Tribhuvan University

Faculty of Humanities and Social Science

A Project Report On

"Quiz Management System with Fisher-Yates Shuffle Algorithm"

In partial fulfillment of requirements for Bachelor in Computer Applications

Submitted by:

Lusana Shakya (6-2-410-128-2021)

Ashad, 2082 B.S.

Under the Supervision of

Mr. Ananda K.C.

Tribhuvan University

Faculty of Humanities and Social Science

A Project Report On

"Quiz Management System with Fisher-Yates Shuffle Algorithm"

In partial fulfillment of requirements for Bachelor in Computer Applications

Submitted by:

Lusana Shakya (6-2-410-128-2021)

Ashad, 2082 B.S.

Under the Supervision of

Mr. Ananda K.C.

Tribhuvan University

Faculty of Humanities and Social Science

Prime College

SUPERVISORS MANAGEMENT

It is my pleasure to recommend that a project report on "Quiz Management System" has been prepared under my supervision by Lusana Shakya in partial fulfillment of the requirement of the degree of Bachelor of Computer Applications. Her report is satisfactory and is an original work done by her to process for the future evaluation.

……

Mr. Ananda K.C.

SUPERVISOR

Lecturer, Department of IT

Prime College

Tribhuvan University

Faculty of Humanities and Social Science

Prime College

LETTER OF APPROVAL

This is to certify that this project report, prepared by Lusana Shakya on "Quiz Management System with Fisher-Yates Shuffle Algorithm" in partial fulfillment of the requirements for the degree of Bachelor in Computer Application, has been evaluated.

In our opinion, it is satisfactory in the scope and quality as a project for the required degree.

…….. **Mr. Ananda KC** **Supervisor** **Prime College** **Khusibu, Nayabazar, Kathmandu**	…….. **Ms. Rolisha Sthapit** **Program Co-Ordinator/ Internal Examiner** **BCA Department** **Prime College** **Khusibu, Nayabazar, Kathmandu**
……..	…….. **Mr.** **External Examiner**

ACKNOWLEDGEMENT

This project on building a Quiz Management System, which provides personalized quizzes using the Fisher-Yates Shuffle algorithm, would not have been possible without the help and support of many people. I am deeply grateful to Mr. Ananda K.C., my Project Supervisor, and Coordinator Er. Rolisha Sthapit for their unwavering guidance and support throughout the course of this project. Their knowledge and encouragement helped me move in the right direction and successfully complete the system. I would also like to thank my friends, teachers, mentors, and classmates who supported me, provided valuable feedback, and helped improve the project. Their insights made the system better and more useful for real users. I truly appreciate everyone's support and am thankful for their contributions in making this project a success.

-Lusana Shakya

ABSTRACT

These days, many people enjoy playing quizzes tailored to their interests, such as selecting the number of questions and preferred categories. To enhance this experience, this project introduces a Quiz Management System that allows users to play quizzes dynamically and track their results through a personalized dashboard. The system uses the Fisher-Yates Shuffle algorithm to randomize the order of quiz questions, ensuring a fair and unpredictable quiz experience every time. Users can customize their quiz by choosing the number of questions they want to attempt, selecting specific categories that interest them, and selecting the difficulty level. The system then presents the randomized questions accordingly, and upon completion, users can immediately view their scores and detailed results in the dashboard. Developed using Node.js for the backend and ReactJS for the frontend, the system provides a seamless and interactive interface accessible on both web and mobile platforms. This allows quiz enthusiasts to engage with quizzes that fit their preferences while enabling administrators to manage quiz content efficiently. The system has demonstrated effectiveness in delivering engaging quizzes with accurate result tracking, and future improvements will include the launch of a dedicated mobile quiz app for both Android and iOS platforms, enabling users to participate in quizzes. Additionally, features like real-time multiplayer quizzes, voice-based questions are envisioned to boost user engagement. Overall, the Quiz Management System offers an enjoyable, flexible, and user-centered platform for quiz playing and performance monitoring.

Keywords: Quiz Management System, Fisher-Yates Shuffle, *Dashboard*

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
HTML	Hypertext Markup Language
JSON	JavaScript Object Notation
Node.js	JavaScript runtime for server-side apps
React JS	JavaScript framework for building web apps

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1. Introduction

In today's fast-growing digital learning and entertainment landscape, quizzes have become a popular tool for both education and engagement. As more users look for interactive ways to test their knowledge or have fun, providing personalized and dynamic quiz experiences is essential. The Quiz Management System is designed to address this need by offering a customizable and intelligent platform where users can select their quiz preferences, play randomized quizzes, view their results through a dashboard, and also view their history.

The system allows users to choose their desired number of questions and select specific categories, making the experience tailored to individual interests. To ensure each quiz session is unique and fair, the platform utilizes the Fisher-Yates Shuffle algorithm, which efficiently randomizes the order of questions before presenting them to the user. This prevents repetition and ensures that no two quiz sessions are exactly the same. Additionally, the quiz management system incorporates AI-powered difficulty levels, allowing users to select a challenge that matches their knowledge and experience, ranging from easy to hard. This intelligent feature dynamically adjusts question complexity, enhancing personalization and engagement. To further increase focus and challenge, each quiz is equipped with a timer, which varies based on difficulty level. This ensures time-bound answering, reduces distractions, and keeps users engaged throughout the session.

Built using Node.js for the backend and ReactJS for the frontend, the application offers a smooth interface suitable for web users. It simplifies quiz management for administrators by allowing easy categorization. Users can choose one category among other categories and then choose the number of questions in the quiz.

The demand for personalized quiz platforms is increasing as users seek meaningful and engaging experiences. This Quiz Management System not only reduces manual effort in selecting and organizing quizzes but also supports better user engagement through customization and real-time scoring. It serves as an effective solution for educational institutions, training platforms, or general entertainment, and sets the foundation for future features like time-based quizzes, performance analytics, and leaderboards.

In essence, this system streamlines the process of quiz creation and participation, ensuring an enjoyable, efficient, and personalized quiz experience for all users.

1.2. Problem Statement

The Quiz Management System tries to solve the following problems:

- Most systems offer limited personalization, failing to cater to different user preferences, like topic or question count.
- Fairness and variety in question delivery are often lacking, especially when users attempt the same quiz multiple times.
- Manual quiz creation is time-consuming for multiple quizzes and users.

1.3. Objectives

The following are the objectives of the project:

- To enable users to generate quizzes based on their selected quiz category.
- To enable users to select their level of difficulty.
- To display accurate results after quiz completion.
- To provide a dashboard that shows quiz results and maintains a history of past attempts.
- To enable the admin to view and delete user accounts.
- To set different time limits depending on the quiz difficulty.

1.4. Scope and Limitations

1.4.1. Scope

The following are the scopes of the project:

- The system is designed to create quizzes for users based on their choice of category.
- It uses the Fisher-Yates Shuffle algorithm to shuffle quiz questions.
- In this system, the user can choose their difficulty level, ensuring a quiz experience for all levels of users.
- The system will be available as a simple and user-friendly application to help users.

1.4.2. Limitations

The following are the limitations of the project:

- The system is limited to presenting only multiple-choice questions (MCQs).
- It does not currently support features like timed quizzes and negative marking.
- Quiz suggestions are based on pre-entered question banks; newly added quizzes may not appear unless the data is updated.

1.5. Development Methodology

The Prototype Model is a software development method that emphasizes creating a simplified version of the system early in the process. This early model or prototype allows users and stakeholders to see how the system might work, provide feedback, and refine it before full-scale development begins. It's especially helpful when requirements are unclear at the start. In the context of the Quiz Management System, this model ensures that the system aligns with user expectations through continuous iterations.

Requirement Gathering: In the first phase of the Prototype Model, initial requirements for the Quiz Management System are gathered. At this point, the focus is on understanding the basic expectations of users, such as the need for a login system, the ability to select quiz categories, answer questions, and receive scores at the end. Admin-side requirements may include managing users, quiz categories, and questions.

Quick Design: Once the basic requirements are identified, a quick design of the system is created. This design doesn't go into deep technical details but instead focuses on how the system will appear and function from the user's perspective. For the Quiz Management System, this may include wireframes or simple screens for login, dashboard, quiz interface, and an admin panel. It also outlines the basic navigation between pages and how users will interact with the system.

Build Prototype: Using the quick design, a working prototype is built with limited functionalities. This early version of the Quiz Management System may allow users to select a quiz category, answer a few sample questions, and view a mock score at the end. It also includes a basic admin interface to add or view quiz questions. While the prototype may not be connected to a database or include features like a timer or authentication at this point, it gives a feel of the system's core idea.

User Evaluation: After building the prototype, it is presented to end-users and stakeholders for feedback. In the case of the Quiz Management System, students, teachers, or supervisors interact with the prototype and provide their opinions on its usability and functionality. For example, they might suggest adding features like adding difficulty level and, history of past attempts of the users.

Refining Prototype: Based on the feedback collected, the prototype is revised and improved. This stage may go through multiple cycles of feedback and enhancement to meet user expectations. For the Quiz Management System, this might involve implementing navigation between questions, real-time score calculation, better UI/UX design, integration with a sample database, and refining user roles (like admin, student).

Implement and Maintain: After the system is deployed, it enters the maintenance phase. During this phase, the system is monitored for bugs, performance issues, and changing user needs. Updates may include adding new quiz categories, fixing glitches in score calculations, enhancing security measures, or including new features like leaderboards. Regular maintenance ensures that the Quiz Management System stays relevant, secure, and efficient over time as user expectations and technologies evolve.

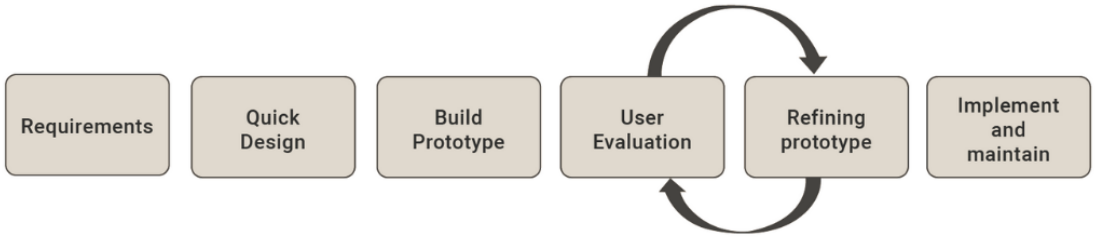

Figure 1.1: Prototype Model [9]

1.6. Report Organization

The report on the Quiz Management System is organized into five chapters, each addressing a crucial aspect of the project. This structured approach provides a comprehensive overview of the system's development, implementation, and evaluation.

Here's a summary of each chapter:

The first chapter introduces the Quiz Management System, offering a detailed overview of its purpose, scope, and importance. It answers key questions such as what the system aims to achieve, how it works, and why it is useful in the context of quizzes. The chapter sets the

foundation for understanding the project's goals and its impact on improving the user quiz experience.

Likewise, in the second chapter, the report reviews related work in the area of quiz Management systems and the use of artificial intelligence. It presents a study of existing systems, highlighting commonly used methods, algorithms, and their outcomes. This literature review helps to identify gaps and explains how the proposed system contributes new value to the field of quiz management.

The third chapter provides a detailed analysis and design of the Quiz Management System. It includes system requirements, feasibility study, and various design diagrams such as use case, class, sequence, activity, and state diagrams. This chapter also explains the use of the Fisher-Yates Shuffle algorithm for providing personalized quiz suggestions. Additionally, it evaluates technical and operational feasibility and outlines the system architecture.

The fourth chapter focuses on the development and testing of the Quiz Management System. It describes the tools and technologies used, such as ReactJS for frontend, Node.js for backend, and MongoDB for database. The chapter explains how the system processes user inputs and recommends quizzes using the algorithm. It also includes code snippets, data visualization, and test cases to validate the performance and accuracy of the system.

The fifth chapter summarizes the overall findings of the project. It evaluates the effectiveness of the Management system and the role of the selected algorithm in improving quiz suggestions. The chapter provides insights into the system's success and offers suggestions for future improvements, such as using more advanced algorithms, expanding the dataset, and adding features like live tracking and real-time management.

This report structure ensures a complete presentation of the Quiz Management System, from concept and research to design, implementation, and future development. Each chapter is designed to provide detailed, clear information to help readers understand the full scope and results of the project.

CHAPTER 2

BACKGROUND STUDY AND LITERATURE REVIEW

2.1. Background Study

The increasing integration of digital platforms in education has led to the development of intelligent systems for managing and evaluating student performance. Among these, Quiz Management Systems (QMS) have emerged as effective tools for conducting assessments, providing immediate feedback, and monitoring learner progress. These systems allow educators to create, store, and manage quizzes while enabling students to access and attempt them conveniently through web or mobile interfaces.

One of the critical aspects of an efficient QMS is randomization—the ability to present questions in a unique order for each attempt or user. This feature is essential for minimizing cheating, maintaining fairness, and ensuring that assessments reflect individual understanding. Traditional static quiz formats can lead to repetitive patterns and memorization, which may compromise the integrity of evaluations. Therefore, implementing a robust algorithm for randomizing questions and answer choices is crucial.

The Fisher-Yates Shuffle Algorithm, also known as the Knuth Shuffle, provides an optimal solution for this requirement. It is a well-known algorithm for generating a random permutation of a finite sequence—in this case, quiz questions or options. The algorithm operates in linear time complexity (O(n)), ensuring efficiency even for large datasets. Its unbiased nature ensures that every possible ordering of elements is equally likely, making it ideal for educational applications where fairness is critical. [1]

The integration of the Fisher-Yates Shuffle algorithm within the QMS architecture ensures a dynamic and fair assessment environment. This algorithm works well across various programming platforms and can be easily implemented on both the client-side (e.g., JavaScript) and server-side.

Moreover, combining the Fisher-Yates Shuffle with backend logic and user session tracking allows the system to maintain quiz integrity while also recording and analyzing student performance metrics. These features collectively support personalized learning paths and adaptive assessments, aligning with modern educational goals.

In conclusion, the Fisher-Yates Shuffle algorithm plays a vital role in enhancing the functionality and fairness of a Quiz Management System. Its ability to efficiently and

uniformly randomize quiz components makes it a preferred choice in educational software development. Ensuring variability and reducing predictability helps maintain academic integrity and supports a more accurate evaluation of student understanding.

2.2. Literature Review

This study applies the Fisher-Yates Shuffle algorithm for the randomization of quiz questions, as demonstrated in previous research. The Fisher-Yates algorithm efficiently performs the randomization process, supporting problem training applications and improving question paper variability. Black Box Testing was employed to verify the functionality of the application according to specifications, ensuring the system meets user requirements [1].

The objective of improving automated examination processes has led to the development of systems like the Ad-Hoc Question Paper Application (AQPA) at UiTM Perlis Branch. This system utilizes the Fisher-Yates algorithm to generate randomized test papers for university exams, limited to one-time randomization and validated by faculty experts [2].

Enhancements in question paper quality are supported by combining randomization algorithms with Text Mining techniques to remove duplicate questions, thereby maintaining the uniqueness and integrity of the papers. The generated papers are typically formatted in editable Word documents to facilitate further customization by educators [3].

Research also indicates that automated question randomization and arrangement can improve the fairness and unpredictability of exam content. For instance, a study showed that randomizing visual elements in educational games using similar algorithms resulted in fair play and dynamic user experiences, validated through small-scale Black Box Testing [4].

In addressing challenges associated with manual question paper creation, other systems have been developed using web-based platforms such as ASP.Net, integrating fuzzy logic and the apriori algorithm to generate question papers from existing databases. These systems undergo rigorous testing and demonstrate reliable performance and functionality [5].

Moreover, modern automatic question generation systems often include essential features such as user administration, subject and topic selection, difficulty level specification, and

comprehensive paper management modules. These components enable educators and administrators to easily manage question banks, customize exam parameters, and oversee the entire examination lifecycle. Such modular designs contribute to more efficient, streamlined, and flexible exam paper creation and maintenance processes, aligning well with institutional requirements [7].

In addition to algorithmic randomization, some advanced quiz systems leverage Artificial Intelligence (AI) and linguistically analyzed corpora to generate questions automatically. A notable example is the Arik system, which is specifically designed for generating Basque language test questions. This system utilizes a morphologically and syntactically analyzed database of sentences represented in Extensible Markup Language (XML) format. By combining linguistic analysis with NLP techniques, Arik can formulate linguistically accurate and contextually appropriate questions. This illustrates the potential of combining computational linguistics with quiz management to develop specialized and high-quality assessment tools [8].

CHAPTER 3

SYSTEM ANALYSIS AND DESIGN

3.1. System Analysis

3.1.1. Requirement Analysis

3.1.1.1. Functional Requirements

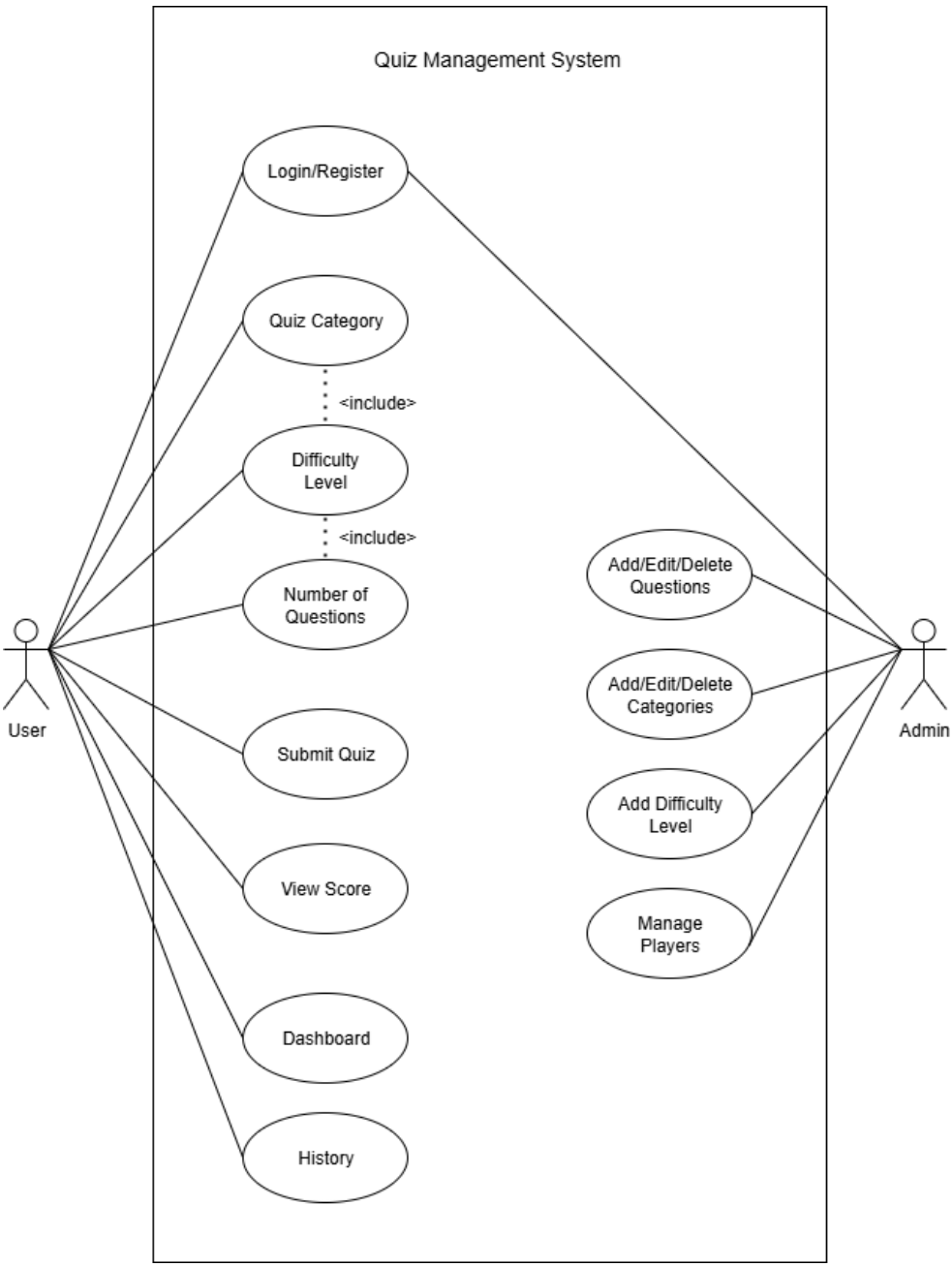

Figure 3.1: Use Case diagram for Quiz Management System

User-Side Use Cases

1. Login/Register

This use case allows users to create an account or log into the system. Registration captures essential details to identify users, and login ensures that only authorized users can access quiz functionalities. It also sets up the foundation for personal tracking of performance and secure access.

2. Quiz Category

Once logged in, users can select a quiz category of their interest. This helps them take quizzes in specific domains such as Science, History, Technology, or General Knowledge. Category selection filters the question bank accordingly, making the quiz relevant and personalized.

3. Difficulty Level

When a user decides to take a quiz, they are given the option to select a preferred difficulty level, such as *Easy*, *Medium*, or *Hard*. This allows users to tailor their quiz experience based on their knowledge or comfort level. Once a difficulty level is selected along with a category, the system filters the questions and presents only those that match the chosen criteria.

4. Number of Questions

After selecting their difficulty level, users are asked to choose how many questions they want in the quiz. This adds flexibility, allowing users to engage in short quizzes (like 5 questions) or longer ones, depending on their preference or time availability.

5. Submit Quiz

Once the user finishes answering all the questions, they can submit the quiz. This use case sends the user's responses to the system, which then evaluates the answers and prepares the final score. It acts as the transition between answering and reviewing performance.

6. View Score

Once the quiz is submitted, the system instantly calculates and displays the user's score. The result section typically includes the total score, number of correct and incorrect answers, to help users understand their performance and areas for improvement.

7. Dashboard

The dashboard provides users with a personal overview of their quiz activity. It displays total quizzes taken, average scores, highest scores, and history. The dashboard helps users track their learning progress and motivates them to improve over time.

8. History

The history section maintains a complete record of the user's past quiz attempts. It shows details like quiz categories, scores, attempt dates, and the number of tries. This feature helps users review their past performances, retake quizzes if desired, and monitor their long-term progress.

Admin-Side Use Cases

8. Login

The admin uses this functionality to securely access the backend of the system. Only verified admins are allowed to manage the questions, categories, and view user data. It protects sensitive system features from unauthorized access.

9. Add/Edit/Delete Questions

This core administrative feature allows the admin to maintain the quiz database. They can add new questions, modify existing ones if there are errors, or delete outdated or irrelevant questions. It ensures the question set stays updated and accurate.

10. Add/Edit/Delete Categories

Admins can also manage the classification of questions through categories. They can create new categories to introduce new quiz topics, rename existing ones, or remove unused categories. This helps in organizing the quiz content efficiently.

11. Add Difficulty Level

The difficulty level is managed during the process of adding or editing quiz questions. While creating a new question or updating an existing one, the admin is required to assign it a difficulty level. This helps categorize questions effectively in the system, ensuring that when users select a specific level, they receive relevant questions according to their chosen level of difficulty.

12. Manage Players

The admin can view and manage user accounts through this feature. This includes viewing player profiles, monitoring quiz participation, tracking scores, and, if necessary, deleting user accounts. Managing players helps the admin maintain the integrity and security of the platform.

3.1.1.2. Non-Functional Requirements

1. User Friendly:

The Quiz Management System must provide a smooth and intuitive user experience. Both users and admins should be able to navigate the system without confusion. Clear layouts, straightforward buttons, and easy-to-read content will enhance overall usability.

2. Simple and Easy to Use:

The system should be simple enough for users of all technical backgrounds to operate without any training. The quiz-taking process, category selection, and question submission should all follow a clear and logical flow.

3. Easy Access:

The system should be accessible from any standard web browser without requiring complex installations or configurations. Users should be able to access the system from desktops, laptops, tablets, or smartphones with a stable internet connection.

4. Responsive Design:

The system must be fully responsive, adapting to various screen sizes and resolutions. Whether accessed from a mobile device or a desktop, the layout and functionality should remain consistent and effective.

5. Performance and Speed:

The system should load quickly and perform smoothly during all operations, whether loading questions, submitting answers, or displaying scores. Efficient use of resources and optimized queries are essential for maintaining good performance.

3.1.2. Feasibility Analysis

3.1.2.1. Technical Feasibility

The Quiz Management System is technically feasible due to its simple architecture, efficient algorithms, and use of widely adopted web technologies. It integrates the Fisher-Yates Shuffle algorithm, which is a lightweight and optimal method for randomizing quiz questions, ensuring unique and fair quiz attempts. The backend is developed using Node.js, with MongoDB as the database to store questions, categories, and quiz results. These technologies are well-suited for real-time data handling and can efficiently manage dynamic quiz content. On the frontend, ReactJS is used to build an interactive, responsive, and user-friendly web interface. The system operates through RESTful APIs, enabling smooth communication between the frontend and backend. The use of modular, open-source tools ensures the system performs effectively on standard machines or cloud servers, making it scalable and adaptable for increasing numbers of users and quiz data.

3.1.2.2. Operational Feasibility

The Quiz Management System ensures operational feasibility through its intuitive design and simple navigation flow, making it easy for users of all backgrounds to take quizzes and view results. Admins can seamlessly manage quiz content, including adding, editing, or deleting questions and categories. The system supports instant quiz submission and result generation, ensuring minimal delays in the user experience. Built using JavaScript-based technologies (Node.js and ReactJS), the system is modular and easy to maintain. Additionally, the system can be expanded to include advanced features like user authentication, leaderboard tracking without requiring a major overhaul.

3.1.2.3. Economic Feasibility

The Quiz Management System is cost-effective to build and maintain due to its reliance on open-source technologies like Node.js, Express.js, MongoDB, and ReactJS, eliminating licensing costs. These tools require only basic hardware and can be hosted on budget-friendly cloud platforms such as Heroku, Vercel, or MongoDB Atlas. For academic or small-to-medium scale deployments, the infrastructure cost remains minimal. Moreover, the lightweight Fisher-Yates algorithm does not require computationally intensive processing, reducing the need for high-end servers.

In addition, AI is integrated into the system to assist in selecting appropriate difficulty levels for users based on their past performance or input, ensuring that users of all skill levels—

from beginners to advanced—can engage meaningfully with the quizzes. This personalization increases user satisfaction and retention.

In terms of return on investment (ROI), the system adds significant value by enhancing learning experiences, supporting self-assessment, and providing a platform that can be monetized or integrated into larger educational services or training tools.

3.1.2.4. Schedule Feasibility

- **Brainstorming**: The initial brainstorming phase is scheduled from March 18 to March 24 (7 days). During this period, core system objectives, user stories, and requirements were identified. The timeline is suitable for team discussions and requirement gathering.

- **Design**: The design phase runs from March 25 to April 10 (17 days). This includes preparing UI/UX wireframes, database schema designs, and outlining system flowcharts. The time allocated is appropriate for laying a strong foundation before development.

- **Development**: Development is planned from April 11 to May 5 (25 days). This phase involves building frontend components, setting up backend routes and database integration, and implementing the Fisher-Yates Shuffle logic. The duration is realistic for a focused scope with agile iteration.

- **Testing**: Testing is scheduled from May 6 to May 9 (4 days). This period focuses on unit testing, usability testing, bug fixing, and performance checks. With prior integration of testing during development, the schedule is feasible.

- **Deployment:** Deployment will occur between May 10 to May 13 (4 days). It involves deploying the system to a live environment, ensuring proper configuration, and monitoring real-time behavior. This schedule supports a smooth and controlled release.

- **Documentation**: Documentation will be created from March 22 to May 25 (60 days). This includes technical documentation, user manuals, admin guides, and final project reporting. The time is sufficient to produce quality, maintainable documentation.

Table 3.1: Schedule Feasibility

Task	Start Date	End Date	Days to Complete
Brainstorm	18 March	24 March	7
Design	25 March	10 April	17
Development	11 April	5 May	25
Testing	6 May	9 May	4
Deployment	10 May	13 May	4
Documentation	22 March	25 May	60

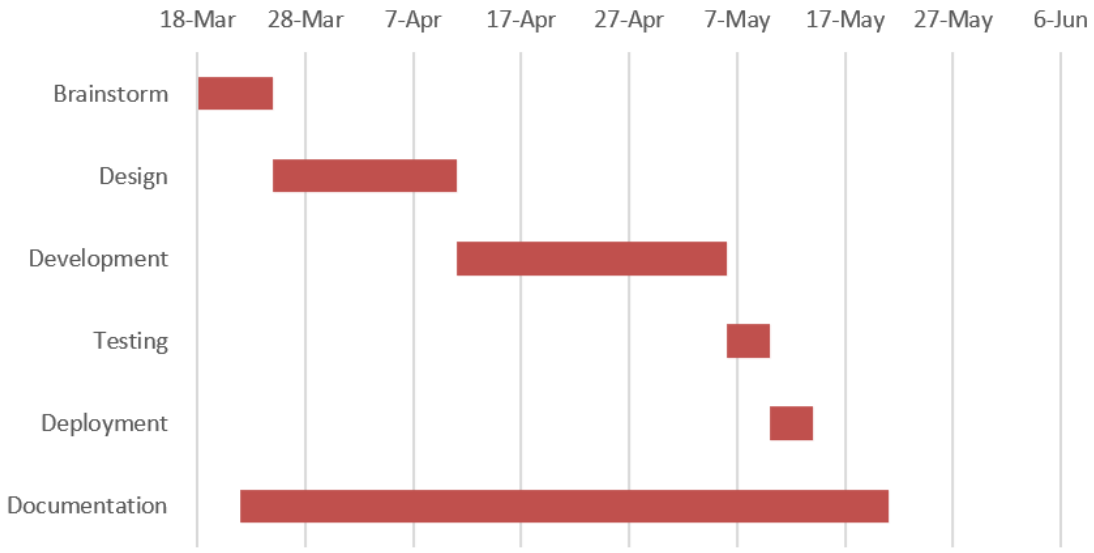

Figure 3.2: Gantt Chart for Quiz Management System

3.1.3. Object Modeling using Class Diagram

The class diagram illustrates a well-structured Quiz Management System composed of several core components, each with a distinct responsibility. At the center is the Quiz Result class, which records the outcome of quiz attempts, including scores, selected answers, and accuracy. Supporting this are the User, Question, and Category classes, where User manages player information and authentication, Question stores quiz content along with correct answers, and Category groups questions by topic. The system captures detailed data for each quiz attempt, linking users to their performance and responses. This modular structure ensures efficient quiz tracking, performance analysis, and organized content management, making the system scalable, maintainable, and easy to use.

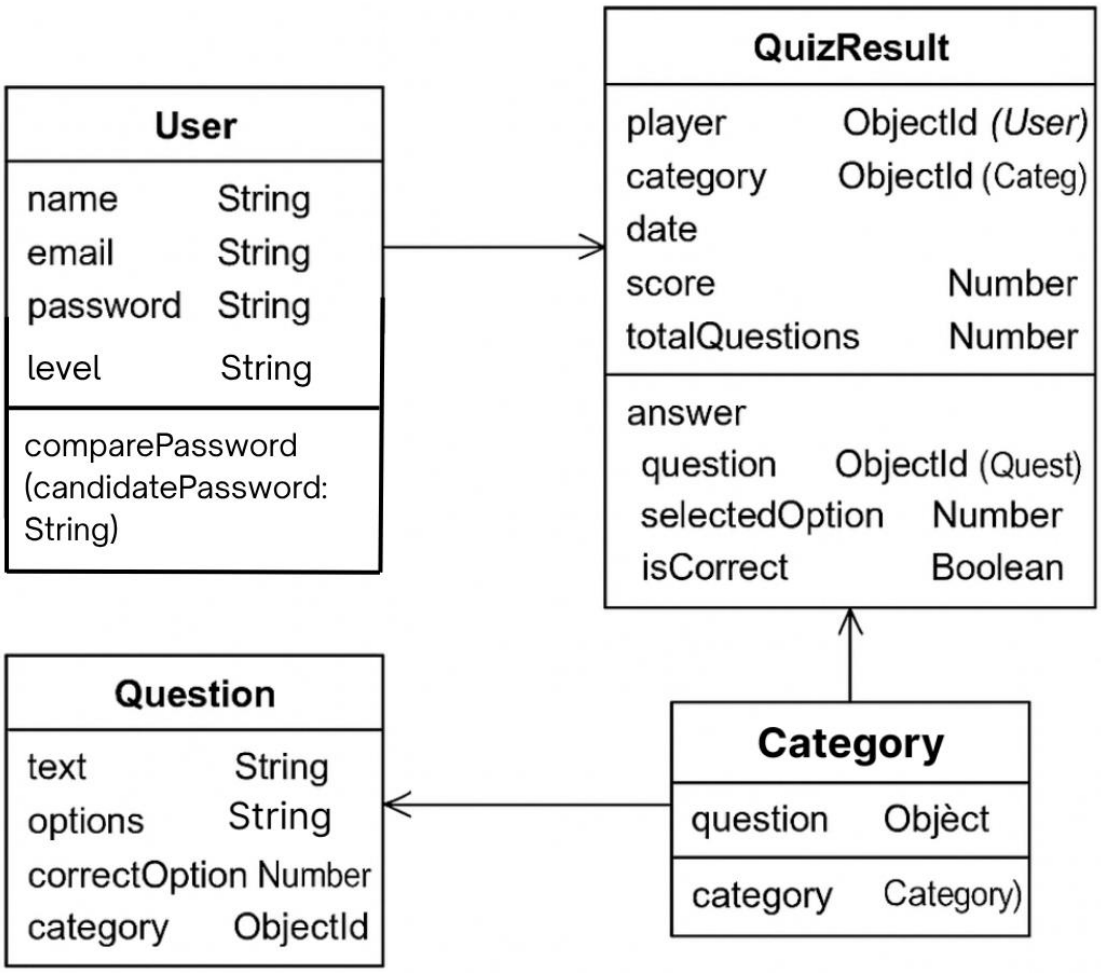

Figure 3.3: Class Diagram for Quiz Management System

Figure 3.3 represents a Quiz Application System with four main entities: User, Question, Category, and QuizResult. Users have basic authentication details and can take multiple quizzes, which are recorded in the QuizResult class along with score, date, and selected answers. Each quiz result links to a specific category and stores individual question responses, including whether the selected option was correct. Users can select the level of difficulty they want, ranging from easy to hard. Questions belong to categories and include the question text, multiple options, and the correct answer. The design ensures organized quiz management, performance tracking, and category-based question grouping.

3.1.4. Dynamic Modelling using State and Sequence Diagrams

The state diagram of the Quiz Management System illustrates a dynamic workflow that begins with user login or registration, followed by validation of credentials. Based on the user's role, the system branches into different states: regular users proceed to take quizzes,

answer questions, and submit responses to receive their results, while admins transition to quiz management functionalities, including creating, editing, and publishing quizzes. Admins can also monitor user performance and analyze quiz outcomes. This diagram effectively demonstrates how the system adapts dynamically to user roles and actions, ensuring smooth handling of both quiz participation and administrative tasks.

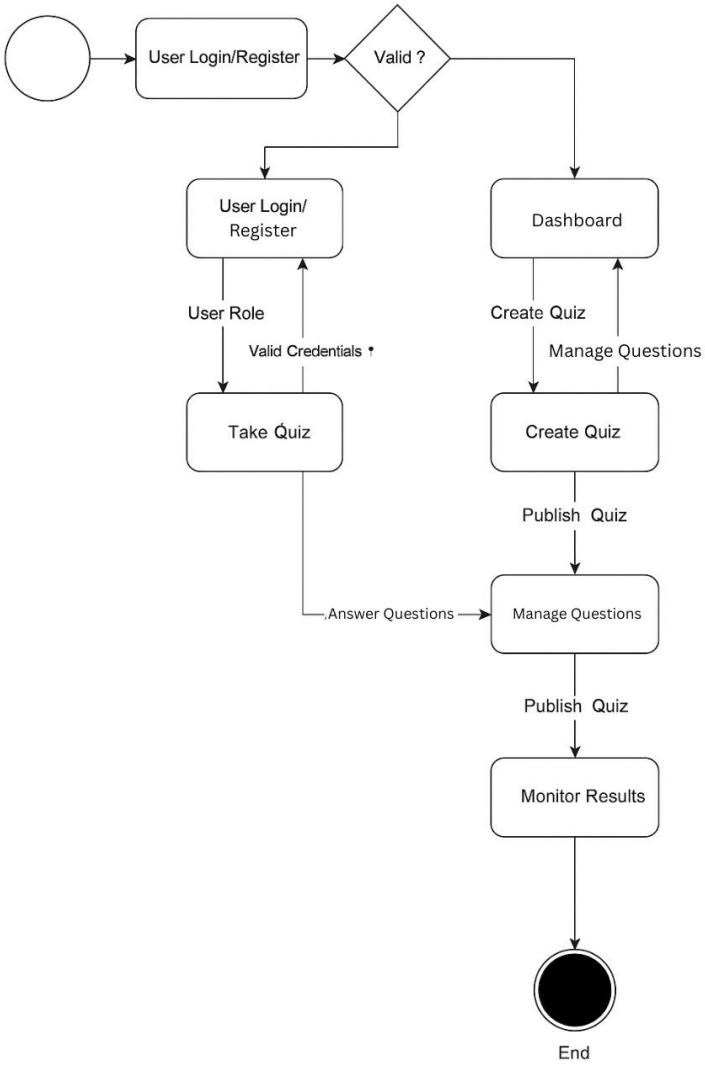

Figure 3.4: State Diagram for Quiz Management System

In figure 3.4, the process begins with the user accessing the system through a login or registration form, transitioning to a validation state where their credentials are checked. Upon successful validation, the system diverges based on user roles; regular users move to the quiz-taking flow, where they select a quiz, answer questions, and submit responses. This leads to the final state where results and scores are displayed. On the other hand, admins enter a different flow, progressing through states such as creating quizzes, managing

questions, and publishing them. The process concludes with monitoring quiz results and user performance. The diagram clearly captures how the system handles role-based transitions, input validation, and dynamic quiz lifecycle management from both the user and system perspectives.

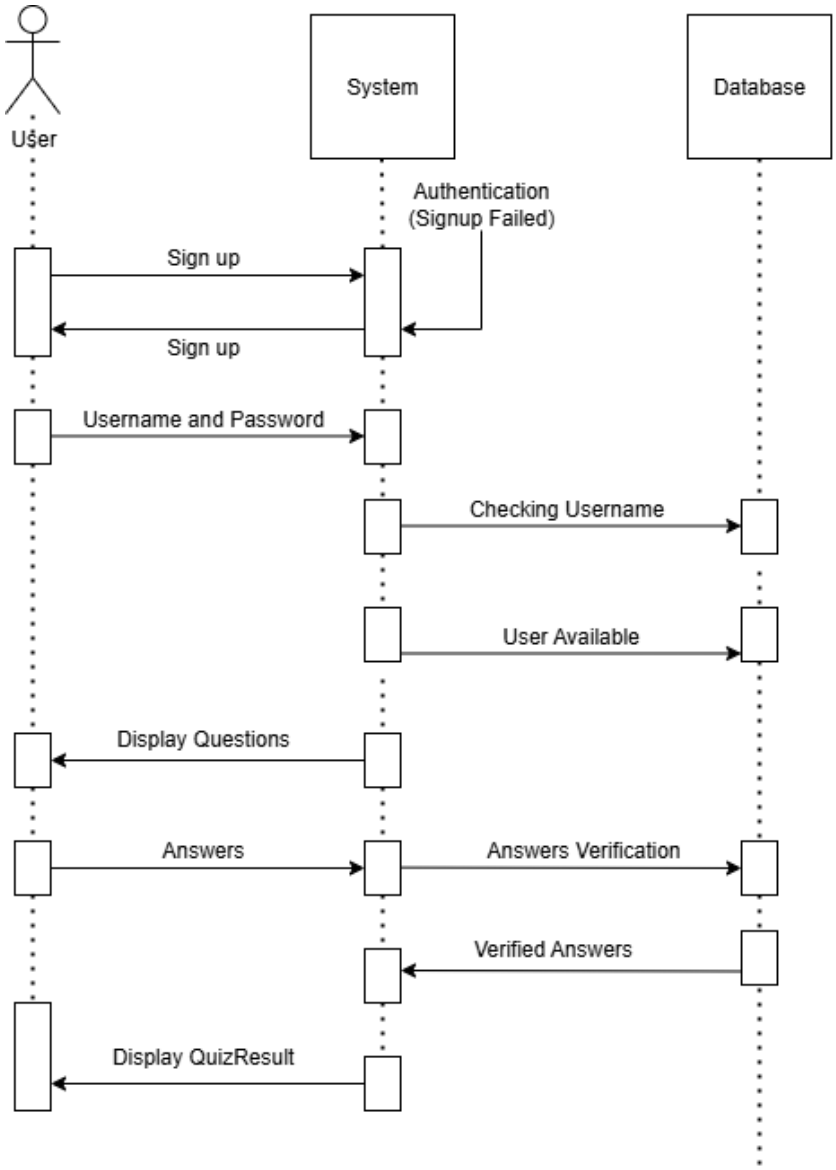

Figure 3.5: Sequence Diagram for Quiz Management System

In Figure 3.6, the process begins when the user logs into the application and selects a quiz category. The application captures this request and forwards it to the server, which authenticates the user and fetches questions related to the selected category. The server sends back a set of questions, which are then displayed to the user through the application interface.

As the user attempts the quiz, their answers are submitted question by question or all at once, depending on the design. The application transmits the responses to the server, where the server evaluates the answers, calculates the score, and stores the result in the database. Once the evaluation is complete, the server sends the result (such as score and correct answers) back to the application, which then displays the final result to the user, completing the quiz process.

3.1.5. Process Modelling using Activity Diagrams

The activity diagram represents the workflow of a diabetes prediction system. The process starts with the user login form and the authentication of user details.

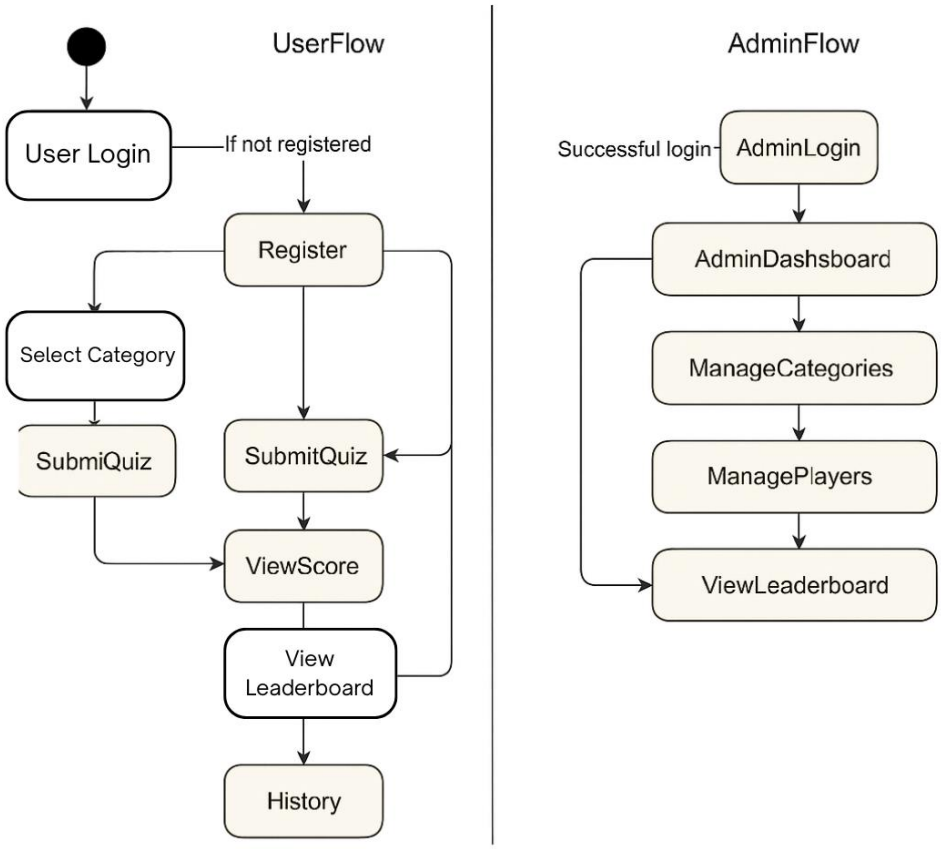

Figure 3.6: Activity Diagram for Quiz Management System

Figure 3.6 illustrates the structured flow of a quiz-based application, starting from user authentication to score display. Initially, the user logs into the system, which then authenticates the credentials to ensure secure access. Upon successful authentication, the

user proceeds to select a quiz category of interest. The system responds by retrieving and displaying the relevant set of questions for the chosen category, difficulty level, and number of questions. The user then attempts the displayed questions, after which the system evaluates the responses and calculates the final score. This score is subsequently displayed to the user. The diagram clearly outlines the sequential interactions between the user and the system, emphasizing key stages such as authentication, content delivery, user engagement, and result presentation.

3.3. Algorithm Details

3.3.1. Fisher-Yates Shuffle Algorithm

The algorithm used in the Quiz Management System effectively combines the Fisher-Yates Shuffle with an AI-driven difficulty selection mechanism to ensure both fairness and personalization in quiz delivery. The Fisher-Yates Shuffle is a time-tested algorithm known for generating a completely random permutation of elements within a list, ensuring that every possible arrangement is equally likely. In this system, it is used to shuffle the order of questions each time a quiz is generated. This randomness helps eliminate predictability and reduces the chances of memorizing question sequences, thereby promoting integrity and fairness. The algorithm works by iterating backward through the list and swapping each element with another randomly selected element that comes before it. Because it performs these swaps in-place and operates in linear time, it requires minimal memory and is highly efficient for real-time quiz applications.

In addition to randomness, the system introduces AI-based adaptability by analyzing a user's recent performance data to estimate their skill level. If a user has consistently answered more than 80% of their questions correctly, the system classifies them as advanced and assigns them a quiz with more difficult questions. Users with a success rate between 50% and 80% receive a balanced mix of medium and a few hard questions, while those with lower performance are given mostly easy questions. This performance-based classification allows the system to personalize the quiz experience for each user, making it inclusive for learners of all levels—whether beginner, intermediate, or advanced.

After determining the user's skill level, the algorithm applies a corresponding distribution of question difficulties. It filters the full set of questions by their assigned difficulty level—easy, medium, or hard—and selects the appropriate number of questions from each group according to the distribution. The Fisher-Yates Shuffle is again applied within each group

before selecting the final questions to ensure that even questions of the same difficulty are randomized. This final pool of selected questions is then shuffled one more time to remove any pattern in their order, ensuring that the quiz feels unpredictable and well-balanced.

Overall, this algorithmic approach brings together the efficiency of the Fisher-Yates Shuffle with the intelligence of adaptive difficulty modeling. It provides a cost-effective and scalable solution that not only improves the quiz-taking experience but also supports learning by adjusting content difficulty to suit individual user needs. This makes the system highly effective for educational environments where fairness, adaptability, and engagement are critical.

This mathematical formula for the Fisher-Yates Shuffle Algorithm is:

For an array $A=[a_0, a_1, \ldots\ldots, a_{n-1}]$*, for each i from n-1 down to 1:*…….(i)

j = *random integer such that* $0 \leq j \leq I$ …….. *(equation 1)*

$$Swap\ a_i \leftrightarrow a_j$$

Where:

n: Length of the array.

i: Current index in the reverse iteration (starting from n−1 down to 1).

j: Random index selected uniformly such that $0 \leq j \leq i$.

$a_i \leftrightarrow a_j$*: Swap the values at positions i and j.*

Question Distribution Formula:

Let:

N = total number of questions to select

$d \in$ {easy, medium, hard}

p_d = proportion of questions with difficulty level d for the current user skill level

Then, the number of questions of difficulty d is calculated as:

$Q_d = Round(N \times p_d)$

The values of p_d depend on the user's skill level:

For Easy users:

$P_{easy} = 0.7$, $P_{medium} = 0.3$, $P_{hard} = 0$

For Medium users:

$P_{easy} = 0.2$, $P_{medium} = 0.6$, $P_{hard} = 0.2$

For Hard users:

$P_{easy} = 0$, $P_{medium} = 0.3$, $P_{hard} = 0.7$

If the total number of selected questions ($\sum_d Q_d < N$) is less than N, the remaining questions are added to the dominant difficulty level for that user.

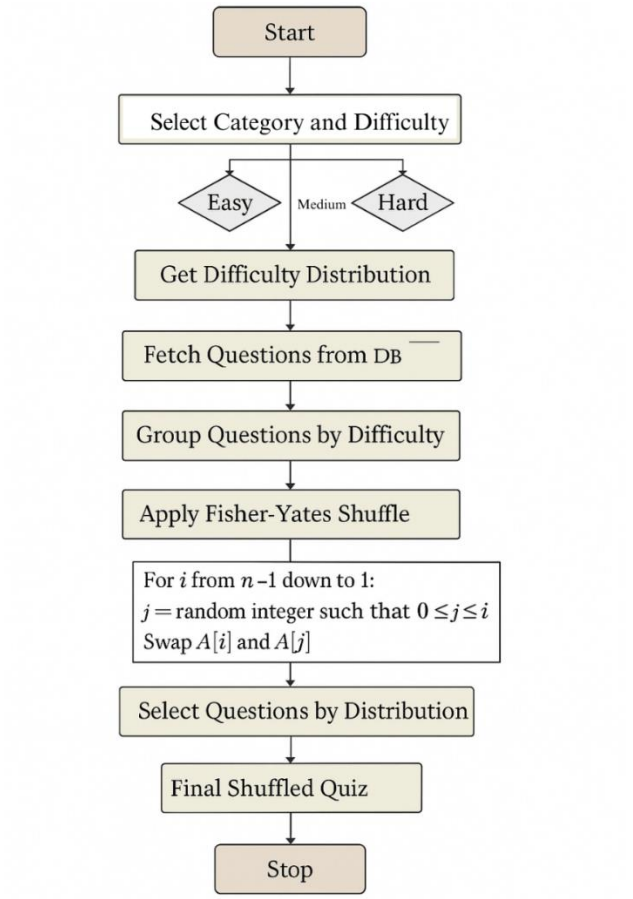

Figure 3.18: Fisher-Yates Shuffle Algorithm [8]

CHAPTER 4

IMPLEMENTATION AND TESTING

4.1. Implementation

4.1.1. Tools Used

MongoDB: MongoDB is a NoSQL, document-oriented database that stores data in flexible, JSON-like documents. It allows developers to handle large volumes of unstructured or semi-structured data with ease. In the MERN stack, MongoDB serves as the primary database, providing scalability and high performance for storing application data. Its schema-less nature offers flexibility in data modeling, making it suitable for modern web applications where data requirements may evolve over time.

Express.js: Express.js is a lightweight and flexible web application framework for Node.js that simplifies the process of building server-side logic. It provides a robust set of features for handling HTTP requests, routing, middleware, and APIs. In the MERN architecture, Express acts as the backend framework that connects the frontend (React) to the database (MongoDB), enabling efficient request handling and response delivery.

React.js: React.js is a popular JavaScript library developed by Facebook for building dynamic and responsive user interfaces. It allows developers to create reusable UI components and manage application state efficiently through its virtual DOM and declarative syntax. In the MERN stack, React handles the frontend, providing an interactive and fast user experience.

Tailwind CSS: Tailwind CSS is a utility-first CSS framework used to design custom user interfaces quickly and efficiently. In a React Native context, Tailwind-like utility libraries such as nativewind are often used to style components using predefined utility classes, allowing rapid UI development with consistency.

Node.js: Node.js is a server-side JavaScript runtime built on Chrome's V8 engine. It enables developers to run JavaScript on the backend, allowing for a unified development language across the entire application stack. In the MERN stack, Node.js acts as the runtime environment for Express, handling asynchronous operations and executing backend logic. Its event-driven, non-blocking I/O model makes it highly efficient and capable of handling

concurrent connections, which is crucial for modern web applications that require scalability and performance.

4.1.2. Implementation Details of Modules

The following are the modules used for the implementation of Quiz Management System.

4.1.2.1. Fisher-Yates Shuffle Algorithm:

To ensure both fairness and personalization, the system combines user-selected difficulty levels powered by AI with the Fisher-Yates Shuffle Algorithm. At the beginning of each quiz, the AI system allows users to select their preferred difficulty level: easy, medium, or hard, ensuring that the quiz content aligns with their current comfort and confidence. Based on this selection, the system pulls questions that match the chosen difficulty, creating a more tailored and inclusive experience for all types of learners.

Once the questions are fetched, the Fisher-Yates Shuffle Algorithm is applied to randomize their order before the quiz begins. This algorithm operates in linear time and performs in-place swapping, making it both efficient and lightweight. The shuffling ensures that no two quiz attempts are exactly the same, even if the same difficulty level is chosen, effectively reducing predictability and minimizing the chances of cheating or memorizing question patterns.

By combining user-driven difficulty selection with robust randomization, the system delivers a unique, fair, and adaptive quiz experience that enhances both learning outcomes and platform integrity.

```
const mongoose = require('mongoose')
const Question = require('../models/Question')

/**
 * @param {Array} array - Array to be shuffled
 * @returns {Array} - Shuffled array
 */
function fisherYatesShuffle(array) {
  const shuffled = [...array]
  for (let i = shuffled.length - 1; i > 0; i--) {
    const j = Math.floor(Math.random() * (i + 1))
    ;[shuffled[i], shuffled[j]] = [shuffled[j], shuffled[i]]
  }
  return shuffled
}

/**
 * @param {Object} userProfile - User's profile with performance history
 * @returns {String} - Difficulty level (easy, medium, hard)
 */
function determineUserSkillLevel(userProfile) {
  if (!userProfile?.recentPerformance) {
    return 'medium'
  }

  const successRate =
    userProfile.recentPerformance.correct / userProfile.recentPerformance.total

  if (successRate >= 0.8) {
    return 'hard'
  } else if (successRate >= 0.5) {
    return 'medium'
  } else {
    return 'easy'
  }
}

/**
 * @param {String} userSkillLevel - User's current skill level
 * @returns {Object} - Distribution of questions by difficulty
 */
function getQuestionDistribution(userSkillLevel) {
  const distributions = {
    easy: { easy: 0.7, medium: 0.3, hard: 0 },
    medium: { easy: 0.2, medium: 0.6, hard: 0.2 },
    hard: { easy: 0, medium: 0.3, hard: 0.7 },
  }

  return distributions[userSkillLevel]
}
```

Figure 4.1: Code Snippet of Fisher-Yates Shuffle Algorithm

4.1.2.2. Registration and Login Module:

This module handles user authentication and access control by securely managing the processes of user registration and login. During registration, it validates user inputs such as name, email, and password to ensure all required fields are provided and meet defined formatting and security standards, such as proper email format and sufficient password strength. Additionally, the login process includes a validation rule that restricts the use of

special characters in email. It securely hashes passwords before storing them in the database to safeguard against unauthorized access. During login, the system verifies that the entered email exists and that the provided password matches the securely stored hash. If the credentials are correct, a JSON Web Token (JWT) is generated to manage authenticated sessions. Any validation failure or incorrect login attempt triggers informative error messages, helping users correct inputs. Overall, this module ensures a secure, efficient, and well-validated authentication flow that protects user data and access.

```
// Login user
router.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body

    // Check if user exists
    const user = await User.findOne({ email })
    if (!user) {
      return res.status(400).json({ message: "Invalid credentials" })
    }

    // Check password
    const isMatch = await user.comparePassword(password)
    if (!isMatch) {
      return res.status(400).json({ message: "Invalid credentials" })
    }

    // Generate JWT
    const token = jwt.sign({ userId: user._id }, process.env.JWT_SECRET || "your_jwt_secret", { expiresIn: "7d" })

    res.json({
      token,
      user: {
        _id: user._id,
        name: user.name,
        email: user.email,
        role: user.role,
      },
    })
  } catch (error) {
    console.error(error)
    res.status(500).json({ message: "Server error" })
  }
})
```

Figure 4.2 Code Snippet of Registration and Login Module

4.1.2.3. Question Validation Module:

Before storing quiz questions, this module ensures each question meets predefined validation criteria to maintain consistency and reliability. This module performs a series of checks on each question to verify that it adheres to predefined validation rules. It ensures that the question text is not empty, there are at least two answer options, and exactly one of those options is marked as correct. Additionally, it checks for duplicate answer options and validates that all inputs follow the correct data types and formatting standards. If any validation fails, the system generates clear and specific error messages to help

administrators correct the issues promptly. By enforcing these standards, the module guarantees that only high-quality, well-structured questions are added to the quiz database, thereby enhancing the overall quiz experience and ensuring reliable assessment outcomes.

```
const validateQuestion = [
  body("text")
    .trim()
    .notEmpty()
    .withMessage("Question text is required"),

  body("options")
    .isArray({ min: 2 })
    .withMessage("Options must be an array with at least 2 items")
    .custom((opts) => opts.every(opt => typeof opt === "string" && opt.trim().length > 0))
    .withMessage("Each option must be a non-empty string"),

  body("correctOption")
    .isInt({ min: 0 }) // must be an integer >= 0
    .withMessage("Correct option must be a valid index")
    .custom((value, { req }) => {
      const options = req.body.options;
      if (!Array.isArray(options)) return false;
      return value >= 0 && value < options.length; // check index bounds
    })
    .withMessage("Correct option index must be within the range of options"),

  body("categoryId")
    .notEmpty()
    .withMessage("Category ID is required")
    .custom((value) => mongoose.Types.ObjectId.isValid(value))
    .withMessage("Invalid category ID"),
];
```

Figure 4.3 Code Snippet of Question Validation Model

4.1.2.4. Scores of Players Module:

The Scores of Players Module is responsible for calculating, recording, and managing user scores after each quiz attempt. When a user completes a quiz, this module evaluates the submitted answers by comparing them with the correct options stored in the database. Once the evaluation is complete, the score along with additional metrics such as the number of correct and incorrect answers, the time taken to complete the quiz, and the percentage accuracy, is stored in the database under the user's performance history. Each score record is linked to the specific quiz ID and user ID. This module also supports result retrieval for dashboards and leaderboards, allowing users to view their past performance and enabling

administrators to analyze user engagement and quiz difficulty levels. By maintaining a detailed log of user scores, this module enhances transparency.

```
router.get("/leaderboard", [auth, adminAuth], async (req, res) => {
  try {
    const results = await QuizResult.find().populate("player", "name").populate("category", "name")

    const playerStats = {}

    results.forEach((result) => {
      const playerId = result.player._id.toString()
      const playerName = result.player.name
      const categoryId = result.category._id.toString()
      const categoryName = result.category.name
      const score = result.totalQuestions > 0 ? (result.score / result.totalQuestions) * 100 : 0

      if (!playerStats[playerId]) {
        playerStats[playerId] = {
          _id: playerId,
          playerName,
          quizzesTaken: 0,
          totalScore: 0,
          bestScore: 0,
          categories: {},
        }
      }

      playerStats[playerId].quizzesTaken++
      playerStats[playerId].totalScore += score
      playerStats[playerId].bestScore = Math.max(playerStats[playerId].bestScore, score)

      if (!playerStats[playerId].categories[categoryId]) {
        playerStats[playerId].categories[categoryId] = {
          categoryId,
          categoryName,
          quizzesTaken: 0,
          totalScore: 0,
          bestScore: 0,
        }
      }

      playerStats[playerId].categories[categoryId].quizzesTaken++
      playerStats[playerId].categories[categoryId].totalScore += score
      playerStats[playerId].categories[categoryId].bestScore = Math.max(
        playerStats[playerId].categories[categoryId].bestScore,
        score
      )
    })

    const leaderboard = Object.values(playerStats).map((player) => {
      player.averageScore = player.quizzesTaken > 0 ? player.totalScore / player.quizzesTaken : 0

      Object.values(player.categories).forEach((category) => {
        category.averageScore = category.quizzesTaken > 0 ? category.totalScore / category.quizzesTaken : 0
      })
```

Figure 4.4 Code Snippet of Scores of Players Module

4.2. Testing

4.2.1. Test Cases for Unit Testing

Table 4.1 Test Cases for Unit Testing of Registration and Login Module

SN	Action	Input	Expected Outcomes	Actual Outcomes	Test Results
1	Login with valid credentials	Correct email and password	Success: JWT token and user details returned	JWT token and user details returned	Pass
2	Login with invalid email	Incorrect email and correct password	Failure: "Invalid credentials" message	"Invalid credentials" message returned	Pass
3	Login with invalid password	Correct email and incorrect password	Failure: "Invalid credentials" message	"Invalid credentials" message returned	Pass
4	Login with missing email	Missing email and correct password	Failure: "Invalid credentials" message	"Invalid credentials" message returned	Pass
5	Login with missing password	Correct email and missing password	Failure: "Invalid credentials" message	"Invalid credentials" message returned	Pass
6	Login with both fields empty	Missing email and password	Exit the page.	"Invalid credentials" message returned	Fail

Table 4.2: Test Cases for Unit Testing of Fisher Yates Shuffle Algorithm

SN	Action	Input	Expected Outcomes	Actual Outcomes	Test Results
1	Shuffle an array of numbers	[1, 2, 3, 4, 5]	All elements present, order randomized	All elements present, order randomized	Pass
2	Shuffle an array with duplicate values	[1, 2, 2, 3, 4]	All elements present, order randomized	All elements present, order randomized	Pass
3	Shuffle an array with one element	[1]	Returns same array	Returned same array `[1]`	Pass
4	Shuffle an empty array	[]	Returns empty array	Returned empty array `[]`	Pass
5	Shuffle an already shuffled array	[3, 1, 4, 2, 5]	All elements present, order randomized	All elements present, order randomized	Pass
6	Request more questions than are available	Request 100 questions	100 Questions Quiz	Error: Requested 100 questions	Fail

Table 4.3: Test Cases for Unit Testing of Question Validation Model

SN	Action	Input	Expected Outcomes	Actual Outcomes	Test Results
1	Submit empty question text	{ text: "", options: ["Yes", "No"], correctOption: 0, categoryId: "60df599eb60e3e3bb8fa67d3" }	Error: "Question text is required"	Error: "Question text is required"	Pass
2	Only one option provided	{ text: "Is it valid?", options: ["Yes"], correctOption: 0, categoryId: "60df599eb60e3e3bb8fa67d3" }	Error: "Options must be an array with at least 2 items"	Error: "Options must be an array with at least 2 items"	Pass
3	Empty option string	{ text: "Is it valid?", options: ["Yes", ""], correctOption: 0, categoryId: "60df599eb60e3e3bb8fa67d3" }	Error: "Each option must be a non-empty string"	Error: "Each option must be a non-empty string"	Pass
4	Invalid correct Option index	{ text: "Is it valid?", options: ["Yes", "No"], correctOption: 2, categoryId: "60df599eb60e3e3bb8fa67d3" }	Error: "Correct option index must be within the range of options"	Error: "Correct option index must be within the range of options."	Pass
5	Valid input	{ ghyabshdj: "Is it valid?", options: ["Yes", "No"], correctOption: 1 , categoryId: "60df599eb60e3e3bb8fa67d3" }	Error: Enter Valid Question	Question has been added	Fail

Table 4.4: Test Cases for Unit Testing of Scores of Players Module

SN	Action	Input	Expected Outcomes	Actual Outcomes	Test Results
1	One user completes one quiz	`QuizResult`: 10 totalQuestions, score: 7, linked to player and category	Player score = 70%, quizzesTaken: 1, bestScore: 70	Player score = 70%, quizzesTaken: 1, bestScore: 70	Pass
2	User takes two quizzes, best score retained	Two results: score 60% and 90%	bestScore: 90, averageScore: 75	bestScore: 90, averageScore: 75	Pass
3	User with quizzes in two categories	Two results: one per category, scores 80% and 60%	Two categories listed separately under user, each with correct stats	Both categories shown with correct data	Pass
4	Score with totalQuestions = 0	`QuizResult`: totalQuestions: 0, score: any	Score treated as 0%, no division-by-zero error	Score handled as 0, no crash	Pass
5	Incorrect average due to missing quiz count	One quiz result: score 80%, but `quizzesTaken` is incorrectly not incremented in logic	averageScore should be 80%, quizzesTaken = 1	averageScore = 0 due to quizzes Taken = 0 (division by zero or wrong calc)	Fail

4.2.2. Test Cases for System Testing

Table 4.5. Test Cases for System Testing of Registration and Login Module

SN	Action	Input	Expected Outcomes	Actual Outcomes	Test Results
1	Register new user	Valid name, email, password	Success: Account created, login available	Account created, user logged in	Pass
2	Login with valid credentials	Registered email and password	JWT token and user details returned	JWT token and user details returned	Pass
3	Login with wrong password	Correct email, wrong password	Error: Invalid credentials	Error: Invalid credentials	Pass
4	Try accessing dashboard without login	No token in headers	Error: Unauthorized access	Error: Unauthorized access	Pass
5	Register with existing email	Same email used twice	Error: Email already exists	Account created again (duplicate entry allowed)	Fail

Table 4.6. Test Cases for System Testing of Fisher-Yates Shuffle Algorithm

SN	Action	Input	Expected Outcomes	Actual Outcomes	Test Results
1	Shuffle array of 5 numbers	[1, 2, 3, 4, 5]	Same elements, randomized order	Same elements, randomized order	Pass
2	Shuffle array with duplicates	[1, 2, 2, 3, 4]	Elements preserved, order changed	Elements preserved, order changed	Pass
3	Shuffle array with 1 element	[1]	Same array returned	Same array returned	Pass
4	Shuffle empty array	[]	Empty array returned	Empty array returned	Pass
5	Shuffle without providing an array	null input	Error or empty output with proper message	Crashed without handling null input	Fail

Table 4.7. Test Cases for System Testing of Question Validation Module

SN	Action	Input	Expected Outcomes	Actual Outcomes	Test Results
1	Submit complete valid question	Valid text, 4 options, correct index, valid category ID	Question saved successfully	Question saved successfully	Pass
2	Submit with empty question text	"", 4 options, correct index, valid category ID	Error: "Question text is required"	Error: "Question text is required"	Pass
3	Submit with duplicate options	Valid text, options: ["A", "A"], correct index 1, valid category ID	Should reject duplicate values	Question saved successfully	Fail
4	Submit with missing category ID	Valid text, 4 options, correct index, empty category ID	Error: "Category ID is required"	Error: "Category ID is required"	Pass
5	Submit with wrong correctOption	Valid text, options: ["A", "B"], correct index: 5, valid category ID	Error: "Correct option index must be within range"	Error: "Correct option index must be within range"	Pass

Table 4.8. Test Cases for System Testing of Scores of Players Module

SN	Action	Input	Expected Outcomes	Actual Outcomes	Test Results
1	One quiz attempt saved	User finishes quiz with 8/10 correct	Score saved: 80%, leaderboard updated	Score saved and shown correctly	Pass
2	Multiple quizzes across categories	User completes quizzes in "Math" and "Science"	Scores updated in both categories	Stats updated correctly in both categories	Pass
3	User views leaderboard	Valid token, leaderboard route hit	Leaderboard data returned with ranks and averages	Leaderboard data returned	Pass
4	Quiz attempt with totalQuestions = 0	QuizResult: totalQuestions: 0, score: 5	Score treated as 0%, no crash	Score treated as 0%, no crash	Pass
5	Corrupted score entry	QuizResult: score as string "eighty", totalQuestions: 10	Error or ignored score, no crash	App crashes due to type error	Fail

CHAPTER 5

CONCLUSION AND FUTURE MANAGEMENTS

5.1. Conclusion

In summary, the Quiz Management System project presents a comprehensive solution for creating, managing, and evaluating online quizzes with a focus on fairness, flexibility, and efficiency. The system is designed to support a wide range of users—from administrators managing quiz content to participants attempting quizzes and viewing results. Its modular architecture ensures that core features such as registration, quiz creation, question validation, timed quiz attempts, and result tracking function seamlessly within a unified platform.

A key strength of the system lies in its use of the Fisher-Yates Shuffle Algorithm, which ensures each quiz attempt features randomized question order, thereby minimizing predictability and promoting fairness among participants. This, combined with structured question validation, ensures high-quality content integrity. The system also supports secure user authentication using JWT, ensuring that user data remains safe and access is role-restricted. Here, Artificial Intelligence (AI) is integrated for users to choose their difficulty level: easy, medium, or hard. Based on the selected difficulty level, the system assigns a corresponding timer for each quiz, ensuring an appropriate time-bound challenge that aligns with the user's chosen level. Leaderboard generation add significant value by enhancing the assessment experience for learners and educators alike.

The technical implementation leverages modern technologies, including React for the frontend and Node.js with MongoDB for the backend, resulting in a responsive and scalable web-based application. Functional requirements, such as dynamic quiz allocation and instant feedback, have been effectively met.

As the project moves toward deployment, it has the potential to serve educational institutions, training platforms, and certification providers by offering a smart and customizable assessment environment. Future enhancements could include support for multimedia-based questions, adaptive using mobile application integration, and advanced analytics dashboards for educators. Overall, the Quiz Management System stands as a valuable contribution to digital learning and assessment tools, demonstrating how

technology can streamline knowledge evaluation while maintaining integrity and engagement.

5.2. Future Recommendations

In future, this project has an immense room for improvement. The following are some of the future recommendations for the project:

- Adaptive Quiz Feature: Add smart quizzes that change question difficulty based on how well the user is doing. This makes the quiz more personalized and fair for each user.
- Support for Images and Videos: Include questions with pictures, audio, or videos to make quizzes more fun and helpful, especially for learning different subjects.
- Better Reports and Analytics: Provide detailed results for both users and admins, showing things like scores, time taken, and areas to improve. This helps track progress and improve quiz content.
- Gamification and Rewards: Add features like points, badges, and levels to make quizzes more engaging. Users can earn rewards and feel motivated to keep learning.

References

[1] M. A. Al-Zoube, S. El-Seoud, and J. Wyne, "A Cloud Based Learning System: Implementation and Performance Evaluation," *International Journal of Emerging Technologies in Learning (iJET)*, vol. 14, no. 04, pp. 242-258, 2019. [Online]. Available: https://doi.org/10.3991/ijet.v14i04.9728

[2] S. K. Yadav, S. Pal, and D. K. Singh, "Automated Question Paper Generator System Using Fisher-Yates Shuffle Algorithm," International Journal of Computer Applications, vol. 179, no. 46, pp. 1-6, Apr. 2018. [Online]. Available: https://www.ijcaonline.org/archives/volume179/number46/yadav-2018-ijca-916492

[3] M. Kaur, "Automated Question Paper Generation System Using Randomization Algorithm," International Journal of Advanced Research in Computer Science, vol. 8, no. 5, pp. 1330-1334, May-Jun. 2017.

[4] A. K. Singh and S. Tiwari, "An Intelligent Quiz Management System Using Machine Learning," Journal of King Saud University - Computer and Information Sciences, vol. 34, no. 6, pp. 3526-3538, 2022. [Online]. Available: https://doi.org/10.1016/j.jksuci.2020.12.008

[5] M. A. Hossain and M. F. Rahman, "An Adaptive Quiz Generation System Using Item Response Theory," IEEE Access, vol. 9, pp. 118234-118247, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3107042

APPENDIXES

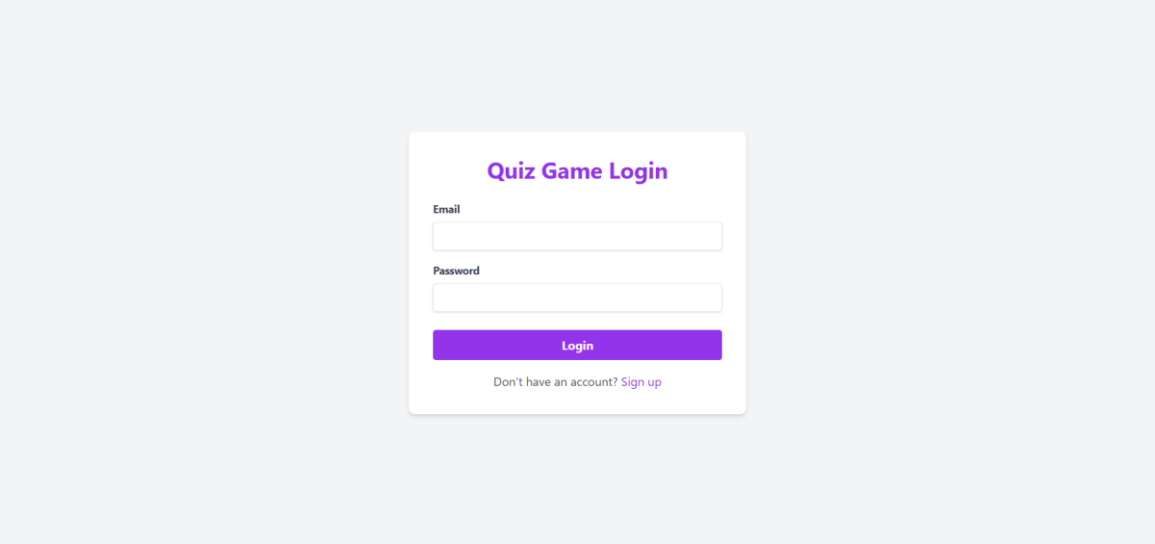

Figure: Login/Signup Page

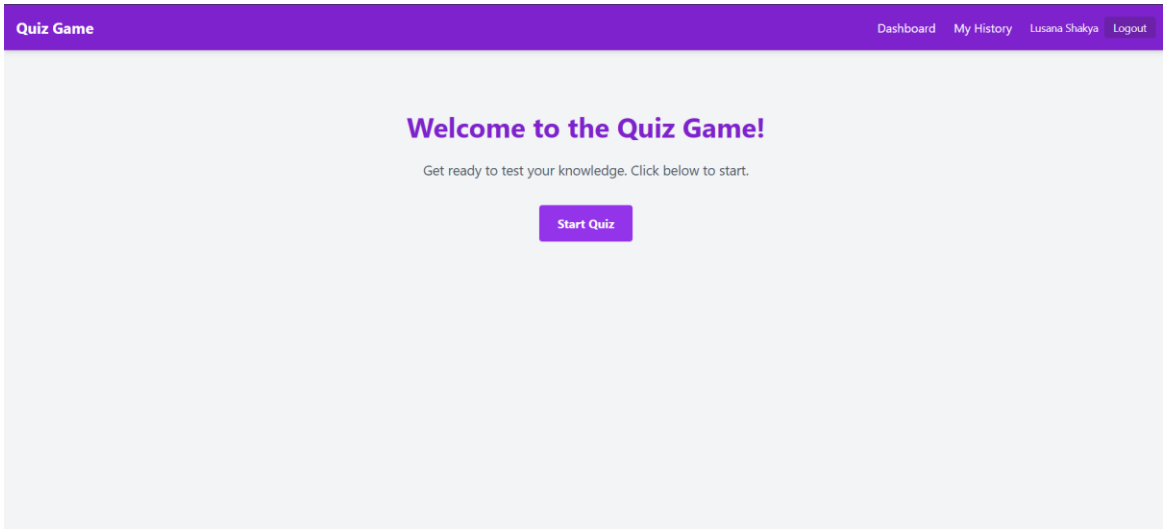

Figure: Landing Page

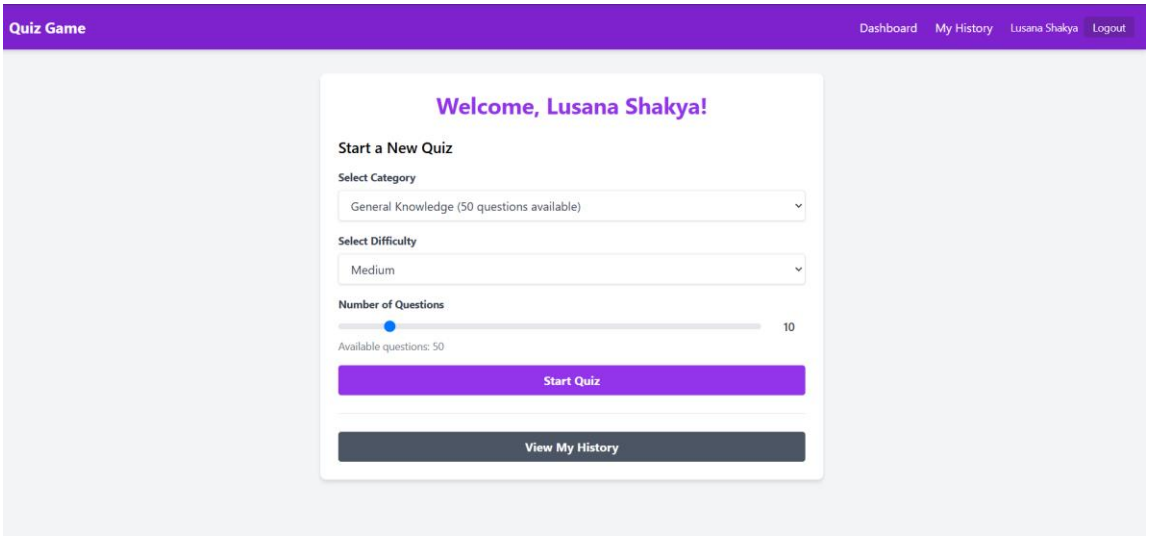

Figure: Quiz Page

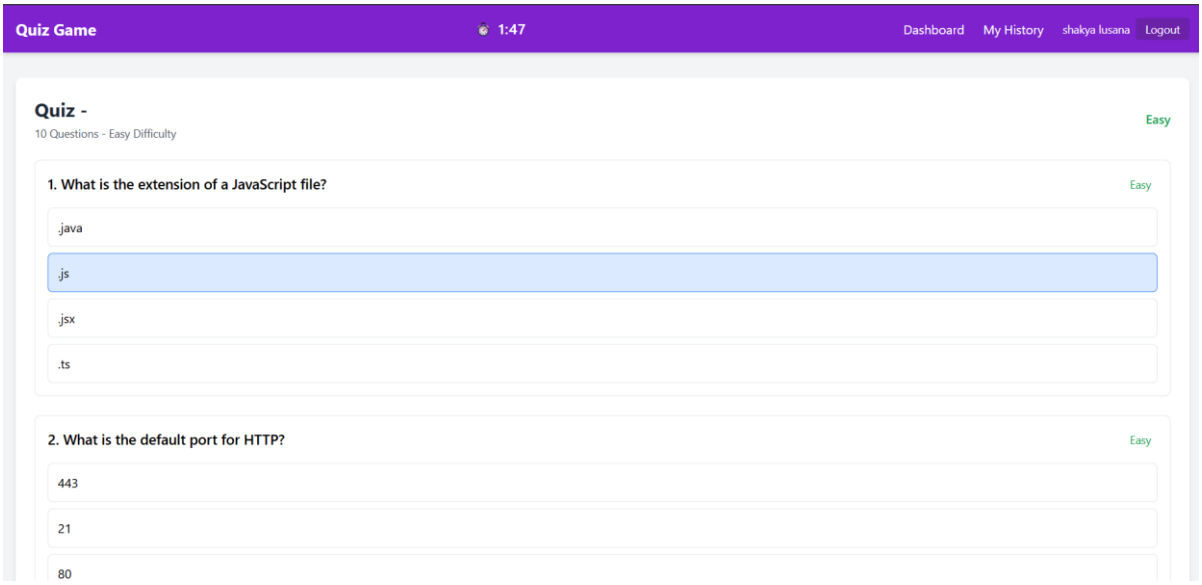

Figure: Questions Page

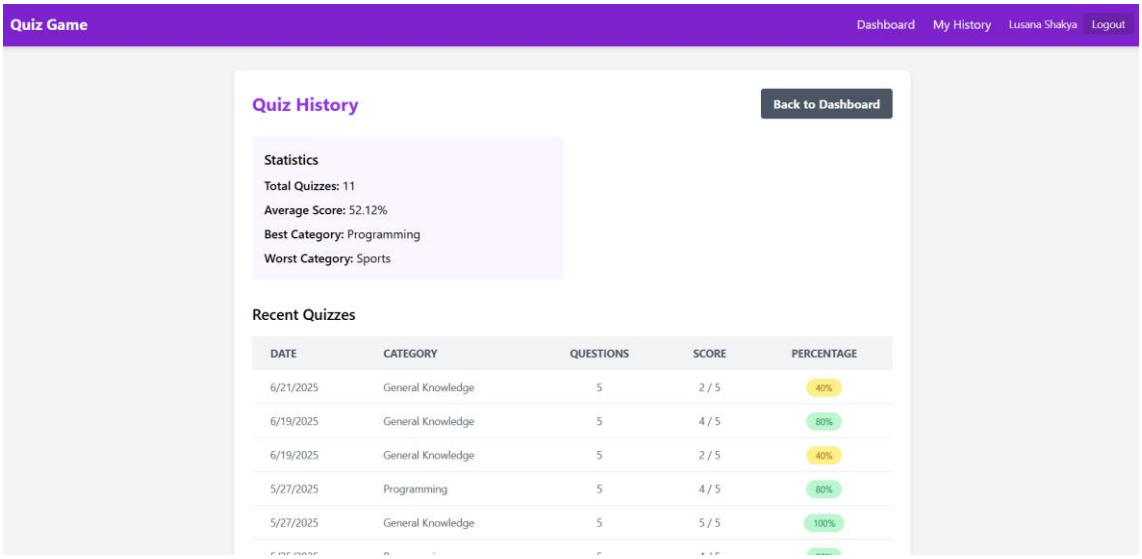

Figure: History Page

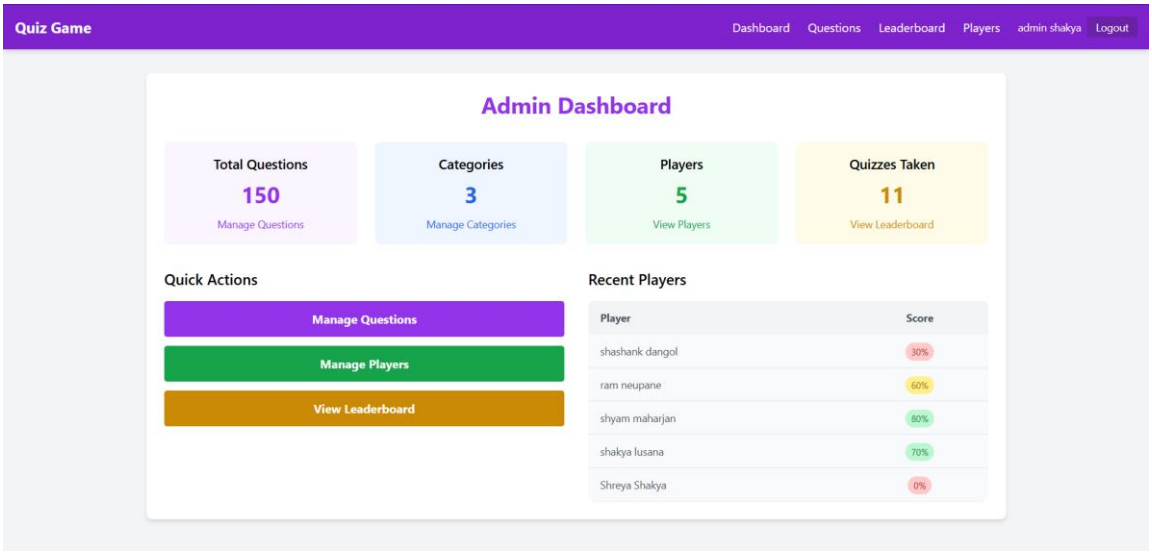

Figure: Admin Dashboard

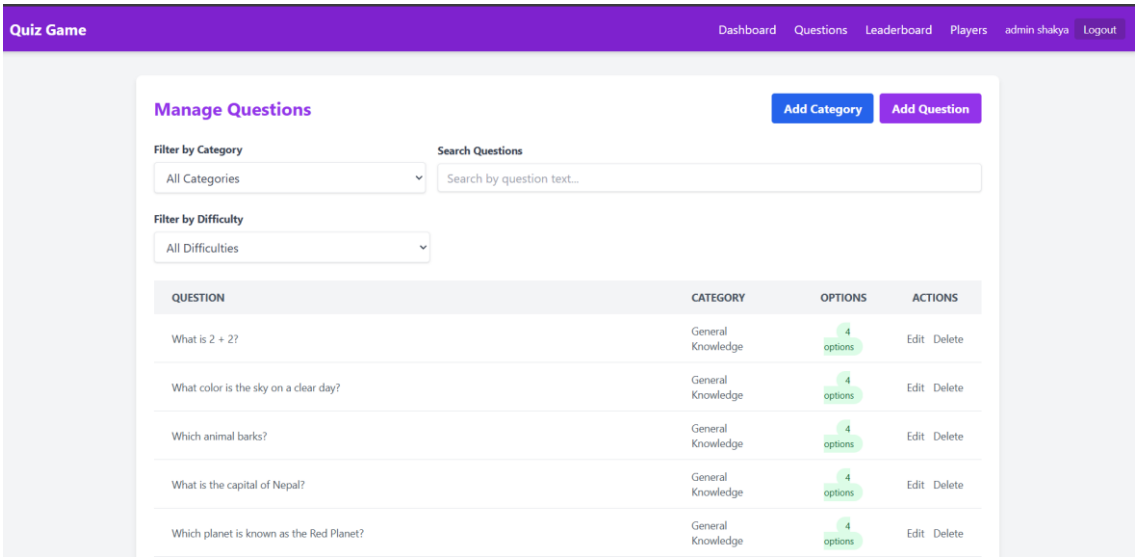

Figure: Admin Questions.

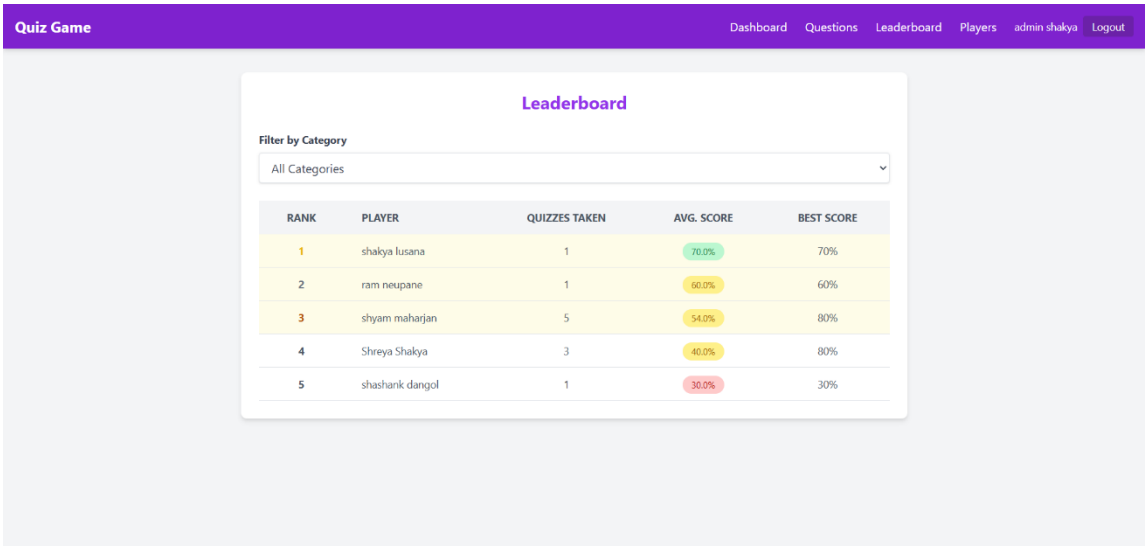

Figure: Players Leaderboard