Guide to Test a Spring Boot App

Découvrez comment tester une application Web de *Spring boot* . Nous verrons quelques exemples très rapides (en utilisant Junit 5) et des configurations pour :

- Vérifier que l'application a été initialisée avec succès
- Test unitaire REST Controller avec`@WebMvcTest`
- Service Layer de test unitaire avec Mockito
- Test unitaire DAO Layer avec `@DataJpaTest` et `@AutoConfigureTestDatabase`
- Tests d'intégration à l'aide de`@SpringBootTest`
- Test du système à l'aide de `RestTemplate`

À des fins de démonstration, nous avons créé une application de gestion des employés très simple. Il a quelques appels d'API CRUD pour créer, récupérer et supprimer les employés de la base de données.

N'oubliez pas d'utiliser `@ExtendWith(SpringExtension.class)`pour exécuter les tests.

1. Maven

Cette application de démonstration utilise Spring boot 2 et Java 15. Elle dispose *de mysql-connector-java* pour se connecter à la base de données pour les opérations liées à la base de données MySQL.

Il inclut l'auto-configuration à partir des modules suivants :

- `spring-boot-starter-web`
- `spring-boot-starter-validation`
- `spring-boot-starter-test` avec Junit 5
- `spring-boot-starter-data-jpa`

Découvrez comment tester une application Web de *Spring boot* . Nous verrons quelques exemples très rapides (en utilisant Junit 5) et des configurations pour :

- Vérifier que l'application a été initialisée avec succès
- Test unitaire REST Controller avec `@WebMvcTest`
- Service Layer de test unitaire avec Mockito
- Test unitaire DAO Layer avec `@DataJpaTest` et `@AutoConfigureTestDatabase`
- Tests d'intégration à l'aide de `@SpringBootTest`
- Test du système à l'aide de `RestTemplate`

À des fins de démonstration, nous avons créé une application de gestion des employés très simple. Il a quelques appels d'API CRUD pour créer, récupérer et supprimer les employés de la base de données.

N'oubliez pas d'utiliser `@ExtendWith(SpringExtension.class)`pour exécuter les tests.

1. Maven

Cette application de démonstration utilise Spring boot 2 et Java 15. Elle dispose *de mysql-connector-java* pour se connecter à la base de données pour les opérations liées à la base de données MySQL.

Il inclut l'auto-configuration à partir des modules suivants :

- `spring-boot-starter-web`
- `spring-boot-starter-validation`
- `spring-boot-starter-test` avec Junit 5
- `spring-boot-starter-data-jpa`

```
<parent>
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.4.5</version>
  <relativePath /> 
</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>15</java.version>
</properties>

<dependencies>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

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

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

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
  </dependency>

  <dependency>
```

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

  <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>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
  </dependency>

</dependencies>
```

2. Tester si l'application se charge correctement

C'est le plus simple de tous. Écrivez une classe Test annotée avec `@SpringBootTest`et vérifiez si tout bean important initialisé avec impatience a été injecté avec succès dans un attribut auto-câblé ou non.

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
public class EmployeesApplicationTests {

  @Autowired
  EmployeeController employeeController;

  @Test
  public void contextLoads() {
    Assertions.assertThat(employeeController).isNot(null);
  }
}
```

3. Test unitaire du contrôleur REST

Écrivez une classe Test annotée avec `@WebMvcTest`. Nous pouvons spécifier quel contrôleur nous voulons tester dans la valeur d'annotation elle-même.

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(EmployeeController.class)
public class StandaloneControllerTests {

  @MockBean
  EmployeeService employeeService;

  @Autowired
  MockMvc mockMvc;

  @Test
  public void testfindAll() throws Exception {
    Employee employee = new Employee("Lokesh", "Gupta");
    List<Employee> employees = Arrays.asList(employee);


Mockito.when(employeeService.findAll()).thenReturn(employees)
;
```

```
    mockMvc.perform(get("/employee"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$", Matchers.hasSize(1)))
        .andExpect(jsonPath("$[0].firstName",
Matchers.is("Lokesh")));
  }

}
```

4. Test unitaire de la couche de service

Pour tester unitairement la couche service, nous devons utiliser mock la couche DAO. Ensuite, nous pouvons exécuter les tests en utilisant `MockitoExtension`.

5. Test unitaire DAO / Repository Layer

Pour tester unitairement la couche DAO, nous avons d'abord besoin d'une base de données de test en mémoire. Nous pouvons y parvenir en utilisant @AutoConfigureTestDatabase .

Ensuite, nous devons utiliser `@DataJpaTest`qui désactive la configuration automatique complète et n'applique à la place que la configuration pertinente pour les tests JPA.

6. Tests d'intégration

Les tests d'intégration couvrent l'ensemble du parcours de l'application. Dans ces tests, nous envoyons une requête à l'application et vérifions qu'elle répond correctement et a modifié l'état de la base de données conformément à nos attentes.

La base de données peut être une base de données physique réelle ou une base de données en mémoire à des fins de test.

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
public class IntegrationTests {

  @Autowired
  EmployeeController employeeController;

  @Test
  public void testCreateReadDelete() {
    Employee employee = new Employee("Lokesh", "Gupta");

    Employee employeeResult =
employeeController.create(employee);

    Iterable<Employee> employees = employeeController.read();

Assertions.assertThat(employees).first().hasFieldOrPropertyWi
thValue("firstName", "Lokesh");

    employeeController.delete(employeeResult.getId());

Assertions.assertThat(employeeController.read()).isEmpty();
  }

  @Test
  public void errorHandlingValidationExceptionThrown() {


Assertions.assertThatExceptionOfType(ValidationException.clas
s)
        .isThrownBy(() ->
employeeController.somethingIsWrong());
  }
}
```

7. Test du système à l'aide de RestTemplate

Nous pouvons utiliser `RestTemplate` la classe pour effectuer des tests système. Cela aide à vérifier l'application telle qu'elle apparaît au client en dehors de l'application.