**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** |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

[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):

* For unit testing, we will use 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>

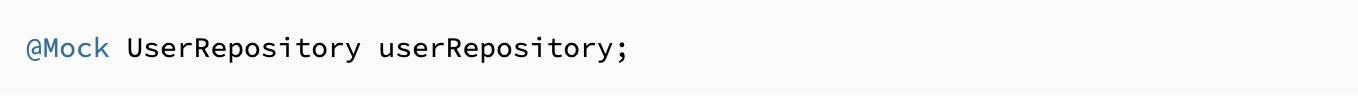
<dependency>

<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):

* For integration testing, we will use 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/>

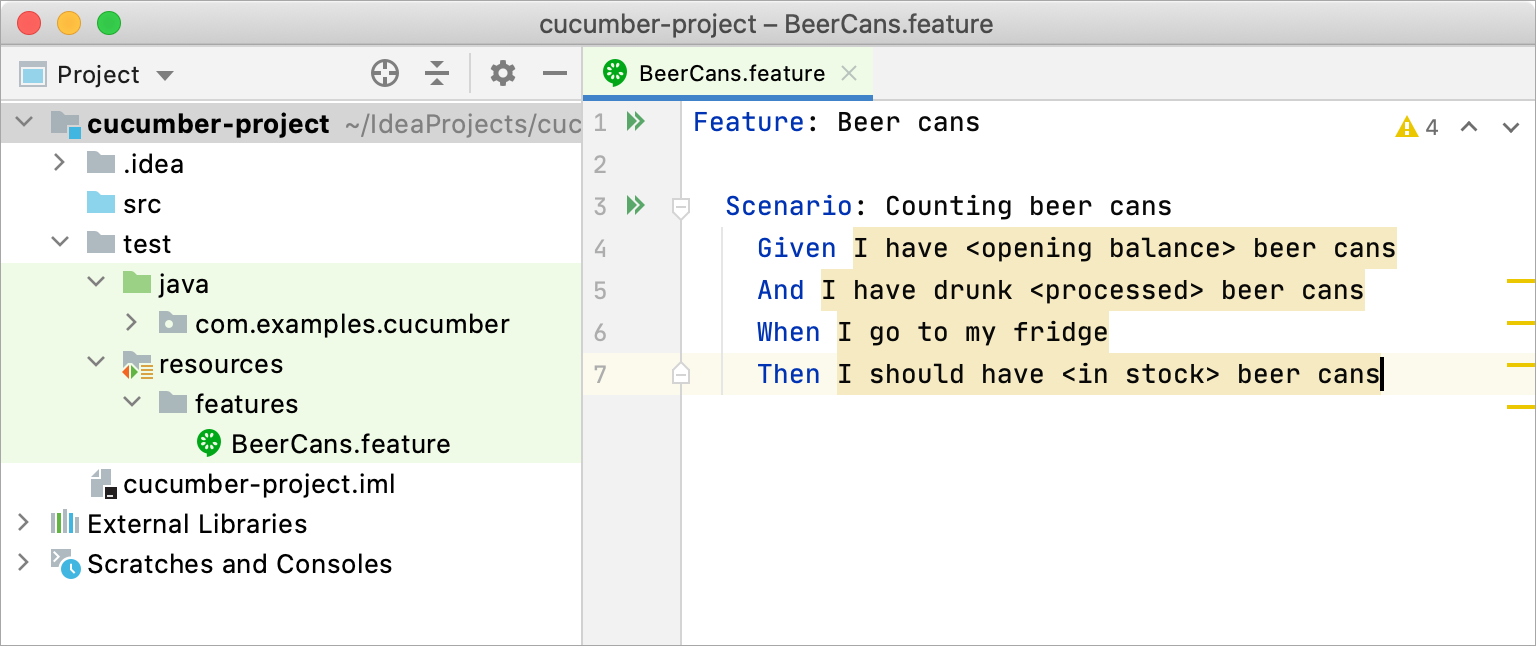
<https://howtodoinjava.com/spring-boot2/testing/spring-boot-mockmvc-example/>

Acceptance Test: (Jisoo / Chaitanya)

* jUnit5 and Cucumber

For acceptance testing, we will use Cucumber which 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 iteration 1, we implemented three tests for our code. Since the main focus was getting the database up and running for this iteration, code was only recently added to Github for testing purposes. The focus of the next iteration will be adding unit and integration tests to the existing code and acceptance tests once there is engine code to be tested.

Unit tests:

Test case ID, name: Test0, testFindByIdSuccess

New or old: New

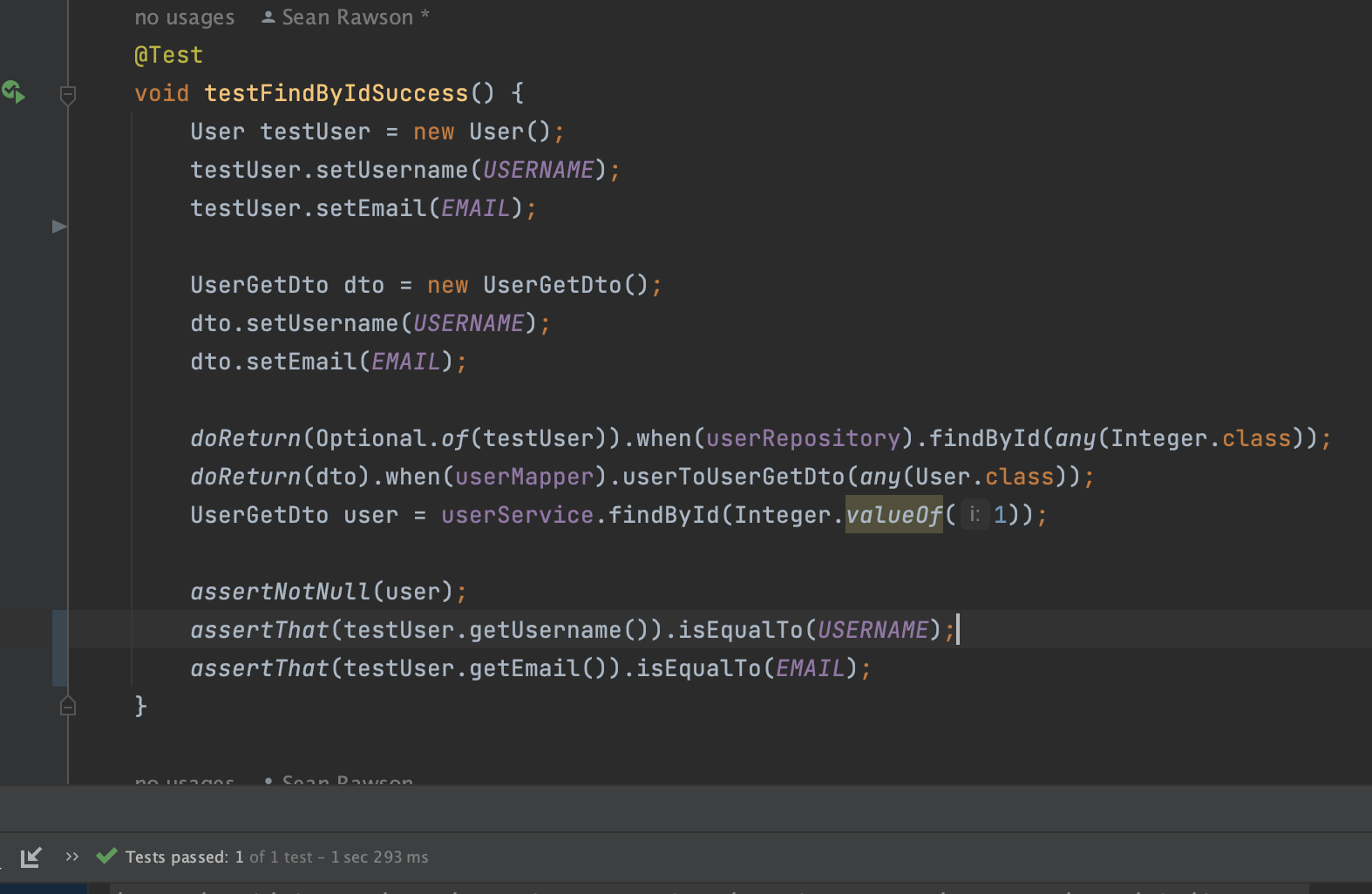
Test items: userService

Test priority: 1

Description: unit test with mockito and JUnit5 to test if the userService can return user by userId

Expected result: return correct username and email

Actual output: returned correct username and email



Test case ID, name: Test1,testFindByIdThrowsException

New or old: New

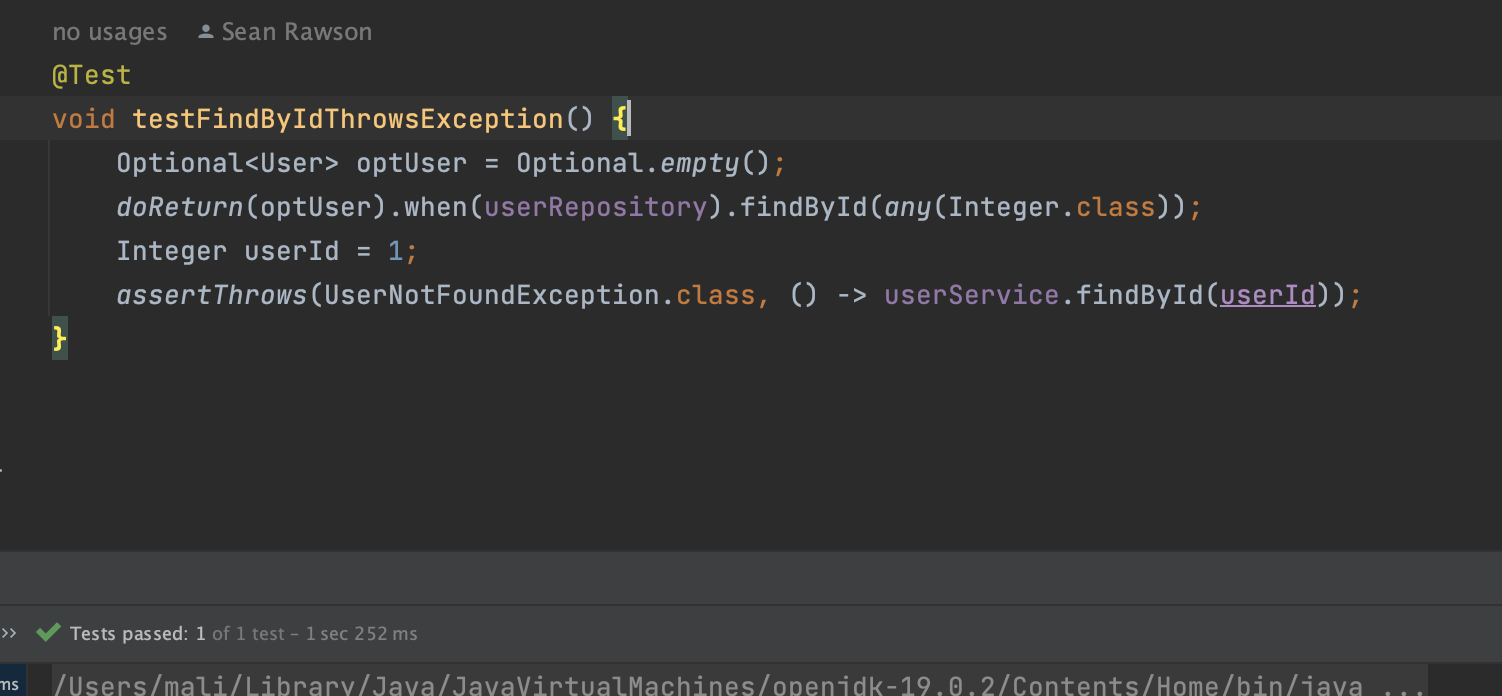
Test items: userService

Test priority: 1

Description: unit test with mockito and JUnit5 to test if the userService throws error when it cannot find user by userId

Expected result: UserNotFoundException

Actual output: UserNotFoundException



Integration test:

Test case ID, name: Test3, testExpenseController\_addUserExpense

New or old: New

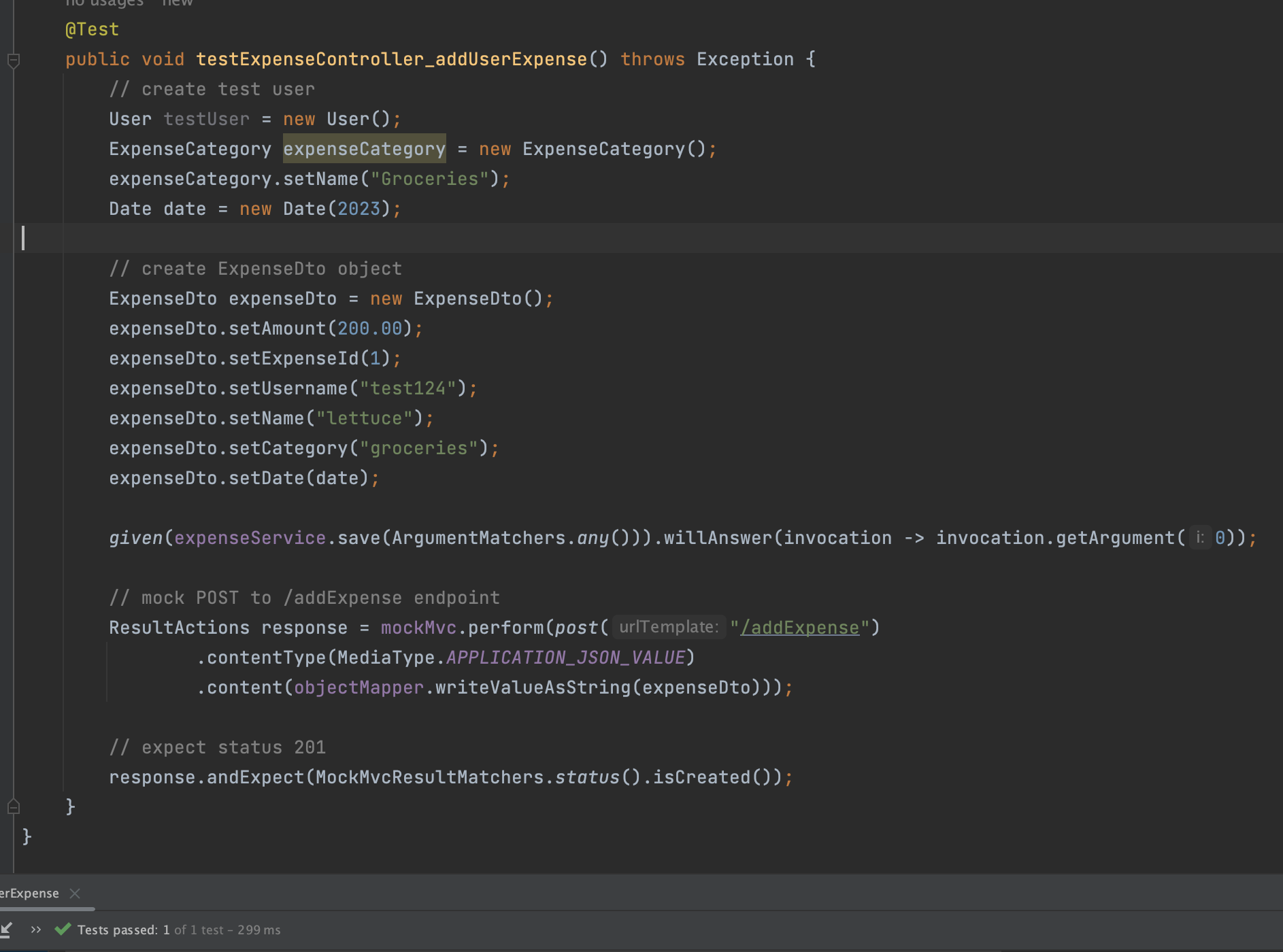
Test items: ExpenseController

Description: integration test with mockMVC and JUnit5 to test if the ExpenseConroller can use ExpenseService to save an ExpenseDto object

Test priority: 1

Expected result: status 201

Actual output: status 201



# 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 iteration 1, we have not incorporated automated testing. We plan to use JUnit5 for automated testing once there is code in our repository to test.

# 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.

| 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 |

Test count: 3

Test Coverage: ⅓ controller classes, ¼ service classes

Defect Density: 0

Execution Avg: 329 ms

Execution Rate: 100% under 1s

Test Case Effectiveness: 0

Test Case Pass Rate: 100%

Defect Escape Rate: N/A

Test Automation Coverage: 0%

# 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