



A Java Spring Boot Web Application

Advanced Techniques and Tools for Software Development

Mohammadreza Motallebi | Spring 2024 | UNIFI

1. Introduction







This report presents an in-depth analysis and documentation of a Java web application. The application is structured using several key techniques and frameworks, adhering to industry best practices. This report aims to provide a comprehensive overview of the design patterns, transaction management, ACID properties, and implementation choices made during the development of the application. Additionally, it includes instructions on how to run the program, details on testing, continuous integration and deployment (CI/CD) using GitHub Actions, and information about achieving 100% code coverage using JaCoCo. The target audience for this report includes software developers, architects, and anyone interested in understanding the intricacies of a Java-based web application.

The application in question is a quiz management system that allows users to register, log in, create quizzes, and submit quiz responses. It is built using the

Spring framework, which provides a robust and flexible environment for developing enterprise-level applications. The choice of technologies and design patterns ensures that the application is scalable, maintainable, and easy to extend.

Comprehensive testing has been conducted to ensure the reliability and functionality of the application. Unit and integration tests have been successfully executed, achieving 100% code coverage on local environments using JaCoCo. Furthermore, continuous integration and continuous deployment (CI/CD) have been implemented using GitHub Actions, automating the build, test, and deployment processes to ensure a streamlined and error-free development lifecycle.

hello

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.userhello.hello.controller		100%		100%	0	29	0	67	0	23	0	3
com.userhello.hello.model		100%	n/a	n/a	0	43	0	75	0	43	0	3
com.userhello.hello.Service		100%		100%	0	15	0	20	0	14	0	2
com.userhello.hello		100%	n/a	n/a	0	3	0	7	0	3	0	1
Total	0 of 575	100%	0 of 14	100%	0	90	0	169	0	83	0	9

2. Techniques and Frameworks

Design Patterns

Design patterns are crucial for creating a clean, efficient, and maintainable codebase. The application employs several well-known design patterns:

Model-View-Controller (MVC)

The MVC pattern is the cornerstone of this application's architecture. It divides the application into three interconnected components:

- **Model:** Represents the application's data and business logic. In this application, the model consists of entities such as `User`, `Quiz`, `QuizSubmission`, and `QuizResult`.
- **View:** Responsible for presenting data to the user and handling user interaction. The views are typically HTML pages rendered by Thymeleaf, a server-side Java template engine.
- **Controller:** Manages user input and interacts with the model to render the appropriate view. Controllers like `UserController`, `QuizController`, and `WebController` handle HTTP requests, process input, and return responses.

Singleton

The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. This is particularly useful for classes that manage resources such as database connections or configuration settings. For instance, the `ApplicationConfig` class may use the Singleton pattern to ensure a single configuration instance is used throughout the application.

Repository

The Repository pattern abstracts the data access layer, providing a clean API for performing CRUD operations. This pattern separates the business logic from data access logic, making the code more modular and testable. In this application, repositories such as `UserRepository` and `QuizRepository` extend Spring Data JPA's `JpaRepository` interface, enabling seamless interaction with the database.

Transaction Management

Transactions are vital for ensuring data integrity and consistency. The application leverages Spring's robust transaction management capabilities, supporting both programmatic and declarative approaches.

Programmatic Transaction Management

Programmatic transaction management involves explicitly beginning, committing, or rolling back transactions in the code. This approach offers fine-grained control over transaction boundaries but can lead to more complex and less maintainable code. For example, you might see code like this in a service method:

```
@Transactional
public void someBusinessMethod() {
    // Transactional code here
}
```

Declarative Transaction Management

Declarative transaction management simplifies transaction handling by using annotations or XML configuration. This approach reduces boilerplate code and enhances readability. Spring's `@Transactional` annotation is commonly used for declarative transaction management. For instance:

```
@Transactional
public class UserService {
    // Methods within this class will be transactional
}
```

ACID Properties

ACID (Atomicity, Consistency, Isolation, Durability) properties are fundamental principles that ensure reliable transaction processing in a database system.

Atomicity

Atomicity ensures that a transaction is treated as a single, indivisible unit of work. If any part of the transaction fails, the entire transaction is rolled back, leaving the system in its previous state. This guarantees that incomplete transactions do not affect the system.

Consistency

Consistency ensures that a transaction brings the database from one valid state to another. It enforces data integrity constraints before and after the transaction. This means that any transaction will leave the database in a valid state, adhering to all predefined rules.

Isolation

Isolation ensures that concurrent transactions do not interfere with each other. Each transaction is isolated from others, preventing issues such as dirty reads, non-repeatable reads, and phantom reads. This is crucial in multi-user environments where multiple transactions are processed simultaneously.

Durability

Durability guarantees that once a transaction is committed, it remains so, even in the event of a system failure. This means that the changes made by a committed transaction are permanent and will survive system crashes or power failures.

3. Design and Implementation Choices

3.1 Repository Layers

The repository layer is responsible for data access and manipulation. It acts as an intermediary between the service layer and the database, providing a clean API for data operations.

3.1.1 Database Connection

The application uses Spring Data JPA for database interactions. Spring Data JPA abstracts the boilerplate code required for database operations, enabling developers to focus on business logic. The database connection settings are configured in the `application.properties` file, including the database URL, username, and password.

Example configuration in `application.properties` :

```
spring.datasource.url=jdbc:postgresql://postgres:5432/user_db
spring.datasource.username = postgres001
spring.datasource.password = ****
```

3.2 Service Layers

The service layer contains the business logic of the application. It interacts with the repository layer to perform CRUD operations and other business-related tasks. The service layer is where the core functionality of the application resides.

UserService

The `UserService` class manages user-related operations such as registration, authentication, and profile management. It interacts with the `UserRepository` to perform CRUD operations on user entities.

Example of `UserService` :

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public User registerUser(User user) {
        // Business logic for user registration
        return userRepository.save(user);
    }
}
```

```

    }

    public User authenticateUser(String username, String password) {
        // Business logic for user authentication
        return userRepository.findByUsernameAndPassword(username, password);
    }
}

```

QuizService

The `QuizService` class handles operations related to quiz creation, submission, and result calculation. It interacts with `QuizRepository` and other repositories to manage quiz-related data.

Example of `QuizService`:

```

@Service
public class QuizService {
    @Autowired
    private QuizRepository quizRepository;

    public Quiz createQuiz(Quiz quiz) {
        // Business logic for creating a quiz
        return quizRepository.save(quiz);
    }

    public QuizSubmission submitQuiz(QuizSubmission submission) {
        // Business logic for submitting a quiz
        return quizRepository.save(submission);
    }

    public List<QuizResult> calculateResults(Long quizId) {
        // Business logic for calculating quiz results
        return quizRepository.findResultsByQuizId(quizId);
    }
}

```

3.3 Model

The model layer represents the application's data structures. It consists of Java classes that map to database tables, defining the entities and their relationships. Key models in the application include:

User

The `User` class represents a user in the system with attributes like username, password, and email.

Example of `User` model:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    private String email;

    // Getters and setters
}
```

Quiz

The `Quiz` class represents a quiz with attributes like title, description, and a list of questions.

Example of `Quiz` model:

```
@Entity
public class Quiz {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String description;
    @OneToMany(mappedBy = "quiz", cascade = CascadeType.ALL)
    private List<Question> questions;
```

```

        private List<Question> questions;

        // Getters and setters
    }

```

QuizSubmission

The `QuizSubmission` class captures the details of a user's quiz attempt, including the user's responses and the submission time.

Example of `QuizSubmission` model:

```

@Entity
public class QuizSubmission {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @ManyToOne
    private User user;
    @ManyToOne
    private Quiz quiz;
    private LocalDateTime submissionTime;
    @ElementCollection
    private Map<Long, String> responses;

    // Getters and setters
}

```

3.4 REST Controller and Web Controller

The application uses both REST and Web controllers to handle different types of requests. Controllers are annotated with `@RestController` or `@Controller` and handle HTTP requests by mapping them to appropriate handler methods.

REST Controllers

REST controllers provide APIs for frontend applications or other services to interact with the backend. They return data in JSON format and use annotations such as `@GetMapping`, `@PostMapping`, `@PutMapping`, and `@DeleteMapping` to map HTTP requests to handler methods.

Example of `UserController` :

```
@RestController
@RequestMapping("/api/users")
public class UserController {
    @Autowired
    private UserService userService;

    @PostMapping("/register")
    public ResponseEntity<User> registerUser(@RequestBody User user) {
        User registeredUser = userService.registerUser(user);
        return new ResponseEntity<>(registeredUser, HttpStatus.CREATED);
    }

    @PostMapping("/login")
    public ResponseEntity<User> loginUser(@RequestBody User user) {
        User authenticatedUser = userService.authenticateUser(user.getUsername(), user.getPassword());

        if (authenticatedUser != null) {
            return new ResponseEntity<>(authenticatedUser, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.UNAUTHORIZED);
        }
    }
}
```

Web Controllers

Web controllers handle web page requests and return HTML views. They use annotations such as `@GetMapping` and `@PostMapping` to map HTTP requests to handler methods, and return view names that are resolved by the view resolver.

Example of `WebController` :

```
@Controller
public class WebController {
    @GetMapping("/")
    public String home() {
        return "index";
    }

    @GetMapping("/login")
    public String login() {
        return "login";
    }

    @GetMapping("/register")
    public String register() {
        return "register";
    }
}
```

3.5 Errors

Error handling is crucial for providing meaningful feedback to users and developers. The application uses Spring's `@ControllerAdvice` to manage exceptions globally, ensuring consistent and user-friendly error responses.

Example of global exception handling:

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(ResourceNotFoundException ex) {
        ErrorResponse errorResponse = new ErrorResponse(HttpStatus.NOT_FOUND.value(), ex.getMessage());
        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }
}
```

```

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleException(Exception ex) {
        ErrorResponse errorResponse = new ErrorResponse(HttpStatus.INTERNAL_SERVER_ERROR.value(), ex.getMessage());
        return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

```

4. How to Run the Program

Prerequisites

Before running the application, ensure you have the following installed:

- **Java Development Kit (JDK) 8 or higher:** Required for compiling and running the application.
- **Apache Maven:** A build automation tool used for managing dependencies and building the project.
- **A relational database** (e.g., MySQL, PostgreSQL): The application uses a relational database to store data.

Steps to Run

1. Clone the Repository

Clone the project repository from GitHub or another version control system:

```

git clone <https://github.com/mrmlb94/hello>
cd hello

```

2. Run the Application using Docker

If you prefer to run the application using Docker, follow these steps:

1. Build the Docker images and start the containers using Docker Compose:

```

mvn clean package
docker-compose --build

```

```
docker-compose up
```

2. Docker will handle the setup and configuration, including the database and the application server.

3. Access the Application

Open a web browser and navigate to `http://localhost:8080`. You should see the application's home page.

Testing

The application includes two testing profiles, allowing for comprehensive testing of both unit and integration tests, as well as end-to-end (E2E) tests.

1. Unit and Integration Tests:

- The `skip-e2e-tests` profile is used to run only the unit and integration tests, skipping the end-to-end tests. This ensures that individual components of the application work correctly and that they integrate well together without running the full suite of E2E tests.
- To run these tests, use the following command:

```
mvn test -P skip-e2e-tests
```

2. End-to-End Tests:

- To run the full suite of 76 end-to-end tests, which cover the entire application's behavior and ensure that the system works as expected from the user's perspective, use the default testing profile (or omit the profile flag).
- To run these tests, execute the following Maven command:

```
mvn test
```

These testing profiles allow you to verify both the individual components and the overall functionality of the application, ensuring that it meets the required quality standards.

Conclusion

This report provides a detailed exploration of the Java web application, highlighting its architectural design, key frameworks, and development practices. Through the application of well-known design patterns such as MVC, Singleton, and Repository, the project achieves a high level of modularity, scalability, and maintainability. The implementation of robust transaction management and adherence to ACID principles ensure data integrity and consistency, crucial for reliable operations in a multi-user environment.

The report also outlines the steps required to run the application, both through traditional methods and using Docker for simplified deployment. Additionally, comprehensive testing has been conducted using distinct profiles for unit, integration, and end-to-end tests, ensuring that the application functions correctly both at the component level and as a complete system. The integration of continuous integration and deployment (CI/CD) using GitHub Actions further enhances the development workflow, automating the build, test, and deployment processes.

In summary, this application exemplifies best practices in modern software development, combining a solid architectural foundation with rigorous testing and automated deployment strategies. It serves as a reliable, scalable, and maintainable solution, well-suited to meet the demands of enterprise-level applications.