GitClout

Manuel développement

Projet de Java

OEUVRARD Dilien

Partie A : Présentation des technos

Nous avons eu pour mission de faire une application back-end en utilisant des technologies qu'on n'a jamais vu avant.

Le but de cette application est de faire de l'analyse de repositories notamment grâce à jGit.

Nous avons dû utilisé comme back-end **SPRING reactive,** comme persistence **JPA**, come base de données **Sqlite**, comme front-end **riotjs** et enfin comme ui **bulma**

I – Back-end

Nous avons décomposé notre back-end en plusieurs parties :

- Repositories
- Analyze
- Tags

Dans ces trois packages, nous gérons différentes choses.

A – Repositories

Nous retrouvons le même parttern que pour le package Tags.

Dans Repositories, il y a une classe Controller, c'est l'api de notre application. Mais comme nous sommes dans le package repositories ce controller est seulement là pour la gestion des repositories. Comme la création d'un repositories, la récupération des repositories déjà analysés, le rafraîchissement ou encore la suppression d'un repositories.

La classe repositories fait appel à la classe RepositoryService dans laquelle on retrouve de multiples méthodes qui servent à la création du projet.

Nous allons aussi retrouver des records qui serviront à l'envoie vers le front (json)…

B – Tags

Le package tags est construit exactement comme le package repositories.

C – Analyze

Par contre ce package est différent, car c'est grâce à lui qu'on va pouvoir faire l'analyse complète d'un repositories, notamment en utilisant jGit.

On s'occupe aussi de faire le système de rafraîchissement.

Quant aux différents langages gérés par notre application, nous avons choisi d'utiliser un enum. L'énum Language regroupe donc les différentes informations sur tous les langages que notre application peut gérer.

Pour certains un regex est mis afin de faire la gestion des commentaires.

Cette approche nous a permis d'avoir aucun mal à gérer toute la partie d'analyse, surtout quand on a dû commencer à analyser les commentaires. (nous avons pensé à une amélioration qui pourrait être de gérer les langages de façon dynamique grâce à un input dans le front par exemple.

Pour l'analyse des fichiers nous avons utilisé des Threads afin d'augmenter la rapidité du temps de calcul, avec des callables (qu'on a pu voir en cours de concurrence).

Nous avions du mal à comprendre une partie du sujet par rapport à l'analyse.

Est-ce qu'il fallait analyser pour chacun des tags le repositories au moment T du tag en entier depuis le début ou seulement les nouvelles contributions.

Dans le doute nous avons tout analysé depuis le début sur chacun des tags même si cela voulait dire que le temps de calcul deviendrait plus long, car nous devions réanalyser les fichiers à chaque tag.

Nous avions pensé à la mise en place d'un cache pour éviter de recalculer les fichiers déjà calculés. Néanmoins nous avions vu que le temps de calcul était le même avec ou sans cette modification pour des repositories de taille moyenne.

D – La base de données

Pour gérer notre base de données on a mis en place des classes NameStorage qu'on peut retrouver dans le sous-package jpa d'autres packages.

Dans ces classes nous nous sommes permis de mettre en protected le contructeur vide, nous savons que cela a été interdit dans le sujet, mais il s'agit d'une exigence JPA. Car JPA a besoin d'un constructeur par défault pour créer des instances de l'entité quand nous récupérons des objets depuis la base de données. Protected sert de protection.

Néamoins nous avons choisi d'utiliser CrudRepository au lieu de ReactiveCrudRepository car ReactiveCrudRepository n'est pas nativement compatible avec JPA.

E – les APIs

La gestion API était plutôt simple et ne nous a apporté aucun problème.

Pour rendre notre programme réactif, il a seulement fallu que nous retournions des Mono et des Flux.

Nous avons donc retourné des flux de record que nous avons créé. Ce sont des records vident qui agissent comme des json.

F – SSE

Pour gérer la barre de progression nous avons choisi de se pencher vers le SSE et non un **web-socket**. Car le **SSE** a un avantage que le web-socket n'a pas, si le SSE se coupe, il tentera de rouvrir automatiquement la connexion.

Si nous avions dû faire une communication **bidirectionnelle** nous aurions pris un web-socket et dans notre cas ce n'est pas utile.

G – Tests

Nous ne pensions pas, mais les tests nous ont énormément aidé, notamment pour le débogage. Il existe des tests pour la plupart des méthodes, même les méthodes privées…

L'utilisation de la bibliothèque **Mockito**, nous a servi pour simuler des dépendances qui sont utiles pour certaines méthodes.

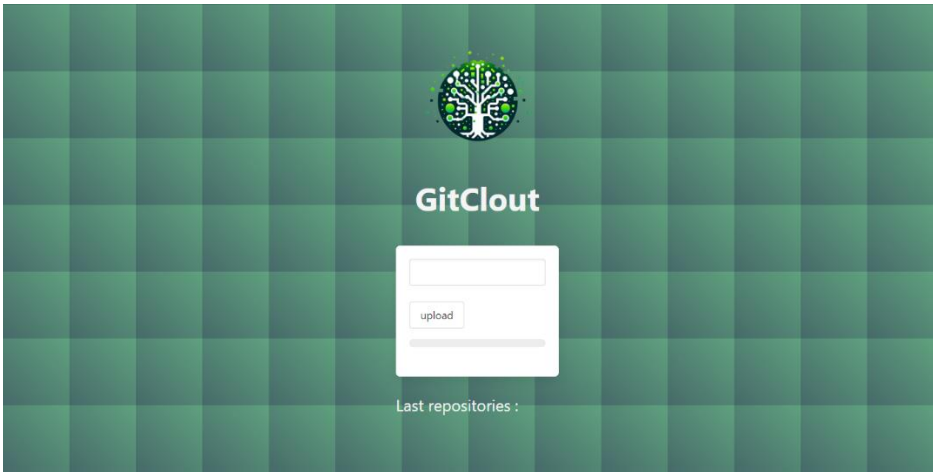
Néanmoins cette bibliothèque nous renvoie un warning lors du package du projet.

Après une investigation nous avons vu que ce warning n'impactait pas du tout notre projet, nous avons donc choisi de le laisser.

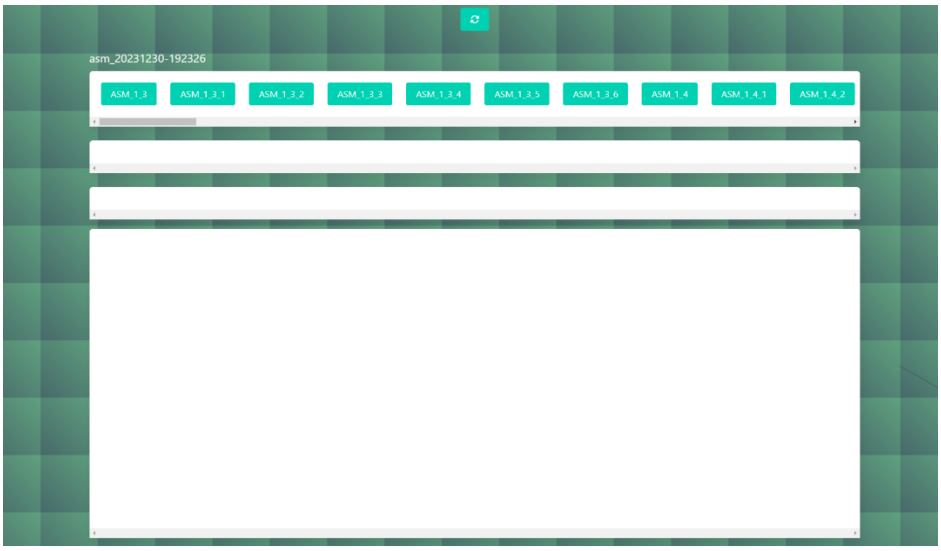
II – Front-end

Le front-end comprend deux pages dynamiques (grâce à un routeur) :

- La première, sur laquelle l'utilisateur va pouvoir rentrer le lien de son repositories.



- La seconde sur laquelle l'utilisateur peut observer l'analyse de son repositories.



Notre programme **riot** se décompose en plusieurs components se trouvant dans components/global.

Ce fichier comprend une partie analyse, index et app.

Analyse pour l'analyse des tags.

Index pour la page d'accueil.

App pour l'initialisation du routeur.

A – chartjs

Afin de pouvoir créer nos différents graphiques nous avons utilisé la bibliothèque chartjs.

Cette bibliothèque nous a permis de gagner énormément de temps sur la partie front-end car nous n'avions pas eu besoin de recréer tous les graphs à la main.

III – Idées d'améliorations

Nous avons pensé à plusieurs idées d'amélioration possibles pour rendre notre projet plus utilisable.

- Utilisation d'un cache pour les gros répositories. Nous avions implémenté un cache mais sans succès pour les fichiers moyens. Nous nous y sommes surement mal pris pour gagner du temps.
- Faire un système qui affiche les tags déjà analysés pour éviter de devoir attendre que tous les tags soient analysés pour regarder un seul tag.
- Ajouter ou retirer des langages de façon dynamique via le front. L'enum c'est très bien, mais la gestion dynamique c'est mieux.

IV – Difficultés ?

La plus grosse difficulté a été de savoir comment fonctionne tel techno, et encore on a plus vu cela comme un jeu. De plus ce challenge nous a permis d'apprendre beaucoup.

V – Conclusion

Ce projet a été une vraie course d'orientation.

La course car nous avons passé énormément de temps dessus. L'orientation car il a fallu qu'on s'y retrouve à travers les multiples documentations.

Mise à part, nous avons énormément grâce à ce projet. Mais nous aurions préféré pouvoir avoir des technos plus utilisées notamment pour le front-end.