Spring JdbcTemplate Unit Testing

Raúl Estrada

Octubre 2020

1. Overview

Spring *JdbcTemplate* is a powerful tool for developers to focus on writing SQL queries and extracting results. It connects to the back-end database and executes SQL queries directly.

Therefore, we can use integration tests to make sure that we can pull data from the database properly. Also, we can write unit tests to check the correctness of the related functionalities.

In this tutorial, we'll show how to unit test *JdbcTemplate* code.

2. *JdbcTemplate* and Running Queries

Firstly, let's start with a data access object (DAO) class that uses *JdbcTemplate*:

```java
public class EmployeeDAO {
    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int getCountOfEmployees() {
        return jdbcTemplate.queryForObject("SELECT COUNT(*) FROM EMPLOYEE", Integer.class);
    }
}
```

We dependency-inject a *DataSource* object into the *EmployeeDAO* class. Then, we create the *JdbcTemplate* object in the setter method. Also, we use *JdbcTemplate* in an example method *getCountOfEmployees()*.

There are two ways to unit test methods that use *JdbcTemplate*.

We can use an in-memory database such as the H2 database as the data source for testing. However, in real-world applications, the SQL query could have complicated relationships, and we need to create complex setup scripts to test the SQL statements.

Alternatively, we can also mock the *JdbcTemplate* object to test the method functionality.

3. Unit Test With H2 Database

We can create a data source that connects to the H2 database and inject it into the *EmployeeDAO* class:

```java
@Test
public void whenInjectInMemoryDataSource_thenReturnCorrectEmployeeCount() {
    DataSource dataSource = new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2)
      .addScript("classpath:jdbc/schema.sql")
      .addScript("classpath:jdbc/test-data.sql")
      .build();

    EmployeeDAO employeeDAO = new EmployeeDAO();
    employeeDAO.setDataSource(dataSource);

    assertEquals(4, employeeDAO.getCountOfEmployees());
}
```

In this test, we first construct a data source on the H2 database. During the construction, we execute *schema.sql* to create the *EMPLOYEE* table:

```
CREATE TABLE EMPLOYEE
(
    ID int NOT NULL PRIMARY KEY,
    FIRST_NAME varchar(255),
    LAST_NAME varchar(255),
    ADDRESS varchar(255)
);
```

Also, we run *test-data.sql* to add test data into the table:

```sql
INSERT INTO EMPLOYEE VALUES (1, 'James', 'Gosling', 'Canada');
INSERT INTO EMPLOYEE VALUES (2, 'Donald', 'Knuth', 'USA');
INSERT INTO EMPLOYEE VALUES (3, 'Linus', 'Torvalds', 'Finland');
INSERT INTO EMPLOYEE VALUES (4, 'Dennis', 'Ritchie', 'USA');
```

Then, we can inject this data source into the *EmployeeDAO* class and test the *getCountOfEmployees* method over the in-memory H2 database.

4. Unit Test With Mock Object

We can mock the *JdbcTemplate* object so that we don't need to run the SQL statement on a database:

```java
public class EmployeeDAOUnitTest {
    @Mock
    JdbcTemplate jdbcTemplate;

    @Test
    public void whenMockJdbcTemplate_thenReturnCorrectEmployeeCount() {
        EmployeeDAO employeeDAO = new EmployeeDAO();
        ReflectionTestUtils.setField(employeeDAO, "jdbcTemplate", jdbcTemplate);
        Mockito.when(jdbcTemplate.queryForObject("SELECT COUNT(*) FROM EMPLOYEE", Integer.class))
          .thenReturn(4);

        assertEquals(4, employeeDAO.getCountOfEmployees());
    }
}
```

In this unit test, we first declare a mock *JdbcTemplate* object with the *@Mock* annotation. Then we inject it to the *EmployeeDAO* object using *ReflectionTestUtils*. Also, we use the *Mockito* utility to mock the return result of the *JdbcTemplate* query. This allows us to test the functionality of the *getCountOfEmployees* method without connecting to a database.

We use an exact match on the SQL statement string when we mock the *JdbcTemplate* query. In real-world applications, we may create complex SQL strings, and it is hard to do an exact match. Therefore, we can also use the *anyString()* method to bypass the string check:

```java
Mockito.when(jdbcTemplate.queryForObject(Mockito.anyString(), Mockito.eq(Integer.class)))
  .thenReturn(3);
assertEquals(3, employeeDAO.getCountOfEmployees());
```

5. Spring Boot @JdbcTest

Finally, if we're using Spring Boot, there is an annotation we can use to bootstrap a test with an H2 database and a *JdbcTemplate* bean: *@JdbcTest*.

Let's create a test class with this annotation:

```java
@JdbcTest
@Sql({"schema.sql", "test-data.sql"})
class EmployeeDAOIntegrationTest {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Test
    void whenInjectInMemoryDataSource_thenReturnCorrectEmployeeCount() {
        EmployeeDAO employeeDAO = new EmployeeDAO();
        employeeDAO.setJdbcTemplate(jdbcTemplate);

        assertEquals(4, employeeDAO.getCountOfEmployees());
    }
}
```

We can also note the presence of the *@Sql* annotation which allows us to specify the SQL files to run before the test.

6. Conclusion

In this tutorial, we showed multiple ways to unit test *JdbcTemplate*.