Exercice Spring Test

Il est demandé d'utiliser Spring Boot, JUnit 5 et Spring-test. La différence entre les tests unitaires et les tests d'intégration dans Spring Boot réside dans les annotations @WebMvcTest et @SpringBootTest.

L'annotation @SpringBootTest indique à Spring Boot d'aller chercher une classe de configuration principale (une avec @SpringBootApplication par exemple), et de l'utiliser pour démarrer un contexte d'application Spring. SpringBootTest charge l'application complète et injecte tous les beans qui peuvent être lents. @WebMvcTest - pour tester la couche de contrôleur et vous devez fournir les dépendances restantes requises à l'aide d'objets simulés.

Exercice1

Spring dispose de son propre framework de test pour tous les objets créés pour le test. Il permet aussi de faire des tests paramétrés et de sortir les données de test des classes de test.

Dans ce projet il est demandé de s'intéresser à un classe métier Rational muni de ses 4 opérations et d'en effectuer les tests avec des jeux de données externalisées dans des fichiers CSV.

Remarque: le CSV utilisé sera la version simple sans structure imbriqué.

Faire un projet créer un projet *exercice8-spring* contenant un package *metier* contenant:

- une classe *Rational*
- une interface *Operator* générique sur les 4 opérations: add, sub, mult, div
- une classe *RationalOperator* implémentant l'interface précédente.

1- Dans le répertoire test, il est demandé de construire une 1ere classe de test pour la classe *RationalOperator*, 2- Dans le même répertoire, il est demandé de construire une 2ème classe de test pour l'externatisation des données dans un fichier CSV. Il est important de prendre soin de la conception des données de test, 3- Modifier le fichier build.gradle afin d'ajouter un plugin pour le calcul du graphe de couverture des tests dans un rapport. Le plus simple est jacoco et le plus complexe sonarcube.

```
apply plugin: 'spring-boot'
apply plugin: "jacoco"

sourceCompatibility = 1.8
targetCompatibility = 1.8

jacocoTestReport {
    group = "Reporting"
    reports {
        xml.enabled true
        csv.enabled false
        html.destination "${buildDir}/reports/coverage"
    }
}
```

Après l'analyse du rapport obtenu, modifier les données de test de la question précédente afin d'améliorer la couverture.

Exercice2

Soit un projet déjà développé nommé *exercice9-spring* qui comprends un ensemble de code pour stocker des départemnts et des catégories dans une base Postgresql depuis une application Web via des service HTTP.

Spring boot a une configuration de projet hiérarchique où la classe bootstrap se trouve à la base d'autres classes qui peuvent être séparées en contrôleurs, modèles, exceptions, repository, packages de services et autres selon les besoins.

1- Dans cette partie, nous nous intéressons au test unitaire. Il est demandé de construire un test unitaire pour la couche controleur. Plus particulièrement:

- l'opération *getDepartment*
- l'opération *getAllDepartments*

Pour ce faire il est important de créer une classe de test *UnitApplicationTests* annotée @WebMvcTest qui utilise 2 mocks:

- un MockMvc injecté par Spring
- un Mock du service DepartmentService construit par Mockito

Afin de construire des assertions, il est important de se référer à la document de Mockito: https://site.mockito.org/

2- Dans cette partie, nous nous intéressons au test d'intégration. Ainsi il est demandé de couvrir une fonctionnalité complète de l'application. Il faut prendre soin:

- la classe principale de test est annotée @SpringBootTest
- de ne pas créer de classe à chaque test : @TestInstance
- d'initialiser le WebApplicationContext avant le lancement global du test

Exercice3

Dans cet exercice, nous souhaitons mettre en oeuvre un test bout en bout appliqué à un batch Spring. La dépendance spring-batch-test fournit un ensemble de méthodes d'assistance et d'écouteurs utiles qui peuvent être utilisés pour configurer le contexte Spring Batch pendant les tests.

Créons une structure de base pour notre test de notre projet *exercice8-batch*:

```
@RunWith(SpringRunner.class)
@SpringBatchTest
@EnableAutoConfiguration
@ContextConfiguration(classes = { BatchConfiguration.class })
@TestExecutionListeners({ DependencyInjectionTestExecutionListener.class,
  DirtiesContextTestExecutionListener.class})
@DirtiesContext(classMode = ClassMode.AFTER_CLASS)
public class SpringBatchIntegrationTest {
```

```
    // other test constants

    @Autowired
    private JobLauncherTestUtils jobLauncherTestUtils;

    @Autowired
    private JobRepositoryTestUtils jobRepositoryTestUtils;

    @After
    public void cleanUp() {
        jobRepositoryTestUtils.removeJobExecutions();
    }

    private JobParameters defaultJobParameters() {
        JobParametersBuilder paramsBuilder = new JobParametersBuilder();
        paramsBuilder.addString("data.input", TEST_INPUT);
        paramsBuilder.addString("table.output", TEST_OUTPUT);
        return paramsBuilder.toJobParameters();
    }
```

L'annotation @SpringBatchTest fournit les classes d'assistance JobLauncherTestUtils et JobRepositoryTestUtils. Nous les utilisons pour déclencher le Job et les Steps dans nos tests.

Notre application utilise la configuration automatique de Spring Boot, qui active un JobRepository en mémoire par défaut. Par conséquent, l'exécution de plusieurs tests dans la même classe nécessite une étape de nettoyage après chaque exécution de test (méthode *cleanup*).

Enfin, si nous voulons exécuter plusieurs tests à partir de plusieurs classes de test, nous devons marquer notre contexte comme "dirty". Cela est nécessaire pour éviter le conflit entre plusieurs instances de JobRepository utilisant la même source de données.

La première chose que nous allons tester est un Job de bout en bout complet avec une petite entrée de jeu de données. Nous pouvons ensuite comparer les résultats avec une sortie de test attendue: