Université Gustave Eiffel

CloneWar

Rapport du projet de Java avancé
Master 1 Informatique
2022-2023

- Ramaroson Rakotomihamina Johan
- Tellier Quentin

I - Présentation

Ce rapport à pour but de présenter l'organisation générale du projet, nos choix d'implémentation, les problèmes rencontrés, résolus ou non.

II - Organisation générale

CloneWar est une application web possédant une partie back-end pour interagir avec la base de données et détecter les clones. Elle sera écrite en Java et utilisera comme framework **Spring webFlux**, la version asynchrone de Spring.

Elle possède en complément, une partie front-end via le framework **React** pour l'interface utilisateur, en complément nous utiliserons le framework graphique **Bulma CSS**.
La base de données utilisée pour la persistance est **SQLite**.
Pour le mapping entre les classes et les tables de la base de données, nous avons dû utiliser JPA avec **Jakarta**.

III - Back-end

Cette section présentera la structure et la mise en place de notre application back-end. Ainsi que les détails sur la stratégie adoptée pour répondre à la demande.

III. 1 - Organisation

Notre partie back-end s'organise en 3 couches distincts :

- La couche de persistance, la couche contenant les classes permettant d'interagir directement avec la base de donnée, ou de représenter les tables et colonnes de celle-ci.
- La couche de service, qui contiendra toutes la logique, les librairies que nous avons créées afin de répondre au besoin de l'application, cette couche est la couche centrale qui communiquera avec toutes les autres.
- La couche Rest, qui contiendra les classes nécessaire pour communiquer avec l'extérieur de l'application.

Ce choix d'organisation permet d'avoir une bonne vision sur les 3 couches à travailler de l'api Rest, de faciliter l'organisation du travail séparé afin d'éviter les conflits.

III. 2 - Base de donnée

Notre base de données est composée de 3 tables, dont 2 tables d'entités et une table relationnelle :

- Artifact(id: UUID, name: varchar, url: varchar, version: varchar, srcJar: byte[], mainJar: byte[])
- Instruction(instructionId: UUID, content: varchar, filename: varchar, hash_value: long, start_line: int)
- artifact_instructions(artifact_id: UUID, instructions_id: UUID)

Note :
artifact_id dans artifact_instructions est une foreign_key faisant référence à id dans Artifact.
instructions_id dans artifact_instructions est une foreign_key faisant référence à id dans instruction.

III. 3 - Stratégie de détection de clone

Pour pouvoir détecter les clones, lors de l'indexation d'un artefact nous parcourons les instructions en bytecode, puis nous appliquons l'algorithme de karp-rabin dessus. L'application de la détection de clone se présente sous plusieurs étapes :

1) L'application Back reçoit le jar, elle parcourt entièrement les fichiers .class et via la librairie ASM elle transforme ces instructions en bytecode. On abstrait les instructions de la manière suivante :
    - On ignore tous les noms qui peuvent être donnés par un humain, que ce soit le nom des classes, le nom des variables, le nom des méthodes.
    - Les instructions ne dépendent pas du type ou de la classe qu'ils utilisent, par exemple on abstrait le type des "returns", chaque return aura le même bytecode.
2) On récupère le résultat des instructions abstraites, on applique l'algorithme uniquement pour hasher les instructions en les stockant dans une collection.
3) On persiste les instructions dans la base de données.

Ainsi, il ne nous reste plus qu'à récupérer les instructions correspondant à l'artefact à comparer et à comparer chaque instructions avec ceux en bases selon le hash calculé lors de l'indexation. On détecte un clone si et seulement si deux instructions ont le même hash et que les instructions sont les mêmes. Lors de la comparaison les instructions sont triés selon leur hash, ainsi on s'arrête dès que le hash devient plus grand que celui comparé.
On effectue un calcul pour pouvoir récupérer le taux de hash pareil en fonction du taux du nombre de groupe d'instructions inséré en base.

III.4 - Gestion des tests

Pour un projet de cette envergure, il nous a paru important d'effectuer des tests unitaire et d'intégration afin de pouvoir tester la stabilité de l'application, mais aussi pour contrôler que le refactoring de code ne fasse pas planter les fonctionnalités déjà implémentées.

L'organisation générale décrite au ***III.1*** permet également de découper les tests de la façon suivante :

- Test d'intégration service + persistence

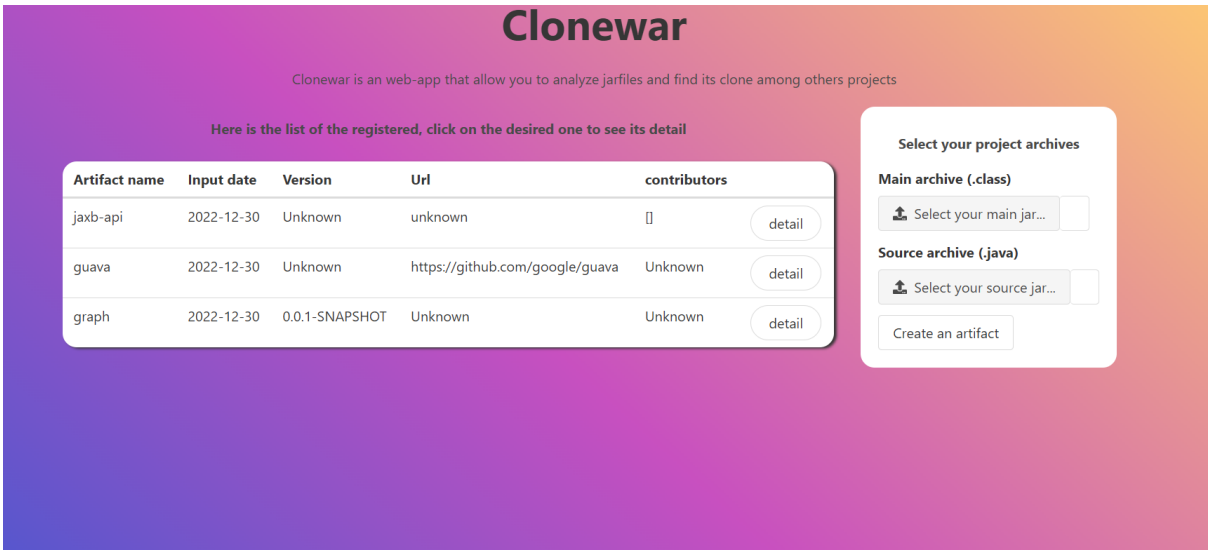
- Test d'intégration service + api rest
- Test unitaires des services

Dans le dossier **ressources** du package de test, il existe un sous dossiers *tests* qui contiennent tous les fichiers jar qui permettent de tester de la stabilité de l'algorithme de Karp-Rabin.
Il existe également un autre dossier "*samples"* qui, lui, répertorie les artefacts avec laquelle on peut tester notre application, elle contient donc pour chaque artefact son jar principale (avec les .class) et son jar sources.

IV - Front-end

IV. 1 - Rendu

Première page :



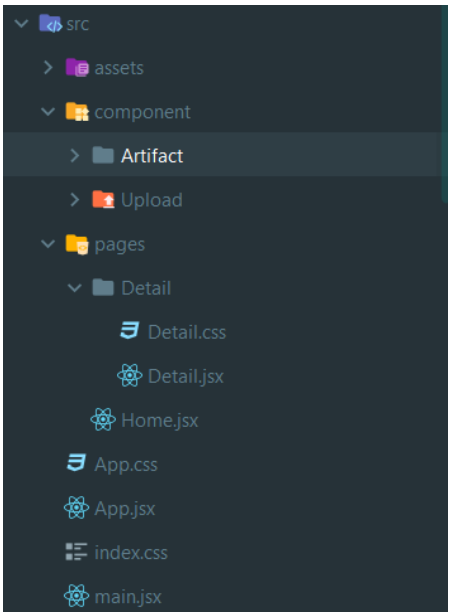
A gauche, voici le tableau affichant les informations nécessaires pour chaque artefact indexé, en appuyant sur le bouton "détail" d'un artefact, vous pouvez accéder à sa page de détail. A droite, voici le formulaire permettant d'indexer un artefact et de l'enregistrer dans la base de données.

Deuxième page :

Cette seconde page s'affiche lorsqu'on appuie sur le détail de Jaxb-api de la première page. Pour chaque autre artefact que l'on a indexé dans l'application, on observe le taux de clone de l'artefact en question dans les autres.

IV.2 - Organisation

Notre partie front-end est assez simple, elle se décompose en 2 pages composé de plusieurs composants graphiques.
Notre hiérarchie s'organise comme ceci :



Le module App est la page d'accueil contenant la liste des artifacts déjà indexer ainsi que la possibilité d'en indexer d'autres.
La deuxième page est la page de détail, elle permet d'avoir les détails des clones d'un artefact selon les autres.

Nous avons deux composants graphiques : Le composant **Artifact** qui permet d'afficher sur un tableau la liste des artefacts et le composant **Upload** permettant de créer le formulaire d'indexation d'un artefact.

V - Nos axes d'améliorations

Voici ci-dessous une liste des améliorations possibles pour le projet :

- Interface utilisateur mieux gérer, plus user-friendly et simplement plus agréable à l'œil. Le choix du design actuel n'est pas celui prévu à la fin. Mais nous avons privilégié l'écriture et la maintenance du back-end. Elle nous paraît cependant suffisante pour une première version de l'application.
- Afficher une page contenant les instructions clonés par les autres artefacts en faisant une comparaison.
- Documenter notre api-rest avec Open-api, la configuration en version réactive de Spring ne nous à pas permis malgré les multiples tentatives de pouvoir la faire fonctionner.
- Permettre à l'application d'être 100% asynchrone avec le front, en implémentant par exemple une barre de progression pour les chargement de données.
- Finir l'écriture des tests d'intégration, toutes les méthodes de notre api rest ne sont pas encore testées.

V - Nos difficultées

Notre difficulté principale lors du développement de ce projet a été la manipulation et la configuration du framework en version asynchrone avec Spring webflux. Le quart du temps consacré à ce projet a été de faire fonctionner et manipuler correctement la librairie reactor, comprendre les différentes méthodes des Flux et Mono.

Une autre difficulté fut l'utilisation de l'algorithme de Karp-Rabin. Au début nous pensions que l'algorithme devait être exécuté uniquement après injection des hash dans la base. Or il fallait lancer l'algorithme lors de l'injection en base. La confusion a été corrigée après la soutenance.

Les performances de l'algorithme ont été une longue épreuve, l'indexation d'un projet telle que Guava prenait un temps astronomique. Il nous a fallu beaucoup de temps pour comprendre toutes les raisons pour lesquelles il y a cette perte de performance. Pour l'indexation, l'opération qui prend le plus de temps a été l'insertion dans la base de données des instructions. Plus précisément, si l'on utilise les méthodes des repository pré faits par Spring via son CrudRepository, alors les requêtes ne sont pas optimisées. Il a fallu écrire des requêtes natives pour pouvoir gagner en performances.

Au niveau des comparaisons de hash également il y avait des ralentissements, comparer deux listes d'instructions était long lorsque la source était un gros artefact, nous avons donc utilisé des threads pour pouvoir répartir le calcul des instructions sur plusieurs cœurs.

Conclusion

Malgré la difficulté et le temps non-négligeable dépensé pour ce projet, il a été très utile en termes d'expérience sur les bonnes pratiques de la programmation avancée en Java.

Ce projet nous a également donné les clefs pour développer une application web, sur les technologies que nous n'avons pas eu l'occasion d'utiliser, et de nous permettre d'améliorer notre autonomie et notre prise de décisions. Nous avons notamment pu apprendre plus sur la conception d'une api rest, la communication entre la partie back et front d'une application web, la conception d'un projet dirigées et maintenus par des tests, une organisation basée sur git.

Le future de notre application dans des conditions plus appropriées peut se poursuivre par la mise en place et la correction des axes d'améliorations cités précédemment. Une amélioration intéressante mais non demandée aurait également été de permettre à notre application de s'adapter à l'outil de build fourni par l'utilisateur, par exemple pouvoir faire en sorte de faire fonctionner le parsing des métadonnées même si le projet utilise gradle et non maven.