Unit Testing and Unit Testing in Spring Boot with JUnit 5 and Mockito

Introduction to Unit Testing

What is Unit Testing?

Unit testing is a software testing method where individual units (smallest testable parts) of an application are tested independently to ensure they work as expected.

Why Unit Tests Are Important

Key Benefits:

- **Early Bug Detection**: Catch errors before they reach production.
- **Improved Code Quality**: Encourages modular, maintainable code.
- **Faster Development**: Reduces manual testing and debugging time.
- **Facilitates Refactoring**: Tests act as a safety net when changing existing code.

Real-World Use Case:

Imagine you're building a banking app — unit tests ensure that a method like withdraw(amount) behaves correctly under different scenarios (enough balance, insufficient balance, etc.).

Purpose:

- Validate correctness of individual components (e.g., service or utility methods).
- Act as documentation for what the code is supposed to do.
- Enable safer refactoring and code changes.

Example in Java:

```
public int add(int a, int b) {
   return a + b;
}

// Test
@Test
void testAdd() {
   assertEquals(5, add(2, 3));
}
```

Java unit testing frameworks, specifically:

- **JUnit** – Most widely used framework for unit testing in Java.
    - Versions: JUnit 4, JUnit 5 (a.k.a. JUnit Jupiter)
    - **JUnit 4**: Annotations like @Test, @Before, @After.

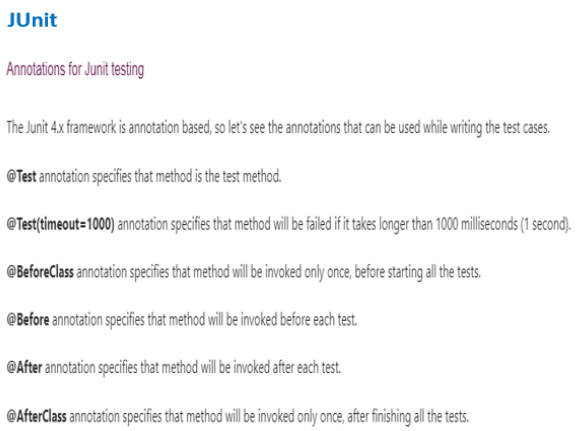
    - **JUnit 5 (Jupiter)**: More modular, flexible, supports @BeforeEach, @DisplayName, etc.

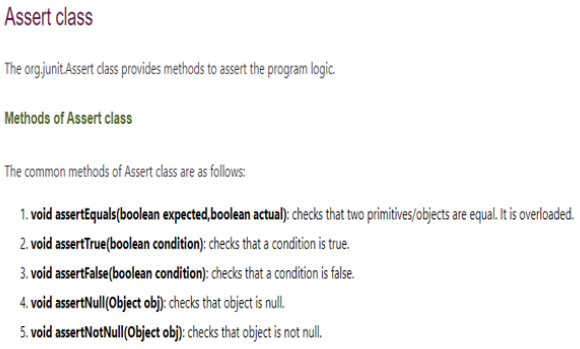
- **TestNG (Test Next Generation)** – Another popular testing framework with additional features like parallel testing.

- **Mockito** – Used for **mocking** dependencies in unit tests (often used with JUnit).
    - **Mock dependencies** in unit tests
    - Verify **interactions** between components

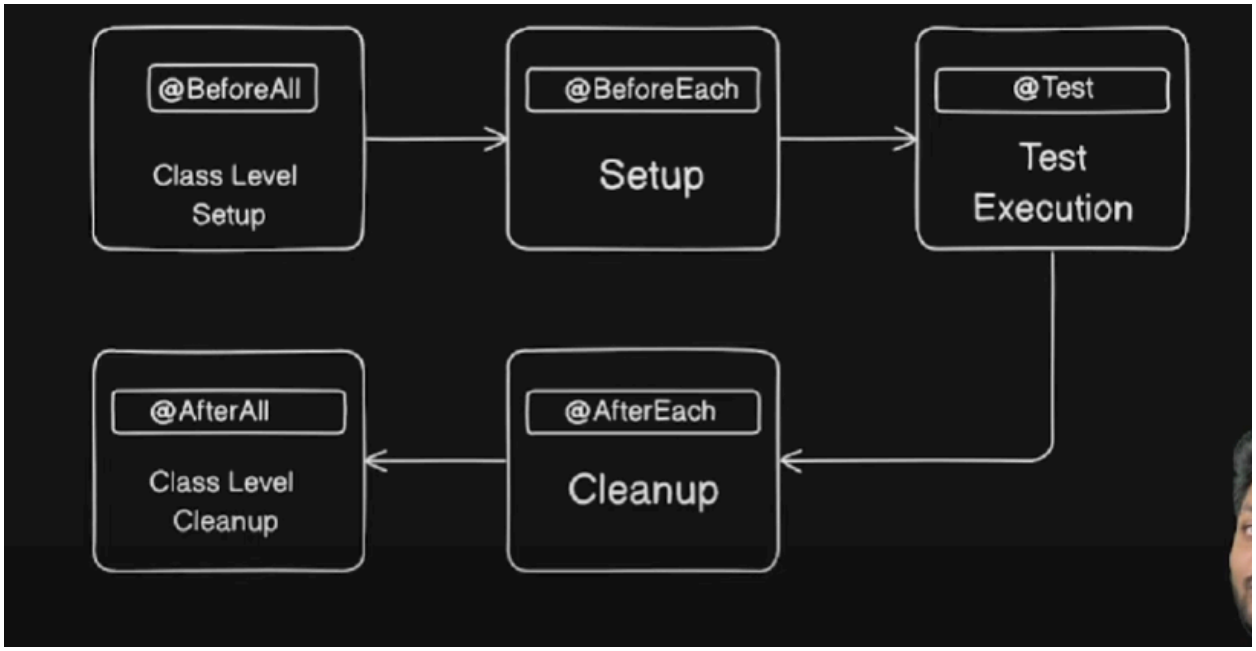
What is JUnit?

JUnit is a **unit testing framework for Java**. It allows developers to write and run tests to ensure their code works as expected. It's part of the **xUnit family** of testing frameworks and is widely used in **test-driven development (TDD)**.





Life Cycle of test methods



Example: Calculator and CalculatorTest

```
package com.junit;
public class Calculator {
	public static void main(String[] args) {
		Calculator cal = new Calculator();
		System.out.println(cal.add(1, 2));
		System.out.println(cal.divide(6, 3));
		System.out.println(cal.getNullValue());
	}
	public int add(int a, int b) {
		return a + b;
	}
	public int divide(int a, int b) {
		return a / b;
	}
	public String getNullValue() {
		return null;
	}
}
3
```

2
null

CalculatorTest.java (JUnit 4 version with all annotations & assertions)

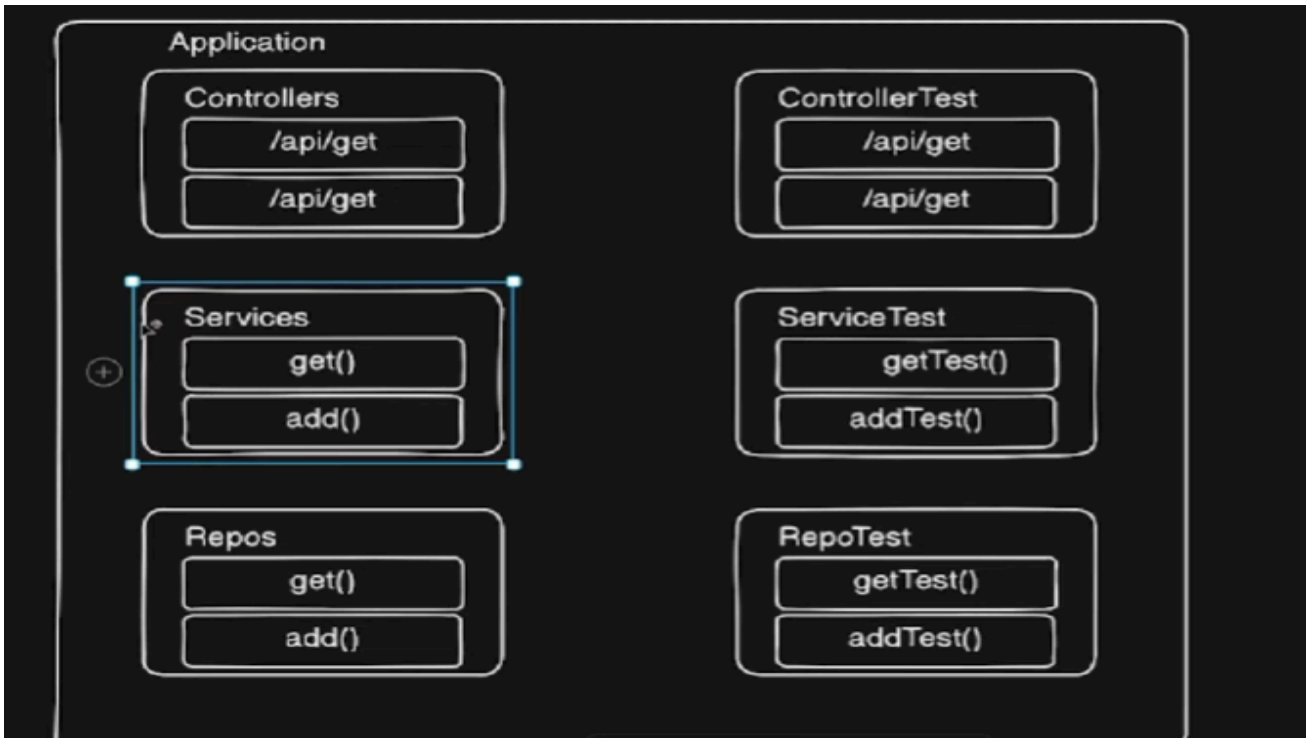
```
package com.junit;
public class CalculatorTest {
  private static Calculator calculator;
  @BeforeClass
  public static void setupBeforeAllTests() {
     System.out.println("BeforeClass: Executed once before all test methods.");
     calculator = new Calculator();
  }
  @Before
  public void setupBeforeEachTest() {
     System.out.println("Before: Executed before each test method.");
  }
  @Test
  public void testAddition() {
     int result = calculator.add(10, 5);
     assertEquals("Addition test failed", 15, result);            // assertEquals
     assertTrue("Result should be greater than 10", result > 10);   // assertTrue
     assertFalse("Result should not be negative", result < 0);      // assertFalse
  }
  @Test(timeout = 1000)
  public void testWithTimeout() {
     // simple operation that completes within 1 second
     int result = calculator.add(3, 2);
     assertEquals(5, result);
  }
  @Test
  public void testNullAndNotNull() {
     String value = calculator.getNullValue();
     assertNull("Expected null value", value);                  // assertNull
     String notNull = "JUnit";
     assertNotNull("Expected non-null value", notNull);          // assertNotNull
  }
  @After
  public void tearDownAfterEachTest() {
     System.out.println("After: Executed after each test method.");
```

```
    }
    @AfterClass
    public static void tearDownAfterAllTests() {
        System.out.println("AfterClass: Executed once after all test methods.");
    }
}
```

Output

BeforeClass: Executed once before all test methods.
Before: Executed before each test method.
After: Executed after each test method.
... (repeated for each test)
AfterClass: Executed once after all test methods.

Getting Started With Unit Testing in a Spring Boot App



To write unit tests in a Spring Boot project, you typically use:

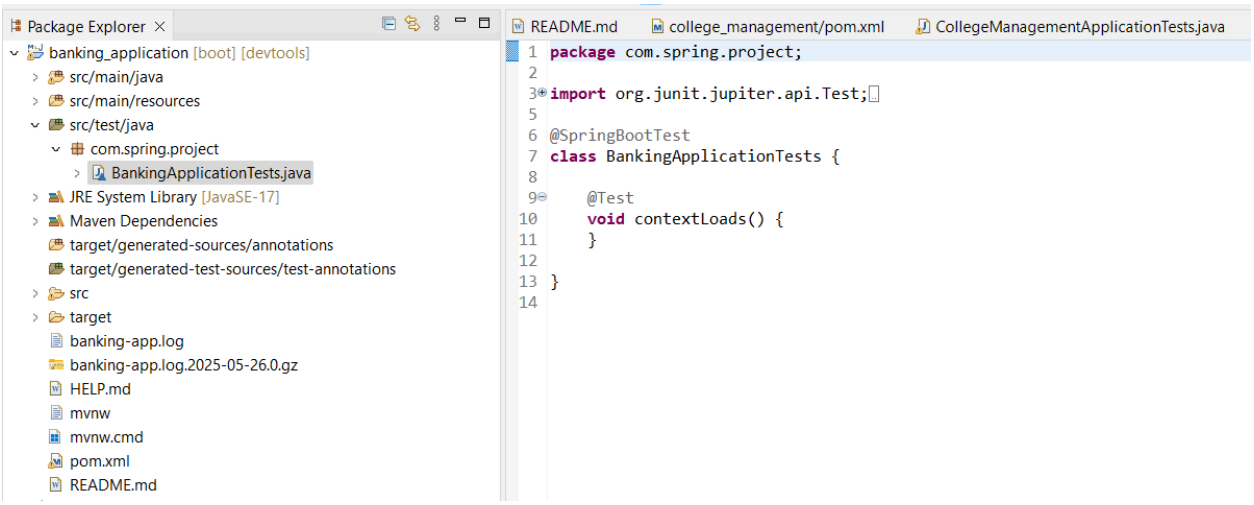
- **JUnit 5 (Jupiter)** for writing and running tests.
- **Mockito** for mocking dependencies.

Setup in pom.xml:

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

Note: This includes JUnit 5, Mockito, Hamcrest, and AssertJ.

Basic Folder Structure:



Write Test Cases Using JUnit 5 & Mockito

Ex : creating test cases for service layer

Key Annotations

@ExtendWith(MockitoExtension.class) – enables Spring support in JUnit 5.

@Mock – mocks dependencies like repositories.

@InjectMocks or **@Autowired** – injects the service under test.

Service Layer

@Service

```
public class AccountService {

@Autowired
private AccountRepository accountRepository;

public Account create(Account account) {
return accountRepository.save(account);
}

public List<Account> getAccounts() {
List<Account> accounts = accountRepository.findAll();
 if (accounts.isEmpty()) {
 throw new AccountsNotFoundException("No accounts found in the database.");
 }
return accounts;
}
}
```

TestClass

```
package com.spring.project.service;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.mockito.Mockito.when;
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;
import com.spring.project.entity.Account;
import com.spring.project.repository.AccountRepository;
@ExtendWith(MockitoExtension.class)
public class AccountServiceTest {
	@Mock
	private AccountRepository accountRepository;
	@InjectMocks
	private AccountService accountService;
	@Test
	void createAccountShouldReturnSavedAccount() {
		// 1. Data Preparation
```

```
                Account account = new Account(1L, "Ramesh", 5000.00);
                // 2. Mocking
                when(accountRepository.save(account)).thenReturn(account);
                // 3. Call actual method
                Account createdAccount = accountService.create(account);
                // 4. Assertions
                assertNotNull(createdAccount);
                assertEquals(account.getAccountHolderName(),
createdAccount.getAccountHolderName());
                assertEquals(5000.00, createdAccount.getBalance());
        }

@Test
void getAccountsShouldReturnList() {
        Account a1 = new Account(1L, "Ramesh", 5000.0);
        Account a2 = new Account(2L, "Suresh", 7000.0);
        when(accountRepository.findAll()).thenReturn(Arrays.asList(a1, a2));
        List<Account> result = accountService.getAccounts();
        assertEquals(2, result.size());
        }
}
```

Test Class Overview

```
@ExtendWith(MockitoExtension.class)
public class AccountServiceTest {
```

@ExtendWith(MockitoExtension.class)

- Tell JUnit 5 to enable Mockito support in this test class.
- It allows the use of @Mock and @InjectMocks.
- Automatically initializes mocks and injects them into your test class before each test method.

Mocking Dependencies:

```
@Mock
private AccountRepository accountRepository;
```

@Mock creates a **fake repository**. No actual DB operation will happen.

Used to simulate repository behavior.

- Creates a mock (fake version) of the accountRepository.
- This means no actual DB call is made — Mockito just simulates the behavior.

```
@InjectMocks
private AccountService accountService;
```

@InjectMocks tells Mockito to inject the **mocked repository** into AccountService.
This allows testing only AccountService logic.

- Creates an instance of AccountService and **injects the mocked AccountRepository** into it.
- This allows you to test the AccountService class in isolation, without worrying about its real dependencies.

Test Method – Breakdown

```
@Test
void createAccountShouldReturnSavedAccount() {
```

@Test

- Marks this as a test method.
- JUnit runs this method when the test suite is executed.

Step-by-Step Execution Process:

1. Data Preparation

```
Account account = new Account(1L, "Ramesh", 5000.00);
```

You need to add this line **before** the when(...) to create test data.
Creates a dummy account object with ID, name, and balance for the test.

2. Mocking the Method Call

```
when(accountRepository.save(account)).thenReturn(account);
```

- You're telling Mockito:

  "When the save method is called on **AccountRepository** with this account, return the same account back." or tells Mockito: "If save(account) is called, return the same account."

- This simulates what would happen if the product were saved in a real database.

  Simulates successful DB save operation.

3. Calling the Actual Method

```
Account createdAccount = accountService.create(account);
```

- This is the real method call you're testing.
- Internally, AccountService .create(account) calls **AccountRepository** .save(account) — which is mocked.

4. Assertions

```
assertNotNull(createdAccount);
assertEquals(account.getAccountHolderName(), createdAccount.getAccountHolderName());
assertEquals(5000.00, createdAccount.getBalance());
```

Verifies:

- The result is not null.
- ID, name, and balance match expected values.
- A hardcoded check confirms ID is 1.

Controller Layer Testing

For controller layer, we use:

- @WebMvcTest(AccountController.class) – focuses only on the **web layer**

- MockMvc – simulates HTTP requests without starting the full server

Controller layer

```
@RestController
@RequestMapping("/api/accounts")
```

```
public class AccountController {
	@Autowired
	private AccountService accountService;
	@PostMapping
	public Account create(@Valid @RequestBody Account account) {
		return accountService.create(account);
	}
	@GetMapping
	public List<Account> getAccounts() {
		return accountService.getAccounts();
	}
}
```

TestClass

```
@WebMvcTest(AccountController.class)
public class AccountControllerTest {
	@MockBean
	private AccountService accountService;
	@Autowired
	MockMvc mockMvc;
	@Autowired
	private ObjectMapper objectMapper;
	@Test
	void createAccount_shouldReturnCreatedAccount() throws Exception {
		Account account = new Account(1L, "Alice", 5000.0);
		when(accountService.create(account)).thenReturn(account);

mockMvc.perform(post("/api/accounts").contentType(MediaType.APPLICATION_JSON)

.content(objectMapper.writeValueAsString(account))).andExpect(status().isOk())
				.andExpect(jsonPath("$.accountHolderName").value("Alice"))
				.andExpect(jsonPath("$.balance").value(5000.0));
	}
	@Test
	void getAllAccounts_shouldReturnList() throws Exception {
		List<Account> accounts = List.of(new Account(1L, "John", 1000.0));
		when(accountService.getAccounts()).thenReturn(accounts);
```

```
mockMvc.perform(get("/api/accounts")).andExpect(status().isOk()).andExpect(jsonPath("$.size()").value(1))
                                        .andExpect(jsonPath("$[0].accountHolderName").value("John"));
        }
}
```

Test Class Overview

```
@WebMvcTest(AccountController.class)
```

This tells Spring Boot to **only load the controller layer** (not the whole application).

It sets up a **minimal Spring context** just for testing AccountController.

Ideal for **unit testing HTTP endpoints**.

```
@MockBean
private AccountService accountService;
```

Creates a mock of the service so the controller can call it without real service logic.

No real database or business logic is triggered.

```
@Autowired
MockMvc mockMvc;
```

MockMvc is used to simulate HTTP requests like POST, GET, etc., without starting the actual server.

```
@Autowired
private ObjectMapper objectMapper;
```

Converts Java objects to JSON strings and vice versa.

Used here to simulate a real JSON HTTP body in the request.

Step-by-Step Explanation of the Test Case

1. Data Preparation

```
Account account = new Account(1L, "Alice", 5000.0);
```

You create a dummy Account object to simulate the input and expected return.
account will act as both the input and the mock return.

2. Mocking the Service Call

This mocks the accountService.create() method.

- It tells Mockito:

  "If the controller calls create(), then return the dummy account we created."

- No real service method runs — this simulates the controller behavior in isolation.

3. **Simulating an HTTP Request**

```
mockMvc.perform(post("/api/accounts").contentType(MediaType.APPLICATION_JSON)
                                    .content(objectMapper.writeValueAsString(account)))
```

mockMvc.perform(...) simulates an actual HTTP POST call to /api/accounts.

.contentType(...) sets the request type to JSON.

.content(...) sends the JSON representation of the account as the request body.

This simulates sending an account creation request from a frontend.

4. **Verifying the Response**

```
.andExpect(jsonPath("$.accountHolderName").value("Alice"))
                                    .andExpect(jsonPath("$.balance").value(5000.0));
```

status().isOk() checks if HTTP status code is **200 OK**.

jsonPath(...) is used to check values from the **JSON response** body.

It checks if the response contains the correct name "Alice" and balances 5000.0.

Testing of whole Spring Boot application (It's an integration test.)

To **test a whole Spring Boot application**, you typically perform **integration testing or end-to-end (E2E) testing**. This ensures all components — controllers, services, repositories — work together as expected.

Here's a complete guide on **how to test a whole Spring Boot application**:

Let's assume you have:

- Entity: Account
- Repository: AccountRepository
- Service: AccountService
- Controller: AccountController

1. Use @SpringBootTest

```
@SpringBootTest
@AutoConfigureMockMvc
        class MyApplicationTests {
            @Test
            void contextLoads() {
                // This checks if the application context starts successfully.
            }
        }
```

- Loads the **entire Spring context**, simulating a full application run.

- Good for verifying **overall integration** between layers.

EX:

```
@SpringBootTest
@AutoConfigureMockMvc
public class UserControllerIntegrationTest {
@Autowired
private MockMvc mockMvc;
@Autowired
private ObjectMapper objectMapper;
@Autowired
```

```
private UserRepository userRepository;
@Test
public void testCreateAndFetchUsers() throws Exception {
   // Create a user
   User user = new User();
   user.setName("Alice");
   String json = objectMapper.writeValueAsString(user);
   mockMvc.perform(post("/api/users")
                .contentType(MediaType.APPLICATION_JSON)
                .content(json))
          .andExpect(status().isOk())
          .andExpect(jsonPath("$.name").value("Alice"));
   // Fetch all users
   mockMvc.perform(get("/api/users"))
          .andExpect(status().isOk())
          .andExpect(jsonPath("$.size()").value(1));
   List<User> users = userRepository.findAll();
   assertThat(users).hasSize(1);
   assertThat(users.get(0).getName()).isEqualTo("Alice");
}
}
```

Purpose of @AutoConfigureMockMvc

It tells Spring Boot to:

- Set up the MockMvc bean for your test class.

When to Use It

You use @AutoConfigureMockMvc **in combination with** @SpringBootTest when you want to:

- Test the full application context (controllers, services, repositories, etc.)
- Use MockMvc to perform and validate HTTP requests (like GET, POST).

@SpringBootTest + @AutoConfigureMockMv = Full application context (controllers, services, repos...)

@WebMvcTest = Web layer only (Controller, MVC config)

Use @WebMvcTest for fast, focused **unit tests**.

Use @SpringBootTest + @AutoConfigureMockMvc when testing the **end-to-end flow** including business logic and database.

While testing the controller, we will take service as a mock bean... while testing whole, we will use the repository directly.

🧠 Key Differences

Aspect	`@WebMvcTest`	`@SpringBootTest + @AutoConfigureMockMvc`
Application Context	Loads **only** web layer (controller, filters)	Loads **entire** Spring context (all beans)
Injected Beans	Only Controller & MVC-related beans	Controllers, Services, Repos, etc.
Service Layer	❌ Not real – you must `@MockBean` services	✅ Real service implementation is used
Repository Layer	❌ Not loaded	✅ Loaded (real DB or in-memory DB like H2)
Testing Focus	Unit test controller behavior	Integration test (flow: controller → repo)