FACULTY OF ENGINEERING AND APPLIED SCIENCE

SOFE 3980U Software Quality Winter 2023

LAB TWO

Implementing and Testing Web Application and API Service using Apache Maven and Spring Boot

Ontario Tech University

SOFE 3980U Software Quality

CRN: 73385

Instructor(s): Mohamed El-Darieby

2022-02-21

Tasfia Alam – 100584647

Ahmaad Ansari – 100785574

Jacky Lee – 100787870

Mohammad Mohammadkhani – 100798165

Ashley Tyrone – 100786450

Table of Contents

I. Introduction

A. Spring Boot

Maven is a popular build automation tool for Java projects, and Spring Boot is a powerful framework for creating production-ready, standalone Spring-based applications. Together, they offer developers a streamlined approach to building and deploying Java applications. Maven manages project dependencies and automates the build process, while Spring Boot provides pre-configured settings and a range of components that simplify application development. With Maven and Spring Boot, developers can focus on writing code and creating value, rather than worrying about the underlying infrastructure. The combination of these two tools has become a popular choice for developing and deploying web applications, APIs, and microservices in the Java ecosystem.

B. API Service

Java API services are a fundamental building block for developing web applications, microservices, and other distributed systems. They provide a standardized interface for accessing and interacting with application functionality and data. With Java API services, developers can define a set of endpoints that expose their application's functionality over HTTP, making it accessible to other systems and clients.

C. Video Submission

Click here for the video submission. If the hyperlink is not working copy and paste the following link into your browser:

> https://drive.google.com/file/d/1GxULX9TaOFy-4LzidNrhyHfLipiXUkan/view?usp=share_link

The video can also be found in our GitHub repository. If the hyperlink is not working copy and paste the following link into your browser:

> https://github.com/ahmaad-ansari/SOFE3980U-Lab2-D2/tree/main

II. Discussion

A. Binary Controller Test

Source Code: Sample *getDefault* Test Case

```
@Test
public void getDefault() throws Exception {
    this.mvc.perform(get("/"))//.andDo(print())
        .andExpect(status().isOk())
        .andExpect(view().name("calculator"))
        .andExpect(model().attribute("operand1", ""))
        .andExpect(model().attribute("operand1Focused", false));
}
```

This is a JUnit test case for testing the default behaviour of a web application. It tests the GET request to the root URL ("/") and verifies the response status is "OK" (200) and the view name is ***calculator***. It also verifies that the model contains two attributes, ***operand1*** and ***operand1Focused***, and their values are an empty string and false, respectively.

Source Code: Sample *getParameter* Test Case

```
@Test
public void getParameter1() throws Exception {
    this.mvc.perform(get("/").param("operand1","111"))
        .andExpect(status().isOk())
        .andExpect(view().name("calculator"))
        .andExpect(model().attribute("operand1", "111"))
        .andExpect(model().attribute("operand1Focused", true));
}
```

This test case sends a GET request to the root URL ("/") with a query parameter ***operand1*** set to "111". It then expects the response status to be OK (200) and the view name to be ***calculator***. The test also checks if the model has two attributes named ***operand1*** and ***operand1Focused*** with values "111" and true respectively. This test case verifies that the controller correctly retrieves the value of ***operand1*** parameter and sets it in the model, along with the focus attribute for ***operand1***.

Source Code: Sample *postParameter* Test Cases

```
@Test
public void postParameterAdd1() throws Exception {
    this.mvc.perform(post("/").param("operand1","1000").param("operator","+").param("operand2","1111"))
        .andExpect(status().isOk())
        .andExpect(view().name("result"))
```

```
        .andExpect(model().attribute("result", "10111"))
        .andExpect(model().attribute("operand1", "1000"));
}
@Test
public void postParameterOr1() throws Exception {
    this.mvc.perform(post("/").param("operand1","1000").param("operator","|").param("operand2","1111"))
        .andExpect(status().isOk())
        .andExpect(view().name("result"))
        .andExpect(model().attribute("result", "1111"))
        .andExpect(model().attribute("operand1", "1000"));
}
@Test
public void postParameterAnd1() throws Exception {
    this.mvc.perform(post("/").param("operand1","1000").param("operator","&").param("operand2","1111"))
        .andExpect(status().isOk())
        .andExpect(view().name("result"))
        .andExpect(model().attribute("result", "1000"))
        .andExpect(model().attribute("operand1", "1000"));
}
@Test
public void postParameterMultiply1() throws Exception {
    this.mvc.perform(post("/").param("operand1","1000").param("operator","*").param("operand2","1111"))
        .andExpect(status().isOk())
        .andExpect(view().name("result"))
        .andExpect(model().attribute("result", "1111000"))
        .andExpect(model().attribute("operand1", "1000"));
}
```

These test cases above use the Spring MVC Test framework to test a POST request with three parameters - ***operand1***, ***operator***, and ***operand2*** - to the root URL ("/"). The expected result of the operation is either the sum, bitwise OR, bitwise AND, or product of ***operand1*** and ***operand2***. These test cases expect the response status to be OK (200), the view name to be "result", and the model attributes ***result*** and ***operand1*** to have specific values. The purpose of these test cases is to verify that the POST request is handled correctly and the expected result is displayed on the ***result*** view.

BinaryController.java Test Case Summary				
Test Case Name	Input Parameters	Expected Output	Target	Purpose
getDefault	operand1 =	status = OK view = calculator model[operand1] = model[operand1Focused] = false	get("/")	Verify that the model attributes operand1 and operand1Focused are set correctly when a GET request with an empty operand1 parameter is made to the root URL

getParameter1	operand1 = 111	status = OK view = calculator model[operand1] = 111 model[operand1Focused] = true	get("/")	Verify that the model attributes operand1 and operand1Focused are set correctly when a GET request with operand1 parameter is made to the root URL with 3 bits
getParameter2	operand1 = 1010	status = OK view = calculator model[operand1] = 1010 model[operand1Focused] = true	get("/")	Verify that the model attributes operand1 and operand1Focused are set correctly when a GET request with operand1 parameter is made to the root URL with 4 bits
getParameter3	operand1 = 11	status = OK view = calculator model[operand1] = 11 model[operand1Focused] = true	get("/")	Verify that the model attributes operand1 and operand1Focused are set correctly when a GET request with operand1 parameter is made to the root URL with 2 bits
getParameter4	operand1 = 0	status = OK view = calculator model[operand1] = 0 model[operand1Focused] = true	get("/")	Verify that the model attributes operand1 and operand1Focused are set correctly when a GET request with operand1 parameter is made to the root URL with 1 bit
getParameter5	operand1 = 0000	status = OK view = calculator model[operand1] = 0000 model[operand1Focused] = true	get("/")	Verify that the model attributes operand1 and operand1Focused are set correctly when a GET request with operand1 parameter is made to the root URL with 4 zero bits
getParameter6	operand1 =	status = OK view = calculator model[operand1] = model[operand1Focused] = false	get("/")	Verify that the model attributes operand1 and operand1Focused are set correctly when a GET request with operand1 parameter is made to the root URL with 0 bits
postParameterAdd1	operand1 = 1000 operator = + operand2 = 1111	status = OK view = result model[result] = 10111 model[operand1] = 1000	post("/")	Test that the POST request is handled correctly using two binary numbers of the same length using the

				+ operator
postParameterAdd2	operand1 = 1010 operator = + operand2 = 11	status = OK view = result model[result] = 1101 model[operand1] = 1010	post("/")	Test that the POST request is handled correctly when the first operand is greater than the second using the + operator
postParameterAdd3	operand1 = 11 operator = + operand2 = 1010	status = OK view = result model[result] = 1101 model[operand1] = 11	post("/")	Test that the POST request is handled correctly when the first operand is less than the second using the + operator
postParameterAdd4	operand1 = 0 operator = + operand2 = 1010	status = OK view = result model[result] = 1010 model[operand1] = 0	post("/")	Test that the POST request is handled correctly when one operand is a zero using the + operator
postParameterAdd5	operand1 = 0 operator = + operand2 = 0	status = OK view = result model[result] = 0 model[operand1] = 0	post("/")	Test that the POST request is handled correctly when both operands are zero using the + operator
postParameterOr1	operand1 = 1000 operator = | operand2 = 1111	status = OK view = result model[result] = 1111 model[operand1] = 1000	post("/")	Test that the POST request is handled correctly using two binary numbers of the same length using the | operator
postParameterOr2	operand1 = 1010 operator = | operand2 = 11	status = OK view = result model[result] = 1011 model[operand1] = 1010	post("/")	Test that the POST request is handled correctly when the first operand is greater than the second using the | operator
postParameterOr3	operand1 = 11 operator = | operand2 = 1010	status = OK view = result model[result] = 1011 model[operand1] = 11	post("/")	Test that the POST request is handled correctly when the first operand is less than the second using the | operator
postParameterOr4	operand1 = 0 operator = | operand2 = 1010	status = OK view = result model[result] = 1010 model[operand1] = 0	post("/")	Test that the POST request is handled correctly when one operand is a zero using the | operator
postParameterOr5	operand1 = 0 operator = | operand2 = 0	status = OK view = result model[result] = 0 model[operand1] = 0	post("/")	Test that the POST request is handled correctly when both operands are zero

				using the | operator
postParameterAnd1	operand1 = 1000 operator = & operand2 = 1111	status = OK view = result model[result] = 1000 model[operand1] = 1000	post("/")	Test that the POST request is handled correctly using two binary numbers of the same length using the & operator
postParameterAnd2	operand1 = 1010 operator = & operand2 = 11	status = OK view = result model[result] = 10 model[operand1] = 1010	post("/")	Test that the POST request is handled correctly when the first operand is greater than the second using the & operator
postParameterAnd3	operand1 = 11 operator = & operand2 = 1010	status = OK view = result model[result] = 10 model[operand1] = 11	post("/")	Test that the POST request is handled correctly when the first operand is less than the second using the & operator
postParameterAnd4	operand1 = 0 operator = & operand2 = 1010	status = OK view = result model[result] = 0 model[operand1] = 0	post("/")	Test that the POST request is handled correctly when one operand is a zero using the & operator
postParameterAnd5	operand1 = 0 operator = & operand2 = 0	status = OK view = result model[result] = 0 model[operand1] = 0	post("/")	Test that the POST request is handled correctly when both operands are zero using the & operator
postParameterMultiply1	operand1 = 1000 operator = * operand2 = 1111	status = OK view = result model[result] = 1111000 model[operand1] = 1000	post("/")	Test that the POST request is handled correctly using two binary numbers of the same length using the * operator
postParameterMultiply2	operand1 = 1010 operator = * operand2 = 11	status = OK view = result model[result] = 11110 model[operand1] = 1010	post("/")	Test that the POST request is handled correctly when the first operand is greater than the second using the * operator
postParameterMultiply3	operand1 = 11 operator = * operand2 = 1010	status = OK view = result model[result] = 11110 model[operand1] = 11	post("/")	Test that the POST request is handled correctly when the first operand is less than the second using the * operator
postParameterMultiply4	operand1 = 0 operator = * operand2 = 1010	status = OK view = result model[result] = 0	post("/")	Test that the POST request is handled correctly when one

		model[operand1] = 0		operand is a zero using the * operator
postParameterMultiply5	operand1 = 0 operator = * operand2 = 0	status = OK view = result model[result] = 0 model[operand1] = 0	post("/")	Test that the POST request is handled correctly when both operands are zero using the * operator

B. Binary API Controller Test

Source Code: Sample *add* and *add_json* Test Cases

```
@Test
public void add1() throws Exception {
    this.mvc.perform(get("/add").param("operand1","1000").param("operand2","1111"))
        .andExpect(status().isOk())
        .andExpect(content().string("10111"));
}
@Test
public void add_json1() throws Exception {
    this.mvc.perform(get("/add_json").param("operand1","1000").param("operand2","1111"))
        .andExpect(status().isOk())
        .andExpect(MockMvcResultMatchers.jsonPath("$.operand1").value(1000))
        .andExpect(MockMvcResultMatchers.jsonPath("$.operand2").value(1111))
        .andExpect(MockMvcResultMatchers.jsonPath("$.result").value(10111))
        .andExpect(MockMvcResultMatchers.jsonPath("$.operator").value("add"));
}
```

Source Code: Sample *or* and *or_json* Test Cases

```
@Test
public void or1() throws Exception {
    this.mvc.perform(get("/or").param("operand1","1000").param("operand2","1111"))
        .andExpect(status().isOk())
        .andExpect(content().string("1111"));
}
@Test
public void or_json1() throws Exception {
    this.mvc.perform(get("/or_json").param("operand1","1000").param("operand2","1111"))
        .andExpect(status().isOk())
        .andExpect(MockMvcResultMatchers.jsonPath("$.operand1").value(1000))
        .andExpect(MockMvcResultMatchers.jsonPath("$.operand2").value(1111))
        .andExpect(MockMvcResultMatchers.jsonPath("$.result").value(1111))
        .andExpect(MockMvcResultMatchers.jsonPath("$.operator").value("or"));
}
```

Source Code: Sample *and* and *and_json* Test Cases

```
@Test
```

```
public void and1() throws Exception {
    this.mvc.perform(get("/and").param("operand1","1000").param("operand2","1111"))
        .andExpect(status().isOk())
        .andExpect(content().string("1000"));
}
@Test
public void and_json1() throws Exception {
    this.mvc.perform(get("/and_json").param("operand1","1000").param("operand2","1111"))
        .andExpect(status().isOk())
        .andExpect(MockMvcResultMatchers.jsonPath("$.operand1").value(1000))
        .andExpect(MockMvcResultMatchers.jsonPath("$.operand2").value(1111))
        .andExpect(MockMvcResultMatchers.jsonPath("$.result").value(1000))
        .andExpect(MockMvcResultMatchers.jsonPath("$.operator").value("and"));
}
```

Source Code: Sample *multiply* and *multiply_json* Test Cases

```
@Test
public void multiply1() throws Exception {
   this.mvc.perform(get("/multiply").param("operand1","1000").param("operand2","1111"))
       .andExpect(status().isOk())
       .andExpect(content().string("1111000"));
}
@Test
public void multiply_json1() throws Exception {
   this.mvc.perform(get("/multiply_json").param("operand1","1000").param("operand2","1111"))
       .andExpect(status().isOk())
       .andExpect(MockMvcResultMatchers.jsonPath("$.operand1").value(1000))
       .andExpect(MockMvcResultMatchers.jsonPath("$.operand2").value(1111))
       .andExpect(MockMvcResultMatchers.jsonPath("$.result").value(1111000))
       .andExpect(MockMvcResultMatchers.jsonPath("$.operator").value("multiply"));
}
```

The first test case in each of the sample source codes above is targeting a function that is associated with the ***/add***, ***/or***, ***/and***, or ***/multiply*** endpoints, and its purpose is to test whether the function returns the correct result after adding two operands. The test is performed by making an HTTP GET request to the corresponding endpoint and passing two parameters named ***operand1*** and ***operand2***.

The second test case in each of the sample source codes above is targeting a function that is associated with the ***/add_json***, ***/or_json***, ***/and_json***, or ***/multiply_json*** endpoints, and its purpose is to test whether the function returns the correct JSON response after adding two operands. The test is performed by making an HTTP GET request to the corresponding endpoint and passing two parameters named ***operand1*** and ***operand2***. The expected JSON response includes the operands, the

operator, and the result. The test checks whether these values are correctly returned in the response.

BinaryAPIController.java Test Case Summary				
Test Case Name	Input Parameters	Expected Output	Target	Purpose
add1	operand1 = 1000 operand2 = 1111	10111	/add	Test the add functions with two binary numbers of the same length
add_json1	operand1 = 1000 operand2 = 1111	{"operand1":1000, "operand2":1111, "result":10111, "operator":"add"}	/add_json	
add2	operand1 = 1010 operand2 = 11	1101	/add	Test the add functions with two binary numbers, the length of the first argument is greater than the second
add_json2	operand1 = 1010 operand2 = 11	{"operand1":1010, "operand2":11, "result":1101, "operator":"add"}	/add_json	
add3	operand1 = 11 operand2 = 1010	1101	/add	Test the add functions with two binary numbers, the length of the first argument is less than the second
add_json3	operand1 = 11 operand2 = 1010	{"operand1":11, "operand2":1010, "result":1101, "operator":"add"}	/add_json	
add4	operand1 = 0 operand2 = 1010	1010	/add	Test the add functions with a binary number with zero
add_json4	operand1 = 0 operand2 = 1010	{"operand1":0, "operand2":1010, "result":1010, "operator":"add"}	/add_json	
add5	operand1 = 0 operand2 = 0	0	/add	Test the add functions with two zeros
add_json5	operand1 = 0 operand2 = 0	{"operand1":0, "operand2":0, "result":0, "operator":"add"}	/add_json	
add6	operand1 = abc operand2 = 1000	1000	/add	Test the add functions with the first argument as invalid input
add_json6	operand1 = abc operand2 = 1000	{"operand1":"0", "operand2":"1000", "result":1000, "operator":"add"}	/add_json	
add7	operand1 = 1000	1000	/add	Test the add functions

	operand2 = abc			with the second argument as invalid input
add_json7	operand1 = 1000 operand2 = abc	{"operand1":1000, "operand2":"abc", "result":1000, "operator":"add"}	/add_json	
add8	operand1 = abc operand2 = abc	0	/add	Test the add functions with both arguments as invalid input
add_json8	operand1 = abc operand2 = abc	{"operand1":"abc", "operand2":"abc", "result":0, "operator":"add"}	/add_json	
or1	operand1 = 1000 operand2 = 1111	1111	/or	Test the or functions with two binary numbers of the same length
or_json1	operand1 = 1000 operand2 = 1111	{"operand1":1000, "operand2":1111, "result":1111, "operator":"or"}	/or_json	
or2	operand1 = 1010 operand2 = 11	1011	/or	Test the or functions with two binary numbers, the length of the first argument is greater than the second
or_json2	operand1 = 1010 operand2 = 11	{"operand1":1010, "operand2":11, "result":1011, "operator":"or"}	/or_json	
or3	operand1 = 11 operand2 = 1010	1011	/or	Test the or functions with two binary numbers, the length of the first argument is less than the second
or_json3	operand1 = 11 operand2 = 1010	{"operand1":11, "operand2":1010, "result":1011, "operator":"or"}	/or_json	
or4	operand1 = 0 operand2 = 1010	1010	/or	Test the or functions with a binary number with zero
or_json4	operand1 = 0 operand2 = 1010	{"operand1":0, "operand2":1010, "result":1010, "operator":"or"}	/or_json	
or5	operand1 = 0 operand2 = 0	0	/or	Test the or functions with two zeros
or_json5	operand1 = 0 operand2 = 0	{"operand1":0, "operand2":0, "result":0, "operator":"or"}	/or_json	
or6	operand1 = abc operand2 = 1000	1000	/or	Test the or functions with the first argument as invalid input
or_json6	operand1 = abc	{"operand1":"0",	/or_json	

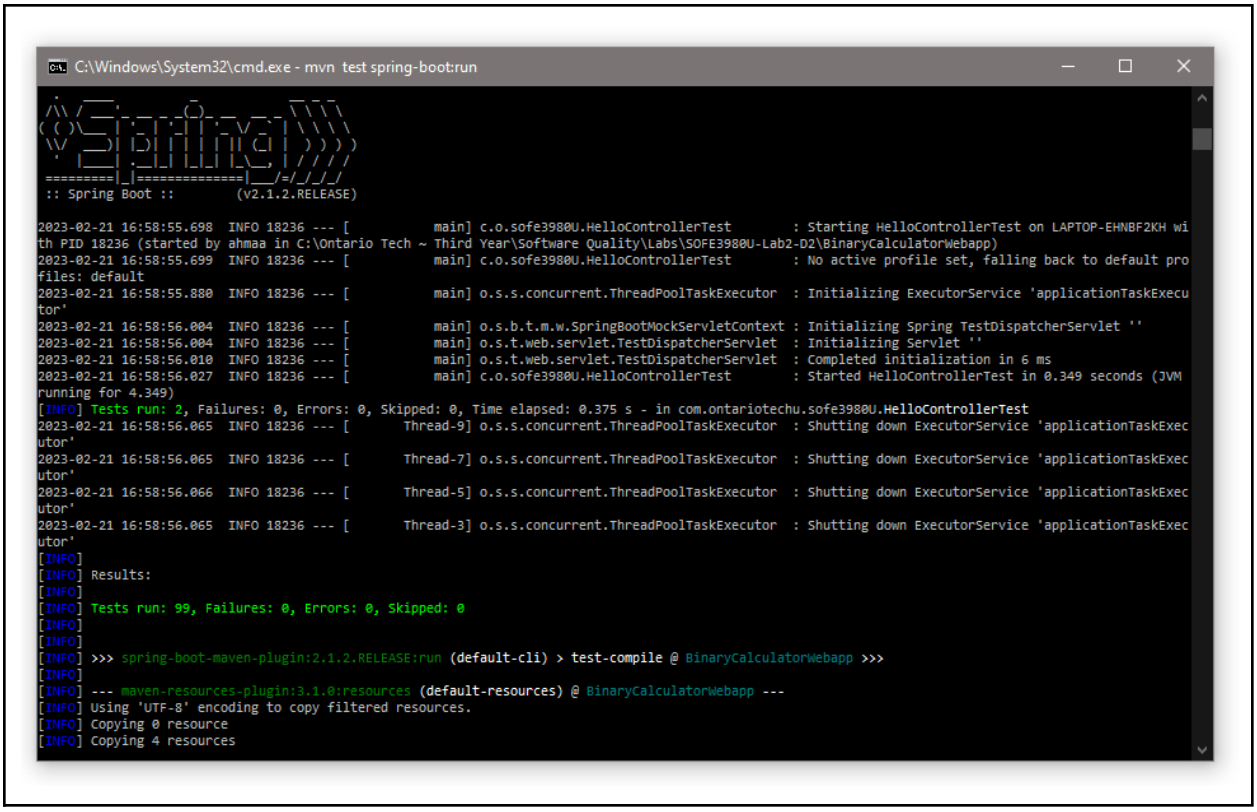
	operand2 = 1000	"operand2":"1000", "result":1000, "operator":"or"}		
or7	operand1 = 1000 operand2 = abc	1000	/or	Test the or functions with the second argument as invalid input
or_json7	operand1 = 1000 operand2 = abc	{"operand1":1000, "operand2":"abc", "result":1000, "operator":"or"}	/or_json	
or8	operand1 = abc operand2 = abc	0	/or	Test the or functions with both arguments as invalid input
or_json8	operand1 = abc operand2 = abc	{"operand1":"abc", "operand2":"abc", "result":0, "operator":"or"}	/or_json	
and1	operand1 = 1000 operand2 = 1111	1000	/and	Test the and functions with two binary numbers of the same length
and_json1	operand1 = 1000 operand2 = 1111	{"operand1":1000, "operand2":1111, "result":1000, "operator":"and"}	/and_json	
and2	operand1 = 1010 operand2 = 11	10	/and	Test the and functions with two binary numbers, the length of the first argument is greater than the second
and_json2	operand1 = 1010 operand2 = 11	{"operand1":1010, "operand2":11, "result":10, "operator":"and"}	/and_json	
and3	operand1 = 11 operand2 = 1010	10	/and	Test the and functions with two binary numbers, the length of the first argument is less than the second
and_json3	operand1 = 11 operand2 = 1010	{"operand1":11, "operand2":1010, "result":10, "operator":"and"}	/and_json	
and4	operand1 = 0 operand2 = 1010	0	/and	Test the and functions with a binary number with zero
and_json4	operand1 = 0 operand2 = 1010	{"operand1":0, "operand2":1010, "result":0, "operator":"and"}	/and_json	
and5	operand1 = 0 operand2 = 0	0	/and	Test the and functions with two zeros
and_json5	operand1 = 0 operand2 = 0	{"operand1":0, "operand2":0, "result":0, "operator":"and"}	/and_json	

and6	operand1 = abc operand2 = 1000	0	/and	Test the and functions with the first argument as invalid input
and_json6	operand1 = abc operand2 = 1000	{"operand1":"0", "operand2":"1000", "result":0, "operator":"and"}	/and_json	
and7	operand1 = 1000 operand2 = abc	0	/and	Test the and functions with the second argument as invalid input
and_json7	operand1 = 1000 operand2 = abc	{"operand1":1000, "operand2":"abc", "result":0, "operator":"and"}	/and_json	
and8	operand1 = abc operand2 = abc	0	/and	Test the and functions with both arguments as invalid input
and_json8	operand1 = abc operand2 = abc	{"operand1":"abc", "operand2":"abc", "result":0, "operator":"and"}	/and_json	
multiply1	operand1 = 1000 operand2 = 1111	1111000	/multiply	Test the multiply functions with two binary numbers of the same length
multiply_json1	operand1 = 1000 operand2 = 1111	{"operand1":1000, "operand2":1111, "result":1111000, "operator":"multiply"}	/multiply_json	
multiply2	operand1 = 1010 operand2 = 11	11110	/multiply	Test the multiply functions with two binary numbers, the length of the first argument is greater than the second
multiply_json2	operand1 = 1010 operand2 = 11	{"operand1":1010, "operand2":11, "result":11110, "operator":"multiply"}	/multiply_json	
multiply3	operand1 = 11 operand2 = 1010	11110	/multiply	Test the multiply functions with two binary numbers, the length of the first argument is less than the second
multiply_json3	operand1 = 11 operand2 = 1010	{"operand1":11, "operand2":1010, "result":11110, "operator":"multiply"}	/multiply_json	
multiply4	operand1 = 0 operand2 = 1010	0	/multiply	Test the multiply functions with a binary number with zero
multiply_json4	operand1 = 0 operand2 = 1010	{"operand1":0, "operand2":1010, "result":0, "operator":"multiply"}	/multiply_json	
multiply5	operand1 = 0 operand2 = 0	0	/multiply	Test the multiply functions with two zeros

multiply_json5	operand1 = 0 operand2 = 0	{"operand1":0, "operand2":0, "result":0, "operator":"multiply"}	/multiply_json	
multiply6	operand1 = abc operand2 = 1000	0	/multiply	Test the multiply functions with the first argument as invalid input
multiply_json6	operand1 = abc operand2 = 1000	{"operand1":"0", "operand2":"1000", "result":0, "operator":"multiply"}	/multiply_json	
multiply7	operand1 = 1000 operand2 = abc	0	/multiply	Test the multiply functions with the second argument as invalid input
multiply_json7	operand1 = 1000 operand2 = abc	{"operand1":1000, "operand2":"abc", "result":0, "operator":"multiply"}	/multiply_json	
multiply8	operand1 = abc operand2 = abc	0	/multiply	Test the multiply functions with both arguments as invalid input
multiply_json8	operand1 = abc operand2 = abc	{"operand1":"abc", "operand2":"abc", "result":0, "operator":"multiply"}	/multiply_json	

C. Test Summary

Sample Output: Maven Spring Boot



The screenshot above provides a section of the test summary generated by Maven. A test summary provided by Maven Spring Boot typically includes the following information:

- Total number of test cases run
- Number of successful test cases
- Number of failed test cases (if any)
- Time taken to execute the tests
- Percentage of successful test cases (in this case, 100%)
- Details of each successful test case, including the name of the test case, input parameters, expected output, and actual output

This summary gives a clear indication of the quality and reliability of the code, as well as any issues that may need to be addressed before deployment. It also provides a record of the testing process and can be used for future reference.

III. Conclusion

In this lab, we were tasked with adding more test cases to a binary web application and API service. We also had to implement other operators in both the web application and API services, and add test cases for each newly implemented operation.

Using Spring Boot with Maven is a powerful and efficient way to develop and deploy robust and scalable web applications. One of the main benefits of using Spring Boot with Maven is that it simplifies and streamlines the development process, making it easier to build, test, and deploy software. Spring Boot provides a wealth of features and libraries that can be easily integrated into the Maven project, saving developers time and effort. Another advantage of using Spring Boot with Maven is that it promotes best practices in software development, such as test-driven development, which helps to ensure the quality and reliability of the software. Additionally, Spring Boot provides a range of tools and utilities that make it easier to manage and monitor the application once it is deployed.

By completing this lab, we were able to further our understanding of test-driven development and the importance of thorough testing to ensure the quality and reliability of software. We were also able to gain practical experience in implementing and testing code in a real-world application.