Documentation de développement

Projet CloneWar

-

Dylan DE JESUS et Vincent RICHARD

Université
Gustave Eiffel

Sommaire

Introduction :

Le but du projet Clone est d'écrire une application web qui analyse des fichiers jar (Java Archive) pour détecter des codes communs (on parle de clones).

L'application CloneWar est composée d'un back-end écrit en Java offrant différents services REST permettant d'accéder aux informations de l'analyse d'une archive et d'un front-end écrit en JavaScript affichant ces informations et en particulier les codes sources considérés comme des clones.

Organisation de travail, contraintes temporelle et contrainte technologique résultant peu de ressources disponibles pour Helidon Nima 4.0 :

En plus de la contrainte de temps qui nous était imposée, notre emploi du temps universitaires et les multiples rendus hebdomadaires ont fait que nous n'avons pas pu passer autant de temps que nous l'avions prévu initialement.

Cependant, la période des vacances de Noël nous a permis d'avoir bien plus de temps à consacrer à ce projet.

Ainsi, avant cette période de vacances, notre travail était davantage orienté vers la recherche concernant les différentes technologies, leur rôle dans le projet, comment les faire fonctionner entre elles, etc. ainsi qu'à des tests visant à vérifier le bon fonctionnement de chacune des technologies qui nous ont été imposées. Durant la période de vacances, notre travail était à la fois orienté recherche mais également production de code.

Les différentes phases de documentation et de recherche de résolution de bugs en tout genre représentent une part importante du temps que nous avons consacré à ce projet. Cela est en partie dû au fait que la technologie Helidon Nima 4.0 est encore en alpha et que sa sortie officielle est prévue pour fin 2023.

Ainsi, très peu de ressources concernant cette technologie sont accessibles sur internet. Nous avons donc dû lire beaucoup de pages de documentation, de codes présents sur GitHub ou sur des forums tels que Medium ou Stack Overflow pour pouvoir comprendre certains concepts, ce qui nous a permis, par la suite, de résoudre petit à petit les problèmes plus spécifiques auxquels nous avons dû faire face et avec le peu de ressources qui était disponible.

Conditions de travail du binôme :

Le travail en binôme a permis à ce que chacun puisse faire des recherches et travaille sur un aspect particulier du projet puis de renseigner à l'autre membre du binôme les solutions trouvées. Cela a donc permis à chacun de pouvoir poursuivre ou améliorer le travail l'autre une fois arrivé à un certain avancement de chaque tâche.

Cependant, notre dynamique de groupe a été freinée par des contraintes personnelles particulières qui ont rendu plus difficiles les réunions de binôme (créneaux horaires différents, parfois impossibilité d'être disponible en vocal, etc.).

Organisation du projet :

Ce projet s'articule autour de sa partie frontend et de sa partie backend.

Pour la partie frontend, nous utilisons les technologies React et Bulma. Les dossiers et fichiers relatifs à cette partie sont stockés dans le dossier suivant :

- src/main/frontend

Cette partie frontend est "buildée" grâce à Maven (cf. le fichier "pom.xml").

Pour la partie backend, nous utilisons la technologie Helidon Nima 4.0 actuellement en alpha. Les dossiers et fichiers relatifs à cette partie sont stockés dans les dossiers suivant :

- "src/main/java" pour les fichiers source Java
- "src/main/resources" pour quelques fichiers de configuration
- "src/test/java" pour le fichier source réalisant les tests de la classe Asm utilisant la technologie JUnit 5.9
- "src/test/resources" pour les artéfacts utilisés par les tests

Une fois le projet compilé à l'aide de la commande Maven "mvn package", une archive jar nommée "clonewar-jar-with-dependencies.jar" sera créée dans le dossier "target".

Lors de l'exécution de l'application, deux fichiers nommés "hashesBackupFile.txt" et "jarsBackupFile.txt" seront automatiquement créés. Ceux-ci sont des fichiers de sauvegarde automatiquement et entièrement gérés par l'application et permettent de maintenir de manière pérenne les données des archives analysées, même après arrêt et réexécution de l'application (veuillez retrouver des informations complémentaires à propos de notre obligation de passer par des fichiers de sauvegarde dans un paragraphe suivant).

Enfin, le fichier "pom.xml" décrivant les dépendances, utilisant des plugins et décrivant la façon de construire (build) l'application et son archive se trouve en racine du projet, tout comme le fichier de présentation du projet nommé "README.md" ainsi que cette présente documentation de développement nommée "dev.pdf".

Abstraction du code :

La détection de clones entre deux artefacts se fait aux moyens d'une reconnaissance entre les différentes instructions en bytecode générées dans les fichiers .class de ces artefacts.

Nous avons dû faire des choix lors de l'abstraction de ces instructions.

Tout d'abord nous avons pensé que les instructions du code que nous trouvions importantes se trouvaient au sein des méthodes, ainsi nous n'avons pas pris en compte les modifier des classes car celles-ci ne suffisent pas à donner un indice sur une éventuelle copie avec une autre classe d'un autre artefact.

Le nom des méthodes rencontrées n'est évidemment pas pris en compte dans le calcul des clones, que ce soit les déclarations ou les appels de méthodes au sein du projet peuvent être différentes entre deux artefacts et pourtant exécuter les mêmes instructions.
Les seuls appels de méthodes que nous avons intégré dans notre calcul des clones sont ceux qui font référence à une méthode appartenant à la librairie Java. En effet, dans ce cas ci il est important de garder comme information l'appel à une de ces méthodes "universelle" ainsi deux artefacts utilisant la méthode Integer.parseInt() auront une instruction similaire.
De plus, nous avons omis l'analyse des paramètres et le type de retour dans la reconnaissance. En effet deux méthodes pourraient induire le calcul de clones en erreur si l'une renvoie une liste en retour et l'autre ne renverrait aucun objet mais modifierait directement une liste donnée en argument.
Un cas pour les arguments serait d'avoir une méthode qui en utilise beaucoup pour exécuter son programme et une autre qui n'en aurait que deux tout en faisant la même chose.

Les instructions de chargement ou de stockage de variables locales dans la pile sont toutes considérées comme égales, en effet on ne prend pas en compte le numéro de la variable dans la pile comme information importante pour la reconnaissance, des variables pouvant avoir un ordre différent dans la pile de la méthode entre deux artefacts différents.

Pour le cas des chargements de valeurs constantes dans la pile ("LDC") nous avons pris en compte les cas des valeurs en compte, le fait qu'une valeur donnée en brute à une méthode par exemple nous a semblé important.

Karp-Rabin :

L'algorithme de Karp-Rabin consiste à faire une reconnaissance de sous chaîne dans une chaîne principale. En effet dans le cas de chaînes de caractères il s'agirait de reconnaître un pattern issu d'un texte. Pour cela on utilise une fonction de hachage et lorsque l'on compare la sous-chaîne courante avec la chaîne principale on renvoie l'indice du premier caractère de l'emplacement du pattern trouvé dans la chaîne principale.

Nous avons donc adapté cet algorithme à notre programme de reconnaissance de clones.

Ainsi nous avons appliqué cet algorithme utilisant un système de rolling hash et adaptant à une analyse sur des instructions en bytecode rencontrées au fur et à mesure de l'analyse d'un fichier .class.

Notre calcul de reconnaissance se fait donc en créant une fenêtre d'instructions. Nous avons choisi au vu des tests et de réflexion de prendre une fenêtre de taille 4, cela n'étant pas trop contraignant et permettant d'avoir un minimum d'informations sur un bout de code En effet, nous avons trouvé qu'une fenêtre de taille 1 est trop stricte car on ne compare que ligne d'instruction par ligne d'instruction sans contexte tandis qu'une fenêtre de taille 6 ou plus serait trop souple et empêcherait ainsi la détection de clones de quelques lignes.

Cette fenêtre sert à calculer la valeur (hash) que l'on peut donner aux paquets d'instructions que l'on rencontre.

Ces valeurs permettront d'identifier ces groupes. En effet, lorsque l'on analyse un autre artefact et que l'on compare les valeurs de hash des différents groupes, on peut savoir s'il existe deux valeurs identiques issues d'artefacts différents.

Nous n'avons pas réussi à trouver le moyen d'obtenir les lignes du fichier sources associées à l'instruction en bytecode correspondante. En lisant le guide de la librairie ASM nous avions trouvé une méthode d'un objet MethodVisitor qui permettait d'obtenir les lignes du fichier source cependant elle n'était appelée uniquement quand un label était rencontré (comme une déclaration de méthode par exemple) par manque de temps et de problème avec la base de données il nous a été impossible d'approfondir la lecture de la documentation pour trouver une solution et implanter une solution permettant de récupérer ces lignes.

Rolling Hash :

Nous avons implémenté un système dit "rolling hash" pour calculer les valeurs de hash des différents groupes d'instructions rencontrées. Cela permet un calcul optimisé d'une valeur de hash pour une fenêtre donnée.

Le rolling hash consiste à réutiliser les anciennes valeurs déjà calculées pour obtenir les nouvelles. Dans le cas d'une chaîne de caractère il suffit de soustraire la valeur du préfixe à la valeur de hash précédemment calculée et ajouter la valeur du nouveau caractère suffixe.

Nous avons adapté cette méthode de calcul au cas de l'analyse d'instructions en bytecode. En effet, chaque ligne d'instruction rencontrée possède une valeur de hash, on ajoute cette valeur de hash dans une liste de taille définie (dans notre cas la taille de la "fenêtre" est 4), lorsque l'on arrive à remplir la liste on calcule la valeur de la liste entière à l'aide d'une fonction de hachage.

Nous stockons la première valeur de hash calculée sur l'entièreté de la liste. Lorsque l'on rencontre une nouvelle instruction on calcule son hash puis on :

- supprime le premier élément (valeur de hash) de la liste
- soustrait le premier élément au hash de la liste stocké
- ajoute la valeur de la nouvelle instruction rencontrée en fin de liste
- ajoute la valeur de la nouvelle instruction rencontrée au hash stocké

Sachant que l'on obtient une valeur de hash pour la fenêtre à l'aide de l'**empreinte de Rabin** qui est une implantation souvent utilisée lors d'implantation de l'algorithme de Rabin-Karp.
On se sert donc d'un polynôme de degré k - 1 avec k la taille de notre fenêtre. Cette méthode nécessite un nombre premier qui sera élevé à la puissance de la position inverse dans la liste du hash et facteur du hash courant. Le nombre premier souvent utilisé est 101 comme convention mais nous avons pris le chiffre 2 dans notre cas.

Dans notre cas pour retirer la valeur de l'ancienne première ligne d'instruction nous devons soustraire : (valeur ancienne première ligne * nombre premier ^ taille de la fenêtre)

Pour ajouter le hash de la nouvelle instruction on doit donc multiplier le hash de la fenêtre par le nombre premier (pour élever les membres du polynôme d'un degré suite à la soustraction de l'ancienne valeur) puis ajouter la nouvelle valeur de hash de la ligne de bytecode.

Stockage en base de données :

À chaque calcul de hash (que ce soit pour la fenêtre entière la première fois ou le calcul des fenêtres suivantes) nous devrions ajouter dans la base de données un enregistrement comportant le nom du fichier, le numéro de la première ligne de la fenêtre ainsi que le hash de ce groupe d'instructions (qui compose la fenêtre). Chaque artéfact analysé représenterait une table à part entière de la base de données.

Il aurait également fallu créer une table servant à stocker les artéfacts eux-mêmes. En effet, lorsque les utilisateurs renseignent un artéfact (un fichier jar) à analyser depuis leur système sur la page web via un formulaire et un champ input de type "file", un principe de sécurité empêche que le chemin absolu de ce fichier soit connu par le serveur afin que l'arborescence des fichiers de l'utilisateur reste inconnues. Cependant, les données du fichier peuvent être transmises au serveur. Ainsi, pour garantir un accès à cet artéfact dans le temps et donc pouvoir par la suite accéder aux fichiers sources de cet artéfact lors de l'affichage du "diff" entre deux artéfacts par exemple, il faut stocker entièrement toute artéfact analysée au sein de la base de données.

Ainsi, pour comparer deux artefacts une de nos méthodes réaliserait une requête SQL permettant d'obtenir les hashes similaires entre deux tables (donc entre deux artéfacts différents ou non). Cela nous permettrait alors d'avoir un pourcentage de clone (de plagiat) entre les deux artéfacts et même les numéros de lignes responsable de ceux-ci.

Nous employons ici le conditionnel car deux de nos technologies imposées n'étant pas compatibles entre elles, nous avons dû faire autrement, tout en assurant les garanties de l'utilisation d'une base de données. Pour des explications à propos de ce problème d'incompatibilité avec lequel nous avons dû faire face et la solution que nous avons proposé, voir le paragraphe suivant "Problèmes persistants", notamment son sous-paragraphe "Incompatibilité entre les dépendances Helidon Nima 4.0 et Helidon SE DBClient". Cependant, les méthodes d'analyse et de comparaison entre deux artéfacts sont, à quelques précisions près, celles que nous aurions utilisé si nous avions pu implanter la base de données; au lieu de récupérer des informations au sein de la base de données, nous les récupérons dans les champs d'instances prévus à cet effet et simulant cette base de données.

Présentation générale de l'interface web :

La réalisation de la partie frontend s'est faite à l'aide de React.js et le framework Bulma.
Nous avons ainsi pu concevoir un site clair, structuré et responsive.

Il existe un composant principal appelé "Home" représentant la page sur laquelle ouvre le site, ce composant possède un membre qui est composé de plusieurs champs comme celui qui représente le chemin de l'archive qui sera envoyé au serveur pour analyse, les chemins des deux artéfacts à comparer, entre autres. La plupart de ces champs sont initialisés avec la chaîne vide et seront ensuite réinitialisés lorsque l'utilisateur effectuera des actions précises.

Nous nous sommes servis de deux formulaires pour l'interaction entre l'utilisateur et le serveur.

Tous deux sont liés à des actions onSubmit permettant d'effectuer les différentes actions demandées lorsqu'un clic est produit (par exemple, clic sur le bouton de soumission d'un formulaire).

Le premier formulaire possède un champ input de type "text" qui attend qu'un chemin soit fourni sous forme de texte. Ce champ input n'est pas de type "file" car dans ce cas la page web ne pourrait pas récupérer le chemin absolu du fichier fourni. Cela est un principe de sécurité garanti par le web. Si le champ reste vide et que le formulaire est soumis par l'utilisateur en cliquant sur le bouton qui lui est associé, le programme affiche une alerte avec un message stipulant qu'aucun fichier n'a été donné.
Le second formulaire demande de sélectionner deux artéfacts déjà analysés. Le formulaire demande également à ce que les deux champs soient remplis avant de soumettre, sans quoi une alerte sera émise.
Une fois qu'un formulaire est soumis, une fonction est appelée. Celle-ci permettant de changer des valeurs de champs et de déclencher des requêtes vers le serveur dont le résultat sera affiché à l'utilisateur.

Enfin une liste des différents artefacts déjà analysés dans la base de données est affichée.

Pour que l'interface soit claire et plaisante nous avons utilisé Bulma qui possède des classes prédéfinies pour certains objets comme les boutons, champs input etc… Les couleurs sont également issues des classes de style de Bulma.

Nous avons eu certaines difficultés avec ce framework, en effet dans le cas des icônes il nous était impossible de charger les icônes existantes données dans la documentation car il fallait les avoir importées en amont.

Afin d'en savoir plus en ce qui concerne notre attention portée à l'ergonomie et le caractère permissif et responsive de cette page web, veuillez lire dans paragraphe "Améliorations réalisées" le sous-paragraphe "Attention portée sur l'expérience utilisateur avec un site web ergonomique et permissif".

Problèmes persistants :

Incompatibilité entre les dépendances Helidon Nima 4.0 et Helidon SE DBClient :

Initialement, nous devions utiliser Helidon Nima 4.0, actuellement en alpha, pour la partie serveur de notre application et Helidon SE DBClient en ce qui concerne la persistance de celui-ci.

Après de très nombreuses recherches, documentation et de nombreux essais, nous avons appris lors de notre soutenance intermédiaire que ces deux technologies sont en fait incompatibles. Nous avons compris que pour pouvoir utiliser ces deux technologies conjointement, il nous fallait écrire 3 fichiers pom.xml. Un fichier parent qui permet l'utilisation simultanée des deux fichiers pom enfants et donc un fichier pom enfant qui renomme la dépendance Helidon SE DBClient en utilisant le plugin Maven shade ainsi qu'un second fichier pom enfant qui recense toutes les autres dépendances et utilise la dépendance Helidon SE DBClient renommée.

Ayant compris cela, nous nous sommes documentés sur l'utilisation du plugin shade de Maven et la façon dont utiliser conjointement ces 3 fichiers pom. Malgré nos recherches et de multiples essais, aucun de ceux-ci n'était suffisamment fructueux pour que l'on puisse s'en servir dans le projet.

Comme nous avions déjà écrit des classes Java utilisant simultanément ces deux technologie, déjà recopié les méthodes de test du serveur et de l'accès à la base de données présentées sur le dépôt GitHub d'exemple d'utilisation d'Helidon de son auteur et déjà écrit le fichier de configuration "src/main/resources/application.yaml" ainsi que des requêtes SQL au sein de ce fichier en permettant d'éviter des injections de code au vu de la

manière dont nous les utilisions, nous avons décidé de ne pas supprimer tout ce travail et de le déposer sur notre dépôt GitLab dans un dossier à part nommé “clonewarWithDBClient”.

Comme ce problème nous empêchait de continuer l’avancement du projet, nous avons décidé de réfléchir à une solution qui permette d’avoir une utilisation de l’application et un comportement de celle-ci identique par rapport à si nous avions pu utiliser la base de données Helidon SE DBClient.

Nous avons donc fait en sorte que tout ce que nous aurions stocké dans le base de données serait stocké dans un champ d’instance privé qui est un dictionnaire et qui stocke donc une chaîne de caractère représentant de manière unique l’artéfact analysé ainsi que la liste de ses hashes; son type est donc HashMap<String, ArrayList>Long>> et se nomme “map” . Un autre champ d’instance privé de type ArrayList<String> et nommé “analyzedJars” stocke les chemins absolus des artéfacts analysés. Son type n’est pas ArrayList<Path> car nous n’utilisons pas les éléments qu’il stocke comme des chemins mais comme des simples chaînes de caractères. Le contenu de ces deux champs est sauvegardé dans des fichiers de sauvegarde autonomes (cf. “hashesBackupFile.txt” et “jarsBackupFile.txt”).

De plus, afin de faire en sorte que les utilisateurs n’aient pas à réanalyser leurs artéfacts après chaque extinction de l’application, nous avons fait en sorte d’écrire des méthodes de restauration des données sauvegardées dans ces fichiers afin de garantir une pérennisation des informations issues des analyses d’artéfacts et donc de garantir un comportement identique à l'utilisation d’une base de donnée. Cependant, comme notre application doit réaliser des écritures dans un fichier durant les analyses d’artéfacts, celles-ci sont ralenties par ces écritures. En implantant une base de données à notre application dans le futur, notre méthode d’analyse sera plus performante.

Réalisation des tests :

Avec Eclipse, l'intégralité de nos tests passent correctement. Cependant, sous IntelliJ, les tests ne passent que si les fichiers de sauvegarde sont présents, ainsi les analyses sont déjà faites donc l’instance de la classe Asm recharge les informations qu’ils stockent et les tests de comparaisons entre artéfacts peuvent être effectués.

En revanche, IntelliJ refuse d'appeler la méthode analysis() au sein des tests. Nous nous rendons compte de cela dès le test nommé “firstProject”. En effet, dans la stack trace, on se rend compte que le problème vient du fait qu'IntelliJ ne veut pas appeler cette méthode. Donc aucune analyse ne peut se faire et donc la plupart de nos tests ne peuvent donc pas passer sous IntelliJ. Cependant nous nous sommes bien assurés que tous ceux-ci passent bien sous Eclipse et donc que nos méthodes sont toutes correctes.

Aussi, lorsque l'on réalise la commande mvn package, une erreur nous dit "There was an error in the forked process TestEngine with ID 'junit-jupiter' failed to discover tests". Nous avons essayé de résoudre ceci mais après plusieurs recherches et tests, nous n'y sommes pas parvenus. Pour pallier ce problème, nous avons mis le contenu de notre class AsmTest en commentaire.

Bug interne à Helidon Nima 4.0 :

Helidon Nima 4.0 étant actuellement en alpha et sa sortie officielle n'étant prévue que pour fin 2023, des bugs sont toujours présents en son sein.

Nous en avons notamment rencontré une qui induit qu'un rechargement de la page web provoque l'affichage du message suivant : "If-None-Match".

"If-None-Match" est un nom d'en-tête HTML. Cet en-tête permet de retourner un "304 Not modified" si le contenu de la page web n'a pas été modifié.

Pour pallier ce bug, il suffit de recharger de nouveau la page web une ou deux fois. Toujours dans un contexte d'attention portée à l'expérience utilisateur, nous prévenons l'utilisateur du site web de ce léger bug et de la manière de retrouver l'affichage du site grâce à un encart de texte dédié à cela en tête de page.

Difficultés rencontrées :

Lors de la conception de ce projet nous avons été confrontés à plusieurs difficultés qui nous ont freinés dans notre avancement sur celui-ci.

Création d'un jar exécutable incluant les dépendances :

Nous avons rencontré des problèmes lors du lancement du jar qu'une instruction "mvn package" produisait car aucun fichier "manifest" n'était détécté. Nous avons donc fait en sorte qu'un fichier "manifest" soit créé automatiquement en utilisant le plugin "maven-jar-plugin" de "org.apache.maven.plugins".

Ensuite, nous avons rencontré cette erreur :
"Error: Unable to initialize main class fr.uge.clonewar.NimaMain Caused by: java.lang.NoClassDefFoundError: io/helidon/common/http/InternalServerException"
Nous avons donc permis la création d'un jar contenant toutes les dépendances nécessaire grâce à l'utilisation du plugin "maven-assembly-plugin" de "org.apache.maven.plugins". Comme le stipule la documentation de ce plugin, nous avons ajouté quelques clause dans la configuration de ce plugin afin de préciser quel est la fonction main à appeler lors de l'exécution du jar et nous avons aussi ajouté quelques clauses à la configuration de ce même plugin afin de renommer le jar produit (en ajoutant un suffixe à notre nom de projet). Cela permet ainsi la création automatique d'un jar directement exécutable nommé "clonewar-jar-with-dependencies.jar".

Configuration des CORS du serveur Helidon Nima 4.0 :

Nous avons aussi rencontré des problèmes lors des requêtes que nous effectuions depuis notre site web. En analysant la console de notre site, nous nous sommes rendus compte que cela était dû au “Cross-Origin Resource Sharing” (CORS).

En effet, il était possible de récupérer des réponses du serveur depuis notre page web en cliquant sur un bouton et en faisant un fetch sur une url de l’API du serveur par exemple, lorsque le serveur et le site web étaient tous les deux sur http://127.0.0.1:8080.

Cependant, des problèmes de CORS apparaissaient si le serveur était sur http://127.0.0.1:8080 et que le site web était quand à lui sur http://localhost:8080. Il en était de même lorsque le serveur était sur http://127.0.0.1:8080 et que le site web était lancé en mode de développement et donc sur http://127.0.0:3000. En effet, client et serveur n'avaient pas la même origine donc des problèmes de ressources cross-origin (CORS) survenaient.

Nous avons donc dû importer la dépendance “helidon-nima-webserver-cors” de io.helidon.nima.webserver. Ensuite, nous avons créé une instance de la classe CorsSupport en tant que constante dans les classes qui définissaient les routes du serveur. Pour créer le CorsSupport ayant le comportement que nous désirions, nous avons utilisé un CrossOriginConfig qui autorise les origines suivantes "http://127.0.0.1:8080", "http://localhost:8080", "http://127.0.0.1:3000" et "http://localhost:3000" pour toutes les méthodes (GET, POST, PUT, DELETE, etc.) et n'avons donc pas spécifié de méthode autorisées étant donné que l'on autorise toutes celles par défaut (donc pas d'appel à la méthode allowMethods()).

Enfin, nous avons utilisé cette instance de CoreSupport partout où l'on configurait le routage de notre serveur. Par exemple :
“.register("/", StaticContentSupport.builder("/static-content"));” devient alors
“.register("/", CORS_SUPPORT, StaticContentSupport.builder("/static-content"));”.

Nous pouvons donc désormais communiquer entre notre site web React et notre serveur Helidon Nima 4.0 à la fois lors de la phase de développement (serveur sur le port 8080 et React sur le port 3000) ce qui est très pratique, mais aussi lors de la phase de production (le serveur et le site web sur le port 8080, et ce, peu importe que l'utilisateur utilise l’adresse “http://127.0.0.1” ou “http://localhost” ce qui est une fois de plus plus pratique et plus convivial pour l'utilisateur et son expérience sur le site.

Requêtes et réponses asynchrones (async/await) :

Le site web réalise différentes requêtes au serveur. Celui-ci peut fournir une réponse en plus ou moins de temps. L’application se voulant dynamique, nous désirions que les réponses données par le serveur s’affichent dynamiquement dès leur réception par le site web.

Ne connaissant précédemment pas le principe de fonction asynchrone, nous avons eu des problèmes dans un premier temps car nous n’obtenions les réponses du serveur qu’après une autre action produisant une réinitialisation de certains champs ou alors tout simplement

pas du tout, du fait que le champ censé être réinitialisé à la valeur de la réponse était réinitialisé à sa valeur par défaut car la réponse n'était pas encore arrivée.

Après quelques recherches, nous avons compris le principe de fonction asynchrone. Toutes les que l'on effectue depuis le site web à destination du serveur requêtes sont asynchrones et vérifiées par une capture (catch) des erreur ainsi qu'un affichage adéquat en fonction du statut de la réponse (500 si tout va bien ou non) et s'il y a erreur ou non. Dans tous les cas, un message clair sera affiché à l'utilisateur afin de l'informer correctement.

Détection de l'emplacement où stocker notre site web pour les phases de production et de développement :

Lors de la phase de développement, nous avons été confrontés au fait que nous ne savions pas où développer notre page web ni où installer les modules qui lui était nécessaire. Nous ne savions pas non plus comment faire pour que le serveur Helidon Nima 4.0 charge correctement notre site web lorsque l'on se rendait en un point précis de celui-ci, comme par exemple "127.0.0.1::8080/index.html".

Après de nombreuses recherches, documentation et essais, nous avons finalement trouvé que nous avions besoin d'utiliser la dépendance "helidon-nima-webserver-static-content" de "io.helidon.nima.webserver". Ensuite, il nous a fallu utiliser la classe StaticContentSupport puis ajouter au routeur du serveur l'instruction suivante :
".register("/", StaticContentSupport.builder("/static-content"));"

Celle-ci signifie que lorsque l'on se rend à la racine ("/") du serveur, les ressources statiques stockées dans le dossier "target/classes/static-content" sont chargées et la page "index.html" présente dans ce dossier est alors accessible à l'adresse "127.0.0.1:8080/index.html".

Dans le sous-paragraphe suivant, nous détaillons comment nous installons de manière locale les dépendances et modules nécessaires au bon fonctionnement de notre site web et comment nous les copions vers ce dossier afin que le site puisse se charger correctement une fois l'archive de notre application créée puis exécutée.

Installation automatique par Maven des dépendances de npm et du framework Bulma :

Pour faire fonctionner React, il est nécessaire d'installer des modules node. Ces modules peuvent être installés par le gestionnaire de paquets de Node.js nommé npm.

Nous avons configuré notre pom.xml afin que Maven télécharge automatiquement npm en local dans un dossier que nous lui avons spécifié nommé "/src/main/frontend" puis lance la commande "npm run build" qui permet de créer un répertoire de "build" et de créer un version de déploiement optimisée de notre site web. Par la suite, la commande "npm install bulma" est lancée afin d'ajouter les modules bulma aux modules nécessaires. Tout ceci a

été rendu possible par le plugin “frontend-maven-plugin” de “com.github.eirslett” dont nous avons dû étudier la documentation.

Ensuite, nous avons configuré notre pom.xml pour qu’une copie du dossier “/src/main/frontend/build” soit effectué vers le dossier “/target/classes/static-content”. Tout ceci a été rendu possible par le plugin “maven-resources-plugin” de “org.apache.maven.plugins” dont nous avons dû également étudier la documentation.

Tout cela permet donc d’installer localement et aux bons endroits toutes les dépendances et modules nécessaires à la bonne paquétisation sous forme d’archive de notre application.

Améliorations réalisées :

Affichage des chemins des artéfacts dépendant du système de l’utilisateur :

La site web de notre application affiche dynamiquement la liste des artéfacts analysés et donc leur chemin absolu. Nous avons permis à ce que les séparateurs de dossiers soient bien dépendant du système de l'utilisateur pour permettre à ce que ceux-ci puissent en faire des copier/coller aisément, ce qui améliore l'expérience utilisateur et évite que que quelqu'un que ne soit pas suffisamment expérimenté ne comprenne pas pourquoi il ne peut pas retrouver son jar dans son arborescence directement en copiant collant un chemin dans son explorateur de fichiers à cause du fait que les séparateurs de dossier ne soit pas ceux de son système.

Procédé d’anti-conflit entre les différents artéfacts analysés et comparés:

La méthode responsable de la comparaison entre deux artéfacts est la méthode nommée “percentageCloning”. Dans celle-ci, nous récupérons pour les deux artéfacts donnés à comparer tous les hashes qui leur sont respectivement associés. Lors du parcours du dictionnaire (le champ d’instance privé “map”) tous les hashes qui ont pour début de clé le nom du chemin de l’artéfact concaténé avec " : " sont récupérés, et ce pour les deux artéfacts. Cela permet à ce qu’il n’y ait pas de collisions entre des artéfacts de chemins très similaires. Par exemple, si on analyse un artéfact au chemin absolu de la forme “src/toto/MyToto.jar” et que l’on analyse par la suite un artéfact au chemin absolu de la forme “src/toto/MyToto.jarsrc/toto/MyToto.jar” alors grâce au séparateur " : " que nous avons ajouté entre le chemin de l’artéfact analysé et le fichier de cet artéfact analysé, cela évite un conflit entre les deux.

Nous aurons dans ce cas, possiblement les clés suivantes :
“src/toto/MyToto.jar : fr/uge/Toto.class” ainsi que “src/toto/MyToto.jarsrc/toto/MyToto.jar : fr/uge/Toto.class”

Si un utilisateur demande de comparer un artéfact avec l'artéfact "src/toto/MyToto.jar", aucune collision ne sera possible avec l'instruction :
"key.startsWith("src/toto/MyToto.jar" + " : ");"

Attention portée sur l'expérience utilisateur avec un site web ergonomique et permissif :

En réalisant ce projet, nous avons porté une attention particulière sur l'expérience utilisateur générale et notamment l'ergonomie de notre site web.

Nous avons donc fait en sorte que tout soit le plus simple possible pour l'utilisateur. Une fois que l'utilisateur a renseigné le chemin absolu de l'artéfact qu'il désire analyser, l'analyse démarre et l'utilisateur voit que celle-ci est en cours grâce à l'apparition automatique d'une barre de progression. Une fois l'analyse terminée, cette barre de progression est remplacée par un encart spécifique affichant la réponse du serveur. Il en est de même lors d'une comparaison entre deux artéfacts. Par la même occasion, la liste des artéfacts déjà analysés est elle-aussi automatiquement mise à jour ainsi que les sélecteurs d'artéfacts pour la comparaison.

De plus, si des analyses d'artéfacts ont déjà été réalisées, la liste de ceux-ci est automatiquement mise à jour dès l'ouverture du site web, et ce, même après extinction puis redémarrage de l'application (serveur y compris).

Nous avons également veillé à expliquer suffisamment clairement ce qu'il faut faire pour analyser ou comparer des artéfacts.

Aussi, nous avons fait en sorte que le site web soit le plus permissif possible. Ainsi, si l'utilisateur déclenche une analyse ou une comparaison sans avoir renseigné d'artéfact, une alerte lui apparaîtra alors en lui renseignant la raison de cette alerte et ce qu'il doit faire. De plus, même si l'utilisateur ajoute involontairement des espaces avant ou après le chemin qu'il aura renseigné dans le but d'effectuer une analyse de celui-ci, le site web réalisera tout de même une requête correct au serveur car nous avons fait en sorte que l'entrée utilisateur soit transformé en remplaçant les espaces encodés par les caractères "%20" par de réels caractères espace puis en appelant la fonction trim() sur le résultat obtenu.

Les éléments cliquables comme les boutons ou les éléments de la liste des artéfacts déjà analysés ont un design faisant inconsciemment comprendre que cet élément est cliquable, en plus d'avoir ajouté cette information sous forme de texte.

Enfin, notre site web est responsive et s'adapte donc parfaitement à tous types de plateformes, du smartphone au téléviseur en passant par l'ordinateur. Il se peut cependant que la taille des chemins renseignés induisent un léger décalage sur de très petits écrans.

Création d’un README de présentation de notre dépôt GitLab :

Toujours dans une optique d’expérience utilisateur accrue, nous avons également écrit un README détaillant le but de ce projet, comment réaliser l’installation de l'application, comment s’en servir en mentionnant également les auteurs du projets et son statut d’avancement.

Possibilité de réanalyser un même artéfact après modification de celui-ci:

Afin de pouvoir analyser un artéfact qui a été modifié mais qui est placé au même endroit et qui a le même nom que précédemment, nous avons fait en sorte qu’avant chaque analyse, les informations stockées à propos d’un artéfact de même nom et de même emplacement soient supprimées afin de pouvoir faire une toute nouvelle analyse de ce nouvel artéfact sans qu’il ne puisse y avoir de données résiduelles de l’ancien artéfact de même nom et de même emplacement.

Pour ce faire, nous avons écrit cette instruction avant chaque analyse :
“map.entrySet().removeIf(entry -> entry.getKey().startsWith(jarLocation + " : "));”

Avec une base de données implantée, dans le cas où l’on ait une table par artéfact analysé, il aurait suffit de supprimer entièrement la table de l’artéfact à analyser si celle-ci existe (et donc si cet artéfact placé au même endroit et ayant le même nom a déjà été analysé) puis de réaliser l’analyse de celui-ci en créant une nouvelle table vide dédiée à cet artéfact.

Conclusion :

Développer ce projet nous a donné beaucoup de fil à retordre, nous a fait poser beaucoup de questions et nous avons à de très nombreuses reprises cru que nous ne trouverions pas la solution à nos problèmes. Mais, à force de lecture de documentation, de recherches et d’essais, nous y sommes parvenus.

Nous avons non seulement dû implanter à la fois la partie frontend et backend d’une application d’analyse d’artéfacts et de reconnaissance de clones mais également permettre une liaison entre ce back et ce front.

Développer cette application nous a donc permis d’acquérir une première expérience fullstack, de comprendre les différentes étapes de réflexion nécessaires à la réalisation d’applications fullstack, de savoir comment implanter concrètement certains concepts nécessaires au bon développement de telles applications ainsi que de comprendre la nécessité d'adapter certains d’algorithmes permettant de répondre à un besoin spécifique.

Ce projet nous a appris à rechercher, parfois profondément, voire très profondément, des solutions aux problèmes que nous rencontrons en utilisant toutes les ressources que nous avons à notre disposition, soit, principalement, la documentation des technologies.

Ce projet nous a aussi appris, lorsque nous faisons face à un problème trop spécifique, à récolter différentes informations, que ce soit dans les documentations, sur les dépôts git officiels des auteurs des technologies avec lesquelles nous devons travailler ou simplement des dépôt git de particuliers ainsi que sur des forums tels que Medium ou Stack Overflow puis à adapter les solutions récoltées à notre problème plus spécifique ou bien à réorienter nos recherches grâce aux connaissances que celles-ci nous ont permis d'acquérir.

Enfin, réussir à développer un projet d'une si grande ampleur pour la première fois nous a permis d'appliquer de nombreuses connaissances ainsi que des bonnes pratiques que nous avons pu acquérir au cours de nos précédentes années universitaires et donc de nous rendre compte de notre compétence dans le domaine du développement, bien que nous ne pensions pas y parvenir à la lecture de l'énoncé de ce projet et au vu du temps qui nous était imposé et du peu de ressources disponibles sur certaines des technologies imposées, notamment Helidon Nima 4.0 qui est actuellement en alpha et dont sa sortie officielle est prévue dans un an, fin 2023.