**Documentation Technique du site :**

TOdoList

[**1 - Présentation du site :**](#_pd5q0o68g6y9) **2**

[1.1 - Présentation générale :](#_l3w7xfrvfw66) 2

[1.2 - Arborescence :](#_97tnwpfvhe68) 2

[1.3 - Diagramme de classes :](#_h3p1c75elj8w) 3

[**2 - Fonctionnement général du site :**](#_6ikreo8otgo) **3**

[2.1 - Les controllers :](#_uawt5x88j3hb) 3

[2.1.1 - DefaultController.php](#_xlfjalj282dz) 3

[2.1.2 - UserController.php](#_l2xrke1d8v1n) 4

[2.1.3 - TaskController.php](#_5djf392009sw) 4

[2.1.4 - SecurityController.php](#_iccc3sfunai5) 6

[2.2 - Les entités](#_bbmyzhty8sp7) 6

[2.2.1 - User.php](#_2vwbt04te2ce) 6

[2.2.2 - Task.php](#_q7vrdk94d4br) 7

[2.3 - La sécurité :](#_mk9zh6u0bue4) 8

[2.3.1 - Rappels généraux sur la sécurité Symfony](#_l7rkwpfqho2e) 8

[2.3.2 - L’implémentation sur le site :](#_n8vocdi99vn5) 9

[2.4 - Les tests :](#_rsa6yt82oz0x) 11

[**3 - Bibliographie :**](#_pgbl5oaepwz1) **14**

# 1 - Présentation du site :

## 1.1 - Présentation générale :

TodoList est un site internet dédié à la mise en place de listes. Le site à été développé à l’aide du framework “Symfony” dans sa version 3.1.

Il a été développé rapidement afin de fournir une M.V.P. dans les délais les plus brefs possibles, ce qui implique qu’il s’agit d’une version minimaliste, qui doit servir de “proof of concept”.

## 1.2 - Arborescence :



Schéma numéro 1

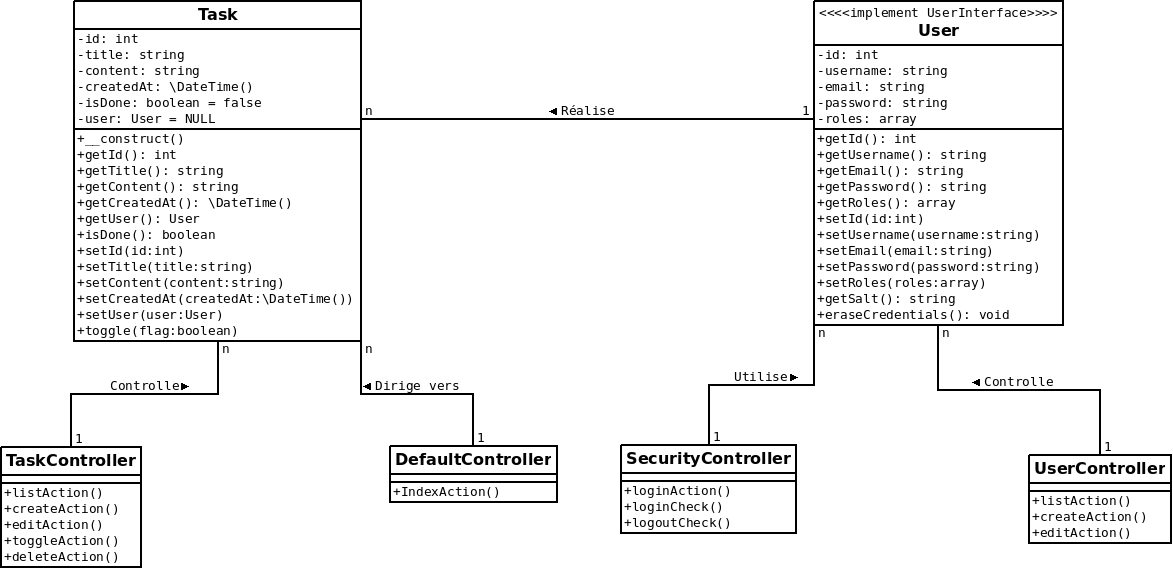
L’architecture de base du site, permet à tout utilisateur disposant du rôle “ROLE\_USER”, d’accéder à différentes actions possibles (schéma N°1). Parmis ces actions, nous pouvons citer :

* La Connection / Déconnexion.
* La page de création d’une tâche.
* La page de consultation des tâches à réaliser.
* La page de consultation des tâches réalisées.

Il est à noter que la gestion des utilisateurs se fera grâces aux URI “/users” pour lister les utilisateurs inscrits, “/users/create” pour créer un nouvel utilisateur et “/users/id\_de\_l\_user/edit” pour modifier un utilisateur.

Ces pages de gestion n’ont pas de liens sur le site, et ne sont accessibles qu’aux utilisateurs disposant du rôle : “ROLE\_ADMIN”.

## 1.3 - Diagramme de classes :



# 2 - Fonctionnement général du site :

## 2.1 - Les controllers :

Tous les contrôleurs étendent la classe “Symfony\Bundle\FrameworkBundle\Controller\Controller”. Ceci correspond au fonctionnement normal d’une application Symfony. Il est optionnel et apporte quelques facilités dans le développement de l’application et permet par exemple d'accéder facilement au conteneur de services. *Pour en savoir plus :* [*https://symfony.com/doc/3.1/controller.html*](https://symfony.com/doc/3.1/controller.html)

### 2.1.1 - DefaultController.php

**Namespace :** AppBundle\Controller

**indexAction :** “/”

Affiche la page d’accueil du site, si l’utilisateur n’est pas identifié, redirige vers “/login”.

### 2.1.2 - UserController.php

**Namespace :** AppBundle\Controller

Important : Les URI derrière “/users” nécessitent d’être identifié, et de disposer du rôle “ROLE\_ADMIN”. Si l’utilisateur n’est pas identifié, il sera redirigé vers la page “/login”, si il n’a pas les autorisations pour accéder à une URI, une erreur “403 Forbidden” est levée.

**listAction :** “/users”

Affiche la liste des utilisateurs inscrits sur le site.

**createAction:** “/users/create”

Affiche le formulaire d’inscription pour un nouvel utilisateur. Dans ce cas, on fait appel au formbuilder de Symfony, et à la classe “AppBundle/Form/UserType” pour définir le formulaire. Cette page est rappelée en POST par le formulaire, vérifie la validité des informations transmises, grâce au formbuilder et à son composant validator, lui même configuré via les annotations dans l’entité “AppBundle/Entity/User”. Les données valides sont ensuite persistées en base de donnée via Doctrine.

**editAction :** “/users/id\_user/edit”

Affiche le formulaire de modification d’un utilisateur, cette page est rappelée en POST par le formulaire, vérifie la validité des informations transmises, et persiste les informations en base de données. Le fonctionnement est similaire à la méthode “createAction”.

### 2.1.3 - TaskController.php

**Namespace :** AppBundle\Controller

**listAction :** “/tasks”

Affiche la liste des tâches appartenant à un utilisateur. Pour cela, on récupère les tâches via la méthode “findAllForUser” du répository “AppBundle/Repository/TaskRepository”.

**createAction :** “/tasks/create”

Permet de créer une tâche. Cette méthode est appelée :

* Pour afficher le formulaire de création, dans ce cas, on fait appel au formbuilder de symfony et à la classe “AppBundle/Form/TaskType” pour définir le formulaire.
* Pour valider le formulaire et enregistrer les données en base de données. Ceci, grâce au formBuilder de Symfony et son composant validator, lui même configuré via les annotations dans l’entité “AppBundle/Entity/Task”.

**editAction :** “/tasks/{id}/edit”

Permet la modification d’une tâche. Tout comme pour la méthode createAction, cette méthode peut, de la même manière, être appelée pour l’affichage du formulaire, ou pour valider et persister les données en base de données.

**toggleAction :** “/tasks/{id}/toggle”

Cette action a pour rôle de marquer une tâche comme faite, ou non faite. Pour cela, nous agissons sur l’attribut “isDone” de la classe “AppBundle/Entity/Task”, via la méthode “toggle()”. Le résultat est persisté via Doctrine.

Cette méthode “toggle()” se contente d’inverser le booléen “isDone”. S’il vaut TRUE, la tâche est marquée réalisée, s’il vaut FALSE, la tâche est marquée comme non réalisée.

**deleteTaskAction :** “/tasks/{id}/delete”

Cette action a pour but de supprimer une tâche. Il est d’abord vérifié que la tâche que nous cherchons à supprimer appartient bien à l’utilisateur courant. Dans le cas ou la tâche ne serait pas rattachée à un utilisateur, on vérifie que l’utilisateur courant ait bien le rôle “ROLE\_ADMIN”. Si ces conditions ne sont pas satisfaites, un message est renvoyé, et la suppression n’a pas lieu. Dans le cas contraire, la suppression est effectuée, et un message de confirmation est renvoyé.

L’utilisateur est ensuite redirigé vers la liste des tâches.

### 2.1.4 - SecurityController.php

**Namespace :** AppBundle\Controller

Cette classe est utilisée pour l'authentification et sera plus détaillée dans le chapitre sécurité de cette documentation.

**loginAction :** “/login”

Ce contrôleur se contente de récupérer les éventuels messages d’erreur concernant l’authentification, et de les renvoyer à la vue “security/login.html.twig”.

Cette vue est juste un formulaire en dur, destiné à l’authentification. Il renvoie sur la route nommée “login\_check”. Cette méthode est vide dans le controlleur, mais est interceptée par symfony. Plus de détails sur l’authentification, dans le chapitre qui lui est consacré.

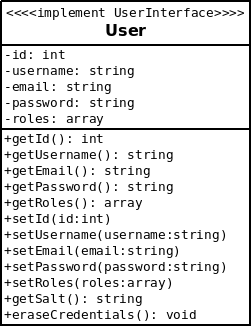
## 2.2 - Les entités

### 2.2.1 - User.php

**NameSpace :** AppBundle\Entity\User

Cette entité représente un utilisateur. Afin de pouvoir servir à l'authentification via Symfony, elle implémente “Symfony\Component\Security\Core\User\UserInterface”, condition obligatoire pour le processus.

**Diagramme de l’entité :**



Les attributs *username, password et roles,* servent à l’authentification. Les méthodes *getRoles(), getPassword(), getSalt(), getUsername(), eraseCredentials(),* sont requises pour le processus d’authentification, et sont imposées par l’interface “UserInterface”.

Les autres méthodes sont les getters et setters classiques pour une entité.

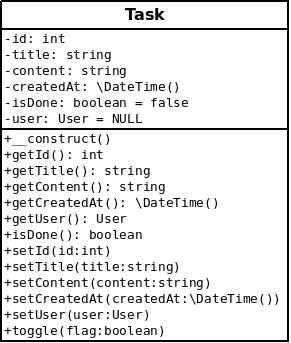
Les commentaires sur la classe et les attributs servent à la configuration de Doctrine pour la gestion de la base de données, et au formbuilder pour les assertions qui gèrent la sécurité des formulaires.

### 2.2.2 - Task.php

**NameSpace :** AppBundle\Entity\Task

Cette entité représente une tâche, réalisée ou à réaliser par l’utilisateur.

**Diagramme de l’entité :**



Un tâche est caractérisée par son *id, son titre, son contenu, sa date de création, sa réalisation ou non, et l’utilisateur propriétaire de la tâche*. Toutes ces caractéristiques sont des attributs de la classe et disposent des méthodes courantes d'accès et de modifications “getters et setters”.

**Quelques particularités :**

**Le constructeur “\_\_construct()”**, est là pour initialiser par défaut la date de création de la tâche à la valeur de la date courante, et l’attribut *isDone* à false (par défaut une tâche est marquée comme non réalisée lors de sa création).

**La méthode “toggle()”**, est ici le setter de l’attribut *isDone*.

**La méthode “isDone()”** est quand à elle, le getter de l’attribut *isDone*.

L’attribut *User* représentant l’entité utilisateur propriétaire de la tâche, le setter “**setUser()**” attend donc en paramètre une entité “AppBundle\Entity\User”.

Pour finir, **la relation** de “AppBundle\Entity\Task” vers “AppBundle\Entity\User” a été définie comme étant de type “ManyToOne”, en effet, il faut s’attendre logiquement à ce qu’un utilisateur puisse avoir plusieurs tâches en cours.

*En savoir plus sur les entitées et Doctrine ORM :* [*https://symfony.com/doc/3.1/doctrine.html*](https://symfony.com/doc/3.1/doctrine.html)

*En savoir plus sur les assertions et les formulaires :* [*https://symfony.com/doc/3.1/forms.html*](https://symfony.com/doc/3.1/forms.html)

## 2.3 - La sécurité :

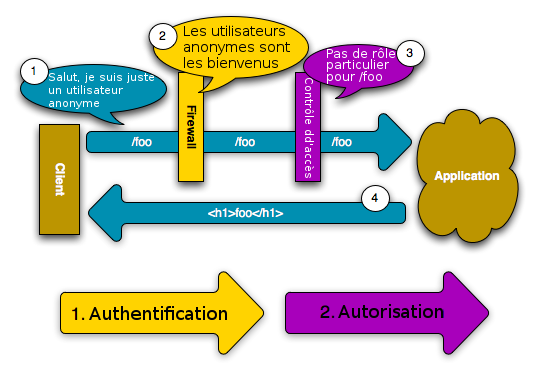
### 2.3.1 - Rappels généraux sur la sécurité Symfony

La sécurité Symfony est très puissante et permet une personnalisation à l'extrême. Il n’est pas ici question de faire un cours sur ce composant. Pour bien comprendre la mise en place et la configuration sur le site, il convient tout de même de faire un petit rappel.

Il faut différencier l’autorisation, et l’authentification :

* L’authentification, c’est le processus qui vise à savoir qui est le visiteur. Il est géré par le firewall de Symfony. On peut par exemple, choisir de n’autoriser l'accès à une certaine partie du site qu’aux utilisateurs authentifiés sur le site sans particulièrement tenir compte de leur niveau d’autorisation. Ou bien, à l’inverse afficher certaines choses sur la page, ou autoriser l'accès à une certaine partie du site aux utilisateurs non authentifiés : les utilisateurs anonymes.
* L’autorisation est le processus par lequel on va définir si un utilisateur authentifié dispose des droits nécessaires pour accéder à la ressource demandée. Il est géré par l’acces controll de Symfony. C’est ici qu’interviennent les roles, “ROLE\_ADMIN” ou “ROLE\_USER” sont par exemple très souvent utilisés, car très parlant, mais il est possible de créer n’importe quel rôle.

**Schéma représentant le processus authentification / autorisation de Symfony :**



*source :* [*openclassrooms*](https://openclassrooms.com/courses/developpez-votre-site-web-avec-le-framework-symfony/securite-et-gestion-des-utilisateurs-1) *licence creative commons*

*En savoir plus sur le composant sécurité :* [*https://symfony.com/doc/3.1/security.html*](https://symfony.com/doc/3.1/security.html)

### 2.3.2 - L’implémentation sur le site :

**Configuration :**

La configuration s’effectue via le fichier “*app/config/security.yml*”.

Sous la clef “security:” nous retrouvons différentes clefs définissant les différents paramètres du composant :

* **encoders :** Défini le type d’encodage pour les mots de passe. Dans notre cas, la méthode de cryptage “bcrypt” a été choisie pour l’entité “AppBundle\Entity\User”. Nous aurions aussi pu choisir “plaintext” (text en clair), ou bien “sha512” par exemple.
* **role\_hierarchy :** Défini les différents rôles et leurs relations. Dans notre cas, nous voyons que le rôle “ROLE\_ADMIN” dispose aussi des droits du rôle “ROLE\_USER”.
* **providers :** Indique au service à quel endroit retrouver les informations nécessaires au processus. Ici, il s’agit de l’entité Doctrine “*AppBundle\Entity\User*”, l’attribut retenu comme identifiant étant : *username*
* **firewalls :** Défini les paramètres des différents firewalls Symfony. Par exemple, nous retrouvons le firewall “dev:” qui permet l’accès aux fichiers css, images et js. Ou le firewall “main:” qui active la firewall pour toutes les URI du site, en interdisant les connexions anonymes, et redirige les utilisateurs non authentifiés vers la page de login. A savoir que login\_check et logout sont ici interceptés par symfony, et traités par défaut via un listener présent dans Symfony. **Attention,** l’ordre des firewalls est important, le premier firewall dont la route correspond, fourni ces paramètres pour la requête en cours.
* **access\_control :** Défini quant à lui les paramètres concernant les droits d'accès nécessaires à l'accès aux ressources du site. Par exemple, pour accéder à l’URI “/login”, le role : “IS\_AUTHENTICATED\_ANONYMOUSLY” sera requis (rôle attribué par défaut aux utilisateurs reconnus anonymes par le firewall, donc non identifiés), le rôle : “ROLE\_ADMIN” sera requis pour toutes les routes derrière l’URI “/users”, et le rôle : “ROLE\_USER” sera requis pour toutes les autres routes de l’application. **Attention,** l’ordre est important, la première route qui correspond au paramètre “path:” de la ligne d’access\_control fera fois pour l’access control. Ce qui veut dire que, si la ligne : - { path: ^/, roles: ROLE\_USER } avait été en premier, toutes les routes du site auraient nécessité le rôle “ROLE\_USER”, ce qui aurait rendu impossible l'accès à la route “/login” par exemple.

**SecurityController :**

**NameSpace :** AppBundle\Controller

Ce contrôleur fait partie intégrante du processus de sécurité Symfony en fournissant des contrôleurs à mettre en face des routes directement liées à la sécurité.

La classe “SecurityController” contient ici 3 méthodes :

* loginAction(), qui, comme détaillée au chapitre des contrôleurs, s’occupe d’afficher le formulaire d’identification lors de l’appel à l’URI “/login”.
* loginCheck() et logoutCheck() qui sont deux méthodes laissées vide, pour laisser le comportement par défaut de Symfony qui effectue ces actions directement via un listener.

## 2.4 - Les tests :

Les tests ont été rédigés pour être exécutés via phpunit. Une fois phpunit installé, il est possible de les exécuter avec la commande : “phpunit” dans un terminal, situé à la racine du projet.

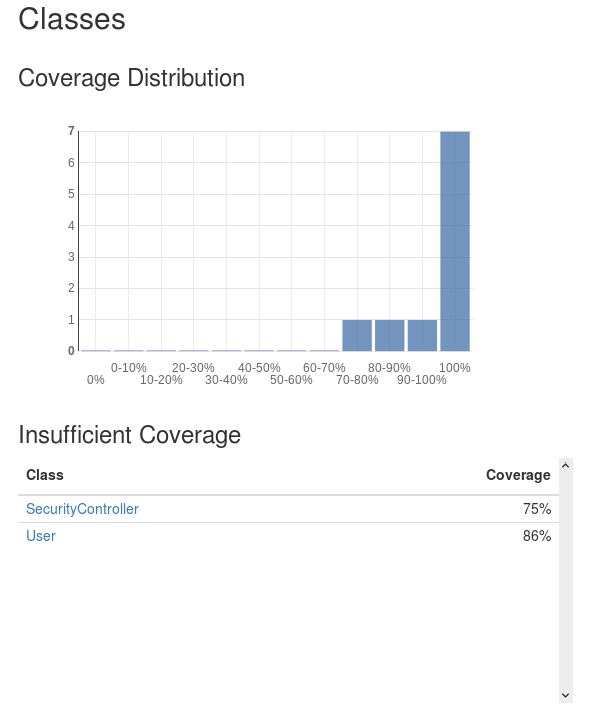
Il est également possible de vérifier le code-coverage en exécutant la commande : “phpunit --coverage-html ./coverage”, toujours dans un terminal situé à la racine du projet. Dans ce cas, vous aurez un ensemble de fichiers html, consultables en ouvrant ./coverage/index.html. L’objectif de cette manipulation est de vérifier quel est le taux de couverture des tests, et quelles parties du code ne sont pas testées.

A l’heure de la rédaction de cette documentation technique, et une fois mis en place les correctifs demandés, le taux de couverture des tests est de 95.90 % du code.

Nous allons voir juste après que ce taux est tout à fait correct, et aussi voir quels sont les points qui viennent baisser le taux, et pourquoi.

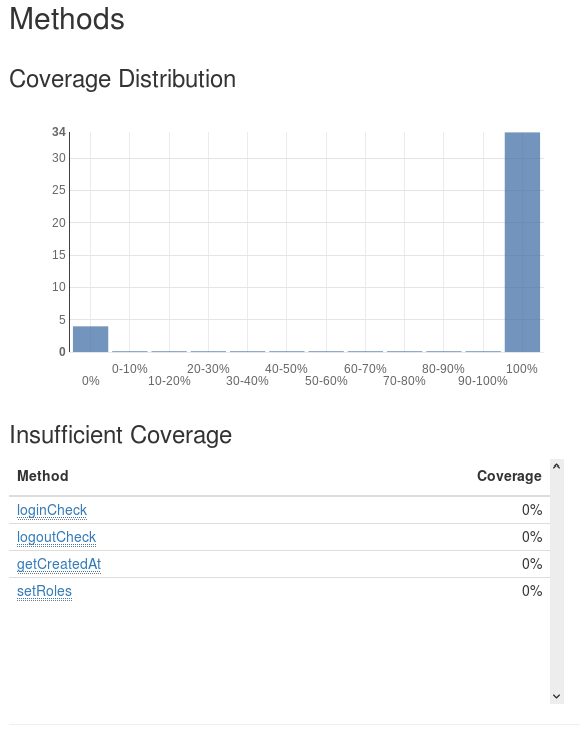
*En savoir plus sur phpunit :* [*https://phpunit.de/*](https://phpunit.de/)

*En savoit plus sur phpunit au sein de Symfony :* [*https://symfony.com/doc/3.1/testing.html*](https://symfony.com/doc/3.1/testing.html)



Comme le montre le graphique ci dessus, toutes les classes montrent un code-coverage supérieur à 75 %. Cependant, il est recommandé, pour obtenir un très bon taux de couverture du code, de se situer en dessus de 90%. Deux classes apparaissent donc en dessous de ce seuil, dans la section “Insufficient Coverage”. Il s’agit de *SecurityController* et *User*.

Voyons donc la raison de ceci.



Si nous regardons plus en détail le coverage par méthode, nous nous rendons compte que 4 méthodes ne sont pas testées. Il s’agit de :

* **loginCheck()**, **logoutCheck()** : Ces deux méthodes font parti de la classe SecurityController et viennent donc mathématiquement baisser le taux de couverture de cette classe. Il n’est cependant pas nécessaire de tester des méthodes laissées vides.
* **getCreatedAt(), setRoles()** : Ce sont des getters et setters de l’entité “*AppBundle\Entity\User*”. Ceci vient donc aussi mathématiquement diminuer le taux de couverture de cette classe. Cependant, les getters, et setters des entitées sont des méthodes tellement simples, que beaucoup considèrent qu’il n’est pas nécessaire de les tester. Attention cependant, ces méthodes seraient un peu plus complexe que le simple fait d’accéder ou de définir la valeur d’un attribut (sans le modifier), il serait alors recommandé de la tester.

Nous voyons donc que les quelques points qui baissent le taux de couverture sont des détails qui trouvent une justification logique au fait qu’ils n’ont pas été testés.

# 3 - Bibliographie :

* Documentation Symfony : [*https://symfony.com/doc/current/index.html*](https://symfony.com/doc/current/index.html)
* Les controleurs Symfony : [*https://symfony.com/doc/3.1/controller.html*](https://symfony.com/doc/3.1/controller.html)
* *Les entitées et Doctrine ORM :* [*https://symfony.com/doc/3.1/doctrine.html*](https://symfony.com/doc/3.1/doctrine.html)
* *Les assertions et les formulaires :* [*https://symfony.com/doc/3.1/forms.html*](https://symfony.com/doc/3.1/forms.html)
* *Le composant sécurité :* [*https://symfony.com/doc/3.1/security.html*](https://symfony.com/doc/3.1/security.html)
* *Phpunit au sein de Symfony :* [*https://symfony.com/doc/3.1/testing.html*](https://symfony.com/doc/3.1/testing.html)
* *Phpunit :* [*https://phpunit.de/*](https://phpunit.de/)
* [Cour Openclassrooms en accès libre](https://openclassrooms.com/courses/developpez-votre-site-web-avec-le-framework-symfony)