\* Junit: this is one of unit testing tool, its developed by Kent Beck, Erich Gamma, David Saff, Kris Vasudevan.

\*

\* by using Junit we can test our java code either its working as expected or not.

\* -> using junit we can test both positive cases and negative case.

\*

\* if we want to use junit in our java development , we need to use 2 third party libraries

\* which release by junit company like log4j jar files.

\* useage of Junit:

\* ====================

\* 1. download these 2 jar fiels form junit.org or maven repository

\* Junit.jar

\* hamcrest.jar files

\*

\* https://mvnrepository.com/artifact/org.hamcrest/hamcrest/3.0

\* https://mvnrepository.com/artifact/junit/junit/4.12

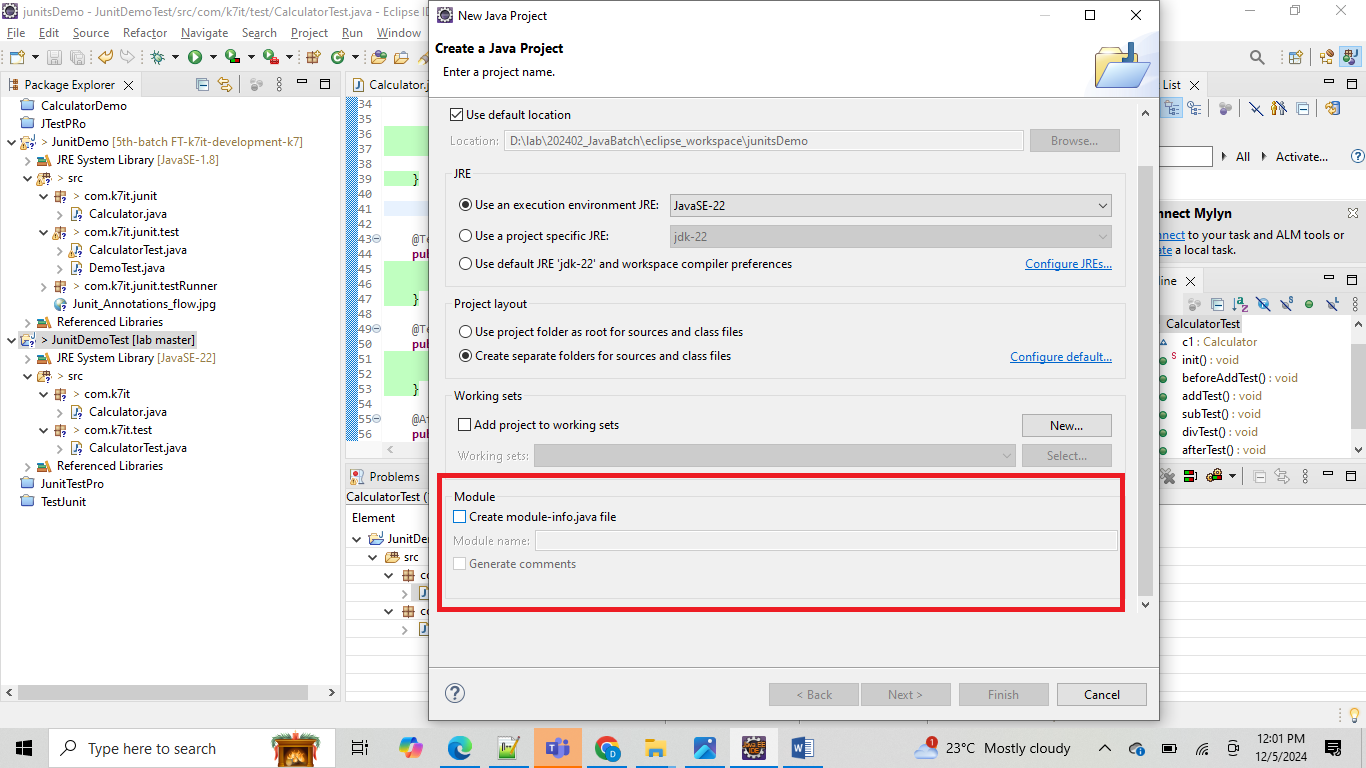
\*

\* 2. create java project

\* 3. link these 2 jar files into our java project from buildpath.

\* project -> rt click -> build path -> libraries-> add external jars ->

**Note: while creating java project if u r using jdk 22 or above , don’t select module-ing.java option , just make sure that check box is un selected.**

****

\* 4. write a java program which is requored for ur business.

\* 5. write a Test class for your java program and test the each methods in ur java class.

\*

\* 6. use assert statements to compare the actual method outputs

\* // important assert statements to compare our method results from unit test class.

assertEquals(30, c1.add(10,20));// this assert will compare first value and second

// value

// if both are equal it will return true else it

// return false and it raise the error message.

//assertTrue(true);// this will check for boolean true value else it will raise exception

//assertFalse(false);// it will check for booelan false value else it will raise exception

//assertNotNull(); // here compare value should not be a null value or null obj

//assertNull();// here compare value should null, else it will raise the excpetion

//assertNotEquals(); // the comapring 2 values should be equal else it will raise the exception

7. how to compile and how to run the junit and how to verify the result and where to see the console output

how to compile : same line normal java class using javac cmd. in case of

eclipse compilation will done automatically.

how to run : rt click on test class and select run as junit or debug as junit

Junit Annotations flow hierarchy:

=================================

1. @BeforeClass : this annotation test method will execute only once for one Test class

execution, we can use this annotation if we have requirement to

execute any common task for all test methods only once before starting

executing those test class.

2. @Before : this annotation test method will execute before every test method execution.

we can use this annotation if we have any specific requirement

to execute for every test method.

3. @Test : this annotation we will use it for main test methods.

4. @After : this annotation will execute after finishing every test method execution

5. @AfterClass : this annotation will execute after finishing entire test class execution.

How to Run the Junits using TestRunner class instead if running each class manually.

\*

\*

\*

\*

\*/

}

package com.k7it;

public class Calculator {

public int add(int i, int j) {

return i+j;

}

public int sub(int i, int j) {

return i-j;

}

}

package com.k7it.test;

import static org.junit.Assert.assertEquals;

import org.junit.After;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.Test;

import com.k7it.Calculator;

public class CalculatorTest {

Calculator c1 = null;

@BeforeClass

public static void init() {

System.out.println("inside init method");

// c1 = new Calculator();

}

@Before

public void beforeAddTest() {

System.out.println("beforeTest");

c1 = new Calculator();

}

@Test

public void addTest() {

System.out.println("inside addTest");

// from here we need to verify our calculator class functionality

// Calculator c1 = new Calculator();

int sum = c1.add(10, 20);

assertEquals(30, sum);

}

@Test

public void subTest() {

System.out.println("inside subTest");

assertEquals(10, c1.sub(20, 10));

}

@After

public void afterTest() {

System.out.println("after Test");

c1 = null;

}

}

output:

inside init method

beforeTest

inside subTest

after Test

beforeTest

inside addTest

after Test

here : for @BeforeClass method should be static type , since it will invoke before creating object for that class

its like a SIB. it will execute only once even we have more test methods

-> @Before : this method will execute for every test method once before start executing main test method.

its like a IIB, for every test this @Before method will execute commonly.

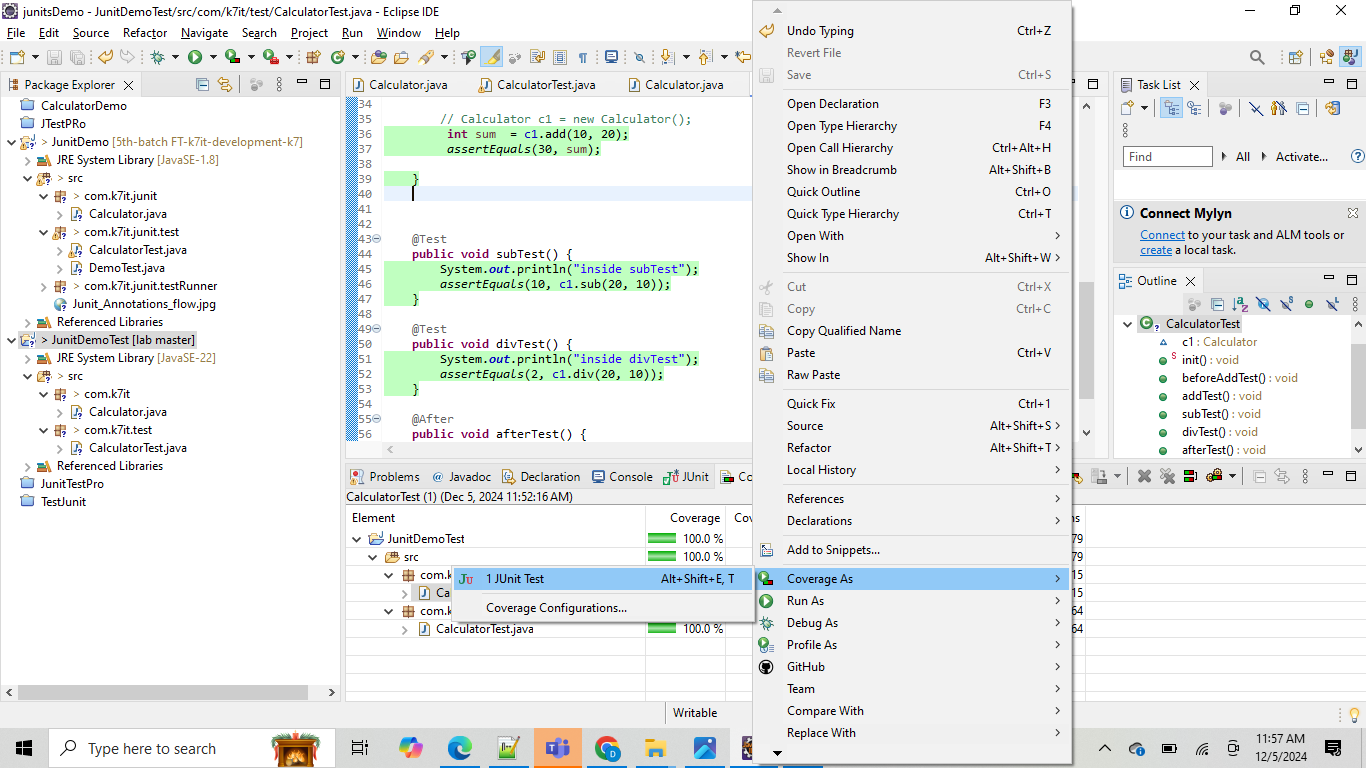
-> @after this methods same as @Bef, but it will execute as post operation after executing @Test methods.

@AfterClass -> this is again same as @BeforeClass, it will execute post operation of all

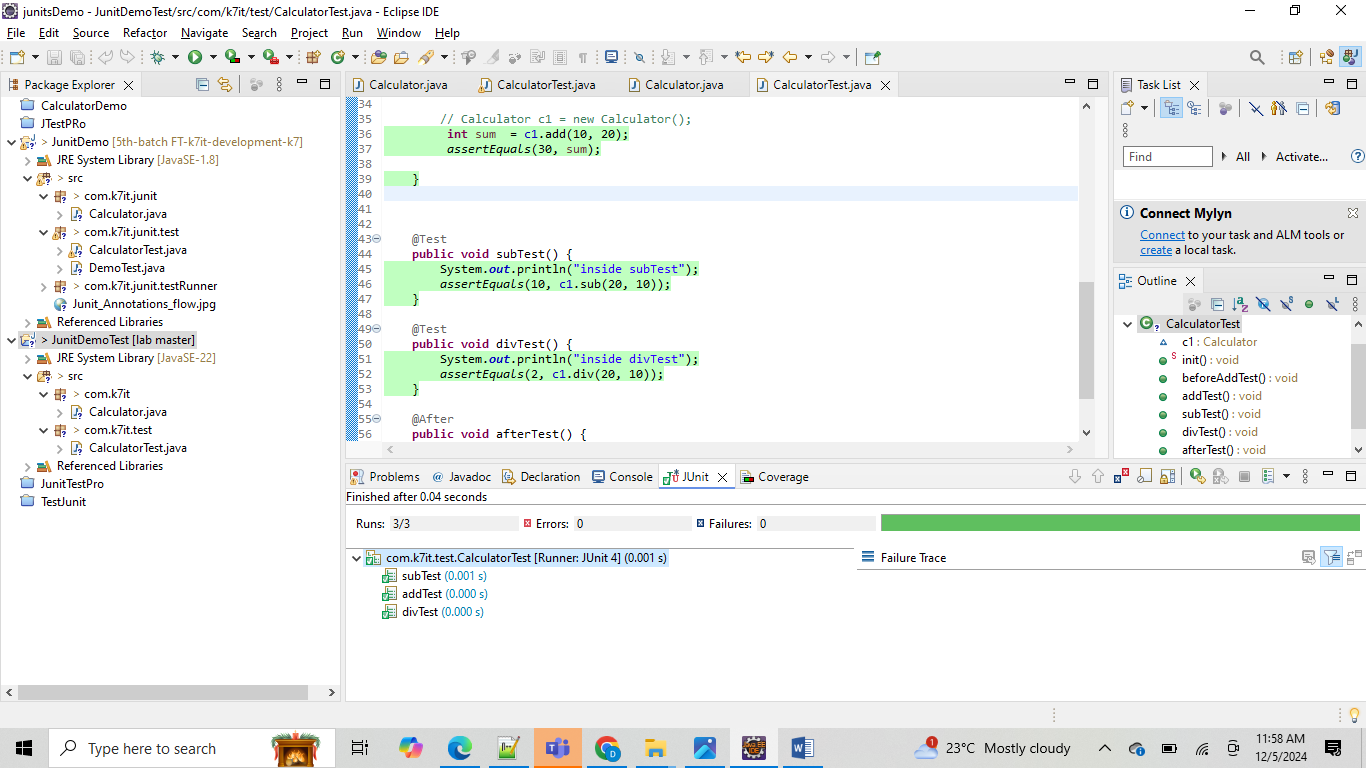
the @Test methdos and all the @After method.

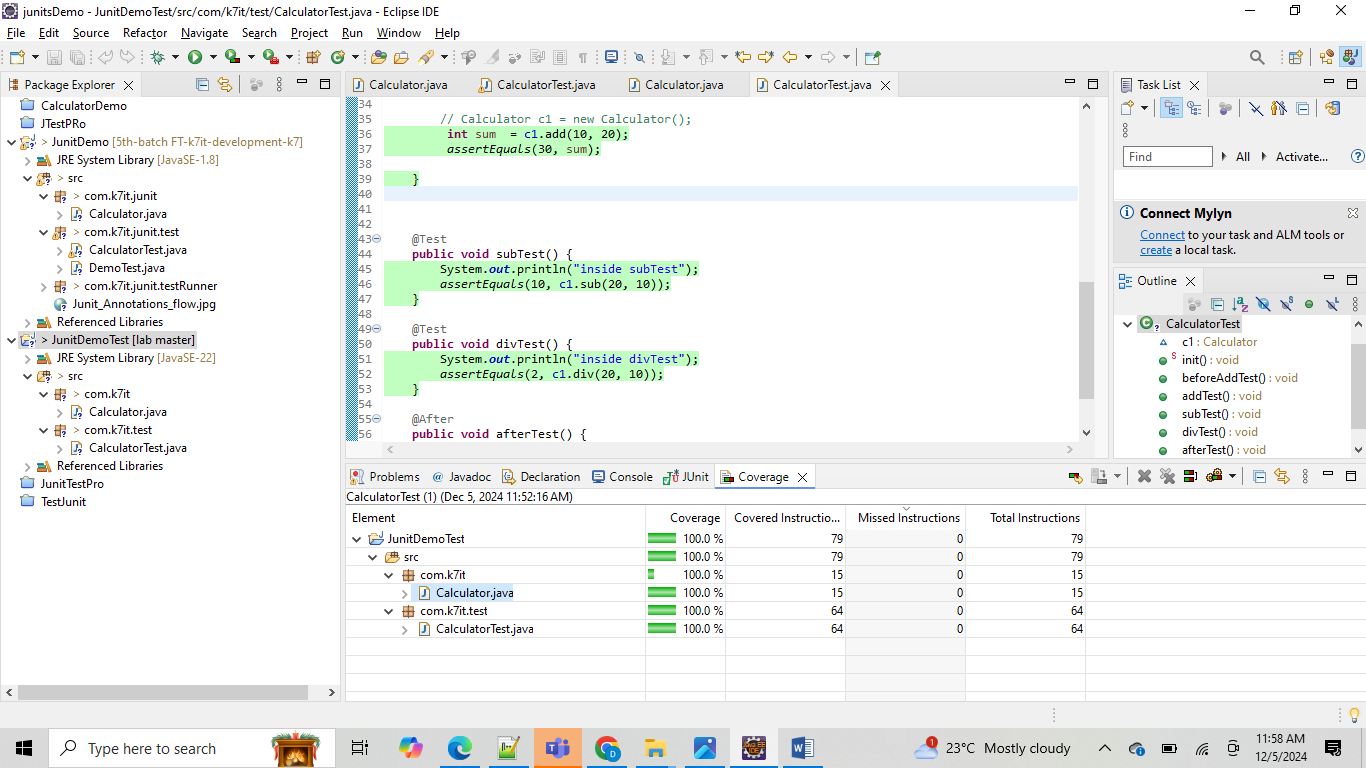
Code coverage :

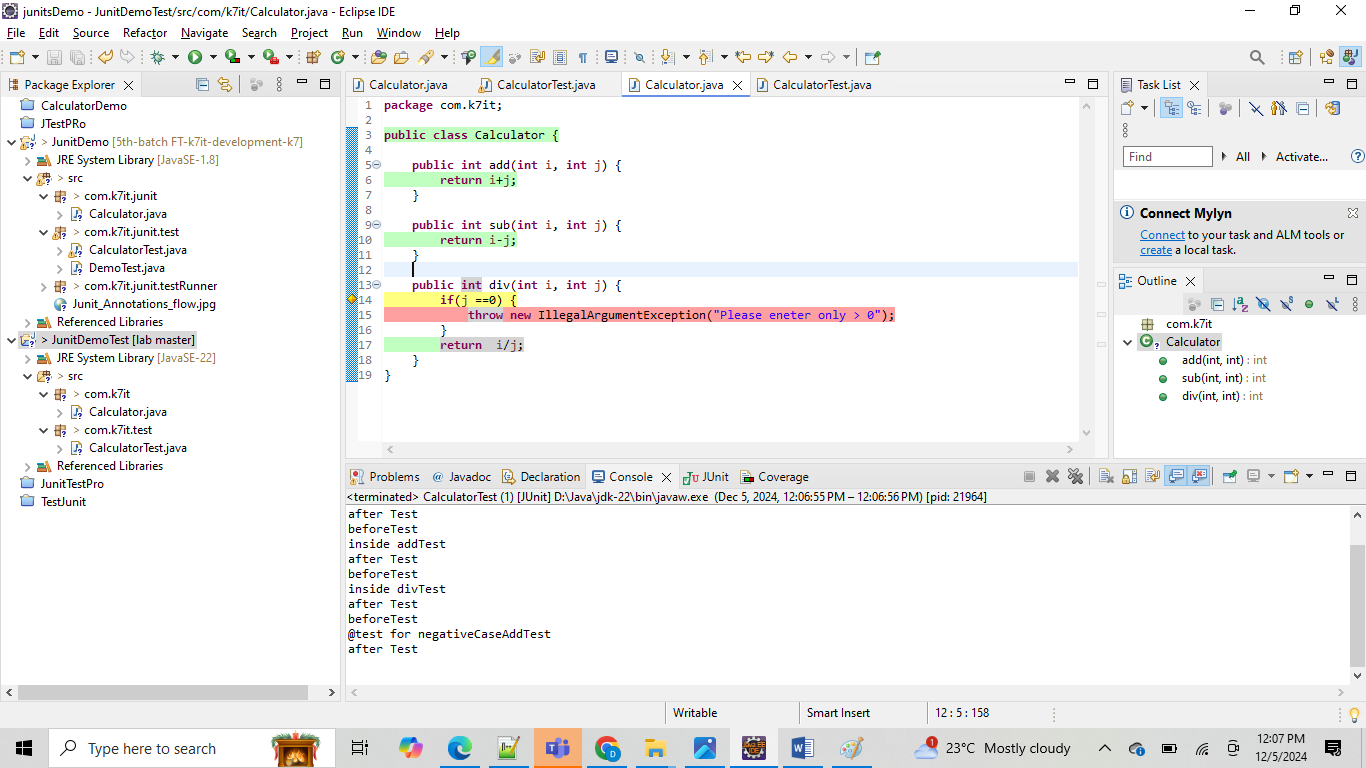
If we want to see the code coverage of our project, run our Junitclass using coverage as-> JunitTest



It will run all the testcases in junit class and it will open two tabs one for Junit results and one for code coverage result :







After running this option our code will highlighted with green color bar what ever the lines of code is covered by our junit test cases.

If we want to verify any validation exceptions from junit :

**package** com.k7it;

**public** **class** Calculator {

**public** **int** add(**int** i, **int** j) {

**return** i+j;

}

**public** **int** sub(**int** i, **int** j) {

**return** i-j;

}

**public** **int** div(**int** i, **int** j) **throws** IllegalArgumentException {

**if**(j ==0) {

**throw** **new** IllegalArgumentException("Please eneter only > 0");

}

**return** i/j;

}

}

**package** com.k7it.test;

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

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

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

**import** org.junit.After;

**import** org.junit.Before;

**import** org.junit.BeforeClass;

**import** org.junit.Test;

**import** com.k7it.Calculator;

**public** **class** CalculatorTest {

Calculator c1 = **null**;

@BeforeClass

**public** **static** **void** init() {

System.***out***.println("inside init method");

// c1 = new Calculator();

}

@Before

**public** **void** beforeAddTest() {

System.***out***.println("beforeTest");

c1 = **new** Calculator();

}

@Test

**public** **void** addTest() {

System.***out***.println("inside addTest");

// from here we need to verify our calculator class functionality

// Calculator c1 = new Calculator();

**int** sum = c1.add(10, 20);

*assertEquals*(30, sum);

}

@Test

**public** **void** subTest() {

System.***out***.println("inside subTest");

*assertEquals*(10, c1.sub(20, 10));

}

@Test

**public** **void** divTest() {

System.***out***.println("inside divTest");

*assertEquals*(2, c1.div(20, 10));

}

@After

**public** **void** afterTest() {

System.***out***.println("after Test");

c1 = **null**;

}

@Test

**public** **void** negativeCaseAddTest() {

System.***out***.println("@test for negativeCaseAddTest");

*assertNotEquals*(25, c1.add(20,50));

// 1st value is called what value we are expecting from calling method.

// 2nd parameter is actual value from method.

}

@Test

**public** **void** negativeCaseDivTest() {

*assertThrows*(IllegalArgumentException.**class**, () -> c1.div(10, 0));

}

}

Default code coverage tool in eclipse is: **JaCoCo : java code coverage.**

Here are some popular code coverage tools for Java projects:

1. **JaCoCo**: An open-source tool that provides detailed information about code coverage, including line, branch, and method coverage.
2. **Cobertura**: Tracks code execution by instrumenting bytecode and integrates well with build tools like Maven and Ant.
3. **Emma**: Allows for both offline and real-time instrumentation, providing insights into code coverage.
4. **Codecov**: A versatile tool that supports multiple languages, including Java, and integrates with various CI/CD pipelines.
5. **SonarQube**: Not only measures code coverage but also provides an overview of code quality by identifying duplications, bugs, and other issues.
6. **Clover**: Offers detailed code coverage reports and integrates with popular build tools and IDEs.
7. **Parasoft JTest**: Provides comprehensive code coverage analysis and integrates with various development environments.
8. **Jenkins**: While primarily a CI/CD tool, Jenkins can be configured to generate code coverage reports using plugins like JaCoCo.
9. **OpenClover**: A commercial tool that provides detailed code coverage metrics and integrates with various build tools.
10. **JCov**: A tool developed by Oracle that provides detailed code coverage analysis for Java applications.

Junits for Spring based classes:  
======================

If we want to write a junits for spring class we can write in two ways

1. Using @ExtendWith(MockitoExtension.**class**) and BDD
2. @ExtendWith(MockitoExtension.**class**) without BDD

If we want to write junits using above approach no need to add any additional dependencies in pom.xml, just normal spring dependencies are enough.

**package** com.k7it.sms.test.service;

**import** **static** org.junit.jupiter.api.Assertions.*assertEquals*;

**import** **static** org.junit.jupiter.api.Assertions.*assertNotNull*;

**import** **static** org.mockito.BDDMockito.*given*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.Arrays;

**import** java.util.List;

**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.k7it.sms.model.Person;

**import** com.k7it.sms.repo.PersonRepository;

**import** com.k7it.sms.service.PersonService;

@ExtendWith(MockitoExtension.**class**)

**public** **class** PersonServiceWithMockitoExtensionBDDTest {

@Mock

**private** PersonRepository personRepo;

@InjectMocks

**private** PersonService personService;

@Test

**public** **void** getAllPersonsTest() {

//

Person p1 = **new** Person(101,39,"k7it",5.5,60.26);

Person p2 = **new** Person(102,25,"java",5.6,55.50);

// when or given

//*given*(personRepo.findAll()).willReturn(Arrays.*asList*(p1,p2));

*when*(personRepo.findAll()).thenReturn(Arrays.*asList*(p1,p2));

List<Person> result = personService.getAllPersons();

*assertNotNull*(result);

*assertEquals*(2, result.size());

//then(result).isNotNull();

//then(result.size()).equals(2);

//then

// verify

}

}

Note: Here given().willReturn(); are part of the BDD framework methods.

If we want to use without BDD methods, we can go ahead with when().thenReturn(); methods

1. @ExtendWith(SpringExtension.**class**) or @RunWith(SpringRunner.**class**)

@SpringBootTest

Here we want to write the Junits based on above SpringRunner then we need to add

1 dependencies in pom.xml i.e junit4 related jar, since @RunWith() is annotation from junit4 . but @ExtendWith() annotation is junit5, spring by default its providing junit5 annotations automatically.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

**package** com.k7it.sms.test.service;

**import** **static** org.junit.jupiter.api.Assertions.*assertEquals*;

**import** **static** org.junit.jupiter.api.Assertions.*assertNotNull*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.Arrays;

**import** java.util.List;

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

**import** org.junit.runner.RunWith;

**import** org.mockito.InjectMocks;

**import** org.mockito.Mock;

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

**import** org.springframework.test.context.junit4.SpringRunner;

**import** com.k7it.sms.model.Person;

**import** com.k7it.sms.repo.PersonRepository;

**import** com.k7it.sms.service.PersonService;

//@ExtendWith(SpringExtension.class)

@RunWith(SpringRunner.**class**)

@SpringBootTest

**public** **class** PersonServiceWithSpringRunnerTest {

@Mock

**private** PersonRepository personRepository;

@InjectMocks

**private** PersonService personService;

@Test

**public** **void** getAllPersonTest() {

Person p1 = **new** Person(101,39,"k7it",5.5,60.25);

Person p2 = **new** Person(102,25,"java",5.6,55.5);

*when*(personRepository.findAll()).thenReturn( Arrays.*asList*(p1,p2));

List<Person> results = personService.getAllPersons();

*assertNotNull*(results);

*assertEquals*(2, results.size());

}

}

Same class we can rewrite with Junit5 also like bellow only deffrence is we need to change class level annotations   
@RunWith(SpringRunner.**class**) -> to -> @ExtendWith(SpringExtension.class)

Rest everything will be same

**package** com.k7it.sms.test.service;

**import** **static** org.junit.jupiter.api.Assertions.*assertEquals*;

**import** **static** org.junit.jupiter.api.Assertions.*assertNotNull*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.Arrays;

**import** java.util.List;

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

**import** org.junit.runner.RunWith;

**import** org.mockito.InjectMocks;

**import** org.mockito.Mock;

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

**import** org.springframework.test.context.junit4.SpringRunner;

**import** com.k7it.sms.model.Person;

**import** com.k7it.sms.repo.PersonRepository;

**import** com.k7it.sms.service.PersonService;

@ExtendWith(SpringExtension.class)

@SpringBootTest

**public** **class** PersonServiceWithSpringRunnerTest {

@Mock

**private** PersonRepository personRepository;

@InjectMocks

**private** PersonService personService;

@Test

**public** **void** getAllPersonTest() {

Person p1 = **new** Person(101,39,"k7it",5.5,60.25);

Person p2 = **new** Person(102,25,"java",5.6,55.5);

*when*(personRepository.findAll()).thenReturn( Arrays.*asList*(p1,p2));

List<Person> results = personService.getAllPersons();

*assertNotNull*(results);

*assertEquals*(2, results.size());

}

}

Both the cases main test method logic will be same , only deference is class level annotations will changes.

Note: when we want to use @Mock and @InjectMocks annotations:

For main class which we want to write Test method those classes we can use

@InjectMocks annotations , inside that class if we are using any derived data type attributes with @Autowired annotations for those classes we need to use

@Mock annotations before our main @InjectMocks.

Junit4 VS junit 5

==================

Switching from JUnit 4 to JUnit 5 brings several changes and improvements. Here are some key differences:

1. **Annotations**:
   * JUnit 4: Uses @Before, @After, @BeforeClass, and @AfterClass.
   * JUnit 5: Replaces these with @BeforeEach, @AfterEach, @BeforeAll, and @AfterAll.
2. **Test Runner**:
   * JUnit 4: Uses @RunWith to specify a test runner.
   * JUnit 5: Uses @ExtendWith to extend the test with additional functionality.
3. **Assertions**:
   * JUnit 4: Assertions are in the org.junit.Assert class.
   * JUnit 5: Assertions are in the org.junit.jupiter.api.Assertions class, with additional methods like assertThrows.
4. **Assumptions**:
   * JUnit 4: Uses org.junit.Assume for assumptions.
   * JUnit 5: Uses org.junit.jupiter.api.Assumptions.
5. **Parameterized Tests**:
   * JUnit 4: Uses @RunWith(Parameterized.class) for parameterized tests.
   * JUnit 5: Uses @ParameterizedTest along with @ValueSource, @CsvSource, etc.
6. **Tagging and Filtering**:
   * JUnit 4: Uses @Category for tagging tests.
   * JUnit 5: Uses @Tag for tagging and @IncludeTags/@ExcludeTags for filtering.
7. **Nested Tests**:
   * JUnit 4: Does not support nested tests.
   * JUnit 5: Supports nested tests with @Nested.
8. **Dynamic Tests**:
   * JUnit 4: Does not support dynamic tests.
   * JUnit 5: Supports dynamic tests with @TestFactory.
9. **Extensions**:
   * JUnit 4: Uses @Rule and @ClassRule for extensions.
   * JUnit 5: Uses the Extension API and @ExtendWith.
10. **Test Lifecycle**:
    * JUnit 4: Test lifecycle methods are executed in a specific order.
    * JUnit 5: Provides more flexibility with lifecycle methods and allows for more complex setups

Junit with positive and negative exceptions cases :

**package** com.k7it.sms.test.service;

**import** **static** org.junit.jupiter.api.Assertions.*assertEquals*;

**import** **static** org.junit.jupiter.api.Assertions.*assertNotNull*;

**import** **static** org.junit.jupiter.api.Assertions.*assertThrows*;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.Optional;

**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.k7it.sms.model.Person;

**import** com.k7it.sms.repo.PersonRepository;

**import** com.k7it.sms.service.PersonService;

@ExtendWith(MockitoExtension.**class**)

**public** **class** PersonServiceWithMockitoExtensionBDDTest {

@Mock

**private** PersonRepository personRepo;

@InjectMocks

**private** PersonService personService;

@Test

**public** **void** getAllPersonsTest() {

//

Person p1 = **new** Person(101,39,"k7it",5.5,60.26);

Person p2 = **new** Person(102,25,"java",5.6,55.50);

// when or given

//given(personRepo.findAll()).willReturn(Arrays.asList(p1,p2));

*when*(personRepo.findAll()).thenReturn(Arrays.*asList*(p1,p2));

List<Person> result = personService.getAllPersons();

*assertNotNull*(result);

*assertEquals*(2, result.size());

}

@Test

**public** **void** addPersonTest() {

Person p1 = **new** Person(101,39,"K7it",5.5,60.25);

*when*(personRepo.save(p1)).thenReturn(p1);

Person result = personService.addPerson(p1);

*assertNotNull*(result);

*assertEquals*(p1.getName(), result.getName());

}

@Test

**public** **void** updatePersonTest() {

Person p1 = **new** Person(101,39,"K7it",5.5,60.25);

*when*(personRepo.findById(101)).thenReturn(Optional.*of*(p1));

Person updatedPerson = **new** Person(101,40,"Java",5.5,70.25);

*when*(personRepo.save(p1)).thenReturn(updatedPerson);

Person result = personService.updatePerson(101,updatedPerson);

*assertNotNull*(result);

*assertEquals*("Java", result.getName());

}

@Test

**public** **void** deletePersonTest() {

Person p1 = **new** Person(101,39,"K7it",5.5,60.25);

*when*(personRepo.findById(101)).thenReturn(Optional.*of*(p1));

Person result = personService.deletePerson(101);

*assertNotNull*(result);

*assertEquals*("K7it", result.getName());

}

@Test

**public** **void** getPersonDetailsTest() {

Person p1 = **new** Person(101,39,"K7it",5.5,60.25);

*when*(personRepo.findById(101)).thenReturn(Optional.*of*(p1));

Person result = personService.getPersonDetails(101);

*assertNotNull*(result);

*assertEquals*("K7it", result.getName());

}

@Test

**public** **void** deletePersonWithExceptionTest() {

*when*(personRepo.findById(102)).thenReturn(Optional.*empty*());

*assertThrows*(IllegalArgumentException.**class**, () ->personService.deletePerson(102) );

}

@Test

**public** **void** updatePersonWithExceptionTest() {

Person p1 = **new** Person(101,39,"K7it",5.5,60.25);

*when*(personRepo.findById(102)).thenReturn(Optional.*empty*());

*assertThrows*(IllegalArgumentException.**class**, () ->personService.updatePerson(102,p1) );

}

@Test

**public** **void** getPersonDetailsWithExceptionTest() {

*when*(personRepo.findById(102)).thenReturn(Optional.*empty*());

*assertThrows*(IllegalArgumentException.**class**, () ->personService.getPersonDetails(102) );

}

}

Junits for Controller class:  
====================

If we want to write the junits for Controller class, we should setup the configuration similar to post man or front end code or browser to send the request and receive the response from our end points .

For any endpoint or api if we want to send request we need 3 things

1. Method type : GET, POST, PUT,DELETE,PATCH
2. Method url : “/api/v1/person” or “/api/v1/person/{id}” etc
3. Input params: like request params or request body or path variable or path param .
4. Content type : application/json

Important content types:

=====================

 **Textual Data**

* text/plain: Plain text (e.g., "Hello, world!")
* text/html: HTML content (e.g., a web page)
* text/css: CSS stylesheets
* text/javascript: JavaScript code
* text/xml: XML documents

 **Application Data**

* application/json: JSON data (e.g., {"name":"John", "age":30})
* application/xml: XML data
* application/x-www-form-urlencoded: Form data encoded in key-value pairs (e.g., name=John&age=30)
* application/octet-stream: Binary data (used for file downloads)

 **Multimedia Data**

* image/jpeg: JPEG images
* image/png: PNG images
* image/gif: GIF images
* audio/mpeg: MP3 audio
* audio/ogg: Ogg audio
* video/mp4: MP4 video
* video/webm: WebM video

 **Multipart Data**

* multipart/form-data: Used for submitting forms that include files

 **Other Specialized Types**

* application/pdf: PDF documents
* application/msword: Microsoft Word documents
* application/vnd.ms-excel: Microsoft Excel files
* application/vnd.openxmlformats-officedocument.wordprocessingml.document: Office Open XML document (Word)
* application/vnd.openxmlformats-officedocument.spreadsheetml.sheet: Office Open XML spreadsheet (Excel)

Now we need to see how to set above 4 things from Test class to test our controller class apis based on our api config.

To achieve these things from Test class we need few extra annotations and class from spring MVC model.

Class level:

1. @WebMvcTest(YourController.class)
2. @ExtendWith(MockitoExtension.class)

Class body:  
==========

1. We need to Autowired MvcMock class this is the main class which we are using to perform out API calls. Using mvcMock.perform() method.

@Autowired

**private** MockMvc mockMvc; // this is class which we are going to use to perform end point access

// operations like post man.

1. We need to mock the required derived Attributes which are using inside our Controller class and their dependency classes . like Service class and Repository class using @MockBean   
   @~~MockBean~~

**private** PersonRepository personRepository;

@~~MockBean~~ **private** PersonService personService;

1. Setup the mvcMock object for every Test method using @BeforeEach

@BeforeEach

**public** **void** setup() {

***log***.info("I am at setup method");

mockMvc = MockMvcBuilders.*standaloneSetup*(**new** PersonController()).build();

}

Method level codes:

* 1. In side method level as same as normal Test methods we need to start with @Test annotation

@Test

Public void ourTestmethod(){

}

* 1. Perform the our API call action based on our API method type and Input param type with the help mockMvc.perform() and it will take one input param i.e corresponding API method type i.e get or post or put etc.

For each method type there is one predefined methods inside

MockMvcRequestBuilders class as static methods.

Like MockMvcRequestBuilders.get() or MockMvcRequestBuilders.post()

Or MockMvcRequestBuilders.put() or MockMvcRequestBuilders.delete()

If we don’t want to use this MockMvcRequestBuilders. Everytime for all methdos then use static import for each method separately

**import** **static** org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*get;*

**import** **static** org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*POST;*

here again inside each method of get() or post() or put() they will expect one string parameter that is our API end point url

mockMvc.perform(MockMvcRequestBuilders.get("/api/v1/test"));// without static import

mockMvc.perform(get("/api/v1/test"));// with static import

* 1. Verify the status of our api end point response either success or failure using andExpect() from ResultActions object which is output from perform() method.

mockMvc.perform(MockMvcRequestBuilders.get("/api/v1/test")).andExpect(MockMvcResultMatchers.*status*().isOk());

here

MockMvcResultMatchers.*status*().isOk() : this method will verify either our api is returning 200 status code or not. If we get 200 our testcase will get success or pass else testcase will fail.

* 1. Incase status is 200 then we can verify result of the api or return values of the api response of the api for this we need to take another andExpect() with help of content() from MockMvcResultMatchers class based on our method return type we need to select next following methods

.string() or .json() or .xml() etc . this methods will decide by contentType of our api response

mockMvc.perform(MockMvcRequestBuilders.get("/api/v1/test"))

.andExpect(MockMvcResultMatchers.*status*().isOk())

.andExpect(MockMvcResultMatchers.*content*().string("Hello World"));

If we want to set the content type for request or response we can set like bellow

mockMvc.perform(*get*("/api/v1/test")).andExpect(MockMvcResultMatchers.*status*().isOk())

.andExpect(MockMvcResultMatchers.*content*().contentType("application/json"))

.andExpect(MockMvcResultMatchers.*content*().string("Hello World"));

Example:

**package** com.k7it.sms.test.controller;

**import** **static** org.mockito.Mockito.*when*;

**import** **static** org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*get*;

**import** org.springframework.test.web.servlet.result.MockMvcResultMatchers;

**import** **static** org.springframework.test.web.servlet.setup.MockMvcBuilders.*standaloneSetup*;

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

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

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

**import** org.mockito.Mock;

**import** org.mockito.junit.jupiter.MockitoExtension;

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

**import** org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

**import** org.springframework.boot.test.mock.mockito.~~MockBean~~;

**import** org.springframework.test.context.bean.override.mockito.MockitoBean;

**import** org.springframework.test.web.servlet.MockMvc;

**import** org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

**import** org.springframework.test.web.servlet.setup.MockMvcBuilders;

**import** com.k7it.sms.controller.PersonController;

**import** com.k7it.sms.model.Person;

**import** com.k7it.sms.repo.PersonRepository;

**import** com.k7it.sms.service.PersonService;

**import** lombok.extern.slf4j.Slf4j;

@ExtendWith(MockitoExtension.**class**)

@WebMvcTest(PersonController.**class**) // it will provide browser based control or post man based controlls or front

// end based control

// for sending the request and receiving the response

@Slf4j

**public** **class** PersonControllerTest {

@Autowired

**private** MockMvc mockMvc; // this is class which we are going to use to perform end point access

// operations like post man.

@~~MockBean~~

**private** PersonRepository personRepository;

@~~MockBean~~

**private** PersonService personService;

@BeforeEach

**public** **void** setup() {

***log***.info(">>>>>>>>>>>>>>>>>>>>>>I am at setup method<<<<<<<<<<<<<<<<<<<<<<<<<<<<");

mockMvc = MockMvcBuilders.*standaloneSetup*(**new** PersonController()).build();

}

@Test

**public** **void** addPersonTest() **throws** Exception {

Person p1 = **new** Person(101,39,"k7it",5.6,60.25);

*when*(personService.addPerson(p1)).thenReturn(p1);

String reqBody = "{\"id\":101,\"age\":39,\"name\":\"k7it\",\"height\":5.6,\"weight\":60.25}";

mockMvc.perform(

MockMvcRequestBuilders.*post*("/api/v1/person")

.contentType("application/json")

.content(reqBody)

).andExpect(MockMvcResultMatchers.*status*().isOk())

.andExpect(MockMvcResultMatchers.*content*().contentType("application/json"))

.andExpect(MockMvcResultMatchers.*content*().json(reqBody));

}

}

Here in this code if we try to run addPersonTest() we will get the output as NullPointerException from Controller class method

Where we are using personService.addPerson(p1); why bzc inside setup()

We are initializing mockMvc obj with our controller class object using no arg constructor so all dependency injections will be null inside Controller class.

If we want to avoid this error we need to reinitialize mockMvc with WebApplicationContect class object like bellow.

@Autowired

**private** WebApplicationContext webApplicationContext;

@BeforeEach

**public** **void** setup() {

***log***.info(">>>>>>>>>>>>>>>>>>>>>>I am at setup method<<<<<<<<<<<<<<<<<<<<<<<<<<<<");

mockMvc = MockMvcBuilders.*webAppContextSetup*(webApplicationContext).build();

}

Full code for PersonController Test class:

**package** com.k7it.sms.test.controller;

**import** **static** org.mockito.Mockito.*when*;

**import** java.util.Arrays;

**import** java.util.List;

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

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

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

**import** org.mockito.junit.jupiter.MockitoExtension;

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

**import** org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

**import** org.springframework.boot.test.mock.mockito.~~MockBean~~;

**import** org.springframework.test.web.servlet.MockMvc;

**import** org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

**import** org.springframework.test.web.servlet.result.MockMvcResultMatchers;

**import** org.springframework.test.web.servlet.setup.MockMvcBuilders;

**import** org.springframework.web.context.WebApplicationContext;

**import** com.fasterxml.jackson.databind.ObjectMapper;

**import** com.k7it.sms.controller.PersonController;

**import** com.k7it.sms.model.Person;

**import** com.k7it.sms.repo.PersonRepository;

**import** com.k7it.sms.service.PersonService;

**import** lombok.extern.slf4j.Slf4j;

@ExtendWith(MockitoExtension.**class**)

@WebMvcTest(PersonController.**class**) // it will provide browser based control or post man based controlls or front

// end based control

// for sending the request and receiving the response

@Slf4j

**public** **class** PersonControllerTest {

@Autowired

**private** MockMvc mockMvc; // this is class which we are going to use to perform end point access

// operations like post man.

@~~MockBean~~

**private** PersonRepository personRepository;

@~~MockBean~~

**private** PersonService personService;

@Autowired

**private** WebApplicationContext webApplicationContext;

@Autowired

**private** ObjectMapper objectMapper;

@BeforeEach

**public** **void** setup() {

***log***.info(">>>>>>>>>>>>>>>>>>>>>>I am at setup method<<<<<<<<<<<<<<<<<<<<<<<<<<<<");

mockMvc = MockMvcBuilders.*webAppContextSetup*(webApplicationContext).build();

}

@Test

**public** **void** addPersonTest() **throws** Exception {

Person p1 = **new** Person(101, 39, "k7it", 5.6, 60.25);

*when*(personService.addPerson(p1)).thenReturn(p1);

String reqBody = "{\"id\":101,\"age\":39,\"name\":\"k7it\",\"height\":5.6,\"weight\":60.25}";

mockMvc.perform(MockMvcRequestBuilders.*post*("/api/v1/person").contentType("application/json").content(reqBody))

.andExpect(MockMvcResultMatchers.*status*().isOk())

.andExpect(MockMvcResultMatchers.*content*().contentType("application/json"))

.andExpect(MockMvcResultMatchers.*content*().json(reqBody));

}

@Test

**public** **void** updatePersonTest() **throws** Exception {

Person p1 = **new** Person(101, 40, "java", 5.6, 60.25);

*when*(personService.updatePerson(101, p1)).thenReturn(p1);

String updateRequestBody = objectMapper.writeValueAsString(p1);// "{\"id\":101,\"age\":40,\"name\":\"java\",\"height\":5.6,\"weight\":60.25}";

***log***.info(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>"+updateRequestBody+"<<<<<<<<<<<<<<<<<<<<<<<<<<<");

mockMvc.perform(MockMvcRequestBuilders.*put*("/api/v1/person/101").contentType("application/json")

.content(updateRequestBody)).andExpect(MockMvcResultMatchers.*status*().isOk())

.andExpect(MockMvcResultMatchers.*content*().contentType("application/json"))

.andExpect(MockMvcResultMatchers.*content*().json(updateRequestBody));

}

@Test

**public** **void** deletePersonTest() **throws** Exception {

Person p1 = **new** Person(101, 40, "java", 5.6, 60.25);

*when*(personService.deletePerson(101)).thenReturn(p1);

mockMvc.perform(MockMvcRequestBuilders.*delete*("/api/v1/person/101"))

.andExpect(MockMvcResultMatchers.*status*().isOk())

.andExpect(MockMvcResultMatchers.*content*().contentType("application/json"))

.andExpect(MockMvcResultMatchers.*content*()

.json("{\"id\":101,\"age\":40,\"name\":\"java\",\"height\":5.6,\"weight\":60.25}"));

}

@Test

**public** **void** getPersonDetailsTest() **throws** Exception {

Person p1 = **new** Person(101, 40, "java", 5.6, 60.25);

*when*(personService.getPersonDetails(101)).thenReturn(p1);

String result = "{\"id\":101,\"age\":40,\"name\":\"java\",\"height\":5.6,\"weight\":60.25}";

mockMvc.perform(MockMvcRequestBuilders.*get*("/api/v1/person/101"))

.andExpect(MockMvcResultMatchers.*status*().isOk())

.andExpect(MockMvcResultMatchers.*content*().contentType("application/json"))

.andExpect(MockMvcResultMatchers.*content*().json(result));

}

@Test

**public** **void** getAllPeronsTest() **throws** Exception {

Person p1 = **new** Person(101, 40, "java", 5.6, 60.25);

Person p2 = **new** Person(102, 50, "mongodb", 6.0, 90.25);

List<Person> listOfPersons = Arrays.*asList*(p1, p2);

*when*(personService.getAllPersons()).thenReturn(listOfPersons);

String resultJson = "[" + "{\"id\":101,\"age\":40,\"name\":\"java\",\"height\":5.6,\"weight\":60.25},"

+ "{\"id\":102,\"age\":50,\"name\":\"mongodb\",\"height\":6.0,\"weight\":90.25}" + "]";

mockMvc.perform(MockMvcRequestBuilders.*get*("/api/v1/persons")).andExpect(MockMvcResultMatchers.*status*().isOk())

.andExpect(MockMvcResultMatchers.*content*().contentType("application/json"))

.andExpect(MockMvcResultMatchers.*content*().json(resultJson));

}

@Test

**public** **void** getHelloWordTest() **throws** Exception {

mockMvc.perform(MockMvcRequestBuilders.*get*("/api/v1/test")).andExpect(MockMvcResultMatchers.*status*().isOk())

.andExpect(MockMvcResultMatchers.*content*().string("Hello World"));

}

}

package com.k7it.sms.controller;

import java.util.List;

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

import org.springframework.web.bind.annotation.DeleteMapping;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.PutMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import com.k7it.sms.model.Person;

import com.k7it.sms.model.PersonDTO;

import com.k7it.sms.service.PersonService;

@RestController

@RequestMapping("api/v1")

public class PersonController {

@Autowired

PersonService personService;

@PostMapping("/person") // classlevel +method level

public Person addPerson(@RequestBody PersonDTO personDTO) {

Person person = new Person(personDTO.getId(),personDTO.getAge(),

personDTO.getName(),personDTO.getHeight(),personDTO.getWeight());

return personService.addPerson(person);

}

@PutMapping("/person/{id}")

public Person updatePerson(@PathVariable int id,@RequestBody Person person) {

return personService.updatePerson(id,person);

}

@DeleteMapping("/person/{id}")

public Person deletePerson(@PathVariable int id) {

return personService.deletePerson(id);

}

@GetMapping("/person/{id}")

public Person getPersonDetails(@PathVariable int id) {

return personService.getPersonDetails(id);

}

@GetMapping("/persons")

public List<Person> getAllPerons(){

return personService.getAllPersons();

}

@GetMapping("/test")

public String getHelloWord(){

return "Hello World";

}

}

package com.k7it.sms.service;

import java.util.List;

import java.util.Optional;

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

import org.springframework.stereotype.Service;

import com.k7it.sms.model.Person;

import com.k7it.sms.repo.PersonRepository;

import com.k7it.sms.repo.StudentRepo;

@Service

public class PersonService {

@Autowired

PersonRepository personRepo;

public Person addPerson(Person person) {

return personRepo.save(person);

}

public Person updatePerson(int id, Person person) {

// 1st get the person details with given id

Optional<Person> dbPerson = personRepo.findById(id);

if(dbPerson.isPresent()) {

Person p= dbPerson.get();

p.setAge(person.getAge());

p.setName(person.getName());

p.setHeight(person.getHeight());

p.setWeight(person.getWeight());

return personRepo.save(p);

}else {

throw new IllegalArgumentException("There is no Person avaiable with given Id: please try with valid id");

}

}

public Person deletePerson(int id) {

Optional<Person> dbPerson = personRepo.findById(id);

if(dbPerson.isPresent()) {

personRepo.deleteById(id);

return dbPerson.get();

}else {

throw new IllegalArgumentException("There is no Person avaiable with given Id: please try with valid id");

}

}

public Person getPersonDetails(int id) {

Optional<Person> dbPerson = personRepo.findById(id);

if(dbPerson.isPresent()) {

return dbPerson.get();

}else {

throw new IllegalArgumentException("There is no Person avaiable with given Id: please try with valid id");

}

}

public List<Person> getAllPersons() {

return personRepo.findAll();

}

}

package com.k7it.sms.repo;

import org.springframework.data.mongodb.repository.MongoRepository;

import com.k7it.sms.model.Person;

public interface PersonRepository extends MongoRepository<Person, Integer>{

}

Note: in above code when ever we need our API result in JSON format we are trying to create json string our self like this   
  
 String result = "{\"id\":101,\"age\":40,\"name\":\"java\",\"height\":5.6,\"weight\":60.25}";

Instead of converting our derived class Object into Json format directly. If we want to achieve this process easier we can use one third party library i.e Jackson.databind dependency jars in this jar we can take ObjectMapper class help to covert java class objects into JSON format.

Add this maven dependency in pom.xml

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.14.1</version>

</dependency>

Then create Object for ObjectMapper class using @Autowired in spring boot or new operator in normal java project.

Example:

@ExtendWith(MockitoExtension.**class**)

@WebMvcTest(PersonController.**class**)

@Slf4j

**public** **class** PersonControllerTest {

@Autowired

**private** ObjectMapper objectMapper;

@Test

**public** **void** updatePersonTest() **throws** Exception {

Person p1 = **new** Person(101, 40, "java", 5.6, 60.25);

*when*(personService.updatePerson(101, p1)).thenReturn(p1);

String updateRequestBody = objectMapper.writeValueAsString(p1);// "{\"id\":101,\"age\":40,\"name\":\"java\",\"height\":5.6,\"weight\":60.25}";

***log***.info(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>"+updateRequestBody+"<<<<<<<<<<<<<<<<<<<<<<<<<<<");

mockMvc.perform(MockMvcRequestBuilders.*put*("/api/v1/person/101").contentType("application/json")

.content(updateRequestBody)).andExpect(MockMvcResultMatchers.*status*().isOk())

.andExpect(MockMvcResultMatchers.*content*().contentType("application/json"))

.andExpect(MockMvcResultMatchers.*content*().json(updateRequestBody));

}

}

Output in console:

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>{"id":101,"age":40,"name":"java","height":5.6,"weight":60.25}<<<<<<<<<<<<<<<<<<<<<<<<<<<

How to convert JSON String to Java class Objects :

String result = "{\"id\":101,\"age\":40,\"name\":\"java\",\"height\":5.6,\"weight\":60.25}";

Person p1 = objectMapper.readValue(result, Person.**class**);