaml*, il est possible de configurer plusieurs fournisseurs d'authentification.
L'entité User étant utilisée dans ce projet, l'authentification est configurée dans la section *providers*. Il est également nécessaire de définir l'attribut de la classe *User* qui sera utilisé comme identifiant de connexion.

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

Veuillez noter que le nom du provider doit également être renseigné dans la partie *main* du *firewall*

```yaml
  firewalls:
      dev:
          pattern: ^/(_(profiler|wdt)|css|images|js)/
          security: false
      main:
          anonymous: lazy
          provider: users_in_memory
          guard:
              authenticators:
                  - App\Security\LoginFormAuthenticator
          logout:
              path: logout
```

Pare-feu

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

- La classe chargée de réaliser l'authentification (Guard : authenticators)
- Le nom du provider utilisé (provider)
- La route permettant la déconnexion (logout)

Le pare-feu indique la partie du site qui 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*.

```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        anonymous: lazy
        provider: users_in_memory
        guard:
            authenticators:
                - App\Security\LoginFormAuthenticator
        logout:
            path: logout
```

Contrôle des accès

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

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

Sécurisation et mise à jour

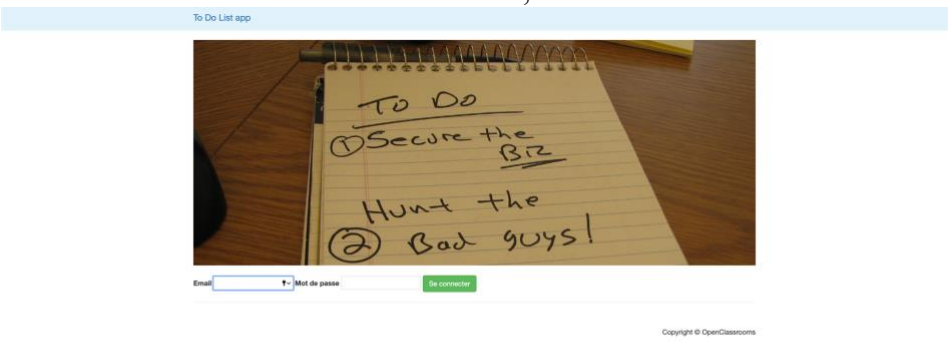
Lors du passage à la version 4.4 de Symfony, le formulaire de connexion utilisateur a également été mis à jour.
La clé unique est maintenant l'email et non plus le pseudo.

Ce choix a été fait puisque l'email est un identifiant de connexion bien plus actuel et une adresse mail est plus facilement unique sans trop restreindre la création d'un utilisateur.

L'authentification

Formulaire

Pour afficher le formulaire de connexion, il est nécessaire d'accéder à l'url **/login**



Soumission

1. L'utilisateur remplit le formulaire et le soumet grâce au bouton « Se connecter ».
2. Comme nous avons pu voir ci-dessus, dans la partie « Pare-feu », l'authentification est configurée pour utiliser le composant de sécurité de Symfony, qui est ensuite gérée par la classe *LoginFormAuthentificator* qui se trouve dans

src/Security/LoginFormAuthentificator.php.

```php
/**
 * Class LoginFormAuthenticator
 * @package App\Security
 * @codeCoverageIgnore
 */
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator implements PasswordAuthenticatedInterface
{
    use TargetPathTrait;

    public const LOGIN_ROUTE = 'login';

    private $entityManager;
    private $urlGenerator;
    private $csrfTokenManager;
    private $passwordEncoder;

    public function __construct(EntityManagerInterface $entityManager, UrlGeneratorInterface $urlGenerator, CsrfTokenManagerInterface $csrfTokenManager, UserPas
    {
        $this->entityManager = $entityManager;
        $this->urlGenerator = $urlGenerator;
        $this->csrfTokenManager = $csrfTokenManager;
        $this->passwordEncoder = $passwordEncoder;
    }

    public function supports(Request $request)
    {
        return self::LOGIN_ROUTE === $request->attributes->get('_route')
            && $request->isMethod('POST');
    }
```

3. La soumission est alors interceptée, puis les données envoyées via le formulaire sont stockées dans le tableau **$credentials** et la valeur email (champ unique) est ajoutée dans une variable de session.
   Un 3ème paramètre est envoyé, le paramètre « csrf_token » est une clé de sécurité envoyé par le formulaire de connexion permettant de protéger le formulaire contre le « *Cross site request forgery* » ou contre les éventuelles attaques par des tiers.

```php
    public function getCredentials(Request $request)
    {
        $credentials = [
            'email' => $request->request->get('email'),
            'password' => $request->request->get('password'),
            'csrf_token' => $request->request->get('_csrf_token'),
        ];
        $request->getSession()->set(
            Security::LAST_USERNAME,
            $credentials['email']
        );

        return $credentials;
    }
```

4. Après que le token CSRF a été vérifié, La valeur du champ email envoyée est utilisé pour rechercher un utilisateur associé dans la base de données grâce à l'utilisation de l'entité User et Doctrine.
   Si une correspondance est trouvée, l'utilisateur est stocké dans un objet $user, sinon une exception est levée.

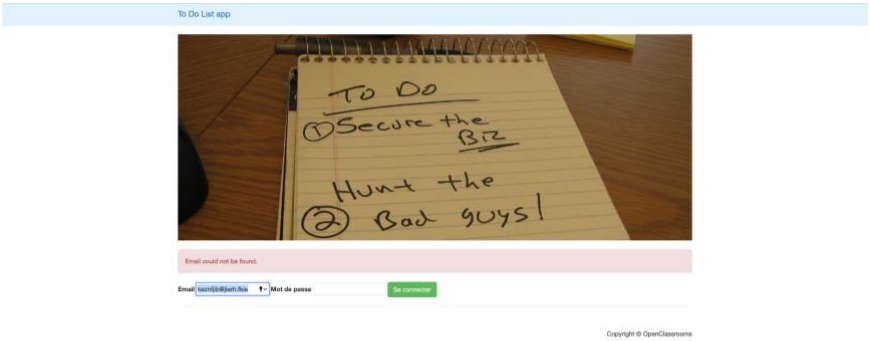
```php
    public function getUser($credentials, UserProviderInterface $userProvider)
    {
        $token = new CsrfToken('authenticate', $credentials['csrf_token']);
        if (!$this->csrfTokenManager->isTokenValid($token)) {
            throw new InvalidCsrfTokenException();
        }

        $user = $this->entityManager->getRepository(User::class)->findOneBy(['email' => $credentials['email']]);

        if (!$user) {
            // fail authentication with a custom error
            throw new CustomUserMessageAuthenticationException('Email could not be found.');
        }

        return $user;
    }
```

5. Le mot de passe crypté envoyé est testé pour vérifier sa correspondance avec celui de l'utilisateur chargé.

```php
public function checkCredentials($credentials, UserInterface $user)
{
    return $this->passwordEncoder->isPasswordValid($user, $credentials['password']);
}
```

6. Une fois toutes les informations réunies, vérifiées et qui correspondent parfaitement, alors l'utilisateur est connecté et une redirection est effectuée vers la dernière url visitée, si elle existe ou vers la liste des tâches.
   En cas d'échec la fonction *login()* du contrôleur « *SecurityController* » est appelée à nouveau, et le formulaire de connexion est chargé, avec l'erreur affichée.

```php
public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
        return new RedirectResponse($targetPath);
    }

    // For example : return new RedirectResponse($this->urlGenerator->generate('some_route'));
    return new RedirectResponse($this->urlGenerator->generate('homepage'));
}

protected function getLoginUrl()
{
    return $this->urlGenerator->generate(self::LOGIN_ROUTE);
}
```



Gestions des rôles

Les rôles de chaque utilisateur sont définis dans le champ « roles » de la table *user*. Cette information est accessible après avoir instancié l'objet User avec la méthode *getRoles()*.

Pour autoriser ou refuser l'accès à une route ou une fonctionnalité, plusieurs solutions sont possibles ; elles sont toutes disponibles via le lien suivant : https://symfony.com/doc/4.4/security.html#roles

Dans ce projet, les restrictions aux pages sont toutes configurées dans le fichier *config/packages/security.yaml* dans la section « *access_control* ».

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

Installation du projet

1. Assurez-vous d'avoir Git installé et à jour sur votre machine :
   www.git-scm.com

2. Cloner le repository sur votre serveur local
   git clone https://github.com/damientabet/ToDoList.git

3. Bien s'assurer que composer est installé et à jour sur votre machine :
   www.getcomposer.org/doc/00-intro.md

4. Après avoir installé composer, veuillez lancer *composer install* à la racine de votre projet. Toutes les dépendances vont s'installer et se stocker dans le dossier **/vendor**.

5. Modifier les accès à votre base de données dans le fichier .env *DATABASE_URL (l.28)*.

6. Créer la base de données en utilisant exécutant la commande suivante :
   php bin/console doctrine:database:create

7. Créer les tables dans la base de données en utilisant exécutant la commande suivante :
   php bin/console doctrine:schema:create

8. Installer les data fixtures afin de pouvoir interagir avec le site.
   php bin/console doctrine:fixtures:load

9. Si vous êtes en local, vous pouvez démarrer le serveur de Symfony avec la commande :
   php bin/console server:run

10. Si vous installez le projet sur une serveur, n'oubliez pas de faire pointer votre domaine vers le fichier */public*

11. Voici les identifiants du compte administrateur pour pouvoir ajouter de nouveaux utilisateurs et/ou tâches.
    a. Email : administrateur@todolist.com
    b. Mot de passe : admintest

Normes

Certaines normes de codage doivent être respecter afin de mener le projet à bien.
Pour cela, il est nécessaire de suivre les points présents dans la documentation Symfony via le lien suivant :
https://symfony.com/doc/4.4/contributing/code/standards.html#structure

Tests unitaires et fonctionnels

Tests Contrôleurs	DefaultControllerTest	tests/Controller/DefaultControllerTest.php	Tests unitaires de la page d'accueil
	RegistrationControllerTest	tests/Controller/RegistrationControllerTest.php	Tests unitaires de la création utilisateur
	SecurityControllerTest	tests/Controller/SecurityControllerTest.php	Tests unitaires de la connexion utilisateur
	TaskControllerTest	tests/Controller/TaskControllerTest.php	Tests unitaires de la gestion des tâches
	UserControllerTest	tests/Controller/UserControllerTest.php	Tests unitaires de la gestion des utilisateurs
Tests Formulaire	FormTypeTest	tests/Form/FormTypeTest.php	Tests unitaires des formulaires de l'application

Documentation Symfony

La documentation de Symfony est très bien expliquée et possède avec beaucoup d'exemples. Pour toutes questions ou pour comprendre certaines parties du framework, ne pas hésitez pas à suivre le lien suivant : https://symfony.com/doc/4.4/index.html