AUDIT DE PERFORMANCE ET DE QUALITE

I. CONTEXTE

Ce projet a été créé par la société **ToDo & Co** afin de développer une application de liste à faire. Le principal objectif de cette application est de pouvoir laisser l'utilisateur créé plusieurs taches qu'il fera évoluer sur l'interface en fonction de son déroulement. Il pourra donc laisser la tâche « à faire » ou la cocher pour signaler à l'application qu'elle est déjà faite et qu'il n'a plus à s'en soucier.

Le projet initial, écrit en Symfony 3, a été forké du projet principal sur GitHub. Afin d'améliorer le projet, et vu la taille de celui-ci, il a été décidé de migrer l'intégralité du code vers la dernière version de Symfony : Symfony 5. Certains bundles ont changé entre les deux versions, mais globalement peu de bugs ont été engendrés par cette montée de version.

1. Amélioration de fonctionnalités

Une fois que le projet a été réécrit sur la version 5 de Symfony, certaines fonctionnalités ont été améliorées. En effet, plusieurs anomalies persistaient sur le projet.

Par exemple, la première anomalie constatée sur le projet est celle du manque d'appropriation d'une tache par un utilisateur. Une tâche n'avait donc ni de propriétaire à sa création, ni à son édition, ni à sa suppression.

La seconde anomalie concerne la gestion des rôles utilisateurs. En effet, aucun rôle n'était attribué aux utilisateurs. Par conséquent, aucune fonction d'administration ne pouvait être développée pour un type d'utilisateur. Cette fonctionnalité a été insérée à partir du SecurityBundle de Symfony comme c'est mentionné dans la documentation technique du projet. Les rôles sont attribués à la création du compte. La possibilité de changer son rôle est laissé à l'utilisateur lorsqu'il édite son compte.

2. Création de fonctionnalités

En plus de cette amélioration de version et de fonctionnalités, de nouvelles fonctions ont été ajoutées au projet. En effet, jusqu'alors il n'était pas possible pour les utilisateurs qui n'étaient pas inscrits comme administrateur, d'avoir des fonctions d'éditions ou de suppression sur d'autres utilisateurs. Les pages d'administration ont donc été réservées aux utilisateurs loggés ayant le rôle « ROLE_ADMIN ».

De plus, la fonctionnalité de suppression n'était, quant à elle pas encore développée. Cette nouvelle fonctionnalité est donc apparue dans les fonctions d'administration.

De la même manière, les taches pouvaient jusqu'à présent être supprimées par n'importe quel utilisateur puisqu'elles n'étaient pas rattachées à un propriétaire. Pour que le fonctionnement

soit plus logique, il n'est maintenant possible de supprimer une tache uniquement si l'utilisateur loggé est le même que le propriétaire de celle-ci.

II. AUDIT DE PERFORMANCE

Différents outils ont été utilisés pour attester de la performance de l'application. Le plus important est **Blackfire**. Cet outil permettant de profiler des pages web nous permet d'avoir des métriques complètes sur la performance de l'application. Il a été utilisé à l'aide de **CURL** pour pouvoir faire les requêtes http et par **phpUnit** pour l'utiliser sur des tests fonctionnels.

Les deux principales métriques utilisées sont :

- La mémoire utilisée par la méthode
- Le temps d'exécution de la méthode

Un test de performance a par exemple été effectué sur la page principale de l'application. Le but de ce test de performance était d'améliorer le temps de chargement des classes. Ce test et cette amélioration de performance ont été effectués sur la classe :

Symfony\Component\ErrorHandler\DebugClassLoader\loadClass

Cette classe permet le chargement automatique des classes de Symfony. Il permet de cacher aussi leurs lieux de stockage afin d'améliorer les performances de l'application. Le test a été effectué sur la page principale de l'application 'homepage'. Cette page avait un temps d'exécution de **781 ms** et **18.2 MB** de mémoire (**Homepage = 781 ms + 18.2 MB**).

1. Avant :

Avant l'optimisation, cette classe prenait 315 ms d'exécution et 5.48 MB de mémoire, lors du profilage de la page principale :

- **WallTime 315 ms**
- **Memory 5.48MB**

2. Amélioration de performance

Pour optimiser le chargement des classes, il a été choisi d'opter pour une optimisation du chargement des classes grâce à l'autoload de composer. La commande suivante est exécutée à la racine du projet :

```
composer dump-autoload -o
```

3. Après

Le temps d'exécution de la page d'accueil a été réduit de plus de **71%** en passant de **781 ms** à **228 ms**. La mémoire, quant à elle, est restée stable. L'optimisation du chargement des classes

n'a eu aucun impact sur la mémoire occupée (**18.2 MB**). Ce n'était effectivement pas le but recherché ici. Nous recherchions avant tout un gain de temps d'exécution.

En ce qui concerne la classe Symfony\Component\ErrorHandler\DebugClassLoader\loadClass, nous observons une forte amélioration de ces métriques. En effet, le gain de performance sur le temps d'exécution de la page d'accueil est très clairement dû à l'optimisation de la loadClass de Symfony. Là encore, la mémoire occupée par la classe est restée stable. En revanche on voit une forte diminution du temps d'exécution passant alors de **315 ms** à **88.5 ms**. On observe une optimisation d'environ **71%** du temps d'exécution :

- **WallTime 88.5 ms**
- **Memory 5.48MB**

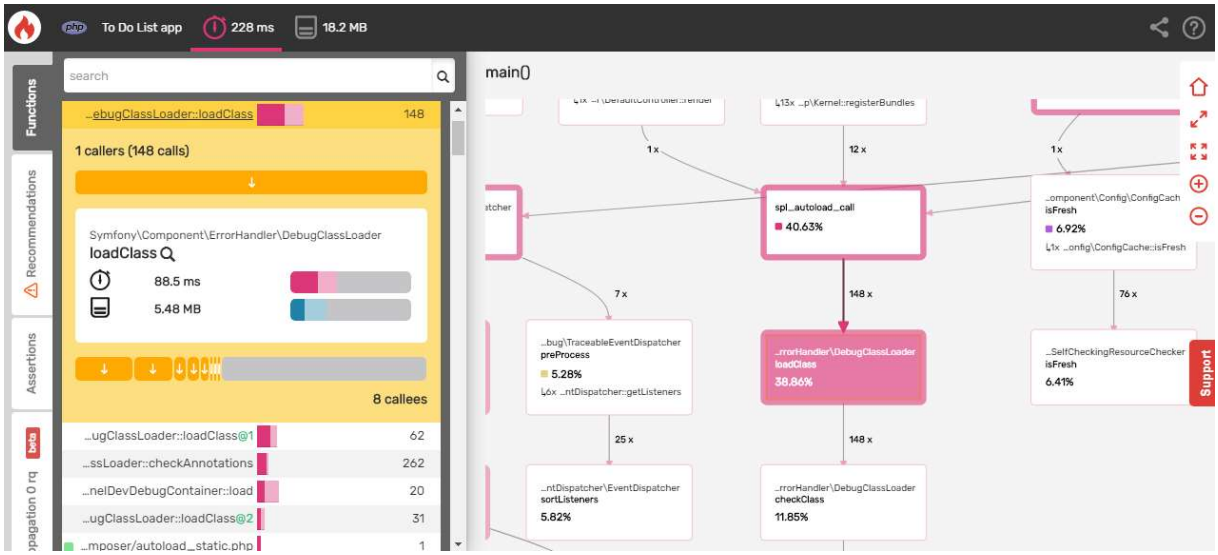

Figure 1 : Résultat de l'analyse du profiler de la homepage après optimisation

III. RÉALISATION DE TESTS FONCTIONNELS ET UNITAIRES

Afin de réaliser les tests unitaires et fonctionnels, PhpUnit a été installé sur le projet Symfony. Le bundle a été téléchargé sur le projet à partir de composer via la commande suivante :

```
composer require –dev symfony/phpunit-bridge
```

Après avoir téléchargé PhpUnit et avoir développé les tests unitaires et fonctionnels. Il nous a fallu les tester et vérifier la couverture des tests sur l'application. Cependant, avant de lancer le test coverage du projet, différentes étapes ont été nécessaires.

Effectivement, dans un premier temps, il a été nécessaire d'insérer un jeu de données test au sein de l'application Symfony. Différentes DataFixtures ont été créées en fonction des Entités créées au préalable. Les commandes suivantes ont permis de créer la base de données nécessaire au projet, mais aussi de l'alimenter en données tests :

```
php bin/console doctrine:database:create : To create database which is configured on .env file

php bin/console doctrine:schema:update --dump-sql : To show SQL statement will be executed

php bin/console doctrine:schema:update --force : To execture SQL statement and create table on database

php bin/console doctrine:fixtures:load : To load data fixture on tables
```

Enfin, après avoir terminé l'insertion du jeu de données dans la base, un rapport de couverture est effectué. Un outil supplémentaire est cependant nécessaire à la création du rapport, si celui-ci n'est pas déjà installé sur la machine en local : XDebug. Une fois que cet outil est installé grâce à un module PHP, le rapport de couverture peut être exécuté à partir de la commande ci-dessous :

```
php bin/phpunit - - coverage-html tests/coverage
```

La commande ci-dessus rend alors un rapport de couverture des tests présents au chemin App\tests\coverage. La couverture est en moyenne supérieure à 75%. Nous n'avons pas jugé nécessaire de tester les DataFixtures permettant entre autres de faire les tests.

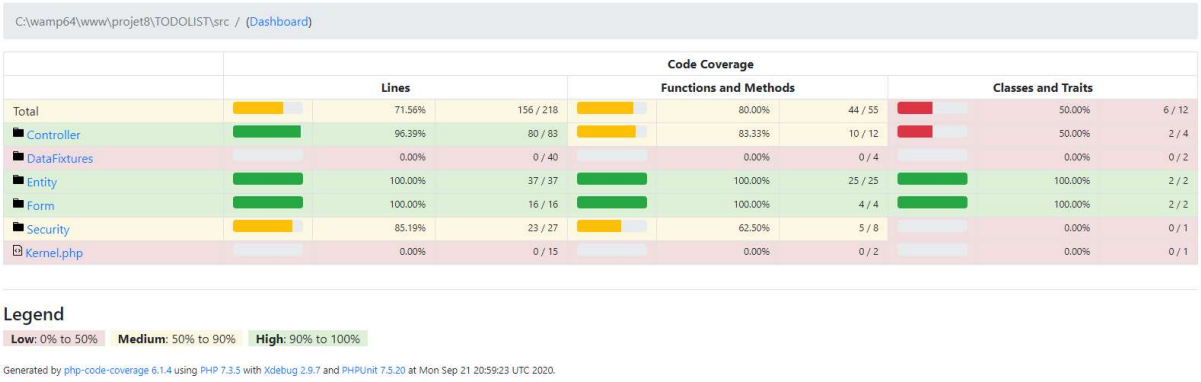

Figure 2 : résultat du code coverage généré par PhpUnit

Les outils qui ont donc été utilisés pour les tests sont donc les suivants :

- PhpUnit : pour l'exécution des tests
- XDebug : pour avoir le pattern du coverage
- DoctrineFixturesBundle : pour la création du jeu de données test

IV. QUALITÉ DE CODE

La qualité du code passe par plusieurs étapes de validation. En effet, comme évoqué sur la documentation technique. Le code a été soumis à plusieurs vérifications lors du processus de développement.

La première étape de validation d'un projet se déroule lorsqu'un développeur exécute une pull request sur le GitHub du projet. En effet, le développement de l'application est géré par GitHub. Cet outil nous permet de versionner le projet et de collaborer facilement sur le développement de celui-ci. Les versions stables se situent sur la branche master. Les développeurs, quant à eux travaillent sur la branche de dev ainsi que sur des branches annexes qui servent au développement de nouvelles fonctionnalités. Chaque branche créée pour le développement d'une nouvelle fonctionnalité est ensuite fusionnée à la branche de dev qui sert de validation et de correction de bugs si des anomalies se sont créées entre les différentes branches. Enfin, une fois que la branche de dev a accumulé assez d'améliorations par rapport à la version de la master précédente, une pull request est effectuée de la dev à la master afin de fusionner les deux branches.

L'étape principale de validation du code s'effectue à la création de pull request. Des développeurs et validateurs annexes se voient attribuer la pull request afin que ceux-ci puissent attester de la qualité du code. Ils vérifient alors si certaines anomalies sont présentes sur cette pull request et peuvent ensuite valider la fusion des deux branches. Cette étape permet donc d'externaliser la vérification du code à des développeurs qui n'ont pas forcement travaillé sur cette fonctionnalité et qui auront un œil différent du développeur qui a été en charge de la programmation.

En plus de cette étape très importante concernant la qualité du code. Le projet GitHub est soumis à validation par outil externe : Codacy. Chaque version présente sur la dev est soumise à validation sur Codacy. Là encore, l'application permettra d'attester de la qualité du code et des correctifs pourront être appliqués rapidement à la branche de dev pour pouvoir ensuite la merger à la master. Le projet doit obligatoirement obtenir une appréciation de B pour pouvoir être considéré comme stable par notre équipe. En plus de cette appréciation, nous considérons que 100% des issues concernant la sécurité de l'application doivent être traités, sans quoi la branche de développement ne sera pas fusionnée à la branche principale.

V. AMÉLIORATION A PREVOIR

Pour terminer, des améliorations peuvent toujours être à prévoir du point de vue technique du projet. En effet, la couverture des tests sur l'application n'atteint pas encore 100% de couverture. Il serait donc nécessaire d'approfondir les tests afin qu'ils puissent couvrir l'intégralité des méthodes, des classes et des traits de l'application.

En plus de cette amélioration, il serait surement utile d'améliorer la stylisation du code. En effet, plusieurs issues ont été ignorées sur Codacy (comme par exemple dans les assets du projet où parfois des simples quotes étaient présentes à la place de doubles quotes). Ce type d'issues a été ignoré sur Codacy. Cependant, il pourrait être utile de les résoudre.

Enfin, d'autres tests de performance pourront surement être faits à l'avenir lorsque l'application se développera davantage et deviendra plus gourmande en performance.