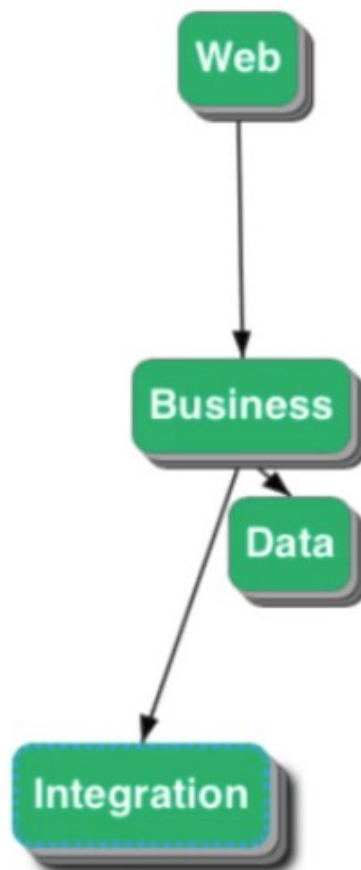


Mocking

When we are going to develop the application it is in Layer like web layer, Business layer, data layer and integration layer, whenever we are wrting unit test for web layer then no need worry about business layer just it mockit out.

whenever we are writing the unit test for business layer then no need worry about business layer just mockit out.



Mocking enable as to write great unit test case.

Why we need Mocking ?

Simple Testing Results.

Business Loginc class.

```
public class SomeBussinessImpl {  
  
    int sum=0;  
    public int sumCalculate(List<Integer> list) {  
  
        for(Integer value :list) {  
            sum+=value;  
        }  
        return sum;  
    }  
}
```

Testing the SomeBusinessImpl class with different scenario

```
public class SomeBusinessTest {  
  
    @Test  
    public void CalculateSomeTest() {  
        SomeBussinessImpl business = new SomeBussinessImpl();  
        int actualResult=business.sumCalculate(Arrays.asList(new Integer[]  
{1,2,3,4,5,6}));  
        int expectedResult=21;  
        assertEquals(expectedResult, actualResult);  
    }  
  
    @Test  
    public void CalculateEmptyValueTest() {  
        SomeBussinessImpl business = new SomeBussinessImpl();  
        int actualResult=business.sumCalculate(Arrays.asList(new Integer[]  
{}));  
        int expectedResult=0;  
        assertEquals(expectedResult, actualResult);  
    }  
  
    @Test  
    public void CalculateOneValueTest() {  
        SomeBussinessImpl business = new SomeBussinessImpl();  
        int actualResult=business.sumCalculate(Arrays.asList(new Integer[]  
{5}));  
        int expectedResult=5;  
        assertEquals(expectedResult, actualResult);  
    }  
}
```

Why we need Stub?

Stub means create dependency class object suppose we have business layer and business layer is dependant on Data layer but we don't want to test the data layer so how to solve the problem, So we need to create a stub class and hardcode the value of data layer and just passed to business layer means just we are creating dummy object.

Business Layer

```
public class SomeBusinessImpl {
    private SomeDataService someDataService;

    public void setSomeDataService(SomeDataService someDataService) {
        this.someDataService = someDataService;
    }
    public int calculateFromDataService() {
        List<Integer> list = someDataService.retrieveAllData();
        for (Integer value : list) {
            sum += value;
        }
        return sum;
    }
}
```

Data Layer

```
public interface SomeDataService {

    public List<Integer> retrieveAllData();
}
```

Test class create stub(create the object and passed the hardcoded values)

```
class SomeDataStub implements SomeDataService{

    @Override
    public List<Integer> retrieveAllData() {
        return Arrays.asList(new Integer[] {1,2,3,4});
    }
}
```

Testing class logic

```
public class SomeBusinessTest {

    @Test
    public void CalculateUsingDataService() {
        SomeBusinessImpl business = new SomeBusinessImpl();
        // database dependency which doesn't need to test just create stub
        business.setSomeDataService(new SomeDataStub());
        int actualResult=business.calculateFromDataService();
        int expectedResult=10;
        assertEquals(expectedResult, actualResult);
    }
}
```

What is Problems with Stub

Suppose we want to test the API with different scenarios then we need to create multiple stub or change the value of Stub class every scenarios,

Example

TestCase

- 1) first test the with actualValue 10 and expected value 10
- 2) actual value 20 and expected value 20
- 3) Empty value check
- 4) one value checked

In above test case need to created 4 stub or we need to change stub value Every time to test the API with different test case.

Like example

StubFirst for TestCase1

```
class SomeDataStub implements SomeDataService{

    @Override
    public List<Integer> retrieveAllData() {
        return Arrays.asList(new Integer[] {1,2,3,4});
    }

}
```

Stub2 for TestCase 2 (Test with multiple value)

```
class SomeDataStub2 implements SomeDataService{

    @Override
    public List<Integer> retrieveAllData() {
        return Arrays.asList(new Integer[] {1,2,3,4,10});
    }

}
```

Stub3 for TestCase 3 (Test with Empty object)

```
class SomeDataStub3 implements SomeDataService{

    @Override
    public List<Integer> retrieveAllData() {
        return Arrays.asList(new Integer[] {});
    }

}
```

Stub4 for TestCase 4 (Test with single value)

```
class SomeDataStub4 implements SomeDataService{

    @Override
    public List<Integer> retrieveAllData() {
        return Arrays.asList(new Integer[] {5});
    }

}
```

Test Class with different TestCase

```
public class SomeBusinessTest {

    @Test
    public void CalculateUsingDataService() {
        SomeBussinessImpl business = new SomeBussinessImpl();
        business.setSomeDataService(new SomeDataStub());
        int actualResult=business.calculateFromDataService();
        int expectedResult=10;
        assertEquals(expectedResult, actualResult);
    }

    @Test
    public void CalculateDiffValueTest() {
        SomeBussinessImpl business = new SomeBussinessImpl();
        business.setSomeDataService(new SomeDataStub2());
        int actualResult=business.calculateFromDataService();
        int expectedResult=20;
        assertEquals(expectedResult, actualResult);
    }

    @Test
    public void CalculateEmptyValueTest() {
        SomeBussinessImpl business = new SomeBussinessImpl();
        business.setSomeDataService(new SomeDataStub3());
        int actualResult=business.calculateFromDataService();
        int expectedResult=0;
        assertEquals(expectedResult, actualResult);
    }

    @Test
    public void CalculateOneValueTest() {
        SomeBussinessImpl business = new SomeBussinessImpl();
        business.setSomeDataService(new SomeDataStub4());
        int actualResult=business.calculateFromDataService();
        int expectedResult=5;
        assertEquals(expectedResult, actualResult);
    }
}
```

So avoid the above stub problem we can go for Mock

Instead of creating stub we can use mock object or fake object to test the API, Mock means create a fake object or fake dependent object,

Step to create the mock objects

1) create the mock/fake object calling the static mock() method which is return the mock object

Exa: **SomeDataService SomeDataServiceMock=mock(SomeDataService.class);**

2) we can use mockito when method to Register the specific mock method and return the expected results.(return mock object/fake object)

Exa: **when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new Integer[] {1,2,3,4}));**

3) pass the fake object to the business layer

Exa: **business.setSomeDataService(SomeDataServiceMock);**

```
public class SomeBusinessTest {

    @Test
    public void CalculateUsingDataService() {
        SomeBussinessImpl business = new SomeBussinessImpl();
        SomeDataService SomeDataServiceMock=mock(SomeDataService.class);
        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new
        Integer[] {1,2,3,4}));
        business.setSomeDataService(SomeDataServiceMock);
        int actualResult=business.calculateFromDataService();
        int expectedResult=10;
        assertEquals(expectedResult, actualResult);
    }
    @Test
    public void CalculateDiffValueTest() {
        SomeBussinessImpl business = new SomeBussinessImpl();
        SomeDataService SomeDataServiceMock=mock(SomeDataService.class);
        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new
        Integer[] {1,2,3,4,10}));
        business.setSomeDataService(SomeDataServiceMock);
        int actualResult=business.calculateFromDataService();
        int expectedResult=20;
        assertEquals(expectedResult, actualResult);
    }
    @Test
    public void CalculateEmptyValueTest() {
        SomeBussinessImpl business = new SomeBussinessImpl();
        SomeDataService SomeDataServiceMock=mock(SomeDataService.class);
        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new
        Integer[] {}));
        business.setSomeDataService(SomeDataServiceMock);
        int actualResult=business.calculateFromDataService();
        int expectedResult=0;
        assertEquals(expectedResult, actualResult);
    }
    @Test
    public void CalculateOneValueTest() {
        SomeBussinessImpl business = new SomeBussinessImpl();
        SomeDataService SomeDataServiceMock=mock(SomeDataService.class);
        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new
        Integer[] {5}));
        business.setSomeDataService(SomeDataServiceMock);
        int actualResult=business.calculateFromDataService();
        int expectedResult=5;
        assertEquals(expectedResult, actualResult);
    }
}
```

To avoid the boilerplate/redundency code to create the mock objects

```
public class SomeBusinessTest {
    SomeBussinessImpl business = new SomeBussinessImpl();
    SomeDataService SomeDataServiceMock=mock(SomeDataService.class);

    @Before
    public void before() {
        business.setSomeDataService(SomeDataServiceMock);
    }

    @Test
    public void CalculateUsingDataService() {

        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new
        Integer[] {1,2,3,4}));
        int actualResult=business.calculateFromDataService();
        int expectedResult=10;
        assertEquals(expectedResult, actualResult);
    }

    @Test
    public void CalculateDiffValueTest() {

        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new
        Integer[] {1,2,3,4,10}));
        int actualResult=business.calculateFromDataService();
        int expectedResult=20;
        assertEquals(expectedResult, actualResult);
    }

    @Test
    public void CalculateEmptyValueTest() {

        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new
        Integer[] {}));
        int actualResult=business.calculateFromDataService();
        int expectedResult=0;
        assertEquals(expectedResult, actualResult);
    }

    @Test
    public void CalculateOneValueTest() {

        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new
        Integer[] {5}));
        int actualResult=business.calculateFromDataService();
        int expectedResult=5;
        assertEquals(expectedResult, actualResult);
    }
}
```

Again we need to make coding standard

@Mock annotation basic used for create the moc/fake object

@InjectMocks annotation are used to inject the mock/depended obj to target class.

```
@RunWith(MockitoJUnitRunner.class)
```

```
public class SomeBusinessTest {
```

```
    @InjectMocks
```

```
    SomeBussinessImpl business;
```

```
    @Mock
```

```
    SomeDataService SomeDataServiceMock;
```

```
    @Test
```

```
    public void CalculateUsingDataService() {
```

```
        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new  
        Integer[] {1,2,3,4}));
```

```
        assertEquals(10, business.calculateFromDataService());
```

```
    }
```

```
    @Test
```

```
    public void CalculateDiffValueTest() {
```

```
        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new  
        Integer[] {1,2,3,4,10}));
```

```
        assertEquals(20, business.calculateFromDataService());
```

```
    }
```

```
    @Test
```

```
    public void CalculateEmptyValueTest() {
```

```
        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new  
        Integer[] {}));
```

```
        assertEquals(0, business.calculateFromDataService());
```

```
    }
```

```
    @Test
```

```
    public void CalculateOneValueTest() {
```

```
        when(SomeDataServiceMock.retriveAllData()).thenReturn(Arrays.asList(new  
        Integer[] {5}));
```

```
        assertEquals(5, business.calculateFromDataService());
```

```
    }
```

```
}
```


Mocking with java.util.list interface mock/fake object

```
public class ListMockTest {

    List mock = Mockito.mock(List.class);

    @Test
    public void testSize() {
        when(mock.size()).thenReturn(5);
        assertEquals(5, mock.size());
    }

    @Test
    public void returnDiffValue() {

        when(mock.size()).thenReturn(5).thenReturn(10);
        assertEquals(5, mock.size());
        assertEquals(10, mock.size());
    }

    @Test
    public void testWithParameters() {
        when(mock.get(0)).thenReturn("index 0 hi");
        assertEquals("index 0 hi", mock.get(0));
        assertEquals(null, mock.get(1));
    }

    @Test
    public void returnWithGenericParamets() {
        //Mockito.anyInt() Arguments matchers
        when(mock.get(Mockito.anyInt())).thenReturn("specific value");
        assertEquals("specific value", mock.get(0));
        assertEquals("specific value", mock.get(1));
    }

    @Test
    public void verifyingMethod() {
        //STUB
        String value1= mock.get(0);
        String value2= mock.get(1);
        //Verify
        verify(mock).get(0);
        verify(mock,times(2)).get(Mockito.anyInt());
        verify(mock,atLeastOnce()).get(Mockito.anyInt());
        verify(mock,atMost(2)).get(Mockito.anyInt());
        verify(mock,never()).get(2);
    }

}
```

Mock verify the method calls.

How you can verify the specific method is called in Mockito

Example

```
public int calculateFromDataService() {
    List<Integer> list = someDataService.retrieveAllData();
    for (Integer value : list) {
        sum += value;
    }
    //someDataService.storeSumValue(sum)
    return sum;
}
```

In these example how can verify the storeSumValue() is executed or not

```
@Test
public void verifyingMethod() {
    //STUB
    String value1= mock.get(0);
    String value2= mock.get(1);

    //Verify
    verify(mock).get(0);
    verify(mock,times(2)).get(Mockito.anyInt());
    verify(mock,atLeastOnce()).get(Mockito.anyInt());
    verify(mock,atMost(2)).get(Mockito.anyInt());
    verify(mock,never()).get(2);
}
```

Mocks verify Method arguments

argument capture are used to verify the what data we are passing in method arguments in these example

mock.add("first args") add method we are passing first args.

```
@Test
public void argumentCapture() {
    mock.add("first args");
    ArgumentCaptor<String> captor=ArgumentCaptor.forClass(String.class);
    verify(mock).add(captor.capture());
    assertEquals("first args", captor.getValue());
    //capture the value which are passed in arguments
}
```

Multiple arguments verify from methods calls

```
@Test
public void multipleArgumentCapture() {
    mock.add("first args");
    mock.add("second args");

    ArgumentCaptor<String> captor= ArgumentCaptor.forClass(String.class);
    verify(mock,times(2)).add(captor.capture());
    assertEquals("first args", captor.getAllValues().get(0));
    assertEquals("second args",captor.getAllValues().get(1));
}
```

Spy(spying)

Spy is used for original behaviour return. Means it treat as original not mock/fake object .

Example

- 1) create ArrayList `spy = Mockito.spy(ArrayList.class);`
- 2)it return original behaviour
- 3)if we call `get(1)` it will throws `ArrayIndexOutOfBoundsException` exception because array list doesn't have elements.but not in mock, mock return default behaviour(`null`).
`System.out.println("--first time get---"+mocks.get(0));`
- 4)when can override the original behaviour of spy like
`when(spy.size()).thenReturn(5);` // now it will return 5

```
@Test
public void spying() {
    ArrayList spy = Mockito.spy(ArrayList.class);
    //System.out.println("--first time get---" + spy.get(0));
    //System.out.println("--first time size---" + spy.size());
    spy.add("Index 0");
    spy.add("Index 1");
    System.out.println("--second time get---" + spy.get(0));
    System.out.println("--second time size---" + spy.size());
    when(spy.size()).thenReturn(5);
    System.out.println("--third time get---" + spy.get(0));
    System.out.println("--third time size---" + spy.size());
    verify(spy).add("Index 1");
}
```

-----Unit testing with Spring boot and mockito -----

Controller layer test--> so how we can test only controller not other layer ?

So we using `@RunWith(SpringRunner.class)` and `@WebMvcTest` annotation that is basically used to test specific Controller

and passed the controller class name which controller you are going to test

example

```
@RestController
public class HelloWorldController {
    @GetMapping("/hello-world")
    public String getHelloWorld() {

        return "Hello world";
    }
}
```

Test class

RequestBuilder basically used to http request and `mockMvc.perform()` used for Http request(GET,POST,DELETE etc) the result `MvcResult` have the response we can extract the results also.

```
@RunWith(SpringRunner.class)
@WebMvcTest(HelloWorldController.class)
public class HelloControllerTest {
    //Spring bean
    @Autowired
    MockMvc mockMvc;

    @Test
    public void helloWorldBasicTest() throws Exception {
        RequestBuilder reqBuilder = MockMvcRequestBuilders.get("/hello-world").accept(MediaType.APPLICATION_JSON);
        MvcResult result = mockMvc.perform(reqBuilder).andReturn();
        assertEquals("Hello world", result.getResponse().getContentAsString());
    }
}
```

MockMvc have some method like `andExpect()` which is used for check status code, contenttype, data match with json etc.

```
@Test
public void helloWorldBasicTest() throws Exception {
    RequestBuilder reqBuilder = MockMvcRequestBuilders.get("/hello-world").accept(MediaType.APPLICATION_JSON);
    MvcResult result = mockMvc.perform(reqBuilder)
        .andExpect(status().isOk())
        .andExpect(content().string("Hello world"))
        .andReturn();

}
```

Get method testing with Real world example

Bean class

```
public class Item {  
  
    private int id;  
    private String name;  
    private int price;  
    private int quantity;  
  
    public Item() {}  
  
    public Item(int id, String name, int price, int quantity) {  
        super();  
        this.id = id;  
        this.name = name;  
        this.price = price;  
        this.quantity = quantity;  
    }  
}
```

Controller class

```
@RestController  
public class ItemController {  
  
    @GetMapping("/get-items")  
    public Item getItem() {  
  
        return new Item(1, "omega", 10, 120);  
    }  
}
```

Test Class

JUnit test compare with json response

```
@RunWith(SpringRunner.class)  
@WebMvcTest(ItemController.class)  
public class ItemControllerTest {  
  
    @Autowired  
    MockMvc mockMvc;  
  
    @Test  
    public void getItem() throws Exception {  
        RequestBuilder reqBuilder = MockMvcRequestBuilders.get("/get-items").accept(MediaType.APPLICATION_JSON);  
  
        mockMvc.perform(reqBuilder).andExpect(status().isOk()).andExpect(content().json("{\"id\":1,\"name\":\"omega\",\"price\":10,\"quantity\":120}"));  
    }  
}
```

```

@Test
public void getItemSimple() throws Exception {

    RequestBuilder reqBuilder = MockMvcRequestBuilders.get("/get-
items").accept(MediaType.APPLICATION_JSON);
    MvcResult result =
mockMvc.perform(reqBuilder).andExpect(status().isOk()).andReturn();

    assertEquals("{\"id\":1,\"name\":\"omega\",\"price\":10,\"quantity\":120}",
result.getResponse().getContentAsString());
}

}

```

JSONAssert.assertEquals()

JSONAssert is very important to debug and identify the error in json data and also simplify compare the the objects.

Advantage

- 1) Simplify identify the error in json data
- 2) not need to specify the escap character to create json object
- 3) we can use strict compare also

Example:

```

public class JsonAssertTest {
    String
actualResult="{\"id\":1,\"name\":\"omega\",\"price\":10,\"quantity\":120}";
    @Test
    public void testAssert() throws JSONException {
        String
expected="{\"id\":1,\"name\":\"omega\",\"price\":10,\"quantity\":120}";
        JSONAssert.assertEquals(expected, actualResult,true);
        //removed some content form expected json
        String expectedCh="{\"id\":1,\"name\":\"omega\"}";
        JSONAssert.assertEquals(expectedCh, actualResult,false);
    }
    @Test
    public void testAssertStrictTrue() throws JSONException {
        //removed some content form expected json
        String expectedCh="{\"id\":1,\"name\":\"omega\"}";
        // strict true means actual expected result must be same
        JSONAssert.assertEquals(expectedCh, actualResult,true);
    }

    @Test
    public void testAssertWithOutEscapChar() throws JSONException {
        String actualResult="{id:1,name:omega,price:10,quantity:120}";
        String expectedCh="{id:1,name:omega}";
        JSONAssert.assertEquals(expectedCh, actualResult,false);
    }
}

```

Now Example for Mock real Time example if your testing the controller then we can not depends on BusinessService such scenarios Mocking will be come on picture.

So let mock the business service and test the ItemController

Note- when we are going to test Controller then no need to check bussiness logic of business service just mock or just create fake object and return then to avoid dependency of other layer.

Example

Bean class

```
public class Item {

    private int id;
    private String name;
    private int price;
    private int quantity;

    public Item() {}

    public Item(int id, String name, int price, int quantity) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }
}
```

Service class

```
public interface ItemService {

    public Item getItem();
}
```

ServiceImpl Class

```
@Service
public class ItemServiceImpl implements ItemService{

    @Override
    public Item getItem() {
        return new Item(1,"omega",10,120);
    }
}
```

Controller class

```
@RestController
public class ItemController {

    @Autowired
    ItemService itemSerive;
    @GetMapping("/item-from-bussiness-service")
    public Item getItemFromService() {
        return itemSerive.getItem();
    }
}
```

TestClass

MockBean create the mock object of dependent class

```
@RunWith(SpringRunner.class)
@WebMvcTest(ItemController.class)
public class ItemControllerTest {

    @Autowired
    MockMvc mockMvc;

    @MockBean
    ItemServiceImpl itemServiceImpl;

    @Test
    public void testItemFromBusinessService() throws Exception {
        when(itemServiceImpl.getItem()).thenReturn(new
            Item(1, "omega", 10, 120));

        RequestBuilder reqBuilder = MockMvcRequestBuilders.get("/item-from-
            bussiness-service").accept(MediaType.APPLICATION_JSON);
        MvcResult result =
            mockMvc.perform(reqBuilder).andExpect(status().isOk()).andReturn();
        JSONAssert.assertEquals("{id:1,name:omega,price:10,quantity:120}",
            result.getResponse().getContentAsString(), false);
    }
}
```

TestCase for Controller

Entity

```
@Entity
public class Item {

    @Id
    private int id;
    private String name;
    private int price;
    private int quantity;

    @Transient
    int value;

    public Item() {}

    public Item(int id, String name, int price, int quantity) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }
}
```


Service

```
public interface ItemBeanService {  
    public ItemBean getItem();  
}
```

ServiceImpl

```
@Service  
public class ItemServiceImpl implements ItemService{  
    @Autowired  
    ItemRepository itemRepository;  
    @Override  
    public List<Item> getAllItems() {  
        List<Item> items = itemRepository.findAll();  
        for(Item item:items) {  
            System.out.println("--call service--"+item.getName()+"  
"+item.getPrice()*item.getQuantity());  
            item.setValue(item.getPrice()*item.getQuantity());  
        }  
        return items;  
    }  
}
```

Repository

```
public interface ItemRepository extends JpaRepository<Item,Integer>{  
  
}
```

Controller

```
@RestController  
@RequestMapping("/api")  
public class ItemController {  
    @Autowired  
    ItemService itemService;  
    @GetMapping("/get-items")  
    public List<Item> getItems() {  
        return itemService.getAllItems();  
    }  
}
```

TestController

for testing the MVC controller need following annotation on top of TestClass

```
@RunWith(SpringRunner.class)
```

```
@WebMvcTest(ItemController.class) //name the controller class which you want to  
test it load only that class not that depended class like service, repository  
etc.
```

Here we have to create mock object of bussiness layer we don't want to test the bussiness layer just want to test Controller layer

So create mockMvc object using **@MockMvc** anotation, with help of mockMvc we can call http request etc.

@MockBean used to create the mock object.

```
@Autowired  
MockMvc mockMvc;  
@MockBean  
ItemServiceImpl itemServiceImpl;
```

Example

```
@RunWith(SpringRunner.class)  
@WebMvcTest(ItemController.class)  
public class ItemControllerTest {  
    @Autowired  
    MockMvc mockMvc;  
    @MockBean  
    ItemServiceImpl itemServiceImpl;  
  
    @Test  
    public void getItemFromDB() throws Exception {  
        List<Item> items = new ArrayList<>();  
        items.add(new Item(1, "omega", 10, 120));  
        when(itemServiceImpl.getAllItems()).thenReturn(items);  
  
        RequestBuilder reqBuilder = MockMvcRequestBuilders.get("/api/get-  
items").accept(MediaType.APPLICATION_JSON);  
        MvcResult result =  
mockMvc.perform(reqBuilder).andExpect(status().isOk()).andReturn();  
        JSONAssert.assertEquals("[{id:1,name:omega,price:10,quantity:120}]",  
result.getResponse().getContentAsString(), false);  
    }  
}
```

Test the Business Layer

for testing the bussiness layer required following annotation

`@RunWith(MockitoJUnitRunner.class)`

`@InjectMocks` // inject the depened object to target class

`@Mock` // create mock object of depended class

In below example we don't worry about data layer just create mock object of data layer and performs the business operation without call data layer.

```
@RunWith(MockitoJUnitRunner.class)
public class ItemServiceTest {

    @InjectMocks
    ItemServiceImpl itemServiceImpl;

    @Mock
    ItemRepository itemRepository;

    @Test
    public void getItemsFromBusiness() {
        List<Item> items = new ArrayList<>();
        items.add(new Item(1,"item1",10,120));
        items.add(new Item(2,"item2",10,60));
        when(itemRepository.findAll()).thenReturn(items);

        List<Item>testItems = itemServiceImpl.getAllItems();

        System.out.println("---first Index
value--"+testItems.get(0).getValue());
        assertEquals(1200, testItems.get(0).getValue());
        System.out.println("---second Index
value--"+testItems.get(1).getValue());
        assertEquals(600, testItems.get(1).getValue());
    }
}
```

Test the data/persistent Layer.

For testing repository we can use following anotation

@DataJpaTest to test the spring repository it test only inmemory database

@RunWith(SpringRunner.class)

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class ItemRepositoryTest {

    @Autowired
    ItemRepository itemRepository;

    @Test
    public void saveItem() {
        List<Item> items = new ArrayList<>();
        items.add(new Item(1,"item1",10,120));
        items.add(new Item(2,"item2",10,60));
        items.add(new Item(4,"item4",30,30));
        itemRepository.saveAll(items);
    }

    @Test
    public void findAll() {

        List<Item> items=itemRepository.findAll();
        items.forEach(item->{
            System.out.println(" id "+item.getId()+" name "+item.getName()
+" price "+item.getPrice());

        });
    }

}
```

Integration Test(Controller,Service,Data Layer)

@SpringBootTest annotation is used to execute the whole application not specific class, because @SpringBootTest search the @SpringBootApplication class and load the spring context and execute that.

@SpringBootTest(webEnvironment=WebEnvironment.**RANDOM_PORT**) Spring used random port number to executed the API

we can check all layer communication is working fine.

We can not test real database or interface because if test with real database and some one change the database then test result is failed for testing use inmemory database.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class ItemControllerIntegrationTest {

    @Autowired
    TestRestTemplate restTemplate;

    @Test
    public void integrationTest() throws JSONException {
        String response= restTemplate.getForObject("/get-items",
String.class);
        System.out.println(response);
        JSONAssert.assertEquals("{id:1,name:omega,price:10,quantity:120}",
response, false);
    }

}
```

Note: Also we can use mock for any Layer to create the fake object for example we don't want to test the real database record so then created the mock object to annotated with @MockBean annotation

How to create different properties for test db or in memory db configuration?

/src/test/ create new folder **resources** in these folder create properties file to configure in memory database.

Example:

/src/test/resources have application.properties

Here we can override application.properties value in test properties
database name password may be different for both properties

`@TestPropertySource(locations="{classpath:test-configurariion.propeties}")`
these annotation is used for if you have multiple properties file and we want to use it in specific properties in specific class

Example:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)

public class ItemControllerIntegrationTest {

    @Autowired
    TestRestTemplate restTemplate;

    @MockBean
    ItemRepository repository;

    @Test
    public void integrationTest() throws JSONException {
        String response= restTemplate.getForObject("/get-items",
String.class);
        System.out.println(response);
        JSONAssert.assertEquals("{id:1,name:omega,price:10,quantity:120}",
response, false);
    }
}
```

-----2nd exa. Integration Testing-----

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
@AutoConfigureMockMvc
public class IntegrationTesting {

    @Autowired
    MockMvc mockMvc;

    @MockBean
    ItemRepository repository;

    ObjectMapper objectMapper = new ObjectMapper();

    @Test
    public void testPostMethod() throws JsonProcessingException, Exception {
        Item item = new Item(110, "test", 20, 20);
        String inputJson = objectMapper.writeValueAsString(item);
        RequestBuilder reqBuilder =
        MockMvcRequestBuilders.post("/api/save-item").

        contentType(MediaType.APPLICATION_JSON).content(inputJson);
        MvcResult result = mockMvc.perform(reqBuilder)
            .andExpect(status().isOk())
            .andReturn();

    }

    @Test
    public void getMethodTest() throws Exception {
        RequestBuilder reqBuilder = MockMvcRequestBuilders.get("/get-items").
            contentType(MediaType.APPLICATION_JSON);
        MvcResult result = mockMvc.perform(reqBuilder)
            .andExpect(status().isOk())
            .andReturn();
        String resp=result.getResponse().getContentAsString();
        Item item = objectMapper.readValue(resp, Item.class);
    }
}
```

How to test RestApi for different type http method like

- 1)GET
- 2)POST
- 3)DELETE
- 4)PUT

Example Test the application with Post methods

```
@RunWith(SpringRunner.class)
@WebMvcTest(ItemController.class)
public class TestPostMethod {
    @Autowired
    MockMvc mvc;
    @MockBean
    ItemService itemService;

    @Test
    public void testPostMethod() throws JsonProcessingException, Exception {

        Item item = new Item(110, "test", 20, 20);

        when(itemService.saveItem(item)).thenReturn(true);

        String inputJson = mapToJson(item);
        RequestBuilder reqBuilder =
MockMvcRequestBuilders.post("/api/save-item").
contentType(MediaType.APPLICATION_JSON).content(inputJson);
        MvcResult result = mvc.perform(reqBuilder)
            .andExpect(status().isOk())
            .andReturn();
        System.out.println(result.getResponse().getContentAsString());

    }
    //Convert Object to Json
    protected String mapToJson(Object obj) throws JsonProcessingException {
        ObjectMapper objectMapper = new ObjectMapper();
        return objectMapper.writeValueAsString(obj);
    }
}
```


Assert Static method to check the validation of data.

```
//Checks if an object isn't null.
assertNotNull(object);
//Checks if two primitive types or objects are equal.
assertEquals(object1,object2);
//Checks if input condition is true.
assertTrue(boolean);
//Checks if input condition is false.
assertFalse(boolean);
//Checks if an object is null.
assertNull(null);
//Checks if two object references do not point to the same
object in memory.
assertNotSame(object1,object2);
```

```
@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
public class UserControllerTest {

    @Autowired
    MockMvc mockMvc;

    @MockBean
    UserServiceImpl userService;

    private MvcResult andReturn;

    ObjectMapper om = new ObjectMapper();

    @Test
    public void testGetAllUser() throws Exception {
        List<UserBean> uBeans = getUsersData();
        when(userService.getAllUser()).thenReturn(uBeans);
        RequestBuilder builder = MockMvcRequestBuilders.get("/api/user-
list").accept(MediaType.APPLICATION_JSON);
        andReturn =
mockMvc.perform(builder).andExpect(status().isOk()).andReturn();
        String result = andReturn.getResponse().getContentAsString();
        ResponseBean response = om.readValue(result, ResponseBean.class);
        String usreString = om.writeValueAsString(response.payload);
        List<obj> objs = om.readValue(usreString, List.class);
        Iterator iterator = objs.iterator();
        while (iterator.hasNext()) {
            UserBean userBean = new DozerBeanMapper().map(iterator.next(),
UserBean.class);

            //Checks if an object isn't null.
            assertNotNull(userBean.getId());
            //Checks if two primitive types or objects are equal.
            assertEquals(userBean.getId(),userBean.getId());
            //Checks if input condition is true.
            assertTrue(Boolean.TRUE);
            //Checks if input condition is false.
            assertFalse(Boolean.FALSE);
            //Checks if an object is null.
            assertNull(null);
            //Checks if two object references do not point to the same
object in memory.
            assertNotSame(userBean.getAge(),userBean.getId());

        }
    }
}
```

Some annotation Descriptions

@RunWith(SpringRunner.class)

Spring Runner is used to launch up spring context in unit tests.

@SpringBootTest

This annotation indicates that the context under test is a `@SpringBootApplication`. Basically these annotation complete launch the project during the unit test.

`@MockBean` annotation creates the mock/fake objects instead of real object. This mock is used in the spring context instead of the real objects.

@WebMvcTest()

this is used to test the specific controller. It loads the specific class instead of whole project which we want to test.

@AutoConfigureMockMvc

that can be applied to test class to enable and configure auto configuration of `MockMvc`. Means `MockMvc` will be available in Context which is used by `@Autowired`

@AutoConfigureTestDatabase(replace=Replace.NONE)

This annotation is helpful for testing the application with real database.

@MockBean

helpful for to create the mock/fake object in context.

@SpringBootTest

helpful for integration testing

webEnvironment=WebEnvironment.RANDOM_PORT

it is basically used to load the web server with random port number to test the application.

@InjectMock

it is helpful to inject the dependent object to target class

@Mock

create mock object

@DataJpaTest

to test the persistence logic or model layer, we can also test the in-memory database with help of these annotation.

DOZAR BEAN MAPPER

Dozer is a powerful library which can help us in avoiding lots of unnecessary code, while we want to copy data from one bean to another bean. It is mainly a bean-to-bean mapper that recursively copies data from one Java object to another Java object.

Why it is important?

Nowadays all enterprise projects are quite complex in nature and usually to accomplish certain business functionality, we need to call external systems, legacy components which requires transformations of different types of objects whose structure is more or less same, like domain Object to external service request/external service response to domain and so on.

Dozar dependency

```
<dependency>  
  <groupId>net.sf.dozer</groupId>  
  <artifactId>dozer</artifactId>  
  <version>5.5.1</version>  
</dependency>
```

ClassA.java

```
package com.example.howtodoinjava.dozer.simple;
```

```
public class ClassA {  
  
    private String name;  
    private String age;  
    private String address;  
  
    //Getters and Setters  
}
```

ClassB.java

```
package com.example.howtodoinjava.dozer.simple;
```

```
public class ClassB {  
  
    private String name;  
    private int age;  
    private String address;  
  
    //Getters and Setters  
}
```

A simple dozer bean mapping example.

```
SimpleExample.java
package com.example.howtodoinjava.dozer.simple;

import org.dozer.DozerBeanMapper;

public class SimpleExample
{
    public static void main(String[] args)
    {
        ClassA classA = new ClassA();
        classA.setAddress("India");
        classA.setName("Sajal");
        classA.setAge("50");

        ClassB classB = new DozerBeanMapper().map(classA,
ClassB.class);
        System.out.println(classB);
    }
}
```

Object Mapper class

The Jackson ObjectMapper class (com.fasterxml.jackson.databind.ObjectMapper) is the simplest way to parse JSON with Jackson. The Jackson ObjectMapper can parse JSON from a string, stream or file, and create a Java object or object graph representing the parsed JSON. Parsing JSON into Java objects is also referred to as to deserialize Java objects from JSON.

The Jackson ObjectMapper can also create JSON from Java objects. Generating JSON from Java objects is also referred to as to serialize Java objects into JSON.

The Jackson Object mapper can parse JSON into objects of classes developed by you, or into objects of the built-in JSON tree model explained later in this tutorial.

By the way, the reason it is called ObjectMapper is because it maps JSON into Java Objects (deserialization), or Java Objects into JSON (serialization)

click below link to object mapper details

<http://tutorials.jenkov.com/java-json/jackson-objectmapper.html>