Introduction

Cette page est destinée à expliquer les choix techniques, la méthodologie employée pour comprendre, déployer et tester ce projet.

Pour ce projet, nous avons décidé d'utiliser GitHub comme gestionnaire de version. Nous l'avons choisi par rapport à Gitlab car c'est un outil que nous utilisons tous plus fréquemment et qui offre des fonctionnalités similaire à ce dernier. En plus de cela, nous avons pu utiliser le système de GitHub Pages qui nous permet de déployer une branche, nous permettant ainsi de déployer nos coverages et rapports de test.

Code existant

Nous avons voulu, dans un premier temps, comprendre comment fonctionnait l'application. Comme il n'y a pas de documentation ni de README, nous avons observé le schéma de la base de données. Celui-ci à pu nous apporter beaucoup d'informations sur :

- Le but de l'application
- Les entités et leurs relations
- Le fonctionnement primaire

Nous nous sommes en suite penchés sur le code. C'est un code simple, daté et donc non maintenu. Il n'y avait aucune documentation technique sur comment lancer le projet. En plus de ça, nous nous sommes rendus compte de certains problèmes sur l'api.

Parmi les problèmes :

- Mettre à jour un sondage.
- Des méthodes GET qui retournent des listes vides au lieu d'indiquer que la ressource demandée n'existe pas.
- Certaines méthodes retournent des erreurs 500.

API

Déploiement

Dès le début du projet, nous avons mis un point d'honneur à faire en sorte que notre projet soit facilement déployable sur beaucoup d'environnements. Nous avons alors pensé à 2 stratégies :

1. Utilisation des outils de Springboot
2. Utilisation de Docker Ces deux stratégies sont très différentes et sont complémentaires. L'une permet d'installer le projet de façon classique avec un petit peu de mise en place alors que l'autre solution est une solution "clés en main", sous condition d'avoir Docker installé sur sa machine.

Local

Pour l'installation de ce projet en local, Springboot fournit une documentation présente ici.

Celle-ci indique l'utilisation de fichier nommé "application.properties" qui permettent de donner des variables à notre projet afin qu'il puisse s'exécuter proprement. (Ex: Lien de la base de données, Driver de la base, etc...)

Mais également, elle permet d'indiquer le profil de l'application, il s'agit d'une variable qui va indiquer si le projet tourne dans un environnement local, de test ou encore en production. Cela permet à l'application de rechercher d'autres variables d'environnement spécifiques à chaque profil (Ex: profil=local, alors le fichier application-local.properties sera utilisé). Le but étant d'affiner ces variables en fonction de chaque environnement d'exécution.

Le déploiement en local nécessite de la mise en place afin de pouvoir correctement s'exécuter. En effet, la machine doit posséder :

1. JAVA Version 19+
2. Une base de données
3. Un serveur HTTP (Type : nginx ou apache) dans le cas du déploiement sur un serveur à distance.

Docker

Comme expliqué précédemment, nous avons préparé une configuration Docker permettant à l'utilisateur une solution "clés en main". Celui-ci n'a besoin d'avoir sur sa machine que le programme Docker. Cela nous permet de lancer le projet d'une seule ligne de commande

```
docker-compose up -d --build
```

Celle-ci va utiliser les fichiers docker-compose.yml et Dockerfile afin de créer un environnement virtuel (un container) configurant automatiquement :

1. JAVA 19
2. Une base de données Et permettant aux connexions externes de se connecter au projet par un port défini dans la configuration .env du projet.

Pour notre projet, nous avons décidé d'utiliser une base de données Postgres dans la configuration Docker. Celle-ci est accessible en dehors du conteneur de base, mais il est facilement modifiable afin d'éviter cela.

Lors de la mise en place de ce dernier, nous avons rencontré quelques problèmes sur l'utilisation des variables partagées. Comme dans la configuration précédente, il est possible d'intégrer des variables d'environnement propre au projet (Ex: Changer le profil de l'application ou encore l'url de la base de données).

Le fonctionnement sera le même en local ou sur un serveur (Il faut tout de même un serveur HTTP sur le serveur). Une feature est également ajoutée en local (Sur machine perso) : Adminer, une interface de gestion de base de données très légère.

Environnement public

Pour déployer sur notre environnement public, nous possédions déjà un serveur (VPS) loué chez OVH ainsi qu'un nom de domaine. Le serveur possède un serveur HTTP nginx et est sécurisé à l'aide plusieurs systèmes.

Sur le serveur, le projet tourne sur l'environnement Docker et tous les ports, exceptés HTTP et SSH, sont fermés.

Nous avons alors créé une GitHub Action de CD permettant de faire un déploiement automatique sur le serveur grâce à une authentification à l'aide de clef ssh.

Cette Action se connecte donc au serveur, éteint temporairement le projet, effectue un git pull et redémarre le conteneur avec les changements.

Github-pages

Comme nous utilisons Github, nous avons accès à GitHub Pages en suivant ce tutoriel. Ce dernier va nous servir afin de publier les rapports de Coverage de Jacoco et PiTest. Comme vous pouvez le voir ici : Coverage Jacoco (Vous pouvez cliquer) : 

Pitest Mutation Coverage

Tests

Tests unitaires

C'était le type de tests dont nous avions le moins de mal à imaginer et à implémenter. Ce sont les premiers qui ont commencé à émerger dans le projet relativement simplement. C'est également le seul type de tests que nous avions déjà effectué sur des projets. Ces tests nous ont guidé tout le long du projet, notamment à l'amélioration de la qualité du code grâce aux mutations de Pitest et au Coverage de Jacoco.

Nous effectuons nos tests unitaires sur les services et les controllers. Nous ne testons ni la configuration, car ses erreurs se verraient au niveau du build, ni le main pour les mêmes raisons. Quant aux modèles et DTO, nous ne les testons pas car les méthodes sont appelées et donc testées directement depuis les services. Faire cela permet également lors du coverage de repérer les bouts de codes qui ne sont jamais atteints et donc, jamais utilisés.

Tests d'implémentation

Nous avons décidé, après concertation, de ne pas effectuer de tests d'implémentation. Dans un premier lieu, nous avons essayé d'en faire en mockant la base de données grâce à MockMVC. La configuration était compliquée et nous nous sommes finalement rendus compte que ces tests allaient se confondre avec les tests E2E, qui effectuent la même chose en plus de tester les ajouts à la base de données et les codes de retour HTTP. Nous avons donc décidé de ne pas effectuer de tests d'implémentation pour ce projet.

Tests E2E

Nous avons effectué quelques recherches pour savoir comment exécuter des tests E2E car nous n'en avions jamais fait, il nous a été compliqué dans un premier temps de comprendre comment les effectuer alors que nous n'avions pas d'interface graphique. Jusque là, les tests E2E ne représentaient que des tests de clics sur une interface graphique. Par la suite, nous avons compris qu'il fallait créer des scénarios qui permettaient d'exécuter le code qu'effectueraient ces tests de clics.

Ces tests ont été très longs à implémenter et ce, assez chaotiquement. Notre méthode de travail aurait pu être plus optimale à ce sujet. Nous avions un mode de fonctionnement hybride avec du Test-Driven-Development. Nous avons effectué des modifications dans le code à mesure que les tests échouaient et nous avons repéré que beaucoup de codes HTTP renvoyés étaient mauvais ou des fonctionnements qui étaient peu intuitifs ou ne fonctionnaient pas du tout. Également, le fait que nous ayons choisi de faire des scénarios, et donc des tests qui découlent les uns des autres fut horrible à corriger car une erreur en produisait 4 supplémentaires.

Au final, ces tests ont demandé un temps important de développement, peut-être un peu trop ? Ils ne testent pas non plus tous les cas mais ils nous ont permis de grandement améliorer la qualité de code, tester les fonctionnalités et s'assurer qu'elles fonctionnent dans leur comportement "normal".

Code Coverage et qualité de tests

Le code coverage et les mutations de tests est un excellent outil pour améliorer la qualité du code ainsi que pour tester les cas non prévus par notre application. En revanche, ce n'est pas un outil fiable à 100% dans la mesure où les outils sont capables de générer des faux positifs. De plus le fait que nous n'ayons pas 100% de coverage peut également nous aider car cela nous indique que du code est inutilisé. C'est un pourcentage qui s'analyse en profondeur et qu'il ne faut pas essayer de pousser à 100% par tous les moyens. Il est simple d'améliorer son score et passer à 100% de coverage si l'on effectue des evergreen tests.

Vous pouvez consulter cette partie ici