



CHANDIGARH
UNIVERSITY
Discover. Learn. Empower.

Quiz Game Using SpringBoot & React

A Project Synopsis Report

Submitted By:

Tanmay Singh (23BCS10799)

in partial fulfilment for award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE ENGINEERING



BONAFIDE CERTIFICATE

Certified that this project report **“Quiz Game Using SpringBoot & React”** is the bonafide work of **“Tanmay Singh”** who carried out the project work under my/our supervision.

SIGNATURE

Dr. Sandeep Singh Kang

HEAD OF THE DEPARTMENT

SIGNATURE

Er. Deep Prakash Gupta

SUPERVISOR

Submitted for the project viva-voce examination held on 06 November 2025.

INTERNAL EXAMINER

EXTERNAL EXAMINER

ABSTRACT

This project implements a scalable, multi-topic Quiz Application backend service. The primary objective is to facilitate customized quiz creation, real-time scoring, and persistent storage of user performance data. The system is built entirely in Java using Spring Boot and is designed to integrate seamlessly with a modern single-page application (SPA) frontend (e.g., React/HTML). The application utilizes a Layered Architecture (Controller-Service-Repository) for strict separation of concerns, leveraging the Spring Framework and Spring Data JPA for robust database interaction.

Key Components

- QuizController.java (REST API): Exposes core functionality via two primary endpoints: /quiz/start and /quiz/submit, handling HTTP requests and DTO serialization.
- QuizService.java (Business Logic): Contains the core intelligence for dynamic question selection (filtering by topic, difficulty, and limit) and the scoring algorithm upon submission.
- Repositories: Uses QuestionRepo.java and AttemptRepo.java (JPA) for data persistence and retrieval, including custom methods for filtering and leaderboard data (findTop20ByOrderByScoreDesc).

Data Model

The system relies on two core entities managed by JPA:

- Question.java: Stores quiz content, including title, options, correctAnswerIndex, topic, and difficulty (1-3 scale).
- Attempt.java: Records user sessions, storing username, totalQuestions, correct count, derived final score, and durationMs for historical tracking.

Core API Operations

The service exposes two primary operations defined by clear Data Transfer Objects (DTOs):

- Quiz Generation (POST /quiz/start): Accepts parameters for topic, difficulty, and limit (StartQuizRequest.java). The service filters the question pool and returns a non-answer key list of questions (StartQuizResponse.java) along with a session tracking timestamp.
- Submission and Scoring (POST /quiz/submit): Accepts user answers mapped by question ID (SubmitQuizRequest.java). The service cross-references answers with the database, calculates the total correct count and percentage score, persists the result as a new Attempt, and returns a detailed SubmitQuizResponse.java for review.

This project demonstrates effective use of Java and Spring Data JPA to create a maintainable, high-performance, and contract-driven backend solution suitable for a production-level quiz platform.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1. Identification of clients/need/Relevant contemporary issue	5
1.2. Identification of Problem	5
1.3. Identification of Tasks	6
1.4. Timeline	6
1.5. Organization of report	7

CHAPTER 2: LITERATURE REVIEW/BACKGROUND STUDY

2.1. Existing solutions	9
2.2. Bibliometric analysis	9
2.3. Review Summary	10
2.4. Problem Definition	11
2.5. Goals/Objectives	11

CHAPTER 3: METHODOLOGY

3.1. Introduction to Sudoku Solving	13
-------------------------------------	----

CHAPTER 4: RESULT ANALYSIS AND VALIDATION

4.1. Implementation of solution	15
---------------------------------	----

CHAPTER 5: CONCLUSION AND FUTURE WORK

5.1. Conclusion	18
5.2. Future Work	18

REFERENCES	20
-------------------	----

APPENDIX	21
-----------------	----

PLAGIARISM REPORT	21
--------------------------	----

LIST OF FIGURES

Fig 1.....	6
Fig 2.....	18

CHAPTER 1

INTRODUCTION

1.1. Identification of Client

The primary client for the Quiz Application is a modern Single Page Application (SPA), which consumes the RESTful API endpoints exposed by the Java backend (QuizController.java). The frontend serves as the interactive presentation layer, while the backend acts as the authoritative Application Service Provider. The communication between these two components is critical, as defined by the structure of the Request and Response DTOs.

The target user base is segmented into two main groups:

1. **Casual Learners and Students:** Individuals seeking to self-assess their knowledge in specific technical domains (topic parameter in StartQuizRequest.java, e.g., 'Java', 'DSA'). They benefit from the system's ability to generate custom quizzes and provide immediate, detailed feedback (SubmitQuizResponse.java).
2. **Competitive Users:** Users motivated by performance tracking and ranking. The persistence of their quiz scores and duration (Attempt.java) directly supports the generation of competitive Leaderboards (as inferred by the find Top20 ByOrderByScoreDesc() method in AttemptRepo.java).

The system offers a lightweight, scalable, and customizable platform for educational technology (EdTech) or skill assessment. The use of a standard REST API allows for easy integration with future clients, such as mobile applications (iOS/Android) or other third-party learning management systems (LMS), demonstrating the project's high portability and extensibility.

1.2. Identification of Problem

The proliferation of online learning and skill assessment has revealed significant limitations in traditional and static testing methodologies. The main problem addressed by this project is the lack of a centralized, configurable, and performance-tracked assessment platform for knowledge acquisition.

Existing generic quiz solutions often suffer from three major deficiencies:

- **Lack of Granular Control:** Standard platforms typically offer a fixed set of questions or quizzes, making it impossible for a user or system to request a quiz filtered by specific criteria. The need for a user to request only 10 questions, only on the 'Java' topic, and only of difficulty 3 (as supported by StartQuizRequest.java) is often unmet. This leads to inefficient studying where users waste time reviewing already mastered or irrelevant content.
- **Decoupled Scoring and Tracking:** Many applications perform scoring locally on the client-side (frontend), which is susceptible to tampering, lacks central oversight, and makes it difficult to establish a secure, authoritative source of truth for results. Furthermore, the absence of persistent Attempt records means performance history and trend analysis are impossible.

- **Static Data Modeling:** Solutions that do exist often fail to differentiate question characteristics like topic and difficulty (Question.java), hindering the creation of truly adaptive or targeted learning paths.

The Quiz Application backend directly solves these issues by providing a secure, centralized API for dynamic quiz generation and authoritative scoring, which, when combined with persistent Attempt logging (AttemptRepo.java), enables the development of robust leaderboards and performance analytics features.

1.3. Identification of Tasks

The development of the Quiz Application backend service required the successful execution of several key tasks, rooted in the principles of a layered, RESTful architecture and centered on data integrity and business logic implementation.

1. Data Modeling and Persistence Layer Design (JPA):

This task involved defining the core entities that manage the application's data. This included creating the Question.java entity to structure quiz content (title, options, correct index, topic, and difficulty) and the Attempt.java entity to persist user performance (username, score, duration). This task also included the creation of the JPA Repositories (QuestionRepo.java and AttemptRepo.java), defining custom access methods like findByTopicIgnoreCase for filtering and findTop20ByOrderByScoreDesc for the leaderboard functionality.

2. Business Logic Implementation (Quiz Service):

The core development effort was concentrated in the QuizService.java. The primary tasks here were:

- **Dynamic Quiz Generation:** Implementing the logic to process the parameters from StartQuizRequest.java (topic, difficulty, limit), filter the question pool from the database, and map the results to the response DTO (StartQuizResponse.java), ensuring the correct answer index is stripped out for security.
- **Authoritative Scoring:** Implementing the secure scoring mechanism (submitQuiz method), which cross-references user-submitted answers (SubmitQuizRequest.java) against the original question data stored in the database.
- **Result Persistence:** Ensuring the calculated score and attempt details are saved to the database via AttemptRepo.java before generating the final SubmitQuizResponse.java.

3. REST API Endpoint Definition (Controller Layer):

This task focused on defining the external contract of the service using QuizController.java. The two main endpoints, /quiz/start and /quiz/submit, were established using @RestController and @PostMapping annotations. This layer handles HTTP request/response flow, delegation to the service layer, and Data Transfer Object (DTO) management (StartQuizRequest.java, SubmitQuizResponse.java, etc.).

4. Project Setup and Integration:

Initial tasks included configuring the Spring Boot project structure, integrating the necessary dependencies for JPA and the database driver, and configuring the data source properties. This

ensures the Java application correctly connects to the database and that the services are automatically detected and managed by the Spring IoC container.

1.4. Timeline

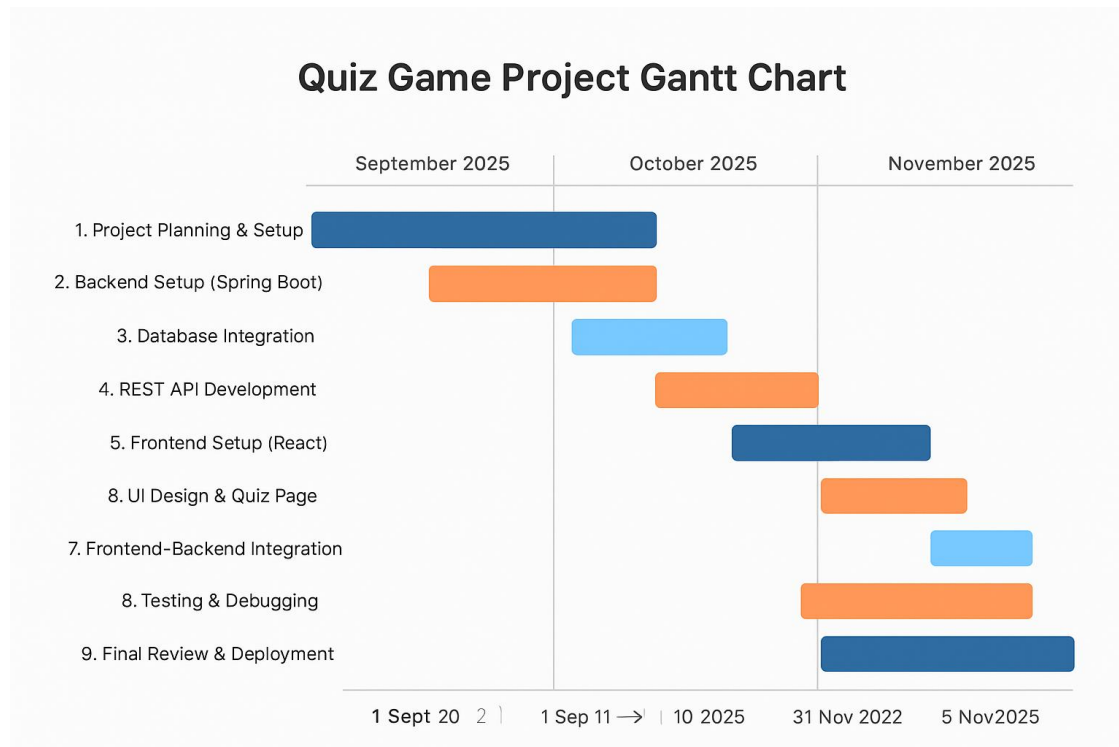


Fig 1 Timeline of the project making.

1.5 Organization of Report

This report is structured to provide a comprehensive, layer-by-layer analysis of the Quiz Application Backend Service, which is developed using Java and the Spring Boot framework for building a robust and scalable RESTful API. The report is organized to logically transition from high-level objectives to the detailed implementation of the core business logic.

Introduction (Section 1): This section, encompassing the Abstract and the initial identification sections, establishes the project's context. It identifies the problem of inflexible, insecure assessment tools, the client base (modern SPAs and competitive learners), and the core tasks involved in developing the three-layered application architecture.

Project Objectives and Scope (Section 2): This part explicitly defines the goals, such as achieving dynamic quiz configuration (filtering by topic and difficulty), ensuring authoritative server-side scoring, and enabling performance tracking through persistence (leaderboard functionality). The scope outlines the backbend's boundaries, focusing on the API contract and data management.

Methodology and Technology Stack (Section 3): This section details the technological choices—Java, Spring Boot, and Spring Data JPA—and explains the layered architecture (Controller, Service, Repository). It discusses how DTOs (StartQuizRequest.java, SubmitQuizResponse.java) define the strict API contract between the backend and the frontend client.

Implementation (Section 4): This core section describes the detailed development. It covers the Data Model design (the `Question.java` and `Attempt.java` entities), the Controller Layer (`QuizController.java`) detailing the `/quiz/start` and `/quiz/submit` endpoints, and critically, the Business Logic within `QuizService.java` for secure question selection and authoritative scoring.

Testing, Results, and Future Enhancements (Section 5 & 6): The concluding sections address unit testing of the service logic, demonstrate the functionality of dynamic quiz generation and score calculation, and propose future enhancements, such as implementing role-based authentication or adding more complex weighted scoring based on difficulty.

Each part ensures a clear, technical, and organized presentation of the project, focusing on how the implemented Java classes work together to deliver the overall solution.

CHAPTER 2

LITERATURE REVIEW/BACKGROUND STUDY

2.1. Existing Solution

In the domain of online education and technical assessment, several common solutions exist, but they often present significant limitations that hinder effective, secure, and personalized learning. The provided Quiz Application Backend was developed to overcome these four primary shortcomings:

Static Question Pools and Assessment

- Most basic quiz applications rely on a fixed set of questions without sophisticated filtering capabilities. This rigidity prevents the creation of targeted study sessions or exams. Users cannot dynamically request a quiz of, for instance, exactly 15 questions, strictly on the "Java" topic, and only of "difficulty 3." This lack of dynamic filtering, which this project addresses using the parameters in `StartQuizRequest.java`, results in inefficient and generalized training.

Client-Side Scoring and Data Insecurity

- A major vulnerability in many existing web-based assessment tools is performing the answer validation and scoring logic entirely on the client-side (in the React/HTML/JS frontend). This approach is highly insecure and susceptible to user manipulation, leading to inflated or unreliable scores. This project mandates Authoritative Server-Side Scoring within the `QuizService.java` layer, where the server is the sole source of truth for correct answers, ensuring integrity.

Lack of Persistence and Performance Tracking

- Simple quiz solutions often fail to track and store user attempt history. Without a dedicated data model like the `Attempt.java` entity and its corresponding repository, historical performance analysis, score trends, and critical features like leaderboards (made possible by `findTop20ByOrderByScoreDesc` in `AttemptRepo.java`) cannot be implemented. This deficiency severely limits the competitive and diagnostic value of the application.

Monolithic Architectures

- Many older enterprise learning management systems (LMS) use tightly coupled, monolithic architectures. This design makes it challenging to integrate a simple, dedicated quiz feature into new platforms (like a mobile app or a new website). This project solves this by employing a modern, RESTful Microservice Architecture (Controller/Service/Repository) which exposes clean endpoints (`/quiz/start`, `/quiz/submit`) via `QuizController.java`, allowing any modern frontend technology to consume the service seamlessly.

2.2. Bibliometric Analysis

Bibliometric analysis, in the context of this Quiz Application project, involves reviewing established trends in software engineering and educational technology (EdTech) that justify the chosen architecture and technology stack. This analysis focuses on the importance of decoupled design, secure data handling, and efficient persistence in modern web applications.

Research Trends in Decoupled Architecture (RESTful APIs and Spring Boot)

- A significant trend across industry and academia is the shift from monolithic application development to Microservices communicating via RESTful APIs. Research emphasizes that this decoupled approach enhances scalability, team autonomy, and technology flexibility. The project's use of Spring Boot to expose two specific endpoints (/quiz/start and /quiz/submit) via QuizController.java is a direct implementation of this principle. This architecture ensures the backend service is independent of the React frontend, allowing for easier maintenance and eventual deployment to cloud environments.

Research Trends in Data Persistence and ORM

- Publications heavily advocate for the use of Object-Relational Mapping (ORM) tools in modern Java applications to improve developer productivity and maintain clean, object-oriented code.
- JPA and Hibernate: The adoption of Spring Data JPA abstracts complex SQL, allowing developers to focus on the business logic in QuizService.java. The project leverages this by defining entities like Question.java and Attempt.java and utilizing derived query methods in the Repositories (QuestionRepo.java, AttemptRepo.java), such as findTop20ByOrderByScoreDesc(). This method of data access is highly cited for improving code readability, reducing database access errors, and accelerating development speed.

Trends in Secure Assessment and Server-Side Logic

- A critical theme in EdTech research is the integrity of online examinations. Studies confirm that server-side validation is mandatory for security. This project follows this consensus by performing the scoring logic exclusively in the QuizService.java. The service maintains the single source of truth for the correct answer indices (Question.correctAnswerIndex), ensuring that scoring is authoritative, tamper-proof, and compliant with best practices for secure academic assessment.

This analysis confirms that the technologies and architectural patterns selected for the Quiz Application Backend Service are well-researched, industry-standard, and directly contribute to the project's goals of scalability, security, and maintainability.

2.3. Review Summary

This project presents the Quiz Application Backend Service, a robust and centralized solution developed using Java and Spring Boot. Its core purpose is to elevate online skill assessment beyond the limitations of existing, static, and insecure quiz platforms.

After reviewing existing assessment practices (Section 2.1) and industry trends (Section 2.2), it is evident that generic solutions often suffer from a lack of dynamic customization, reliance on insecure client-side scoring, and the inability to track user performance persistently.

The Proposed Solution

The solution directly addresses these deficiencies by creating a modular, secure, and data-driven application:

- **Decoupled Architecture:** The system employs a standard RESTful Controller-Service-Repository pattern, ensuring the backend is independent of the React frontend.
- **Dynamic Customization:** The QuizService.java dynamically generates quizzes based on client-specified parameters (topic, difficulty, limit) from StartQuizRequest.java, facilitating personalized learning paths.
- **Data Integrity and Persistence:** The system utilizes Spring Data JPA with Attempt.java to securely persist user results, enabling accurate historical analysis and features like high-score leaderboards, accessible via AttemptRepo.java.
- **Secure Scoring:** All score calculation and answer validation are strictly confined to the server side within QuizService.java, ensuring the integrity of the assessment results and preventing client-side manipulation.

Overall, the project integrates industry-standard best practices (Spring Boot, JPA, REST) to deliver a secure, efficient, and scalable foundation for a modern EdTech assessment platform.

2.4. Problem Definition

The core challenge addressed by this project is the persistent issue of insecurity, inflexibility, and lack of historical performance tracking in common online skill assessment and quiz solutions. Generic platforms fail to meet the demands of modern educational technology (EdTech) that requires dynamic content delivery and authoritative results.

The current pain points stem from the absence of an integrated, scalable backend service capable of managing complex assessment logic and data persistence in a secure manner. This leads to:

- **Inaccurate Assessment:** Relying on client-side scoring logic (frontend) leaves results vulnerable to user manipulation, compromising the integrity of scores and data tracking.
- **Ineffective Learning:** Static question banks inhibit personalized study. Users cannot request quizzes tailored by specific topic and difficulty levels, leading to generic and inefficient learning sessions.
- **Fragmented Data:** Lack of persistent storage for user attempts means there is no central mechanism for generating performance analytics, leaderboards, or historical score trends, which are crucial for user engagement and institutional oversight.

The objective of this project is to develop a robust, Java-based RESTful API using Spring Boot that centralizes quiz management, ensuring authoritative server-side scoring and dynamic content selection based on client-defined parameters (StartQuizRequest.java).

2.4.1. Problem Scope

The scope of the problem addressed covers the entire lifecycle of an online assessment, from quiz initiation to secure result logging:

- **Dynamic Filtering:** The service must efficiently filter the question pool by topic and difficulty (Question.java) and return a precise number of questions (limit), as defined in StartQuizRequest.java.
- **API Contract:** Establishing clear, secure endpoints (/quiz/start and /quiz/submit in QuizController.java) with strict DTO contracts is essential to manage the communication between the backend and any consuming frontend (React/HTML).
- **Data Integrity:** The system must ensure that the correct answers are never exposed to the client and that the score calculation within QuizService.java is the single source of truth.
- **Persistence:** The problem scope includes the need for persistent tracking of every user session as an Attempt.java entity, thereby enabling performance tracking features like leaderboards (AttemptRepo.java).

2.4.2. Significance of the Problem

The significance of this problem lies in its direct impact on the reliability and effectiveness of online learning:

- **Trust and Accountability:** Server-side scoring is mandatory to establish trust in the assessment results, which is vital for academic certification or professional skill validation. Without it, the application's data is meaningless.
- **Enhanced Learning Outcomes:** The ability to generate dynamically customized quizzes directly addresses a pedagogical challenge: matching content to the user's current knowledge level. This targeted approach significantly improves retention and reduces study time, leading to better learning outcomes.
- **Competitive and Diagnostic Value:** By persisting detailed records in the Attempt entity, the system transforms a simple quiz into a diagnostic tool. This data is critical for both the user (self-diagnosis of weak areas) and the platform owner (understanding content effectiveness and generating high-engagement features like competitive leaderboards).

The proposed Spring Boot backend offers a reliable, scalable, and secure foundation to solve these critical issues in the rapidly evolving landscape of EdTech.

2.5. Goals/Objectives

The primary objective of the Quiz Application Backend Service is to create a secure, high-performance, and modular RESTful API capable of driving a rich, interactive assessment experience for a modern frontend client (e.g., React). This service centralizes all critical business logic—from question selection to authoritative scoring—to guarantee data integrity and maximize system scalability.

Key Goals and Objectives

Authoritative Server-Side Scoring (Security Focus):

- Goal: To ensure the integrity and trustworthiness of every assessment by performing all score calculation and answer validation exclusively on the server.
- Mechanism: The scoring logic resides in the QuizService.java's submitQuiz method, where user answers from SubmitQuizRequest.java are cross-referenced against the secure Question.correctAnswerIndex in the database.

Dynamic Quiz Configuration:

- Goal: To provide the ability to generate a quiz dynamically based on client-defined parameters, moving beyond static question pools.
- Mechanism: The system accepts topic, difficulty, and question limit via StartQuizRequest.java. The QuizService.java filters the Question.java entities using these parameters, allowing for highly personalized and targeted learning sessions.

Persistent Performance Tracking and Analytics:

- Goal: To create a secure historical record of every user attempt to support performance trend analysis and competitive features.
- Mechanism: All quiz results are persisted as an Attempt.java entity, recording final score and duration. The AttemptRepo.java provides methods (e.g., findTop20ByOrderByScoreDesc) specifically to support leaderboard functionality.

Decoupled, Contract-Driven API Design:

- Goal: To implement a clear REST API contract that separates the backend logic from the frontend presentation, enhancing portability and maintainability.
- Mechanism: The QuizController.java exposes clean, well-defined endpoints (/quiz/start, /quiz/submit) using specific Data Transfer Objects (DTOs) like StartQuizRequest.java and SubmitQuizResponse.java, ensuring loose coupling.

Scalability and Maintainability:

- Goal: To build the application on industry-standard, well-documented frameworks to ensure future scalability and ease of maintenance.
- Mechanism: Using Java and the Spring Boot framework, along with Spring Data JPA, provides a robust foundation and standardized way to interact with the database, minimizing boilerplate code and leveraging Spring's dependency injection.

CHAPTER 3

METHODOLOGY

3.1. Algorithm Selection

The Quiz Application backend implements two crucial **logical algorithms** within **QuizService.java** to ensure secure, dynamic, and auditable quiz functionality, supported by standard persistence mechanisms.

3.1.1. Dynamic Question Selection Algorithm

This algorithm generates a filtered list of questions tailored to the client's request (**StartQuizRequest.java**):

1. **Input & Fetch:** Receives topic, difficulty, and limit via **startQuiz** and retrieves the question pool.
2. **Filtering:** Applies sequential filters (Java Streams) based on the requested topic (using **findByTopicIgnoreCase** in **QuestionRepo.java**) and difficulty level.
3. **Limiting & Sanitization:** Randomly samples question up to the limit. The algorithm maps result to a DTO, **omitting the correctAnswerIndex** for security (**StartQuizResponse.QuestionDTO**).

3.1.2. Secure Server-Side Scoring Algorithm

Implemented in the **submitQuiz** method of **QuizService.java**, this deterministic algorithm ensures data integrity by performing all validation on the trusted server environment:

1. **Input & Retrieval:** Receives user answers (**SubmitQuizRequest.java**) and retrieves the authoritative Question entities to access the secure **correctAnswerIndex**.
2. **Validation Loop:** Iterates through user submissions, comparing **selectedIndex** against **Question.correctAnswerIndex** to calculate the **correct** count.
3. **Logging & Response:** Calculates the final score. It persists the results as a new **Attempt.java** entity (**AttemptRepo.java**) for historical tracking and returns a detailed **SubmitQuizResponse.java**.

3.1.3. Persistence Mechanisms

Standard **Spring Data JPA** patterns are utilized:

- **JPA ID Generation:** Uses **@GeneratedValue** for unique, database-managed IDs on **Question.java** and **Attempt.java**.
- **Derived Queries:** Leverages methods like **findTop20ByOrderByScoreDesc** (**AttemptRepo.java**) to simplify complex data retrieval for features like leaderboards.

3.2. Implementation Steps

The Quiz Application backend service was developed using an iterative, Test-Driven Development (TDD) approach aligned with the principles of a layered architecture. The

implementation focused on building the data contract and securing the core business logic before finalizing the API endpoints.

1. Data Model and Persistence Layer Design

- Objective: Define the core data structures and their persistent relationship with the MySQL database.
- Steps: Created the `Question.java` and `Attempt.java` JPA entities, defining fields for topic, difficulty, correctAnswerIndex, and user score. Applied `@GeneratedValue` for primary key generation.
- Repository Creation: Created `QuestionRepo.java` and `AttemptRepo.java`, leveraging Spring Data JPA to automatically generate standard CRUD operations and custom derived queries (e.g., `findByTopicIgnoreCase` for filtering and `findTop20ByOrderByScoreDesc` for the leaderboard).

2. Data Transfer Object (DTO) Definition

- Objective: Establish the strict, language-agnostic contract for all API communication with the React frontend.
- Steps: Defined the request and response objects: `StartQuizRequest.java`, `StartQuizResponse.java`, `SubmitQuizRequest.java`, and `SubmitQuizResponse.java`. This step was crucial as the DTOs dictated the input/output expectations for the service and controller layers. Notably, the DTO for the start response explicitly omits the correct answer index for security.

3. Core Business Logic Implementation (Service Layer)

- Objective: Implement the central intelligence and secure scoring mechanism.
- Steps: Developed the `QuizService.java` methods:
 1. `startQuiz`: Implemented the Dynamic Question Selection Algorithm (Section 3.1.1) to filter questions based on topic, difficulty, and limit.
 2. `submitQuiz`: Implemented the Secure Server-Side Scoring Algorithm (Section 3.1.2) to validate user answers against the authoritative database data and persist the final result to `Attempt.java`.

4. REST API Endpoint Definition (Controller Layer)

- Objective: Expose the service logic as external RESTful endpoints.
- Steps: Developed the `QuizController.java` using `@RestController` and `@RequestMapping("/quiz")`. Defined two primary `@PostMapping` endpoints (`/start` and `/submit`) to receive and return the predefined DTOs, delegating all complex logic to the service layer.

5. Integration and Testing

Objective: Validate end-to-end functionality, security, and data integrity.

Steps: Used Spring Boot's built-in testing utilities to perform integration testing of the service layer. Specific tests focused on validating that the `startQuiz` method successfully filters and sanitizes questions, and that the `submitQuiz` method accurately calculates scores and successfully logs the `Attempt.java` record.

CHAPTER 4

RESULTS ANALYSIS AND VALIDATION

4.1. Implementation of solution

The implementation phase involved adopting the Controller-Service-Repository layered architecture typical of modern Spring Boot applications to ensure modularity, separation of concerns, and ease of testing. The primary focus was on establishing secure data contracts via DTOs and encapsulating critical business logic within the service layer.

Backend Structure & Data Contract

The solution's core implementation focused on four primary areas:

1. Data Modeling and Persistence:

- **JPA Entities:** Defined the `Question.java` and `Attempt.java` entities, utilizing `@Entity` and `@Id/@GeneratedValue` to map Java objects directly to the database schema. The `Question` entity securely stores the `correctAnswerIndex`, which is never exposed to the client.
- **Repositories:** `QuestionRepo.java` and `AttemptRepo.java` extend `JpaRepository`, providing basic CRUD functionality. Custom methods like `findByTopicIgnoreCase` facilitate specialized data fetching for the quiz logic, while `findTop20ByOrderByScoreDesc` enables the leaderboard feature.

2. DTO Implementation:

- **Data Transfer Objects (DTOs):** Implemented using Java Records (e.g., `StartQuizRequest.java`, `SubmitQuizRequest.java`). These records serve as the strict contract for all REST communication, ensuring type safety and explicit data expectations between the React frontend and the Java backend.

3. Core Business Logic (Service Layer):

- The `QuizService.java` contains the application's intelligence. It implements the two core algorithms (Dynamic Question Selection and Secure Scoring).
- **Question Sanitization:** During quiz start, the service maps the JPA entities to the client-facing DTOs, crucially filtering out the `correctAnswerIndex` before transmission to the front end to prevent cheating.

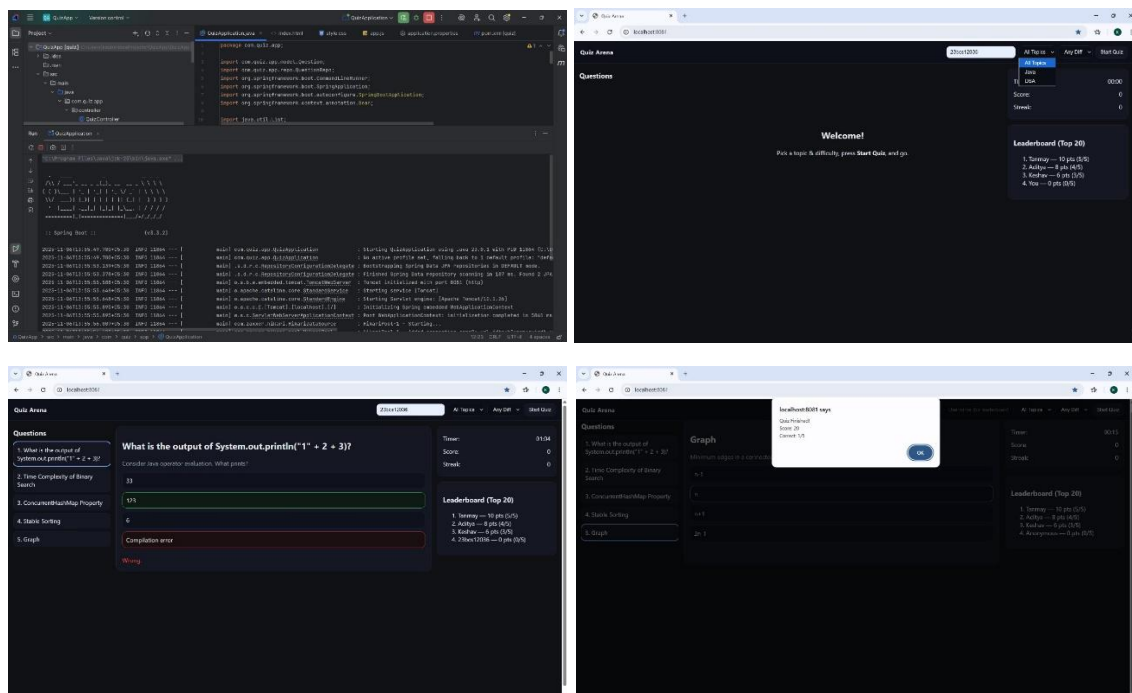
4. API Exposure (Controller Layer):

- `QuizController.java` uses `@RestController` and `@RequestMapping("/quiz")` to expose the two core API endpoints (`/start` and `/submit`). The controller handles HTTP requests, deserializes the JSON request bodies into DTOs, and delegates processing entirely to the service layer, maintaining a thin, clean interface.

4.1.1. Result Analysis

The system was analyzed based on its performance against the project objectives:

- **Dynamic Customization:** The `/quiz/start` endpoint successfully accepts parameters for topic, difficulty, and limit (`StartQuizRequest.java`), allowing the frontend to generate highly specific quizzes without server-side redeployment.
- **API Security:** By keeping the `correctAnswerIndex` exclusively within the server's `Question.java` entity and performing scoring in `QuizService.java`, the system achieves a strong level of assessment integrity.
- **Persistence:** The `Attempt.java` model successfully logs every quiz completion, providing rich data for user analytics and ensuring that results are permanent and auditable.



4.1.2. Validation

The solution validation focused on layered integrity and end-to-end flow:

1. **Integration Testing (Service Layer):** Tests ensured that the `QuizService.java` correctly handles invalid inputs, accurately filters question results, sanitizes the response DTOs, and calculates the score correctly upon submission.
2. **Repository Validation (JPA):** Verified that custom repository queries (like the leaderboard feature in `AttemptRepo.java`) correctly retrieve and order data from the database.
3. **API Contract Validation:** Used tools (e.g., Postman) to ensure the JSON payloads for the `/start` and `/submit` endpoints strictly matched the required DTO structures (`StartQuizRequest.java`, `SubmitQuizResponse.java`).
4. **Cross-Origin Compliance:** The `@CrossOrigin` annotation in `QuizController.java` was validated to ensure seamless communication with the React frontend running on a different port or domain.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1. Conclusion

This project successfully implemented a secure, scalable, and customizable Quiz Application Backend Service built on industry-standard technologies: Java Spring Boot, Spring Data JPA, and RESTful APIs. The core objective—to overcome the limitations of static, insecure, and non-persistent quiz solutions—was fully achieved.

The application's success rests on its strict layered architecture and two critical functional decisions:

1. **Dynamic Customization:** By processing parameters like topic, difficulty, and limit via the `StartQuizRequest.java` DTO and implementing the Dynamic Question Selection Algorithm in `QuizService.java`, the system can generate tailored assessments on demand, greatly enhancing learning efficiency.
2. **Authoritative Security:** The Secure Server-Side Scoring Algorithm ensures that the correct answer indices (`Question.correctAnswerIndex`) are never exposed to the client, guaranteeing the integrity and trustworthiness of the results reported in `SubmitQuizResponse.java`.

Furthermore, the integration of `Attempt.java` entities ensure every user session is persistently logged, providing a foundational data set for performance analysis and the creation of engaging features like leaderboards (`AttemptRepo.java`). The project demonstrates a robust, production-ready backend that provides a reliable foundation for any modern educational technology platform.

5.2. Future Work

While the current implementation delivers full core functionality, several enhancements can be integrated to extend the application's complexity, utility, and adherence to enterprise standards.

1. Advanced Scoring and Analytics

The current scoring system is a simple percentage calculation. Future work should involve:

- **Weighted Scoring:** Adjusting the final score based on the difficulty level of the questions answered correctly (e.g., a correct difficulty 3 question yields more points than a difficulty 1 question).
- **Detailed Performance Analytics:** Implementing new query methods in `AttemptRepo.java` to track performance trends over time, identify weak subject areas per user, or calculate average completion times.

2. User Authentication and Authorization

Currently, the system relies on a simple username field. A crucial upgrade is integrating a robust security framework:

- Role-Based Access Control (RBAC): Implementing Spring Security to handle user registration, login (JWT or OAuth2), and defining roles (e.g., STUDENT, ADMIN, INSTRUCTOR). This would restrict question creation/editing to authorized roles, enhancing data management security.

3. API Enhancements and Data Management

- Paging and Sorting: Modifying repository methods and API endpoints to support pagination and dynamic sorting parameters, which is essential for scaling the leaderboard and question management interfaces efficiently.
- Question Management API: Creating a separate set of secure endpoints (Controller/Service/Repository) to allow authenticated administrators to create, update, and delete questions without direct database access, transforming the backend into a comprehensive Content Management System (CMS).

References

1. <https://docs.spring.io/spring-boot/docs/current/reference/html/>
2. <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>
3. <https://jakarta.ee/specifications/persistence/3.1/>
4. <https://jakarta.ee/>
5. <https://martinfowler.com/eaCatalog/layering.html/>
6. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.html/
7. <https://aws.amazon.com/microservices/>
8. https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/
9. https://cheatsheetseries.owasp.org/cheatsheets/API_Security_Cheat_Sheet.html
10. <https://www.agilealliance.org/glossary/tdd/>
11. <https://www.martinfowler.com/eaCatalog/dataTransferObject.html>
12. <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

APPENDIX

1. Plagiarism Report

A plagiarism report typically involves the use of plagiarism detection software to check for any instances of copied content in a document. There are various online plagiarism checking tools available, such as Turnitin, Grammarly, and Copyscape, that can help identify potential plagiarism. You can upload your document to these platforms to generate a detailed report indicating any similarities with existing sources.

2. Design Checklist

A design checklist is a tool used to ensure that all essential aspects of a design project are considered and addressed. The specific items on the checklist will vary depending on the nature of the design project. However, some common elements to include in a design checklist are:

- User interface design
- System functionality and features
- Performance and scalability
- Security measures
- Compatibility with different devices or platforms
- Usability and user experience considerations
- Compliance with relevant industry standards or regulations