Testing in Spring Boot 2

1. Tests unitaires vs tests d'intégration

Généralement, toute application logicielle est divisée en différents modules et composants. Lorsqu'un de ces composants est testé isolément, on parle de test unitaire. Il est écrit pour vérifier qu'un morceau de code relativement petit fait ce qu'il est censé faire.

Les tests unitaires ne vérifient pas si le code de l'application fonctionne correctement avec les dépendances externes. Il se concentre sur un seul composant et se moque de toutes les dépendances avec lesquelles ce composant interagit.

Une fois que différents modules sont développés et intégrés, des tests d'intégration sont effectués. Son objectif principal est de découvrir les problèmes lorsque différents modules interagissent les uns avec les autres pour traiter les demandes des utilisateurs de bout en bout.

Les tests d'intégration peuvent inclure l'ensemble de l'application ou seulement certains composants, en fonction de ce qui est testé. Ils peuvent avoir besoin que des ressources telles que des instances de base de données et du matériel leur soient allouées. Bien que ces interactions puissent également être simulées pour améliorer les performances du test.

En termes d' application Spring boot crud typique , les tests unitaires peuvent être écrits pour tester séparément les contrôleurs REST, la couche DAO, etc. Il ne nécessitera même pas le serveur intégré .

Dans les tests d'intégration, nous nous concentrerons sur le test du traitement complet des requêtes du contrôleur à la couche de persistance. L'application doit s'exécuter à l'intérieur du serveur intégré pour créer un contexte d'application et tous les beans. Certains de ces beans peuvent être remplacés pour se moquer de certains comportements.

2. Dépendances

2.1. Tests Junit 4 (par défaut)

Pour écrire des tests dans les applications de Spring boot , le meilleur moyen est d'inclure spring-boot-starter-test dans `pom.xml`le fichier. Il apporte les dépendances Junit 4 , AssertJ, Hamcrest, Mockito , JSONassert et JsonPath dans l'application avec une portée de test.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

2.2. Tests Junit 5

Spring Boot prend également en charge les tests Junit 5 . Pour utiliser Junit 5, incluez sa dépendance et excluez Junit 4 de *spring-boot-starter-test* .

Une dépendance de base de données intégrée est pratique lors de l'écriture de tests d'intégration.

```
dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>

    <!-- exclude junit 4 -->
    <exclusions>
        <exclusion>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </exclusion>
    </exclusions>

</dependency>

<!-- junit 5 -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
    <version>1.4.194</version>
</dependency>
```

3. Testeurs

Les tests écrits au spring peuvent être exécutés de différentes manières. Voyons quelques façons les plus courantes.

3.1. @RunWith(SpringRunner.class) – [Junit 4]

Par défaut, les tests écrits sont en Junit 4. Pour exécuter de tels tests, nous pouvons utiliser la classe SpringRunner (étend *SpringJUnit4ClassRunner*) avec une `@RunWith`annotation au niveau de la classe.

```
@RunWith(SpringRunner.class)
@WebFluxTest(controllers = EmployeeController.class)
public class EmployeeRestControllerTest {
  //tests
}
```

3.2. @RunWith(MockitoJUnitRunner.class) – [Junit 4 avec Mockito]

Il teste l'utilisation `@Mock`des objets préférés à l'aide *de MockitoJUnitRunner* . Il initialise les simulations annotées avec Mock, de sorte que l'utilisation explicite de `MockitoAnnotations.initMocks(Object)`n'est pas nécessaire. Les simulations sont initialisées avant chaque méthode de test.

```
@RunWith(MockitoJUnitRunner.class)
public class EmployeeRestControllerTest
{
  @Mock
    private Repository repository;
}
```

3.3. @ExtendWith(SpringExtension.class) – [Junit 5]

SpringExtension intègre le Spring TestContext Framework dans le modèle de programmation Jupiter de JUnit 5.

```
//@ExtendWith(SpringExtension.class)  // included in @WebFluxTest
@WebFluxTest(controllers = EmployeeController.class)
@Import(EmployeeService.class)
public class EmployeeControllerTest
{
  //
}
```

3.4. @ExtendWith(MockitoExtension.class) – [Junit 5]

MockitoExtension initialise les simulations et gère les stubbings stricts. C'est l'équivalent du`MockitoJUnitRunner`.

La plupart des annotations de test incluent cette annotation avec elles, il n'est donc pas nécessaire de l'inclure explicitement.

```
@ExtendWith(MockitoExtension.class)
public class EmployeeControllerTest
{
  //
}
```

4. Spring boot *Annotations de test

Spring Boot fournit diverses annotations pour activer l'infrastructure de test liée uniquement à certaines parties de l'application. Il fournit également des annotations qui facilitent également les tests d'intégration. Allons leur rendre visite.

4.1. @SpringBootTest

Cette annotation aide à écrire des tests d'intégration . Il démarre le serveur embarqué et initialise complètement le contexte de l'application. Nous pouvons injecter les dépendances dans la classe de test à l'aide d' `@Autowired`annotations.

Nous pouvons également fournir une configuration de beans spécifique au test à l'aide de la classe @Configuration imbriquée ou de classes @TestConfiguration explicites .

Il prend également en charge différents modes d'environnement Web et l'exécution d'un serveur Web à l'écoute sur un port défini ou aléatoire. Il enregistre également un bean TestRestTemplate et/ou WebTestClient à utiliser dans les tests Web

```
@SpringBootTest(classes = SpringBootDemoApplication.class,
    webEnvironment = WebEnvironment.RANDOM_PORT)
public class EmployeeControllerIntegrationTests
{
  @LocalServerPort
  private int port;

  @Autowired
  private TestRestTemplate restTemplate;

  //tests
}
```

4.2. @WebMvcTest

Cette annotation est utilisée pour les tests Spring MVC . Il désactive la configuration automatique complète et applique à la place uniquement la configuration pertinente pour les tests MVC.

Il configure également automatiquement l'instance MockMvc . Nous ne pouvons initialiser qu'un seul contrôleur Web en `.class` le passant comme attribut d'annotation.

```
@WebMvcTest(EmployeeRESTController.class)
public class TestEmployeeRESTController {

    @Autowired
    private MockMvc mvc;

  //
}
```

4.3. @WebFluxTest

Cette annotation désactive la configuration automatique complète et applique à la place uniquement la configuration pertinente pour les tests WebFlux . Par défaut, les tests annotés avec *@WebFluxTest* configurent également automatiquement un WebTestClient .

En règle générale , *@WebFluxTest* est utilisé en combinaison avec @MockBean ou `@Import`pour créer les collaborateurs requis par le bean contrôleur.

```
@WebFluxTest(controllers = EmployeeController.class)
@Import(EmployeeService.class)
public class EmployeeControllerTest
{
  @MockBean
  EmployeeRepository repository;

  @Autowired
  private WebTestClient webClient;

  //tests
}
```

4.4. Autres annotations fréquemment utilisées

- @JdbcTest - peut être utilisé pour un test jdbc typique lorsqu'un test se concentre uniquement sur les composants basés sur jdbc. Il désactive la configuration automatique complète et applique à la place uniquement la configuration pertinente pour les tests jdbc.
  Par défaut, les tests annotés avec @JdbcTest sont transactionnels et annulent à la fin de chaque test. L'annotation configure une base de données intégrée en mémoire et JdbcTemplate .
- @JooqTest - Il peut être utilisé lorsqu'un test se concentre uniquement sur les composants basés sur jOOQ. Attention, par défaut, les tests annotés avec @JooqTest utilisent la base de données configurée par l'application. Pour utiliser la base de données intégrée en mémoire, l'annotation @AutoConfigureTestDatabase peut être utilisée pour remplacer ces paramètres.
- @JsonTest - Il est utilisé lorsqu'un test se concentre uniquement sur la sérialisation JSON. Il initialise les champs , `@JsonComponent`et `JacksonTester`.`JsonbTesterGsonTester`
- @DataJpaTest - Il peut être utilisé pour tester les applications JPA. Par défaut, il recherche les classes @Entity et configure les référentiels Spring Data JPA. Si une base de données intégrée est disponible sur le classpath, il en configure une également.
  Par défaut, les tests JPA de données sont transactionnels et annulent à la fin de chaque test.

Les tests JPA de données peuvent également injecter un bean TestEntityManager , qui fournit une alternative au JPA standard `EntityManager` spécialement conçu pour les tests.

- @DataMongoTest - est utilisé pour tester les applications MongoDB. Par défaut, il configure un MongoDB intégré en mémoire (si disponible), configure un MongoTemplate , analyse les `@Document`classes et configure les référentiels Spring Data MongoDB.
- @DataRedisTest - est utilisé pour tester les applications Redis. Par défaut, il recherche les classes @RedisHash et configure les référentiels Spring Data Redis.
- @DataLdapTest – est utilisé pour tester les applications LDAP. Par défaut, il configure un LDAP intégré en mémoire (si disponible), configure un LdapTemplate , recherche les classes @Entry et configure les référentiels Spring Data LDAP.
- @RestClientTest - est utilisé pour tester les clients REST. Par défaut, il configure automatiquement la prise en charge de Jackson, GSON et Jsonb, configure un`RestTemplateBuilder` et ajoute la prise en charge de`MockRestServiceServer`.

5. Tester la configuration

@TestConfiguration est une forme spécialisée `@Configuration`qui peut être utilisée pour définir des beans supplémentaires ou des personnalisations pour un test.

Au démarrage du spring, tous les beans configurés dans une classe de niveau supérieur annotée avec `@TestConfiguration`ne seront pas récupérés via l'analyse des composants. Nous devons enregistrer explicitement la classe @TestConfiguration avec la classe qui contient les cas de test.

La meilleure chose est que ces configurations de test ne font pas automatiquement partie de la configuration principale de l'application. Ils sont disponibles uniquement à la demande en utilisant l'une des deux méthodes ci-dessous pour inclure cette configuration de test supplémentaire, c'est-à-dire

5.1. @Importer une annotation

Il peut être utilisé pour importer une ou plusieurs classes de configuration dans un contexte d'application ou un contexte de test de spring.

```
@Import(MyTestConfiguration.class)
@SpringBootTest(webEnvironment =
WebEnvironment.RANDOM_PORT)
public class SpringBootDemoApplicationTests
{
    @LocalServerPort
    int randomServerPort;

    @Autowired
    DataSource datasource;

    //tests
}
```

5.2. Classes imbriquées statiques

Nous pouvons définir les configurations de test dans des classes imbriquées à l'intérieur de la classe de test. La classe imbriquée peut être annotée avec des annotations *@Configuration* ou *@TestConfiguration* .

- Dans le cas d'une classe imbriquée `@Configuration`, la configuration donnée serait utilisée « à la place » de la configuration principale de l'application.
- Une classe imbriquée `@TestConfiguration`est utilisée « en plus » de la configuration principale de l'application.

6. Mock

Spring boot a un excellent support pour se moquer des dépendances avec ou sans utiliser Mockito.

6.1. Avec Mockito – @Mock

@Mock est utilisé pour la création de maquettes. Cela rend la classe de test plus lisible. En classe de test, pour traiter les annotations mockito, `MockitoAnnotations.initMocks(testClass)` doit être utilisé au moins une fois.

Veuillez noter que si vous utilisez `RunWith(MockitoJUnitRunner.class)`, l'utilisation explicite de *MockitoAnnotations.initMocks()* n'est pas nécessaire. Les simulations sont initialisées avant chaque méthode de test.

À utiliser `@Mock`dans les tests unitaires où le contexte du texte de spring n'est pas nécessaire.

6.2. Sans Mockito – @MockBean

Annotation @MockBean utilisée pour ajouter des simulations à un Spring ApplicationContext. Il permet de se moquer d'une classe ou d'une interface et d'enregistrer et de vérifier les comportements sur celle-ci.

Fait intéressant, tout bean existant du même type défini dans le contexte sera remplacé par le mock. Si aucun bean existant n'est défini, un nouveau sera ajouté.

`@MockBean`est similaire à mockito `@Mock`mais avec le support de Spring. Nous utiliserons généralement `@MockBean`avec soit `@WebMvcTest`ou `@WebFluxTest`des annotations. Ces annotations sont pour une tranche de test Web et limitées à un seul contrôleur.

Dans l'exemple donné, nous nous moquons du `EmployeeRepository`Bean. De cette manière, tout le code de l'application sera invoqué mais toutes les interactions du référentiel seront simulées.

```
@WebFluxTest(controllers = EmployeeController.class)

@Import(EmployeeService.class)

public class EmployeeControllerTest

{

    @MockBean

    EmployeeRepository repository;



    @Autowired

    private WebTestClient webClient;



    //tests

}
```

7. Conclusion

Spring Boot fournit un excellent support pour les tests unitaires et les tests d'intégration des applications et de ses différents modules. Nous utiliserons le support fourni par l'utilisation d'annotations - très soigneusement.

Utilisez `@SpringBootTest` les annotations pour les tests d'intégration tandis que d'autres annotations de configuration automatique pour les tests unitaires de composants spécifiques.

Se moquer de certains comportements est une exigence très courante et nous pouvons utiliser l'annotation mockito `@Mock` ou Spring `@MockBean` à cette fin.