Spring Boot UT and IT

UT

1. Comment écrire un test unitaire correct pour les contrôleurs de repos

Lors de l'écriture de test junit pour une méthode de contrôleur de repos, nous garderons à l'esprit que :

- Un test unitaire est censé tester seulement une certaine partie du code (c'est-à-dire du code écrit dans la classe contrôleur), nous allons donc simuler toutes les dépendances injectées et utilisées dans la classe contrôleur.
- Si le test utilise d'autres dépendances (par exemple, base de données/réseau), il s'agit d'un test d'intégration et non d'un test unitaire.
- Nous ne devons pas utiliser de serveur Web , sinon cela ralentira les tests unitaires.
- Chaque test unitaire doit être indépendant des autres tests.
- Par définition, les tests unitaires doivent être rapides.

2. Contrôleurs de test unitaire utilisant Junit 5 et Mockito

2.1. Dépendances Maven

Commencez par inclure les dépendances requises. Nous utilisons ici l'application de Spring-boot.

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

<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>org.junit.platform</groupId>
    <artifactId>junit-platform-launcher</artifactId>
    <scope>test</scope>
</dependency>

<!-- Mockito extention -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <scope>test</scope>
</dependency>
```

2.2. Contrôleur REST

Voici le *contrôleur Spring Boot Rest* , pour lequel nous allons écrire des tests unitaires.

- Le contrôleur dépend de la classe *EmployeeDAO* pour la persistance.
- *L'api addEmployee() a* besoin d'accéder au contexte de la requête à l'aide de `ServletUriComponentsBuilder`.
- *addEmployee()* api renvoie le statut HTTP et l'en-tête à l'aide `ResponseEntity` de la classe.

```
@RestController
@RequestMapping(path = "/employees")
public class EmployeeController
{
    @Autowired
    private EmployeeDAO employeeDao;

    @GetMapping(path="/", produces = "application/json")
    public Employees getEmployees()
    {
        return employeeDao.getAllEmployees();
    }

    @PostMapping(path= "/", consumes = "application/json", produces =
"application/json")
    public ResponseEntity<Object> addEmployee(@RequestBody Employee
employee) {

        //add resource
        employeeDao.addEmployee(employee);

        //Create resource location
        URI location = ServletUriComponentsBuilder.fromCurrentRequest()
                                    .path("/{id}")
                                    .buildAndExpand(employee.getId())
                                    .toUri();

        //Send location in response
        return ResponseEntity.created(location).build();
    }
}
```

2.3. Tests unitaires

La classe de test ci-dessous contient des tests unitaires pour le contrôleur Spring Boot Rest mentionné ci-dessus. Cette classe de test :

- utilise l'annotation @Mock pour créer un objet fictif pour la dépendance *EmployeeDAO* .
- utilise `@InjectMocks`pour créer la classe *EmployeeController* et également injecter l' instance simulée de *employeeDAO* .
- MockitoExtension initialise les simulations et gère les stubbings stricts. Cette extension est l'équivalent JUnit Jupiter de notre JUnit4 *MockitoJUnitRunner* .
- L'utilisation de JUnitPlatform est facultative. Il permet aux tests junit 5 d'être exécutés avec des IDE et de créer des systèmes qui prennent en charge JUnit 4 mais ne prennent pas encore directement en charge la plate-forme JUnit.
- MockHttpServletRequest et RequestContextHolder fournissent le contexte de la requête là où le code sous test en a besoin.
- Utilisez les API org.mockito.Mockito.when() et thenReturn() pour simuler le comportement souhaité.
- Enfin, utilisez les assertions junit 5 pour affirmer les résultats du test avec les résultats attendus.

IT

1. Que tester dans les tests d'intégration

Lors des *tests d'intégration dans les applications de spring boot* , nous garderons à l'esprit que :

- Un test d'intégration est censé tester si différents modules sont correctement délimités et s'ils fonctionnent comme prévu.

- Les tests d'intégration ne doivent pas utiliser les dépendances de production réelles (par exemple, base de données/réseau) et ils peuvent imiter certains comportements.
- L'application doit s'exécuter dans *ApplicationContext* et y exécuter des tests.
- Spring Boot fournit l'annotation @SpringBootTest qui démarre le serveur intégré, crée un environnement Web, puis active les méthodes @Test pour effectuer des tests d'intégration. Utilisez son `webEnvironment` attribut pour cela.
- Il crée également l' `ApplicationContext` utilisé dans nos tests.
- Il est bon d'utiliser la base de données en mémoire h2 pour imiter la base de données. Bien que ce ne soit pas obligatoire et que nous puissions utiliser mockito pour simuler les interactions de la base de données.
- Il est recommandé d'utiliser des configurations spécifiques au test à l' aide de l'annotation @TestConfiguration .

CONSEIL: Spring Boot fournit de nombreuses annotations spécialisées pour tester certaines parties des applications, par exemple `@WebMvcTest`pour tester la couche Web ou `@DataJpaTest`pour tester la persistance ultérieurement.

À utiliser `@SpringBootTest`pour les tests qui couvrent l'ensemble de l'application Spring Boot, de la demande entrante à la base de données.

2. Écrire des tests d'intégration avec @SpringBootTest et Junit 5

2.1. Dépendances Maven

Commencez par inclure les dépendances requises. Nous devons utiliser spring-boot-starter-test qui utilisera en interne spring-test et d'autres bibliothèques dépendantes.

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

<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>org.junit.platform</groupId>
  <artifactId>junit-platform-launcher</artifactId>
  <scope>test</scope>
</dependency>
```

2.2. Contrôleur REST

Voici le *contrôleur Spring Boot Rest* , pour lequel nous allons écrire des tests.

- Le contrôleur dépend de la classe *EmployeeRepository* pour la persistance.

- *La méthode getEmployees()* renvoie la liste de tous les employés. Généralement, dans les applications réelles, il accepte les paramètres de pagination .
- *L'api addEmployee() a* besoin d'accéder au contexte de la requête à l'aide de `ServletUriComponentsBuilder`.
- *addEmployee()* api renvoie le statut HTTP et l'en-tête à l'aide `ResponseEntity`de la classe.

2.3. Essais d'intégration

La classe de test ci-dessous contient des tests d'intégration pour le contrôleur Spring Boot Rest mentionné ci-dessus. Cette classe de test :

- utilise l'annotation @SpringBootTest qui charge le contexte d'application réel.
- utilise WebEnvironment.RANDOM_PORT pour créer l'exécution de l'application sur un port de serveur aléatoire.
- @LocalServerPort obtient la référence du port sur lequel le serveur a démarré. Cela aide à créer les URI de requêtes réelles pour imiter les interactions réelles des clients.
- Utiliser la classe TestRestTemplate aide à invoquer les requêtes HTTP qui sont gérées par la classe de contrôleur.
- L'annotation @Sql aide à remplir la base de données avec certaines données préalables si le test en dépend pour tester correctement le comportement.
- `org.junit.jupiter.api.Test` les annotations proviennent de Junit 5 et marquent la méthode comme méthode de test à exécuter.

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

  @Autowired
  private TestRestTemplate restTemplate;
```

```
  @Sql({ "schema.sql", "data.sql" })
  @Test
  public void testAllEmployees()
  {
    assertTrue(
        this.restTemplate
          .getForObject("http://localhost:" + port + "/employees",
Employees.class)
          .getEmployeeList().size() == 3);
  }

  @Test
  public void testAddEmployee() {
    Employee employee = new Employee("Lokesh", "Gupta",
"howtodoinjava@gmail.com");
    ResponseEntity<String> responseEntity = this.restTemplate
      .postForEntity("http://localhost:" + port + "/employees", employee,
String.class);
    assertEquals(201, responseEntity.getStatusCodeValue());
  }
}
```