I'm not a robot

Spring boot starter test

Spring Boot provides a rich set of testing features, consisting of the following two modules: spring-boot-test: Provides core functionality for testing. spring-boot-test-autoconfigure: Provides automatic configuration of tests. Spring Boot provides a spring-boot-starter-test one-stop starter, as shown in the following dependency configuration. 1 2 3 4 5 org.springframework.boot spring-boot-starter-test test The Test starter dependency contains not only the two Spring Boot modules above, but also the Spring Test test module, as well as other third-party testing libraries, as shown below. JUnit 5: Java's most dominant unit testing framework. AssertJ: A fast assertion library. Hamcrest: A unit test matching library. Mockito: a Mock testing framework. JSONassert: a JSON assertion library . JsonPath: A JSON XPath library. More test-related dependencies can be found in the specific dependency tree, as shown in the following figure. These are the common test libraries provided by Spring Boot. If the above test libraries don't meet your needs, you can add any libraries that are not available above. Basically, JUnit 5 is used now. if your application is still using JUnit 4 to write unit test cases, you can also use JUnit 5's Vintage engine to run them, as shown in the dependency configuration below. 1 2 3 4 5 6 7 8 9 10 11 12 org.junit.vintage junit-vintage-engine test org.hamcrest hamcrest-core The hamcrest-core dependency needs to be excluded because it has changed coordinates and is built into Spring Boot dependency management by default, as shown in the dependency tree above, and the latest Hamcrest coordinates are now org.hamcrest:hamcrest. Spring Boot provides a @SpringBootTest annotation to be used on unit test classes to enable unit tests that support Spring Boot features, and if you are using JUnit 4, then the @RunWith(SpringRunner.class) annotation is required on the test class. Then just add @Test annotations to the test class methods, each @Test annotated method is a unit test method. The @SpringBootTest annotation has one of the most important webEnvironment environment parameters and supports the following environment settings: MOCK (default): loads a Web ApplicationContext and provides a Mock Web Environment, but does not start the embedded web server and can be used in combination with the @AutoConfigureMockMvc and @AutoConfigureWebTestClient annotations for Mock testing. RANDOM_PORT: Loads a WebServerApplicationContext, as well as providing a real WebEnvironment and starting the embedded server with a random port. DEFINED_PORT: Same as RANDOM_PORT, the difference is that DEFINED_PORT runs on the port specified by the application, the default port is 8080. NONE: loads an ApplicationContext, but does not provide any Web Environment. If the @SpringBootTest annotation is used without any parameters, it defaults to the Mock environment. Real Environment Testing Specifying a real web environment based on random ports in @SpringBootTest annotation and then injecting TestRestTemplate instances on class member variables or method parameters will complete the real environment testing of Spring MVC interfaces. The following is a test case for a real environment based on a random port: 1 2 3 4 5 6 7 8 9 10 11 12 @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT) public class MvcTest { @Test public void getUserTest(@Autowired TestRestTemplate testRestTemplate) { Map multiValueMap = new HashMap(); multiValueMap.put("username", "Java"); Result result = testRestTemplate.getForObject("/user/get?username={username}", Result.class, multiValueMap); assertThat(result.getCode()).isEqualTo(0); assertThat(result.getMsg()).isEqualTo("ok"); } } Test the /user/get interface under the current application, pass in the corresponding user name parameter, and finally check if the interface returns the same result as expected, as shown below. The unit test passes, and as you can see from the execution log, it starts an embedded Tomcat container to test the real web application environment. Mock Environment Testing Mock testing of Spring MVC interfaces can be accomplished by annotating @AutoConfigureMockMvc on classes and then injecting MockMvc instances on class member variables or method parameters. The following is a test case based on the default Mock environment. 1 2 3 4 5 6 7 8 9 10 11 @SpringBootTest @AutoConfigureMockMvc class MockMvcTests { @Test public void getUserTest(@Autowired MockMvc mvc) throws Exception { mvc.perform(MockMvcRequestBuilders.get("/user/get?username={username}", "test")) .andExpect(status().isOk()) .andExpect(content().string("{\"code\":0,\"msg\":\"ok\",\" data\":\"test\"}")); } } Test the /user/get interface under the current application, pass in the corresponding username parameter, and finally check whether the request status is OK (200) and whether the response is consistent with the expected content. The unit test passes, and as you can see from the execution log, it does not start a real web environment to test, but uses a Mock environment to test. Mock Component Testing There may be times when you need to mock some components, such as some services that can only be invoked after going live and are not available during the development phase, and then you need Mock mock tests that provide various mock components to complete the tests. Spring Boot provides a @MockBean annotation that defines Mock tests based on Mockito for Bean components in Spring. It can create a new Bean to override an existing Bean in the Spring environment, It can be used on test classes, member variables, or @Configuration configuration classes, member variables, and the mocked bean is automatically reset at the end of each test. Suppose you now have a remote service userService that cannot be invoked locally, and now you perform a Mock test as shown in the following usage example. 1 2 3 4 5 6 7 8 9 10 11 12 @SpringBootTest class MockBeanTests { // @Autowired // private UserService userService; @MockBean private UserService userService; @Test public void countAllUsers() { BDDMockito.given(this.userService.countAllUsers()).willReturn(88); assertThat(this.userService.countAllUsers()).isEqualTo(88); } } The @MockBean annotation here is used on the UserService variable to indicate that this userService instance is covered by Mock in the current test case. If there are multiple beans to be mocked, you can use the @Qualifiifier annotation to specify them, and then create the mocked return data through the proxy tool class method provided by Mockito. The test will pass when the mocked data matches the expected result. Here we simulate userService#countAllUsers method by BDDMockito tool class and let it return the total number of users counted (88), and finally check if the return value of the method is as expected. The unit tests pass, and you can also use the @SpyBean annotation instead of the @MockBean annotation, the difference between the two is: @SpyBean - If no Mockito proxy method is provided, the real bean will be called to get the data. @MockBean - The Mock bean will be called to get the data whether or not a Mockito proxy method is provided. The @MockBean and @SpyBean annotations can be used in both Mock and real environments. They are only used to simulate and replace the specified bean in the environment, but cannot be used to simulate the behavior of the bean during the application context refresh, because the application context has been refreshed when the test case is executed, so it is impossible to simulate it again. It is recommended to use the @Bean method to create a mock configuration. Reference: JUnit framework is an excellent choice for writing and executing unit tests and integration tests for any Java application. With Spring Boot 3, it comes inbuilt as part of spring-boot-starter-test module. In this Spring boot tutorial, we will learn to configure JUnit 5 and to write unit tests. By default, the latest spring-boot-starter-test dependency imports the JUnit 5 dependencies into the Spring boot application. The JUnit versions have changed with the following Spring Boot releases: Prior to Spring Boot v2.2.0, by default, JUnit 4 was imported. From Spring Boot v2.2.0 to Spring Boot 2.4.0, JUnit 5 was included by default, and the JUnit-vintage module was included for backward compatibility. [Release Notes v2.2] Since Spring Boot 2.4.x, the JUnit 5 is included without the vintage engine. [Release Notes v2.4] You are highly encouraged to migrate JUnit 4 tests to JUnit 5. But if you still need time, then you can import the vintage dependency exclusively. We are using the latest Spring boot version (at the time of updating this article) i.e. v3.1.2. It includes JUnit 5 framework, by default. org.springframework.boot spring-boot-starter-parent 3.1.2 ... org.springframework.boot spring-boot-starter-test test ... Now we can start writing the test classes and methods using the JUnit 5 annotations. For example, we can use @SpringBootTest to load the Spring Boot application context for integration testing. import org.junit.jupiter.api.Test; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; @SpringBootTest public class MyServiceTest { @Autowired private MyService myService; @Test public void testSomeMethod() { // Your JUnit 5 test logic here // You can use assertions and other JUnit 5 features } } The tests are written in the /src/test/java/ directory in a suitable package hierarchy. Spring boot maven plugin does an excellent job of simplifying the process of running tests and provides various testing-related features. For example, it captures and displays the test output, including test results and any exceptions thrown during testing. We can run the JUnit 5 tests as we would with any IDE or build tool such as Maven. mvn test Let us see an example of JUnit 5 tests written in a Spring Boot 3 application. Here is the Spring boot REST controller for which we will be writing unit tests. @RestController @RequestMapping(path = "/employees") public class EmployeeController { @Autowired private EmployeeDAO employeeDao; @GetMapping(path="/", produces = "application/json") public Employees getEmployees() { return employeeDao.getAllEmployees(); } @PostMapping(path= "/", consumes = "application/json", produces = "application/json") public ResponseEntity addEmployee(@RequestBody Employee employee) { employeeDao.addEmployee(employee); URI location = ServletUriComponentsBuilder.fromCurrentRequest() .path("/{id}") .buildAndExpand(employee.getId()) .toUri(); return ResponseEntity.created(location).build(); } } We will use the @WebMvcTest annotation, which is used for a Spring MVC test that focuses only on Spring MVC components. It disables full auto-configuration and instead applies only configuration relevant to MVC tests. It also configures the Spring Security and MockMvc. @WebMvcTest(EmployeeController.class) public class TestEmployeeRESTController { @Autowired private MockMvc mvc; } Finally, use MockMvc bean instance to invoke the APIs and verify the results. @Test public void getAllEmployeesAPI() throws Exception { mvc.perform(MockMvcRequestBuilders .get("/employees") .accept(MediaType.APPLICATION_JSON)) .andDo(print()) .andExpect(status().isOk()) .andExpect(MockMvcResultMatchers.jsonPath("$.employees").exists()) .andExpect(MockMvcResultMatchers.jsonPath("$.employees[*].employeeId").isNotEmpty()); } Similarly, we can write the other tests. This tutorial discusses how to include JUnit 4 or JUnit 5 dependencies in a Spring Boot application. It discusses how and where to create these tests. And finally, we saw a demo of JUnit 5 tests in action. Happy Learning !! Source Code on Github Spring Boot's testing support is one of its greatest strengths, and the spring-boot-starter-test dependency bundles everything you need to get started with writing robust tests for your application. In this post, we'll explore what this starter brings to the table and focus on three key libraries that it provides: JUnit, Mockito, and Hamcrest.The spring-boot-starter-test starter is a curated dependency that includes several libraries to simplify the testing of Spring Boot applications. It provides integration with the Spring TestContext framework along with several third-party libraries that help you write and execute unit and integration tests.When you include this starter in your project, you automatically get:JUnit – for writing and executing test casesMockito – for creating mock objects and stubbing dependenciesHamcrest – for writing expressive assertionsAdditionally, the starter bundles other useful libraries such as AssertJ, JSONassert, and the Spring TestContext framework. However, if you're asked to name three primary dependencies provided by this starter, the answer is JUnit, Mockito, and Hamcrest.JUnit is the foundation of testing in the Java ecosystem. With spring-boot-starter-test, you get support for JUnit Jupiter (JUnit 5) out of the box. Moreover, the @SpringBootTest annotation in Spring Boot automatically integrates with JUnit 5, so you don't need to manually add any additional JUnit extension annotations.@SpringBootTest public class ApplicationTests { @Test void contextLoads() { // Simple test to ensure the Spring context loads successfully } } The @SpringBootTest annotation is a composite annotation that brings together several essential pieces for integration testing in Spring Boot:@BootstrapWith(SpringBootTestContextBootstrapper.class):This instructs Spring Boot to use a special bootstrapper to load the application context, ensuring that the test environment is set up properly.@ExtendWith(SpringExtension.class):Although not explicitly required with JUnit 5 when using @SpringBootTest, this meta-annotation integrates Spring's testing support with JUnit Jupiter. It automatically configures the Spring TestContext Framework.@ContextConfiguration:@SpringBootTest leverages context configuration to load your application's configuration classes, making sure that the Spring context is appropriately set up for tests.Mockito is a powerful mocking framework that lets you simulate the behavior of complex dependencies. This is especially useful for isolating the unit of work you're testing and for verifying interactions between components.@ExtendWith(MockitoExtension.class) public class UserServiceTest { @Mock private UserRepository userRepository; @InjectMocks private UserService userService; @Test public void testFindUser() { when(userRepository.findByUsername("john")).thenReturn(new User("john", "password")); User user = userService.findUser("john"); assertNotNull(user); verify(userRepository).findByUsername("john"); } } Hamcrest provides a set of matcher objects that allow you to write more expressive and readable assertions. Instead of writing verbose assertions, you can use Hamcrest's matchers to improve the clarity of your tests.import static org.hamcrest.MatcherAssert.assertThat; import static org.hamcrest.Matchers.*; @SpringBootTest public class UserControllerTest { @Test public void testUserName() { String actualUsername = "john.doe"; assertThat(actualUsername, is(equalTo("john.doe"))); assertThat(actualUsername, not(emptyString())); } } To include spring-boot-starter-test in your project, simply add the following dependency in your Maven pom.xml: org.springframework.boot spring-boot-starter-test test dependencies { testImplementation 'org.springframework.boot:spring-boot-starter-test' }By doing so, you ensure that all the key testing libraries, including JUnit, Mockito, and Hamcrest, are available for your tests.The spring-boot-starter-test dependency is a powerful toolkit that enables efficient and effective testing of your Spring Boot applications. The three key libraries it provides—JUnit for writing tests, Mockito for mocking dependencies, and Hamcrest for expressive assertions—form the backbone of a solid testing strategy. By leveraging these tools, you can write tests that are not only robust but also easy to read and maintain. The Spring Boot Starter Test dependency is a primary dependency for testing the Spring Boot Applications. It holds all the necessary elements required for the testing. In Spring Boot, there are lots of tests that can be performed over a Spring Boot application. To perform these tests we have to integrate the testing framework. The beauty of the Spring Boot framework is that for any tasks we just have to put a single dependency in our pom.xml file, instead of writing too many lines of boilerplate code. So, for the testing, we just need to put the spring-boot-starter-dependency in our Spring Boot Application. In this section of the learning Spring Boot Series, we will create a simple Spring Boot Application using the Spring Initializr and perform a Junit test on our Spring Boot application. Let's see the schema of Spring Boot Starter Test dependency: spring-boot-starter-test: org.springframework.boot spring-boot-starter-test 2.4.2 test We can always get the latest version of the Spring Boot Starter Test dependency from Here. The above schema will automatically fetch all the required dependencies and jars for the testing. After adding this schema, we will be able to perform a simple unit test for our Spring Boot application. We can either create a Spring Boot project or generate it using the easiest way Spring Initializr. If we want to add the test dependency manually, we will add the above schema to the bottom of the pom.xml file under the tag. The scope of the test declared like test is available for when the application is bundled and packaged for deployment, it will ignore any dependency that is declared with the test scope. The test scope dependencies are only used when running in the development and Maven test modules. The Spring Boot Test dependencies are packaged with a Spring Boot project when we bootstrap it. It contains the test dependency in the pom.xml file and ApplicationNameTest.java file under the src/test/java. So, we do not need to put it manually in our Spring Boot project. Let's understand it with a simple web application: Before diving into our main topic, let's have a quick look at the Unit Testing: What is Unit Testing in Spring Boot Unit Testing is a type of software testing in which the individual units of a software component are tested. The main purpose of performing the unit test is to ensure whether every unit of the project is performing as expected or not. The Unit Test operation is performed by the developers during the software development phase. Unit test takes a piece of code and validate its correctness. An individual unit may be a function, method, module, project, or object. Let's understand how to perform unit tests in a Spring Boot project using the Spring Boot Starter Test. How to Perform Unit Test in a Spring Boot Project We are going to create a basic Spring Boot project and perform the test operation over it. Follow the below steps to perform a unit test in a Spring Boot application: Step1: Create a Web Project In this tutorial, we are using the Spring Initializr to generate the Spring Boot project. But, you can use any of the methods to create a Spring Boot Project. Open Spring Initializr. Step2: Provide the Project details The Next step is to provide the project details, in our example, we are providing the Group name as com.javasterling.testing and Artifact as SpringBootTestingDemo. You can choose the same or your desired project description. Step3: Add the Spring Web Dependency Since we are developing a web application so make sure to add the Spring Web dependency. Step4: Generate the Project Now, Generate the Project by clicking on the Generate option. Our project details will look like as follows: Step5: Extract the Project The above steps will download a jar file of the project. Now, extract this project to your Spring Boot workspace. Step6: Import the Project into IDE Now, import this project into your favorite IDE such as Eclipse or STS (Spring Tool Suite). We are using the STS, but, it will work the same in other IDEs. To import in STS, navigate to File-> Open project from External File System or File-> Import-> Existing Maven Projects and browse the directory and select it. It will take some time to import the jar files. See How to Import project in Spring Tool Suite and Eclipse. After importing the project, we can see our project directory by exploring the project. It will look something as follows: We can see by default our application contains the follwing files and data: SpringBootTestingDemoApplication.java (src/main/java): package com.javasterling.testing.springBootTestingDemo; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; @SpringBootApplication public class SpringBootTestingDemoApplication { public static void main(String[] args) { SpringApplication.run(SpringBootTestingDemoApplication.class, args); } } SpringBootTestingDemoApplicationTest.java (src/test/java): package com.javasterling.testing.springBootTestingDemo; import org.junit.jupiter.api.Test; import org.springframework.boot.test.context.SpringBootTest; @SpringBootTest class SpringBootTestingDemoApplicationTests { @Test void contextLoads() { } } In this section, Our main focus is on the Test file so let's understand the code of the SpringBootTestingDemoApplicationTest.java file. As we can see from the above code, by default, there are two annotations in the above code; @SpringBootTest, and @Test. @SpringBootTest The @SpringBootTest annotation applies on a Test Class that runs Spring Boot based tests. It provides the following features and functionality to our Spring Boot project: By default, it implements SpringBootContextLoader as the default ContextLoader if no specific context loader is defines. It searches for a @SpringBootConfiguration when nested @Configuartion is not used, and no explicit classes are specified. It enables the support for different WebEnvironment modes. It registers a TestRestTemplate or WebTestClient bean for use in web tests that are using the webserver. It enables the application arguments to be defined using the args attribute. Step7: Perform the Unit Testing Now, our final step is to perform the Unit testing. To perform the unit testing, open the SpringBootTestingDemoApplicationTest.java, right-click and run it as Junit Test. When we run it as Junit Test, it will start our Spring Boot Application: And, displays the following Output: Conclusion: Hence, we can see how to use Spring Boot Starter Test Dependency to test our Spring Boot Applications. However, there is no need to add the spring-boot-starter-test dependency manually because the Spring Boot by default packaged it. Download the Above Example by clicking on the below Download button: springBootTestingDemoDownload See Also,Spring Boot StartersSpring Boot Starter WebSpring Boot Starter ParentSpring Boot Starter Data JPASpring Boot DependenciesSpring Boot Annotations