**Testing RESTful Services with Spring Boot and Mockito:**

**Example 1: HelloWorldController :**

**Create an Spring boot web application and write below simple controller which does not have any dependency as of now:**

package com.example.demo.controller;

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

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

@RestController

public class HelloWorldController {

@RequestMapping("/hello-world")

public String helloWorld() {

return "Hello World";

}

}

In this type of unit test, you don't want to launch the entire application while testing the web application through unit test. If I launch the application, load the all the controller and component then it would become the Integration Test but here, we are writing the Unit Test , not Integration Test.

**Question: Now** we want to test this above one specific 'HelloWorldController' , by invoking "/hello-world" and checking whether this is returning "Hello-World" or not?

**Ans.:**

**HelloWorldControllerTest.java:**

package com.example.demo.controller;

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;

@RunWith(SpringRunner.class) **// This helps to launch the Spring Unit Test.**

**/\*\*\*\* This tells that this is MVC test and telling this test which controller need to be test \*\*\*\*/**

@WebMvcTest(value=HelloWorldController.class)

public class HelloWorldControllerTest {

**/\*\*\*\* TO call the "/hello-world" , we need 'mockMvc' from spring \*\*\*\*/**

@Autowired

private MockMvc mockMvc;

**/\*\*\*\* To test the URL mapping (/hello-world) in HelloWorldController.below class, we need to do below two things:**

**\*         1. //call GET "/hello-world" application/json**

**\*         2. //verify "Hello World"**

**\*/**

@Test

public void helloWorld\_basic() throws Exception {

**//call GET "/hello-world" application/json**

**//Building the request which takes the URI "/hello-world" and which accepts the format in JSON.**

RequestBuilder request =

MockMvcRequestBuilders

.get("/hello-world")

.accept(MediaType.APPLICATION\_JSON);

**//Need to invoke 'perform' the above built request.**

MvcResult result = mockMvc.perform(request)

.andExpect(status().isOk()) // to expect the status back and checking whether status is OK or not.

.andExpect(content().string("Hello World")) //to expect the content.

.andReturn(); // need to use this method to return the result back.

**//verify "Hello World"**

**//assertEquals("Hello World",**

**/\*\*\*\* We should use 'getContentAsString()' to convert the response into String in Spring MVC, don't use 'toString' method.**

**\* If we use toString' method, we will get exception while testing this test class.**

**\*/**

result.getResponse().getContentAsString();

}

}

**Example 2: Basic JSON Assertion in web application:**

**Item.java:**

package com.example.demo.model;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Transient;

@Entity

public class Item {

@Id

private int id;

private String name;

private int price;

private int quantity;

@Transient

private int value;

protected Item() {

}

public Item(int id, String name, int price, int quantity)

{

this.id = id;

this.name = name;

this.price = price;

this.quantity = quantity;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

public int getPrice() {

return price;

}

public int getQuantity() {

return quantity;

}

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

public String toString() {

return String.format("Item[%d, %s, %d, %d]", id, name, price, quantity);

}

}

**ItemController.java:**

package com.example.demo.controller;

import java.util.List;

import

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

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

import

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

import com.example.demo.model.Item;

@RestController

public class ItemController {

@GetMapping("/dummy-item")

public Item dummyItem() {

return new Item(1, "Ball", 10, 100);

}

}

**ItemControllerTest.java:**

package com.example.demo.controller;

import static org.mockito.Mockito.when;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import java.util.Arrays;

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.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.example.demo.model.Item;

@RunWith(SpringRunner.class)

@WebMvcTest(ItemController.class)

public class ItemControllerTest {

@Autowired

private MockMvc mockMvc;

@Test

public void dummyItem\_basic() throws Exception {

RequestBuilder request =

MockMvcRequestBuilders

.get("/dummy-item")

.accept(MediaType.APPLICATION\_JSON);

MvcResult result = mockMvc.perform(request)

.andExpect(status().isOk())

.andExpect(content().json("{\"id\":1,\"name\":\"Ball\",\"price\":10,\"quantity\":100}"))

.andReturn();

//Under the hoodm this below line is getting invoked.

//JSONAssert.assertEquals(expected,actual, false);

result.getResponse().getContentAsString();

}}

**Above example worked internally for JSON like shown in below example:**

package com.example.demo.controller;

import org.json.JSONException;

import org.junit.Test;

import org.skyscreamer.jsonassert.JSONAssert;

public class JsonAssertTest {

String actualResponse = "{\"id\":1,\"name\":\"Ball\",\"price\":10,\"quantity\":100}"

;

@Test

public void

jsonAssert\_StrictTrue\_ExactMatchExceptForSpaces() throws

JSONException {

String expectedResponse = "{\"id\": 1,\"name\":\"Ball\", \"price\":10, \"quantity\":100}";

JSONAssert.assertEquals(expectedResponse,

actualResponse, true);

}

@Test

public void jsonAssert\_StrictFalse() throws

JSONException {

String expectedResponse = "{\"id\": 1,\"name\":\"Ball\", \"price\":10}";

JSONAssert.assertEquals(expectedResponse,

actualResponse, false);

}

@Test

public void jsonAssert\_WithoutEscapeCharacters()

throws JSONException

{

String expectedResponse = "{id:1,name:Ball, price:10}";

JSONAssert.assertEquals(expectedResponse,actualResponse, false);

} }

**Example 3: Testing a web service which is talking to Business Layer and creating the test which is independent of Business Layer:**

**ItemRepository.java:**

package com.example.demo.repo;

import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Item;

public interface ItemRepository extends JpaRepository<Item,Integer>{

}

**ItemController.java:**

package com.example.demo.controller;

import java.util.List;

import

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

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

import

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

import com.example.demo.model.Item;

import com.example.demo.service.ItemBusinessService;

@RestController

public class ItemController {

@Autowired

private ItemBusinessService businessService;

@GetMapping("/dummy-item")

public Item dummyItem() {

return new Item(1, "Ball", 10, 100);

}

@GetMapping("/item-from-business-service")

public Item itemFromBusinessService() {

Item item = businessService.retreiveHardcodedItem();

return item;

}

@GetMapping("/all-items-from-database")

public List<Item> retrieveAllItems() {

return businessService.retrieveAllItems();

}

}

**ItemBusinessService.java:**

package com.example.demo.service;

import java.util.List;

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

import org.springframework.stereotype.Component;

import com.example.demo.model.Item;

import com.example.demo.repo.ItemRepository;

@Component

public class ItemBusinessService {

@Autowired

private ItemRepository repository;

public Item retreiveHardcodedItem() {

return new Item(1, "Ball", 10, 100);

}

public List<Item> retrieveAllItems()

{

List<Item> items = repository.findAll();

for(Item item:items) {

item.setValue(item.getPrice() \* item.getQuantity());

}

return items;

}

}

**Example 4: ItemControllerTest.java: Writing the test for ItemController.java**

package com.example.demo.controller;

import static org.mockito.Mockito.when;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import java.util.Arrays;

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.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.example.demo.model.Item;

import com.example.demo.service.ItemBusinessService;

@RunWith(SpringRunner.class)

@WebMvcTest(ItemController.class)

public class ItemControllerTest {

**/\*\*\*\* This annotation is a shorthand for the Mockito.mock() method. As well,**

**\* we should only use it in a test class. Unlike the mock() method, we need to enable Mockito annotations to use this annotation.**

**\* We can do this either by using the MockitoJUnitRunner to run the test or calling the MockitoAnnotations.initMocks() method explicitly.**

**\*/**

@Autowired

private MockMvc mockMvc;

**/\*\*\*\* We can use the @MockBean to add mock objects to the Spring application context.**

**\* The mock will replace any existing bean of the same type in the application context.**

**\* If no bean of the same type is defined, a new one will be added.**

**\* This annotation is useful in integration tests where a particular bean – for example, an external service – needs to be mocked.**

**\*/**

@MockBean

private ItemBusinessService businessService;

@Test

public void dummyItem\_basic() throws Exception {

RequestBuilder request =

MockMvcRequestBuilders

.get("/dummy-item")

.accept(MediaType.APPLICATION\_JSON);

MvcResult result = mockMvc.perform(request)

.andExpect(status().isOk())

.andExpect(content().json("{\"id\":1,\"name\":\"Ball\",\"price\":10,\"quantity\":100}"))

.andReturn();

//Under the hoodm this below line is getting invoked.

//JSONAssert.assertEquals(expected,actual, false);

result.getResponse().getContentAsString();

}

**/\*\*\*\* Testing the URI "/item-from-business-service" which is dependent on ItemBusinessService**

**\* but we are writing the test which is independent of ItemBusinessService using mock.**

**\*/**

@Test

public void itemFromBusinessService\_basic() throws Exception {

when(businessService.retreiveHardcodedItem()).thenReturn(

new Item(2,"Item2",10,10));

RequestBuilder request =

MockMvcRequestBuilders

.get("/item-from-business-service")

.accept(MediaType.APPLICATION\_JSON);

MvcResult result = mockMvc.perform(request)

.andExpect(status().isOk())

.andExpect(content().json("{id:2,name:Item2,price:10}"))

.andReturn();

result.getResponse().getContentAsString();

}

**/\*\*\*\* Testing the URI "/all-items-from-database" which is dependent on ItemBusinessService**

**\* but we are writing the test which is independent of ItemBusinessService using mock.**

**\*/**

@Test

public void retrieveAllItems\_basic() throws Exception {

when(businessService.retrieveAllItems()).thenReturn(

Arrays.asList(new Item(2,"Item2",10,10),

new Item(3,"Item3",20,20)));

RequestBuilder request = MockMvcRequestBuilders.get("/all-items-from-database")

.accept(MediaType.APPLICATION\_JSON);

MvcResult result = mockMvc.perform(request)

.andExpect(status().isOk())

//In JSON, array is represent in square bracket like below line:

.andExpect(content().json("[{id:3,name:Item3,price:20}, {id:2,name:Item2,price:10}]"))

.andReturn();

result.getResponse().getContentAsString();

}

@Test

public void retrieveAllItems\_noitems() throws Exception {

when(businessService.retrieveAllItems()).thenReturn(

Arrays.asList()

);

RequestBuilder request =

MockMvcRequestBuilders

.get("/all-items-from-database")

.accept(MediaType.APPLICATION\_JSON);

MvcResult result =        mockMvc.perform(request)

.andExpect(status().isOk())

.andExpect(content().json(" "))

.andReturn();

result.getResponse().getContentAsString();

}

}

**Example 5:ItemBusinessServiceTest.java: Writing the test for service class like here, for ItemBusinessService.java**

package com.example.demo.services;

import static org.junit.Assert.assertEquals;

import static org.mockito.Mockito.when;

import java.util.Arrays;

import java.util.List;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.mockito.InjectMocks;

import org.mockito.Mock;

import org.mockito.junit.MockitoJUnitRunner;

import com.example.demo.model.Item;

import com.example.demo.repo.ItemRepository;

import com.example.demo.service.ItemBusinessService;

@RunWith(MockitoJUnitRunner.class)

public class ItemBusinessServiceTest {

**/\*\*\*\***

**\* @Mock creates a mock. @InjectMocks creates an instance of the class and**

**\* injects the mocks that are created with the @Mock (or @Spy) annotations into this instance.**

**\***

**\* This tells Mockito which class to inject mocks into:**

**\***

**\* Note that you must use @RunWith(MockitoJUnitRunner.class) or Mockito.initMocks(this) to initialize these mocks and inject them.**

**\*/**

@InjectMocks

private ItemBusinessService business;

@Mock

private ItemRepository repository;

**/\*\*\*\***

**\* Testing method of 'retrieveAllItems' of ItemBusinessService.java**

**\*/**

@Test

public void retrieveAllItems\_basic() {

**//We have to Mock 'findAll()' method of JPA to get the data. We don't want to be dependent on outside of test**

when(repository.findAll()).thenReturn(Arrays.asList(new Item(2,"Item2",10,10),

new Item(3,"Item3",20,20)));

List<Item> items = business.retrieveAllItems();

assertEquals(100, items.get(0).getValue());

assertEquals(400, items.get(1).getValue());

}

}

**Example 6: ItemRepositoryTest.java: Writing the test for data JPA repository like here, for ItemRepository.java:**

Here, we don’t have any method in ItemRepository but if we use other custom repository like hibernate repository, we would be writing the test case as below:

package com.example.demo.repo;

import static org.junit.Assert.assertEquals;

import java.util.List;

import org.junit.Test;

import org.junit.runner.RunWith;

import

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

import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

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

import com.example.demo.model.Item;

@RunWith(SpringRunner.class)

@DataJpaTest **// To test JPA Repository**

public class ItemRepositoryTest {

@Autowired

private ItemRepository repository;

@Test

public void testFindAll() {

List<Item> items = repository.findAll();

assertEquals(3,items.size());

}

@Test

public void testFindOne() {

Item item = repository.findById(10001).get();

assertEquals("Item1",item.getName());

}

}

**Example 7: ItemRepositoryTest.java: Writing the integration test.**

package com.example.demo.controller;

import org.json.JSONException;

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.autoconfigure.SpringBootApplication;

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

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

import

org.springframework.boot.test.web.client.TestRestTemplate;

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

@RunWith(SpringRunner.class)

**/\*\*\*\***

**\* @SpringBootTest is saying “bootstrap with Spring Boot’s support” (e.g. load application.properties and give me all the Spring Boot goodness)**

**\* The webEnvironment attribute allows specific “web environments” to be configured for the test.**

**\* You can start tests with a MOCK servlet environment or with a real HTTP server running on either a RANDOM\_PORT or a DEFINED\_PORT.**

**\*If we want to load a specific configuration, we can use the classes attribute of @SpringBootTest.**

**\*In this example, we’ve omitted classes which means that the test will first attempt to load @Configuration from any inner-classes,**

**\*and if that fails, it will search for your primary @SpringBootApplication class.**

**\***

**\*By default, @SpringBootTest will not start a server. You can use the webEnvironment attribute of @SpringBootTest to further refine how your tests run:**

**\*MOCK(Default) : Loads a web ApplicationContext and provides a mock web environment. Embedded servers are not started when using this annotation.**

**\*If a web environment is not available on your classpath, this mode transparently falls back to creating a regular non-web ApplicationContext.**

**\*It can be used in conjunction with @AutoConfigureMockMvc or @AutoConfigureWebTestClient for mock-based testing of your web application.**

**\***

**\*RANDOM\_PORT: Loads a WebServerApplicationContext and provides a real web environment. Embedded servers are started and listen on a random port.**

**\***

**\*DEFINED\_PORT: Loads a WebServerApplicationContext and provides a real web environment. Embedded servers are started and listen on a**

**\*defined port (from your application.properties or on the default port of 8080).**

**\***

**\*NONE: Loads an ApplicationContext by using SpringApplication but does not provide any web environment (mock or otherwise).**

**\***

**\*@SpringBootTest loads the in-memory DB so that we are not dependent in real DB.**

**\*/**

@SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT)

public class ItemControllerIT {

**/\*\*\*\***

**\* Convenient alternative of RestTemplate that is suitable for integration tests.**

**\* If you are using the @SpringBootTest annotation, a TestRestTemplate is automatically available and can be @Autowired into your test.**

**\* If you need customizations (for example to adding additional message converters) use a RestTemplateBuilder @Bean.**

**\***

**\* TestRestTemplate is already aware which RANDOM\_PORT will be assigned.**

**\*/**

@Autowired

private TestRestTemplate restTemplate;

**/\*\*\*\* When we launch this test, then complete spring boot application will launch up.**

**\* What it does that, it sees all the packages and parent packages of naming package "com.example.demo.controller"**

**\* classes which has the @SpringBootApplication**

**\* If @SpringBootApplication is not found in package "com.example.demo.controller", then it goes to its parent package "com.example.demo."**

**\* so on.....**

**\*/**

@Test

public void contextLoads() throws JSONException

{

String response = this.restTemplate.getForObject("/all-items-from-database",String.class);

JSONAssert.assertEquals("[{id:10001},{id:10002},{id:10003}]",response, false);;

}

}

**Note:** you should monitor the logs as well.

Like we will see that tomcat stared on random available port no from logs: Tomcat started on port(s): 50010

**Points: Test Configuration File:**

* 1. If want to have different configuration file like 'application.properties' file specially for Test classes. So create one folder 'resources' under "src/test" and create 'application.properties' file and write own configuration.

If our test is using 'application.properties' file , first test classes looks this file from "src/test/resources"

If application.properties' file is not found in "src/test/resources", then test classes will use the application.properties' file from main resources.

Same thing applied for all configuration files.

* 1. Whatever we write any configuration file under "src/test/resources", will be used for all test classes by default.

* 1. When we want to have test-based configuration file. We can have separate property file and we need to define the path of new property file using @TestPropertySource. This configuration file will have the highest priority from another configuration file for specific test class.

package com.example.demo;

import org.junit.Test;

import org.junit.runner.RunWith;

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

import org.springframework.test.context.TestPropertySource;

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

@RunWith(SpringRunner.class)

@SpringBootTest

@TestPropertySource(locations= {"classpath:test-configuration.properties"})

public class UnitTestingApplicationTests {

@Test

public void contextLoads() {

}

}

**Example 8: Writing unit tests for other Request Methods(Put, Post, Delete, Get) and response status.**

* 1. **GET:**

@Test

public void helloWorld\_basic() throws Exception {

//call GET "/hello-world"  application/json

//Building the request which takes the URI "/hello-world" and which accepts the format in JSON.

RequestBuilder request =

MockMvcRequestBuilders

**.get("/hello-world")**

.accept(MediaType.APPLICATION\_JSON);

//Need to invoke 'perform' the above built request.

MvcResult result = mockMvc.perform(request)

**.andExpect(status().isOk())** // to expect the status back and checking whether status is OK or not.

.andExpect(content().string("Hello World")) //to expect the content.

.andReturn(); // need to use this method to return the result back.

//verify "Hello World"

//assertEquals("Hello World",

/\*\*\*\* We should use 'getContentAsString()' to convert the response into String in Spring MVC, don't use 'toString' method.

\* If we use toString' method, we will get exception while testing this test class.

\*/

result.getResponse().getContentAsString();

}

* 1. **POST:** We need to send the content along with POST request and we are writing with Header as well:

RequestBui1der requestBuiider — MockMvcRequestBu 
" / items • ) 
.accept(MediaType.APPLICATION JSON) 
ontent( 1, \ 
\ \ •t 100 • ) 
.contentType(MediaType.APPLICATION JSON) 
MvcResuit result — mockMvc.perform( requestBuiide. 
• andExpect( status ( ) • isCreatel 
. andExpect ( header( ) 