**13. Step 08 - More Refactoring - @Mock, @InjectMocks and @RunWith(MockitoJUnitRunner**

14. Step 09 - Mockito Tips - Multiple Return Values and Specific Argument Matchers

15. Step 10 - Mockito Tips - Argument Matchers: How to make generic parameters:

16. Step 11 - Mockito Tips - Verify method calls

//17. Step 12 - Mockito Tips - Argument Capture: WE can check the passed value in the method call using ArgumentCaptor

18. Step 13 - Mockito Tips - Argument Capture on Multiple Calls

27. Step 04 - Creating a Basic REST Service in Item Controller

28. Step 05 - Unit Testing Item Controller and Basic JSON Assertions

29. Step 06 - Digging deeper into JSON Assert

|  |
| --- |
| **Step 02 - Writing Unit Test for a Simple Business Service**  **package** com.junit.test.business;  **import** com.junit.test.business.dataservice.DataService;  **public** **class** SomeBusinessImpl {  **public** **int** CalculateSum(**int**[] data) {    **int** sum=0;  **for**(**int** value : data) {  sum=sum+value;  }  **return** sum;  }  } |
| **package** com.junit.test.business;  **import** **static** org.junit.Assert.\*;  **import** org.junit.Test;  **public** **class** SomeBusinessImplTest {  @Test  **public** **void** Calculate\_Basic() {  SomeBusinessImpl testBusiness = **new** SomeBusinessImpl();  **int** actualValue = testBusiness.CalculateSum(**new** **int**[] {1,2,3});  **int** expectedValue=6;  *assertEquals*(expectedValue, actualValue);    }  @Test  **public** **void** Calculate\_Empty() {    SomeBusinessImpl testBussiness = **new** SomeBusinessImpl();  **int** actualValue = testBussiness.CalculateSum(**new** **int**[] {});  **int** expectedValue =0;  *assertEquals*(expectedValue,actualValue);  }  @Test  **public** **void** Calculate\_One() {  SomeBusinessImpl testOne = **new** SomeBusinessImpl();  **int** actualValue = testOne.CalculateSum(**new** **int**[] {5});  **int** expectedValue=5;  *assertEquals*(expectedValue,actualValue);  }  } |

1. Here in the above Unit testing: we are performing unit testing using assertEquals(expected, actual);
2. The actual value is getting just by calling the actual method and passing the int array value which will calculate and return the value.
3. Now using assertEquals(expectedValue, actualValue); will do the unit testing

**Step 03 - Setting up a Business Service to call a Data Service TestUsingMockito**

|  |
| --- |
| **package** com.junit.test.business.dataservice;  **public** **interface** DataService {  **public** **int**[] retriveAllData();  }  Created an interface, an implementation class will implement the method of I and will be invoked in the Controller class method |
| **package** com.junit.test.business;  **import** com.junit.test.business.dataservice.DataService;  **public** **class** BusinessImplRetriveDataFromService {  **private** DataService dataService;  **public** **void** setDataService(DataService dataService) {  **this**.dataService = dataService;  }  **public** **int** CalculateSumUsingDataService() {  **int** sum = 0;  **int**[] actualData = dataService.retriveAllData();  **for** (**int** value : actualData) {  System.***out***.println("actualData::" + value);  sum = sum + value;  } **return** sum;  }  }  Here we are retrieving the data from the interface implemented class |
| **Now test the above** CalculateSumUsingDataService method using stubs  **Writing your first unit test with Stub**  **package** com.junit.test.business.dataservice.using.stubs;  **import** **static** org.junit.jupiter.api.Assertions.*assertEquals*;  **import** org.junit.jupiter.api.Test;  **import** com.junit.test.business.BusinessImplRetriveDataFromService;  **import** com.junit.test.business.dataservice.DataService;  **class** DataServiceStub **implements** DataService{  @Override  **public** **int**[] retriveAllData() {  **return** **new** **int**[] {1,2,3};  }  }  **class** DataServiceStub\_Empty **implements** DataService{  @Override  **public** **int**[] retriveAllData() {  **return** **new** **int**[] {};  }  }  **class** DataServiceStub\_One **implements** DataService{  @Override  **public** **int**[] retriveAllData() {  **return** **new** **int**[] {5};  }  }  **class** BusinessImplTestUsingStub {  BusinessImplRetriveDataFromService test= **new** BusinessImplRetriveDataFromService();    @Test  **void** CalculateSumUsingDataServiceStub() {  test.setDataService(**new** DataServiceStub());  **int** actualValue = test.CalculateSumUsingDataService();  **int** expectedValue=6;  *assertEquals*(expectedValue,actualValue);  }    @Test  **void** CalculateSumUsingDataServiceStub\_Empty() {  test.setDataService(**new** DataServiceStub\_Empty());  **int** actualValue=test.CalculateSumUsingDataService();  **int** expectedValue=0;  *assertEquals*(expectedValue,expectedValue);  }  @Test  **void** CalculateSumUsingDataServiceStub\_One() {  test.setDataService(**new** DataServiceStub\_One());  *assertEquals*(5,test.CalculateSumUsingDataService());  }  }  // AS we can see that Junit test case using stub is very -2lengthy because for each cases we have to create class and since here we are using Interface for implementing stubs so  // in future if we add a new method in the interface then this newly created method must be implemented in all Stub classes otherwise we will get compilation error. So it become more complex  // and its not easy to maintain.  // And this is the reason Mocking came into picture. So instead of creating several stub classes we can programmatically create classes using mock. |
| **Solution: Writing Unit Tests with Mocking using Mockito**  **package** com.junit.test.business.dataservice.using.mock;  **import** **static** org.junit.Assert.*assertEquals*;  **import** **static** org.mockito.Mockito.*atLeastOnce*;  **import** **static** org.mockito.Mockito.*atMost*;  **import** **static** org.mockito.Mockito.*mock*;  **import** **static** org.mockito.Mockito.*times*;  **import** **static** org.mockito.Mockito.*verify*;  **import** **static** org.mockito.Mockito.*when*;  **import** org.junit.Test;  **import** com.junit.test.business.BusinessImplRetriveDataFromService;  **import** com.junit.test.business.dataservice.DataService;  **public** **class** BusinessImplTestUsingMockito {  BusinessImplRetriveDataFromService businessImpl = **new** BusinessImplRetriveDataFromService();  @Test  **public** **void** CalculateSumWithMock\_Basic() {    DataService dataService = *mock*(DataService.**class**);  ***when*(dataService.retriveAllData()).thenReturn(new int[] {1,2,3,4});**  businessImpl.setDataService(dataService);  **int** actualData = businessImpl.CalculateSumUsingDataService();  **int** expectedData=10;  *assertEquals*(expectedData,actualData);  }  Here instead of writing stubs we have used the Mockito.   1. Here we will have to mock the DataService Interface. And for that we will create the mock object of the DataService Interface using Mockito static method mock (). It will return the mock object of the interface.   **Note**: As we know that, we cannot create object of an interface, but using Mockito we can create object of the interface or class. So Here we are performing the unit testing of service class, the class or interface to which unit testing has to perform using Mockito then first we will create mock object of that particular Interface or class   1. Now using Mockito static method when we will perform the mocking of the DataService interface method retriveAllData()   **Note: In the above line thenReturn (new int[] {1,2,3,4}) create implementation class of retriveAllData() and set it to dataService. Since here we are creating our own mocking object so in case service class which retrieves the data from DB fails, we will be able to perform unit testing.** So here we are setting or returning the mocking object (data) for implementation class retriveAllData () and this mocking data only we are trying to perform unit testing. So here thenReturn create the stubs for us internally, we don’t need to create stubs explicitly.  @Test  **public** **void** CalculateSumWithMock\_One() {  DataService dataService = *mock*(DataService.**class**);  *when*(dataService.retriveAllData()).thenReturn(**new** **int**[] {18});  businessImpl.setDataService(dataService);  **int** actualData = businessImpl.CalculateSumUsingDataService();  **int** expectedData= 18;  *assertEquals*(expectedData,actualData);  }    **// Mocking for one element of array**  @Test  **public** **void** CalculateSumWithMock\_Empty() {  DataService dataService = *mock*(DataService.**class**);  *when*(dataService.retriveAllData()).thenReturn(**new** **int**[] {});  businessImpl.setDataService(dataService);  **int** actualData = businessImpl.CalculateSumUsingDataService();  **int** expectedData=0;  *assertEquals*(expectedData,actualData);    *verify*(dataService).retriveAllData();  *verify*(dataService, *times*(1)).retriveAllData();  *verify*(dataService,*atLeastOnce*()).retriveAllData();  *verify*(dataService, *atMost*(2)).retriveAllData();  }  }  **// Mocking for 0 element of array**  //So here if we compare with Test using Stub class then there we have to create the stub classes for each cases, while in case of Mocking we don't need to create stub classes separately, the stub kind of classes gets created internally using (thenReturn() it Stub class on fly by overriding :  /\*  @Override  public OngoingStubbing<T> thenReturn(T value) {  return thenAnswer(new Returns(value));  }  \*/ |
| **This is the code re-factor to reduce the boiler code in each method:**  BusinessImplRetriveDataFromService businessImpl = **new** BusinessImplRetriveDataFromService();  DataService dataService = *mock*(DataService.**class**);  // And the below line will be executed before executing the below test methods.  @Before  **public** **void** before() {  businessImpl.setDataService(dataService);  }  @Test  **public** **void** CalculateSumWithMock\_Basic() {    *when*(dataService.retriveAllData()).thenReturn(**new** **int**[] {1,2,3,4});  **int** actualData = businessImpl.CalculateSumUsingDataService();  **int** expectedData=10;  *assertEquals*(expectedData,actualData);  }  @Test  **public** **void** CalculateSumWithMock\_One() {    *when*(dataService.retriveAllData()).thenReturn(**new** **int**[] {18});  **int** actualData = businessImpl.CalculateSumUsingDataService();  **int** expectedData= 18;  *assertEquals*(expectedData,actualData);  }  @Test  **public** **void** CalculateSumWithMock\_Empty() {    *when*(dataService.retriveAllData()).thenReturn(**new** **int**[] {});  **int** actualData = businessImpl.CalculateSumUsingDataService();  **int** expectedData=0;  *assertEquals*(expectedData,actualData);    *verify*(dataService).retriveAllData(); // verifying the invoking of method  *verify*(dataService, *times*(1)).retriveAllData(); // Verifying only one time  *verify*(dataService,*atLeastOnce*()).retriveAllData(); // At least one time  *verify*(dataService, *atMost*(2)).retriveAllData(); // Max 2 times  }  }  Note: Here apart from using assertEquals we can use static method verify just to verify that method got invoked or not |

**More Refactoring Using - @Mock, @InjectMocks and @RunWith (MockitoJUnitRunner)**

As in the above steps we have used static method mock to create object mock object and then set this object to Interface setter method, we are going to use annotation provided by Mockito framework.

|  |  |
| --- | --- |
| **@RunWith (MockitoJUnitRunner):** | Used at class level so that we could have all the feature inside this class and we can have all the annotation related to Mockito |
| **@InjectMocks** | We can use to create the object of class just like as @Autowired annotation to inject its method here. In the previous example we have created object of the class using new operator and then we have called its method. |
| **@Mock** | As in the previous example we have created the mock object of interface using static method mock and then same mock object we have set inside the setter method. But now we can do the same thing just by using @Mock annotation and we don’t need to set this mock object inside the setter method. |

So below is the re-factored code of the above example.

|  |
| --- |
| **package** com.junit.test.business.dataservice.using.mock;  **import** **static** org.junit.Assert.*assertEquals*;  **import** **static** org.mockito.Mockito.*atLeastOnce*;  **import** **static** org.mockito.Mockito.*atMost*;  **import** **static** org.mockito.Mockito.*times*;  **import** **static** org.mockito.Mockito.*verify*;  **import** **static** org.mockito.Mockito.*when*;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.mockito.InjectMocks;  **import** org.mockito.Mock;  **import** org.mockito.junit.MockitoJUnitRunner;  **import** com.junit.test.business.BusinessImplRetriveDataFromService;  **import** com.junit.test.business.dataservice.DataService;  @RunWith(MockitoJUnitRunner.**class**)  **public** **class** BusinessImplTestUsingMockitoCodeRefactorUsintInject\_RunnerWith {  // Here we have re-factor the code from the previous Instead of declaring in each class we have  // BusinessImplRetriveDataFromService businessImpl = new BusinessImplRetriveDataFromService ();  @InjectMocks  BusinessImplRetriveDataFromService businessImpl;  // DataService dataService = mock(DataService.class);  @Mock  DataService dataService;    // And the below line line of code (@Before) is not required because @Mock will create the Mock object of DataService and set it automatically in its setter method. So let’s comment the setter  /\* @Before  public void before() {  businessImpl.setDataService(dataService);  }  \*/  @Test  **public** **void** CalculateSumWithMock\_Basic() {    *when*(dataService.retriveAllData()).thenReturn(**new** **int**[] {1,2,3,4});  **int** actualData = businessImpl.CalculateSumUsingDataService();  **int** expectedData=10;  *assertEquals*(expectedData,actualData);  }  @Test  **public** **void** CalculateSumWithMock\_One() {    *when*(dataService.retriveAllData()).thenReturn(**new** **int**[] {18});  **int** actualData = businessImpl.CalculateSumUsingDataService();  **int** expectedData= 18;  *assertEquals*(expectedData,actualData);  }  @Test  **public** **void** CalculateSumWithMock\_Empty() {    *when*(dataService.retriveAllData()).thenReturn(**new** **int**[] {});  **int** actualData = businessImpl.CalculateSumUsingDataService();  **int** expectedData=0;  *assertEquals*(expectedData,actualData);    *verify*(dataService).retriveAllData();  *verify*(dataService, *times*(1)).retriveAllData();  *verify*(dataService,*atLeastOnce*()).retriveAllData();  *verify*(dataService, *atMost*(2)).retriveAllData();  }  }  1- We have used @RunWith(MockitoJUnitRunner.class) at the class level  2- Inject the SomeBusinessImpl class method using @@InjectMocks annotations just similar to @AutoWired  3- Used @Mock annotation at the method or interface level instead of creating mock() method and this annotation will create mock object of DataService and set this into its setter method so now in this case we don't need to set this mock object manually.  So we have Refactor the even more and reduced several line of the code just by using annotations. |

**Mockito Tips - Multiple Return Values and Specific Argument Matchers**

|  |
| --- |
| **package** com.junit.test.list.mocking;  **import** **static** org.junit.Assert.*assertEquals*;  **import** **static** org.mockito.Mockito.*mock*;  **import** **static** org.mockito.Mockito.*when*;  **import** java.util.List;  **import** org.junit.Test;  **public** **class** ListMocking {  @Test  **public** **void** list\_basic() {    List mockList = *mock*(List.**class**);  *when*(mockList.get(0)).thenReturn("Arun");  *assertEquals*("Arun",mockList.get(0));  }  @Test  **public** **void** mockMultiListValue() {    List mockMultiListvalue= *mock*(List.**class**);  *when*(mockMultiListvalue.size()).**thenReturn(5).thenReturn(10);**  *assertEquals*(5,mockMultiListvalue.size());  *assertEquals*(10,mockMultiListvalue.size());  }  @Test  **public** **void** mockMultiListStringValue() {  List mockMultiString = *mock*(List.**class**);  **for**(**int** i=0;i<=2;i++) {  *when*(mockMultiString.get(i)).**thenReturn("Arun").thenReturn("Kumar").thenReturn("Gupta");**  *assertEquals*("Arun",mockMultiString.get(i));  *assertEquals*("Kumar",mockMultiString.get(i));  *assertEquals*("Gupta",mockMultiString.get(i));  }  }  } |

**Mockito Tips - Argument Matchers**

|  |
| --- |
| // 15. Step 10 - Mockito Tips - Argument Matchers: How to make generic parameters:  **package** com.junit.test.using.any.arg.matcher;  **import** **static** org.mockito.Mockito.*when*;  **import** **static** org.junit.Assert.*assertEquals*;  **import** **static** org.mockito.Mockito.*anyInt*;  **import** java.util.List;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.mockito.Mock;  **import** org.mockito.junit.MockitoJUnitRunner;  @RunWith(MockitoJUnitRunner.**class**)  **public** **class** AnyArgumentMatcher {  @Mock  List anyListValueMock;    @Test  **public** **void** anyArugmentMatcher() {  *when*(anyListValueMock.get(*anyInt*())).thenReturn("Arun").thenReturn("Kumar").thenReturn("Gupta");  *assertEquals*("Arun",anyListValueMock.get(*anyInt*()));  *assertEquals*("Kumar",anyListValueMock.get(*anyInt*()));  *assertEquals*("Gupta",anyListValueMock.get(*anyInt*()));  }  }  // Here anyInt() is called argument matcher. Like anyInt() we have some other argument matcher like - anyString(), anyChar(), anyFloat(); |

**Mockito Tips - Verify method calls**

We already use this verify method. It does not return anything. It just verify the method call.

**- Mockito Tips - Argument Capture**

|  |
| --- |
| //17. Step 12 - Mockito Tips - Argument Capture: WE can check the passed value in the method call using ArgumentCaptor  **package** com.junit.test.verify.captor.argument;  **import** **static** org.junit.Assert.*assertEquals*;  **import** **static** org.mockito.Mockito.*times*;  **import** **static** org.mockito.Mockito.*verify*;  **import** java.util.List;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.mockito.ArgumentCaptor;  **import** org.mockito.Mock;  **import** org.mockito.junit.MockitoJUnitRunner;  @RunWith(MockitoJUnitRunner.**class**)  **public** **class** VerifyN\_Captor\_Argument {  @Mock  List listMock;    @Test  **public** **void** verifyAndCapture() {    listMock.add("CheckArgument");    // Verification  ArgumentCaptor<String> captor= ArgumentCaptor.*forClass*(String.**class**);  *verify*(listMock).add(captor.capture());  *assertEquals*("CheckArgument", captor.getValue());  }  **// Mockito Tips - Argument Capture on Multiple Calls**    @Test  **public** **void** verifyMultipleArgumentsAndCapture() {    listMock.add("CheckArgument1");  listMock.add("CheckArgument2");    // Verification  ArgumentCaptor<String> captor= ArgumentCaptor.*forClass*(String.**class**);  *verify*(listMock,*times*(2)).add(captor.capture());    *assertEquals*("CheckArgument1", captor.getAllValues().get(0));  *assertEquals*("CheckArgument2", captor.getAllValues().get(1));  }  } |

**Introduction to Spy**

|  |
| --- |
| **package** com.junit.test.mocking.vs.spy;  **import** **static** org.junit.Assert.*assertEquals*;  **import** **static** org.mockito.Mockito.*mock*;  **import** **static** org.mockito.Mockito.*spy*;  **import** **static** org.mockito.Mockito.*verify*;  **import** **static** org.mockito.Mockito.*when*;  **import** java.util.ArrayList;  **import** org.junit.Test;  **public** **class** MockVsSpy {  /\*  \*Until now we have been using mock but now let's use spy. Let's try to understand the different between them. In case of Mocking the entire behavior of the Interface or class gets lost. But in case of spy the entire behavior of interface or class persist i.e. A mock does not retain behavior of the original class because it is not real world class..We can mock the classes the same way we mock the interfaces  \*/  @Test  **public void moking()** {  ArrayList arrayListMock= *mock*(ArrayList.**class**);  arrayListMock.get(0); // will return Null: even-tough it does not have any value in the list so far. So instead of throwing null pointer exception it returns null the reason behind is that we are just mocking the ArrayList class and in case of mocking class does not retain its behavior  arrayListMock.size(); // will return 0  arrayListMock.add("One");  arrayListMock.add("two");  arrayListMock.size(); // As now we have added two element in the list so the size should be 2 but it will return 0 only because Mock does not retain behavior of the original class  *when*(arrayListMock.size()).thenReturn(5);// Now it will list size as 5 because here we are mocking the list size 5  *assertEquals*(5,arrayListMock.size());  }  @Test  **public void spying()** {  ArrayList arrayListSpy = *spy*(ArrayList.**class**);  //arrayListSpy.get(0); // Will get NPE because in spy class behavior does not change, so if from the emply list if we try to get value then will return NPE  arrayListSpy.size();  arrayListSpy.add("One");  arrayListSpy.add("Two");  arrayListSpy.size();  System.***out***.println(arrayListSpy.get(0)); // Now here we will not get any NPE because  list contains two elements  *when*(arrayListSpy.size()).thenReturn(10);  *assertEquals*(10, arrayListSpy.size());    // spy can be used where we want to use original dependencies just to findout what is happening with it and we don't want to mock the dependencies  // Let's say we dont have access to see the original code and we just want to verify the original bahavior then we can use spy.  *verify*(arrayListSpy).add("Two"); // like here we are just verifying the add method of ArrayList  }  } |

**Section Introduction - Unit Testing with Spring Boot and Mockito**

Here in the below example we have written a Rest Controller which directly returning the hard code response.

So here we will perform the unit testing of Controller class itself.

|  |
| --- |
| @RestController  **public** **class** HelloWorld {  @GetMapping("/hello-world")  **public** String helloWorld() {  **return** " Hi this is my frist Hello word class";  }  } |
| **Using Mock Mvc to test Hello World Controller**  **package** com.junit.test.spring.boot.helloworld.controller.test;  **import** **static** org.junit.Assert.*assertEquals*;  **import** **static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.*content*;  **import** **static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.*status*;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;  **import** org.springframework.http.MediaType;  **import** org.springframework.test.context.junit4.SpringRunner;  **import** org.springframework.test.web.servlet.MockMvc;  **import** org.springframework.test.web.servlet.MvcResult;  **import** org.springframework.test.web.servlet.RequestBuilder;  **import** org.springframework.test.web.servlet.request.MockMvcRequestBuilders;  **import** com.junit.test.spring.boot.helloworld.controller.HelloWorld;  /\* 1- To perform junit testing for the spring controller spring framework provides two things  - @WebMvcTest(Mention the Controller class name to be tested Here)  - Autowired the private MockMvc mockMvc  2- Since this is Spring Mvc Test so for that we have to mention @RunWith(SpringRunner.class) instead of JunitRunner.class  3- Now inside the @Test Method we want to make a call to /hello-world for performing unit test and want to verify the response Since this /hello-world an URI, not a java method so to call this URI we will use [private MockMvc mockMvc;]  \*/  @RunWith(SpringRunner.**class**)  @WebMvcTest(HelloWorld.**class**) // This will launch only HelloWorld Controller class  **public** **class** HelloWordControllerTest {  @Autowired  **private** MockMvc mockMvc;  @Test  **public** **void** helloWorldTest\_Basic() **throws** Exception {    // Generate RequestBuilder which will hit Get method and with URI- /hello-world who accept JSON  // Builds a MockHttpServletRequest.  RequestBuilder requestBuilder= MockMvcRequestBuilders.*get*("/hello-world") .accept(MediaType.***APPLICATION\_JSON***);  // pass this requestBuilder object which contains Http Method and request URI to MockMvc perform method to get response  MvcResult resultResponse = mockMvc.perform(requestBuilder).andReturn();  System.***out***.println(":::::"+resultResponse.getResponse().getContentAsString());    // Now finally verify the accepted content and actual content using the assertEquals  ***assertEquals*(" Hi this is my frist Hello word class", resultResponse.getResponse().getContentAsString());**  }  Step1- Annotate @RunWith(SpringRunner.**class**) to perform Spring framework testing  Step2- Annotate @WebMvcTest(HelloWorld.**class**) the controller class to which unit testing has to  perform  Step3- Declare [**private** MockMvc mockMvc;] to perform and process the MockHttpServletRequest built  by static method MockMvcRequestBuilders and return the MvcResult  Step4- Now gets the Http response from MvcResult and compare this actual result with expected  Result using assertEquals method of junit.  **// 26. Step 03 - Using Response Matchers to check status and content**  Here in this approach instead of using assertEquals to perform junit testing we will use Response matcher andExpect (). Here in the response we will perform and expect the Http status OK and compare the http content with the expected content    @Test  **public** **void** helloWorldTestUsingResponseMatcher() **throws** Exception {    // Generate RequestBuilder which will hit Get method and with URI- /hello-world who accept JSON  // Builds a MockHttpServletRequest.  RequestBuilder requestBuilder= MockMvcRequestBuilders.*get*("/hello-world")  .accept(MediaType.***APPLICATION\_JSON***);    // pass this requestBuilder object which contains Http Method and request URI to MockMvc perform method to get response  MvcResult resultResponse = mockMvc.perform(requestBuilder)  .andExpect(*status*().isOk())  .andExpect(*content*().string**(" Hi this is my frist Hello word class"))**  .andReturn();  System.***out***.println(":::::"+resultResponse.getResponse().getContentAsString());    // Here instead of using assertEquals we have used ResutlMatcher to test the httpStatus and response. Now finally verify the accepted content and actual content using the assertEquals  // **assertEquals(" Hi this is my frist Hello word class", resultResponse.getResponse().getContentAsString());**  **// Note: even though here we are not using assertEqual for verification. But content().string internally uses the assertEquals method as given below**  /\*\*  \* Assert the response body content as a String.    public ResultMatcher string(String expectedContent) {  **return result -> assertEquals("Response content", expectedContent**  **, result.getResponse().getContentAsString());**  } \*/  }  } |

**Creating a Basic REST Service in Item Controller**

As in the earlier example we have created one controller class and return the string and then perform the unit testing of the same.

Now let’s create Item Controller which will return the item object instead of string and let’s perform the unit testing of the same

|  |
| --- |
| **package** com.junit.test.spring.boot.helloworld.controller;  **public** **class** Item {  **private** **int** id;  **private** String name;  **private** **int** price;  **private** **int** quantity;  **public** Item(**int** id, String name, **int** price, **int** quantity) {  **this**.id=id;  **this**.name=name;  **this**.price=price;  **this**.quantity=quantity;  }  } |
| **package** com.junit.test.spring.boot.helloworld.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.GetMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  **public** **class** ItemController {    @GetMapping("/dummy-item")  **public** Item dummyItem() {  **return** **new** Item(1,"Ball", 10, 100);  }  } |
| **package** com.junit.test.spring.boot.helloworld.controller.test;  **import** **static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.*content*;  **import** **static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.*status*;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.skyscreamer.jsonassert.JSONAssert;  **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.http.MediaType;  **import** org.springframework.test.context.junit4.SpringRunner;  **import** org.springframework.test.web.servlet.MockMvc;  **import** org.springframework.test.web.servlet.MvcResult;  **import** org.springframework.test.web.servlet.RequestBuilder;  **import** org.springframework.test.web.servlet.request.MockMvcRequestBuilders;  **import** com.junit.test.spring.boot.helloworld.controller.ItemBusinessService;  **import** com.junit.test.spring.boot.helloworld.controller.ItemController;  @RunWith(SpringRunner.**class**)  @WebMvcTest(ItemController.**class**)  **public** **class** ItemControllerTest {    @Autowired  **private** MockMvc mockMvc;  **// Unit Testing Item Controller and Basic JSON Assertions**  @Test  **public** **void** itemTest\_Part\_Json() **throws** Exception {  RequestBuilder requestBuilder = MockMvcRequestBuilders.*get*("/dummy-item")  .accept(MediaType.***APPLICATION\_JSON***);  MvcResult andReturn = mockMvc.perform(requestBuilder)  .andExpect(*status*().isOk())  .andExpect(*content*().json("{ \"id\": 1, \"name\": \"Ball\"}"))  **// .andExpect(content().json("{ \"id\": 1, \"name\": \"Ball\", \"price\": 10, \"quantity\": 100 }"))**  .andReturn();  System.***out***.println(":::"+andReturn.getResponse().getContentAsString());    **JSONAssert.*assertEquals*("{ \"id\": 1, \"name\": \"Ball\"}", andReturn.getResponse().getContentAsString(), false);**    **//JSONAssert.assertEquals("{ \"id\": 1, \"name\": \"Ball\"}", andReturn.getResponse().getContentAsString(), true);**  // Here it will expect price and quantity element also then only it will suceed  }  @Test  **public** **void** itemTest\_Full\_Json() **throws** Exception {  RequestBuilder requestBuilder = MockMvcRequestBuilders.*get*("/dummy-item")  .accept(MediaType.***APPLICATION\_JSON***);  MvcResult andReturn = mockMvc.perform(requestBuilder)  .andExpect(*status*().isOk())  .andExpect(*content*().json("{ \"id\": 1, \"name\": \"Ball\", \"price\": 10, \"quantity\": 100 }")  //.andExpect(content().json("{ \"id\": 1, \"name\": \"Ball\"}"))  .andReturn();  System.***out***.println(":::"+andReturn.getResponse().getContentAsString());    **// JSONAssert.assertEquals("{ \"id\": 1, \"name\": \"Ball\"}", andReturn.getResponse().getContentAsString(), false);**    JSONAssert.*assertEquals*("{\"id\": 1, \"name\": \"Ball\", \"price\": 10, \"quantity\": 100 }",  andReturn.getResponse().getContentAsString(), **true**);  // Here it will expect price and quantity element also then only it will succeed  }  }  **// In case of Content ().string: it compares each and every string and if any mismatch it will fail even in case of space also.**  **// But in case of Content ().json it will compare any of the json and will succeed. Here it does not compare each and every json string.**  **// Internally content().json uses JSONAssert.assertEquals(jsonContent, content, strict) for verification**  **//Note:**  **/\***  **public ResultMatcher json(String jsonContent, boolean strict) {**  **return result -> {**  **String content = result.getResponse().getContentAsString(StandardCharsets.UTF\_8);**  **JSONAssert.assertEquals(jsonContent, content, strict);**  **};**  **}**  **\*/**  **// Here strict is a boolean which take false by default and because of this only JSONAssert.assertEquals checks only the Json elements including spaces**  **// But if we make it true then Unit test will pass only when the whole structure(content or element) is present. Like in the above example we are testing**  **// only few part of the Json and test succeed, because here JSONAssert.assertEquals internally uses strict => false.** |

**Digging deeper into JSON Assert**

|  |
| --- |
| **package** com.junit.test.spring.boot.helloworld.controller.test;  **import** org.json.JSONException;  **import** org.junit.Test;  **import** org.skyscreamer.jsonassert.JSONAssert;  // Digging deeper into JSON Assert  **public** **class** JsonAssertEqualTestCases {    String actual= "{ \"id\": 1, \"name\": \"Ball\", \"price\": 10, \"quantity\": 100 }";  @Test  **public** **void** jsonAssert() **throws** JSONException {    String expected="{ \"id\": 1, \"name\": \"Ball\", \"price\": 10, \"quantity\": 100 }";    //JSONAssert.assertEquals(expected, actual, strict);  JSONAssert.*assertEquals*(expected, actual, **true**);  // Note: Here the expected and actual both have complete element in the JSON and we test succeeed eventhough we have kept strict as true.  }    @Test  **public** **void** jsonAssert\_part() **throws** JSONException {    String expected="{ \"id\": 1, \"name\": \"Ball\"}";    //JSONAssert.assertEquals(expected, actual, strict);  //JSONAssert.assertEquals(expected, actual, true);    // Note: Here we have kept partial element of the complete JSON and we have kept strict as true, so now JSONAssert in this case will expects complete element of the Json, so in this case we will get AssertionError : and expected the remaining element like : price and quantity  }  @Test  **public** **void** jsonAssert\_witout\_escapeChar() **throws** JSONException {    //String expected="{ id: 1, name: Ball 2}"; // Test will be failed because here we have Ball 2 so we will have to use /" as given below.  String expected="{ \"id\": 1, \"name\": \"Ball 2\"}";  //JSONAssert.assertEquals(expected, actual, strict);  JSONAssert.*assertEquals*(expected, actual, **false**);    // Note: So here we can see the /" is not required inside the json string, without /" test suceed because here element name is written in one word.  // But if we have name lie Ball 2 then we have to put /" otherwise we will get test fails.  }  } |

**Writing a REST Service talking to Business Layer**

|  |  |
| --- | --- |
| **Item.java model class:**  **package**.helloworld.controller;  **public** **class** Item {  **private** **int** id;  **private** String name;  **private** **int** price;  **private** **int** quantity;  **public** Item(**int** id, String name, **int** price, **int** quantity) {  **this**.id=id;  **this**.name=name;  **this**.price=price;  **this**.quantity=quantity;  } | **RestService class: ItemBusinessService.java**  **package** helloworld.controller;  **import** org.springframework.stereotype.Component;  @Component  **public** **class** ItemBusinessService {  **public** Item retriveItems() {  **return** **new** Item(1, "Ball", 10, 100);  }  } |
| // Controller class retrieving data from service class:  @RestController  **public** **class** ItemController {  @Autowired  **private** ItemBusinessService businessService;  // Talking to business service for getting Items.  @GetMapping("/item-from-business-service")  **public** Item itemFromBusinessService() {  **return** businessService.retriveItems();  } | |
| **// Writing Unit Test for REST Service mocking Business Layer**  @RunWith(SpringRunner.**class**)  @WebMvcTest(ItemController.**class**)  **class** ItemFromServiceController {    @Autowired  **private** MockMvc mockMvc;    @MockBean  **private** ItemBusinessService service;    /\* Note: Since here the item are coming from the service layer so apart from mocking Itemcontroller.class we have to mock ItemBusinessService class and for that we have to use @MockBean at the class which needs to be mocked.Shows the junit testing for ItemController class, so we don't need to do unit testing for Service layer classes, and that is why here we are mocking the service layer class\*/  @Test  **public** **void** itemTestFromBusinessService() **throws** Exception {    *when*(service.retriveItems()).thenReturn(**new** Item(2, "Item2", 10, 1000));  /\* Here we are mocking and saying when service.retriveItems() is called then return Item class object, so basically this Item object is not getting from service.retriveItems(), instead of getting from here we are just returning the mock data of type [service.retriveItems()]  So here we are not testing the real world data instead we are just mocking the logic of service or data layer just by using mock data\*/    RequestBuilder requestBuilder = MockMvcRequestBuilders.*get*("/item-from-business-service")  .contentType(MediaType.***APPLICATION\_JSON***);  // Creating requestBuilder for get() method and for URI to be tested    MvcResult andReturn = mockMvc.perform(requestBuilder)  .andExpect(*status*().isOk())  .andExpect(*content*().json("{ \"id\": 2, \"name\": \"Item2\", \"price\": 10, \"quantity\": 1000 }"))  .andReturn();  }  } | |

**Overview of Steps 09 to 15:**

**In the previous sections we have created web layer (Controller) and talked to business layer (Service Layer). We have written the Junit test cases for Web layer and mocked the business layer.**

**Now in the further steps we will create Data layer and we will write unit test cases for each layer –web layer, business layer and data layer.**

**Step 09 - 01 - Prepare Data Layers with JPA,Hibernate and H2**

Let’s add dependency for spring boot data jba and in-memory database h2

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency> |
| <dependency>  <groupId>com.h2database</groupId>  <artifactId>h2</artifactId>  </dependency> |

**Step 10 - Create Item Entity and Populate data with data.sql**

1. Let’s make the ItemEntity.java class as entity class

|  |
| --- |
| **package** com.junit.test.web.service.data.layer;  **import** javax.persistence.Entity;  **import** javax.persistence.Id;  **import** javax.persistence.Transient;  @Entity  **public** **class** ItemEntity {  @Id  **private** **int** id;  **private** String name;  **private** **int** price;  **private** **int** quantity;    @Transient  **private** **int** value;  **public** ItemEntity(**int** id, String name, **int** price, **int** quantity) {  **this**.id=id;  **this**.name=name;  **this**.price=price;  **this**.quantity=quantity;  }  } |
| **application.properties:**  spring.datasource.url=jdbc:h2:mem:testdb  spring.data.jpa.repositories.bootstrap-mode=default  spring.jpa.show-sql=true  spring.h2.console.enabled=true |
| Hibernate: create table item\_entity (id integer not null, name varchar(255), price integer not null, quantity integer not null, primary key (id)) |
|  |

**Insert into Item\_Entity (id, name, price, quantity) values (1001, 'item1', 10, 20);**

**Insert into Item\_Entity (id, name, price, quantity) values (1002, 'Item2', 20, 30);**

**Insert into Item\_Entity (id, name, price, quantity) values (1003, 'Item3', 30, 40);**

So have created **ItemEntity** class and in-memory h2 database and we are able to insert data into database. In the next sprint let’s create [Controller, Service and Database layer] and fetch the data from the database.

**Step 11 - Create a RESTful Service talking to the database**

**Step 12 - Writing Unit Test for Web Layer -Controller - Using Mock MVC**

**13 - Exercise & Solution - Writing Unit Test for Business Layer - Mocking**

**Step 14 - Writing Unit Test for Data Layer - Data JPA Test**

JUnit 5 Alert

**This course works with JUNIT 4 and JUNIT 5**

**BOOKMARK OUR GITHUB REPO** : <https://github.com/in28minutes/spring-unit-testing-with-junit-and-mockito>

(If you are using JUnit 5) In the next lecture, You do NOT need to add

@RunWith(SpringRunner.class) on top of

@DataJpaTest

JUnit 4 Code

@RunWith(SpringRunner.class)

@DataJpaTest

JUnit 5 Code

@DataJpaTest

JUnit 5 Commit Changes:

<https://github.com/in28minutes/spring-unit-testing-with-junit-and-mockito/commit/2c6df429aec248d455b7ad19e60e90743979ba1c>

JUnit 5 Alert

**This course works with JUNIT 4 and JUNIT 5**

**BOOKMARK OUR GITHUB REPO** : <https://github.com/in28minutes/spring-unit-testing-with-junit-and-mockito>

(If you are using JUnit 5) In the next lecture, You do NOT need to add

@RunWith(SpringRunner.class) on top of

@SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT)

JUnit 4 Code

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT)

JUnit 5 Code

@SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT)

JUnit 5 Commit Changes: <https://github.com/in28minutes/spring-unit-testing-with-junit-and-mockito/commit/2c6df429aec248d455b7ad19e60e90743979ba1c>