Testing with JUnit and Mockito (with Spring Boot Context)

1. Introduction

Testing in Java applications ensures **correctness, stability, and confidence during refactoring**. In modern Spring-based applications, testing is commonly divided into:

- **Unit Testing** – Tests a single class in isolation using mocks
- **Integration Testing** – Tests interaction between multiple layers or components

Key Libraries

- **JUnit 5 (Jupiter)** – Test framework (test lifecycle, assertions)
- **Mockito** – Mocking framework (stubbing, verification)
- **Spring Test** – Spring context loading and dependency injection for tests
- **Testcontainers / H2** – For database-backed integration tests

---

2. Unit Testing

2.1 What is Unit Testing?

- Tests **one class at a time**
- No Spring context required
- Dependencies are **mocked**
- Fast execution

Typical Targets

- Service classes
- Utility classes
- Validators

---

3. Setting Up Unit Tests

3.1 Maven Dependencies

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <scope>test</scope>
```

```
</dependency>

<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <scope>test</scope>
</dependency>
```

3.2 Basic Unit Test Structure

```
@ExtendWith(MockitoExtension.class)
class OrderServiceTest {

    @Mock
    private OrderRepository orderRepository;

    @InjectMocks
    private OrderService orderService;

    @Test
    void shouldCreateOrder() {
        when(orderRepository.save(any())).thenReturn(new Order());

        Order order = orderService.createOrder();

        assertNotNull(order);
        verify(orderRepository).save(any());
    }
}
```

---

4. Mock vs InjectMocks vs Manual Mock vs MockBean (Clarified)

This section clarifies **what each annotation does, who creates the object, and when to use it**.

4.1 @Mock

What it is
- A Mockito-created fake object - Contains **no real logic** - All method calls return default values unless stubbed

Who creates it?
- Mockito (not Spring)

Used for
- Dependencies of the class under test

```
@Mock
private PaymentGateway gateway;
```

What happens internally
- Mockito creates a proxy object - Calls are intercepted and recorded

❌ Not injected automatically into any class

4.2 @InjectMocks

What it is
- Tells Mockito to create the **class under test** - Injects available mocks into it

Who creates it?
- Mockito

Injection order
1. Constructor injection 2. Setter injection 3. Field injection

```
@InjectMocks
private PaymentService paymentService;
```

What happens internally
- Mockito instantiates `PaymentService` - Finds matching `@Mock` fields - Injects them automatically

✔️ No Spring context involved

4.3 Manual Mock (Mockito.mock)

What it is
- Explicit creation of a mock using code

```
PaymentGateway gateway = Mockito.mock(PaymentGateway.class);
```

Who creates it?
- Mockito (manually triggered)

When to use
- Dynamic test creation - No JUnit extension available - Legacy tests

❌ No automatic injection

4.4 @MockBean (Spring Integration Testing)

What it is
- Spring Boot annotation - Replaces a real Spring bean with a Mockito mock

Who creates it?
- Spring (using Mockito internally)

```
@MockBean
private PaymentGateway paymentGateway;
```

What happens internally
- Spring removes the real bean from context - Registers a Mockito mock instead - All `@Autowired` injections receive the mock

✔️ Works only with Spring context ✔️ Used in integration tests

---

4.5 Summary Table

Feature	@Mock	@InjectMocks	Manual Mock	@MockBean
Spring context	❌	❌	❌	✅
Creates mock	✅	❌	✅	✅
Injects dependencies	❌	✅	❌	✅
Replaces real bean	❌	❌	❌	✅

Feature	@Mock	@InjectMocks	Manual Mock	@MockBean
Typical use	Unit test	Unit test	Special cases	Integration test

---

4.2 @InjectMocks

- Injects mocks into the class under test
- Uses constructor → setter → field injection order

```
@InjectMocks
private PaymentService paymentService;
```

4.3 Mockito.mock() (Manual)

```
PaymentGateway gateway = Mockito.mock(PaymentGateway.class);
```

Used when: - No JUnit extension - Dynamic test construction

---

5. Useful Mockito Methods (Explained Line by Line)

This section explains **what each method does and how Mockito behaves internally**.

5.1 Stubbing (`when / thenReturn / thenThrow`)

```
when(repo.findById(1L)).thenReturn(Optional.of(entity));
```

What this does
- Intercepts calls to `findById(1L)` - Returns the provided value instead of executing real logic

Why needed
- To control dependency behavior - To make tests deterministic

---

```
when(service.call()).thenThrow(new RuntimeException());
```

What this does
- Forces the method to throw an exception - Used to test error handling paths

5.2 Verification (`verify`)

```
verify(repo).save(any());
```

What this checks
- Confirms that `save()` was called exactly once

If not called
- Test fails

```
verify(repo, times(2)).findAll();
```

What this checks
- Method invoked exactly two times

```
verify(repo, never()).delete(any());
```

What this checks
- Ensures `delete()` was not called at all

5.3 Argument Matching

```
verify(repo).save(argThat(o -> o.getAmount() > 100));
```

What this does
- Validates method arguments using a predicate - Allows fine-grained behavior verification

---

5.4 Argument Matchers (`any`, `eq`, etc.)

```
when(repo.findById(anyLong())).thenReturn(Optional.empty());
```

Important rule
- If one argument uses matcher → all must

```
when(repo.find(eq(1L), anyString()));
```

5.5 Capturing Arguments (`ArgumentCaptor`)

```
ArgumentCaptor<Order> captor = ArgumentCaptor.forClass(Order.class);
verify(repo).save(captor.capture());
Order saved = captor.getValue();
```

What this does
- Captures the actual argument passed - Useful when state is complex

---

5.6 Default Behavior of Mocks

Return Type	Default Value
Object	null
boolean	false
int	0
Collection	empty

❗ Unstubbed calls never execute real code

5.7 Common Mistakes

- Verifying before method execution
- Mixing matchers and raw values
- Mocking value objects
- Over-verifying interactions

5.2 Verification

```
verify(repo).save(any());
verify(repo, times(2)).findAll();
verify(repo, never()).delete(any());
```

5.3 Argument Matching

```
verify(repo).save(argThat(o -> o.getAmount() > 100));
```

---

6. Useful JUnit Assertions

```
assertEquals(expected, actual);
assertNotNull(value);
assertThrows(Exception.class, () -> service.call());
assertTrue(condition);
```

---

7. Integration Testing

7.1 What is Integration Testing?

- Loads **Spring ApplicationContext**
- Multiple beans wired together
- Slower than unit tests
- Validates configuration + wiring

---

8. Setting Up Integration Tests

8.1 Common Annotations

```
@SpringBootTest
@Transactional
@ActiveProfiles("test")
```

- `@SpringBootTest` – Loads full context
- `@Transactional` – Rollback after test
- `@ActiveProfiles` – Uses test config

---

9. Injecting Mocks in Integration Tests

9.1 @MockBean (Spring-managed Mock)

- Replaces real bean in Spring context

- Commonly used for **external services**

```
@SpringBootTest
class OrderIntegrationTest {

    @MockBean
    private PaymentGateway paymentGateway;

    @Autowired
    private OrderService orderService;

    @Test
    void testWithMockedGateway() {
        when(paymentGateway.pay()).thenReturn(true);
        orderService.placeOrder();
    }
}
```

✔️ Service and repository beans remain real ✔️ External dependency mocked

---

9.2 @SpyBean (Partial Mock)

- Real bean, but allows stubbing specific methods

```
@SpyBean
private EmailService emailService;
```

Use when: - Most logic should remain real - Only one method needs mocking

---

10. Repository Layer in Integration Tests

10.1 Using Real Database (H2)

```
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driver-class-name: org.h2.Driver
    username: sa
    password: ''
  jpa:
    hibernate:
      ddl-auto: create-drop
```

```java
@DataJpaTest
class UserRepositoryTest {

    @Autowired
    private UserRepository repository;

    @Test
    void saveAndFind() {
        User u = repository.save(new User("abc"));
        assertTrue(repository.findById(u.getId()).isPresent());
    }
}
```

10.2 Replacing Mock with Real Implementation

Scenario	Approach
Unit test	@Mock + @InjectMocks
Integration test (mock service)	@MockBean
Integration test (real service + repo)	No mocks
Repo test only	@DataJpaTest

Spring automatically injects **real beans** if no @MockBean exists.

---

11. Switching Between Mock and Real DB

Option 1: Profiles

- `application-test.yml` → H2
- `application-prod.yml` → Real DB

```java
@ActiveProfiles("test")
```

Option 2: Testcontainers (Advanced)

- Spins up real DB (Postgres/MySQL)
- No mocks at repository layer
- Production-like behavior

---

12. Test Pyramid Summary

```
        E2E Tests
     Integration Tests
  ----------------------
       Unit Tests
```

- **Most tests** should be unit tests
- **Fewer integration tests**
- **Minimal E2E tests**

---

13. Best Practices

- One assertion focus per test
- Avoid mocking what you don't own
- Mock external systems, not domain logic
- Name tests clearly (given/when/then)
- Keep integration tests deterministic

---

14. Spy vs Mock (Deep Clarification)

This section addresses one of the **most misunderstood Mockito concepts**.

14.1 @Mock (Pure Fake)

What it does
- Creates a complete fake object - **No real methods are executed** - All methods return defaults unless stubbed

```
@Mock
EmailService emailService;
```

```
when(emailService.send()).thenReturn(true);
```

Internal behavior
- Method calls are intercepted - No underlying object exists

✔️ Best for **external dependencies**
❌ Bad for testing partial logic

14.2 @Spy (Partial Mock)

What it does
- Wraps a **real object** - Calls real methods by default - Allows selective stubbing

```
@Spy
EmailService emailService = new EmailService();
```

```
when(emailService.send()).thenReturn(true);
```

Internal behavior
- Real object exists - Only stubbed methods are overridden

✔️ Useful when most logic should run ❌ Dangerous if used blindly

---

14.3 @SpyBean vs @MockBean (Spring Context)

```
@MockBean
EmailService emailService;
```

- Replaces real Spring bean entirely
- No real logic executes

```
@SpyBean
EmailService emailService;
```

- Wraps real Spring bean
- Allows partial override

✔️ Used in integration tests

14.4 Key Difference Table

Aspect	Mock	Spy
Real object exists	❌	✅
Real methods run	❌	✅ (unless stubbed)
Default behavior	Fake	Real

Aspect	Mock	Spy
Risk level	Low	High

14.5 Common Spy Pitfall (VERY IMPORTANT)

```
@Spy
List<String> list = new ArrayList<>();

when(list.get(0)).thenReturn("A"); // ❌ calls real method
```

This fails because: - `when()` invokes the real method - `get(0)` executes immediately

✅ Correct way

```
doReturn("A").when(list).get(0);
```

14.6 When to Use Spy (Rule-Based)

Use **Spy** when: - Legacy code cannot be refactored - Only one method needs isolation - Object creation is expensive

Avoid **Spy** when: - You don't fully understand the code - Testing pure business logic - You can refactor to use Mock

---

14.7 Golden Rules

- Prefer **Mock** over Spy
- Spy only when absolutely necessary
- Never spy on entities or DTOs
- Spy + `doReturn()` is safer than `when()`

---

15. Final When-to-Use Summary

Scenario	Recommended
Unit test business logic	@Mock + @InjectMocks
Partial override	@Spy

Scenario	Recommended
Integration test, fake external service	@MockBean
Integration test, partial override	@SpyBean
Database testing	Real repo + test DB

End of Notes