Testing in Spring Boot

Last modified: August 18, 2019, by baeldung (https://www.baeldung.com/spring-boot-testing)

1. Overview

In this article, we'll have a look at **writing tests using the framework support in Spring Boot**. We'll cover unit tests that can run in isolation as well as integration tests that will bootstrap Spring context before executing tests.

2. Project Setup

The application we're going to use in this article is an API that provides some basic operations on an *Employee* Resource. This is a typical tiered architecture – the API call is processed from the *Controller* to *Service* to the *Persistence* layer.

3. Maven Dependencies

Let's first add our testing dependencies:

```
1    <dependency>
2        <groupId>org.springframework.boot</groupId>
3        <artifactId>spring-boot-starter-test</artifactId>
4        <scope>test</scope>
5        <version>2.1.6.RELEASE</version>
6    </dependency>
7    <dependency>
8        <groupId>com.h2database</groupId>
9        <artifactId>h2</artifactId>
10       <scope>test</scope>
11       <version>1.4.194</version>
12   </dependency>
```

The *spring-boot-starter-test* is the primary dependency that contains the majority of elements required for our tests.

The H2 DB is our in-memory database. It eliminates the need for configuring and starting an actual database for test purposes.

4. Integration Testing with *@DataJpaTest*

We're going to work with an entity named *Employee* which has an *id* and a *name* as its properties:

```
1    @Entity
2    @Table(name = "person")
3    public class Employee {
4
5        @Id
6        @GeneratedValue(strategy = GenerationType.AUTO)
7        private Long id;
8
9        @Size(min = 3, max = 20)
10       private String name;
11
12       // standard getters and setters, constructors
13   }
```

And here's our repository – using Spring Data JPA:

```
1   @Repository
2   public interface EmployeeRepository extends JpaRepository<Employee, Long> {
3
4       public Employee findByName(String name);
5
6   }
```

That's it for the persistence layer code. Now let's head towards writing our test class.

First, let's create the skeleton of our test class:

```
1   @RunWith(SpringRunner.class)
2   @DataJpaTest
3   public class EmployeeRepositoryIntegrationTest {
4
5       @Autowired
6       private TestEntityManager entityManager;
7
8       @Autowired
9       private EmployeeRepository employeeRepository;
10
11      // write test cases here
12
13  }
```

@RunWith(SpringRunner.class) is used to provide a bridge between Spring Boot test features and JUnit. Whenever we are using any Spring Boot testing features in our JUnit tests, this annotation will be required.

@DataJpaTest provides some standard setup needed for testing the persistence layer:

- configuring H2, an in-memory database
- setting Hibernate, Spring Data, and the *DataSource*
- performing an *@EntityScan*
- turning on SQL logging

To carry out some DB operation, we need some records already setup in our database. To setup this data, we can use *TestEntityManager*. **The *TestEntityManager* provided by Spring Boot is an alternative to the standard JPA *EntityManager* that provides methods commonly used when writing tests.**

EmployeeRepository is the component that we are going to test. Now let's write our first test case:

```
1  @Test
2  public void whenFindByName_thenReturnEmployee() {
3      // given
4      Employee alex = new Employee("alex");
5      entityManager.persist(alex);
6      entityManager.flush();
7
8      // when
9      Employee found = employeeRepository.findByName(alex.getName());
10
11     // then
12     assertThat(found.getName())
13       .isEqualTo(alex.getName());
14 }
```

In the above test, we're using the *TestEntityManager* to insert an *Employee* in the DB and reading it via the find by name API.

The *assertThat(...)* part comes from the Assertj library which comes bundled with Spring Boot.

5. Mocking with *@MockBean*

Our *Service* layer code is dependent on our *Repository*. However, to test the *Service* layer, we do not need to know or care about how the persistence layer is implemented:

```
1  @Service
2  public class EmployeeServiceImpl implements EmployeeService {
3
4      @Autowired
5      private EmployeeRepository employeeRepository;
6
7      @Override
8      public Employee getEmployeeByName(String name) {
9          return employeeRepository.findByName(name);
10     }
11 }
```

Ideally, we should be able to write and test our Service layer code without wiring in our full persistence layer.

To achieve this, **we can use the mocking support provided by Spring Boot Test**.

Let's have a look at the test class skeleton first:

```
1  @RunWith(SpringRunner.class)
2  public class EmployeeServiceImplIntegrationTest {
3
4      @TestConfiguration
5      static class EmployeeServiceImplTestContextConfiguration {
6
7          @Bean
8          public EmployeeService employeeService() {
9              return new EmployeeServiceImpl();
10         }
11     }
12
13     @Autowired
14     private EmployeeService employeeService;
15
16     @MockBean
17     private EmployeeRepository employeeRepository;
18
19     // write test cases here
20 }
```

To check the *Service* class, we need to have an instance of *Service* class created and available as a *@Bean* so that we can *@Autowire* it in our test class. This configuration is achieved by using the *@TestConfiguration* annotation.

During component scanning, we might find components or configurations created only for specific tests accidentally get picked up everywhere. To help prevent that, **Spring Boot provides *@TestConfiguration* annotation that can be used on classes in *src/test/java* to indicate that they should not be picked up by scanning.**

Another interesting thing here is the use of *@MockBean*. It creates a Mock for the *EmployeeRepository* which can be used to bypass the call to the actual *EmployeeRepository*:

```
1  @Before
2  public void setUp() {
3      Employee alex = new Employee("alex");
4
5      Mockito.when(employeeRepository.findByName(alex.getName()))
6        .thenReturn(alex);
7  }
```

Since the setup is done, the test case will be simpler:

```
1  @Test
2  public void whenValidName_thenEmployeeShouldBeFound() {
3      String name = "alex";
4      Employee found = employeeService.getEmployeeByName(name);
5
6       assertThat(found.getName())
7        .isEqualTo(name);
8   }
```

6. Unit Testing with *@WebMvcTest*

Our *Controller* depends on the *Service* layer; let's only include a single method for simplicity:

```
1  @RestController
2  @RequestMapping("/api")
3  public class EmployeeRestController {
4
5      @Autowired
6      private EmployeeService employeeService;
7
8      @GetMapping("/employees")
9      public List<Employee> getAllEmployees() {
10         return employeeService.getAllEmployees();
11     }
12 }
```

Since we are only focused on the *Controller* code, it is natural to mock the *Service* layer code for our unit tests:

```
1  @RunWith(SpringRunner.class)
2  @WebMvcTest(EmployeeRestController.class)
3  public class EmployeeRestControllerIntegrationTest {
4
5      @Autowired
6      private MockMvc mvc;
7
8      @MockBean
9      private EmployeeService service;
10
11     // write test cases here
12 }
```

To test the *Controllers*, we can use *@WebMvcTest*. It will auto-configure the Spring MVC infrastructure for our unit tests.

In most of the cases, *@WebMvcTest* will be limited to bootstrap a single controller. It is used along with *@MockBean* to provide mock implementations for required dependencies.

@WebMvcTest also auto-configures *MockMvc* which offers a powerful way of easy testing MVC controllers without starting a full HTTP server.

Having said that, let's write our test case:

```
1  @Test
2  public void givenEmployees_whenGetEmployees_thenReturnJsonArray() throws Exception {
3
4      Employee alex = new Employee("alex");
5      List<Employee> allEmployees = Arrays.asList(alex);
6      given(service.getAllEmployees()).willReturn(allEmployees);
7
8      mvc.perform(get("/api/employees")
9        .contentType(MediaType.APPLICATION_JSON))
10       .andExpect(status().isOk())
11       .andExpect(jsonPath("$", hasSize(1)))
12       .andExpect(jsonPath("$[0].name", is(alex.getName())));
13 }
```

The *get(...)* method call can be replaced by other methods corresponding to HTTP verbs like *put()*, *post()*, etc. Please note that we are also setting the content type in the request.

MockMvc is flexible, and we can create any request using it.

7. Integration Testing with *@SpringBootTest*

As the name suggests, integration tests focus on integrating different layers of the application. That also means no mocking is involved.

Ideally, we should keep the integration tests separated from the unit tests and should not run along with the unit tests. We can do that by using a different profile to only run the integration tests. A couple of reasons for doing this could be that the integration tests are time-consuming and might need an actual database to execute.

However, in this article, we won't focus on that and we'll instead make use of the in-memory H2 persistence storage.

The integration tests need to start up a container to execute the test cases. Hence, some additional setup is required for this – all of this is easy in Spring Boot:

```
1  @RunWith(SpringRunner.class)
2  @SpringBootTest(
3      SpringBootTest.WebEnvironment.MOCK,
4      classes = Application.class)
5  @AutoConfigureMockMvc
6  @TestPropertySource(
7      locations = "classpath:application-integrationtest.properties")
8  public class EmployeeRestControllerIntegrationTest {
9
10     @Autowired
11     private MockMvc mvc;
12
13     @Autowired
14     private EmployeeRepository repository;
15
16     // write test cases here
17 }
```

The *@SpringBootTest* annotation can be used when we need to bootstrap the entire container. The annotation works by creating the *ApplicationContext* that will be utilized in our tests.

We can use the *webEnvironment* attribute of *@SpringBootTest* to configure our runtime environment; we're using *WebEnvironment.MOCK* here – so that the container will operate in a mock servlet environment.

We can use the *@TestPropertySource* annotation to configure locations of properties files specific to our tests. Please note that the property file loaded with *@TestPropertySource* will override the existing *application.properties* file.

The *application-integrationtest.properties* contains the details to configure the persistence storage:

```
1  spring.datasource.url = jdbc:h2:mem:test
2  spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialect
```

If we want to run our integration tests against MySQL, we can change the above values in the properties file.

The test cases for the integration tests might look similar to the *Controller* layer unit tests:

```
1  @Test
2  public void givenEmployees_whenGetEmployees_thenStatus200()
3    throws Exception {
4
5      createTestEmployee("bob");
6
7      mvc.perform(get("/api/employees")
8        .contentType(MediaType.APPLICATION_JSON))
9        .andExpect(status().isOk())
10       .andExpect(content()
11       .contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
12       .andExpect(jsonPath("$[0].name", is("bob")));
13 }
```

The difference from the *Controller* layer unit tests is that here nothing is mocked and end-to-end scenarios will be executed.

8. Auto-Configured Tests

One of the amazing features of Spring Boot's auto-configured annotations is that it helps to load parts of the complete application and test specific layers of the codebase.

In addition to the above-mentioned annotations here's a list of a few widely used annotations:

- *@WebFluxTest* – *w*e can use the *@WebFluxTest* annotation to test Spring Webflux controllers. It's often used along with *@MockBean* to provide mock implementations for required dependencies.

- *@JdbcTest* – *w*e can use the *@JdbcTest* annotation to test JPA applications but it's for tests that only require a *DataSource.* The annotation configures an in-memory embedded database and a *JdbcTemplate.*

- *@JooqTest* – To test jOOQ-related tests we can use *@JooqTest* annotation, which configures a DSLContext.

- *@DataMongoTest* – To test MongoDB applications *@DataMongoTest* is a useful annotation. By default, it configures an in-memory embedded MongoDB if the driver is available through dependencies, configures a *MongoTemplate,* scans for *@Document* classes, and configures Spring Data MongoDB repositories.

- *@DataRedisTest* – *makes it easier to test Redis* applications. It scans for *@RedisHash* classes and configures Spring Data Redis repositories by default.
- *@DataLdapTest* – configures an in-memory embedded *LDAP* (if available), configures a *LdapTemplate*, scans for *@Entry* classes, and configures Spring Data *LDAP* repositories by default

- *@RestClientTest* – *w*e generally use the *@RestClientTest* annotation to test REST clients. It auto-configures different dependencies like Jackson, GSON, and Jsonb support, configures a *RestTemplateBuilder*, and adds support for *MockRestServiceServer* by default.

9. Conclusion

In this tutorial, we took a deep dive into the testing support in Spring Boot and showed how to write unit tests efficiently.

The complete source code of this article can be found over on GitHub. The source code contains many more examples and various test cases.

Injecting Mockito Mocks into Spring Beans

Last modified: November 4, 2018, by baeldung

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

1. Overview

This article will show how to use dependency injection to insert Mockito mocks into Spring Beans for unit testing.

In real-world applications, where components often depend on accessing external systems, it is important to provide proper test isolation so that we can focus on testing the functionality of a given unit without having to involve the whole class hierarchy for each test.

Injecting a mock is a clean way to introduce such isolation.

2. Maven Dependencies

The following Maven dependencies are required for the unit tests and the mock objects:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>1.3.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <version>1.3.1.RELEASE</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.21.0</version>
</dependency>
```

We decided to use Spring Boot for this example, but classic Spring will also work fine.

3. Writing the Test

3.1. The Business Logic

First, we are going to write the business logic that will be tested:

```
@Service
public class NameService {
    public String getUserName(String id) {
        return "Real user name";
    }
}
```

The *NameService* class will be injected into:

```
1  @Service
2  public class UserService {
3
4      private NameService nameService;
5
6      @Autowired
7      public UserService(NameService nameService) {
8          this.nameService = nameService;
9      }
10
11     public String getUserName(String id) {
12         return nameService.getUserName(id);
13     }
14 }
```

For this tutorial, the given classes return a single name regardless of the id provided. This is done so that we don't get distracted by testing any complex logic.

We will also need a standard Spring Boot main class to scan the beans and initialize the application:

```
1  @SpringBootApplication
2  public class MocksApplication {
3      public static void main(String[] args) {
4          SpringApplication.run(MocksApplication.class, args);
5      }
6  }
```

3.2. The Tests

Now let's move on to the test logic. First, we have to configure application context for the tests:

```
1  @Profile("test")
2  @Configuration
3  public class NameServiceTestConfiguration {
4      @Bean
5      @Primary
6      public NameService nameService() {
7          return Mockito.mock(NameService.class);
8      }
9  }
```

The *@Profile* annotation tells Spring to apply this configuration only when the “test” profile is active. The *@Primary* annotation is there to make sure this instance is used instead of a real one for autowiring. The method itself creates and returns a Mockito mock of our *NameService* class.

Now we can write the unit test:

```
1  @ActiveProfiles("test")
2  @RunWith(SpringJUnit4ClassRunner.class)
3  @SpringApplicationConfiguration(classes = MocksApplication.class)
4  public class UserServiceTest {
5
6      @Autowired
7      private UserService userService;
8
9      @Autowired
10     private NameService nameService;
11
12     @Test
13     public void whenUserIdIsProvided_thenRetrievedNameIsCorrect() {
14         Mockito.when(nameService.getUserName("SomeId")).thenReturn("Mock user name");
15         String testName = userService.getUserName("SomeId");
16         Assert.assertEquals("Mock user name", testName);
17     }
18 }
```

We use the *@ActiveProfiles* annotation to enable the "test" profile and activate the mock configuration we wrote earlier. Because of this, Spring autowires a real instance of the *UserService* class, but a mock of the *NameService* class. The test itself is a fairly typical JUnit+Mockito test. We configure the desired behavior of the mock, then call the method which we want to test and assert that it returns the value that we expect.

It is also possible (though not recommended) to avoid using environment profiles in such tests. To do so, remove the *@Profile and @ActiveProfiles* annotations and add an *@ContextConfiguration(classes = NameServiceTestConfiguration.class)* annotation to the *UserServiceTest* class.