**1. Tagging and Filtering Tests**

In JUnit 5, you can use @Tag to categorize your tests, and filtering can be done through your build tool (Maven, Gradle) or within a CI pipeline.

**Code Example (JUnit 5):**

import org.junit.jupiter.api.Tag;

import org.junit.jupiter.api.Test;

class TaggedTest {

@Test

@Tag("smoke")

void testLogin() {

// Simulate login functionality test

}

@Test

@Tag("regression")

void testPayment() {

// Simulate payment functionality test

}

}

**Running only @smoke tests with Maven:**

mvn test -Dgroups=smoke

**2. Conditional Test Execution**

You can conditionally run or skip tests based on environment conditions.

**Code Example (JUnit 5):**

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.condition.EnabledIfSystemProperty;

class ConditionalTest {

@Test

@EnabledIfSystemProperty(named = "os.arch", matches = "amd64")

void runOnlyOn64Bit() {

System.out.println("Running on a 64-bit architecture.");

}

@Test

@EnabledIfSystemProperty(named = "os.name", matches = "Linux.\*")

void runOnlyOnLinux() {

System.out.println("Running on Linux.");

}

}

In this example, the tests will only run if the conditions specified are met (e.g., os.arch is amd64).

**3. Nested Tests**

JUnit 5 allows creating nested tests with @Nested for better test organization.

**Code Example (JUnit 5):**

import org.junit.jupiter.api.Nested;

import org.junit.jupiter.api.Test;

class NestedTest {

@Nested

class LoginTests {

@Test

void testLoginWithValidCredentials() {

System.out.println("Login with valid credentials.");

}

@Test

void testLoginWithInvalidCredentials() {

System.out.println("Login with invalid credentials.");

}

}

@Nested

class PaymentTests {

@Test

void testPaymentWithValidCard() {

System.out.println("Payment with valid card.");

}

@Test

void testPaymentWithExpiredCard() {

System.out.println("Payment with expired card.");

}

}

}

Here, we use @Nested to group related tests under different categories.

**4. Repeated Tests**

@RepeatedTest allows running the same test method multiple times.

**Code Example (JUnit 5):**

import org.junit.jupiter.api.RepeatedTest;

class RepeatedTestExample {

@RepeatedTest(5)

void testLogin() {

System.out.println("Running login test.");

}

}

This will run the testLogin() method 5 times.

**5. Dependency Injection**

JUnit 5 supports dependency injection using @ExtendWith with extensions like Spring.

**Code Example (JUnit 5 with Spring Extension):**

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.extension.ExtendWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest

@ExtendWith(SpringExtension.class)

class MyServiceTest {

@Autowired

private MyService myService;

@Test

void testService() {

assertNotNull(myService);

myService.performService();

}

}

This example assumes you're using Spring, and it will inject the MyService bean into the test.

**6. Test Templates**

JUnit 5 provides @TestTemplate for reusable tests with extensions.

**Code Example (JUnit 5):**

import org.junit.jupiter.api.TestTemplate;

import org.junit.jupiter.api.extension.ExtendWith;

class TemplateTest {

@TestTemplate

@ExtendWith(MyTestTemplateExtension.class)

void templateTest() {

System.out.println("Test template executed.");

}

}

**Test Extension Example (JUnit 5):**

import org.junit.jupiter.api.extension.ExtensionContext;

import org.junit.jupiter.api.extension.TestTemplateInvocationContext;

import org.junit.jupiter.api.extension.TestTemplateInvocationContextProvider;

class MyTestTemplateExtension implements TestTemplateInvocationContextProvider {

@Override

public boolean supportsTestTemplate(ExtensionContext context) {

return true;

}

@Override

public java.util.List<TestTemplateInvocationContext> provideTestTemplateInvocationContexts(ExtensionContext context) {

return java.util.Collections.singletonList(() -> System.out::println);

}

}

**7. Test Interfaces**

In JUnit, interfaces can define common behavior for tests.

**Code Example (JUnit 5):**

interface DatabaseTest {

void connectToDatabase();

}

class MyDatabaseTest implements DatabaseTest {

@Override

public void connectToDatabase() {

System.out.println("Connected to database.");

}

@Test

void testDatabaseConnection() {

connectToDatabase();

}

}

In this example, the DatabaseTest interface defines the contract for connectToDatabase().

**8. Parameterized Tests**

With @ParameterizedTest, you can run the same test with different input values.

**Code Example (JUnit 5):**

import org.junit.jupiter.api.ParameterizedTest;

import org.junit.jupiter.params.provider.ValueSource;

class ParameterizedTestExample {

@ParameterizedTest

@ValueSource(strings = {"apple", "banana", "cherry"})

void testWithParameters(String fruit) {

System.out.println("Testing with: " + fruit);

}

}

This test will run three times, once for each string (apple, banana, cherry).

**9. Timeouts**

You can specify a timeout for your tests to ensure they don't run indefinitely.

**Code Example (JUnit 5):**

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.Timeout;

class TimeoutTest {

@Test

@Timeout(5) // test will fail if it takes longer than 5 seconds

void someLongRunningTest() {

try {

Thread.sleep(4000); // Sleep for 4 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**10. Running Tests**

Tests can be executed from the command line using Maven, Gradle, or in your IDE. Here's an example with Maven:

**Code Example (Maven):**

mvn test

Alternatively, you can run tests using an IDE (IntelliJ IDEA, Eclipse, etc.) by right-clicking on the test class or method and choosing to run.

**11. Test Mocks with Mockito**

Mockito allows you to mock dependencies in unit tests.

**Code Example (Mockito):**

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.Mockito;

import org.mockito.junit.jupiter.MockitoExtension;

import org.junit.jupiter.api.extension.ExtendWith;

@ExtendWith(MockitoExtension.class)

class MockitoTest {

@Mock

private MyService myService; // Mocked service

@InjectMocks

private MyController myController; // Inject mock into controller

@Test

void testServiceMethod() {

// Arrange

Mockito.when(myService.getData()).thenReturn("Mocked Data");

// Act

String result = myController.fetchData();

// Assert

assertEquals("Mocked Data", result);

}

}

In this example, we mock MyService and inject it into MyController. The test verifies that the mocked getData() method returns "Mocked Data".