amU Faculté des sciences Aix Marseille Université

Année 2024-25

Domaine Sciences et Technologies
Master informatique
Génie logiciel avancé : TP 5
Codes UE : SINB33DL
Gestion d'emploi du temps (5/6)

1 Refonte calendrier (suite)

Le but de cette séance est de continuer à prendre en compte les fonctionnalités décrites dans le TP 4 et de sélectionner un nombre raisonnable de fonctionnalités pas encore prises en compte afin de les implémenter pour la semaine suivante. Avant de se lancer directement dans le code, on vous demande de réfléchir aux modifications à faire en termes d'architecture sur l'application (pour le diagramme de classes du client et du serveur ainsi que pour le schéma de la base de donnée). Vous devez donc créer des diagrammes (par exemple à l'aide de plantuml) afin de décrire les changements prévus sur l'application sur le wiki de votre *group* (accessible via le menu `Plan` -> `Wiki` d'etulab).

2 Injection de dépendances

Une première amélioration de code est d'utiliser un outil d'injection de dépendance comme Guice. Pour utiliser *Guice*, il vous faut ajouter les lignes suivantes dans le fichier `build.gradle.kts` :

```
dependencies {
    implementation("com.google.inject:guice:7.0.0")
}
```

Les transparents du quatrième cours donne l'explication pour utiliser guice.

3 Tests

3.1 Test côté client

Afin de tester la manière dont votre client interagit avec le serveur, vous pouvez utiliser wiremock qui permet de créer un *mock* de serveur HTTP. Pour utiliser ce framework, il vous faut rajouter les lignes suivantes dans le fichier `build.gradle.kts` de votre projet :

```
dependencies {
    testImplementation("org.wiremock:wiremock:3.10.0")
    implementation("org.slf4j:slf4j-simple:2.0.16")
}
```

Vous trouverez un exemple de test dans le fichier WireMockTest.java du dépôt GLA tests example.

3.2 Couverture de code par les tests

Il est utile de vérifier la couverture de code avec un outil externe à votre IDE comme *IntelliJ IDEA*. Vous pouvez faire cela grâce à Jacoco. Pour utiliser *Jacoco* via gradle, il faut :

1. Rajouter les plugins pour jacoco au début du fichier `build.gradle.kts` de votre projet avec les lignes suivantes :

```
plugins {
    id("jacoco")
    id("org.barfuin.gradle.jacocolog") version "3.1.0"
}
```

2. Configurer votre projet pour que la couverture soit automatiquement lancé après les tests en rajoutant les lignes suivantes au fichier `build.gradle.kts` :

```
tasks.test {
    finalizedBy(tasks.jacocoTestReport) // report is always generated after tests run
}
```

3. Configurer votre projet pour que la couverture génère un fichier *xml* (`build/reports/jacoco/test/jacocoTestReport.xml`) et *html* (`build/reports/jacoco/test/html/index.html`) :

```
tasks.jacocoTestReport {
    dependsOn("test")
    reports {
        xml.required = true
        html.required = true
    }
}
```

4. Ajouter un fichier .gitlab-ci.yml à la racine de votre projet pour que les tests et la couverture soient lancés à chaque *push*.

4 Outils d'analyse de code

Sur un projet logiciel, il est important d'analyser statiquement le code afin de

— déceler des bugs ou des failles potentielles ;
— vérifier que le code respecte des bonnes pratiques.

On vous demande donc d'utiliser différents outils d'analyse de code afin d'améliorer votre code et

4.1 SpotBugs

SpotBugs est un analyseur statique de programmes *Java*. Il s'agit d'un *fork* du projet *FindBugs*, qui n'est plus aujourd'hui maintenu. Ils sont l'un comme l'autre libres de droit. *SpotBugs* comme de nombreux analyseurs statiques définit et utilise des motifs de bug et réalise donc une analyse approchée. Il est donc possible que certains bugs signalés n'en soit pas vraiment. Il s'agit de "faux positifs". Il est donc possible que, malgré une écriture soignée du code, certains bugs signalés ne puissent pas être éliminés. *SpotBugs* travaille sur le *bytecode* Java et fonctionne donc qu'une fois le code compilé.

Pour lancer *spotbugs* via gradle, il faut :

1. Rajouter le plugin au début du fichier `build.gradle.kts` de votre projet avec les lignes suivantes :

```
plugins {
    id("com.github.spotbugs") version "6.1.3"
}
```

2. Spécifier la version utilisée du logiciel en rajoutant les lignes suivantes dans le fichier `build.gradle.kts`

```
spotbugs {
    toolVersion = "4.9.0"
}
```

3. Lancer la tâche *gradle* `spotbugsMain` (dans `verification`) pour le code dans `src/main` et `spotbugsTest` pour le code dans `src/test`

4.2 PMD

L'analyseur PMD est un autre outil permettant l'analyse de code Java. Contrairement à *SpotBugs*, PMD travaille sur le code source Java du programme et est un outil fortement configurable puisque les règles servant à l'analyse sont définies extérieurement au logiciel dans un fichier *xml*.

Pour lancer *PMD* via gradle, il faut :

1. Rajouter le plugin au début du fichier `build.gradle.kts` de votre projet avec les lignes suivantes :

```
plugins {
    id("pmd")
}
```

2. Spécifier la version utilisée du logiciel et les règles utilisées en rajoutant les lignes suivantes dans le fichier `build.gradle.kts`

```
pmd {
    isConsoleOutput = true
    toolVersion = "7.0.0-rc1"
    rulesMinimumPriority = 5
    ruleSets = listOf("rulesets/java/quickstart.xml", "category/java/errorprone.xml",
↪   "category/java/bestpractices.xml")
}
```

3. Lancer la tâche *gradle* `pmdMain` (dans `other`) pour le code dans `src/main` et `pmdTest` pour le code dans `src/test`

4.3 Sonarlint

SonarLint est un autre outil d'analyse de code qui permet de détecter des erreurs éventuelles dans votre code.

1. Rajouter le plugin au début du fichier `build.gradle.kts` de votre projet avec les lignes suivantes :

```
plugins {
    id ("name.remal.sonarlint") version "5.1.1"
}
```

2. Lancer la tâche *gradle* `sonarlintMain` (dans `verification`) pour le code dans `src/main` et `sonarlintTest` pour le code dans `src/test`