Analyse

Afin de garantir la qualité de l'application, nous avons décidé de mettre en place une batterie de tests unitaires et end-to-end (e2e). Nous avons ensuite utilisé JaCoCo et Pitest, qui sont tous les deux des outils de Code Coverage, afin de générer des rapports de tests. Bien que les deux génèrent des rapports de coverage, nous avons décidé de les utiliser tous les deux car nous les considérons comme étant complémentaires.

Code Coverage : JaCoCo

Notre premier objectif était d'atteindre une couverture de test du code de minimum 80% afin de nous assurer d'avoir une application stable, et nous avons pour cela installé JaCoCo. JaCoCo est un outil de mesure de la couverture de code pour les applications Java permettant de suivre la quantité de code source qui a été testée par des tests unitaires, ainsi que de générer des rapports détaillant la couverture de code. Il peut également être intégré à des outils de build tels que Maven ou Gradle, ce qui nous a permis ici de créer une GitHub Action afin de l'exécuter avec Maven lors de l'intégration continue. Une fois l'action exécutée, une autre GitHub Action génère ensuite une GitHub Page déployée à cette adresse.

Qualité des tests : Pitest

Notre deuxième objectif était d'effectuer des tests de mutation afin de garantir la qualité de nos tests, en nous fixant un objectif de 80% de mutations éliminées sur les contrôleurs et les services. Pour rappel, le test de mutation est une technique qui consiste à introduire délibérément des défauts (ou mutations) dans le code source, puis à exécuter les tests unitaires, et dont l'objectif est de vérifier si les tests détectent les mutations. Nous avons donc pour cela installé Pitest, qui est un outil de test de mutation pour les applications Java. Cet outil analyse le code source, introduit des mutations, puis exécute les tests unitaires pour voir si les mutations sont détectées. Il génère ensuite des rapports indiquant quelles mutations ont été tuées par les tests et lesquelles ont survécu. Comme JaCoCo, il peut être intégré à Maven, ce qui nous a permis de créer une autre GitHub Action afin de l'exécuter lors de l'intégration continue. Une fois cette action exécutée, l'action de génération de GitHub Page en déploie une seconde à cette adresse.

Tests

Tests unitaires

Afin de mettre en place nos tests unitaires, nous avons utilisé JUnit et Mockito. Il sont inclus dans notre projet grâce à l'usage du starter POM spring-boot-starter-test, qui inclut également Hamcrest dont nous ne nous sommes pas servis. Pour programmer nos tests, nous avons utilisé la méthodologie suivante : nous avons créé une multitude de tests de précision très fine afin de tester chaque cas possible pour chaque méthode de nos services et contrôleurs, et nous avons nommé ces tests en suivant la syntaxe Gherkin. Nous avons ainsi recréé une arborescence de fichiers de tests calquée sur l'arborescence de notre projet, en créant ainsi un fichier de tests par service et par contrôleur. Dans chacun de ces fichiers de tests, nous utilisons Mockito afin de mocker les dépendances du service ou du contrôleur testé. Ainsi, à chaque début de test, nous utilisons Mockito afin de simuler le comportement attendu par nos dépendances, puis nous testons la méthode souhaitée et nous vérifions ensuite si le résultat de celle-ci est bien le résultat attendu, qu'il soit une valeur ou un comportement particulier (comme le lancement d'exception). Enfin, nous vérifions si le nombre d'appels aux méthodes de

nos dépendances utilisées par notre méthode est le bon. Comme précisé précédemment, nos tests sont ensuite exécutés par Maven grâce à notre pipeline d'intégration, et nous nous sommes fixés des objectifs de 80% de couverture de code et de mutations éliminées (pour les contrôleurs et les services) en définissant des thresholds sur JaCoCo et Pitest.

Tests d'intégration

PAS DE TESTS D'INTEGRATION DANS LE PROJET : voir ici

Tests E2E

Pour les tests, nous nous sommes servis du package RestAssured, qui permet d'effectuer des requêtes REST dans un fonctionnement similaire à du test unitaire. On ne mock donc rien et on effectue des tests sur tout le routing du code.

Pour le développement, nous avons d'abord codé toutes les requêtes dans la manière proposée par RestAssured :

```
        return given()
                .header("accept", "*/*")
                .header("Content-type", "application/json")
                .body({"Json" : "xxx")
                .when()
                .post(/api/x/x)
                .then()
                .extract().response();
    }
```

Nous estimions que malgré le fait que nous ayons du code dupliqué sur beaucoup de tests, il était plus clair pour le lecteur de comprendre ce que cela pouvait représenter. Finalement, devant la longueur gigantesque de code que cette méthode imposait, nous avons finalement opté pour le refactoring du code dans des méthodes spécifiques aux POST / GET / DELETE et PUT. Nous avons également généré des méthodes pour générer les JSON afin de rendre le code plus propre et plus compact.

```
// Exemple Méthode POST
    public static Response dbPOST(String path, String requestBody ) {
        return given()
                .header("accept", "*/*")
                .header("Content-type", "application/json")
                .body(requestBody)
                .when()
                .post(path)
                .then()
                .extract().response();
    }

// Exemple génération JSON pour un POST depuis un Participant
public static String generateParticipantPOSTBody(Participant participant) {
```

```
        return "{\n" +
                "\"participantId\" : " + participant.getParticipantId() + ",\n" +
                "\"nom\" : \"" + participant.getNom() + "\",\n" +
                "\"prenom\" : \"" + participant.getPrenom() + "\"\n" +
                "}";
    }
// Exemple de Test sur un POST DateSondage
        Date fixedDate = new Date(futureDate.getTime()+300000);
        requestBody = DateSondageSampleE2E.generateDateSondagePOSTBody(fixedDate);
        response = CrudRestAssured.dbPOST("/api/datesondage/"+createdSondageID,
requestBody);
        assertEquals(201, response.statusCode());
```

Grâce à ce refactoring, les classes de test E2E ont pu diviser leurs lignes de codes par 5 et être beaucoup plus agréables à la lecture.

Rapports de tests final :

```
[INFO] Tests run: 31, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.542 s --
in
fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.unit.controllers.SondageControllerUnitTest
[INFO] Running
fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.units.services.CommentaireServiceUnitTest
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.525 s --
in
fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.units.services.CommentaireServiceUnitTest
[INFO] Running
fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.units.services.DateSondageServiceUnitTest
[INFO] Tests run: 11, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.175 s --
in
fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.units.services.DateSondageServiceUnitTest
[INFO] Running
fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.units.services.DateSondeeServiceUnitTest
[INFO] Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.163 s --
in
fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.units.services.DateSondeeServiceUnitTest
[INFO] Running
fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.units.services.ParticipantServiceUnitTest
[INFO] Tests run: 11, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.227 s --
in
fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.units.services.ParticipantServiceUnitTest
[INFO] Running
fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.units.services.SondageServiceUnitTest
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.280 s --
in fr.univ.lorraine.ufr.mim.m2.gi.mysurvey.units.services.SondageServiceUnitTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 150, Failures: 0, Errors: 0, Skipped: 0
[INFO]
```

```
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.11:report (generate-code-coverage-report) @
MySurvey ---
[INFO] Loading execution data file
D:\Cours\M2\Production_Logicielle\Projet\SondageSpringBoot\target\jacoco.exec
[INFO] Analyzed bundle 'MySurvey' with 25 classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  19.113 s
[INFO] Finished at: 2024-01-19T18:33:15+01:00
[INFO] ------------------------------------------------------------------------

Process finished with exit code 0
```