**CS673 Software Engineering** 

**Team 1 - PennyWise**

**Software Test Document**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Clyde Yeung | Configuration Leader |  | 9/14/2023 |
| Jisoo Lee | Requirement leader | *Jisoo Lee* | 09/14/23 |
| Brian Fenstermacher | Team Leader | *Brian Fenstermacher* | 09/17/23 |
| Mali Rivera | QA Leader | *Mali Rivera* | 9/17/23 |
| Sherif zeyada | Design and Implementation leader | *Sherif Zeyada* |  |
| Sean Rawson | Security Leader | *Sean Rawson* | 9/23/23 |
| [Chaitanya Saraogi](mailto:csaraogi@bu.edu) | Requirements Leader | *Chaitanya saraogi* | 9/23/23 |
|  |  |  |  |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| **##** | **Chaitanya Saraogi** | **10/16/2023** | **Added unit testing for budget notification** |
|  |  |  |  |

[Testing Summary](#_heading=h.gjdgxs)

[Manuel Tests Reports](#_heading=h.30j0zll)

[Automated Testing Reports](#_heading=h.1fob9te)

[Testing Metrics](#_heading=h.3znysh7)

[References](#_heading=h.2et92p0)

[Glossary](#_heading=h.tyjcwt)

# Testing Summary

In this section, you will summarize what was tested, who is involved in testing, testing techniques used, and testing result. You may have the following tests

* + Unit Testing
  + Integration testing
  + System Testing
  + Acceptance Testing
  + Regression Testing

Unit Test (Mali):

* Please see the Automated Testing Report below for testing reports and testing examples
* Developers have written the unit tests for their code and the QA leader has written the integration tests
* For unit testing, we used JUnit5 and integrate Mockito
* Mockito is a framework that is used to mock interfaces so that dummy functionality can be added to a mock interface for Unit Testing purposes
* Steps: add dependencies, write tests with Junit5 and Mockito, run unit tests.
* 1. Setup dependencies

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<version>2.0.2-beta</version>

</dependency>

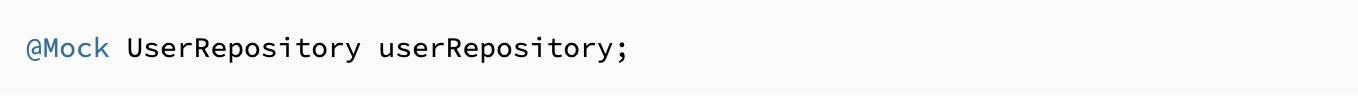
<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.8.2</version>

</dependency>

* 2. Integrate Mockito with Junit5 extension 
* 3. Use @Mock annotation to inject a mock instance that can be used anywhere in the test class (mock objects can be injected in method parameters)
* 
* 4. Inject a mock object into a test method



JUnit 5 User Guide:

<https://junit.org/junit5/docs/current/user-guide/>

Mockito documentation:

<https://site.mockito.org/>

Mockito integration with Junit5 extension:

<https://www.baeldung.com/mockito-junit-5-extension>

How to Write Test Cases in Java Application using Mockito and Junit:

<https://www.geeksforgeeks.org/how-to-write-test-cases-in-java-application-using-mockito-and-junit/>

Integration Test (Mali):

* Please see the Automated Testing Report below for testing reports and testing examples
* Developers have written the unit tests for their code and the QA leader has written the integration tests
* For integration testing, we are using MockMVC to test the web layer.
* Steps: add dependencies, write tests with MVC run integration tests
* 1. Add dependencies

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

</dependency>

* 2. Create JUnit test class to test MVC controller request and responses



* write JUNit tests for HTTP GET, POST, PUT and DELETE APIs.

Integration Testing in Spring:

<https://www.baeldung.com/integration-testing-in-spring>

MockMVC spring:

<https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html#:~:text=The%20Spring%20MVC%20Test%20framework%2C%20also%20known%20as,and%20response%20objects%20instead%20of%20a%20running%20server>.

Spring Boot MockMVC Examples:

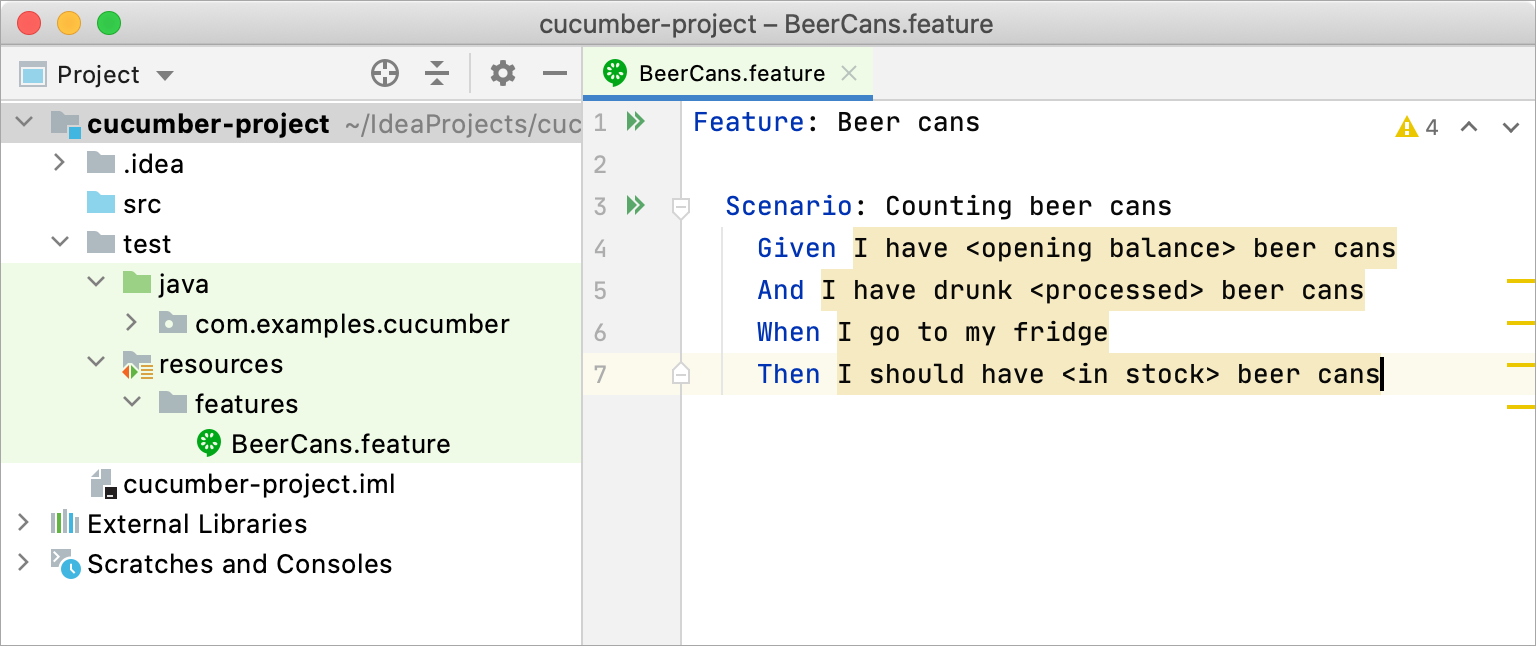
<https://www.geeksforgeeks.org/spring-boot-mockmvc-example/>

<https://howtodoinjava.com/spring-boot2/testing/spring-boot-mockmvc-example/>

Acceptance Test: (Jisoo / Chaitanya)

* jUnit5 and Cucumber
* Given the timeframe for this project, we were unable to add acceptance tests. As we work on PennyWise beyond the scope of this class, we plan to add acceptance tests with the Cucumber framework.
* Cucumber is a behavior-driven development framework.

Steps: add dependencies, write acceptance tests with Cucumber, run acceptance tests.



This is the structure of the project example for using Cucumber in Maven.

1. Add xml form: adding dependencies for our project.

<dependencies>

<!-- JUnit 5 -->

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>5.8.1</version> <!-- Use the latest version -->

<scope>test</scope>

</dependency>

<!-- Cucumber for Java -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>7.2.3</version> <!-- Use the latest version -->

<scope>test</scope>

</dependency>

</dependencies>

2. Writing acceptance tests with Cucumber: Cucumber allows developers to write in human syntax.

MyFeature.feature example:

Feature: My Feature

Scenario: User Registration

Given the user is on the registration page

When the user enters a valid username and password

And clicks the register button

Then the user should be registered successfully

3. Create step definition:

(<https://www.jetbrains.com/help/idea/creating-step-definition.html#add-cucumber-definitions> )

4. Run a Cucumber acceptance test within JUnit:

(<https://www.jetbrains.com/help/idea/running-cucumber-tests.html> )

Example:

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

import org.junit.runner.RunWith;

@RunWith(Cucumber.class)

@CucumberOptions( features =

{"classpath:features/BeerCans.feature"}, glue =

{"com.examples.cucumber"})

public class RunCucumberTest { }

* Cucumber set-up in IntelliJ

(<https://www.jetbrains.com/help/idea/enabling-cucumber-support-in-project.html#add-cucumber-library> )

* Run Cucumber test in IntelliJ (<https://www.jetbrains.com/help/idea/running-cucumber-tests.html#cucumber-run-configuration> )
* Cucumber (<https://www.jetbrains.com/help/idea/running-cucumber-tests.html#cucumber-run-configuration> )

# Manual Testing Report

In this section, you will give a detailed description of each manual test case performed and the result. If this is a previous You shall list what are existing tests developed in the previous semester and what are new tests developed currently.

Here is a sample template that can be used for each test case. For system tests or acceptance tests, you may also include some screenshots.

* Test case ID, name
* New or old:
* Test items: (what do you test )
* Test priority (high/medium/low)
* Dependencies (to other test case/requirement if any):
* Preconditions: (if any)
* input data:
* Test steps:
* Postconditions:
* Expected output:
* Actual output:
* Pass or Fail:
* Bug id/link: (this should link to your github issue id)
* Additional notes:

(You can use an additional spreadsheet for this section as well)

For our project, we used Gray Box Testing to create manual tests. We completed a number of manual tests throughout the course of development. Our manual tests were either completed with Postman or with our User Interface.

**Postman Manual Testing Report:**

[Manual Postman Testing Report](https://docs.google.com/spreadsheets/d/1zMkalAr5TjQV10E_pnp5QDhipSjv-5DffkTAloeut6Q/edit#gid=1737830860)

**UI Manual Tests:**

UITest: #1

Test Goal: Test that the user’s income data is saved successfully in the deployed database

Test priority: high

Preconditions: User exists in the database and has been successfully authorized to navigate to the dashboard page

Input: {name: “check”, amount: 123, username: “[jl3347311@gmail.com](mailto:jl3347311@gmail.com)”, date:”2023-10-16”}

Test Steps:

1. Developer inputs income data in “Add Income” form
2. Developer clicks on “Submit” button
3. Developer inspects console in browser for success/error message
4. Developer reviews database for income data
5. Developer confirms income data matches input income data

Expected result:

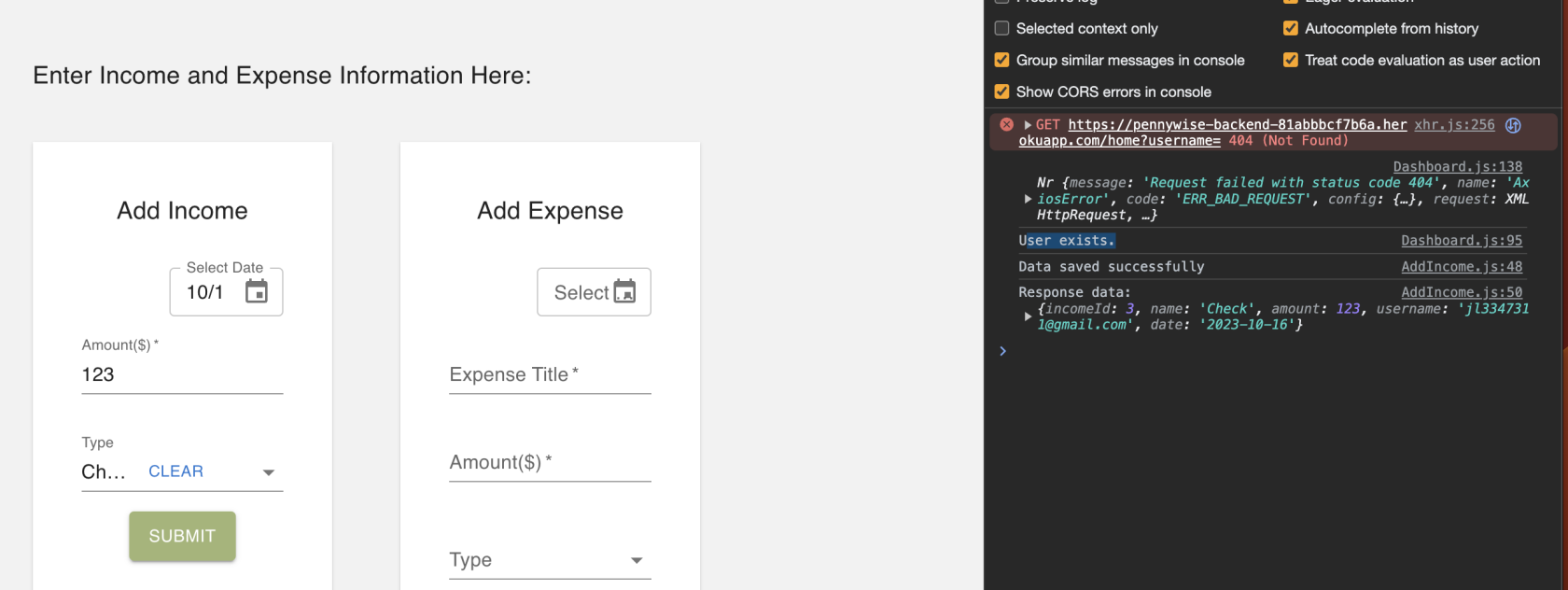
* Income data successfully added to the database and a success message is displayed in the console.

Actual result:

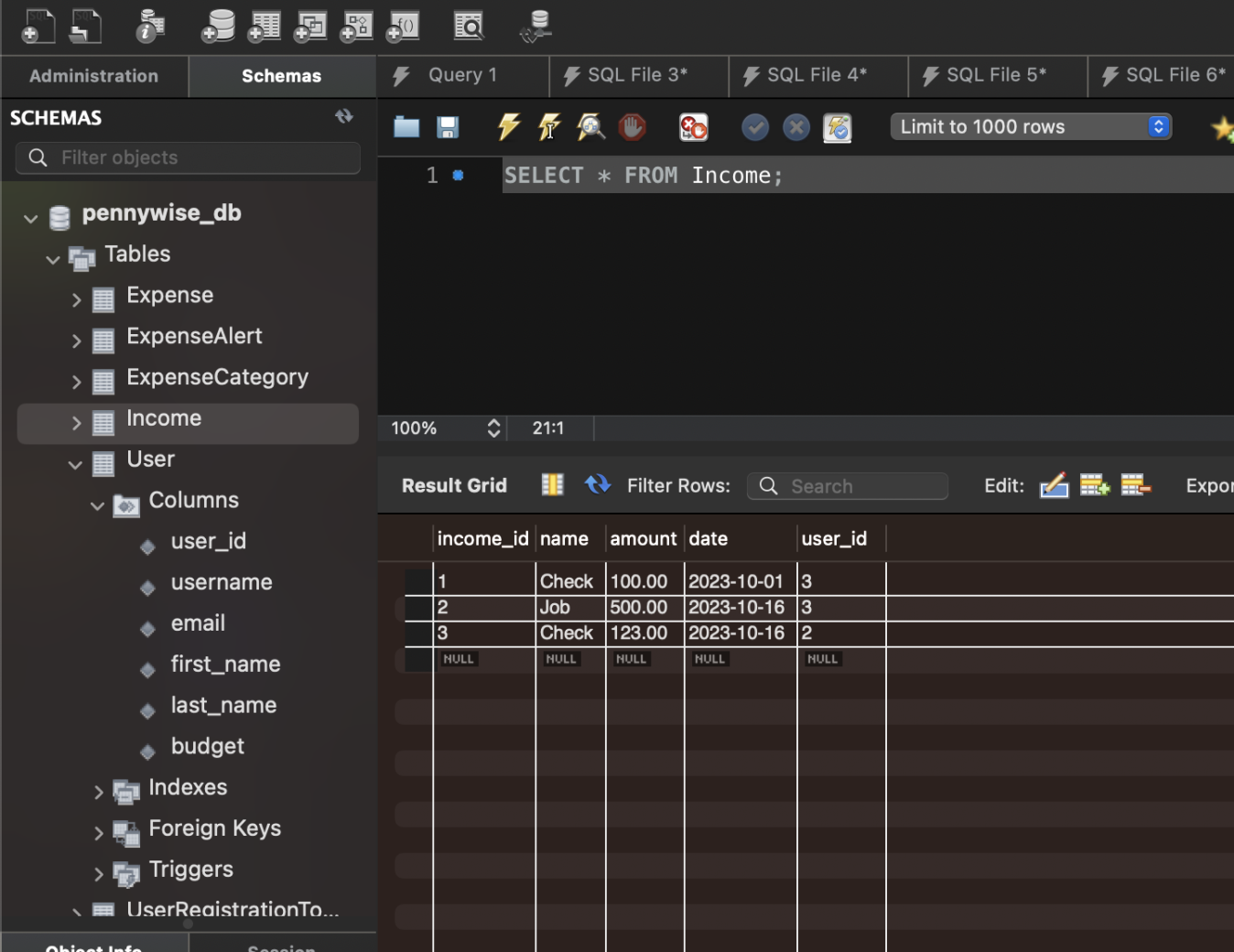
* Income data was successfully added to the database and a success message was displayed in the console.

Pass/Fail: **Pass**

Screenshot of user posting data to backend



Screenshot of data successfully receiving data

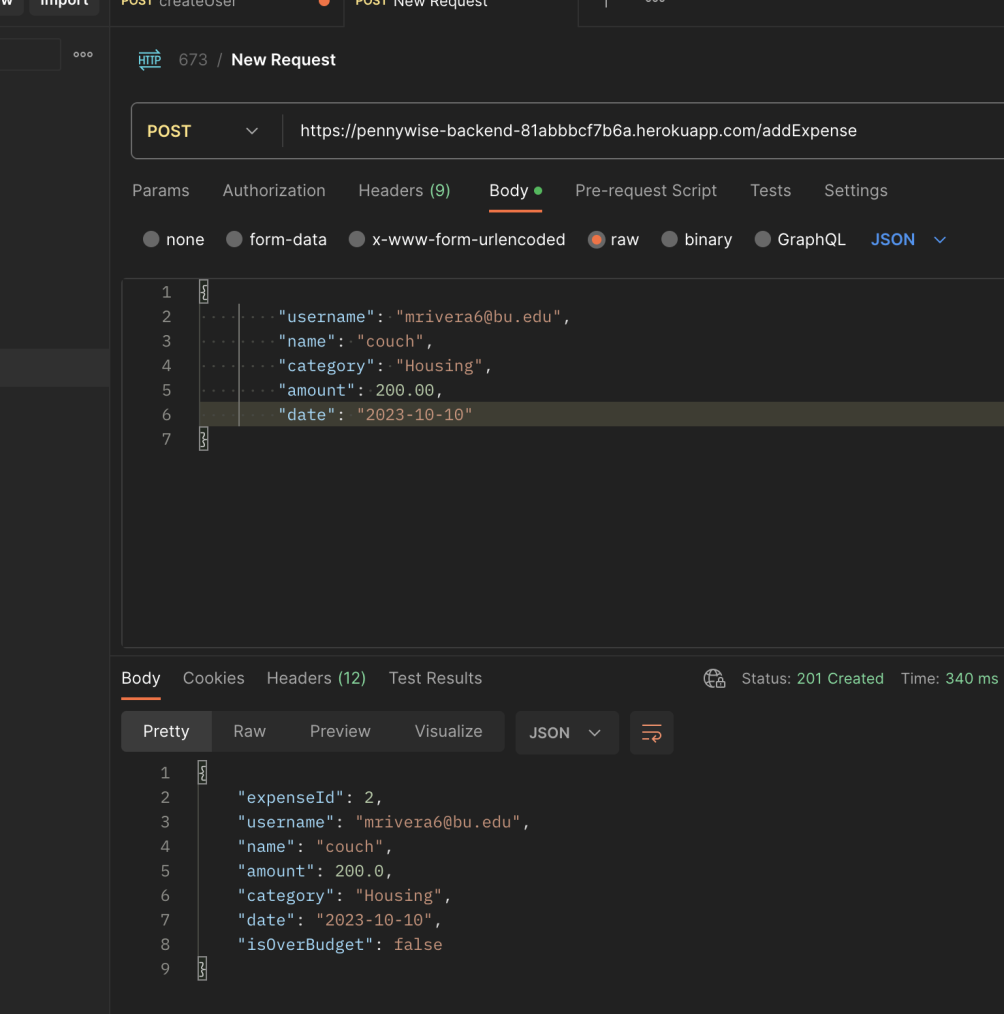


UITest#2

Test Goal: Test that the user’s expense data is saved successfully in the deployed database

Test priority: high

Screenshot of user posting expense in Postman



Test Steps:

1. Developer posts expense data in postman
2. Developer reviews database for expense data
3. Developer confirms expense data matches input expense data

Expected result:

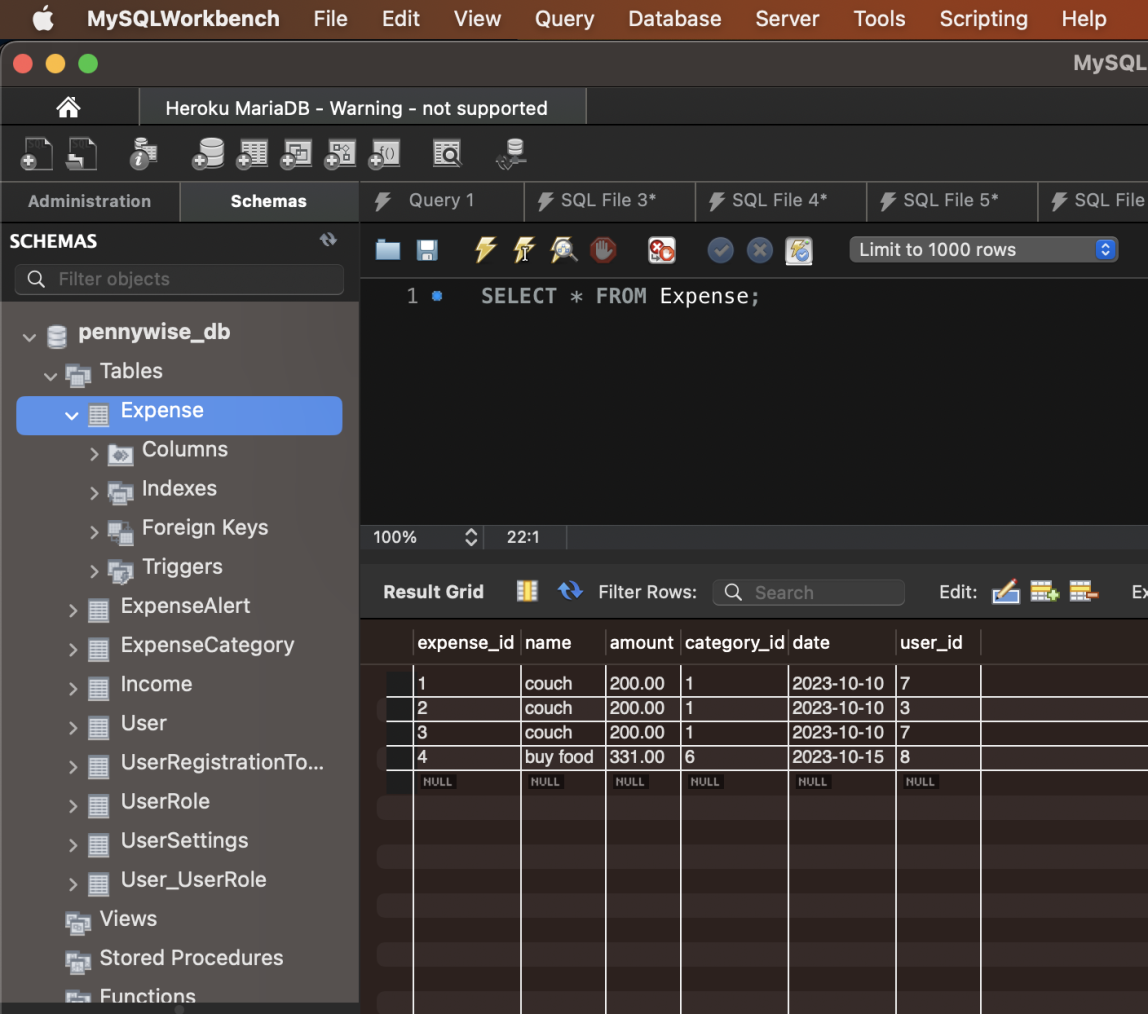
* Expense data is successfully added to the database

Actual result:

* Expense data was successfully added to the database

Pass/Fail: **Pass**

Screenshot of Expense data in DB



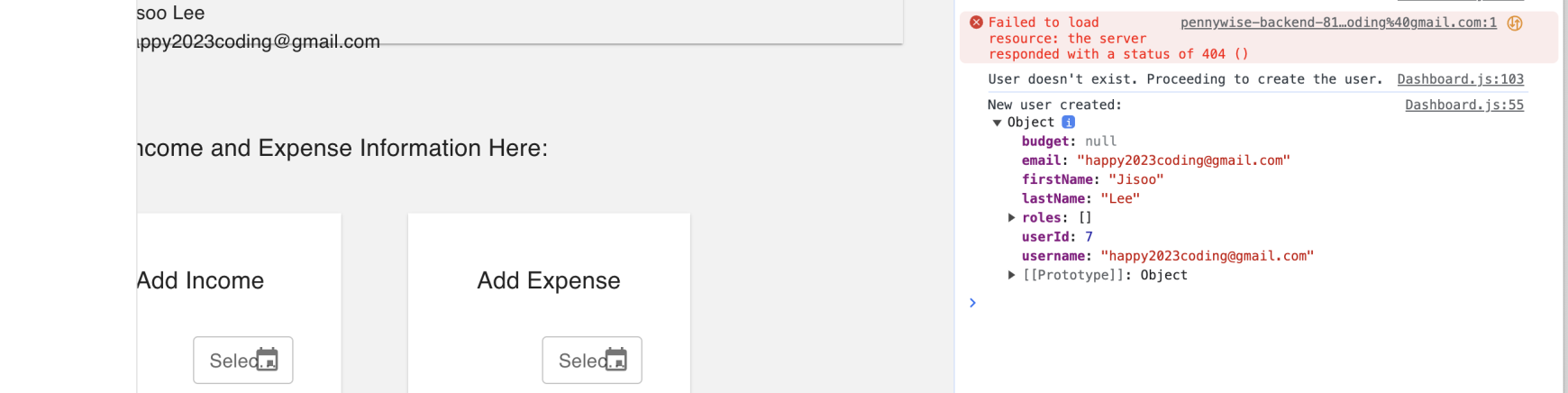
Manual Test #3

Test Goal: Test that user is created and stored in database when a user logins with google

Test priority: high

Input Steps:

1. Developer opens deployed website
2. Clicks on login with google button
3. Inspects console for success or failure message
4. Inspects DB to see if user was successfully stored



Expected result:

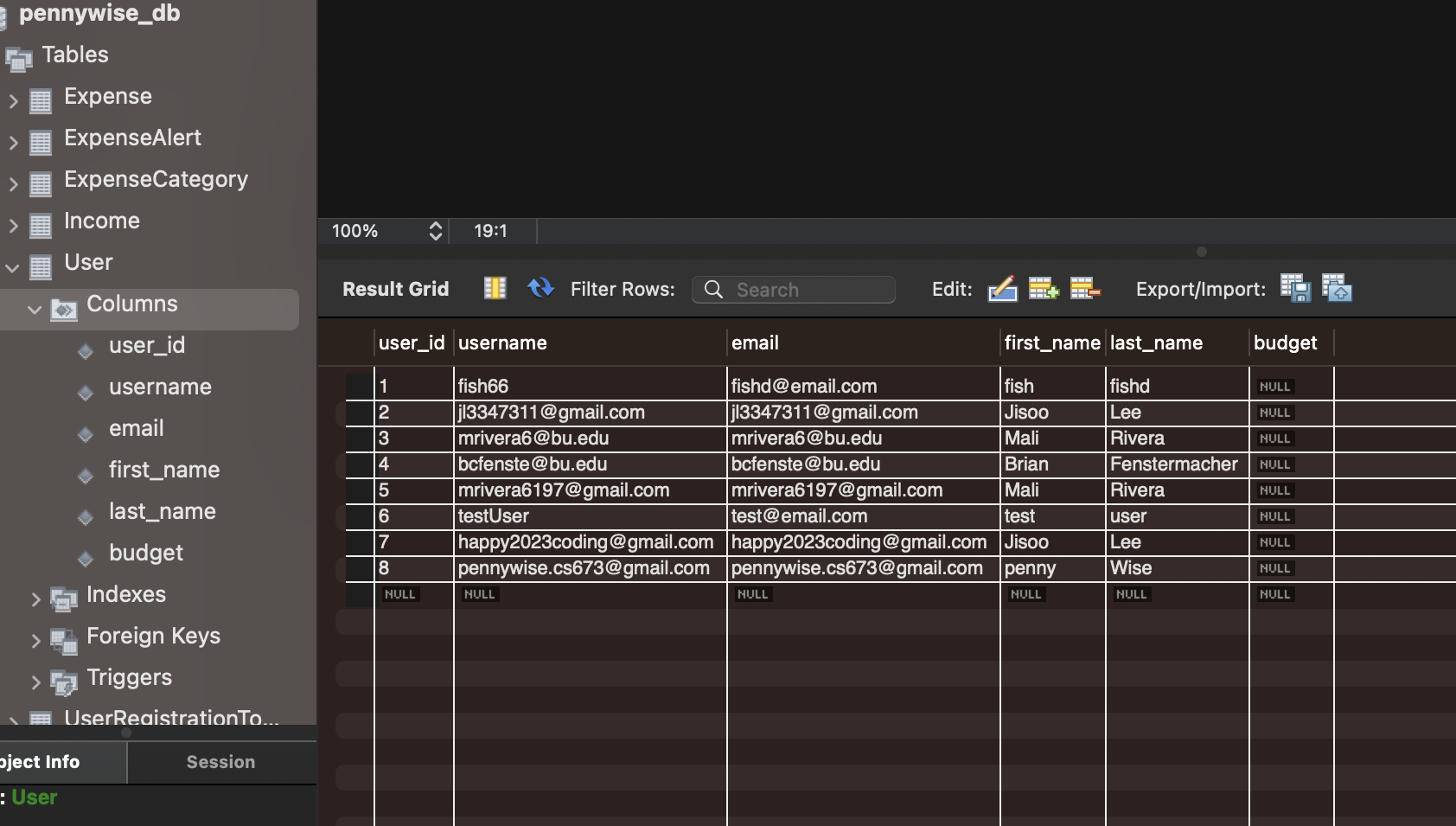
* User successfully added to the database and success message displayed in browser console

Actual result:

* User successfully added to the database and success message displayed in browser console

Pass/Fail: **Pass**

**Screenshot of user in database**

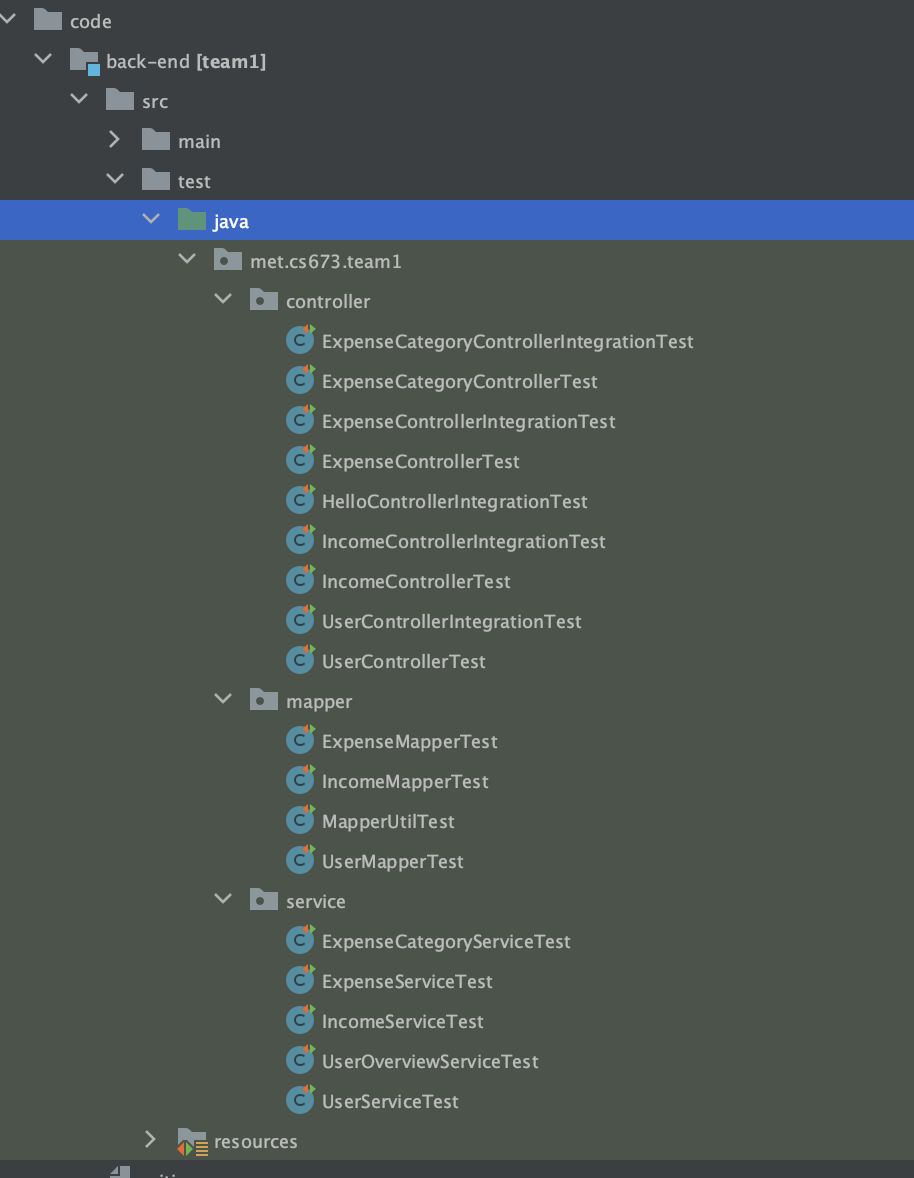


# Automated Testing Report (Mali)

Describe briefly the automated testing you have done, including where the test code resides in your code repository, what test frameworks are used, and the screen shots or generated testing report.

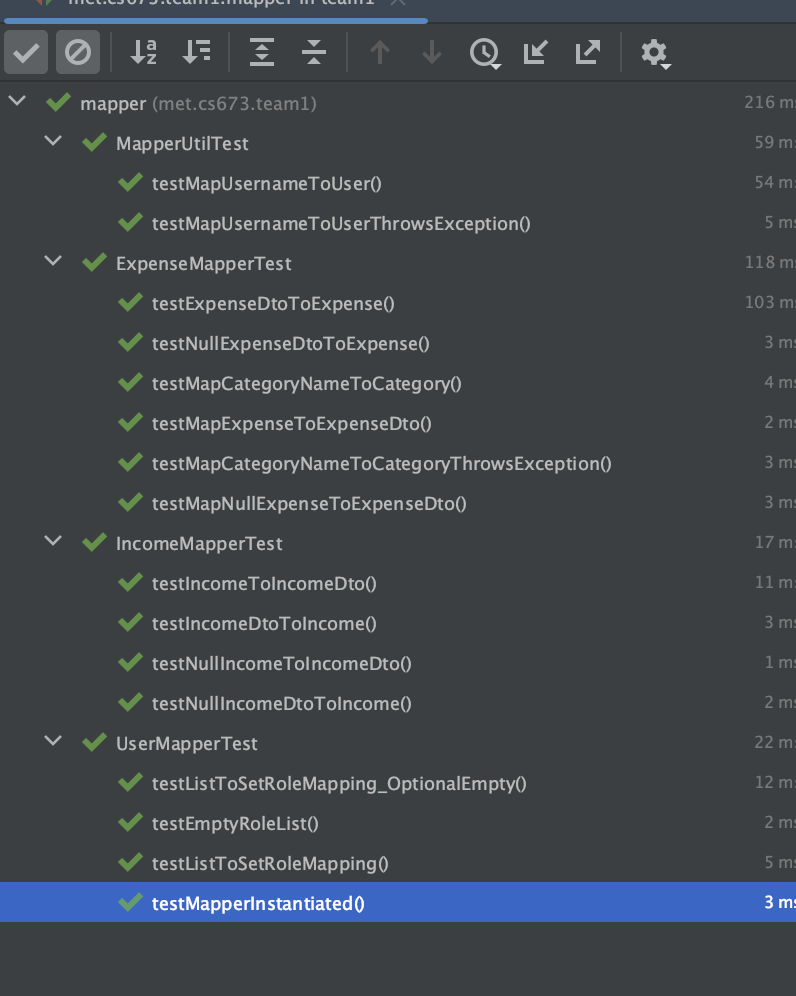
For our project, we have written Unit and Integration tests for our back end code. Unit tests were written by the developers with JUnit5 and Mockito. Integration tests were written by the QA leader with JUnit5, Mockito, and Mock MVC.

The Unit and Integration Tests reside in the following folder:

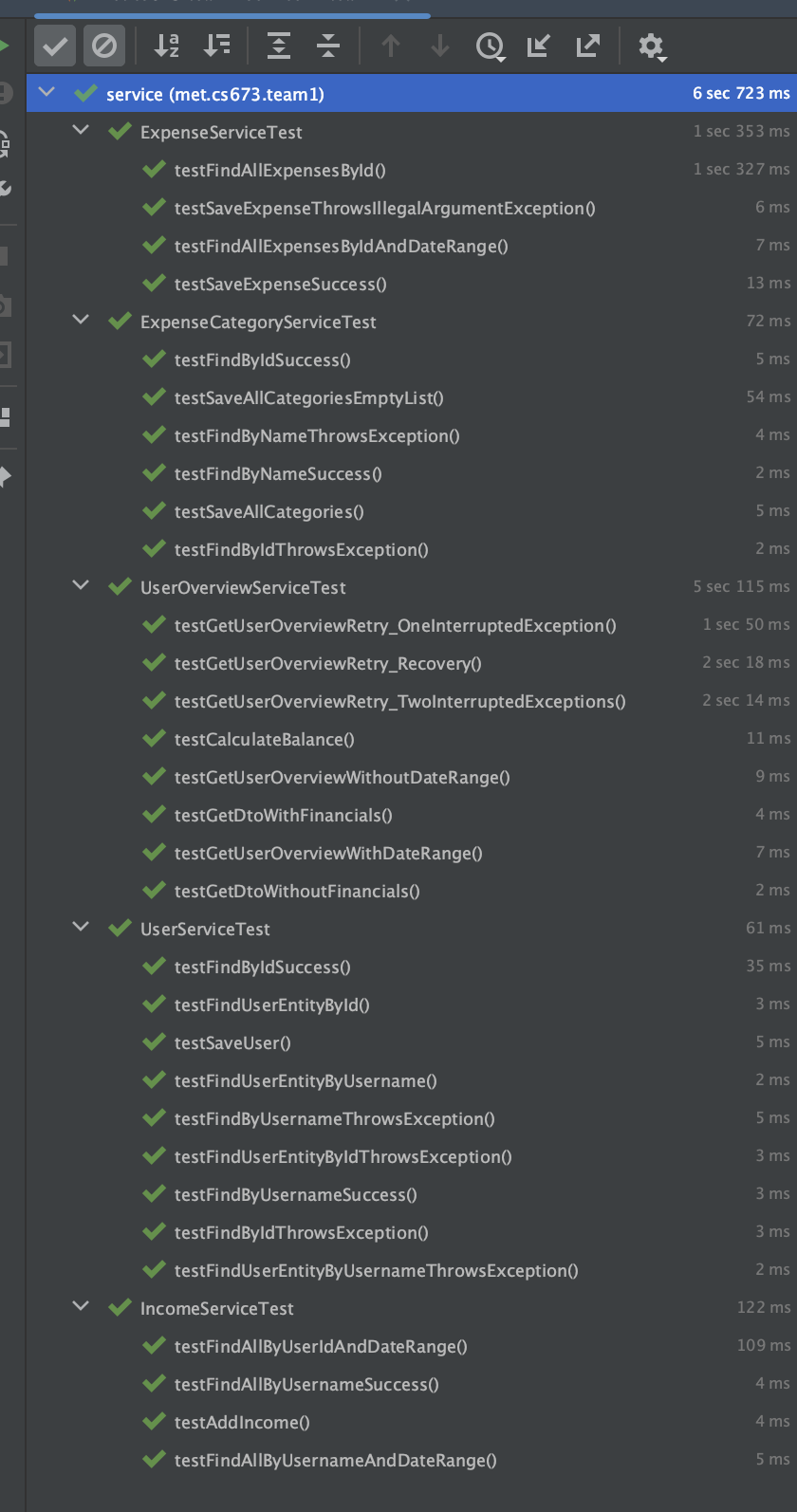


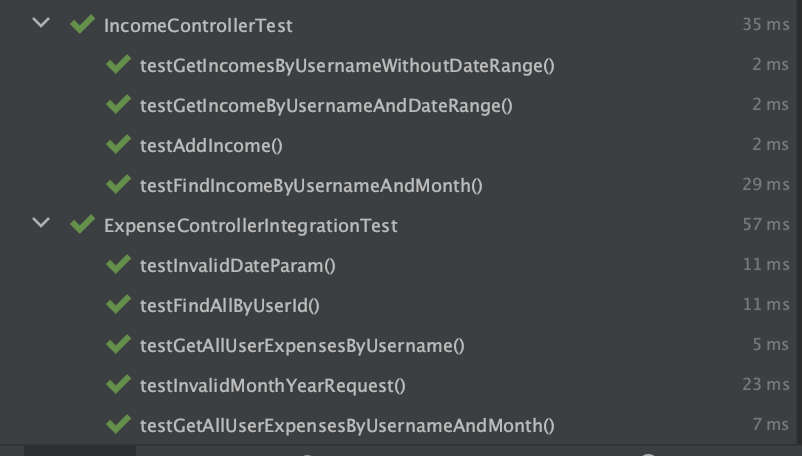
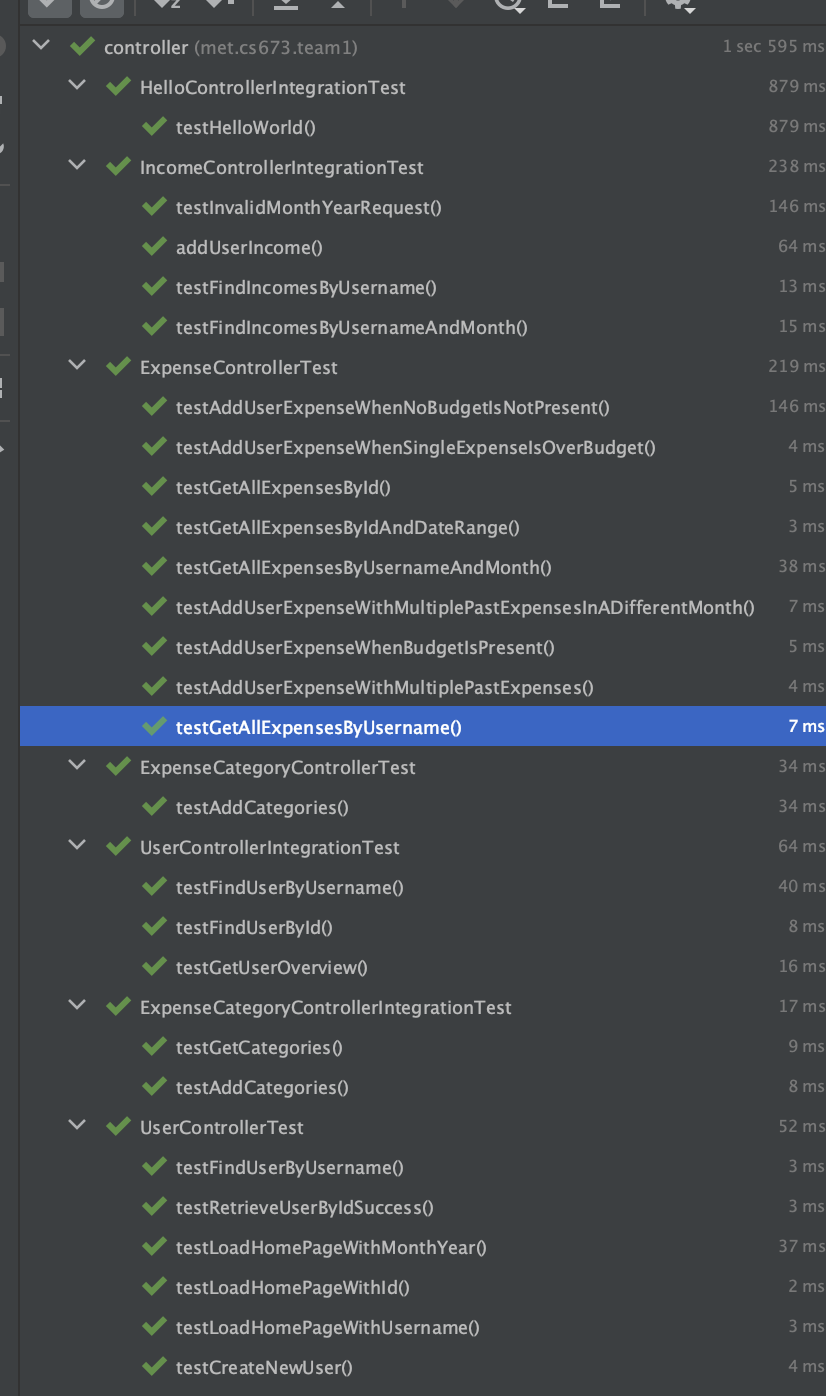
**Testing Reports:**

Mapper Tests (16/16 ✅)

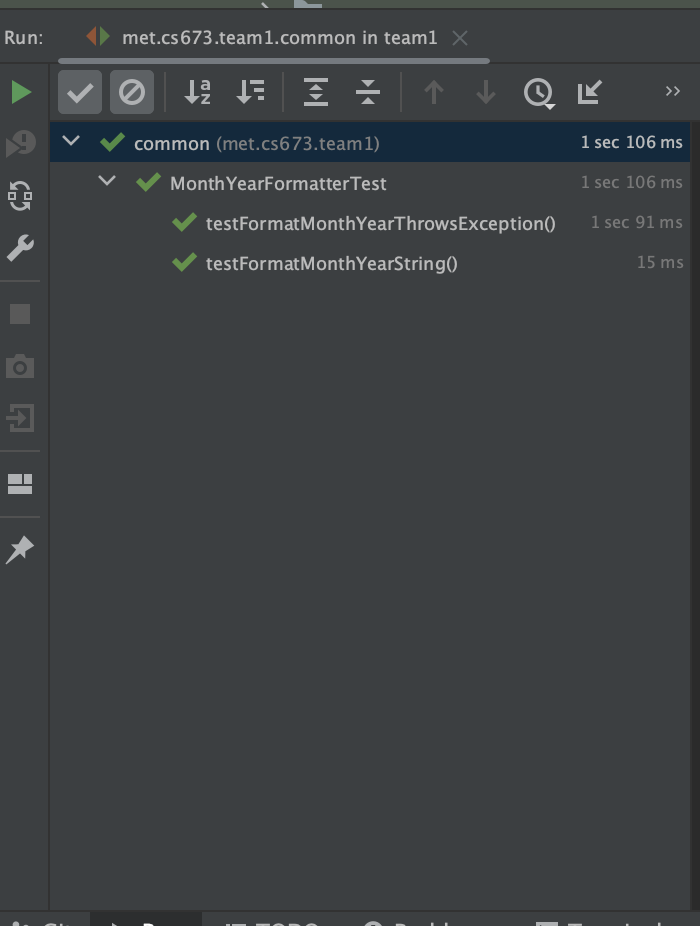


Service Tests (31/31 ✅)

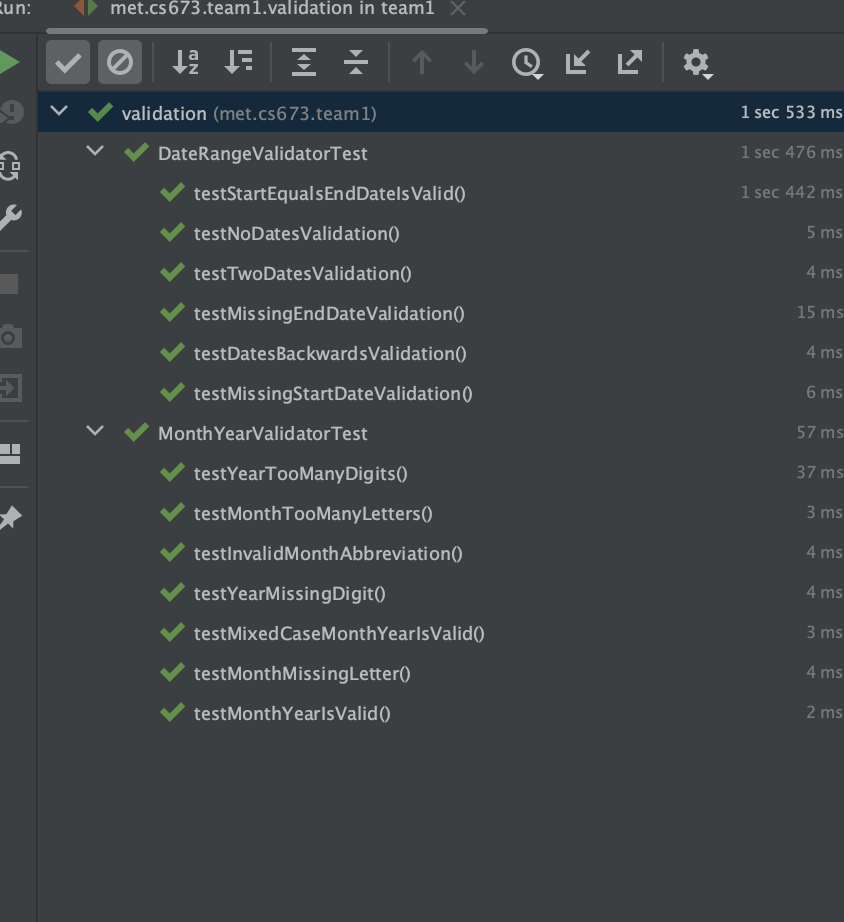


Controller Tests (35/35 ✅ )

Common Tests (2/2 ✅ )



Validation Tests (13/13 ✅)

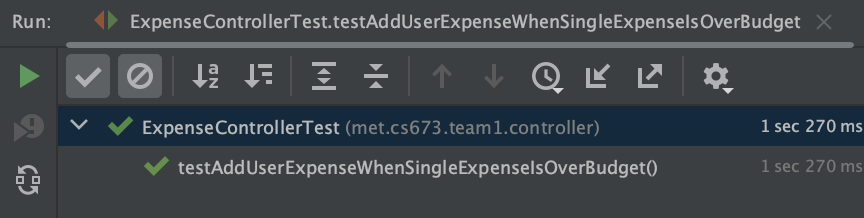


**Testing Examples (Unit Tests):**

Here is an example of a Unit test, that tests the “/addExpense” endpoint of the Expense Controller.

In this test, a mock expenseDto is created and the expenseService save method is mocked to return the mock expenseDto. A mock User is built with a budget of 100.00 and the userService findByUsername method is mocked to return the mock User. The Expense Controller’s addUser method is called with the mock expenseDto.

The test then checks that the response Expense Dto has a isOverBudget property that is indeed “true”. 

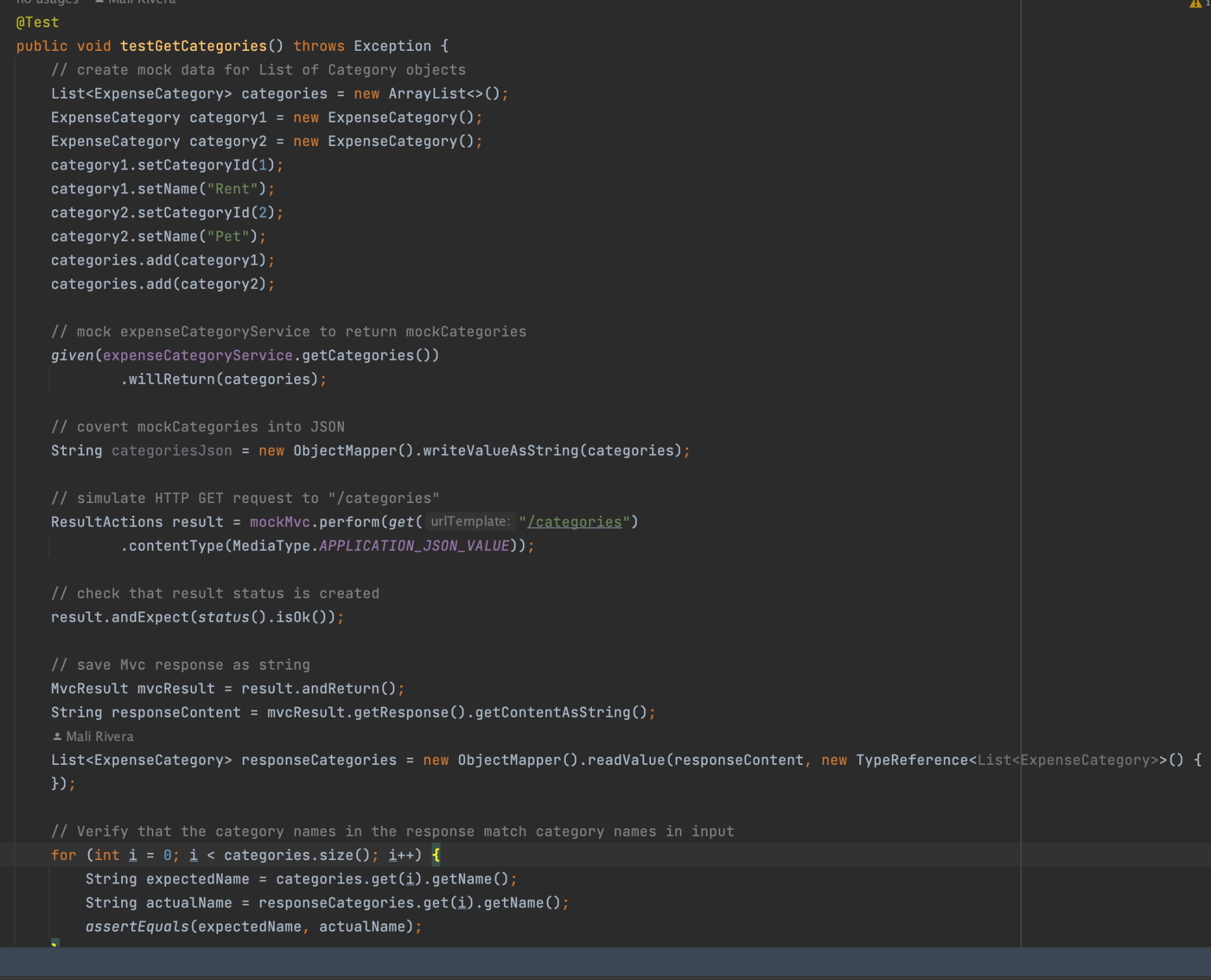


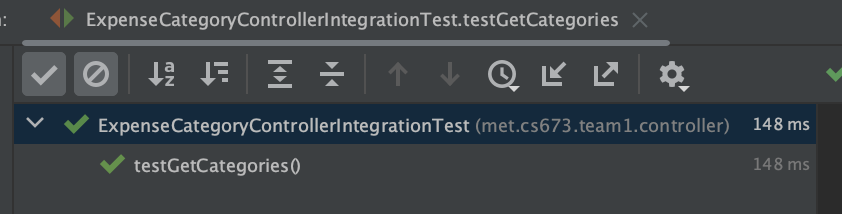
**Testing Examples (Integration Tests):**

Here is an example of an Integration test, that tests the “/categories” endpoint of the Expense Category Controller.

In this test, a mock List of categories is created. The expenseCategoryService getCategories method is mocked to return the mock categories. The categories List is converted into JSON, a GET request is simulated to “/categories”, and the response is saved.

The test then loops over the response list and asserts that the category names in the response are equal to the category names in the input list.



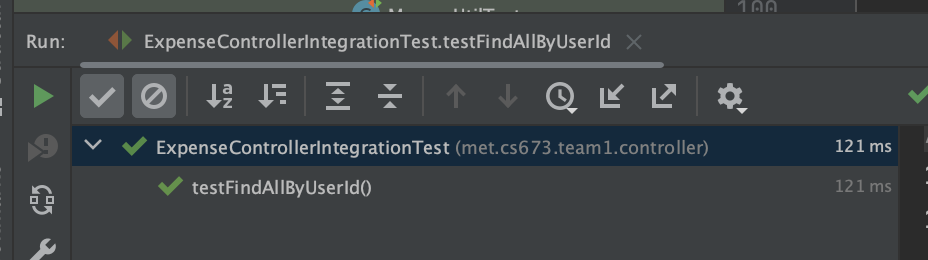


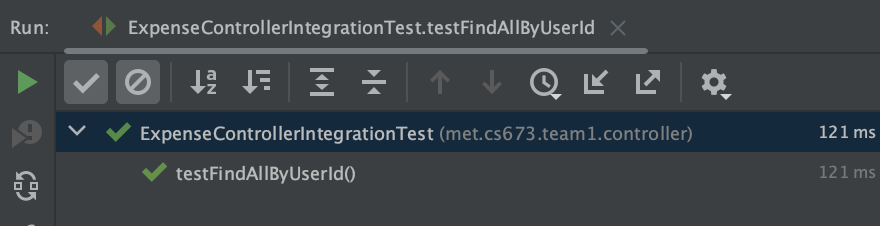
Here is an example of an Integration test, that tests the “/expenses/{userId}” endpoint of the Expense Controller.

In this test, a mock list of expenses and a mock userId are created. The expenseService findAllByUserId method is mocked to return the mock expenses. A GET request to “/expenses/{userId}” is simulated with userId, startDate, and endDate as parameters.

The test then asserts that the response is 200, Ok.







# Testing Metrics (Mali)

In this section, you shall report any metrics used for the evaluation, e.g. # of test cases, test coverage, defects rate, etc.

Testing Metrics Definitions

| Metric Name | Description |
| --- | --- |
| Test Count | The number of test cases |
| Test Coverage | The percentage of code or functionalities covered by test cases. |
| Defect Density (KLOC) | The number of defects per thousand lines of code. |
| Execution Rate | The number of test cases executed in a given time frame. |
| Test Case Effectiveness | The number of defects measured by test cases. |
| Test Case Pass Rate | The percentage of test cases passed by a testing cycle. |
| Defect Escape Rate | The number of defects found post-production compared to the number of defects identified by QA |
| Test Automation Coverage | The percentage of tests that have been automated |

Testing Metrics Report

| **Testing Metrics Summary (Automated Tests)** | | | | | |
| --- | --- | --- | --- | --- | --- |
| **Metric** | **Classes** | | | | |
|  | Controllers (5) | Services (5) | Mappers (4) | Validator (2) | Common (1) |
| Test Count (tests) | 35 | 31 | 16 | 13 | 2 |
| Test Coverage (tests/classes) | 100% | 100% | 100% | 100% | 100% |
| Avg Execution Rate (ms) | 49.12 | 221.71 | 16.06 | 85.08 | 719.5 |
| Execution Rate (ms) | 1719 | 6873 | 257 | 1106 | 1439 |
| Test Case Effectiveness (defects/class) | 0 | 0 | 0 | 0 | 0 |
| Test Case Pass Rate | 100% | 100% | 100% | 100% | 100% |
| Defect Escape Rate | N/A | N/A | N/A | N/A | N/A |
| Test Automation Coverage | 100% | 100% | 100% | 100% | 100% |

# References

* Cucumber (<https://cucumber.io/>)
* Cucumber set-up in IntelliJ (<https://www.jetbrains.com/help/idea/enabling-cucumber-support-in-project.html#add-cucumber-library> )

# Glossary