Agilité : Continuous Integration

TP1 Agilité

Intégration, Livraison, Déploiement continus d'un projet

Jérôme Buisine
jerome.buisine@univ-littoral.fr

3 septembre 2020

Durée : 3 heures

L'objectif de ce TP est de faire communiquer un ensemble d'outils permettant lors du développement d'un projet d'assurer à la fois son suivi, sa robustesse et son déploiement.

1 Travail

La mise en place d'une telle architecture de projet doit se faire dans un ordre bien défini. Il vous sera demandé de bien suivre les différentes étapes du TP **dans le même ordre que présentées**. Durant ce TP vous allez configurer chacun des outils requis par le client afin qu'il puisse procéder au bon développement du projet. Le contenu du projet restera ici à titre d'exemple pour vous permettre la bonne mise en place de l'architecture. Dans les TP suivants, il vous sera demandé d'utiliser la même architecture du projet mais sur un nouveau contenu.

Voici l'ensemble des outils / langages qui seront utilisés pour ce TP :
— 1. IntelliJ/Idea (version Ultimate) comme environnement de développement ;
— 2. git comme système de versionning du projet ;
— 3. Gitlab comme serveur d'hébergement de votre projet git (permettant l'accès à des outils d'intégration, livraison, deploiement continus) ;
— 4. Maven qui sera utilisé dans notre cas pour la gestion des dépendances du projet ;
— 5. Le serveur d'API REST sera fourni et devra être exécuté via Java.
— 6. Le framework JUnit 5 (Java Unit) sera solicité pour l'intégration de tests unitaires du serveur d'API REST ;
— 7. Les langages web HTML/CSS/Javascript côté front permettant un visuel côté client (navigateur web) ;
— 8. PMD comme analyseur de qualité de code ;
— 9. Sonarqube comme serveur de rapports sur la qualité du code (couplé avec l'analyseur PMD) ;

2 Initialisation du projet

L'étape de ce TP consistera à initialiser votre projet Java puis de l'héberger sur la plateforme Gitlab.

2.1 Création du projet

À partir de votre IDE (Intellij), créer un projet Maven. Choisir un SDK 1.8 minimum puis cliquer sur suivant. Nommer votre projet `TP1-Agility`, puis cliquer sur terminer.

2.2 Initialisation de git

Via un terminal, accéder au dossier de votre projet puis initaliser `git` via son interface en ligne de commandes :

```
git init
```

Créer une nouvelle branche nommée ‘develop‘ à partir de la branche ‘master‘ :

```
git checkout -b develop
```

Afin de simuler un environnement projet réel d'entreprise, il vous sera demandé durant ce TP d'utiliser `git` et la gestion de ses branches de la manière suivante :

- **`master`** : branche de production du projet (livrable à fournir pour le client).
- **`develop`** : branche de développement du projet avant livraison.
- **`feature/XXXXX`** : branche liée à un développement en particulier où `XXXXX` est un nom donné à cette branche spécifiquement au développement demandé.

Le processus d'utilisation de git durant le TP sera le suivant :

- 1. Pour chaque nouveau développement, créer une branche ‘feature/XXXXX‘ à partir de la branche ‘develop‘.
- 2. Une fois un développement terminé, fusionner les modifications de la branche ‘feature/XXXXX‘ vers la branche ‘develop‘. Supprimer la branche ‘feature/XXXXX‘.
- 3. Mettre à jour le projet git distant hébergé sur Gitlab.
- 4. Réaliser une livraison si demandée : mise à jour de la branche ‘master‘ par rapport à la branche ‘develop‘ puis mise à jour du projet git distant (serveur Gitlab).

Voici quelques commandes qui vous seront utiles pour mener à bien le versionning de votre projet :

Gestion des branches :

```
// Liste toutes les branches disponibles
git branch

// Crée une nouvelle branche
git branch ma-branche

// Change de branche de développement
git checkout ma-branche

// Création et changement de branche courante
git checkout -b ma-branche

// Supprime une branche
git branch -d ma-branche

// Depuis la branche ‘develop‘, récupération des modifications de ‘ma-branche‘
git merge ma-branche
```

Ajout de modifications :

```
// Montre le status des fichiers versionnés ou non versionnés
git status

// Ajoute le fichier ‘file.txt‘ pour un prochain commit
git add file.txt

// Ajoute tous les fichiers pour un prochain commit
```

```
git add .

// Visualisation des différents commits de la branche courante
git log --oneline
```

Intéraction avec le serveur d'hébergement :

```
// Crée un commit avec les fichiers 'ajoutés'
git commit -m "mon premier commit"

// Énumère les serveurs d'hébergements référencés
git remote -v

// Publie les modifications apportés par une branche sur un serveur
git push origin ma-branche

// Récupère les modifications sur le serveur d'hébergement
git pull origin ma-branche
```

2.3 Premier commit

Depuis la branche 'master' (branche par défaut), créer un fichier '**.gitignore**' qui permet de ne pas tracker des fichiers jugés inutiles ou potentiellement sources d'erreurs et de conflits (fichiers binaires compilés).

Ajouter le contenu suivant dans ce nouveau fichier :

```
.idea
target
*.class
```

Ajouter l'ensemble des modifications du projet puis créer votre premier commit.

2.4 Synchronisation avec Gitlab

Depuis Gitlab, créer un nouveau projet que vous nommerez `TP1-Agility`. Dans le terminal de votre projet, renseigner l'url de la remote comme étant l'origine :

```
git remote add origin git@gitlab.com:XXXXXX/tp1-agility.git
```

Vous pouvez maintenant soumettre vos modifications sur le serveur et vérifier qu'elles sont bien apparentes dans votre projet du serveur Gitlab.

3 Configuration du projet

Note : créer ici une branche de développement nommée '`feature/Configuration`'.

3.1 Installation de dépendances

Nous allons utiliser le framwork JUnit afin de tester le serveur d'API REST qui sera fourni. Ce serveur d'API REST permettra de restituer du contenu rapidement au client (navigateur web). Au sein de votre fichier de configuration **pom.xml** de votre projet Maven. Ajouter les dépendances et propriétés suivantes :

```
<project>
        ...
        <properties>
```

```
            <maven.compiler.source>1.8</maven.compiler.source>
            <maven.compiler.target>1.8</maven.compiler.target>
        </properties>
        <dependencies>
            <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.12</version>
                <scope>test</scope>
            </dependency>
        </dependencies>
</project>
```

3.2 Vérification du fonctionnement du serveur d'API

Afin de gagner en temps de développement, nous n'allons pas utiliser une base de données. Dans le cadre du TP, nous allons exploiter des données issues de l'API de la NASA, plus précisemment l'API nommée APOD pour 'Astronomy Picture of the Day'.

Télécharger le fichier nasa_apod.json composé de données extraites et pré-traitées de l'API de la NASA. Puis placer ce fichier à la racine de votre projet.

Nous allons maintenant vérifier le bon fonctionnement du serveur d'API. Télécharger le fichier **jar**, api.jar et le placer à la racine du projet.

Lancer le serveur d'API :

```
java -jar api.jar
```

Le serveur est accessible depuis http://127.0.0.1:4567. Il est composé de deux routes :

— **article/:date** : retourne un article en fournissant sa date (voir fichier nasa_apod.json).

— **news** : retourne une liste des 5 derniers articles disponibles.

Ajouter maintenant les modifications apportées en réalisant un nouveau commit. Fusionner la branche 'feature/Configuration' dans 'develop', supprimer la branche 'feature/Configuration' puis mettre à jour le serveur d'hébergement.

3.3 Tests unitaires

Note : créer ici une branche de développement nommée 'feature/UnitTests'.

Nous allons maintenant intégrer les tests unitaires au sein du projet. Pour cela, il vous faut ajouter les dépendances suivantes au sein de votre fichier **pom.xml** Celle du framework de Test unitaires JUnit et celle de la librairie Gson pour le traitement des réponses de l'API.

```
<project>
    ...

    <dependencies>
        ...
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
```

```
            </dependency>

        <dependency>
                    <groupId>com.google.code.gson</groupId>
                    <artifactId>gson</artifactId>
                    <version>2.7</version>
                    <scope>compile</scope>
            </dependency>
        </dependencies>
</project>
```

Mettre à jour le projet pour télécharger la nouvelle dépendance puis créer la classe ‘ServiceTest‘ dans le dossier **src/test/java/services**.

Dans la classe ‘ServiceTest‘ en vous basant sur la documentation de JUnit, écrire les tests unitaires suivants vérifiant :

— que la route ‘article/2020-06-03‘ retourne bien un JSON comprenant l'article dont le titre est ‘*The Dance of Venus and Earth*‘.
— que la route ‘news‘ retourne bien un JSON comprenant 5 articles.

Notes : la librairie Gson, permet de convertir un flux JSON en une une instance de classe. N'hésitez pas à créer une classe ‘Article‘ (dites POJO) stockant chacune des informations d'un article en ne gardant que les champs principaux (title, explanation, date, url).

Vous pouvez vous aider également de l'exemple ci-dessous pour réaliser un appel vers un service extérieur depuis votre test unitaire :

```java
// Exécution de la requête
URL url = new URL("http://localhost:4567/maRoute");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("GET");
connection.setDoOutput(true);
connection.connect();

// récupération de la réponse
String body = IOUtils.toString(connection.getInputStream());

// Traitement de la réponse
Gson gson = new Gson();
MaClasse object = gson.fromJson(body, MaClasse.class);

// Votre test
assertEquals(42, object.getId());
```

Une fois que le passage des tests unitaires est validé, n'hésitez pas à créer un simple commit au sein de la feature ‘feature/UnitTests‘ pour valider cette étape.

Note : il est important lors de l'exécution de la classe ‘ServiceTest‘ d'avoir exécuter au préalable le serveur d'API Spark.

3.4 Intégration continue

Nous allons maintenant intégrer la vérification du passage des tests unitaires à chaque nouvelle mise à jour soumise au serveur Gitlab sur certaines branches.

Pour que Gitlab puisse télécharger les dépendances du projet utiles au lancement des tests unitaires, il vous faut ajouter le répertoire de dépendances Maven ciblé dans le fichier **pom.xml** :

```
<project>
        ...

        <dependencies>
        ...
        </dependencies>

        <distributionManagement>
                <repository>
                        <id>central</id>
                        <name>83d43b5afeb5-releases</name>
                        <url>${env.MAVEN_REPO_URL}/libs-release-local</url>
                </repository>
        </distributionManagement>
</project>
```

Gitlab a également besoin de s'identifier sur le répertoire de dépendances distant. À cet effet, créer le dossier `.m2` et y ajouter le fichier '`settings.xml`' avec le contenu suivant :

```
<settings xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0
    http://maven.apache.org/xsd/settings-1.1.0.xsd" xmlns="http://maven.apache.org/SETTINGS/1.1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <servers>
                <server>
                        <id>central</id>
                        <username>${env.MAVEN_REPO_USER}</username>
                        <password>${env.MAVEN_REPO_PASS}</password>
                </server>
        </servers>
</settings>
```

Gitlab se base sur un fichier nommé '`.gitlab-ci.yml`' pour interpréter les opérations d'intégration continues souhaité. Dans l'exemple ci-dessous, qui sera utilisé pour le projet, nous définissons plusieurs choses :

- '`image`' : nous utilisons directement une image Docker propre à Maven (où Maven est déjà installé).
- '`stages`' les différents jobs que nous définissons dans notre pipeline d'intégration.
- '`variables`' : variables d'environnements utiles à l'image et ici propres à Maven.
- '`build`' : la phase de construction du projet correspondant au job (stage) 'build'.
- '`test`' : la phase de lancement des tests unitaires du projet correspondant au job (stage) 'test'. Avec ici les commandes permettant de lancer le serveur d'API puis d'attendre sa disponibilité avant de procéder aux tests unitaires.

```
image: maven:3-openjdk-8

stages:
      - build
      - test

variables:
      MAVEN_CLI_OPTS: "-s .m2/settings.xml --batch-mode"
      MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository"

cache:
```

```
      paths:
             - .m2/repository/
             - target/

build:
       stage: build
       script:
              - mvn $MAVEN_CLI_OPTS compile

test:
       stage: test
       script:
              - apt update
              - apt install -y netcat
          - java -jar api.jar &
          - echo "Waiting api to launch on 4567..."
          - while ! nc -z localhost 4567; do sleep 1; done
              - mvn $MAVEN_CLI_OPTS test
```

Ajouter le fichier à la racine de votre projet. Réaliser un commit avant de procéder la fusion de la branche de développement des tests unitaires (‘`feature/UnitTests`‘) dans la branche ‘`develop`‘. Puis mettre à jour la branche develop sur le serveur Gitlab.

Sur l’interface Gitlab de votre projet, il vous est possible d’accéder à l’onglet ‘CI/CD‘ relatif à l’intégration continue. Vous pouvez vérifier que les différents jobs de la pipeline ont été exécutés correctement.

Note : la configuration de la ‘CI/CD‘ est ici assez minimale. La documentation offre beaucoup de paramètres intéressants, notammment la possibilité d’associer un job à une ou plusieurs branches spécifiques. Cela permet notammennt de séparer les différents environnements de développements et déploiements.

4 Mesure de qualité du code

Dans cette partie du TP, nous allons procéder à la configuration de votre projet sur un serveur SonarQube, permettant de mesurer la qualité de code d’un projet mais aussi de mettre en place des règles de convention de codage relatives au projet.

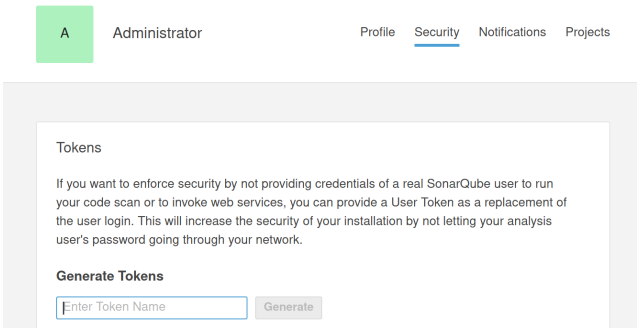
4.1 Configuration du projet

Un serveur sonarqube est disponible pour vous permettre de configurer votre projet et intéragir avec Gitlab. À noter que l’outil PMD, incluant des mesures supplémentaires, est déjà intégré dans le serveur. Un compte vous a été créé sur ce serveur dont l’identifiant est composé de la manière suivante : Jean DUPONT $=>$ jdupont.

Accéder aux paramètres de sécurité de votre compte et générer un token d’identification nommé ‘gitlab‘ :

Créer ensuite un projet avec les informations suivantes :
— ‘`project key`‘ : tp1-agility
— ‘`display name`‘ : tp1-agility

Il vous faut ajouter votre token précédemment généré et préciser que le projet est un projet **Java** et **Maven**. Vous aurez ainsi accès à la commande permettant de générer un rapport de qualité de code sur votre projet.



Dans notre cas, nous allons directement configurer un nouveau job à notre pipeline CI/CD de Gitlab (avec les variables SONAR associées), comme présenté ci-dessous :

```
...
stages:
        - build
        - test
        - sonarqube
...
variables:
        ...
        SONAR_TOKEN: "your-sonarqube-token"
        SONAR_HOST_URL: "http://51.254.210.97:9000"
...
sonarqube:
        stage: sonarqube
        script:
                - mvn verify sonar:sonar -Dsonar.qualitygate.wait=true
        allow_failure: true
        only:
                - merge_requests
                - develop
                - master
```

Analyser le rapport généré puis explorer l'outil et les règles utilisées pour votre projet.

Enfin, réaliser une livraison du projet sur la branche '`master`' en récupérant l'ensemble des modifications de la branche '`develop`' **sans la supprimer**. Vérifier que la pipeline Gitlab a bien été exécutée et consulter le rapport Sonar.

5 Développement côté client

Note : créer ici une branche de développement nommée '`feature/DevWeb`'.

Vous allez maintenant pouvoir exploiter votre serveur Spark afin de générer une visualisation du contenu des articles depuis un navigateur web. Pour cela, vous allez créer un dossier **`web`** à la racine de votre projet qui contiendra l'ensemble des fichiers relatives au développement front.

Vous êtes ici libre de réaliser le développement avec le framework de votre choix (côté Javascript notamment). Pour ceux partant plutôt sur un développement rapide, n'hésitez pas à utiliser bootstrap pour un visuel rapide.

Une fois le développement terminé, réaliser une livraison du projet dans son intégralité. N'hésitez pas à documenter également ce projet au travers d'un fichier **README.md**.