Server-Side Testing

26 Nov. 2024
CSE 731 Software Testing

Instructor
Prof. Meenakshi D Souza

Submitted By:
Subhodip Rudra MT2023103
Rishav Chandel MT2023058

Introduction to Server-Side Web Application Testing:

Server-side web applications testing is a critical aspect of ensuring the reliability, security, and performance of web applications that run on server-side infrastructures. This type of testing involves verifying the functionality, scalability, and robustness of the application components that operate on the server rather than the client side. Server-side testing focuses on validating the core logic, APIs, database interactions, and back-end services that form the foundation of a web application. It ensures that these components handle requests correctly, process data securely, and deliver accurate responses to client-side users.

Key elements of server-side web applications testing include functional testing of APIs, load and stress testing for scalability, database testing for data integrity, and security testing to identify vulnerabilities in the server environment. By thoroughly testing the server-side components, developers and testers can detect and address potential issues early, ensuring a seamless and secure user experience.

Need of Server-Side Web Application Testing:

Traditional testing approaches, such as UI or client-side testing, often focus solely on verifying user interactions and front-end behavior. However, they fail to comprehensively assess the robustness and reliability of the server-side components, which are critical for handling core application logic, data processing, and business operations.

Server-side web applications testing ensures that the back-end systems function as intended, even under complex and high-stress scenarios. This testing is essential to:

- **Verify Core Functionality:** Ensure that APIs, database queries, and server logic perform accurately and consistently.
- **Ensure Data Integrity:** Validate that server-side processes correctly handle data storage, retrieval, and manipulation.
- **Achieve Reliability:** Guarantee seamless operation during simultaneous user interactions or edge-case scenarios.

Without comprehensive server-side testing, even well-designed client-side applications can fail to deliver reliable and secure services to end-users.

Tools Used:

- **JUnit (Unit Testing):**
  - Verifies the correctness of individual server-side methods and functions.
  - Supports integration with Maven for automated testing workflows.
- **Mockito (Mocking):**
  - Simulates external dependencies like APIs or databases for isolated testing.
  - Simplifies unit testing by creating mock objects for server-side components.

- **JMeter (Performance Testing):**
  - Simulates high user traffic to test server-side performance under load.
  - Identifies bottlenecks and ensures scalability of server-side components.
- **IntelliJ IDEA (IDE):**
  - Offers integrated testing, debugging, and refactoring for server-side code.
  - Seamlessly integrates with tools like JUnit, Mockito, and JaCoCo.
- **JArchitect (Code Architecture Analysis):**
  - Analyzes server-side code structure for complexity and dependency issues.
  - Ensures adherence to architectural principles for maintainable designs.
- **JaCoCo (Code Coverage):**
  - Measures the percentage of server-side code covered by test cases.
  - Identifies untested critical paths to improve test coverage.

Objective of Server-Side Web Application Testing

The primary objective of **Server-Side Web Application Testing** is to ensure that the server-side components of a web application (such as APIs, databases, and server logic) function correctly, securely, and efficiently.
Key goals include:

- **Functional Testing**:
  - Verifying that the backend logic and database interactions work as intended.
  - Ensuring that APIs handle inputs and responses correctly under different scenarios.

Server-side testing helps ensure the reliability and security of the backend, contributing to the overall performance and trustworthiness of the web application.

Levels of Server-Side Web Application Testing

- **Unit Testing:**
  Tests individual components like methods or functions in isolation to ensure they work correctly.
- **Integration Testing:**
  Verifies interactions between server-side components, ensuring data flows correctly between them.
- **Performance Testing:**
  Assesses how well the server-side application handles high traffic and varying loads.
- **Acceptance Testing:**
  Confirms the server-side application meets business requirements and is ready for deployment.

Source Code:

The backend server of this codebase is designed to handle various functionalities related to an exam portal. It provides endpoints for user registration, login, quiz management, and results tracking. The system leverages Spring Boot for creating RESTful APIs and uses a MySQL database to store user information, quizzes, categories, and question data. The backend is structured with separate layers for controllers, services, and repositories, adhering to the MVC architecture. It supports role-based authentication and authorization, ensuring secure access to resources. The application is built to scale and perform efficiently with proper error handling and input validation.

Test Coverage:

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %
all classes	100% (24/24)	100% (149/149)	88.2% (30/34)	100% (292/292)

Coverage Breakdown

Package	Class, %	Method, %	Branch, %	Line, %
com.project.examportalbackend	100% (1/1)	100% (4/4)		100% (5/5)
com.project.examportalbackend.configurations	100% (3/3)	100% (16/16)	83.3% (10/12)	100% (65/65)
com.project.examportalbackend.controllers	100% (5/5)	100% (27/27)	92.9% (13/14)	100% (93/93)
com.project.examportalbackend.models	100% (9/9)	100% (72/72)		100% (78/78)
com.project.examportalbackend.services.implementation	100% (6/6)	100% (30/30)	87.5% (7/8)	100% (51/51)

generated on 2024-11-26 13:57

Activity Transition Graph:

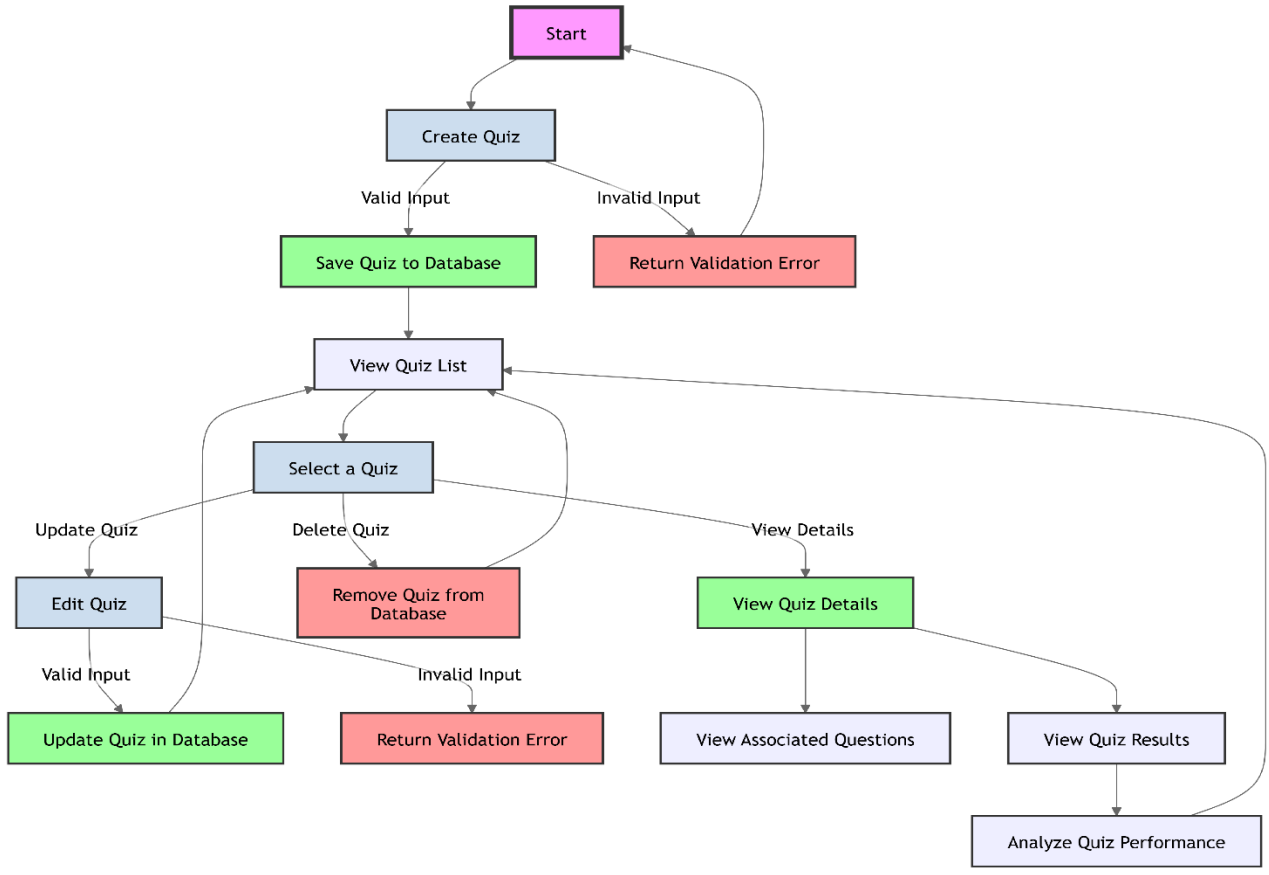


Figure 1 Quiz Activity Transition Graph

Call Graph:

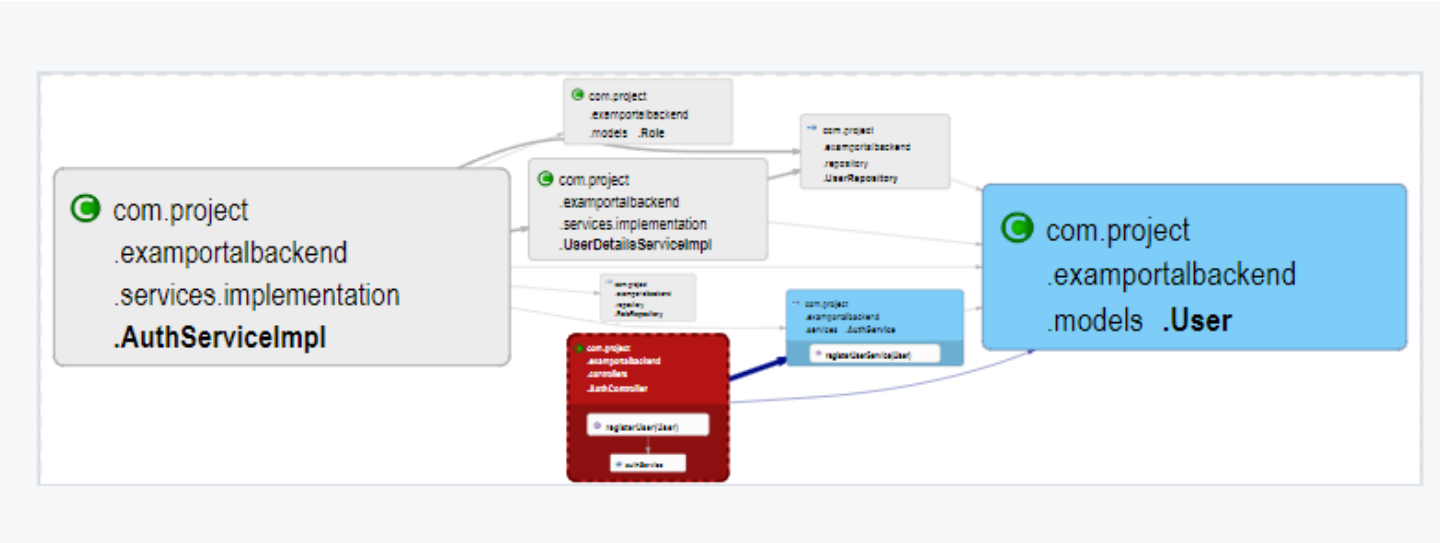
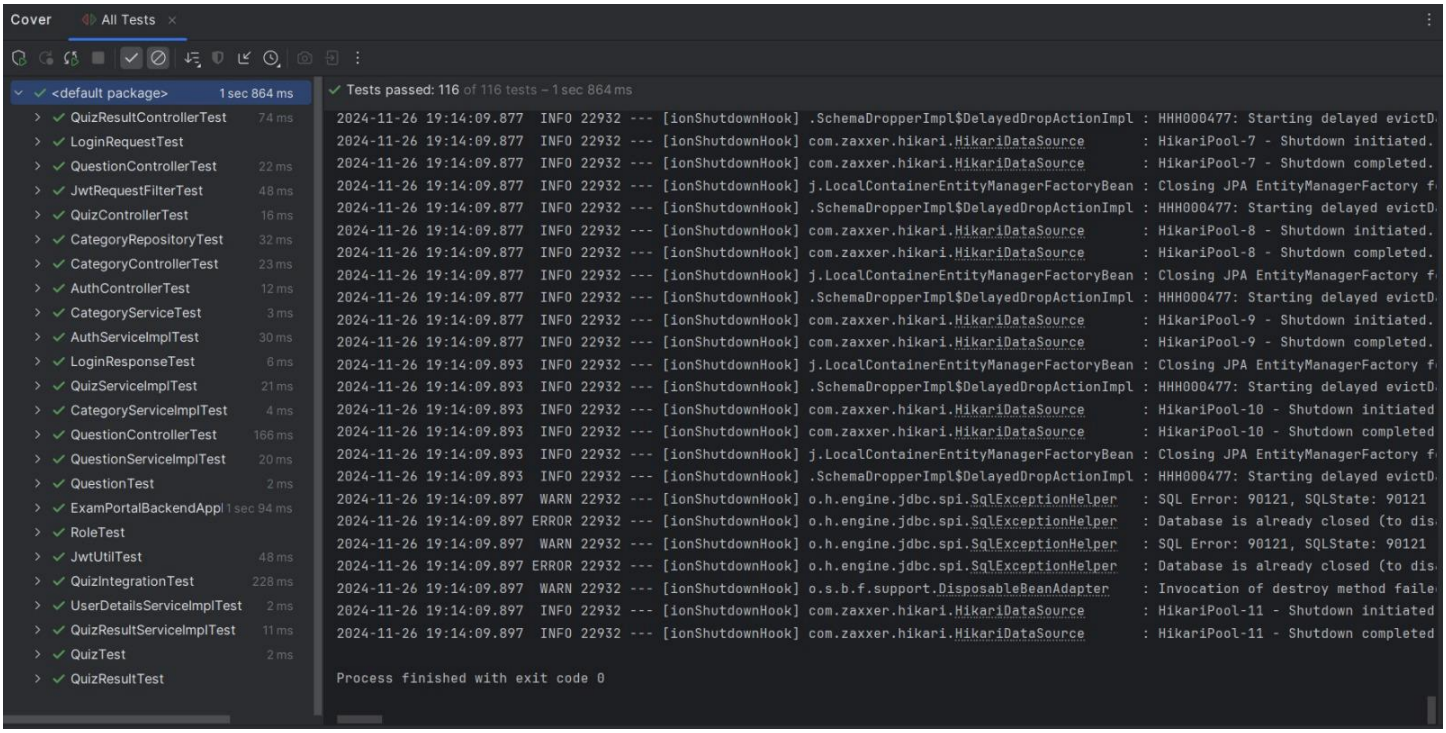


Figure 2 Registration Call Graph

Test Execution:



Class Interaction Map:

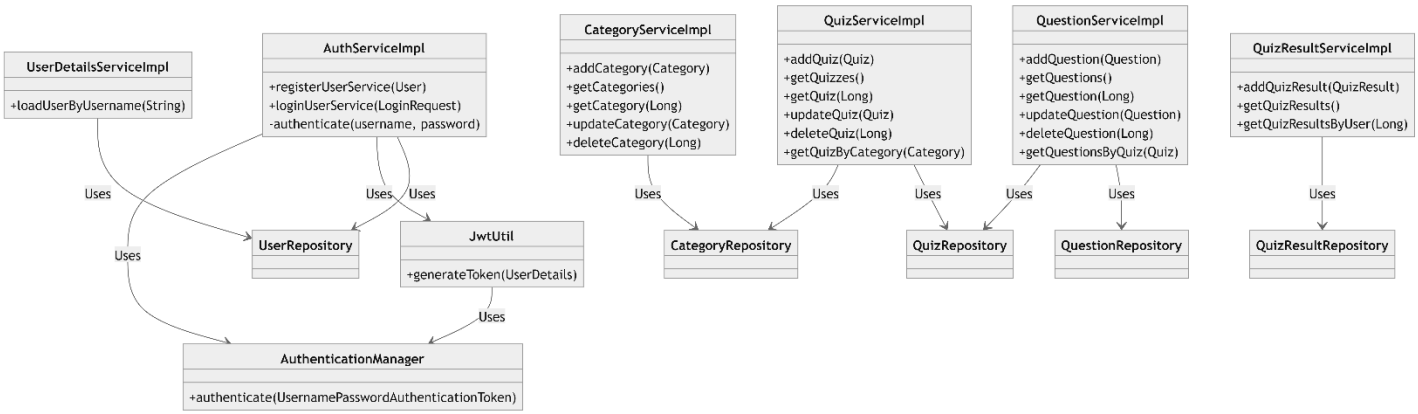


Figure 3 Service layer interactions

Code Quality:

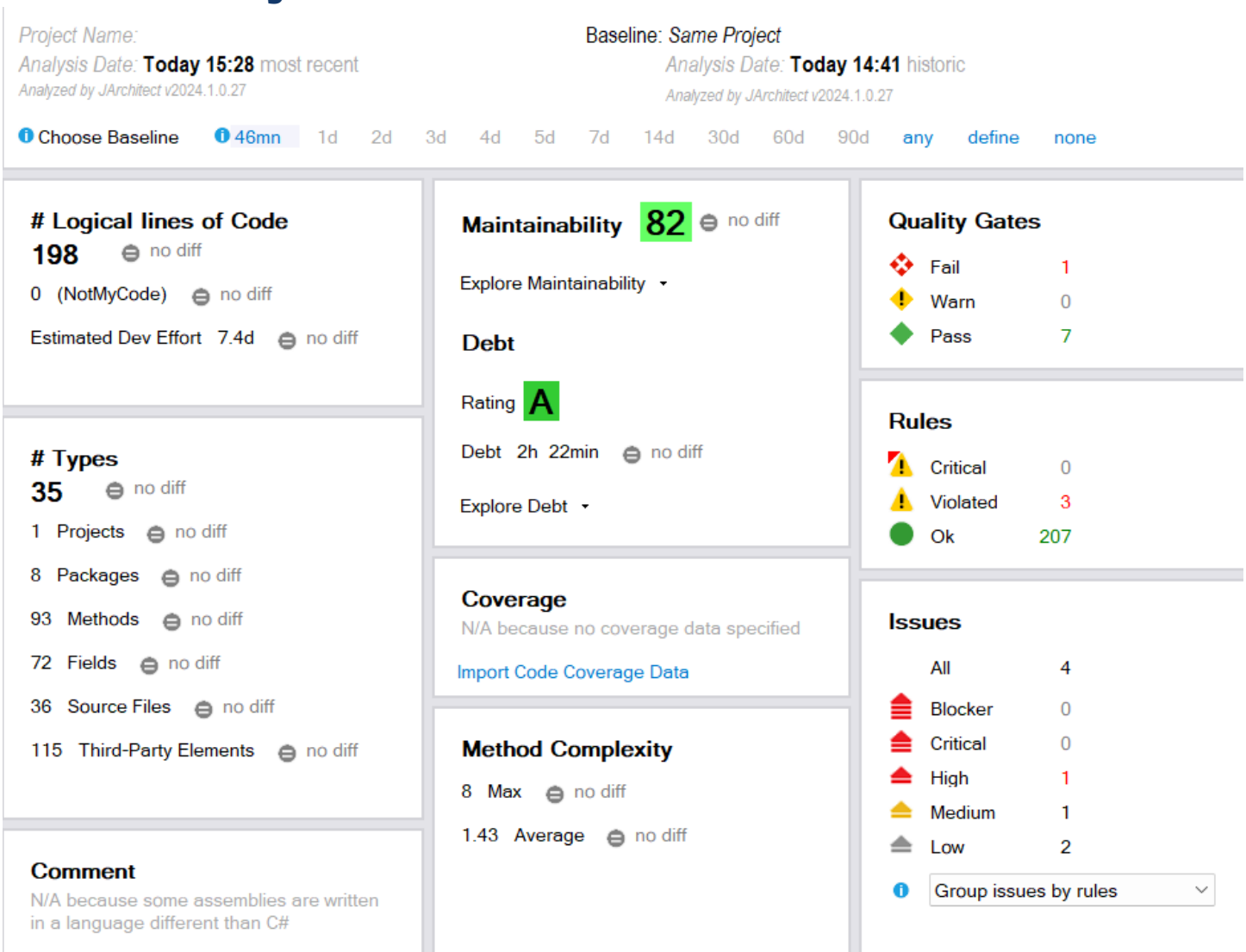
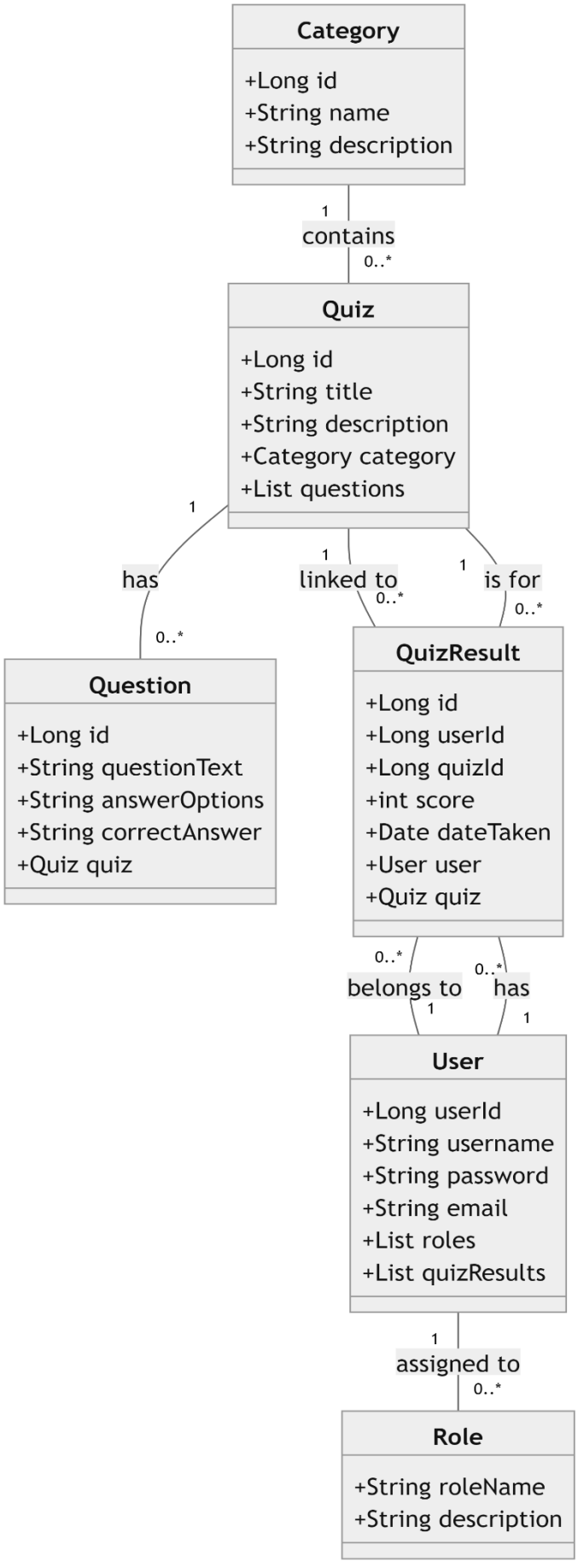


Figure 4 Code Quality According to JArchitect

Conceptual Information Model:



Postman API Testing:

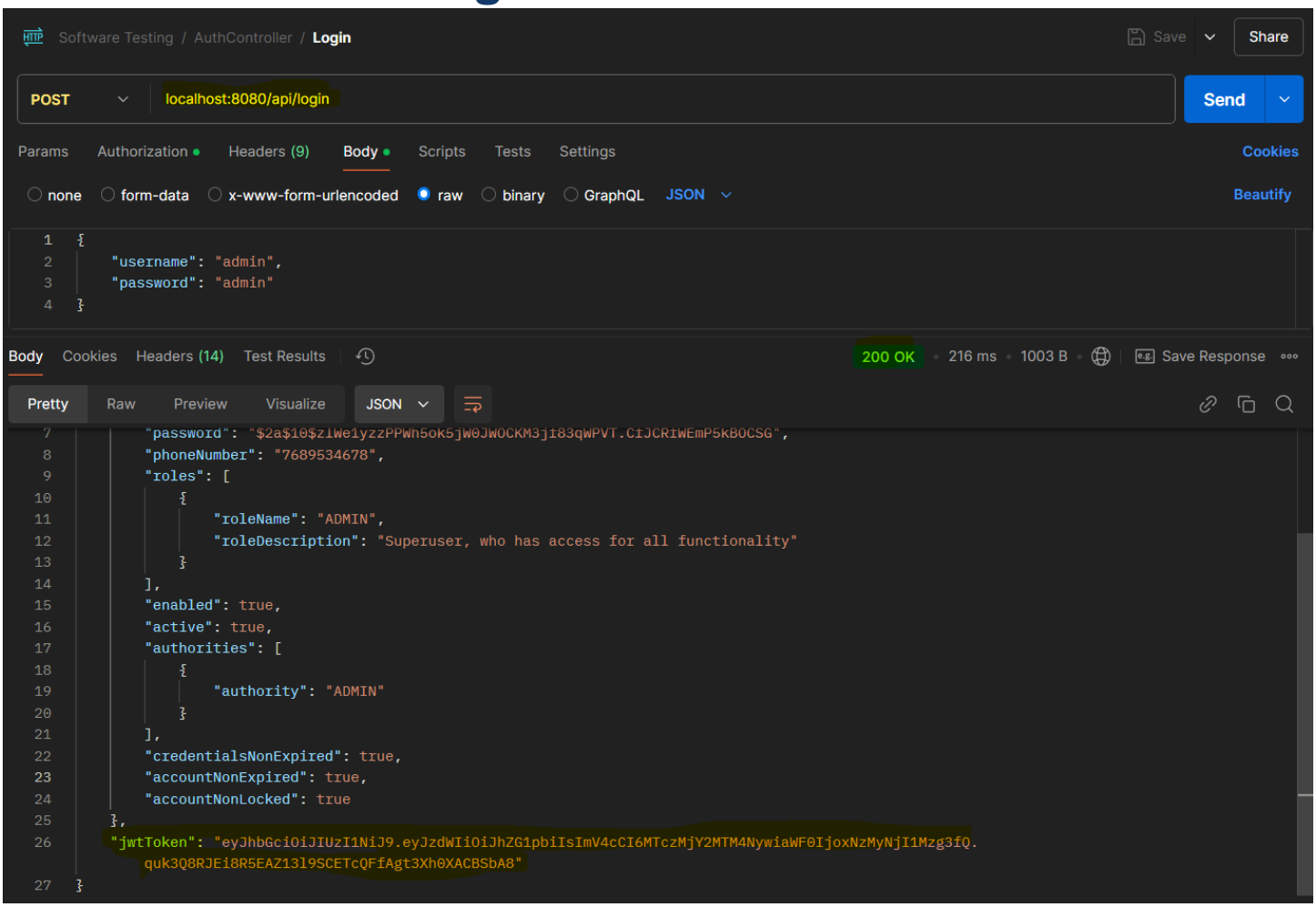


Figure 5 Login Success

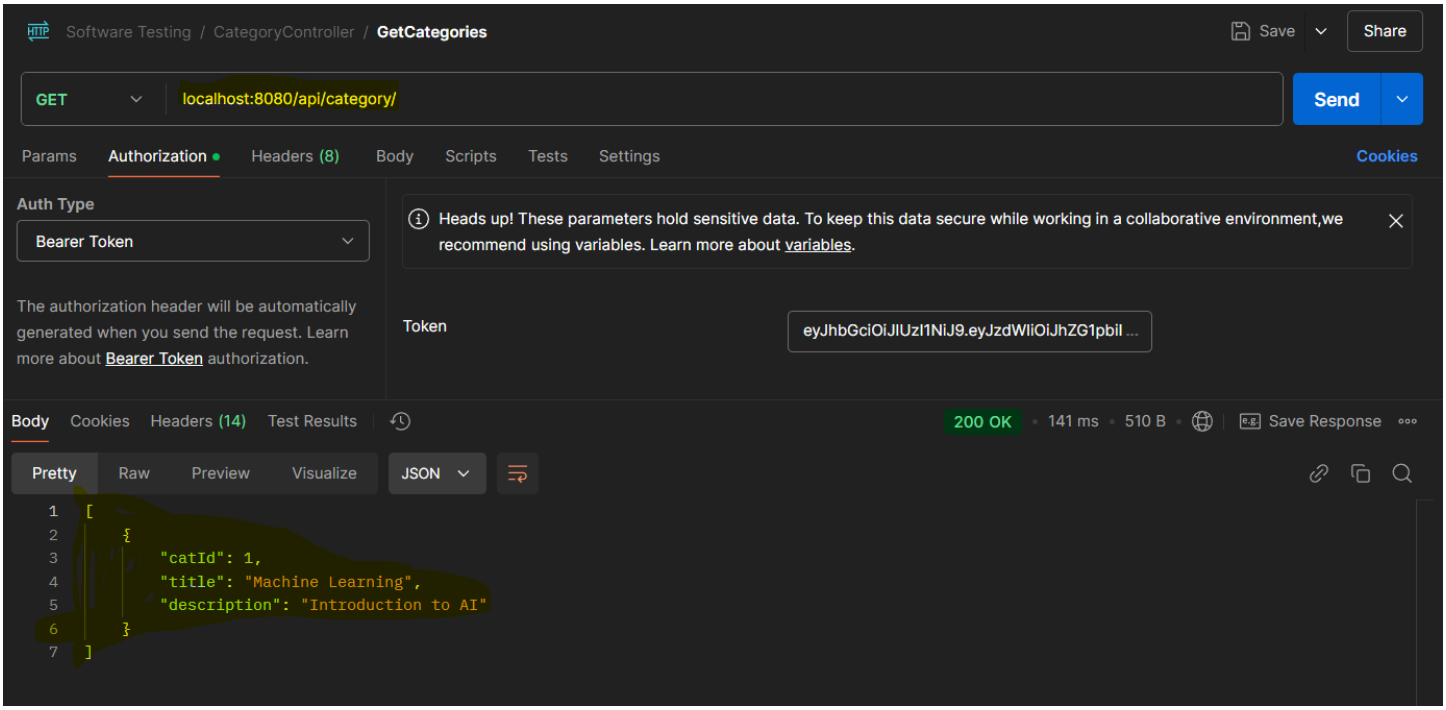
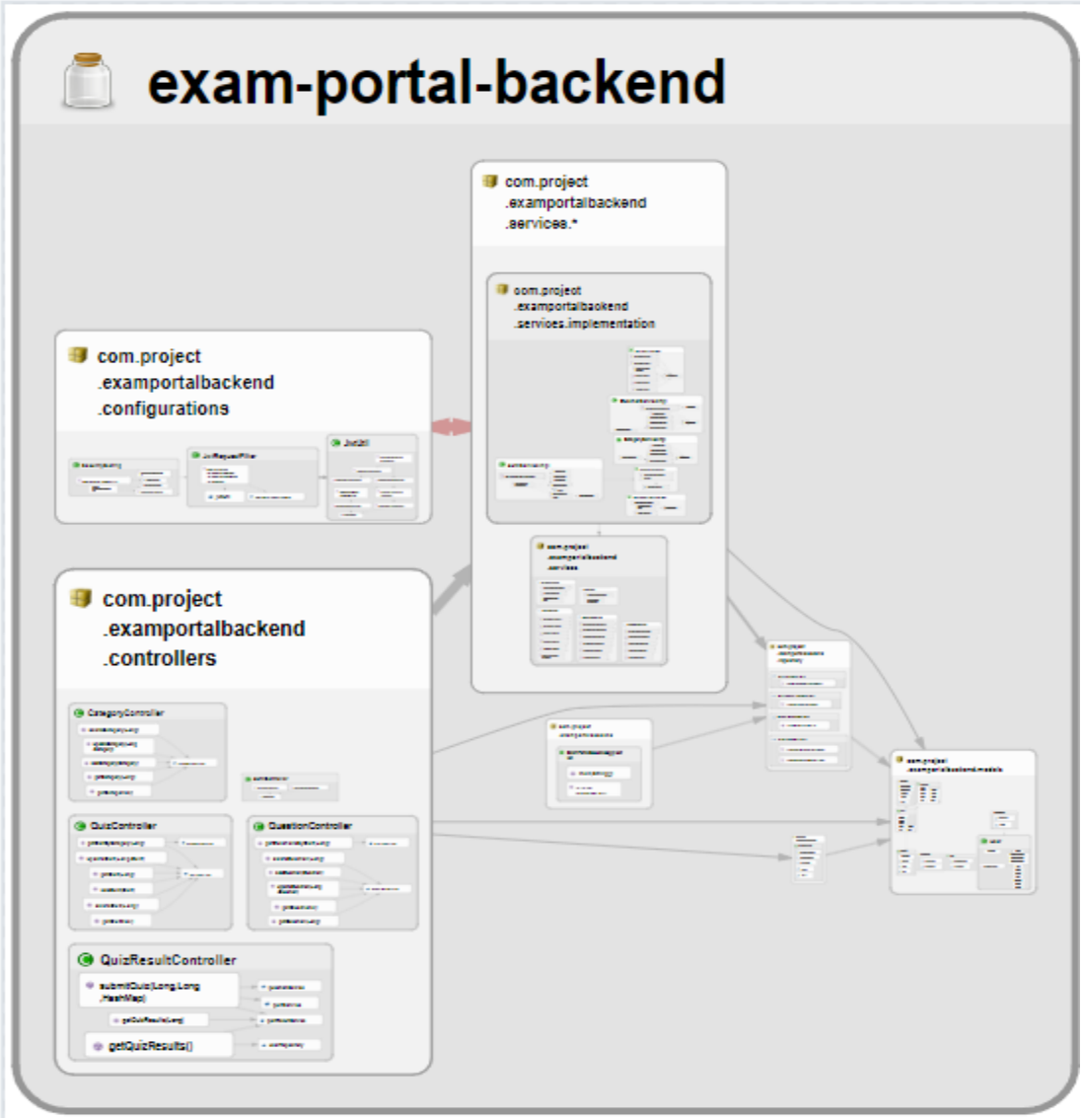


Figure 6 Get Categories

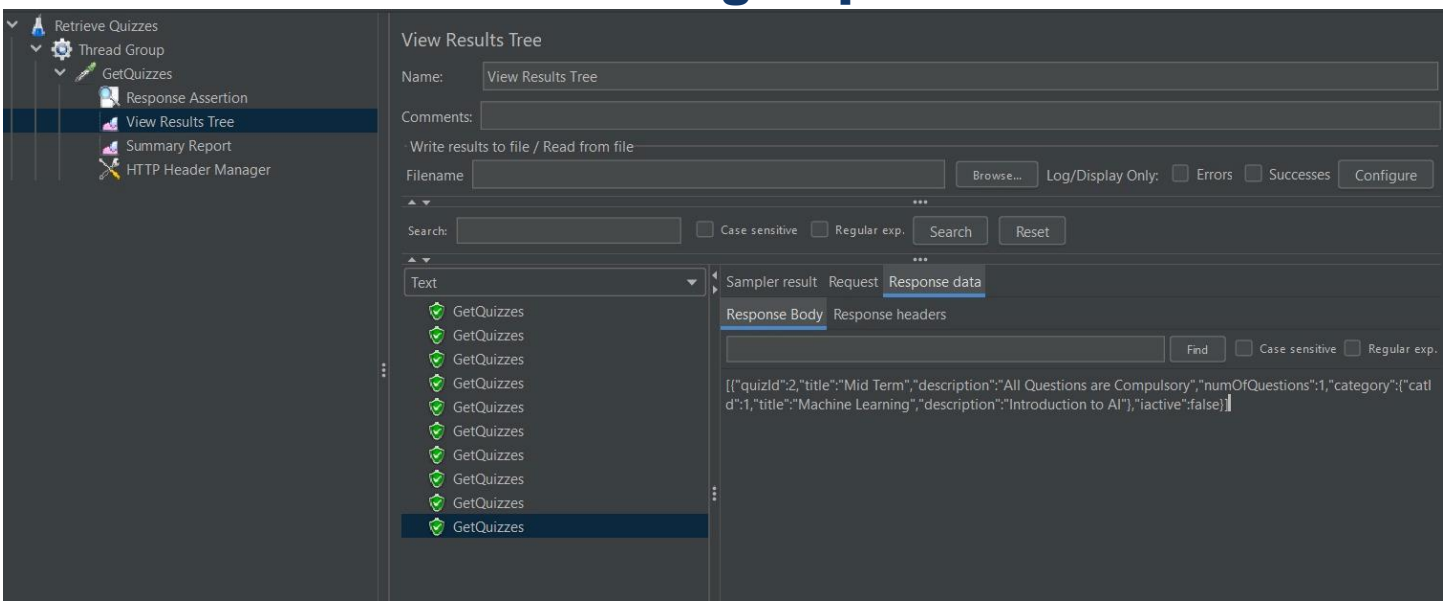
Dependency Graph:

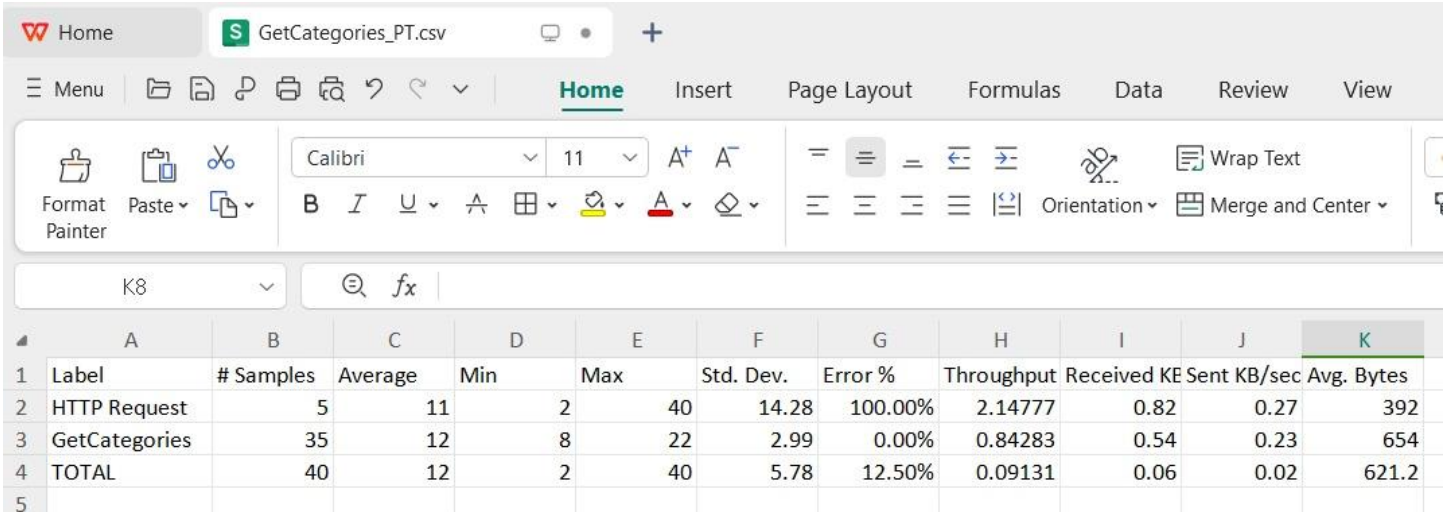


Integration Testing:

```java
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

//End-to-End (Full) Integration Testing - We have combined the controller, service, and
// repository layer in a single end-to-end test to verify that everything works together.
@SpringBootTest  Subho27
@AutoConfigureMockMvc(addFilters = false)
public class QuizIntegrationTest {

    @Autowired  5 usages
    private MockMvc mockMvc;

    @Autowired  2 usages
    private ObjectMapper objectMapper;

    @Autowired  2 usages
    private QuizRepository quizRepository;
```

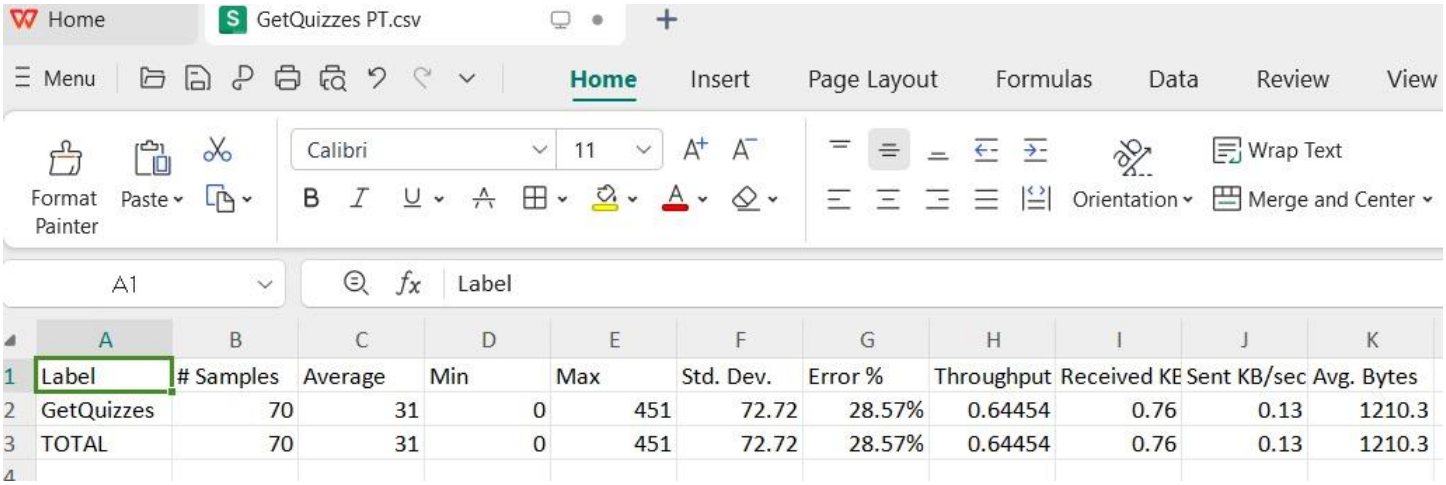
JMeter Performance Testing Report:





	A	B	C	D	E	F	G	H	I	J	K
1	Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB	Sent KB/sec	Avg. Bytes
2	HTTP Request	5	11	2	40	14.28	100.00%	2.14777	0.82	0.27	392
3	GetCategories	35	12	8	22	2.99	0.00%	0.84283	0.54	0.23	654
4	TOTAL	40	12	2	40	5.78	12.50%	0.09131	0.06	0.02	621.2
5											

Figure 7 GET /api/category/



	A	B	C	D	E	F	G	H	I	J	K
1	Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB	Sent KB/sec	Avg. Bytes
2	GetQuizzes	70	31	0	451	72.72	28.57%	0.64454	0.76	0.13	1210.3
3	TOTAL	70	31	0	451	72.72	28.57%	0.64454	0.76	0.13	1210.3
4											

Figure 8 GET /api/quiz/

Contribution:

Rishav Chandel (MT2023058):

Unit Testing (JUnit): Write and run unit tests using JUnit to ensure individual methods and classes perform as expected.
Integration Testing (JUnit): Conduct integration tests to verify the correct interaction between components in the application.
Code Coverage Maintenance (IntelliJ Plugin): Set up and maintain code coverage reports using the IntelliJ IDEA plugin to ensure that the code is sufficiently tested.
Performance Testing (JMeter): Perform load and performance testing to evaluate the application's scalability and response time under various loads.
API Testing (Postman): Create and execute Postman collections to test the API endpoints for correctness and performance.

Subhodip Rudra (MT2023103):
Creating Dependency and Call Graphs (JArchitect): Use JArchitect to generate dependency and call graphs to understand the structure and relationships in the codebase.
Creating CIM and ATG (Mermaid Format): Define the Conceptual Information Model (CIM) and the Architecture Task Graph (ATG) using Mermaid syntax for visual representation.
Coding Controllers, Services, and Repositories: Develop the application layers (controllers, services, repositories) following proper coding standards and architectural patterns.
Role-Based Authentication and Authorization: Implement role-based access control (RBAC) to ensure that only authorized users can access specific resources or perform certain actions.
Generating Code Quality Metrics (JArchitect): Analyze code quality using JArchitect to provide metrics such as complexity, coupling, and maintainability.

Source Code Link:

Online Test Management System