Testing in Spring Boot

Last updated: January 8, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Reviewed by: Michal Aibin (https://www.baeldung.com/editor/michal-author)

Spring Boot (https://www.baeldung.com/category/spring/spring-boot)

Testing (https://www.baeldung.com/category/testing)

Boot Basics (https://www.baeldung.com/tag/boot-basics)

1. Overview

In this tutorial, 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.

If you are new to Spring Boot, check out our intro to Spring Boot (/spring-boot-start).

(/)

Further reading:

Exploring the Spring Boot TestRestTemplate (/?post_type=post&p=18728)

Learn how to use the new TestRestTemplate in Spring Boot to test a simple API.

Read more (/?post_type=post&p=18728) →

Quick Guide to @RestClientTest in Spring Boot (/?post_type=post&p=12147)

A quick and practical guide to the @RestClientTest annotation in Spring Boot

Read more (/?post_type=post&p=12147) →

Injecting Mockito Mocks into Spring Beans (/?post_type=post&p=9159)

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

Read more (/?post_type=post&p=9159) →

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:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <version>3.3.2</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

The *spring-boot-starter-test* (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-test) is the primary dependency that contains the majority of elements required for our tests.

The H2 DB (https://mvnrepository.com/artifact/com.h2database/h2) is our in-memory database. It eliminates the need to configure and start an actual database for test purposes.

4. 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 separate from the unit tests and not run along with the unit tests. We can do this 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:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(
  webEnvironment = SpringBootTest.WebEnvironment.MOCK,
  classes = Application.class)
@AutoConfigureMockMvc
@TestPropertySource(
  locations = "classpath:application-integrationtest.properties")
public class EmployeeRestControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private EmployeeRepository repository;

    // write test cases here
}
```

The *@SpringBootTest* annotation is useful 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.

Next, the *@TestPropertySource* annotation helps configure the locations of properties files specific to our tests. 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:

```
spring.datasource.url = jdbc:h2:mem:test
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:

```
@Test                         (/)
public void givenEmployees_whenGetEmployees_thenStatus200()
  throws Exception {

    createTestEmployee("bob");

    mvc.perform(get("/api/employees")
      .contentType(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk())
      .andExpect(content()
      .contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
      .andExpect(jsonPath("$[0].name", is("bob")));
}
```

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

5. Test Configuration With *@TestConfiguration*

As we've seen in the previous section, a test annotated with *@SpringBootTest* will bootstrap the full application context, which means we can *@Autowire* any bean that's picked up by component scanning into our test:

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

    @Autowired
    private EmployeeService employeeService;

    // class code ...
}
```

However, we might want to avoid bootstrapping the real application context but use a special test configuration. We can achieve this with the *@TestConfiguration* annotation. There are two ways of using the annotation. Either on a static inner class in the same test class where we want to *@Autowire* the bean:

```
@ExtendWith(SpringExtension.class)
public class EmployeeServiceImplIntegrationTest {

    @TestConfiguration
    static class EmployeeServiceImplTestContextConfiguration {
        @Bean
        public EmployeeService employeeService() {
            return new EmployeeService() {
                // implement methods
            };
        }
    }

    @Autowired
    private EmployeeService employeeService;
}
```

Alternatively, we can create a separate test configuration class:

```
@TestConfiguration
public class EmployeeServiceImplTestContextConfiguration {

    @Bean
    public EmployeeService employeeService() {
        return new EmployeeService() {
            // implement methods
        };
    }
}
```

Configuration classes annotated with *@TestConfiguration* are excluded from component scanning. Therefore, we need to import it explicitly in every test where we want to *@Autowire* it. We can do that with the *@Import* annotation:

```
@ExtendWith(SpringExtension.class)
@Import(EmployeeServiceImplTestContextConfiguration.class)
public class EmployeeServiceImplIntegrationTest {

    @Autowired
    private EmployeeService employeeService;

    // remaining class code
}
```

6. Mocking With *@MockBean*

Our *Service* layer code is dependent on our *Repository:*

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Override
    public Employee getEmployeeByName(String name) {
        return employeeRepository.findByName(name);
    }
}
```

However, to test the *Service* layer, we don't need to know or care about how the persistence layer is implemented. 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:

```
@ExtendWith(SpringExtension.class)
public class EmployeeServiceImplIntegrationTest {

    @TestConfiguration
    static class EmployeeServiceImplTestContextConfiguration {

        @Bean
        public EmployeeService employeeService() {
            return new EmployeeServiceImpl();
        }
    }

    @Autowired
    private EmployeeService employeeService;

    @MockBean
    private EmployeeRepository employeeRepository;

    // write test cases here
}
```

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

Another interesting thing here is the use of *@MockBean*. It creates a Mock (/mockito-mock-methods) for the *EmployeeRepository*, which can be used to bypass the call to the actual *EmployeeRepository*.

```
@BeforeEach
public void setUp() {
    Employee alex = new Employee("alex");

    Mockito.when(employeeRepository.findByName(alex.getName()))
      .thenReturn(alex);
}
```

With the setup complete, the test case becomes simpler:

```
@Test                         (/)
public void whenValidName_thenEmployeeShouldBeFound() {
    String name = "alex";
    Employee found = employeeService.getEmployeeByName(name);

     assertThat(found.getName())
      .isEqualTo(name);
 }
```

7. Integration Testing With *@DataJpaTest*

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

```
@Entity
@Table(name = "person")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Size(min = 3, max = 20)
    private String name;

    // standard getters and setters, constructors
}
```

And here's our repository using Spring Data JPA:

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long>
{

    public Employee findByName(String name);

}
```

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

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

```
@ExtendWith(SpringExtension.class)
@DataJpaTest
public class EmployeeRepositoryIntegrationTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private EmployeeRepository employeeRepository;

    // write test cases here

}
```

@ExtendWith(SpringExtension.class) provides 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 DB operations, we need some records already in our database. To setup this data, we can use *TestEntityManager.*

The Spring Boot *TestEntityManager* 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:

```
@Test                         (/)
public void whenFindByName_thenReturnEmployee() {
    // given
    Employee alex = new Employee("alex");
    entityManager.persist(alex);
    entityManager.flush();

    // when
    Employee found = employeeRepository.findByName(alex.getName());

    // then
    assertThat(found.getName())
      .isEqualTo(alex.getName());
}
```

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

The *assertThat(…)* part comes from the Assertj library (/introduction-to-assertj), which comes bundled with Spring Boot.

8. Unit Testing With *@WebMvcTest*

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

```
@RestController
@RequestMapping("/api")
public class EmployeeRestController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/employees")
    public List<Employee> getAllEmployees() {
        return employeeService.getAllEmployees();
    }
}
```

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

```
@ExtendWith(SpringExtension.class)
@WebMvcTest(EmployeeRestController.class)
public class EmployeeRestControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private EmployeeService service;

    // write test cases here
}
```

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

In most cases, *@WebMvcTest* will be limited to bootstrap a single controller. We can also use it along with *@MockBean* to provide mock implementations for any 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:

```
@Test
public void givenEmployees_whenGetEmployees_thenReturnJsonArray()
  throws Exception {

    Employee alex = new Employee("alex");

    List<Employee> allEmployees = Arrays.asList(alex);

    given(service.getAllEmployees()).willReturn(allEmployees);

    mvc.perform(get("/api/employees")
      .contentType(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk())
      .andExpect(jsonPath("$", hasSize(1)))
      .andExpect(jsonPath("$[0].name", is(alex.getName())));
}
```

We can replace the *get(...)* method call with 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.

9. 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:

Annotation	Description
@WebFluxTest	Used to test Spring WebFlux controllers. Often used with *@MockBean* to provide mock implementations for required dependencies.
@JdbcTest	Used to test JPA applications that only require a *DataSource*. Configures an in-memory embedded database and a *JdbcTemplate*.
@JooqTest	Used to test jOOQ-related components. Configures a DSLContext.
@DataMongoTest	Used to test MongoDB applications. Configures an in-memory embedded MongoDB (if the driver is available), a *MongoTemplate*, scans for *@Document* classes, and configures Spring Data MongoDB repositories.
@DataRedisTest	Facilitates testing of Redis applications. Scans for *@RedisHash* classes and configures Spring Data Redis repositories by default.
@DataLdapTest	Configures an in-memory embedded *LDAP* (if available), a *LdapTemplate*, scans for *@Entry* classes, and configures Spring Data *LDAP* repositories by default.

Annotation	Description
@RestClientTest	Used to test REST clients. Auto-configures dependencies such as Jackson, Gson, and Jsonb support; configures a *RestTemplateBuilder*; and adds support for *MockRestServiceServer* by default.
@JsonTest	Initializes the Spring application context only with beans needed to test JSON serialization.

We can read more about these annotations and how to further optimize integration tests in our article on Optimizing Spring Integration Tests (/spring-tests).

10. Conclusion

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

The code backing this article is available on GitHub. Once you're **logged in as a Baeldung Pro Member (/members/)**, start learning and coding on the project.

If you want to keep learning about testing, we have separate articles related to integration tests (/integration-testing-in-spring), optimizing Spring integration tests (/spring-tests), and unit tests in JUnit 5 (/junit-5).

(/)

COURSES

ALL COURSES (/COURSES/ALL-COURSES)
BAELDUNG ALL ACCESS (/COURSES/ALL-ACCESS)
BAELDUNG ALL TEAM ACCESS (/COURSES/ALL-ACCESS-TEAM)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
LEARN SPRING BOOT SERIES (/SPRING-BOOT)
SPRING TUTORIAL (/SPRING-TUTORIAL)
GET STARTED WITH JAVA (/GET-STARTED-WITH-JAVA-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
REST WITH SPRING SERIES (/REST-WITH-SPRING-SERIES)
ALL ABOUT STRING IN JAVA (/JAVA-STRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS/)
PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)
EBOOKS (/LIBRARY/)
FAQ (/LIBRARY/FAQ)
BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)

(/)

PRIVACY MANAGER