Testing in Spring Boot

Last modified: November 6, 2018

by baeldung (/author/baeldung/)

Spring Boot (/category/spring/spring-boot/)
Testing (/category/testing/)

Boot Basics (/tag/boot-basics/)

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.

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

Further reading:

Exploring the Spring Boot TestRestTemplate (https://www.baeldung.com/spring-boot-testresttemplate)

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

Read more (https://www.baeldung.com/spring-boot-testresttemplate) →

Quick Guide to @RestClientTest in Spring Boot (https://www.baeldung.com/restclienttest-in-spring-boot)

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

Read more (https://www.baeldung.com/restclienttest-in-spring-boot) →

Injecting Mockito Mocks into Spring Beans (https://www.baeldung.com/injecting-mocks-in-spring)

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

Read more (https://www.baeldung.com/injecting-mocks-in-spring) →

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:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <version>2.0.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
    <version>1.4.194</version>
</dependency>
```

The *spring-boot-starter-test* (https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.boot%22%20AND%20a%3A%22spring-boot-starter-test%22) is the primary dependency that contains the majority of elements required for our tests.

The H2 DB (https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22com.h2database%22%20AND%20a%3A%22h2%22) 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:

```java
@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:

```java
@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 towards writing our test class.

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

```java
@RunWith(SpringRunner.class)
@DataJpaTest
public class EmployeeRepositoryIntegrationTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private EmployeeRepository employeeRepository;

    // write test cases here

}
```

@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 out 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 such 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:

```
@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 reading it via the find by name API.

The *assertThat(…)* part comes from the Assertj library (/introduction-to-assertj) 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:

```java
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

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

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:

```java
@RunWith(SpringRunner.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 *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 (/mockito-mock-methods) for the *EmployeeRepository* which can be used to bypass the call to the actual *EmployeeRepository*:

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

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

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

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

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

6. Unit Testing with *@WebMvcTest*

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

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

    @Autowired
    private EmployeeService employeeService;

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

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

```java
@RunWith(SpringRunner.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 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:

```java
@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())));
}
```

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:

```
@RunWith(SpringRunner.class)
@SpringBootTest(
  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 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:

```
spring.datasource.url = jdbc:h2:mem:test
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.H2Dialec
```

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.

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* – *we* 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* – *we* 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 – we* 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.

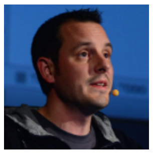
9. Conclusion

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 (https://github.com/eugenp/tutorials/tree/master/spring-boot). The source code contains many more examples and various test cases.

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

▲ newest ▲ oldest ▲ most voted



Guest

Stéphane Nicoll (https://www.about.me/snicoll)

Thanks for the article. I am not sure I understand the `@TestPropertySource` part on the integration test. If you want to enable an `integrationtest` profile (that's really what your file looks like), you can just add `@ActiveProfiles("integrationtest")` and Spring Boot will load that file automatically. Also, you do not need to do that if you want to use H2. Just add `@AutoconfigureTestDatabase` and we'll replace your `DataSource` with an embedded database for you. I am also curious why you need to refer to `Application` in your integration test. Do you have others `@SpringBootApplication` in this project? If you don't, we'll… Read more »

+ 0 −

1 year ago

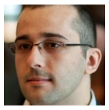
Guest

Dusan Odalovic (https://odalinjo.wordpress.com/)

@snicoll:disqus Stéphane, would it be possible to provide lots more small sample apps so that we can just check them out and learn by examples? Spring Boot helps a lot but IMHO documentation is not at the same level.

+ 0 −

1 year ago



Guest

Eugen Paraschiv (http://www.baeldung.com/)

Hey @snicoll:disqus – thanks for the feedback – I'll ask the author and also have a look at your points and potentially jump in and address them. You're right – the terminology needs a bit of cleanup/clarification here.
Cheers,
Eugen.

+ 0 −

1 year ago

Guest

Fernando Fradegrada

I am trying to follow the @DataJpaTest and I cannot achieve to run the test. It's like all of my application context is being tried to load, and fails to load my controllers, services, etc. What's wrong??

+ 2 −

1 year ago

Guest

Grzegorz Piwowarek

Can you share you stacktrace? Without this we could only guess blindly

2

1 year ago

Guest

Fernando Fradegrada

So I've found what was the problem, but I still not understand why: In my Spring Boot main class I have override the @ComponentScan with this, because I need to @Autowire a util in another jar. So that's is overriding something that makes my test to load all the App Context. If I remove the @ComponentScan, the test runs ok, but then I will not have my autowired component when running my app. I know that this question has nothing to do here, but can you send me a link to understand this? Sorry for my english! @SpringBootApplication @ComponentScan({ "ar.com.myapp.utils"… Read more »

0

1 year ago

Guest

Fernando Fradegrada

How can I deal with spring security in the integration tests? I get 401 response. Is there a way to bypass the security? Or maybe the good practice is to login before perform request?

1

1 year ago

Guest

Grzegorz Piwowarek

The general approach is to set up your restTemplate before testing and then use it freely. Take a look at TestRestTemplate because it has some additional useful methods

0

1 year ago

CATEGORIES

SPRING (/CATEGORY/SPRING/)
REST (/CATEGORY/REST/)
JAVA (/CATEGORY/JAVA/)
SECURITY (/CATEGORY/SECURITY-2/)
PERSISTENCE (/CATEGORY/PERSISTENCE/)
JACKSON (/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (/CATEGORY/HTTP/)
KOTLIN (/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

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