A Web Technologies LAB Project Report

On

QuizCraft a Quiz App

B.Tech COMPUTER SCIENCE & ENGINEERING, 2024-2025

Submitted By	**Roll Number**
ITHARAJU RAKSHITHA	22311A05H9

Instructor: Dr.Dheeraj Sundaragiri

Department of Computer Science and Engineering

SREENIDHI INSTITUTE OF SCIENCE AND TECHNOLOGY (AUTONOMOUS)

Yamnampet (V), Ghatkesar (M), Hyderabad - 501301, T.S.

Contents

1 Introduction

This document presents a project developed as part of the Web Technology lab assignment, aimed at showcasing practical understanding of modern full-stack web development using the MERN stack—MongoDB, Express.js, React, and Node.js. The MERN Stack Quiz Web Application is a comprehensive online quiz platform that allows users to create, take, and manage quizzes. The application features user authentication, role-based access control, real-time quiz taking, and result tracking. Built using the MERN stack (MongoDB, Express.js, React.js, Node.js), it provides a modern, responsive, and user-friendly interface for both quiz takers and administrators.

2 Project Objectives

The objective of the Quiz App project is to develop an interactive, user-friendly web application that allows users to participate in quizzes across various categories and difficulty levels. The app is designed to be responsive and accessible across devices, with an intuitive UI/UX. It also includes an admin panel for quiz creation and user management. Overall, the app promotes self-assessment, learning retention, and competitive participation.

- Create a secure and scalable quiz platform
- Implement user authentication and authorization
- Provide an intuitive interface for quiz creation and management
- Enable real-time quiz taking with timer functionality
- Track and display quiz results with detailed analytics
- Support multiple question types and categories
- Implement role-based access control for admin features

3 System Architecture

The architecture of *quiz app* follows a modular full-stack design based on a refined MERN stack, integrating react.js for improved developer experience and application scalability. The architecture is logically divided into four layers: client-side interface, API layer, server-side logic, and database.

3.1 Client-Side (React.js)

The client-side of the application is built using React.js, providing a dynamic and responsive user interface. The architecture follows a component-based structure, where the application is divided into reusable components and pages. Material-UI is used for consistent and modern UI components, while React Router handles navigation between different pages. State management is implemented using React Hooks, providing a clean and efficient way to manage component state. Axios is used for making HTTP requests to the server, handling API communication in a structured manner. The client-side code is organized into features, with each feature containing its related components, hooks, and utilities. This organization promotes code reusability and maintainability.

2. API Layer (Express.js)

The API layer is implemented using Express.js, serving as the middleware between the client and server. It follows RESTful principles, providing endpoints for various operations such as user authentication, quiz management, and result tracking. The API layer includes middleware for request validation, error handling, and security. JWT (JSON Web Tokens) is used for authentication, ensuring secure access to protected routes. The API routes are organized by feature, with each route file handling specific functionality. This organization makes the codebase easier to maintain and extend. Error handling is implemented consistently across all routes, providing meaningful error messages to the client.

3. Server Logic (Node.js)

The server logic is implemented using Node.js, handling the business logic of the application. It processes requests from the API layer, interacts with the database, and sends responses back to the client. The server logic includes functions for user authentication, quiz creation and management, and result calculation. It uses async/await for handling asynchronous operations, ensuring efficient request processing. The server code is organized into modules, with each module handling specific functionality. This modular approach promotes code reusability and maintainability. Error handling is implemented throughout the server logic, ensuring robust error management.

4. Database (MongoDB)

The database layer uses MongoDB, a NoSQL database, for data storage and retrieval. The database schema is designed using Mongoose, providing data validation, type checking, and relationship management. The schema includes collections for users, quizzes, and results, with appropriate indexes for performance optimization. The database design supports the application's requirements for user management, quiz creation, and result tracking while maintaining data integrity and consistency. MongoDB's flexible schema allows for easy modification and extension of the data model as requirements evolve. The database is hosted on MongoDB Atlas, providing high availability and scalability.

4 Tools and Technologies Used

- **Frontend:** React.js, CSS, HTML5, Material UI.
- **Backend:** Node.js, Express.js, API Routes, Mongoose.
- **Database:** MongoDB.
- **Authentication:** bcrypt, JWT (JSON Web Tokens), custom role-based access control.
- **Development Tools:** Visual Studio Code, Git, GitHub, Postman
- **Deployment:** Vercel (frontend and APIs), render(backend).

4 Project Design and Implementation

1. Database Schema Design

The application uses MongoDB to store user data, authentication credentials, quizzes, results. Each schema is modeled using Mongoose with support for references, embedded documents, and timestamps. The table below summarizes the key collections and their fields:

Collection Name	Field Name	Data Type / Description
quizzes	Id	String(unique,required)
	Title	String (required)
	Description	String (required)
	Category	String (required)
	Questions	Array of Questions
	Timelimit	Time permitted
	Created by	String(unique)
	IsActive	boolean
results	_id	String (required)
	User	String (required)
	Quiz	String(unique, required)
	Score	Number
	totalQuestions	Number
	correctAnswers	Number
	timeTaken	Time
	answers	Array of answers
	completedAt	String (default: ”user”)
users	_id	String(unique, required)
	Username	String(unique)
	Email	String(unique)
	Password	String(unique)
	Role	String(default: user)
	createdAt	TimeStamp

2. API Endpoints

The following RESTful API endpoints are exposed by the backend. These are implemented using Next.js API routes with additional logic handled through Express.js where required. Each route corresponds to a specific function or resource in the application:

- POST /api/auth/register — Register a new user account
- **GET** /api/quizzes - Get all active quizzes
- **GET** /api/quizzes/:id - Get specific quiz details
- **POST** /api/quizzes - Create a new quiz (admin only)
- **PUT** /api/quizzes/:id - Update an existing quiz (admin only)
- **DELETE** /api/quizzes/:id - Delete a quiz (admin only)
- **GET** /api/quizzes/:id/answers - Get quiz with correct answers (admin only)
- **POST** /api/results - Submit quiz result
- **GET** /api/results/my-results - Get user's quiz results
- **GET** /api/results/:id - Get specific result details
- **GET** /api/results - Get all results (admin only)
- **GET** /api/users - Get all users (admin only)

- **GET** /api/users/:id - Get specific user details
- **GET** /api/categories - Get all quiz categories
- **POST** /api/categories - Add new category (admin only)
- **PUT** /api/categories/:id - Update category (admin only)
- **DELETE** /api/categories/:id - Delete category (admin only)

3. Frontend Components

The frontend is built using React.js with Material-UI, providing a modern and responsive user interface. Each component is designed to be reusable and follows a modular architecture. Below is a list of major components and their purposes:

- **Home/** - Landing page with featured quizzes and user dashboard
- **Login/** - User authentication form with JWT token management.
- **Register/** - New user registration form with validation
- **Admin-Register/** - Special registration form for admin users
- **QuizList/** - Displays available quizzes with filtering and search
- **QuizAttempt/** - Interface for taking quizzes with timer and answer submission
- **QuizResult/** - Shows quiz results with detailed analysis
- **AdminDashboard/** - Admin interface for managing quizzes and users
- **NotFound/** - 404 error page for invalid routes.

4. Code

1. Backend API

```
const express = require('express');
const router = express.Router();
const Quiz = require('../models/Quiz');
const { auth, adminAuth } = require('../middleware/auth');

// Get all active quizzes
router.get('/', auth, async (req, res) => {
  try {
    const quizzes = await Quiz.find({ isActive: true })
      .select('-questions.correctAnswer')
      .populate('createdBy', 'username');
    res.json(quizzes);
  } catch (error) {
    res.status(500).json({ message: 'Error fetching quizzes', error: error.message });
  }
});

// Get quiz by ID
router.get('/:id', auth, async (req, res) => {
  try {
    const quiz = await Quiz.findById(req.params.id)
      .select('-questions.correctAnswer')
      .populate('createdBy', 'username');

    if (!quiz) {
      return res.status(404).json({ message: 'Quiz not found' });
    }

    res.json(quiz);
  } catch (error) {
    res.status(500).json({ message: 'Error fetching quiz', error: error.message });
  }
});

// Create new quiz (admin only)
router.post('/', adminAuth, async (req, res) => {
  try {
    const quiz = new Quiz({
      ...req.body,
      createdBy: req.user._id
    });

    await quiz.save();
    res.status(201).json(quiz);
  } catch (error) {
    res.status(500).json({ message: 'Error creating quiz', error: error.message });
  }
});

// Update quiz (admin only)
router.put('/:id', adminAuth, async (req, res) => {
  try {
    const quiz = await Quiz.findByIdAndUpdate(
      req.params.id,
      req.body,
      { new: true, runValidators: true }
    );

    if (!quiz) {
      return res.status(404).json({ message: 'Quiz not found' });
    }

    res.json(quiz);
  } catch (error) {
    res.status(500).json({ message: 'Error updating quiz', error: error.message });
  }
});

// Delete quiz (admin only)
router.delete('/:id', adminAuth, async (req, res) => {
  try {
    const quiz = await Quiz.findByIdAndDelete(req.params.id);

    if (!quiz) {
      return res.status(404).json({ message: 'Quiz not found' });
    }

    res.json({ message: 'Quiz deleted successfully' });
  } catch (error) {
    res.status(500).json({ message: 'Error deleting quiz', error: error.message });
  }
});

// Get quiz with correct answers (admin only)
router.get('/:id/answers', adminAuth, async (req, res) => {
  try {
    const quiz = await Quiz.findById(req.params.id);

    if (!quiz) {
      return res.status(404).json({ message: 'Quiz not found' });
    }

    res.json(quiz);
  } catch (error) {
    res.status(500).json({ message: 'Error fetching quiz answers', error: error.message
});
  }
});

module.exports = router;  }
 });
```

Example Quizzes.js Route

5.4.2 Frontend Component (React)

```
import React, { useState, useEffect } from 'react';
import { useNavigate, useParams } from 'react-router-dom';
import axios from 'axios';
import {
  Container,
  Paper,
  Typography,
  Button,
  Radio,
  RadioGroup,
  FormControlLabel,
  FormControl,
  FormLabel,
  Box,
  CircularProgress,
  Alert
} from '@mui/material';

const QuizAttempt = () => {
  const { id } = useParams();
  const navigate = useNavigate();
  const [quiz, setQuiz] = useState(null);
  const [currentQuestion, setCurrentQuestion] = useState(0);
  const [answers, setAnswers] = useState([]);
  const [timeLeft, setTimeLeft] = useState(null);
  const [error, setError] = useState('');
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchQuiz = async () => {
      try {
        const response = await axios.get(`/api/quizzes/${id}`);
        setQuiz(response.data);
        setTimeLeft(response.data.timeLimit * 60);
        setAnswers(new Array(response.data.questions.length).fill(null));
      } catch (err) {
        setError('Error loading quiz');
      } finally {
        setLoading(false);
      }
    };
    fetchQuiz();
  }, [id]);

  useEffect(() => {
    if (!timeLeft) return;

    const timer = setInterval(() => {
      setTimeLeft(prev => {
        if (prev <= 1) {
          clearInterval(timer);
          handleSubmit();
          return 0;
        }
        return prev - 1;
      });
    }, 1000);

    return () => clearInterval(timer);
  }, [timeLeft]);

  const handleAnswerChange = (event) => {
    const newAnswers = [...answers];
    newAnswers[currentQuestion] = parseInt(event.target.value);
    setAnswers(newAnswers);
  };

  const handleNext = () => {
    if (currentQuestion < quiz.questions.length - 1) {
      setCurrentQuestion(prev => prev + 1);
    }
  };

  const handlePrevious = () => {
    if (currentQuestion > 0) {
      setCurrentQuestion(prev => prev - 1);
    }
  };

  const handleSubmit = async () => {
    const unansweredQuestions = answers.findIndex(answer => answer === null);
    if (unansweredQuestions !== -1) {
      setError(`Please answer question ${unansweredQuestions + 1}`);
      setCurrentQuestion(unansweredQuestions);
      return;
    }

    try {
      const response = await axios.post('/api/results', {
        quizId: id,
        answers: answers.map((answer, index) => ({
          questionId: index,
          selectedOption: answer
        })),
        timeTaken: quiz.timeLimit * 60 - timeLeft
      });
      navigate(`/result/${response.data._id}`);
    } catch (err) {
      setError(err.response?.data?.message || 'Error submitting quiz');
    }
```

```
};

if (loading) {
  return (
    <Box display="flex" justifyContent="center" alignItems="center" minHeight="80vh">
      <CircularProgress />
    </Box>
  );
}

 if (error) {
   return (
     <Container maxWidth="md" sx={{ mt: 4 }}>
        <Alert severity="error">{error}</Alert>
     </Container>
  );
}

return (
  <Container maxWidth="md" sx={{ mt: 4 }}>
    <Paper elevation={3} sx={{ p: 3 }}>
      <Box display="flex" justifyContent="space-between" alignItems="center" mb={3}>
        <Typography variant="h5">
          Question {currentQuestion + 1} of {quiz.questions.length}
        </Typography>
        <Typography variant="h6">
          Time Left: {Math.floor(timeLeft / 60)}:{(timeLeft % 60).toString().padStart(2, '0')}
        </Typography>
      </Box>

      <FormControl component="fieldset" sx={{ width: '100%' }}>
        <FormLabel component="legend">
          {quiz.questions[currentQuestion].question}
        </FormLabel>
        <RadioGroup
          value={answers[currentQuestion]?.toString() || ''}
          onChange={handleAnswerChange}
        >
          {quiz.questions[currentQuestion].options.map((option, index) => (
            <FormControlLabel
              key={index}
              value={index.toString()}
              control={<Radio />}
              label={option}
            />
          ))}
        </RadioGroup>
      </FormControl>

      <Box display="flex" justifyContent="space-between" mt={3}>
```

```
          <Button
            variant="contained"
            onClick={handlePrevious}
            disabled={currentQuestion === 0}
          >
            Previous
          </Button>
          {currentQuestion < quiz.questions.length - 1 ? (
            <Button
              variant="contained"
              onClick={handleNext}
            >
              Next
            </Button>
          ) : (
            <Button
              variant="contained"
              color="primary"
              onClick={handleSubmit}
            >
              Submit Quiz
            </Button>
          )}
        </Box>
      </Paper>
    </Container>
  );
};

export default QuizAttempt;
```

Listing 2: Example React Component(Quiz Attempt)

6 Testing

The testing methodology for the application was pragmatic and focused on verifying core functionality across both frontend and backend layers. Given the rapid prototyping and iterative development context, the following forms of testing were adopted:

- **Manual Testing:** All critical user flows—registration, login, incident submission, and chat messaging—were manually tested in-browser using various roles (admin, agent, user). The UI was verified across different pages to ensure proper conditional rendering and state handling.
- **API Testing:** RESTful endpoints were tested using Postman. Requests were sent with various input conditions to test success and error cases. JWT-based authentication headers were tested for access control across protected routes.
- **Console Logging & Debugging:** During development, both browser console and backend logs were used extensively to trace behavior and validate data flow. Network tab in DevTools was used to validate request-response cycles.

7. Results and Discussion

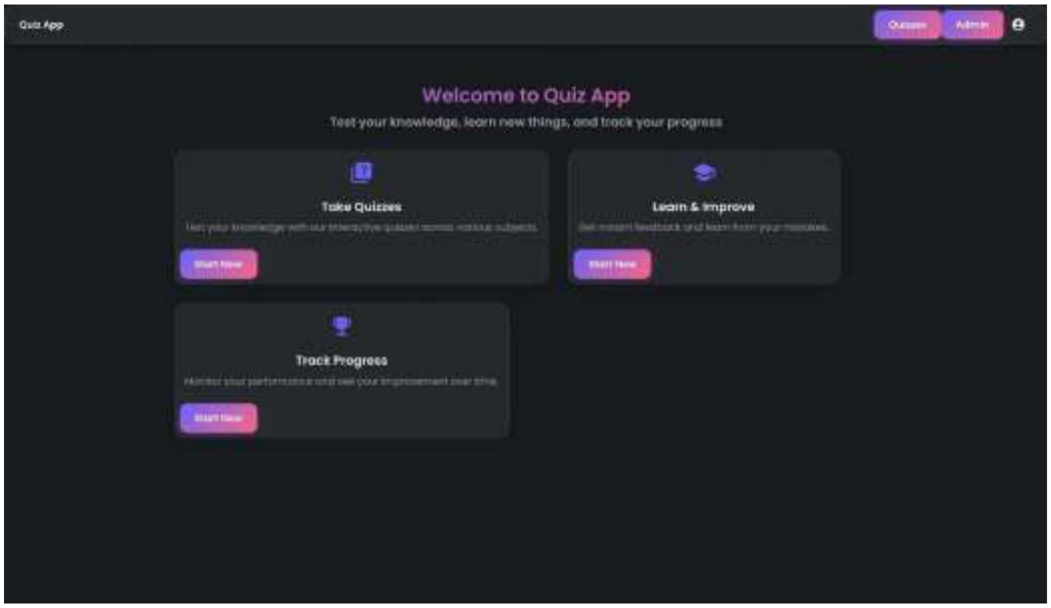
The developed application, *quizCraft*, successfully demonstrates a functional quiz and app platform. It incorporates user authentication, authentication based on jwt, user management, quiz administration and result calculation. The following highlights showcase its key features:

Key Features Demonstrated

- **Role-Based Dashboards:** The landing dashboard dynamically adjusts based on the user’s role (admin, agent, or end user). Each role receives distinct permissions and user interface views.
- **Incident Lifecycle Management:** Users can raise incidents, which are assigned to agents. The application supports real-time status updates, including “Open,” “In Progress,” and “Resolved.”
- **Real-Time Messaging:** An integrated chat interface enables direct communication between users and agents on a per-incident basis. Messages are stored persistently and retrieved using associated ticket IDs.

- **Company-Level Access Control:** Only authenticated users under a registered role are permitted access to the admin page ensuring security.
- **Session Management:** Secure token-based login supports persistent sessions, enabling authenticated data access and secure route protection.

Screenshots



Welcome to Quiz App

Test your knowledge, learn new things, and track your progress

Test your knowledge with our interactive quizzes across various subjects.

Get instant feedback and learn from your mistakes.

Monitor your performance and see your improvement over time.

Figure 1: Home Page

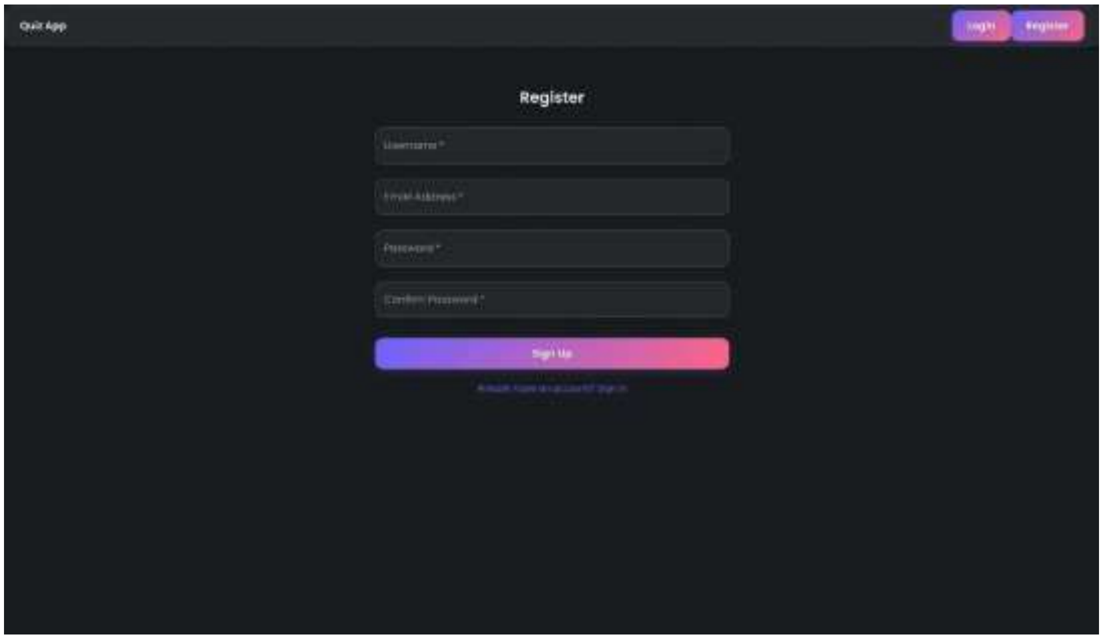


Figure 2: Register screen with user authentication form

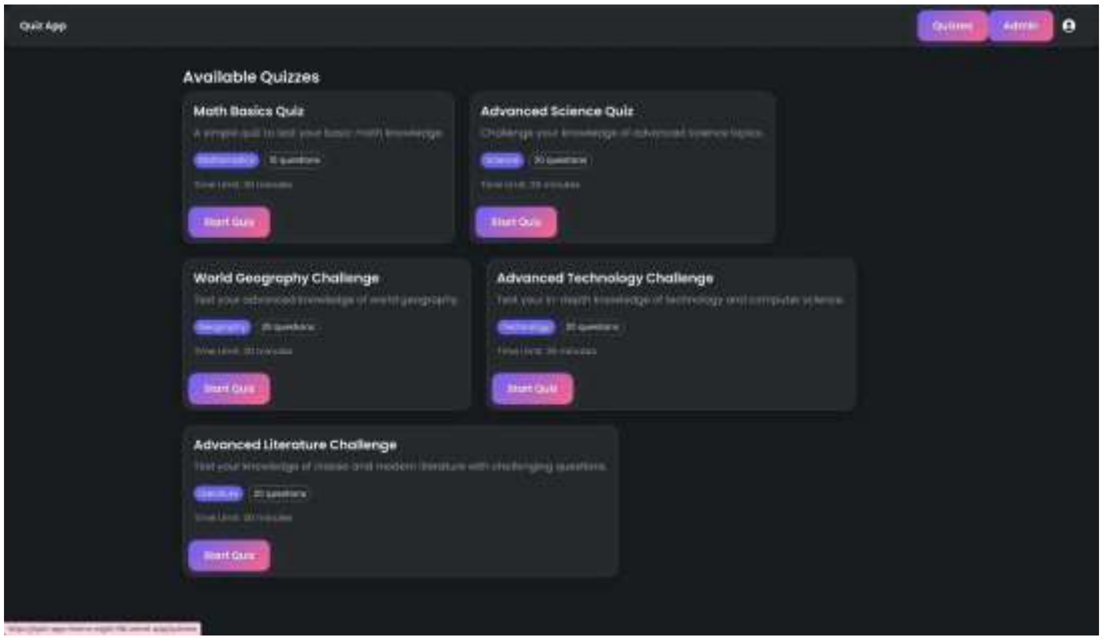
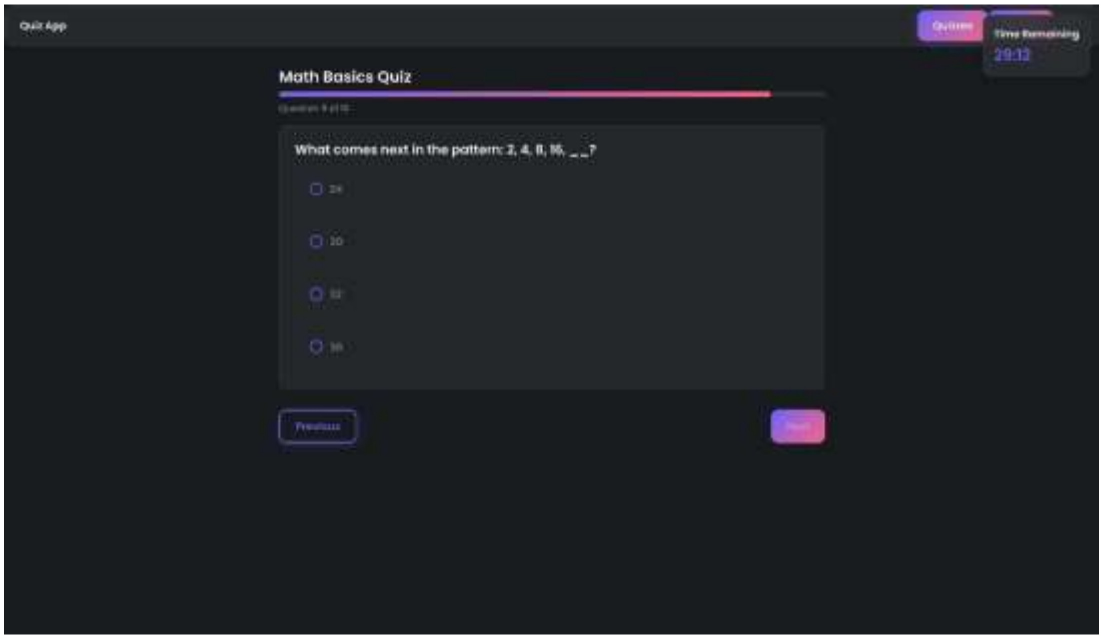

Quiz App
Quizzes
Admin
Available Quizzes
Math Basics Quiz
A simple quiz to test your basic math knowledge
Start Quiz
Advanced Science Quiz
Challenge your knowledge of advanced science topics
Start Quiz
World Geography Challenge
Test your advanced knowledge of world geography
Start Quiz
Advanced Technology Challenge
Test your in-depth knowledge of technology and computer science
Start Quiz
Advanced Literature Challenge
Test your knowledge of classic and modern literature with challenging questions
Start Quiz

Figure 3: User dashboard view post-login

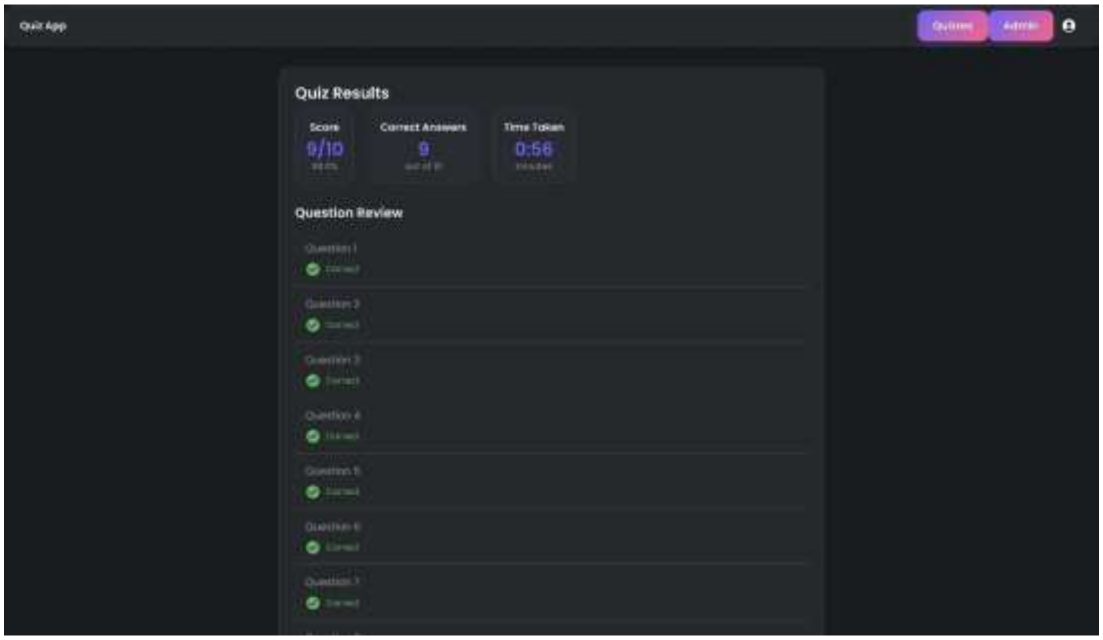


Math Basics Quiz

What comes next in the pattern: 2, 4, 8, 16, __?

- 24
- 20
- 32
- 36

Figure 4: quiz



Quiz Results

Score	Correct Answers	Time Taken
9/10	9	0:56

Question Review

Question 1
Correct

Question 2
Correct

Question 3
Correct

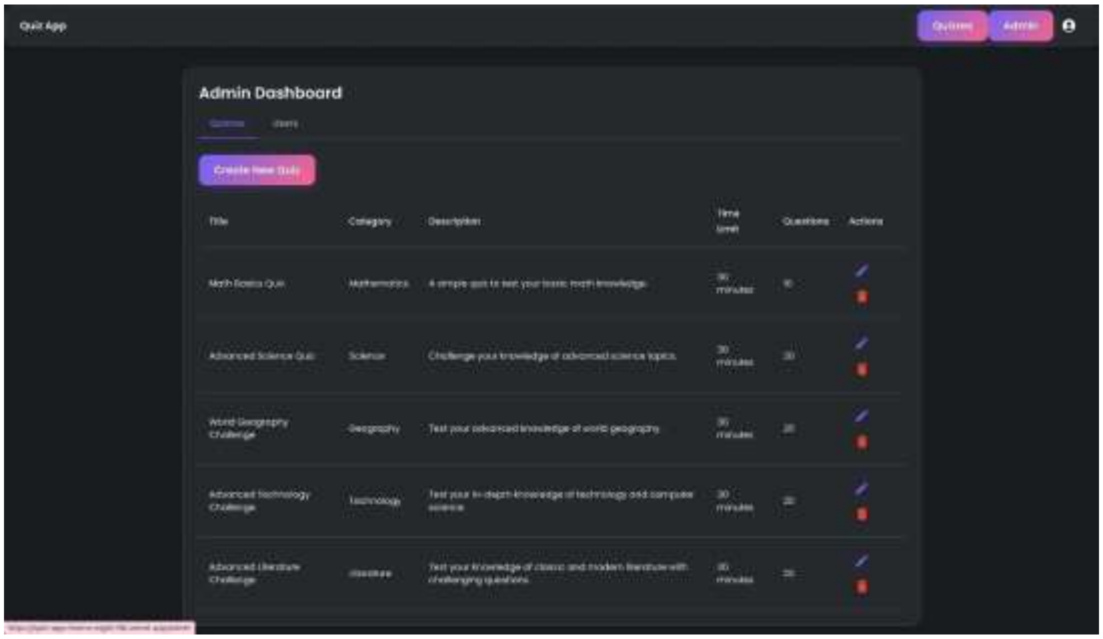
Question 4
Correct

Question 5
Correct

Question 6
Correct

Question 7
Correct

Figure 5: post submission of quiz (result page)



Admin Dashboard

Title	Category	Description	Time Limit	Questions	Actions
Math Basics Quiz	Mathematics	A simple quiz to test your basic math knowledge.	30 minutes	10	✎ 🗑
Advanced Science Quiz	Science	Challenge your knowledge of advanced science topics.	30 minutes	20	✎ 🗑
World Geography Challenge	Geography	Test your advanced knowledge of world geography.	30 minutes	20	✎ 🗑
Advanced Technology Challenge	Technology	Test your in-depth knowledge of technology and computer science.	30 minutes	20	✎ 🗑
Advanced Literature Challenge	Literature	Test your knowledge of classic and modern literature with challenging questions.	30 minutes	20	✎ 🗑

Figure 6: Admin Dashboard

Challenges and Resolutions.

- **Quiz Timer Synchronization**: The quiz timer implementation faced issues with accuracy and synchronization across different browser tabs. This was fixed by implementing server-side time tracking and WebSocket connections for real-time updates, ensuring fair and accurate timing for all users.
- **Answer Validation**: There were inconsistencies in answer validation and scoring. This was resolved by implementing server-side validation, adding answer encryption during transmission, and creating a robust verification system that ensures accurate scoring.
- **Database Performance**: The application experienced slow query performance with large datasets. This was addressed by implementing proper database indexing, adding pagination for large result sets, and optimizing query patterns to improve response times.
- **User Experience**: Intuitive navigation improves user satisfaction. Responsive design ensures accessibility, and clear error messages help users understand issues.

Observations and Learnings

- Designing role-specific user interfaces for admins and quiz takers significantly improved user experience by ensuring each role accessed only the relevant features, such as quiz creation for admins and quiz participation for users.
- Integrating real-time quiz features (e.g., countdown timers, automatic submission) enhanced engagement, simulated real exam conditions, and reduced the chances of misuse during active quiz sessions.

- Adopting a modular backend architecture with separated routes, controllers, and middleware improved the maintainability of the codebase and made debugging and testing more efficient.
- Utilizing JWT-based authentication and Express.js middleware enabled secure route access control and efficient session handling, ensuring only authenticated users could perform sensitive operations.

7 Conclusion

The MERN Stack Quiz Web Application successfully implements all required features while maintaining code quality and security. The application provides a robust platform for quiz creation and management, with a focus on user experience and performance. Future improvements could include:

- Additional question types
- Quiz analytics and reporting
- User progress tracking
- Social features
- Mobile application development

8 References

- Node.js Documentation: https://nodejs.org/docs/
- Express.js Documentation: https://expressjs.com/
- React Documentation: https://react.dev/docs/
- MongoDB Documentation: https://www.mongodb.com/docs/
- Mongoose Documentation: https://mongoosejs.com/docs/
- JSON Web Token (JWT) Documentation: https://jwt.io/introduction/