****

**Attendu :**

Les fondateurs souhaitent pérenniser le développement de l’application. Cela dit, ils souhaitent dans un premier temps faire un état des lieux de la dette technique de l’application.

Au terme de votre travail effectué sur l’application, il vous est demandé de produire un audit de code sur les deux axes suivants : la qualité de code et la performance.

**Partie : Présentation**

**Partie : Audit de qualité du code**

**Partie : Performance de l’application**

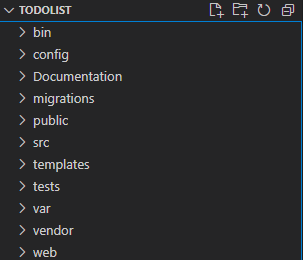
**Partie : Tests automatisés**

**Partie : Comment contribuer**

**Partie : Fonctionnement Authentification**

PRESENTATION

* **Architecture.**



**Bin**: Contient les fichiers de commandes permettant d’effectuer des actions sur un projet Symfony.

**Config** : Configuration des packages, services et routes (YAML)

**Migration** : Contient les fichiers de migrations Doctrine -> BDD

**Public** : Point d’entrée de l’application, index.php. Contient les images.

**SRC** : Cœur du projet ! Dossier qui contient la logique de votre application.

**Templates** : Contient nos Views. Symfony utilise le moteur de Template Twig par défaut.

**Tests**: Contient les fichiers de test pour PHPUNIT

**Var** : Cache et fichiers de log.

**WEB**: Contient les résultats (rapport de couverture) des tests PHPUNIT

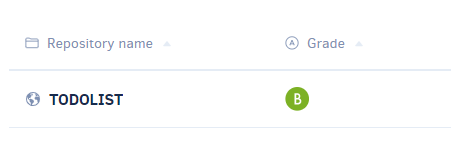
**Vendor** : Packages de Symfony listés dans le Fichier Composer.json

* **Principaux Paquet :**
* **Annotation :** Permet de définir les routes en commentant le code avec #
* **Doctrine :** Doctrine est l'ORM  ( object-relational mapping ) par défaut du framework Symfony.
* **EazyAdmin :** Générateur de page d’administration
* **Email-validator :** Permet la validation d’un compte par l’envoi d’un mail (via google-mailer)
* **Flex :** Pluggin Composer permettant l’installation de paquets supplémentaire
* **Form :** Permet de créer, traiter et réutiliser facilement des formulaires HTML
* **Google-Mailer :** Permet l’envoi de mail via un compte Gmail
* **Security-Bundle :** Intégration des composants de sécurité qui permet notamment la création des Controller d’enregistrement et d’authentification des utilisateurs
* **Password-hasher :** Permet l’encryptage des mots de passe.
* **PHPUnit :** Paquet permettant d’effectuer les tests unitaires.
* **TWIG :** Le moteur de templates pour le langage de programmation PHP, utilisé par défaut par le framework Symfony.

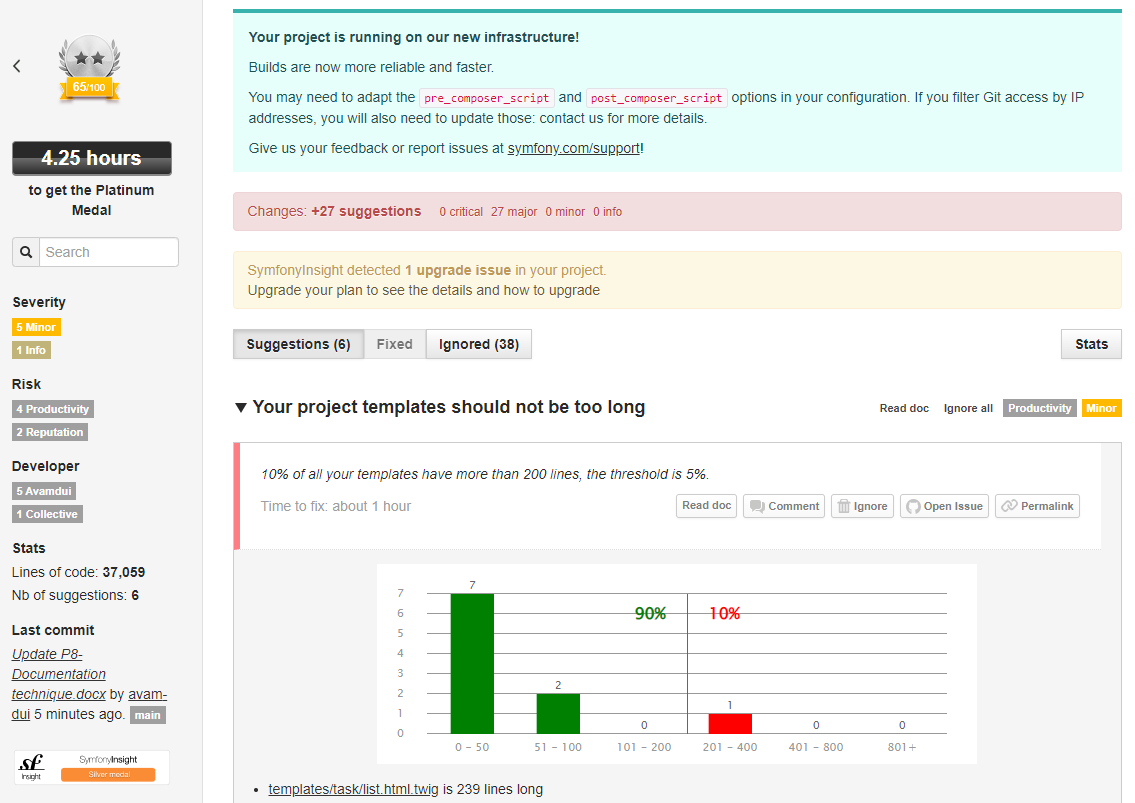
La liste complète des paquets ainsi qu’une description est disponible dans le fichier Paquet.md ou peut être obtenu par la commande *Composer info*

AUDIT DE QUALITE DU CODE

* Analyse du code.



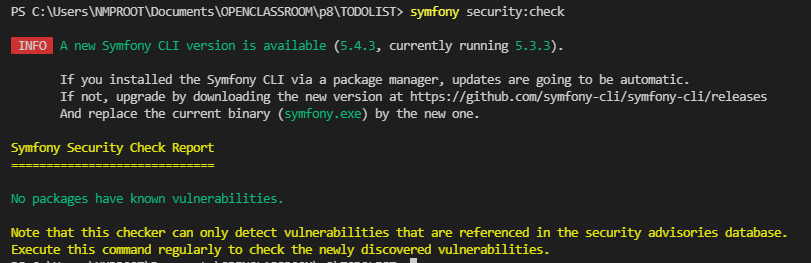
Le projet n’obtient que la note de B sous Codacy. Il y a donc des améliorations à faire. Cependant une grande partie des ISSUES sont dus à l’utilisation des PACKAGES tel que EAZYADMIN et des librairies tels que BOOTSTRAP et JQUERY.



Le projet obtient la médaille d’argent sous Symfony insight, qui nous fait remarquer que nos Templates sont trop gros, ceci étant principalement dus à l’affichage des Taches sous forme de KANBAN qui représente la grande partie du site.

* Sécurité.

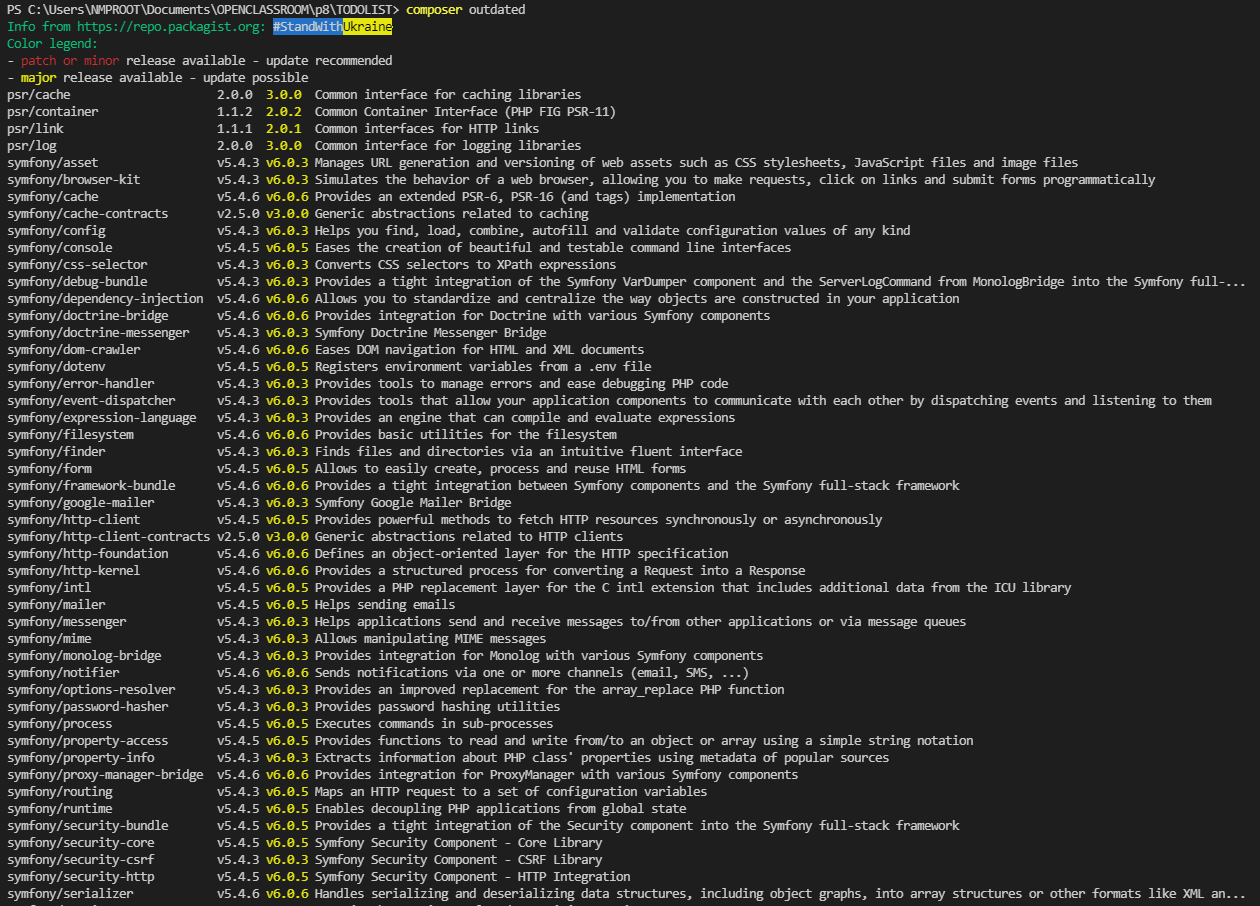
Commande : *symfony security:check*



Pas de vulnérabilité détecté par le rapport de sécurité Symfony.

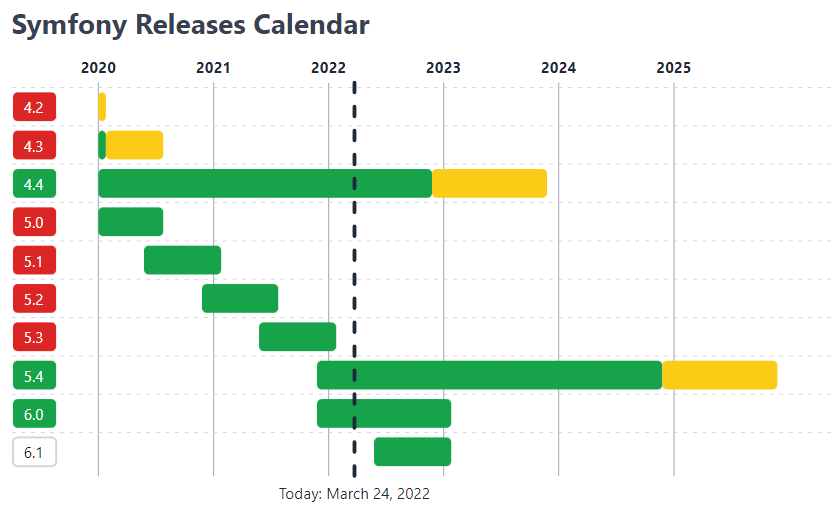
* **Mise à jour.**

Commande : *composer outdated*



Mise à jour possible vers Symfony 6. De nombreux paquets peuvent ainsi être mise à jours se qui tendrai à renforcer la sécurité et la qualité du code.

Cependant nous resterons sur la versions 5.4 qui est la LTS (Long-Term Support Release).



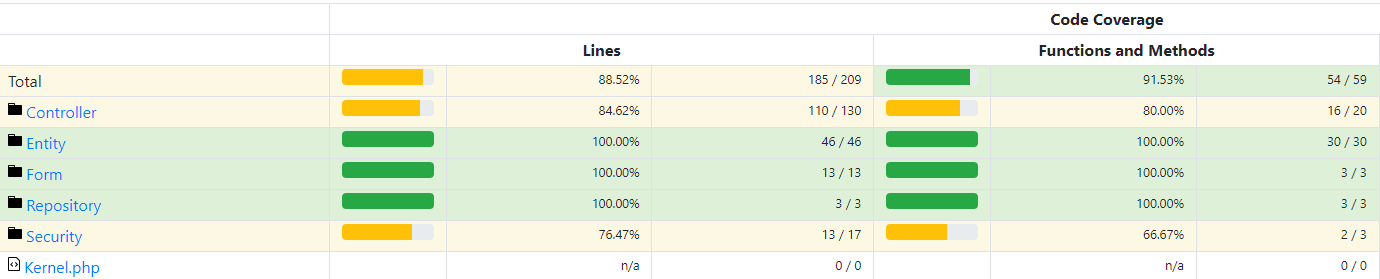
* **Test unitaire**

Écriture d’une série de tests unitaires et fonctionnels afin de garantir le fonctionnement de l’application.

Ci-dessous l’exemple du test de création de tache.

* public function testCreateAction()
* {
* // Login grâce a la fonction loginuser()
* $userRepository = static::getContainer()->get(UserRepository::class);
* $userTest = $userRepository->findOneByEmail('test@test.fr');
* $this->client->loginUser($userTest);
* // Nous accédons à la page "mes taches" de l'user ayant l'ID 1
* $crawler = $this->client->request('GET', '/tasks/1');
* $this->assertEquals(200, $this->client->getResponse()->getStatusCode());
* //Recherche du bouton "ajouter"
* $buttonCrawlerNode = $crawler->selectButton("Ajouter");
* // Sélection du formulaire associé a ce bouton
* $form = $buttonCrawlerNode->form();
* // On remplit le formulaire
* $form['task[title]'] = 'PHPUNIT';
* $form['task[content]'] = 'Faire les test unitaires';
* //On soumet le formulaire
* $this->client->submit($form);
* // Vérification de la redirection et du code retour 200
* $this->assertTrue($this->client->getResponse()->isRedirect());
* $crawler = $this->client->followRedirect();
* $this->assertEquals(200, $this->client->getResponse()->getStatusCode());
* }

A l’issue des taches, PHPUNIT génère un rapport de couverture. Nous couvrons ainsi près de 92% des fonctions et méthodes du site. La couverture de 100% des cas n’est pas nécessaires, mais il est important de couvrir l’intégralité des fonctions clefs du site, à savoir la gestion des taches.



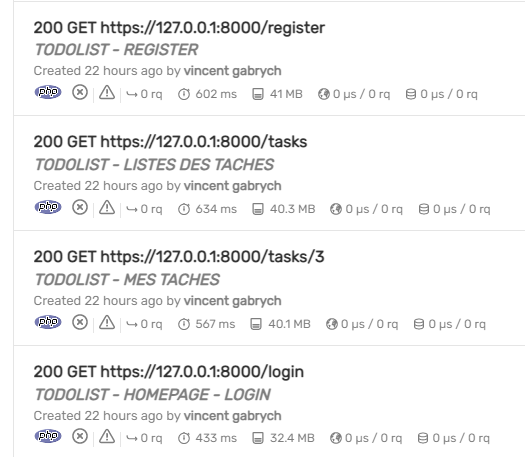
**PERFORMANCE DE L’APPLICATION**

* Tests de performance

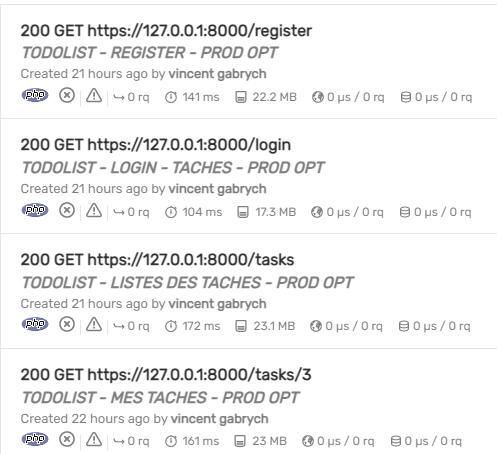
Comme conseillé par Symfony, la performance de l’application « To Do List » est analysée grâce à l’application professionnelle Blackfire.io.

Blackfire Profiler est un outil qui instrumente les applications PHP pour collecter des données sur les ressources serveur consommées comme la mémoire, le temps CPU et les opérations d'E / S

* En Dev avec profiler :



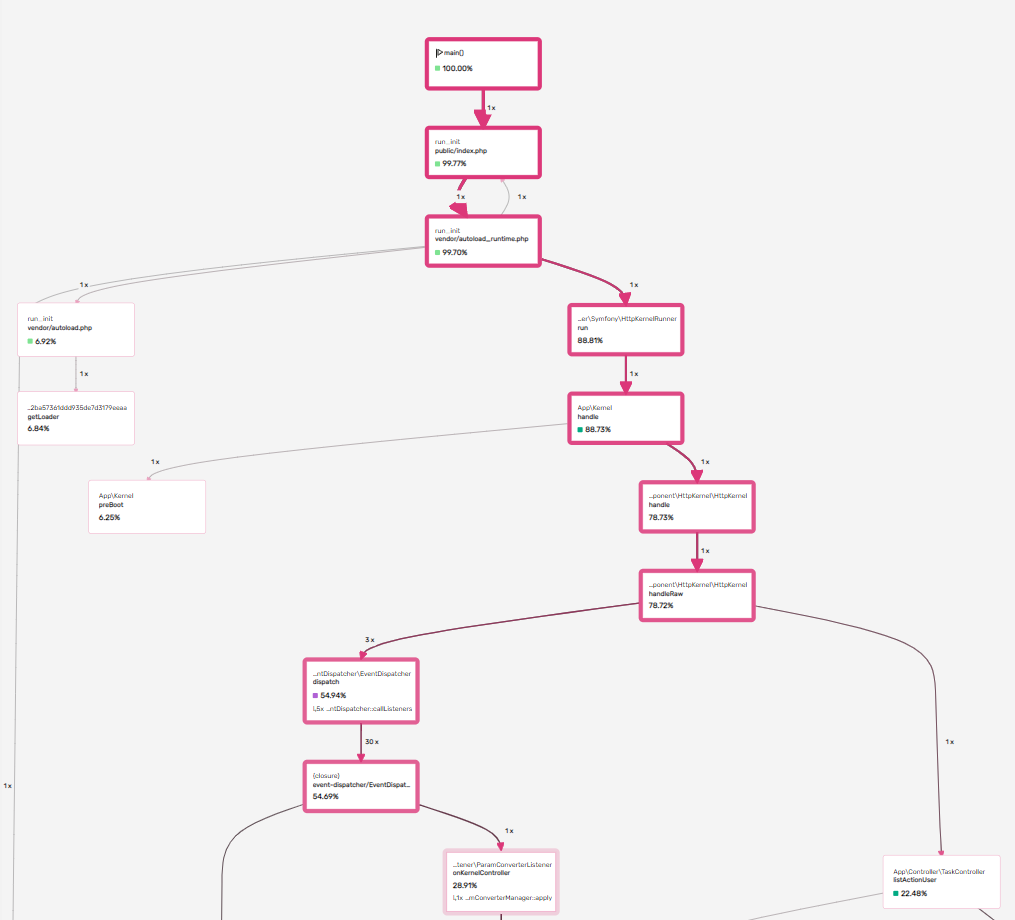
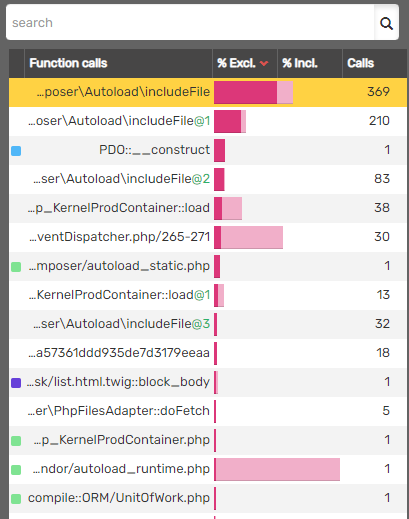
* Passage en Production :



Le passage en production, avec la désactivation du Profiler à un impact majeur sur les performances.

Ainsi la page MES TACHES passe de 567ms à 161ms en temps de chargement.

* Analyse :



Suite à une analyse des graph Blackfire, on constate que le temps de chargement est principalement dû à l’utilisation de l’autoload de Composer.

En effet, l’autoloader de Composer utiliser lors du développement de l'application est optimisé pour rechercher les classes nouvelles et modifiées. Et cela suppose d’aller chercher à chaque requête toutes les classes de l’application.

La documentation de Composer offre une solution à ce problème via la génération d’une “Class Map”.

Cela convertit les namespaces PSR-4 en ClassMap ce qui évite de faire appel au filesystem pour vérifier l'existence des classes.

Cette optimisation se fait via la commande **composer dump-autoload --optimize**.

**CONTRIBUER**

**# TODOLIST**

**# Avant propos**

- Le projet fonctionne sur PHP 8.0.15

- Le projet est basé sur le framework symfony 5.4.4 (Doctrine, Twig et PhpUnit)

- Git du projet : `git clone https://github.com/avamdui/TODOLIST`

**# Comment Contribuer au projet**

1. Cloner et Installer le repository sur votre serveur (voir le README.md)

  -  Modifier le .env avec vos informations.

  -  Installez les dépendances : composer install

  -  Mettre en place la BDD :

      php bin/console doctrine:database:create

       php bin/console doctrine:migrations:migrate

2. Créez une branche à partir de *\*master\** : git checkout -b nom de la branche

3. Ecrivez un Issue sur les modifications que vous allez apporter

4. Ecrivez votre code EN RESPECTANT LES BONNES PRATIQUES

5. Ecrivez des Commit Clairs et précis avant de faire un Push de la branche : git push origin maBranche

6. Mettez a jour vos issues

7. Faites un *\*Pull Request\** et attendez sa validation

**# Les bonnes pratiques**

**#  1. le code**

    Vous devez respecter :

    - Le PSR 2 au minimum

    - Les standards du code de Symfony (`https://symfony.com/doc/current/contributing/code/standards.html`)

    - Les conventions du code de Symfony (`https://symfony.com/doc/5.2/contributing/code/conventions.html`)

**# 2. les bundles**

    - Toute installation de bundle PHP doit se faire avec "Composer OBLIGATOIREMENT"

**# 3. Git**

    Vous devez faire les choses dans cet ordre :

    - Nouvelle branche à partir de master dument nommée

    - Commit Correctement commentés

    - Issue Correctement commentées et documentées

    - pull Request OBLIGATOIRE

    - Seul le chef de projet peu faire un "merge" sur "master" après révision de votre code.

    - Faire un update sur le code principal : git pull origin master

**# 4) Tests unitaires et fonctionnels**

    - Toute nouvelle fonctionnalité doit avoir des tests associés

    - Vous devez respecter un taux de couverture au-delà de 70%

    - PhpUnit est à votre disposition pour créer vos tests

       \* Ecrivez vos tests dans le dossier /test

       \* Utiliser MakerBundle's make:test pour créer une squelette de test!

       \* Lancer les tests avec la commande vendor/bin/phpunit --coverage-html web/test-coverage pour générer un rapport de couverture

**# 5) Architecture de fichier**

    - Respectez l'architecture de Symfony 5 pour vos fichiers PHP ( src\Controller\... )

    - Les vues devront être dans le répertoire Templates.

**# 6) Suggestion d’amélioration**

    - Ajouter un espace membre / profil utilisateur, avec statistique des taches de l’utilisateur

    - Permettre l’ajout de nouvelle catégorie (ex : Taches mises en attente, taches abandonnés, …etc)

- Ajout Date de fin prévisionnelle, date de fin effective, archivage ancienne taches

- Passage à Symfony 6

- Optimisation des Templates (Trop de lignes de code)

-

**Fonctionnement Authentification**

**# Authentification Guide**

La sécurité concernant l'authentification est configuré dans le fichier `config/packages/security.yaml`

Vous trouverez plus d'informations concernant ce fichier et ses différentes parties dans la [documentation officielle de Symfony](https://symfony.com/doc/5.4/security.html).

**## L'entité User**

Avant toute de chose, il est nécessaire d'avoir défini une entité qui representera l'utilisateur connecté.

On crée une entité en passant par le terminal et la console Symfony :

`php bin/console make:entity`

Cette classe doit implémenter l'interface `UserInterface` et donc implémenter les différentes méthodes définis dans celle-ci.

Dans ce cas-ci, cette classe a déjà été implementée et se situe dans la fichier `src/Entity/User.php`.

**## Security - password\_hashers:**

 Lorsque que quelque chose doit être encoder dans l'entité `App\Entity\User` via `UserPasswordEncoderInterface`, dans ce cas-ci cela concerne le mot de passe.

```yaml

# config/packages/security.yaml

security:

     password\_hashers:

        # auto hasher with default options for the User class (and children)

        App\Entity\User: 'auto'

        # auto hasher with custom options for all PasswordAuthenticatedUserInterface instances

        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:

            algorithm: 'auto'

            cost:      15

```

**## L'entité User**

Avant toute de chose, il est nécessaire d'avoir défini une entité qui representera l'utilisateur connecté.

On crée une entité en passant par le terminal et la console Symfony :

`php bin/console make:entity`

Cette classe doit implémenter l'interface `UserInterface` et donc implémenter les différentes méthodes définis dans celle-ci.

Dans ce cas-ci, cette classe a déjà été implementée et se situe dans la fichier `src/Entity/User.php`.

**## Les Providers**

Un provider va nous permettre d'indiquer où se situe les informations que l'on souhaite utiliser pour authentifier l'utilisateur, dans ce cas-ci, on indique qu'on recupérera les utilisateurs via Doctrine grâce à l'entité User dont la propriété username sera utilisé pour s'authentifier sur le site.

Attention, on peut indiquer ici la classe User car celle-ci implémente l'interface `UserInterface` !

```yaml

# config/packages/security.yaml

    providers:

        users:

            entity:

                class: App\Entity\User

                property: email

```

**## Les Firewalls**

Un firewall va définir comment nos utilisateurs vont être authentifiés sur certaines parties du site.

Le firewall `dev` ne concerne que le développement ainsi que le profiler et ne devra à priori pas être modifié.

Le firewall `main` englobe l'entièreté du site à partir de la racine défini via `pattern: ^/`, l'accès y est autorisé en anonyme c-à-d sans être authentifié, on y indique que c'est le provider "doctrine" qui sera utilisé.

Afin de s'authentifier, on définit un formulaire de connexion via `form\_login:` où sont indiqués le nom des routes correspondant à ce formulaire, la route de vérification du login ainsi que la route vers laquelle l'utilisateur devra être redirigé par défaut après son authentification.

```yaml

# config/packages/security.yaml

 firewalls:

        dev:

            pattern: ^/(\_(profiler|wdt)|css|images|js)/

            security: false

        main:

            lazy: true

            provider: users

            form\_login:

                login\_path: login

                check\_path: login

             logout:

                path: logout

             remember\_me:

                secret:   '%kernel.secret%'

                lifetime: 86400

                path:

```

**## Les Access\_Control**

Un access\_control va définir les limitations d'accès à certaines parties du site.

Dans ce cas-ci, on indique que :

- L'url /login est accessible sans authentification.

- L'url /admin n'est accessible qu'en étant authentifié avec un utilisateur ayant le rôle "ROLE\_ADMIN".

- Tout le reste du site n'est accessible qu'aux utilisateurs authentifiés c-à-d ayant le rôle "ROLE\_USER".

```yaml

# config/packages/security.yaml

    access\_control:

        - { path: ^/admin, roles: ROLE\_ADMIN }

        - { path: ^/login$,    roles: IS\_AUTHENTICATED\_ANONYMOUSLY }

        - { path: /, roles: PUBLIC\_ACCESS }

```

**## Les Role\_Hierarchy**

Un role\_hierarchy permet de s'assurer qu'un utilisateur ayant un certain rôle aura automatiquement d'autres rôles.

Dans ce cas-ci, un utiliseur possédant le rôle "ROLE\_ADMIN" aura automatiquement le rôle "ROLE\_USER".

```yaml

# config/packages/security.yaml

role\_hierarchy:

        ROLE\_ADMIN: ROLE\_USER

```