JUNIT:-

* Unit testing
* Integration Testing
* Smoke Testing
* Functional Testing
* Regression Testing
* Performance Testing

Junit is an open source frame work. Which is used to perform unit testing for java projects.

@Test: Which is used to represent one method as a unit test method.

Assert method: It is used to validate our unit test result

When we are developing an application, we will use layered architecture.

In application we will be having below layers

1. Presentation layer(use interface components/ UI)
2. Web layer (Controller components)
3. Business layer(Service components)
4. Persistence layer( Dao component)

In the above four layers, we are going to use Junit to test last 3 layers code only.

In project 2 developers are working john and smith

1.John is working on story called HIS-134

2.Smith is working on story called HIS-135

John changes are related to Service class method and

Smith changes are related do DAO class method.

Here John completed his service layer method implementation Smith changes are not yet completed for Dao implementation.

Can John perform Unit testing for his changes or not?

No, he cannot perform john implemented code is dependent on Dao method implementation which is not yet ready.

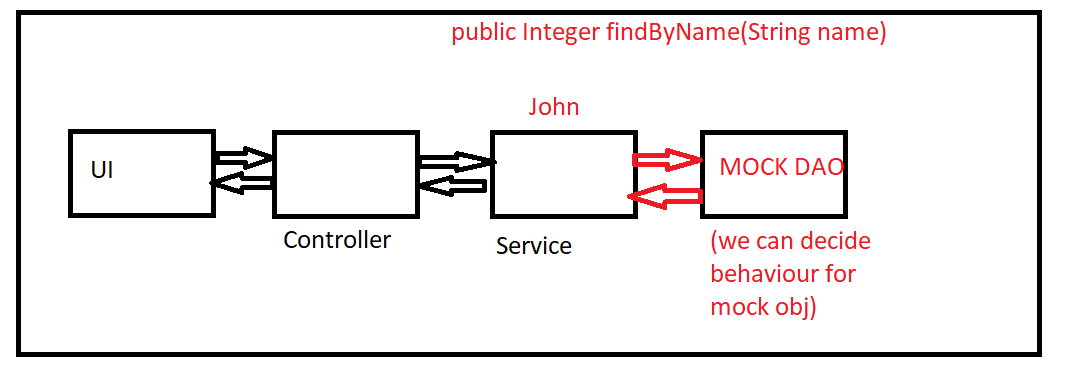
To overcome above problem, we should use mock frame work to resolve further.

What is Mocking:-

Mocking is the process of creating alternate/substitute object for real object.

When to use mocking?

1. If dependent components are not ready
2. To isolate(I don’t want test Dao logic) our component for unit testing



There are several mock frameworks available in market

Mock framework are categorized into 2 types

1. Proxy Based Mock Frameworks

Ex: Easy and Mockito

1. Byte Code manipulation Based Mock Framework

Or

Class remapping Mock framework

Ex: Power Mock and JMockito

Limitation of 1. Proxy Based framework:-

* Does not support for final methods mocking
* Does not support for private methods mocking
* Does not support for static methods mocking
* Does not support for final class mocking

To avoid the above limitations we are going to use Byte Code Manipulation based Mock Framework.

Steps to develop application using Junit and EasyMock:-

JUnit: Unit testing framework

EasyMock: To perform Mocking

Steps to develop JUnit with mock project:-

Step 1: Create spring boot project

package com.app;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ByteCodeManipulationBasedMockApplication {

public static void main(String[] args) {

SpringApplication.run(ByteCodeManipulationBasedMockApplication.class, args);

}

}

Step 2: Add dependencies in pom.xml

🡪 JUnit

🡪EasyMock

<!-- https://mvnrepository.com/artifact/junit/junit -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/org.easymock/easymock -->

<dependency>

<groupId>org.easymock</groupId>

<artifactId>easymock</artifactId>

<version>4.1</version>

<scope>test</scope>

</dependency>

Step 3: Create DAO interface(Only interface no implementation)

(required methods we will create as abstract)

ContactDao:-

**package** com.app.dao;

**public** **interface** ContactDao {

**public** String findNamById(Integer id);

}

Step 4: Create Service interface and implementation

(as part service methods we will call dao method)

ContactService:-

**package** com.app.service;

**public** **interface** ContactService {

**public** String getNameById(Integer id);

}

ContactServiceImpl:-

**package** com.app.service.impl;

**import** com.app.dao.ContactDao;

**import** com.app.service.ContactService;

**public** **class** ContactServiceImpl **implements** ContactService {

**private** ContactDao dao;

// people who want to set the object they can use set the object

**public** **void** setDao(ContactDao dao) {

**this**.dao = dao;

}

@Override

**public** String getNameById(Integer id) {

String name = dao.findNamById(id);

// logic

String formatName = name.toUpperCase();

**return** formatName;

}}

Step 5: Create Unit test class for service class using Junit

EasyMockApplicationTests:-

**package** com.app;

**import** **static** org.junit.Assert.*assertNotNull*;

**import** org.easymock.EasyMock;

**import** org.junit.jupiter.api.Test;

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

**import** com.app.dao.ContactDao;

**import** com.app.service.impl.ContactServiceImpl;

@SpringBootTest

**class** EasyMockApplicationTests {

@Test

**void** contextLoads() {

}

@Test

**public** **void** testGetNameById\_01()

{

// Dao is a interface,there is no implementation so we should use below

ContactDao daoProxy = EasyMock.*createMock*(ContactDao.**class**);

// setting behavior for proxy

EasyMock.*expect*(daoProxy.findNamById(101)).andReturn("Ashok");

EasyMock.*expect*(daoProxy.findNamById(102)).andReturn("Ashok IT");

EasyMock.*replay*(daoProxy);

ContactServiceImpl serviceTest = **new** ContactServiceImpl();

// can i inject into service implements method -yes

serviceTest.setDao(daoProxy);// set to proxy object

String nameById = serviceTest.getNameById(101);

// how we can compare name is null or not null

*assertNotNull*(nameById);

}

}

Junit 5:- Junit Jupitor Engine

Junit 4 & 3:- Junit Vintage Engine

Steps to develop Junit project:-

Step 1:-

Add maven dependency in pom.xml file

<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-engine -->

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.6.2</version>

<scope>test</scope>

</dependency>

TestMethodOrder:-

To specify the order for our junit methods

We have 3 different possibilities

1. Alphanumeric
2. OrderAnnotation(if we want own order)
3. Random(Default)
4. @SpringBootTest
5. @TestMethodOrder(OrderAnnotation.**class**)//predefined
6. [//@TestMethodOrder(OrderAnnotation.**class**)//predefined](mailto://@TestMethodOrder(OrderAnnotation.class)//predefined) first sort 0-9 and then A-Z
7. **class** JunitJupitorEngineAppApplicationTests {
8. @Test
9. @Order(2)
10. **public** **void** testSave() {
11. System.***out***.println("Save method");
12. }
13. @Test
14. @Order(1)
15. **public** **void** testUpdate() {
16. System.***out***.println("Update method");
17. }
18. @Test
19. @Order(3)
20. **public** **void** testDelete() {
21. System.***out***.println("Delete method");
22. }
23. }

@BeforeEach:-

This Annotation is executed before every test method is execution start and @BeforeEach method executed first.

@AfterEach:-

This Annotation is executed after every test method is execution completed and @AfterEach method executed end.

@SpringBootTest

**class** JunitJupitorEngineAppApplicationTests {

@BeforeEach

**public** **void** init()

{

System.***out***.println(" from @BeforeEach");

}

@Test

**public** **void** testSave() {

System.***out***.println("Save method");

}

@Test

**public** **void** testUpdate() {

System.***out***.println("Update method");

}

@Test

**public** **void** testDelete() {

System.***out***.println("Delete method");

}

@AfterEach

**public** **void** clear()

{

System.***out***.println("from @AfterEach");

}

}

Output:-

from @BeforeEach

Save method

from @AfterEach

from @BeforeEach

Delete method

from @AfterEach

from @BeforeEach

Update method

from @AfterEach

@BeforeAll and @AfterAll:- I want executed logic only once per Test Class

@SpringBootTest

**class** JunitJupitorEngineAppApplicationTests {

@BeforeAll //must be method should have static

**public** **static** **void** beforeAllOneTimeSetup()

{

System.***out***.println("only one time setup start");

}

@BeforeEach

**public** **void** init()

{

System.***out***.println(" from @BeforeEach ");

}

@Test

**public** **void** testSave() {

System.***out***.println("Save method");

}

@Test

**public** **void** testUpdate() {

System.***out***.println("Update method");

}

@Test

**public** **void** testDelete() {

System.***out***.println("Delete method");

}

@AfterEach

**public** **void** clear()

{

System.***out***.println("from @AfterEach");

}

@AfterAll //must be method should have static

**public** **static** **void** afterAllOneTimeSetup()

{

System.***out***.println("only one time setup end");

}

}

Output:

only one time setup start

from @BeforeEach

Save method

from @AfterEach

from @BeforeEach

Delete method

from @AfterEach

from @BeforeEach

Update method

from @AfterEach

only one time setup end

@DisplayName("Name of class name/Method name"):-

Above annotation is used to to change method name and class name for required format to show on Junit console

Disabled Test method:- // this annotation is used for Ignore method

@SpringBootTest

**class** JunitJupitorEngineAppApplicationTests {

@Test

**public** **void** testSave() {

System.***out***.println("Save method");

}

@Test

**public** **void** testUpdate() {

System.***out***.println("Update method");

}

@Test

@Disabled // this annotation is used for Ignore method while executing

**public** **void** testDelete() {

System.***out***.println("Delete method");

}

}

**public** **interface** TestInfo:- which is provide information about

and

implementation class is DefaultTestInfo

@SpringBootTest

**class** JunitJupitorEngineAppApplicationTests {

@Test

**public** **void** testMultpleBatch(TestInfo info)

{

System.***out***.println("Hello "+info.getTestClass().get().getName()+info.getTestMethod().get().getName()+info.getDisplayName());

}

}

Output:-

Hello com.app.JunitJupitorEngineAppApplicationTeststestMultpleBatchtestMultpleBatch(TestInfo)

@RepeatedTest(value = 3):-

I want repeated execute test method multiple times

@SpringBootTest

**class** JunitJupitorEngineAppApplicationTests {

@RepeatedTest(value = 3,name =" {displayName}-{currentRepetition}-{totalRepetitions}")

@DisplayName("Multi Test")

**public** **void** testMultpleBatch(TestInfo info)

{

System.***out***.println("Hello :"+info.getDisplayName());

}

}

@Tag:-

It provide meta information our test method

Will be having different environ like dev,prod,qa..ect

We can perform which environment want to execute using above annotation

@SpringBootTest

**class** JunitJupitorEngineAppApplicationTests {

@Tag("dev")

@Test

**public** **void** testTagDev()

{

System.***out***.println("Hello-Dev");

}

@Tag("prod")

@Test

**public** **void** testTagProd()

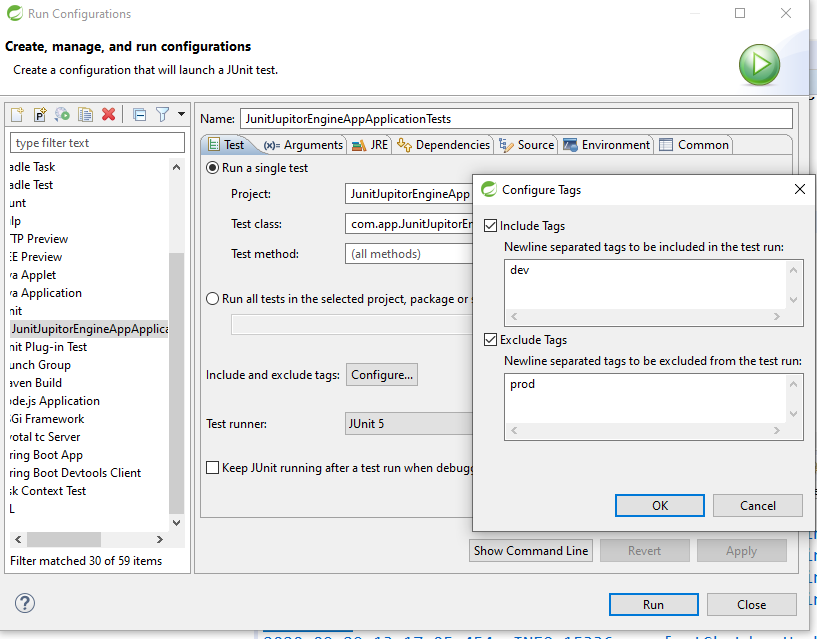
{

System.***out***.println("Hello-prod");

}

}

Run as >run configuration>



Output:

Only dev printed

Instead of running above process, we can maintain add below dependency in pom.xml

## [Maven Surefire Plugin](https://mvnrepository.com/artifact/org.apache.maven.plugins/maven-surefire-plugin):-

## <!-- https://mvnrepository.com/artifact/org.apache.maven.plugins/maven-surefire-plugin -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M5</version><!--$NO-MVN-MAN-VER$-->

<configuration>

<!-- include tags -->

<groups>prod</groups>

<!-- include tags -->

<excludedGroups>dev</excludedGroups>

</configuration>

</plugin>

Right click project>run as>maven test

Assertions:-( Jupitor API)

It is used to validate Test, Is current test s pass/fail?

1. Assertions.*assertEquals*(expected, actual);
2. Assertions.*assertEquals*(expected, actual,may be not match);// if fail case

assertNotNull():-

this method is used to given object is not a null value hold data some data,else Test fail

assertSame(expected, actual,’may not match’):- this method is used to the test given object are pointed to

what is the difference between *assertEquals*(expected, actual) vs assertSame(expected, actual,’may not match’) ?

*assertEquals*(expected, actual):- compares to two objects data

assertSame(expected, actual,’may not match’):- compare object references.

*assertArrayEquals*(expected, actual):-

Compare two arrays

@SpringBootTest

**class** JunitJupitorEngineAppApplicationTests {

@Test

**public** **void** testshowMg()

{

**int** expected[]= {10,20};

**int** actual[]= {10,20};

*assertArrayEquals*(expected, actual,”data may not same”);

}

}

assertAll(Executable…executable):-

This is used to group all asset test methods and execute once.

If all are pass then test is pass, if one is fail then test is fail.

Note: Executable is a functional interface. That is having only one abstract method execute() method