Spring Boot - Starter Test

Last Updated : 23 Jul, 2025

Spring Boot is built on top of the spring and contains all the features of spring. It is becoming a favorite of developers these days because of its rapid production-ready environment, which enables the developers to directly focus on the logic instead of struggling with the configuration and setup. Spring Boot is a microservice-based framework, and making a production-ready application in it takes very little time. Some of the features of Spring Boot are given below:

- It allows avoiding heavy configuration of XML, which is present in spring.
- It provides easy maintenance and creation of REST endpoints.
- It includes an embedded Tomcat server.
- Deployment is very easy; war and jar files can be easily deployed in the tomcat server

Introduction to Testing in Spring Boot

Spring Boot provides a starter dependency, spring-boot-starter-test, which includes essential testing libraries such as **JUnit 5 (Jupiter), AssertJ, Mockito,** and **Testcontainers** for integration testing.

This article demonstrates how to set up and run tests effectively using modern Spring Boot 3.x practices.

Example

In this example, we will create a spring boot project and add spring-boot-starter-test dependency, and run predefined tests of the application class.

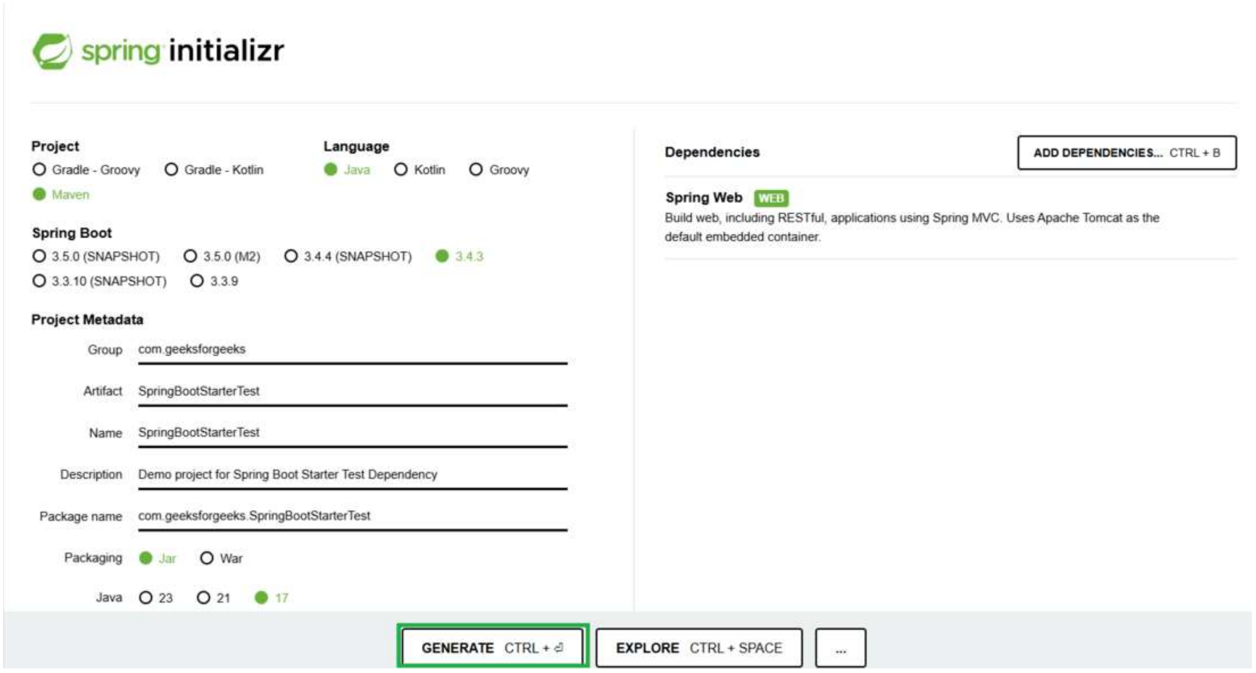
Step-by-Step Implementation

Step 1: Create a Spring Boot Project

In this step, we will create a spring boot project. For this, we will use Spring Initializr and once we create the project we will import it into our Eclipse IDE.

- Spring Boot Version: 3.x
- Java Version: 17
- Dependencies: Spring Web, Spring Boot Starter Test (This dependency comes inside the Spring Web dependency)

Below is the screenshot is given for the spring boot project configuration. Once you are done with your configuration click on Generate button, it will automatically generate a zip file for our project.



Project: Gradle - Groovy, Gradle - Kotlin, [✓ Maven]

Language: [✓ Java], Kotlin, Groovy

Spring Boot: 3.5.0 (SNAPSHOT), 3.5.0 (M2), 3.4.4 (SNAPSHOT), [✓ 3.4.3], 3.3.10 (SNAPSHOT), 3.3.9

Project Metadata

Group	com.geeksforgeeks
Artifact	SpringBootStarterTest
Name	SpringBootStarterTest
Description	Demo project for Spring Boot Starter Test Dependency
Package name	com.geeksforgeeks.SpringBootStarterTest
Packaging	[✓ Jar] War
Java	23 21 [✓ 17]

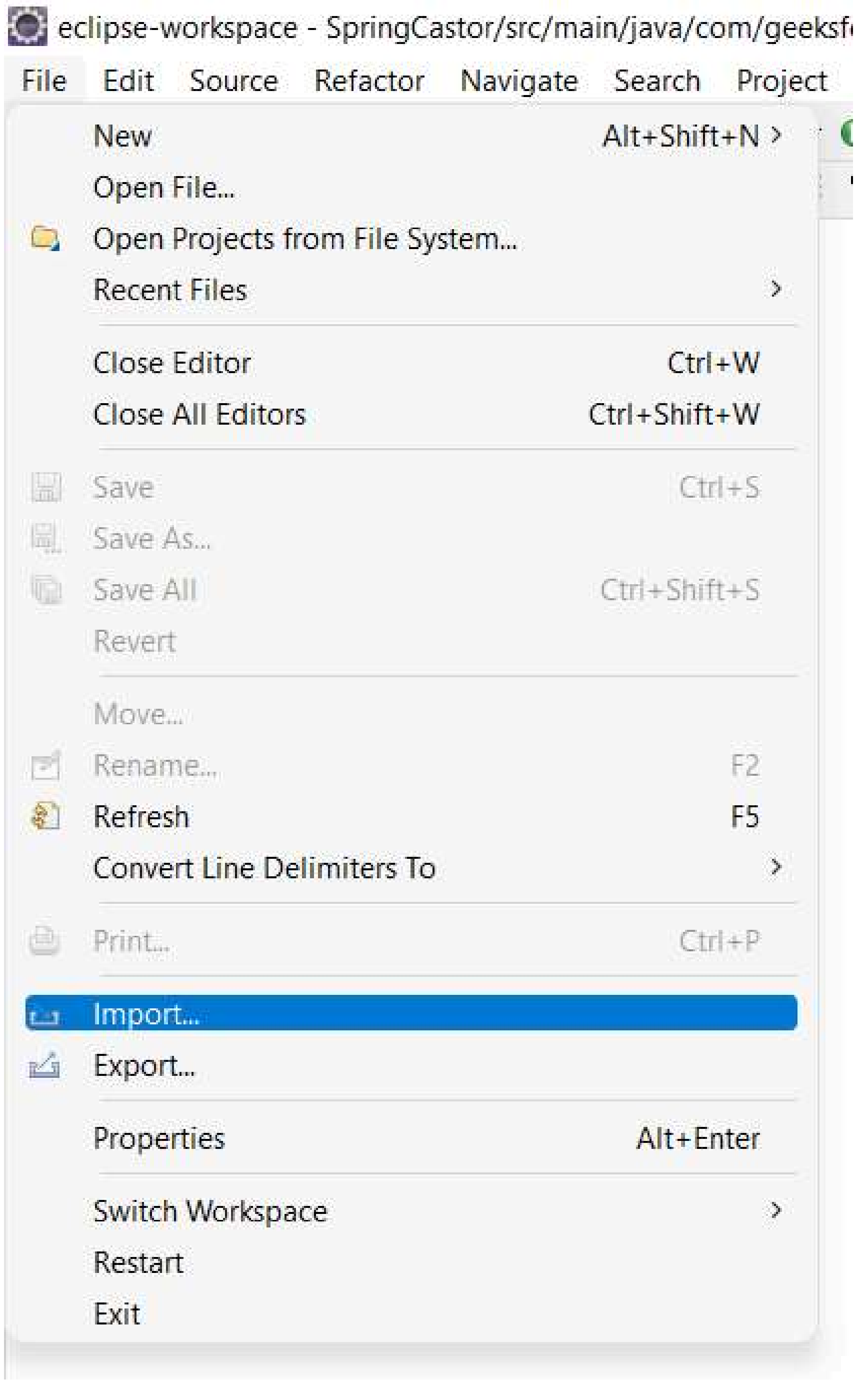
Dependencies

Spring Web WEB

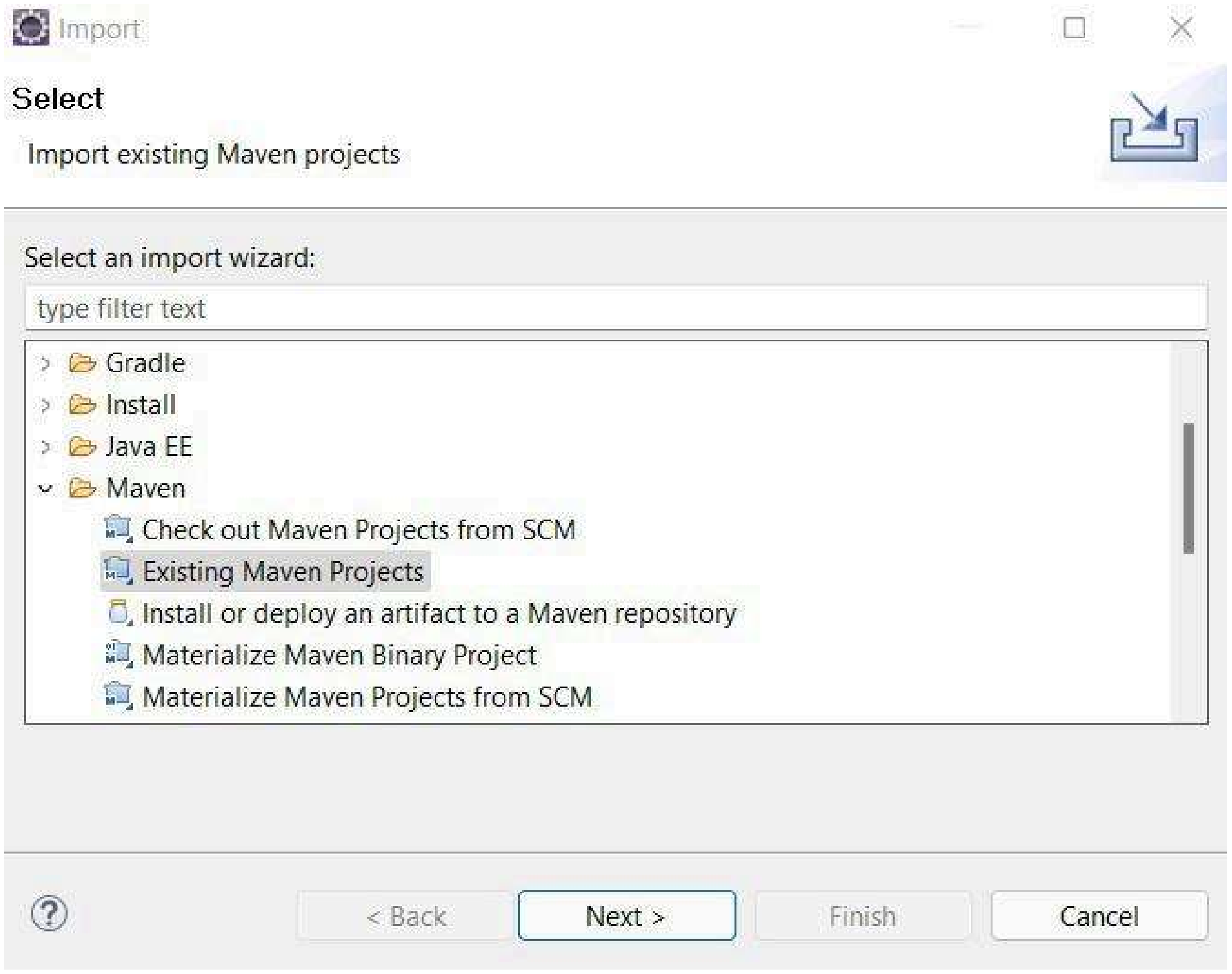
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Step 2: Import Spring Boot Project in Eclipse IDE

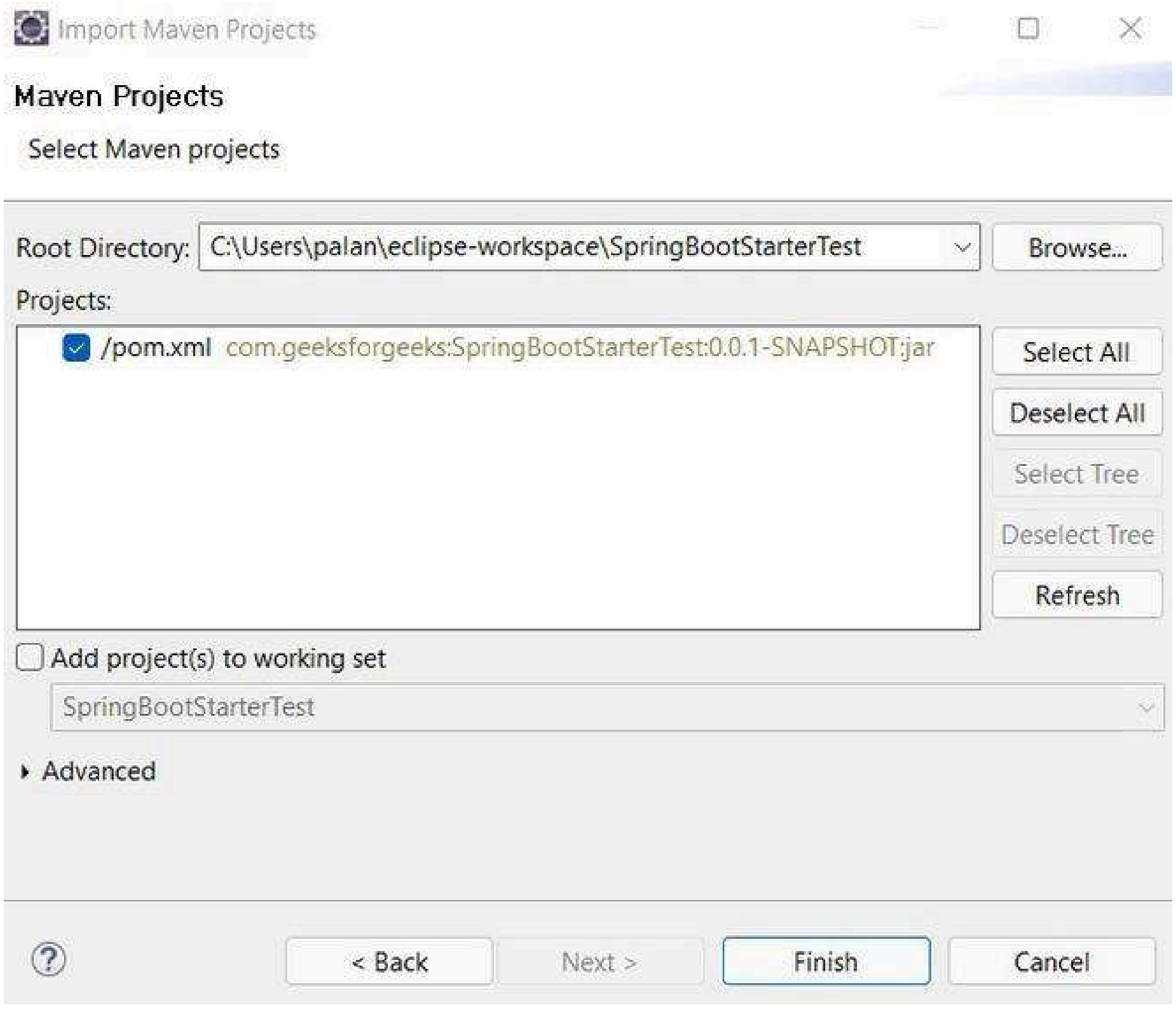
Once we will have the zip file for our project we will import it into our IDE. For this Go to File > Import.



After this go to Maven Projects > Existing Maven Project then click on the Next button.


Import
Select
Import existing Maven projects
Select an import wizard:
type filter text
Gradle
Install
Java EE
Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository
Materialize Maven Binary Project
Materialize Maven Projects from SCM
< Back
Next >
Finish
Cancel

Now we need to browse to the project location and select the project directory and click on the Finish button.



Root Directory: C:\Users\palan\eclipse-workspace\SpringBootStarterTest

Projects:

✓ /pom.xml com.geeksforgeeks:SpringBootStarterTest:0.0.1-SNAPSHOT:jar

☐ Add project(s) to working set

SpringBootStarterTest

Step 3: Adding Dependencies

Once we will create a spring boot project and import it into our IDE, we will find there is a file **pom.xml** that holds the dependencies of the project. In the pom.xml, there is already a dependency is present for the spring boot starter test.

Note: Ensure that the **spring-boot-starter-test** dependency is present in your pom.xml.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
```

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!-- Testcontainers for integration testing -->
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>junit-jupiter</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

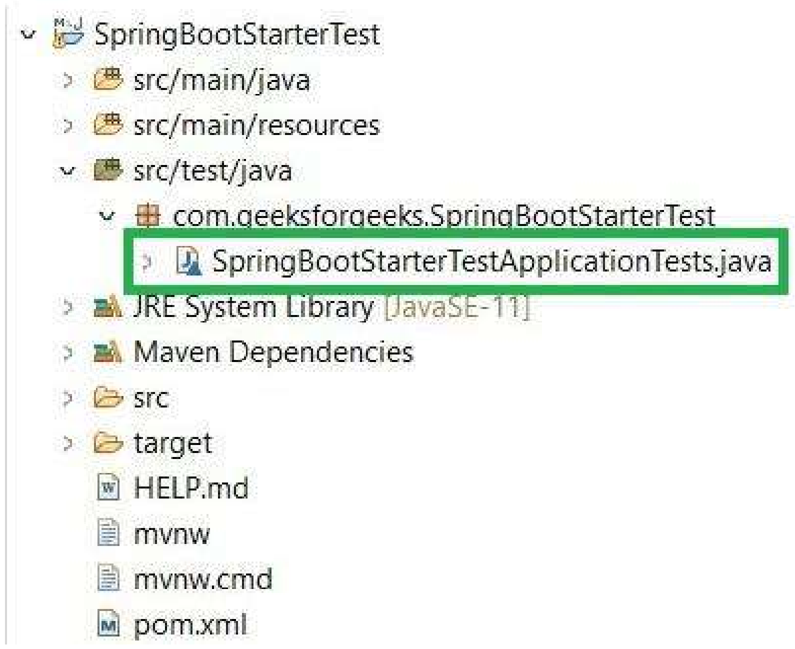
Above, we can look for spring-boot-starter-test dependency, there is **<scope>test</scope>**. It means the scope of the test is only up to the development and test mode.

- JUnit 5 (Jupiter) is included by default in Spring Boot 3.x.
- Testcontainers is added for integration testing.

Once the application is developed and bundled and packaged, all the test scope dependencies are ignored.

Step 4: SpringBootStarterTestApplicationTests Class

After the dependencies are added and the application is successfully built. We can look is our project root path, we will find classes for the test. In our case it is named "**SpringBootStarterTestApplicationTests**". It is present at the **src/test/java** root path.



SpringBootStarterTest
- src/main/java
- src/main/resources
- src/test/java
  - com.geeksforgeeks.SpringBootStarterTest
    - SpringBootStarterTestApplicationTests.java
- JRE System Library [JavaSE-11]
- Maven Dependencies
- src
- target
- HELP.md
- mvnw
- mvnw.cmd
- pom.xml

Step 5: Running the Test Class

In this step, we will run our SpringBootStarterTestApplicationTests class. The default test class contains a simple test method annotated with **@SpringBootTest** and **@Test**. Below is the predefined code for our test class:

```java
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

// @SpringBootTest annotation is
// used to mark a standard Spring test
@SpringBootTest
class SpringBootStarterTestApplicationTests {

    // @Test annotation marks a
    // method as a test method
    @Test
    void contextLoads() {
    }
}
```

In the above, spring boot by default has provided two annotations:

- @SpringBootTest: This annotation is marked over the test class that runs the standard spring boot tests. The @SpringBootTest annotation has the following features over the Spring Test Context framework.

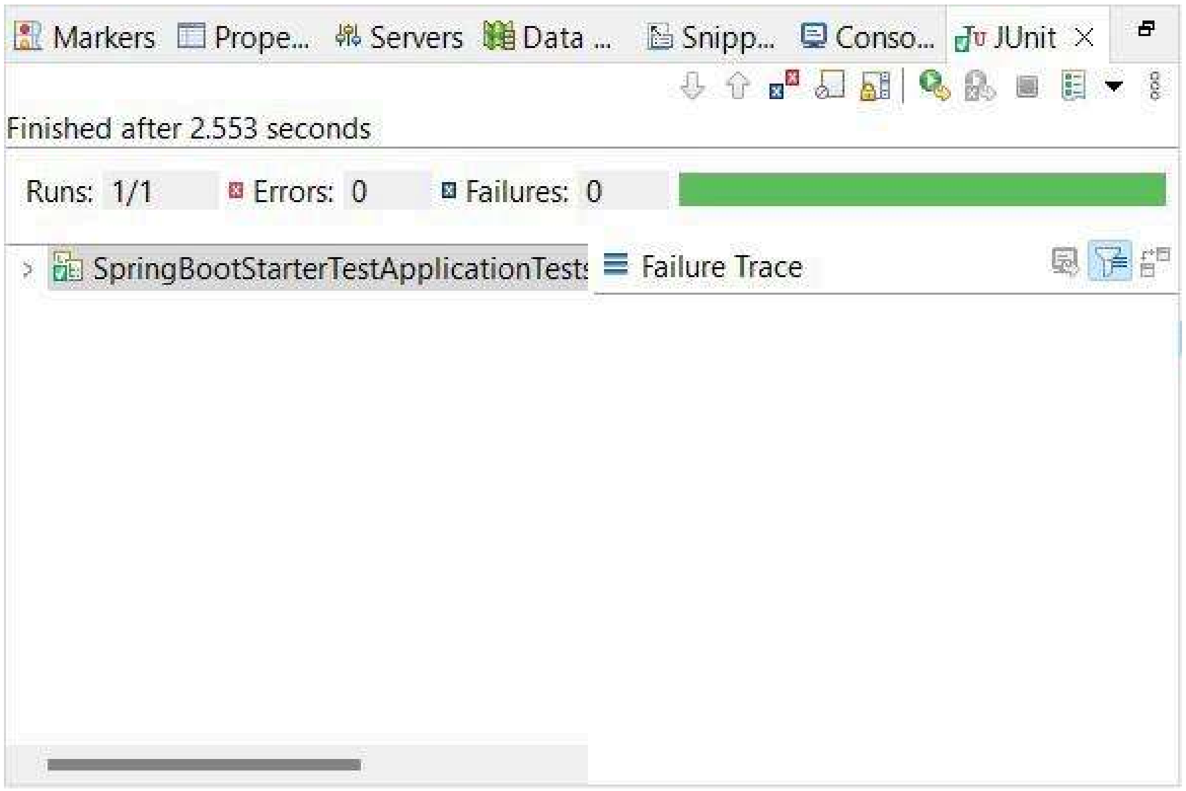
- If @ContextConfiguration(loader="") is explicitly defined, It uses the default SpringBootContextLoader.
- It supports various WebEnvironment modes (e.g., MOCK, RANDOM_PORT)
- If an application uses a web server for a web test then it automatically registers for a TestRestTemplate or WebTestClient bean.
- It provides facilities for application arguments to be defined using the args attribute.

- @Test: This annotation marks a method as the test method.

Step 6: Output

In this step, we will run our application.

- To run the test, right-click the test class and select **Run As > JUnit** Test.
- The **contextLoads()** test will execute, verifying that the application context starts successfully.



```
Finished after 2.553 seconds
Runs: 1/1    Errors: 0    Failures: 0
SpringBootStarterTestApplicationTests
```

Additional Best Practices

1. Using @TestInstance(Lifecycle.PER_CLASS)

Use @TestInstance(Lifecycle.PER_CLASS) to create a single instance of the test class for all test methods.

```
import org.junit.jupiter.api.TestInstance;
import org.junit.jupiter.api.TestInstance.Lifecycle;

@TestInstance(Lifecycle.PER_CLASS)
class MyTestClass {
    // Test methods
}
```

2. Using @AutoConfigureMockMvc for Web Tests

Use @AutoConfigureMockMvc for efficient testing of Spring MVC controllers.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;

@SpringBootTest
@AutoConfigureMockMvc
class MyControllerTest {

    @Autowired
    private MockMvc mockMvc;
    // Test methods using mockMvc
}
```

3. Using Sliced Tests for Better Performance

Use sliced tests like @WebMvcTest and @DataJpaTest to test specific layers of the application.

```
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.web.servlet.MockMvc;

@WebMvcTest(MyController.class)
class MyControllerTest {

    @Autowired
    private MockMvc mockMvc;

    // Test methods
}
```

4. Testcontainers for Integration Testing

Use Testcontainers for integration testing with real databases and services.

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;
```

```
@Testcontainers
class MyIntegrationTest {

    @Container
    static PostgreSQLContainer<?> postgres = new
PostgreSQLContainer<>("postgres:13");

    @Test
    void testDatabaseConnection() {
        // Test logic
    }
}
```

5. AssertJ for Fluent Assertions

Use AssertJ for more readable assertions.

```
import static org.assertj.core.api.Assertions.assertThat;

@Test
void testAssertions() {
    String result = myService.doSomething();
    assertThat(result).isEqualTo("expectedValue");
}
```

6. Mockito for Mocking Dependencies

Use Mockito for mocking dependencies in unit tests.

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class MyServiceTest {

    @Mock
    private MyRepository myRepository;

    @InjectMocks
    private MyService myService;

    @Test
    void testServiceMethod() {
when(myRepository.findById(anyLong())).thenReturn(Optional.of(ne
w MyEntity()));
        // Test logic
    }
}
```