Backend Testing Project Report

Mutation Testing with PIT

Aadil Mohammad Husain

Roll No: MT2024001

Mohammed Mubashir Khan

Roll No: MT2024089

Link:
https://github.com/AadilMdhusain/Mutation-Testing-Project

International Institute of Information Technology, Bangalore

Executive Summary

This project presents a comprehensive testing framework for an Order Management System developed in Java. The primary objective is to evaluate the effectiveness of mutation testing in assessing the quality and robustness of the test suite. The system consists of real-world business logic for handling orders, bookings, and related operations. All functionalities are thoroughly validated using JUnit 5 for both unit and integration testing, while PIT (Pitest) mutation testing is applied to measure how well the tests detect injected faults at the statement and integration level. This approach ensures high reliability of the system and demonstrates strong software testing and quality assurance practices.

Key Achievement: Successfully implemented a robust testing framework that combines unit testing with mutation testing to achieve high code quality and test coverage. The project demonstrates best practices in software testing and quality assurance.

Project Objectives

- **Feature Implementation:** Develop clean, modular, and efficient components for an Order Management / Booking System using Java and Spring Boot.
- **Comprehensive Testing:** Write complete unit and integration test coverage using JUnit 5, ensuring that all service, controller, and repository layers behave as expected.
- **Integration Testing:** Validate end-to-end workflows such as order creation, booking confirmation, payment handling, and data persistence across system modules.
- **Mutation Testing:** Use PIT (Pitest) to measure test effectiveness, identify weak assertions, and expose untested or under-tested logic at both statement and integration levels.
- **Code Quality:** Maintain high code quality through proper layering, clean architecture, meaningful abstractions, and well-documented code.
- **Best Practices:** Demonstrate industry-standard testing practices including dependency injection, mock-based testing, layered test design, and continuous quality assessment via mutation testing.

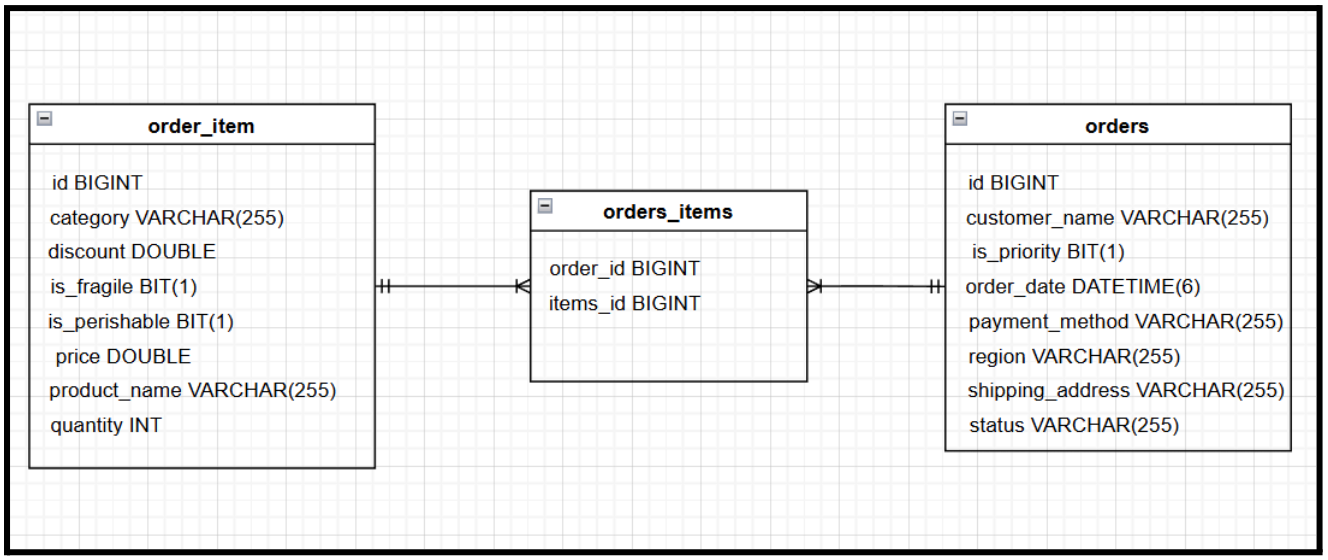
Tools & Technologies

- **Java & Spring Boot**
  The primary backend technology used for building the Order Management System, providing a robust framework for developing scalable business logic, REST APIs, and database integrations.
- **JUnit 5**
  A modern and powerful testing framework used for writing unit and integration tests. It offers annotations, assertions, and extensions that simplify structured and automated testing.
- **PIT Mutation Testing (Pitest Maven Plugin)**
  A mutation testing tool that injects small code modifications (mutations) to determine how effectively the existing test suite detects faulty behaviour. It provides detailed reports on test strength and quality.
- **Maven**
  A build automation and dependency management tool used to configure test plugins, manage project dependencies, and run mutation testing through defined goals.
- **GitHub**
  A version control and collaboration platform used to manage the codebase, track changes, and support team development through branches, pull requests, and repositories.

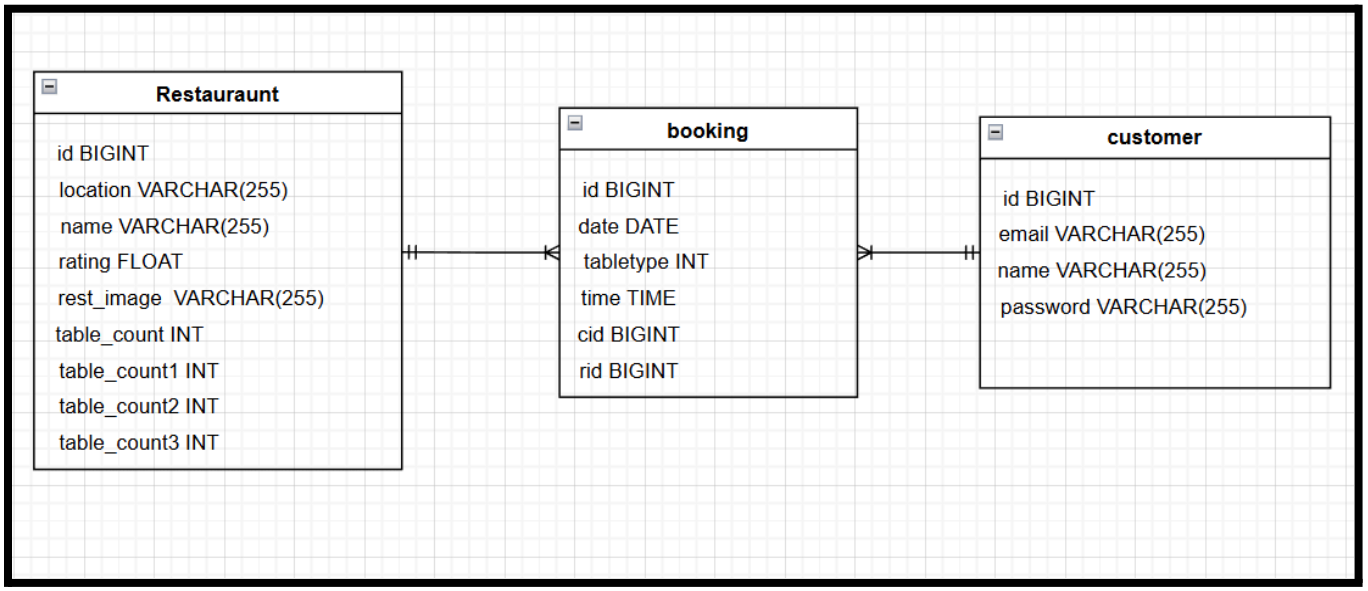
Backend Implementation-

Schema of the Database-

The schema of the Database we were working with are as follows-



The database consists of three tables—**orders**, **order_item**, and a junction table **orders_items**—designed to model a many-to-many relationship between orders and items. The **orders** table stores customer-level information such as the customer's name, priority flag, order date, payment method, region, shipping address, and order status, with each record representing a single order. The **order_item** table describes individual products that can appear in orders, containing details like product name, category, price, discount, quantity, and attributes such as whether the item is fragile or perishable. Since one order can contain multiple items, and the same item can appear in multiple different orders, the **orders_items** table serves as a bridge linking them using two foreign keys—`order_id` and `items_id`.



The database models a restaurant reservation system using three main tables: **Restaurant**, **Customer**, and **Booking**. The **Restaurant** table stores details about each restaurant, including its name, location, rating, image, and the number of available tables of different types. The **Customer** table holds user information such as email, name, and password, with each record representing a registered customer. The **Booking** table serves as the link between customers and restaurants, recording reservation details such as the booking date, time, table type selected, and foreign keys referencing the customer (`cid`) and restaurant (`rid`) involved. This structure allows each customer to make multiple bookings and each restaurant to receive many bookings, enabling a flexible and well-organized reservation management system.

Layers of Backend-

There are four layers defined in the Backend-

Controller Layer-

Controller Layer serves as the primary point of Interaction with the client. Its main responsibilities include, Handling Incoming Requests, Request Mapping, Parameter Validation, Delegation to Service Layer, Response Generation Error Handling, etc. It acts as an intermediary between translating client requests to calls to the application's business logic and then translating the call back into client-friendly response. We have defined four controllers in our project-

1. CustomerController.java - This controller handles customer-related actions such as fetching customer info, getting customer ID, viewing bookings, and deleting a booking. It connects to `CustomerService`, `BookingService`, and `BookingRepository` to perform its operations. It allows retrieving a customer by ID, finding a customer ID using email, and listing all bookings made by a customer. It also provides an endpoint to delete a booking by its booking ID.
2. LoginController.java - This controller manages user authentication. It has two endpoints: one for logging in and another for customer registration. The login endpoint forwards the user's login credentials to `CustomerService.verifyLogin()`, while registration saves a new customer using `CustomerService.saveNewCustomer()`. This controller only deals with authentication logic, not booking or restaurant operations.
3. OrderController.java - This controller handles order processing logic (unrelated to restaurants/bookings). It validates orders, calculates the total price of an order, and computes shipping cost based on region. It uses the `OrderProcessor` service to perform this logic. The `/calculate` endpoint checks if the order is valid and returns the computed total. The `/validate` endpoint validates an order only. The `/shipping-cost` endpoint returns the shipping cost for a given region. It also includes several small response wrapper classes to send structured JSON responses.
4. RestaurantController.java - This controller handles everything related to restaurants and booking tables. It can fetch restaurants with available tables, retrieve all booked tables for a specific date/time, and allow customers to book a table. It uses

`RestraurantService`, `BookingService`, and `BookingRepository` for its logic. It also contains a file-download endpoint to return restaurant images stored on the server.

The `/book-apt` endpoint receives booking details in the request body, sets the table type, and saves the booking in the database.

Service Layer-

Service Layer is responsible for encapsulating and executing the core business logic of the application. It acts as an intermediary between the presentation layer (e.g., controllers) and the data access layers (e.g., controllers). Its main function includes executing business logic, transaction management, data orchestration, abstraction of data access and enforcing business logic.

1. OrderProcessor Service - The **OrderProcessor** service is responsible for all business logic related to processing an order. It calculates the final total price by first computing the price of each item—including category-based discounts, quantity discounts, and item-specific discounts—and then applies order-level discounts such as weekend, priority-customer, large-order, and region-based discounts. It also computes shipping costs using weight, fragile-item surcharges, distance-based charges, and priority shipping fees. Additionally, this service provides an `validateOrder()` method to ensure all required fields are present and all items meet required constraints such as valid price, nonzero quantity, and correct discount values. Overall, this service encapsulates **all pricing, discount, validation, and shipping rules** for an order.

2. Restaurant Service - The **RestaurantService** interface defines two main operations: retrieving restaurants that have available tables and loading restaurant images stored on disk. Its implementation, **RestaurantServiceImpl**, queries the database to return restaurants with table counts greater than zero and loads image files from the server's filesystem based on the configured path. It converts the requested image into a Spring `Resource` object so the controller can return it to the frontend. This service cleanly separates **restaurant retrieval logic** and **file/image loading logic** away from the controller.

3. Customer Service - The **CustomerService** interface defines methods for registering new customers, verifying login details, fetching customer data by ID, and retrieving a customer by email. Its implementation, **CustomerServiceImpl**, interacts with the database using `CustomerRepository`. It handles user registration by checking if the email already exists, inserts a new customer when appropriate, and validates login by checking both email and password. It also supports searching for a customer via ID or email for use in bookings and other features. This service centralizes **all customer-related business logic**, keeping it separate from the controllers and database layer.

Model Layer -

The **Model Layer** is the part of the application responsible for defining the structure of the data your system works with. In this layer, you create classes—often called entities—that directly represent the tables in your database. These model classes contain fields for each column, along with annotations that map them to the database, and they also define the relationships between tables, such as one-to-many or many-to-many associations. The model layer does not contain business logic; its job is simply to describe what the data looks like, how different pieces of data are related, and how they are stored or retrieved from the database. In short, the model layer acts as the blueprint for the application's data and ensures smooth communication between the service layer and the database.

Repository Layer-

The **Repository Layer** is responsible for directly interacting with the database and performing all data access operations such as saving, updating, deleting, and retrieving records. In this layer, you define interfaces that extend Spring Data JPA repositories like `JpaRepository` or `CrudRepository`, allowing the framework to automatically generate SQL queries for you. The repository layer hides all database-related complexity from the rest of the application, making data operations simple, reusable, and clean. It acts as the bridge between the **Model Layer** and the **Service Layer**, ensuring that the service layer gets data in an organized way without ever needing to handle raw SQL or database logic.

Unit Testing-

In unit testing, we test individual pieces of code—usually methods or small components—in isolation to make sure they work exactly as expected. The goal is to verify that each unit of the application behaves correctly with both valid and invalid inputs, without involving external systems like databases or APIs. We typically use testing frameworks such as **JUnit** and **Mockito** to write tests that check the logic, edge cases, error handling, and outputs of a method. By doing unit testing, we can catch bugs early, ensure that new changes don't break existing functionality, and maintain overall code quality and reliability.

We implemented the following Unit Tests-

1. BookingServiceImplTest - Here we test the BookService Implementation. We use this to verify the implementation of the booking service. This tests-
    a. testSaveCallHistory() -
    b. testGetCallHistoryForDoctor_CustomerExists()
    c. testGetCallHistoryForDoctor_CustomerNotExists()
    d. testDeleteAppointment_BookingExists()
    e. testDeleteAppointment_BookingNotExists()
2. CustomerServiceImplTest - These tests validate customer registration, login, and retrieval.
    a. saveNewCustomer()
    b. verifyLogin()
    c. getCustomerFromDb()
    d. findByEmail()
3. OrderProcessorTest - These tests validate the OrderProcess Calculations.
    a. calculateTotalPrice()
    b. calculateBookTotal()
    c. validateOrder(order)
    d. validateOrderItem()
    e. Distance Cost

Integration Testing-

Integration Testing is a software testing level where individual modules or components are combined and tested as a group to ensure they work correctly together. While unit testing focuses on verifying each piece of code in isolation (like a single class or function), integration testing checks how those pieces interact when they are connected.

We implemented the following Integration Tests-

OrderControllerIntegrationTest - We are performing full end-to-end integration testing at the controller layer. We are testing the interaction between OrderController,

OrderProcessor, SpringBoot actual dependency injection, real HTTP response creation and custom response object.

- testValidOrderCalculation() - This test calls the `/calculateOrderTotal` method on the real `OrderController` and checks that the HTTP response is successful, that the controller internally uses the actual `OrderProcessor` to perform the calculation, and that the total returned by the endpoint matches the value produced by the real service.
- testOrderValidation() - This test verifies that when a valid order is passed to the controller's validation endpoint, the controller correctly delegates to the service, recognizes the order as valid, and returns a successful validation response.
- testInvalidOrderCalculation() - This test checks that when an invalid order with missing required fields is submitted for total calculation, the controller identifies the validation failure, returns a 4xx error response, and provides the correct error message in the response body.
- testShippingCostCalculation() - This test ensures that the controller correctly computes shipping cost for different regions by calling the real service and returning a successful response containing the expected region-based cost.
- testWeekendDiscountScenario() - This test confirms that when an order falls on a weekend date, the controller correctly applies the weekend discount through the service and returns a total that reflects the discounted calculation.
- testOrderControllerConstructor() - This test verifies that Spring successfully constructs the `OrderController` and injects a real `OrderProcessor` into it, confirming proper dependency wiring through reflection-based checking.
- testOrderCalculationErrorScenarios() - This test validates that when the service throws an exception due to an improperly structured order, the controller catches it and responds with a structured 4xx error containing an appropriate error message.
- testErrorResponseConstructor() - This test simply confirms that the custom `ErrorResponse` object initializes correctly by storing and returning the provided error message.

Mutation Test -

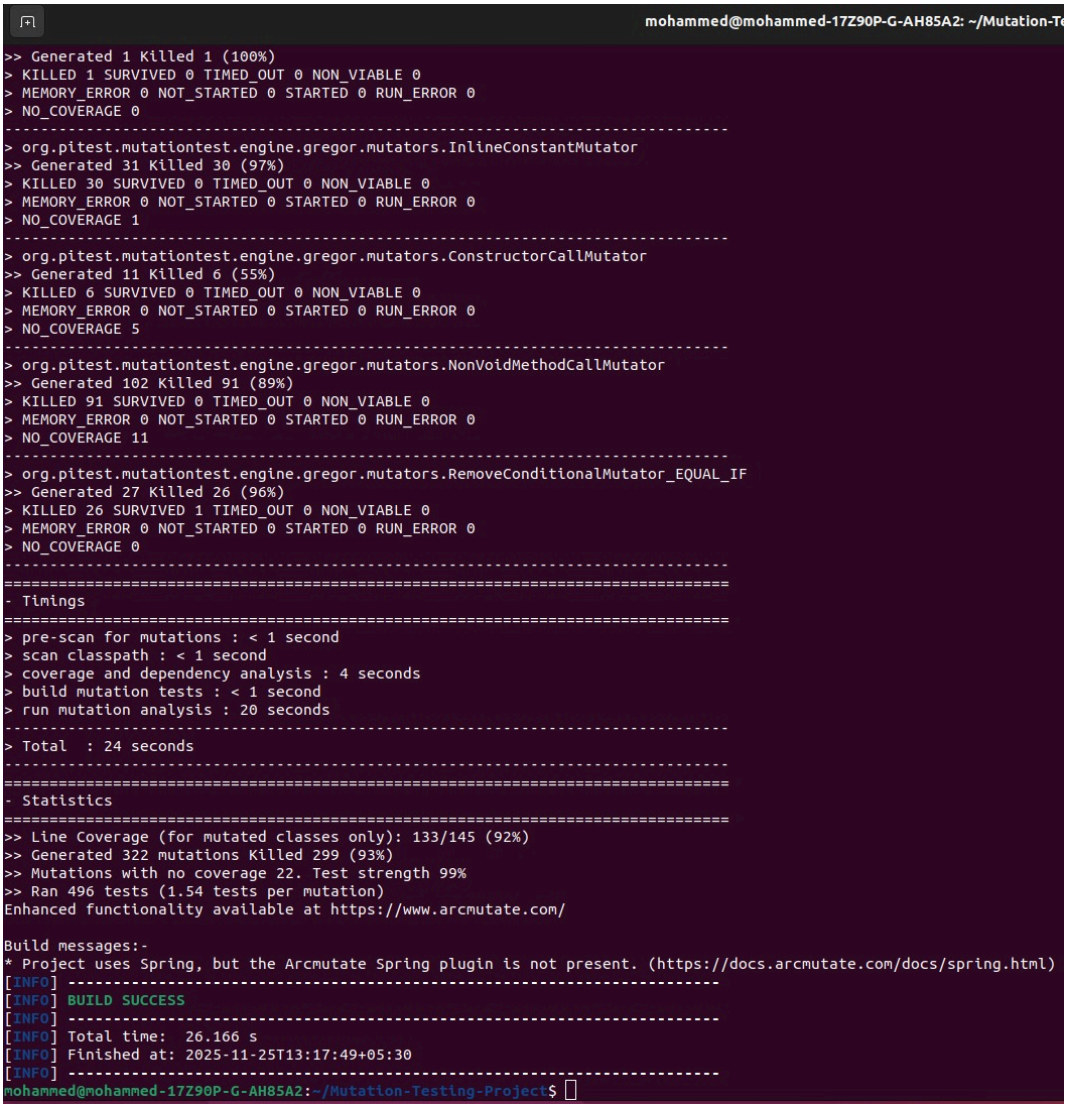
To evaluate the quality of my unit tests, I applied **Mutation Testing** using **PIT (Pitest)**, a widely used mutation testing framework for Java. PIT works by automatically introducing small changes—called *mutations*—into my application code, such as altering operators, removing method calls, or modifying conditional logic. After each mutation, PIT re-runs the entire test suite to check whether the tests detect the change and fail as expected. If a test fails, the mutation is "killed," indicating strong test coverage; if a test passes despite the mutation, it "survives," revealing a gap in the test effectiveness.

In my project, I integrated PIT into the build process using Maven and executed it with the `mvn pitest:mutationCoverage` command. PIT generated a detailed HTML report showing killed, survived, and no-coverage mutations, which helped me identify weak spots in the test suite and improve my assertions and edge-case handling. By using PIT, I ensured that my tests were not only covering the code but were also capable of catching unintended behavior, resulting in more robust and reliable test cases.

Mutation Operators Used-

```
lugin>
  <version>1.17.1</version>
  <configuration>
      <targetClasses>
          <param>com.example.spebackend.service.BookingServiceImpl</param>
          <param>com.example.spebackend.service.OrderProcessor</param>
          <param>com.example.spebackend.service.CustomerServiceImpl</param>
          <param>com.example.spebackend.controller.OrderController</param>
      </targetClasses>
      <targetTests>
          <param>com.example.spebackend.service.BookingServiceImplTest</param>
          <param>com.example.spebackend.service.OrderProcessorTest</param>
          <param>com.example.spebackend.service.CustomerServiceImplTest</param>
          <param>com.example.spebackend.controller.OrderControllerIntegrationTest</param>
      </targetTests>
      <mutators>
          <!-- Conditional Mutators-->
          <mutator>CONDITIONALS_BOUNDARY</mutator>
          <mutator>NEGATE_CONDITIONALS</mutator>
          <mutator>REMOVE_CONDITIONALS_EQUAL_ELSE</mutator>
          <mutator>REMOVE_CONDITIONALS_EQUAL_IF</mutator>
          <mutator>REMOVE_CONDITIONALS_ORDER_ELSE</mutator>
          <mutator>REMOVE_CONDITIONALS_ORDER_IF</mutator>

          <!-- Method Calls -->
          <mutator>VOID_METHOD_CALLS</mutator>
          <mutator>NON_VOID_METHOD_CALLS</mutator>
          <mutator>CONSTRUCTOR_CALLS</mutator>

          <!-- Return Values -->
          <mutator>EMPTY_RETURNS</mutator>
          <mutator>FALSE_RETURNS</mutator>
          <mutator>TRUE_RETURNS</mutator>
          <mutator>NULL_RETURNS</mutator>
          <mutator>PRIMITIVE_RETURNS</mutator>

          <!-- Arithmetic and Numbers -->
          <mutator>INCREMENTS</mutator>
          <mutator>REMOVE_INCREMENTS</mutator>
          <mutator>INVERT_NEGS</mutator>
          <mutator>MATH</mutator>

          <!-- Constants -->
          <mutator>INLINE_CONSTS</mutator>
```

The result of Mutation Operators-



Inference from Mutation Testing-

The result we got from our Mutation Testing are as follows-

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
4	92%	141/154	93%	339/364	99%	339/340

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.example.spebackend.controller	1	52%	13/25	55%	28/51	97%	28/29
com.example.spebackend.service	3	99%	128/129	99%	311/313	100%	311/311

- Project uses Spring, but the Arcmutate Spring plugin is not present.

Report generated by PIT 1.17.1

Enhanced functionality available at arcmutate.com

You can read the output and result of each individual mutation operator and check whether or not the given mutant was killed.

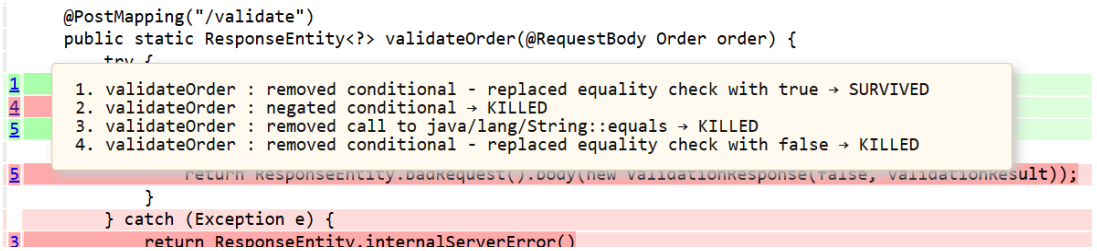
CustomerServiceImpl.java

```
1  package com.example.spebackend.service;
2
3  import com.example.spebackend.model.Customer;
4  import com.example.spebackend.repository.CustomerRepository;
5  import org.springframework.stereotype.Service;
6
7  import java.util.Optional;
8
9  @Service
10 public class CustomerServiceImpl implements CustomerService {
11     public final CustomerRepository customerRepository;
12
13
14     public CustomerServiceImpl(CustomerRepository customerRepository) {
15 1       this.customerRepository = customerRepository;
16     }
17
18     @Override
19     public String saveNewCustomer(Customer customer){
20 4       Customer customer1 = customerRepository.findByEmail(customer.getEmail()).orElse(null);
21 3       if(customer1 == null) {
22 1           customer1 = new Customer();
23 2           customer1.setPassword(customer.getPassword());
24 2           customer1.setEmail(customer.getEmail());
25 2           customer1.setName(customer.getName());
26 2           customerRepository.save(customer1);
27 1           return ("User Registration was Successful!!");
28          }
29          else
30 1           return ("User Email ID already exist");
31     }
```

What you can infer from the Mutation Testing as follows-

- Since no Mutant survived for some service application we can infer that our path is highly deterministic and our unit test case covers all scenarios.
- The mutation testing results for the `OrderController` show that most of the essential business logic and controller–service interactions are **well-protected** by your integration tests.
- The mutation testing results for Service layer shows us very optimistic results and indicates that the functioning of these are very determinate in nature.
- The **NO_COVERAGE mutations** reveal areas where the tests never executed certain branches of the controller, mainly the deeper error-handling paths such as unexpected internal errors, fallback return values, and exceptional response handling (`internalServerError()`, alternate `badRequest()` branches, and secondary constructors in error paths).

- Most Mutants that have survived are most likely equivalent mutants or mutants which are equivalent to original program in terms of execution. For examples-



- The mutant is produced where we replace equality check with true and since most of the time the response we get is valid, the output doesn't differ.