FPT
SOFTWARE

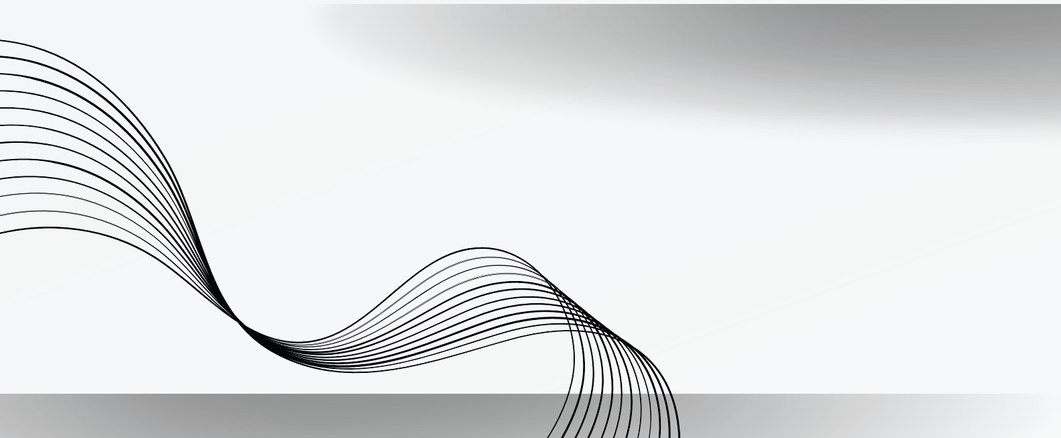
SPRING BOOT
TESTING

KIENNCT3 - MENTOR HAINV21

CONTENT

SPRING BOOT TESTING

- In the context of Spring Boot, testing plays a crucial role in maintaining the quality and reliability of applications.
- It involves the execution of code under controlled conditions to validate that each component functions as intended. This process helps catch bugs early in the development cycle, reducing the likelihood of issues reaching production.

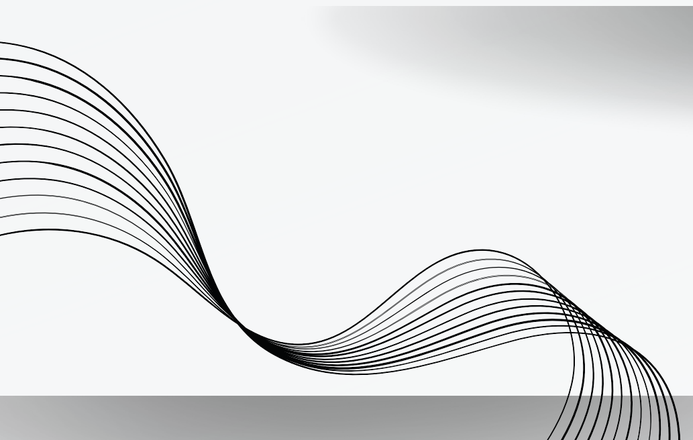


SPRING BOOT TESTING

-Types of Testing in Spring Boot: Unit Testing, Integration Testing, End-to-End (E2E) Testing, etc.

1) **Unit Testing:**

- Purpose: Verify the correctness of individual units or components in isolation.
- Key Features: Use tools like JUnit, TestNG
- Annotations: @Test

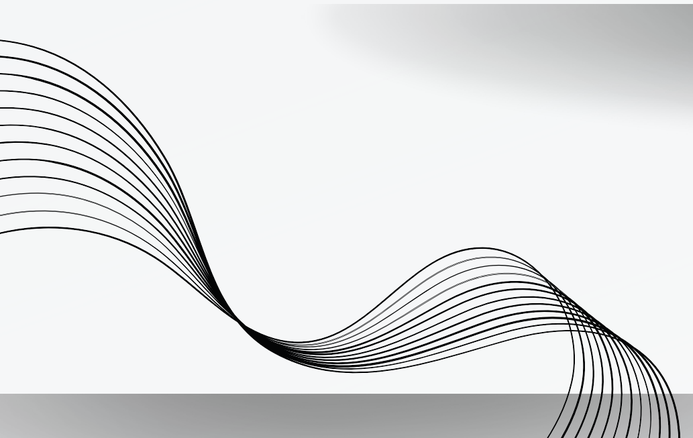


SPRING BOOT TESTING

-Types of Testing in Spring Boot: Unit Testing, Integration Testing, End-to-End (E2E) Testing, etc.

2) **Integration Testing:**

- Purpose: Validate the interactions between different components or services.
- Annotations: @SpringBootTest



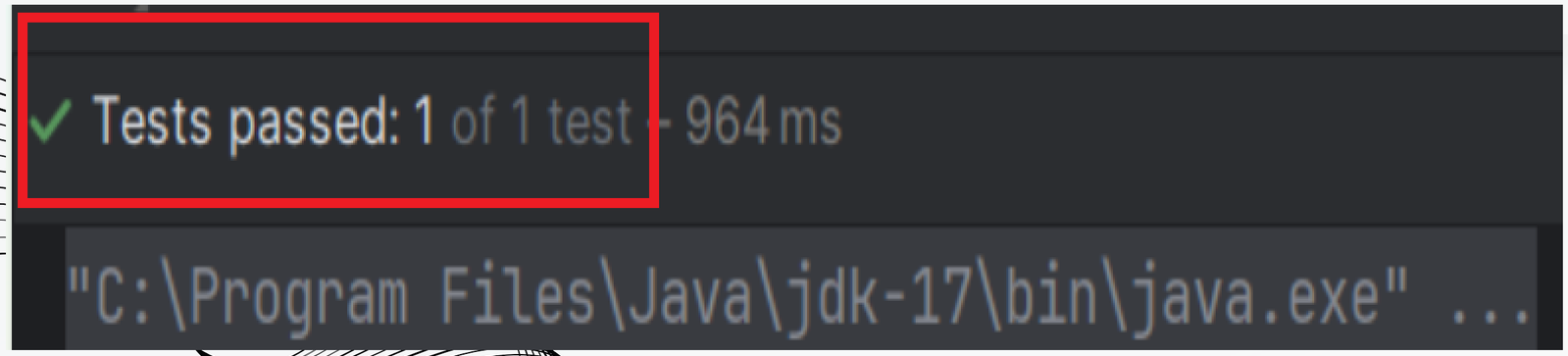
FOR EXAMPLE ABOUT TESTING MULTIPLY OF 2 NUMBERS

```java
public class Calculator {
    1 usage  new *
    public int multiply(int a, int b) {
        return a * b;
    }
}
```

```java
@SpringBootTest
class SpringBootTestingApplicationTests {
    1 usage
    Calculator test = new Calculator();
    new *
    @Test
    void checkMultiply() {
        int a = 20;
        int b = 10;
        assertEquals( expected: 200, test.multiply(a, b));
    }
}
```

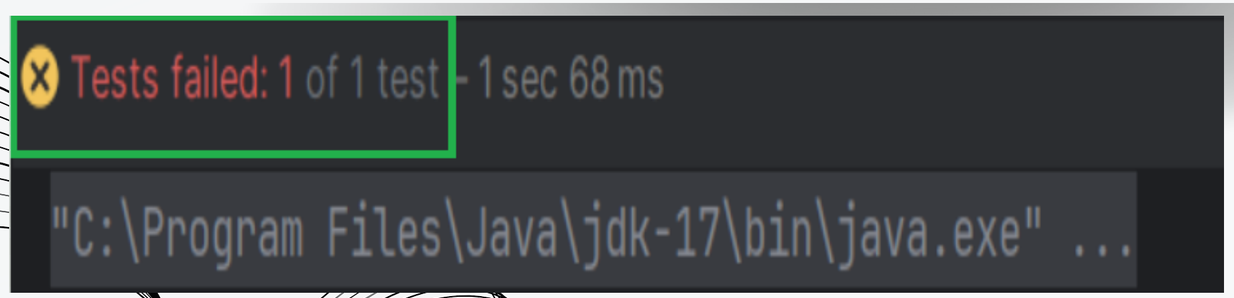
FOR EXAMPLE:

```java
@SpringBootTest
class SpringBootTestingApplicationTests {
    1 usage
    Calculator test = new Calculator();
    new *
    @Test
    void checkMultiply() {
        int a = 20;
        int b = 10;
        assertEquals( expected: 200, test.multiply(a, b));
    }
}
```



FOR EXAMPLE:

```java
@SpringBootTest
class SpringBootTestingApplicationTests {
    1 usage
    Calculator test = new Calculator();
    new *
    @Test
    void checkMultiply() {
        int a = 20;
        int b = 10;
        assertEquals( expected: 300, test.multiply(a, b));
    }
}
```




```
org.opentest4j.AssertionFailedError:
Expected :300
Actual   :200
```

DEPENDENCY MANAGEMENT TESTING

- Spring Boot use Maven or Gradle to manage dependency.
- Dependencies are declared in the pom.xml (Maven) or build.gradle (Gradle) file.
- Spring Boot simplifies dependency management in testing by providing a set of starter dependencies that include commonly used testing libraries.

1) Testing Starter Dependencies:

Spring Boot provides specific starter dependencies for different testing frameworks. These starters include the necessary libraries and configurations to support testing with popular frameworks such as JUnit and TestNG.

DEPENDENCY MANAGEMENT

1) Testing Starter Dependencies:

Spring Boot provides specific starter dependencies for different testing frameworks. These starters include the necessary libraries and configurations to support testing with popular frameworks such as JUnit and TestNG.

JUnit5 Starter:

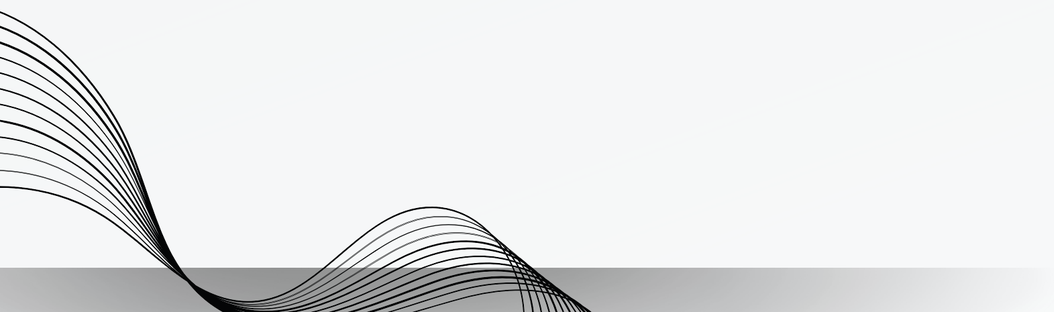
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

DEPENDENCY MANAGEMENT

2) Spring Boot Test Annotations:

Spring Boot provides annotations that simplify testing with Spring components and features. Some commonly used annotations include:

- @SpringBootTest: Loads the entire Spring application context for integration testing.
- @DataJpaTest: Concentrates on testing the data access layer.



JUNIT TESTING FRAMEWORK

1) Overview:

JUnit is one of the most widely used testing frameworks for Java. It provides annotations to define test methods, assertions for testing expected results, and test runners to execute tests

2) Key features:

- Annotations: JUnit uses annotations like @Test to mark methods as test methods, @Before, @After (JUnit 4), and @BeforeEach, @AfterEach (Junit 5) for setup and teardown operations, and more.
- Assertions: JUnit provides a set of assertion methods (assertTrue, assertEquals, etc.) for validating expected results in test methods.

JUNIT TESTING FRAMEWORK

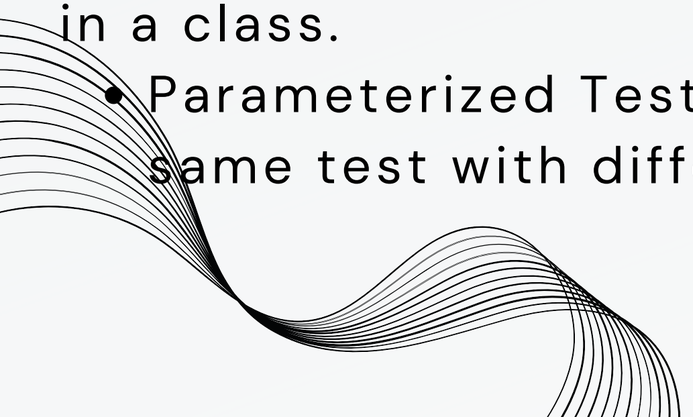
2) Key features:

- Annotations: JUnit uses annotations like @Test to mark methods as test methods, @Before, @After (JUnit 4), and @BeforeEach, @AfterEach (Junit 5) for setup and teardown operations, and more.
- Assertions: JUnit provides a set of assertion methods (assertTrue, assertEquals, etc.) for validating expected results in test methods.
- Test Runners: Tools or components in testing frameworks, responsible for managing and executing test methods in a test suite.

+ JUnit supports test runners like BlockJUnit4ClassRunner and Parameterized for different testing scenarios.

+ BlockJUnit4ClassRunner: Is the default test runner for JUnit 4, runs test methods in a class.

- Parameterized Tests: JUnit supports parameterized tests, allowing to run the same test with different sets of data.

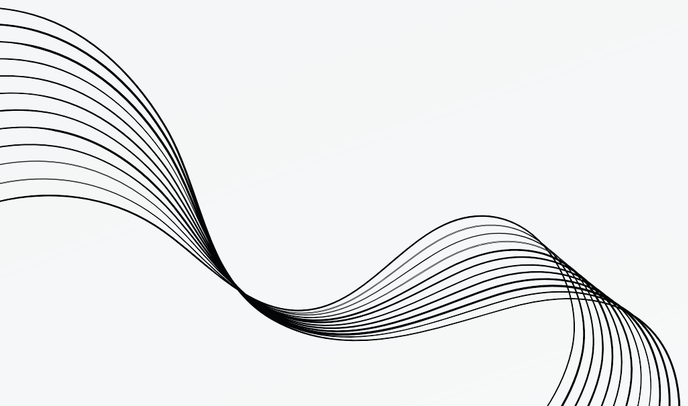


JUNIT JUPITER ANNOTATIONS

Definition:

- JUnit Jupiter annotations are a set of annotations used in JUnit 5 (JUnit Jupiter).

- These annotations help define and configure test methods, test conditions, and operations related to running tests.



JUNIT JUPITER ANNOTATIONS

Definition:

- JUnit Jupiter annotations are a set of annotations used in JUnit 5 (JUnit Jupiter).

- These annotations help define and configure test methods, test conditions, and operations related to running tests.

Important annotations in JUnit Jupiter:

1) @Test:

- Used to mark a method as a test method.

```
@Test
public void checkSum() {
    assertEquals( expected: 3,  actual: 2 + 1);
}
```

JUNIT JUPITER ANNOTATIONS

Important annotations in JUnit Jupiter:

2) @BeforeEach and @AfterEach:

- Used to mark methods that run before (@BeforeEach) and after (@AfterEach) each test method.

```
private Calculator testCal;
new *
@BeforeEach
public void setUp() {
    testCal = new Calculator();
    System.out.println("Prepare for testing: ");
}
new *
@Test
public void checkSum() {
    assertEquals( expected: 6, testCal.multiply( a: 2, b: 3));
}
new *
@AfterEach
public void tearDown() {
    testCal = null;
    System.out.println("Finish testing!");
}
```

JUNIT JUPITER ANNOTATIONS

Important annotations in JUnit Jupiter:

3) @BeforeAll and @AfterAll:

- These annotations are used to indicate methods that should be run before (or after) all test methods in a test class. They must be static methods.

JUNIT JUPITER ANNOTATIONS

Important annotations in JUnit Jupiter:

3) @BeforeAll and @AfterAll:

```java
@BeforeAll
static void setUp() {
    // Initialization code before all test methods
    System.out.println("Before all tests");
}
new *
@Test
void testMethod1() {
    System.out.println("Test Method 1");
}
new *
@Test
void testMethod2() {
    System.out.println("Test Method 2");
}
new *
@AfterAll
static void tearDown() { // Cleanup code after all test methods
    System.out.println("After all tests");
}
```

JUNIT JUPITER ANNOTATIONS

Important annotations in JUnit Jupiter:

4) @DisplayName:

- Used to specify the display name of the test method

```
@Test
@DisplayName("Name of my testing method:")
public void testMultiply() { assertEquals( expected: 200, testCal.multiply( a: 20,  b: 10)); }
```

JUNIT JUPITER ANNOTATIONS

Important annotations in JUnit Jupiter:

5)@ParameterizedTest:

- This annotation is used to indicate that a method is a parameterized test. Parameterized tests allow to run the same test with different sets of parameters.

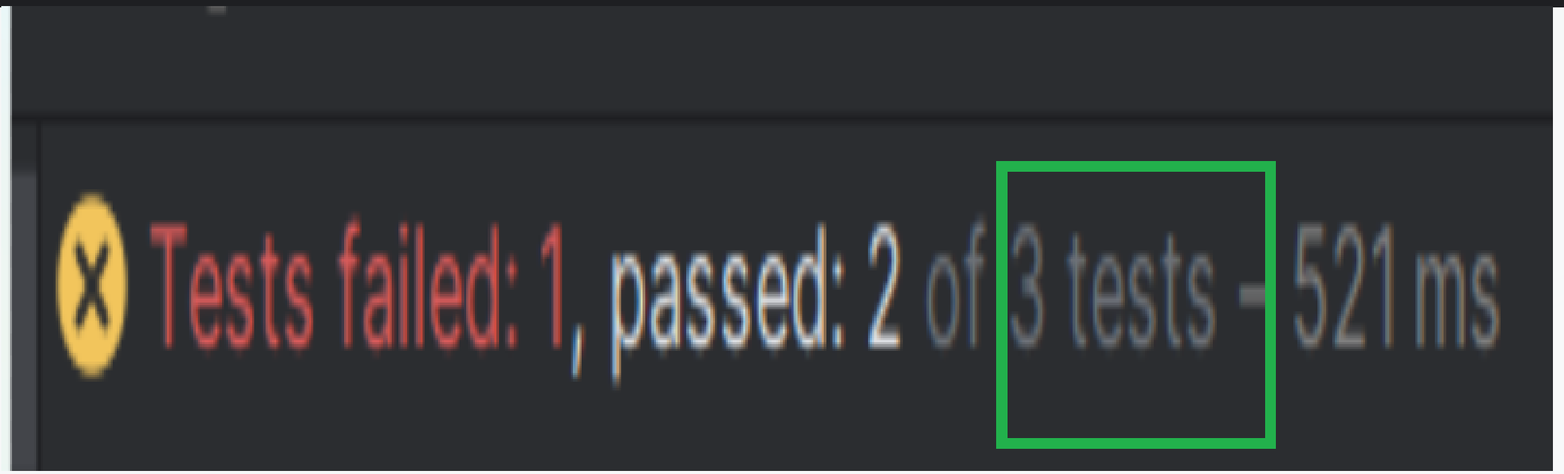
```
@ParameterizedTest
@ValueSource(strings = {"kien trung", "trung dung", "trung nguyen"})
void testNameLength(String name) {
    assertEquals( expected: 10, name.length());
}
```

JUNIT JUPITER ANNOTATIONS

Important annotations in JUnit Jupiter:

5)@ParameterizedTest:

```
@ParameterizedTest
@ValueSource(strings = {"kien trung", "trung dung", "trung nguyen"})
void testNameLength(String name) {
    assertEquals( expected: 10, name.length());
}
```



JUNIT JUPITER ANNOTATIONS

Important annotations in JUnit Jupiter:

6)@Disabled:

- Used to mark a method or test class as disabled. This annotation is useful when we want to temporarily exclude certain tests from being executed.

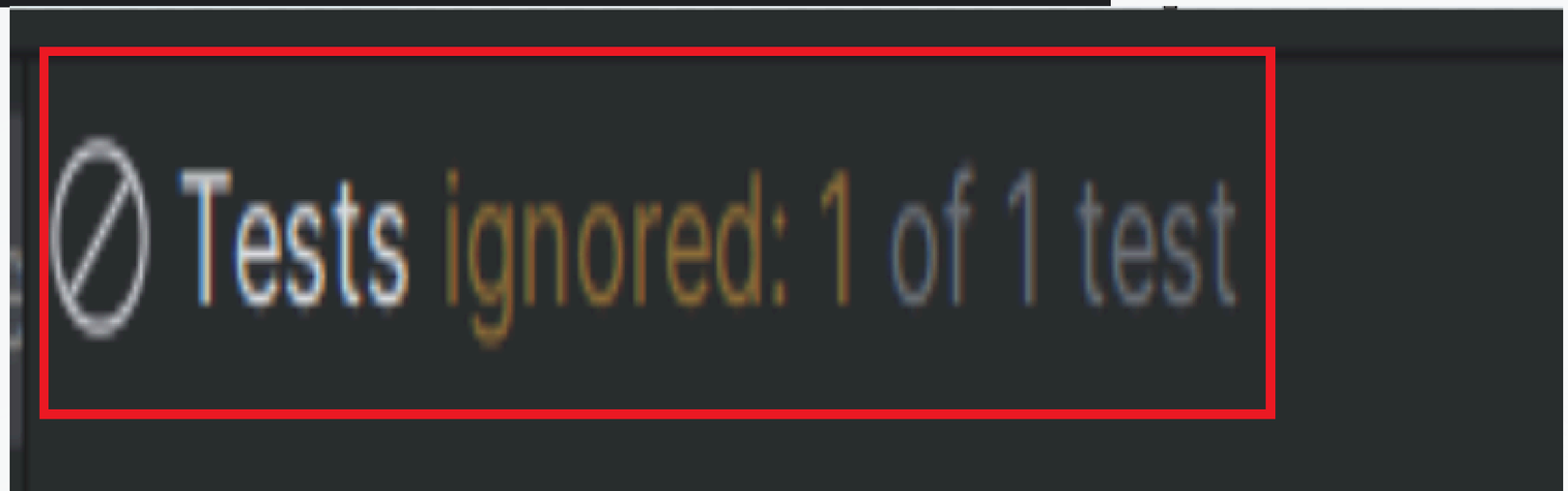
```
@Test
@Disabled
void testNameLength() {
    String name = "Trung Kien";
    assertEquals( expected: 10, name.length());
}
```

JUNIT JUPITER ANNOTATIONS

Important annotations in JUnit Jupiter:

6)@Disabled:

```java
@Test
@Disabled
void testNameLength() {
    String name = "Trung Kien";
    assertEquals( expected: 10, name.length());
}
```



ASSERTION IN JUNIT 5

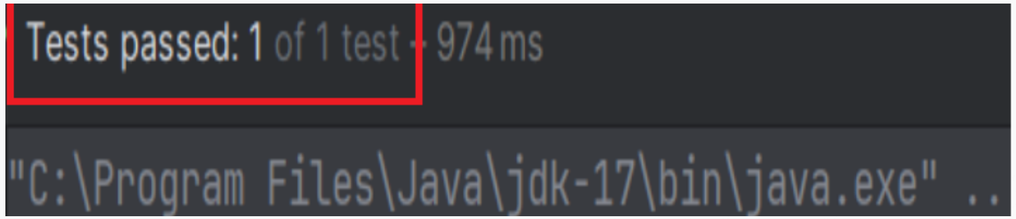
Assertions in Junit5 used to verify that the actual outcome of a test matches the expected result or not

Some commonly used assertions include:

1) assertEquals(expected, actual)

- Asserts that two objects are equal. If they are not, an AssertionError is thrown.

```java
@Test
public void testSubtraction() {
    int res = 5 - 2;
    assertEquals( expected: 3, res);
}
```



1) assertEquals(expected, actual)

- Asserts that two objects are equal. If they are not, an AssertionError is thrown.

```
    @Test
    public void testSubtraction() {
        int result = 5 - 4;
        assertEquals( expected: 2, result);
    }
}
```

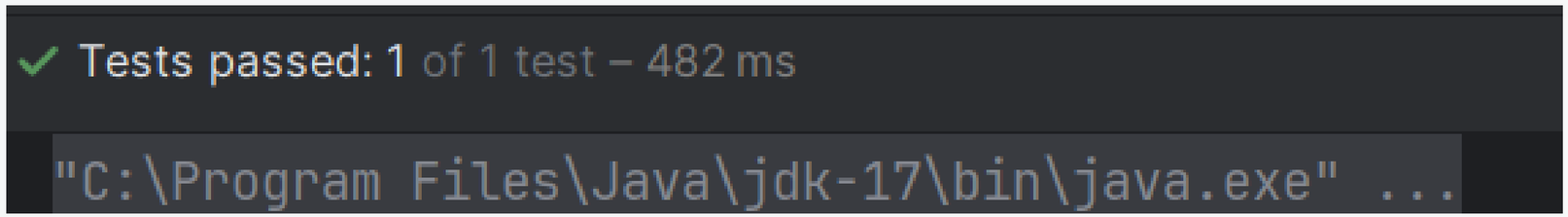
```
Tests failed: 1 of 1 test – 922 ms

org.opentest4j.AssertionFailedError:
Expected :2
Actual   :1
```

2) assertNotEquals(unexpected, actual)

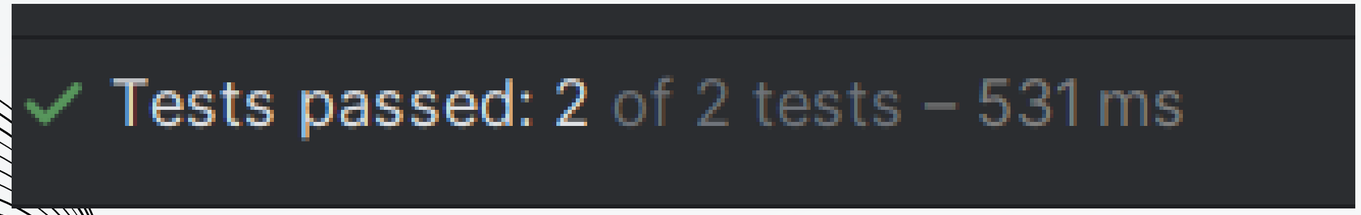
- Asserts that two objects are not equal or not.

```java
@Test
public void testMultiply() {
    int res = 4 * 5;
    assertNotEquals( unexpected: 21, res);
}
```



5) **assertNull and assertNotNull: These assertions check if a value is null or not null.**

```java
@Test
void testAssertNull() {
    String nullString = null;
    assertNull(nullString, message: "The value should be null");
}
new *
@Test
void testAssertNotNull() {
    String nonNullString = "Hello";
    assertNotNull(nonNullString, message: "The value should not be null");
}
```



6) assertArrayEquals: This assertion checks if two arrays are equal or not

```java
@Test
void testArrayEquals() {
    int[] expectedArray = {1, 2, 3};
    int[] actualArray = {1, 2, 3};

    assertArrayEquals(expectedArray, actualArray, message: "The arrays should be equal");
}
```

```
✔ Tests passed: 1 of 1 test – 596 ms

"C:\Program Files\Java\jdk-17\bin\java.exe" ...
```

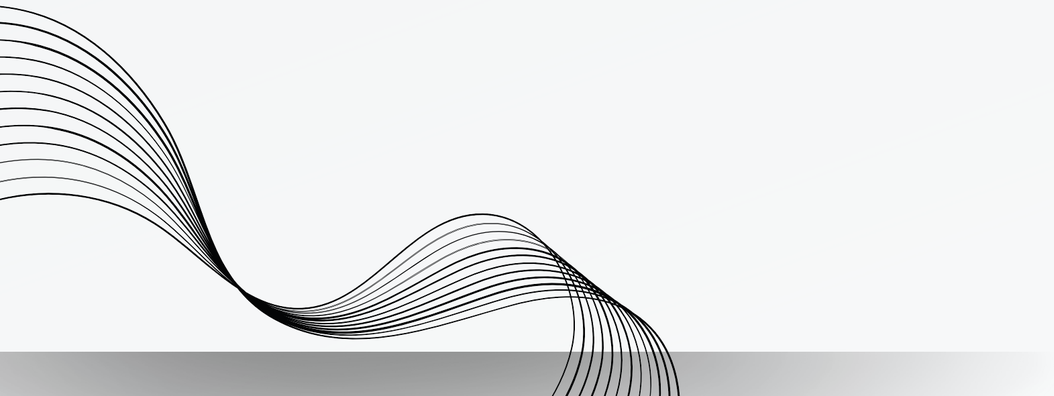
6) assertArrayEquals: This assertion checks if two arrays are equal or not

```java
@Test
void testArrayEquals() {
    int[] expectedArray = {1, 2, 3};
    int[] actualArray = {1, 2, 4};

    assertArrayEquals(expectedArray, actualArray, message: "The arrays should not equal");
}
```

```
Tests failed: 1 of 1 test – 512 ms

org.opentest4j.AssertionFailedError: The arrays should not equal ==> array contents differ at index [2],
Expected :3
Actual   :4
```



7) assertAll: This assertion groups multiple assertions together. It ensures that all assertions are executed, and any failures are reported together.

```java
@Test
void testAssertAll() {
    String name = "Trung Kien";
    int age = 20;
    assertAll( heading: "Person",
            () -> assertEquals( expected: "Kien Trung", name,  message: "Name does not match!"),
            () -> assertEquals( expected: 21, age,  message: "Age does not equal!")
    );
}
```

```
Tests failed: 1 of 1 test – 585 ms

org.opentest4j.AssertionFailedError: Name does not match! ==>
Expected :Kien Trung
Actual   :Trung Kien
```

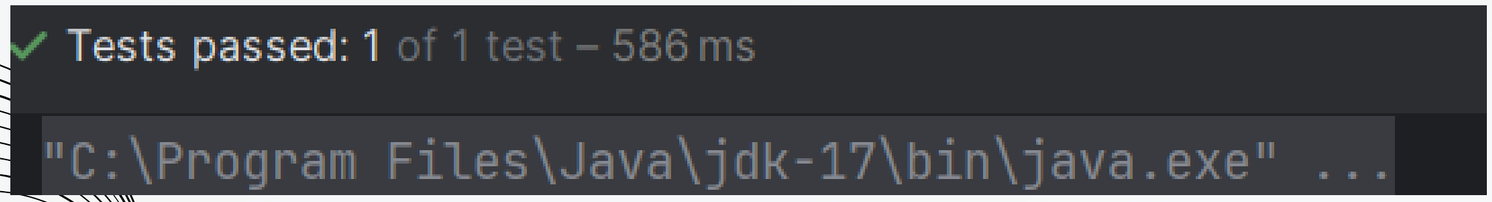
```java
@Test
void testAssertAll() {
    String name = "Trung Kien";
    int age = 20;
    assertAll( heading: "Person",
            () -> assertEquals( expected: "Kien Trung", name,  message: "Name does not match!"),
            () -> assertEquals( expected: 21, age,  message: "Age does not equal!")
    );
}
```

```
Tests failed: 1 of 1 test – 585 ms
org.opentest4j.AssertionFailedError: Name does not match! ==>
Expected :Kien Trung
Actual   :Trung Kien
```

```
Tests failed: 1 of 1 test – 585 ms
org.opentest4j.AssertionFailedError: Age does not equal! ==>
Expected :21
Actual   :20
```

8) assertThat (optional): This assertion is a part of AssertJ library

```java
import static org.assertj.core.api.AssertionsForClassTypes.assertThat;
new *
@SpringBootTest
class SpringBootTestingApplicationTests {
    new *
    @Test
    public void testName() {
        String text = "Trung Kien";
        assertThat(text).hasSize( expected: 10).startsWith("Trung").endsWith("n");
    }
}
```



8) assertThat (optional): This assertion is a part of AssertJ library

```java
import static org.assertj.core.api.AssertionsForClassTypes.assertThat;
new *
@SpringBootTest
class SpringBootTestingApplicationTests {
    new *
    @Test
    public void testName() {
        String text = "Trung Kien";
        assertThat(text).hasSize( expected: 10).startsWith("Trung").endsWith("K");
    }
}
```

Tests failed: 1 of 1 test – 558 ms

```
java.lang.AssertionError:
Expecting actual:
  "Trung Kien"
to end with:
  "K"
```

MOCKITO TESTING FRAMEWORK

1) Overview:

Mockito is a popular framework in the Java world, used to create and manage mock objects during testing. Fake objects are objects created to simulate the behavior of real objects in the system, to test and control interactions between components.

2) Key features:

- Mock Creation: Mockito allows the creation of mock objects using the mock() method. Mocks simulate the behavior of real objects without executing their actual code.
- Stubbing: Developers can use Mockito to define the behavior of mock objects using the when().thenReturn() syntax. This is known as stubbing and is used to set up the expected responses from the mock.

EXAMPLE

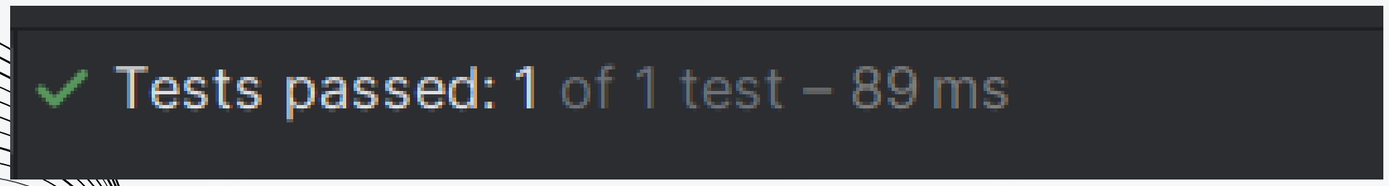
```
public class Calculator {
    2 usages  new *
    public int multiply(int a, int b) { return a * b; }
```

```
public class MathService {
    1 usage
    Calculator cal = new Calculator();
    1 usage  new *
    public int performMultiply(int x, int y) {
        return cal.multiply(x, y);
    }
```

```java
public class MathService {
    1 usage
    Calculator cal = new Calculator();
    1 usage  new *
    public int performMultiply(int x, int y) {
        return cal.multiply(x, y);
    }
```

```java
    @Mock
    private Calculator cal;
    1 usage
    @InjectMocks
    private MathService ms;
    new *
    @Test
    public void testPerformMultily() {
        when(cal.multiply( a: 10,  b: 20)).thenReturn( t: 200);
        int res = ms.performMultiply( x: 10,  y: 20);
        assertEquals( expected: 200, res);
    }
```

```java
@Mock
private Calculator cal;
1 usage
@InjectMocks
private MathService ms;
new *
@Test
public void testPerformMultily() {
    when(cal.multiply( a: 10,  b: 20)).thenReturn( t: 200);
    int res = ms.performMultiply( x: 10,  y: 20);
    assertEquals( expected: 200, res);
}
```



```java
@Test
public void testPerformMultily() {
    when(cal.multiply( a: 10,  b: 20)).thenReturn( t: 200);
    int res = ms.performMultiply( x: 10,  y: 20);
    assertEquals( expected: 5, res);
}
```

```
Tests failed: 1 of 1 test – 86 ms

org.opentest4j.AssertionFailedError:
Expected :5
Actual   :200
```

TEST COVERAGE

1) Overview:

"Test coverage" is an index used to evaluate the level of testing of source code in a software project. It measures the percentage of source code that has been executed ("caught" by test methods) compared to the total source code.

2) The importance:

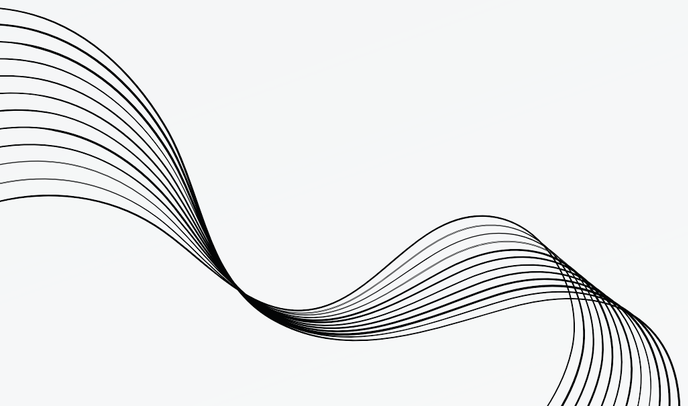
- Quality assurance: Test coverage helps ensure that important parts of the source code have been tested and protected against errors.
- Find errors early: Helps detect errors right from the early development stage, reducing the cost of fixing errors later.
- Increased reliability: The more the source code is tested, the more reliable the application.

TEST COVERAGE

3) Coverage percentage: Was calculated by formula
Coverage percentage = (number of tested codepaths/total codepaths) * 100%

EXAMPLE

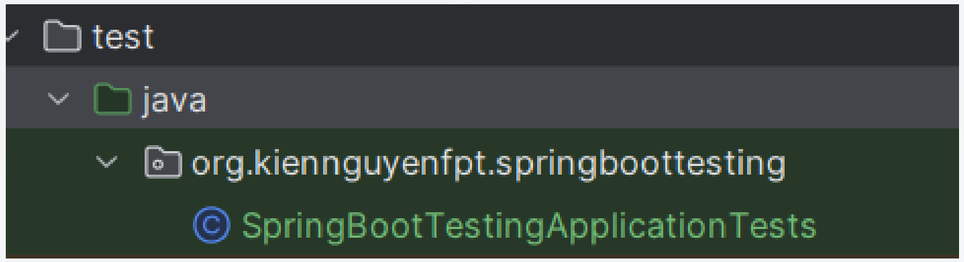
```
3 usages  new *
public String classifyNumber(int number) {
    if (number > 0) {
        return "Positive";
    } else if (number < 0) {
        return "Negative";
    } else {
        return "Zero";
    }
}
```

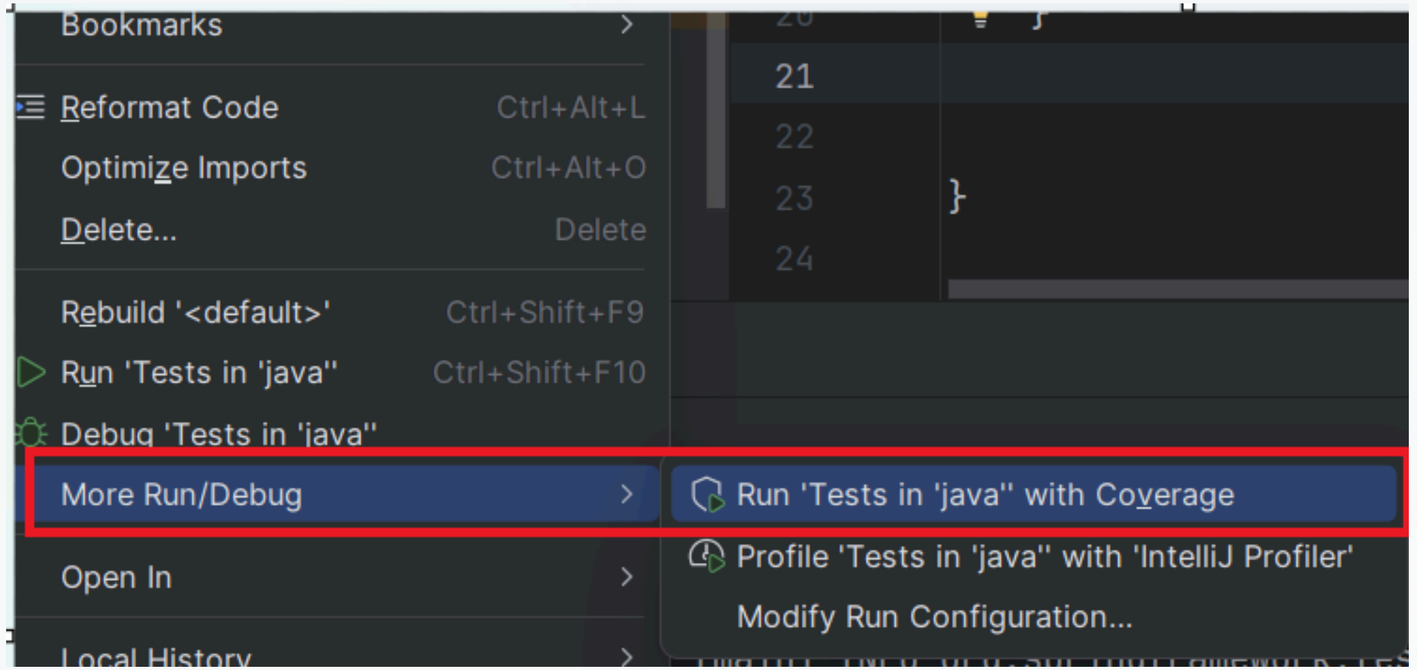


- We have total 3 code paths in this method
- Supposed that we have tested 2 code paths. Then:

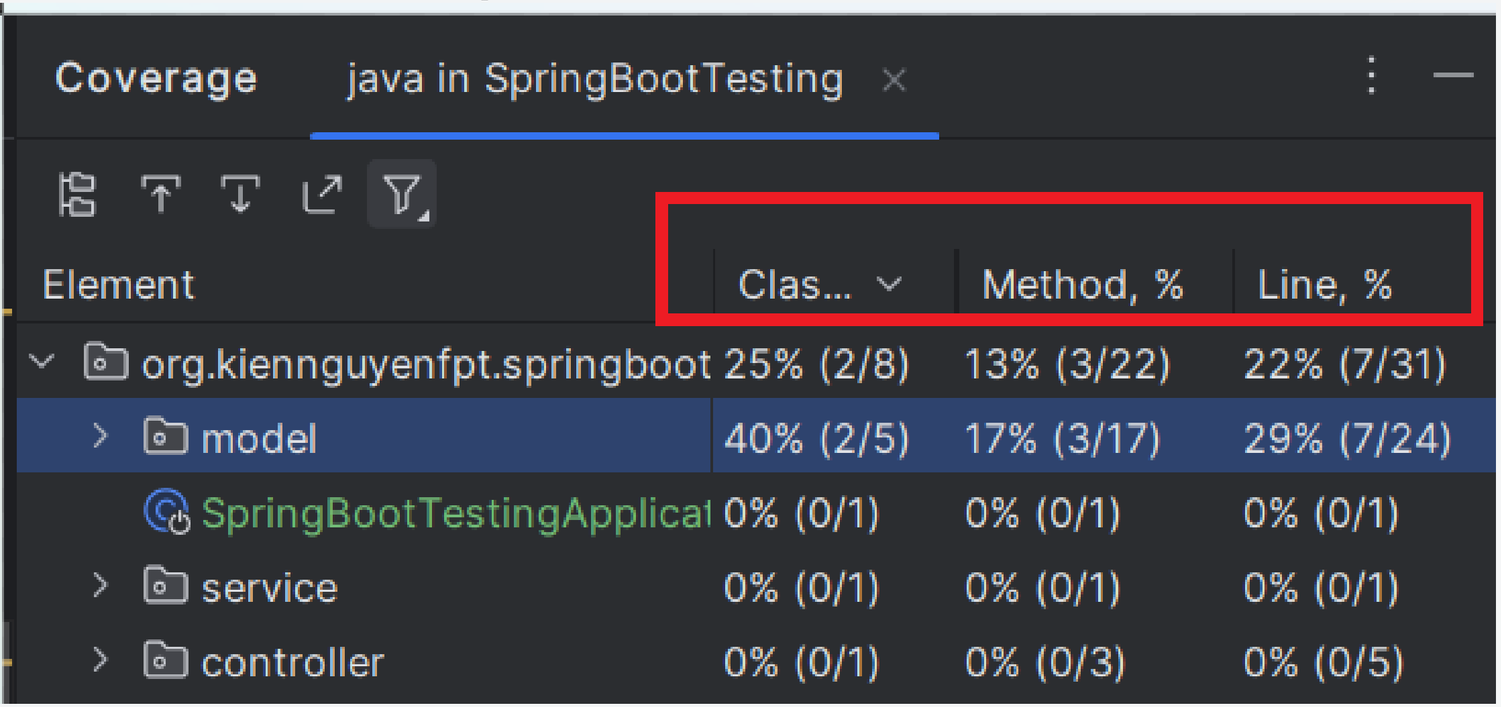
Percentage coverage = (2/3) * 100% = 66.67%

CALCULATE PERCENTAGE COVERAGE IN INTELLIJ





Here is the output:



Class %: The percentage of classes in project that have been executed by our tests.

Method %: The percentage of methods in classes that have been executed by our tests.

Line %: The percentage of lines of code that have been executed by our tests.

```java
public class MathService {
    1 usage
    Calculator cal = new Calculator();
    no usages  new *
    public int performMultiply(int x, int y) {
        return cal.multiply(x, y);
    }

    3 usages  new *
    public String classifyNumber(int number) {
        if (number > 0) {
            return "Positive";
        } else if (number < 0) {
            return "Negative";
        } else {
            return "Zero";
        }
    }
```

Element	Clas...	Method, %	Line, %
org.kiennguyenfpt.springboot	25% (2/8)	13% (3/22)	22% (7/31)
model	40% (2/5)	17% (3/17)	29% (7/24)
MathService	100% (1/1)	50% (1/2)	83% (5/6)
ConfigurationProperty	100% (1/1)	50% (2/4)	50% (2/4)
AppConfig	0% (0/1)	0% (0/1)	0% (0/1)
Calculator	0% (0/1)	0% (0/2)	0% (0/2)
Book	0% (0/1)	0% (0/8)	0% (0/11)
SpringBootTestingApplica	0% (0/1)	0% (0/1)	0% (0/1)
service	0% (0/1)	0% (0/1)	0% (0/1)
controller	0% (0/1)	0% (0/3)	0% (0/5)

EXAMPLE TESTING WITH EXTERNALIZED CONFIGURATION

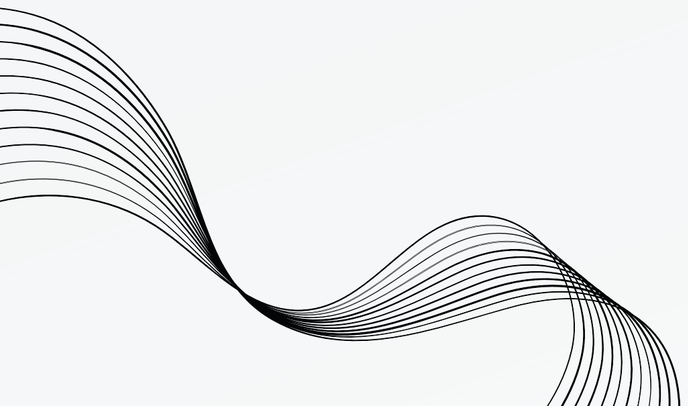
application.yml

```yaml
name:
    person: Trung Kien
```

AppConfig.java

```java
@Configuration
public class AppConfig {
    @Value("${name.person}")
    private String name;

    1 usage  new *
    public String getName() {
        return name;
    }
}
```



EXAMPLE TESTING WITH EXTERNALIZED CONFIGURATION

application.yml × AppConfig.java

```yaml
name:
    person: Trung Kien
```

```java
@SpringBootTest
class SpringBootTestingApplicationTests {

    @Autowired
    private AppConfig app = new AppConfig();
    new *
    @Test
    public void testName() {
        String actualName = app.getName();
        assertEquals( expected: "Trung Kien", actualName);

    }
}
```

EXAMPLE TESTING WITH EXTERNALIZED CONFIGURATION

application.yml × AppConfig.java

```yaml
name:
    person: Trung Kien
```

```java
@SpringBootTest
class SpringBootTestingApplicationTests {

    @Autowired
    private AppConfig app = new AppConfig();
    new *
    @Test
    public void testName() {
        String actualName = app.getName();
        assertEquals( expected: "Trung Kien", actualName);
```

Tests passed: 1 of 1 test – 1 sec 90 ms

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
```

EXAMPLE TESTING WITH EXTERNALIZED CONFIGURATION

application.yml × AppConfig.java

```yaml
name:
    person: Trung Kien
```

```java
@Autowired
private AppConfig app = new AppConfig();
new *
@Test
public void testName() {
    String actualName = app.getName();
    assertEquals( expected: "Trung Dung", actualName);
```

```
org.opentest4j.AssertionFailedError:
Expected :Trung Dung
Actual   :Trung Kien
```

EXAMPLE TESTING WITH API

```java
public class Book {
    3 usages
    private int id;
    3 usages
    private String bookName;
    3 usages
    private String author;
```

```java
4 usages  1 implementation  new *
public interface BookService {
    2 usages  1 implementation  new *
    List<Book> findAllBooks();
}
```

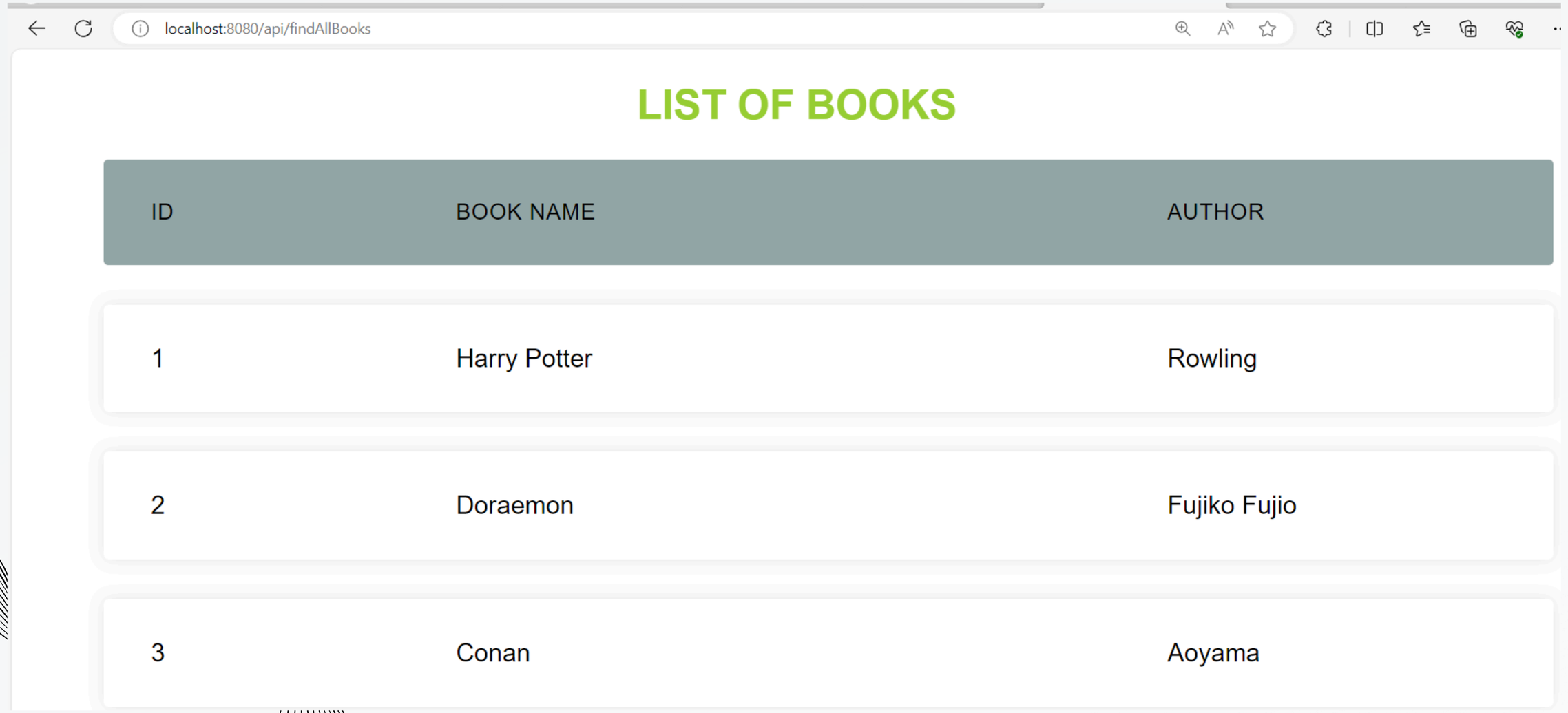
```java
public class BookServiceImp implements BookService{
    2 usages  new *
    @Override
    public List<Book> findAllBooks() {
        return Arrays.asList(new Book( id: 1,  bookName: "Harry Potter",  author: "Rowling"),
                             new Book( id: 2,  bookName: "Doraemon",  author: "Fujiko Fujio"),
                             new Book( id: 3,  bookName: "Conan",  author: "Aoyama"));
    }
}
```

EXAMPLE TESTING WITH API

```java
@RestController
@RequestMapping("api")
public class HomeController {
    2 usages
    private final BookService bs = new BookServiceImp();
    new *
    @GetMapping("/findAllBooks")
    public ModelAndView index(org.springframework.ui.Model model){
        List<Book> res = bs.findAllBooks();
        model.addAttribute( attributeName: "bookAttribute",res);
        return new ModelAndView( viewName: "book");
    }
```

EXAMPLE TESTING WITH API

Here is the output on localhost:



EXAMPLE TESTING WITH API

Testing method findAllBooks() in HomeControler class:

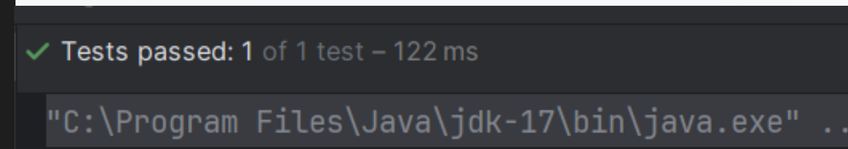
```
@Mock
private BookService bs;
1 usage
@InjectMocks
private HomeController hc;
new *
@Test
public void testFindAllBooks() {
    List<Book> mockBooks = Arrays.asList(
            new Book( id: 1,  bookName: " aaa Harry Potter",  author: "Rowling"),
            new Book( id: 2,  bookName: "Doraemon",  author: "Fujiko Fujio"),
            new Book( id: 3,  bookName: "Conan",  author: "Aoyama")
    );
    when(bs.findAllBooks()).thenReturn(mockBooks);
    Model model = new org.springframework.ui.ConcurrentModel();
    ModelAndView mav = hc.index(model); // Call the index method
    List<Book> actualBooks = (List<Book>) model.asMap().get("bookAttribute");// Retrieve
    assertEquals(mockBooks.get(0).getBookName(), actualBooks.get(0).getBookName());
    assertEquals(mockBooks.get(0).getAuthor(), actualBooks.get(0).getAuthor());
```

```
org.opentest4j.AssertionFailedError:
Expected : aaa Harry Potter
Actual   :Harry Potter
```

EXAMPLE TESTING WITH API

Testing method findAllBooks() in HomeControler class:

```
@Mock
private BookService bs;
1 usage
@InjectMocks
private HomeController hc;
new *
@Test
public void testFindAllBooks() {
    List<Book> mockBooks = Arrays.asList(
            new Book( id: 1, bookName: "Harry Potter", author: "Rowling"),
            new Book( id: 2, bookName: "Doraemon", author: "Fujiko Fujio"),
            new Book( id: 3, bookName: "Conan", author: "Aoyama")
    );
    when(bs.findAllBooks()).thenReturn(mockBooks);
    Model model = new org.springframework.ui.ConcurrentModel();
    ModelAndView mav = hc.index(model); // Call the index method
    List<Book> actualBooks = (List<Book>) model.asMap().get("bookAttribute");// Retrieve
    assertEquals(mockBooks.get(0).getBookName(), actualBooks.get(0).getBookName());
    assertEquals(mockBooks.get(0).getAuthor(), actualBooks.get(0).getAuthor());
```



GREATFUL THANK YOU FOR LISTENING!