UQAM

RAPPORT DU LIVRABLE

ERIC GAGNON - GAGE22037705

JIMMY PELLETIER - PELJ29088708

MARC-ANDRÉ BOUDREAULT - BOUM04086809

TRAVAIL PRÉSENTÉ

À

SEBASTIEN MOSSER

MGL7460

RÉALISATION ET MAINTENANCE DE LOGICIELS

26 AVRIL 2020

Ce rapport présente l'état de situation du projet technique en version finale, ainsi que les améliorations que nous envisagerions pour livrer un logiciel de gestion de produits bancaires viable.

Description du projet :

Comme l'exige la spécification, nous avons trois modules : bank, client et employee.

Tous les modules ont été découpés en architecture 3 couches : présentation, domaine et infrastructure. La couche présentation est ce qu'on expose pour communiquer avec notre domaine. La couche domaine contient notre logique d'affaires et la couche infrastructure contient tout ce qui est services techniques permettant d'accéder aux données et à la configuration technique du module.

Chacun des modules utilise le framework de test JUnit et Mockito permettant d'effectuer les essais unitaires automatisés. Chaque module a son propre fichier « pom » avec sa configuration nécessaire pour générer le rapport JaCoCo utile pour le serveur d'analyse statique SonarQube.

Le module bank a été développé avec l'aide du framework Spring Boot et Spring Data JPA. Nous avons décidé d'exposer des services REST permettant d'effectuer les diverses opérations de gestion de produits bancaires permettant d'interagir avec les modules de client et employee. Ce module contient une suite de tests automatisés et une suite de tests d'acceptations. Cette suite de tests d'acceptations a été développé avec l'aide du framework Cucumber et RestAssured.

Les modules client et employee ont été développés à l'aide d'Apache Commons CLI qui nous permet d'effectuer la gestion d'arguments reçus par ligne de commande. De plus, ces modules sont responsables de faire la gestion des appels HTTP au serveur de la banque.

Finalement, des scripts Bash ont été utilisés pour encapsuler les appels aux .jar.

Retour sur les commentaires du livrable intermédiaire

1. Commentaire sur group id

Le nom du group id de notre application est bel et bien loremipsum. Nous avons décidé de nommer notre équipe loremipsum et même sur docker hub nous avons utilisé ce nom d'équipe : https://hub.docker.com/u/loremipsumbank. Notre miroir public sur github aussi est avec loremipsum : https://github.com/egagnon77/loremipsum-bank.

2. Grand nombre de if/else statements dans le CommandLineProcessor

Nous aurions dû utiliser un design pattern de Command pour éviter les nombreux if/else statement, mais nous ne l'avons pas mis en place, ce sera pour une version ultérieure de notre application. Ce n'était pas nécessaire au bon fonctionnement de l'application.

3. Centralisation des loggers

Nous avons centralisé les loggers dans le but de pouvoir faire des tests unitaires sur l'utilisation de ces loggers. Ce n'est pas vraiment nécessaire, mais on ne considère pas que ça ajoute une complexité si grande que ça.

4. Nom des classes testées dans les classes de test

Nous avons décidé d'appeler nos classes testés « testedClass », car il y a certaines classes avec des classes « Mock » et c'est plus facile de repérer la classe en cours de test.

Forces du projet :

1. Mise en place de pipelines et processus

Nous avons mis en place un pipeline de CI. Ce pipeline est divisé en deux parties:

1. Pipeline de master
2. Pipeline de merge request

La branche master est une branche protégée et en ce sens, il est impossible de pousser directement dans cette branche. Les développeurs doivent faire une demande de merge request sur Gitlab pour pouvoir débuter l'intégration de la nouvelle fonctionnalité dans master. C'est ici que le pipeline de merge request débute. Le merge request a été configuré pour qu'il soit impossible d'accepter un merge à moins que toutes ces conditions soient vraies:

1. La branches du merge request doit être en fast-forward avec master. De cette façon, nous sommes certains que les validations de qualités incluent les derniers développements de master (évite les régressions).
2. Le pipeline de merge request doit être en succès. Ce pipeline exécute le build, les tests et l'analyse statique du code avec SonarQube. Le gating de SonarQube gratuit nécessite une cote de A pour la Reliability, Security et Maintainability, de 80% de couverture de code ainsi que 3% de duplication maximum.
3. Avoir résolu tous les commentaires du code review effectués par les pairs.

Lorsque le merge request est accepté, il est fusionné dans la branche master et c'est dans ce pipeline que la construction de l'image Docker de notre server et elle est ensuite poussée dans le registre de Docker Hub automatiquement.

2. Utilisation de Docker

Nous avons utilisé le container docker PostgreSql pour notre serveur de base de données. Cette décision a facilité la mise en place de la base de données et nous avons pu ajouter la persistance rapidement dans notre application avec ce choix.

Nous faisons aussi le déploiement, de notre server dans une image Docker et elle est envoyée dans le registre de Docker Hub. Le système de version de notre image est généré selon le build number et la dernière version est "tag" à "latest". De cette, façon il est facile de récupérer en tout temps la dernière version.

Nous avons aussi ajouté dans le code source un fichier docker-compose pour lancer notre système bancaire. Ce fichier configure la connexion entre le server avec la BD (port mapping), tout en spécifiant les versions des images à utiliser. Cette méthode, qui en une ligne de commande, simplifie énormément le démarrage du server bancaire.

3. Architecture DDD

Nous avons misé sur une architecture qui s'inspire du DDD (Domain Driven Design). Le domaine n'ayant aucune dépendance sur les autres couches du système est facilement testable et réutilisable. Les autres couches telles que l'infrastructure dépendent seulement du domaine et l'API dépend seulement du domaine et utilise l'infrastructure par injection de dépendances.

4. Traçabilité du code avec les user stories

Chaque commit effectué dans le repository est lié à un user story (issue dans GitLab). Ceci est fait en ajoutant par exemple "Ref #1" dans le message de commit pour que GitLab soit en mesure de lier l'issue #1 avec le commit.

5. Forte couverture par des tests et scénarios d'acceptations automatisés

Deux niveaux de tests sont en place dans notre application soit les tests unitaires et les tests d'acceptations. Pour les tests unitaires, chacun des comportements des méthodes publiques de chacun des modules est testé de façon automatisée. Pour les tests d'acceptations, chacun des scénarios d'acceptations d'un user story est complétement automatisé via le framework de test Cucumber. Les tests d'acceptations possèdent leur propre base de données en mémoire de type H2. Cette base de données est alimentée via un fichier "data.sql" qui se trouve dans le dossier ressources du module de test.

6. Tests d'intégration

Notre stratégie pour les tests d'intégrations est de faire le bout en bout complet. Donc, nous avons utilisé un script bash pour lancer toutes les commandes de client et employee, et ce, en nous connectant sur notre server et BD Postgres. Notre choix de tester toutes les commandes est pour s'assurer qu'il n'y a aucun bris d'interface entre nos modules et ceci est très important pour éviter un bris de service en production.

Faiblesses du projet :

1. Gestion de dépendances non centralisées

Notre gestion de dépendances Maven est aussi déficiente et peut porter à confusion, car les versions sont répétées inutilement dans le fichier de configuration (pom) de chaque module. La centralisation vers le pom parent sera la solution. Une meilleure gestion des dépendances devrait être faite ultérieurement dans le but d'éviter des conflits de version sur des dépendances.

2. Manque d'utilisation de design pattern

Dans les modules client et employee, nous avons une classe « CommandLineProcessor » permettant d'effectuer l'opération reçue. Un design pattern est à prévoir pour être en mesure de gérer toutes les opérations possibles au lieu d'un bloc de if/else statement. Un exemple de design pattern à utiliser serait « Strategy ».

3. Compatibilité d'Apache CLI & Windows

Nous avons découvert une incapacité pour l'api Apache Commons Cli sous Windows qui est incapable de discerner les arguments sans signe d'égalité pour les paramètres de format long (ceux qui sont représentés par un "--"). Par exemple, la librairie n'arrive pas à distinguer le paramètre suivant: "--list CLIENT_NAME" tout simplement à cause de l'espace entre les mots. L'idée est donc de boucler dans les paramètres lors de la lecture des arguments dans le script Bash et d'ajouter le signe "égal" (=). Actuellement, si d'autres paramètres sont ajoutés à la ligne de commande, on ne les passe tout simplement pas au Java, ce qui enfreint l'intégrité puisque ces règles doivent être au niveau du domaine dans les applications Java concernées.

Comparaison de plateformes de CI/CD

Dans le travail demandé, nous avons utilisé deux plateformes différentes pour la réalisation de l'intégration et du déploiement en continu. La première plateforme que nous avons utilisée pour réaliser les pipelines finaux est Gitlab. Par défaut, il offre la possibilité d'utiliser des images Docker pour réaliser les différentes étapes d'un pipeline. Travis CI lui offre deux choix par défaut, utiliser des VM préconfigurer selon le langage que l'on utilise et/ou utiliser des images Docker. Travis donne plus de choix aux utilisateurs gratuit parce que pour avoir une VM sur Gitlab c'est plus complexe, car nous devons générer notre propre Gitlab Runner. Avec Travis, nous avons utilisé la VM préconfigurer pour un environnement Java et les images Docker avec Gitlab. Techniquement les deux plateformes se ressemblent, il est facile de convertir un pipeline d'une à l'autre technologie. Les fichiers de configuration utilisent le même langage YAML.

Il a été possible de faire les mêmes étapes dans les deux plateformes, mais il a été plus simple sur Travis. Par exemple, pour faire les tests d'intégrations et le déploiement avec Docker, la plateforme sur GitHub ne demandait pas de configuration supplémentaire parce que la VM contient déjà tous les outils nécessaires. Avec Gitlab c'est plus compliqué, après plusieurs recherches sur le web, il a fallu utiliser une image Docker avec un service Gitlab DID (docker in docker) pour pouvoir faire le déploiement.

La faiblesse de Travis est qu'on ne peut partager aucun output d'un stage à un autre, ce qu'il est possible de faire dans Gitlab.

La faiblesse de Gitlab est que dès qu'on veut faire de quoi de custom, il faut absolument créer un Gitlab runner. Ce qui rend les processus plus ardus.