



A Java Spring Boot Web Application

Mohammadreza Motallebi | 7029006 | Spring 2024 | UNIFI

Advanced Techniques and Tools for Software Development

User Management and Quiz System

Introduction

This project follows **best-in-class development practices**, integrating **Test-Driven Development (TDD)**, **CI/CD automation**, **Dockerized testing**, and **SonarQube Cloud-based Static Code Analysis** to ensure software quality, maintainability, and long-term scalability.

Technologies and Tools Used

Backend Technologies

- **Spring Boot (v3.2.4)** – RESTful API development
- **Spring Data JPA** – Database interaction (**PostgreSQL**)
- **Spring Boot Actuator** – Application health monitoring

Frontend Technologies

- **Thymeleaf** – Java-based template engine
- **HTML, CSS** – Styling and interactivity

Testing & Quality Assurance

- **JUnit 5 & Mockito** – Unit testing
- **Selenium WebDriver** – End-to-end (E2E) testing
- **Testcontainers** – Database-driven integration testing
- **JaCoCo** – Enforced **100% code coverage**
- **SonarQube Cloud** – **Static Code Analysis & Technical Debt Monitoring**

Security & DevOps

- **Docker & Docker Compose** – Containerized deployment
 - **Maven (Build Automation)** – Automated dependency and build management
 - **GitHub Actions (CI/CD)** – Automated testing & deployment
 - **Environment Variables (.env)** – Secure configuration management
-

System Architecture

The project follows a **layered architecture** to ensure modularity and maintainability:

1. Controller Layer (API Endpoints)

- `UserController.java` , `QuizController.java` , `WebController.java`

2. Service Layer (Business Logic) – Manages data processing

- `UserService.java` , `QuizService.java`

3. Repository Layer (Database Interaction) – Facilitates efficient database operations

- `UserRepository.java` , `QuizResultRepository.java`

4. Persistence Layer – Ensures reliable data storage

- **PostgreSQL, Spring Data JPA, init.sql** (Database initialization)

5. Testing Layer

- **Unit Tests:** `UserServiceTest.java`
 - **Integration Tests:** `UserServiceImplT.java`
 - **E2E Tests:** `CombinedE2ETest.java`
-

Key Features

User Management

- User registration and login
- User profile management (CRUD operations)

Quiz System

- Users can participate in quizzes
- Results are recorded and analyzed

Testing Strategy

- **Unit Tests** – Validate individual methods
- **Integration Tests** – Ensure correct database interactions
- **E2E Tests** – Simulate real-world user behavior

Real-Time Monitoring & Logging

- **SLF4J + Logback** – For structured logging
 - **Spring Boot Actuator** – Provides **real-time health monitoring**
-

Database Schema & Entity Relationships


- **User (id, username, email, birthdate, etc.)**
 - **1:N → QuizResult** (Each user can have multiple quiz results)
- **QuizResult (id, user_id, score, timestamp)**

- **N:1 → User** (Each quiz result belongs to one user)

SonarQube Static Code Analysis

To ensure our code meets industry standards, we use **SonarQube Cloud** for real-time code quality analysis.

SonarQube Analysis Summary

- **Lines of Code (LOC):** 1.3k
- **Quality Gate Status:**  **Passed**
- **Code Coverage:** 100%
- **Duplications:** 0.0%
- **Security Issues:** 0
- **Maintainability Issues:** 0

Mutation Testing (99%)

Mutation testing was performed using **Pitest**, achieving a **99% mutation coverage rate**. The remaining survived mutator is found in `WebController.java` :

- **Mutation Type:** `REMOVE_CONDITIONALS_EQUAL_IF`
- **Line(s) Affected:** Line 95
- **Code Snippet:**

```
if (userId == null || userService.findById(userId).isEmpty())
```

Why This Mutator Survived

This mutator removes the conditional check for `userId == null` or if `userService.findById(userId).isEmpty()` . However, this logic is critical for ensuring only authenticated users access the system. Removing it would break the intended behavior, allowing unauthorized users to proceed.

Solution & Recommendation

Since this check is fundamental for authentication and security, it should remain in place. If removed, the application would be exposed to security risks by allowing unauthenticated users to proceed. Therefore, no further action is required. If 100% mutation coverage is required, revisiting the mutation configuration might be necessary.

Deployment & Execution

Running Locally

1. Clone the repository.
2. Run Docker:

```
docker-compose up --build
```

3. Application available at `http://localhost:8081` .

Automated Testing

- **Run unit tests**

```
mvn test
```

- **Run integration & e2e tests**

```
mvn verify
```

Continuous Integration

- **GitHub Actions + SonarQube Cloud** automates:
 - **Automated builds**
 - Static Code Analysis
 - Quality Gate Verification

Key Challenges and How We Overcame Them

Challenges Faced

Every project has hurdles, and we faced our fair share:

1. **Maven & Test Execution Issues** – Integration and E2E tests were running incorrectly, causing unstable builds.
2. **Docker & Database Issues** – PostgreSQL initialization failures required manual intervention.
3. **CI/CD Pipeline Failures** – Dependencies and execution order in GitHub Actions caused workflow instability.
4. **Configuration & Environment Mismatches** – Different Java versions led to inconsistent behavior.
5. **Selenium WebDriver Issues** – Browser automation tests frequently failed due to compatibility mismatches.

Solutions Implemented

Revised Maven Configuration – Separated **unit, integration, and E2E tests**, ensuring proper execution.

Automated Docker Initialization – Ensured PostgreSQL starts correctly and added health checks.

CI/CD Optimization – Fixed dependency issues and standardized execution order.

Standardized Java Environments – Unified configurations across local and CI/CD environments.

Resolved Selenium Issues – Ensured **ChromeDriver version compatibility** and stabilized automation tests.

Final Outcome

The project is now fully functional, CI/CD-ready, and follows industry best practices!