# Integration Testing in Spring Boot 2

Integration tests can put whole application in scope or only certain components – based on what is being tested. They may need to require resources like database instances and hardware to be allocated for them. Though these interactions can be mocked out as well to improve the test performance.

In terms of typical Spring Boot crud application, unit tests can be written to test REST controllers, DAO layer etc. separately. It will not require even the embedded server as well.

In integration testing, we shall focus on testing complete request processing from controller to persistence layer. Application shall run inside embedded server to create application context and all beans. Some of these beans may be overridden to mock certain behaviors.

**Dependencies**

**Pom.xml**

<dependency>

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

    <artifactId>spring-boot-starter-test</artifactId>

    <scope>test</scope>

</dependency>

**Spring Boot test annotations**

@SpringBootTest

This annotation helps in writing integration tests. It starts the embedded server and fully initializes the application context. We can inject the dependencies in test class using @Autowired annotation.

We can also provide test specific beans configuration using nested @Configuration class or explicit @TestConfiguration classes. It also provides support for different web Environment modes and running web server listening on a defined or random port. It also registers a TestRestTemplate and/or WebTestClient bean for use in web tests.

@SpringBootTest example

@SpringBootTest (classes = SpringBootDemoApplication.class,

webEnvironment = WebEnvironment.RANDOM\_PORT)

public class EmployeeControllerIntegrationTests

{

@LocalServerPort

private int port;

@Autowired

private TestRestTemplate restTemplate;

//tests

}

#### @WebMvcTest

This annotation is used for Spring MVC tests. It disables full auto-configuration and instead apply only configuration relevant to MVC tests.

It also auto-configure **MockMvc** instance as well. We can initialize only one web controller by passing .class as the annotation attribute.

|  |
| --- |
| @WebMvcTest example |
| @WebMvcTest(EmployeeRESTController.class)  public class TestEmployeeRESTController {      @Autowired      private MockMvc mvc;    } |

**@TestConfiguration**

@TestConfiguration is specialized form of @Configuration that can be used to define additional beans or customizations for a test.

In spring boot, any beans configured in a top-level class annotated with @TestConfiguration will not be picked up via component scanning. We must explicitly register the @TestConfiguration class with the class that contains the test cases.

The best thing is that these test configurations are not automatically part of application’s primary configuration. They are available only on-demand using one of below two ways to include this additional test configuration i.e.

**@Test**

@Test the Test annotation tells JUnit that the public void method to which it is attached can be run as a test case. To run the method, JUnit first constructs a fresh instance of the class then invokes the annotated method. Any exceptions thrown by the test will be reported by JUnit as a failure. If no exceptions are thrown, the test is assumed to have succeeded.

**@Before**

@Before: When writing tests, it is common to find that several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before the Test method. The @Before methods of super classes will be run before those of the current class.

**@After**

@After: If you allocate external resources in a before method you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method. All @After methods are guaranteed to run even if a before or Test method throws an exception. The @After methods declared in super classes will be run after those of the current class.