Major Project Report

On

QuizzMe! - A Quiz Platform

Submitted By

Ashwin S. Nambiar

At

**No.13-14, 2nd floor, Kothnur Main Rd,
JP Nagar 7th Phase, Bengaluru,
Karnataka 560078**

Guided by: Lekha Savale Ma'am

Date: 30 October 2024

Abstract

The **"QuizzMe!"** project is a dynamic and interactive quiz application designed to provide users with a fun and engaging platform to test their knowledge on various topics. The primary objective of this project is to offer an intuitive user interface, where users can select quiz categories, answer questions, and receive feedback on their performance. The application is developed with **React** for a seamless, responsive experience and relies on **API** integration to fetch a wide array of quiz questions from diverse categories. This makes the platform adaptable to a broad audience, allowing for continuous updates and expansion of question sets without major modifications to the core application structure.

In terms of functionality, the project includes a streamlined approach to quiz navigation, progress tracking, and scoring. Users can start quizzes by selecting their desired categories, their difficulty level, type of questions and the amount of questions they wish to answer in a simple click. The quiz results are displayed at the end of each session, showing a summary of correct and incorrect answers, along with an overall score. This score can be saved locally, encouraging users to revisit the application to improve their scores over time. Also the website is responsive so users are able to access the website from devices of different screen sizes without any issues.

From a technical perspective, the project architecture emphasizes performance, scalability, and ease of maintenance. **React** was chosen for its component-based architecture, allowing the development team to build reusable components for different parts of the quiz interface. The app also integrates state management to handle dynamic data updates effectively, ensuring that user interactions feel fluid and responsive. With **Vercel** as the hosting platform, the application benefits from reliable and fast deployments, while the API-driven question data can be updated externally without redeployment, making the project a flexible and scalable solution for quiz enthusiasts and learners alike.

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Overview

1.1.1 Project Goals

The website aims to create a user-friendly, interactive quiz application that provides an engaging and enjoyable experience for users of all ages and backgrounds. The core objectives are to deliver a dynamic interface where users can easily select quiz categories, attempt questions, receive immediate feedback, and view their final scores. The application is designed to be easily expandable, allowing the addition of new quiz categories and questions without extensive reconfiguration, making it a versatile platform for both casual and dedicated users interested in knowledge-based games.

1.1.2 Technical Approach

The project is built with **React**, which provides a modular and component-based structure that simplifies the development of a responsive user interface. React's efficient rendering and state management capabilities ensure smooth interactions throughout the app, with components designed for each core function: quiz selection, question display, scoring, and result summary. No other CSS frameworks were used to provide styling for this project. To source questions, the application relies on **Open Trivia DB API**, which dynamically fetch questions for each selected category, difficulty, type of questions and number of questions allowing for flexibility and an up-to-date quiz experience. The application is hosted on **Vercel**, which provides scalable, fast, and reliable deployments, ensuring optimal performance across devices.

1.2 Objectives

The main objectives of this project are:

- **Deliver an Engaging User Experience**: Create an interactive and user-friendly platform where users can select quiz topics, answer questions, and receive instant feedback, making learning enjoyable and motivating.
- **Support a Wide Range of Quiz Categories**: Enable the app to cater to diverse user interests by integrating multiple quiz categories, allowing users to explore different topics and expand their knowledge.

- **Provide Feedback and Scoring**: Offer real-time feedback on each answer after finishing the quiz as well as a final score summary, helping users understand their strengths and areas for improvement.
- **Ensure High Performance and Responsiveness**: Build the application to be fast, responsive, and accessible across devices, with quick load times and smooth transitions to enhance user engagement.
- **Implement a Scalable Structure for Future Expansion**: Design the application with a component-based architecture, enabling easy addition of new categories, questions, and features without extensive changes to the existing codebase.
- **Utilize External APIs for Dynamic Content Updates**: Integrate an external API to fetch questions dynamically, keeping quiz content fresh and allowing for seamless updates without redeployment of the application.
- **Create a Reliable and Maintainable Codebase**: Develop a clean, well-documented codebase using React, ensuring easy maintenance, debugging, and future development for ongoing improvements and optimizations.

Chapter 2

Key Features

The key features of QuizzMe! are:

1. **Dynamic Quiz Generation**
    - **API Integration**: The app dynamically fetches quiz questions from the Open Trivia Database API, allowing users to access a wide variety of questions without needing to hard-code them into the app.
    - **User-Defined Parameters**: Users can customize their quiz by selecting options such as category, difficulty, question type, and the number of questions. These options are passed to the API to generate a quiz tailored to the user's preferences.
2. **Real-Time Question Interaction**
    - **Answer Selection and Validation**: Users can select answers to each question, which are visually highlighted to indicate their choice.
    - **Immediate Feedback**: Upon completion, users can check their answers, revealing correct and incorrect responses with visual feedback such as icons or color changes, enhancing user engagement and learning.
3. **Confetti Celebration on Perfect Score**
    - **Confetti Animation**: A celebratory confetti effect appears if the user achieves a perfect score, using the react-confetti library. This feature adds a sense of reward and accomplishment, especially useful for enhancing user engagement and motivation.
4. **Responsive Design**
    - **Device Compatibility**: The app includes responsive styles, adapting seamlessly across different devices and screen sizes, which allows users to enjoy the app on desktops, tablets, and smartphones without layout issues.
    - **Dynamic Animation Delays**: Questions are animated with incremental delays based on their order, providing a smooth and engaging transition effect on various screen sizes.

5. **Error Handling and Edge Case Management**

   - **No Questions Available Error**: If the selected options yield no questions from the API, the app gracefully handles this by notifying the user and allowing them to adjust their settings.
   - **Empty Answer Prevention**: The "Check Answers" button remains disabled until all questions have an answer selected, preventing incomplete submissions and improving the user experience.

6. **State Management and Modularity**

   - **React State Management**: The app uses React's useState and useEffect hooks to manage user interactions, question data, and animations efficiently, ensuring a smooth user experience.
   - **Component Modularity**: The app is structured with reusable components (Question, QuestionList, etc.), each responsible for specific parts of the UI and logic, resulting in better code organization, readability, and easier maintenance.

7. **Stylized User Interface**

   - **Custom Button and Icon Styling**: The app includes unique button states and icons to distinguish selected, correct, and incorrect answers, providing intuitive visual cues.
   - **Animated Button and Score Display**: Buttons and score displays are styled to give a polished feel to the app, making the quiz experience more interactive and visually appealing.

Chapter 3

Technologies Used

3.1 Front-end

The front end of the application is built using the following technologies:

- **React**: A JavaScript library for building user interfaces.
- **CSS**: For styling the application and ensuring a responsive design.
- **React-Confetti**: An NPM package for creating confetti animation in React.
- **React-Use**: An NPM package which contains some of predefined hooks, in this project we use the useLocalStorage hook from this package.
- **Nano ID**: An NPM package for generating IDs that can be used in React.
- **HTML-Entities**: An NPM package that is used to encode and decode HTML entities.
- **Open Trivia DB API**: The Open Trivia Database provides a completely free JSON API for use in programming projects.

3.2 Project Setup & File Structure

3.2.1 Project Setup

- **Vite**: Vite is a blazing fast front-end build tool powering the next generation of web applications. To get started to create a react app with Vite use:
  npm create vite@latest and then follow the instructions given.

- To launch developer server use: *npm run dev*

3.2.2 File Structure

Figure 3.1: File Structure for the Project

- **dist**: the dist (short for "distribution") folder is typically where the final, compiled, and optimized version of your application is stored. This version is what you deploy to production or serve to users.
- **node_modules**: the node_modules folder is a critical component in any Node.js project, as it stores all the dependencies and packages needed for the application to run.
- **public**: public directory is used to store static assets that should be served as-is, without being processed or bundled by Vite.
- **src**: the src directory is the primary folder where all the source code lives. The src directory houses all the components of your React app, allowing you to organize them into separate files or subdirectories.
- **src/assets**: this directory contains all the needful assets for the website.
- **src/components**: this directory contains the reusable parts of the website like Question.jsx which contains the Question component and QuestionList.jsx which holds all the questions together in a component and also the css files for their own specific styling.
- **src/services**: contains the getQuestions.jsx file which is used to get the questions from the **Open Trivia DB API** and returns them to be exported.
- **App.jsx**: is the main component that serves as the root component of your application. It's typically the primary entry point for your app's component tree and plays a central role in defining the structure, routes, and global functionality of the app.
- **App.css**: the styling of App.jsx is done in this file.
- **index.jsx**: it's the starting place where the application is initialized and rendered into the DOM.
- **index.css**: the styling for the overall website.
- **index.html**: is an essential file that serves as the foundational HTML structure for your application.
- **package.json**: this file is a crucial component of any Node.js or JavaScript project, including those built with frameworks like React or Vue. It serves multiple purposes, acting as a manifest for your project and providing essential information about the project and its dependencies.
- **vite.config.js**: is the configuration file used to customize and optimize the Vite build process. It allows you to specify various settings and plugins to tailor Vite to your specific project needs.
- **.gitignore**: file is a critical part of any Git repository that specifies which files and directories should be ignored by Git when committing changes.
- **README.md**: it is a markdown file which is used to provide an idea about the project at hand.

3.3 Core Components

1. **App.jsx**:
   The App.jsx component is a React application that serves as a quiz game interface. It allows users to select various game options, such as category, difficulty, question type, and the number of questions before starting the quiz. The component includes functionality for dark and light themes, which can be toggled using an icon button. The main interface is divided into two sections: a game introduction with options for users to set their preferences and the game container that displays the questions once the quiz starts. Additionally, it also handles errors related to unavailable questions.

   - **Imports**
     - **React Hooks**: The component imports useEffect and useState from React for managing state and side effects.
     - **CSS**: The component styles are imported from App.css.
     - **Child Components**: It imports QuestionList from ./components/QuestionList/QuestionList for displaying the questions during the quiz.
     - **Custom Hook**: It uses useLocalStorage from the use-local-storage library to manage the theme preference.
   - **Icons**
     - **SunIcon and MoonIcon**: These are functional components that render SVGs representing the sun and moon, respectively. They are used for toggling between dark and light themes.
   - **State Management**
     - **gameStarted**: A boolean state that tracks whether the quiz is currently active.
     - **showNoQuestionsError**: A boolean state that determines if an error message should be displayed when no questions match the selected criteria.
     - **gameOptions**: An object state that holds the user's selected options for the quiz (category, difficulty, type, number of questions).
     - **showFooter**: A boolean state that controls the visibility of the footer containing developer credits.
     - **darkTheme**: A boolean state that indicates whether the dark theme is active, persisted in local storage.
   - **Effects**
     - **useEffect**: This hook runs when the darkTheme state changes. It adds or removes the "dark" class to the body element, which applies the corresponding styles.
   - **Event Handlers/Functions**
     - **toggleFooter**: A function that toggles the showFooter state, controlling the visibility of the footer.
     - **handleGameStart**: This function toggles the game state with a slight delay to allow for any necessary transitions.

- **handleNoQuestionsError**: A function that updates the showNoQuestionsError state based on whether an error occurs.
- **handleChange**: This function updates the gameOptions state based on the user's selections from dropdown menus. It destructures the name and value from the event target to ensure that the appropriate field in the gameOptions object is updated.
- **toggleTheme**: A function that toggles the darkTheme state and the corresponding class on the body element to switch between dark and light themes.

- **Render Logic**
  - The component conditionally renders either the game interface or the introductory setup based on the gameStarted state.
  - Inside the game setup, it includes:
    * A theme toggle button displaying either the SunIcon or MoonIcon.
    * An error message that displays if no questions are found.
    * Dropdown menus for selecting quiz options (category, difficulty, type, number of questions).
    * A button to start the quiz.
  - A footer with developer credits appears based on the showFooter state.

2. **QuestionList.jsx**:
   The QuestionList.jsx component is responsible for managing and displaying a list of quiz questions in a React quiz game. It interacts with the API to fetch questions based on user-selected options and tracks user answers to each question. The component features animations for question rendering and displays the total number of correct answers upon game completion. Additionally, it utilizes a confetti effect to celebrate the player's success if they answer all questions correctly. It handles loading states, user interactions for selecting answers, and game state management.

   - **Imports**
     - **React Hooks**: It imports useState and useEffect from React for managing state and side effects.
     - **Utility Libraries**: The component uses nanoid for generating unique IDs for each question.
     - **CSS**: Styles are imported from QuestionList.css for custom styling.
     - **Child Components**: It imports the Question component for rendering individual questions and the getQuestions service for fetching quiz questions from an external source.
     - **Confetti Effect**: The Confetti component is imported for visual feedback when the user answers all questions correctly.
     - **Window Size Hook**: It imports useWindowSize from react-use for handling window resizing.
   - **State Management**
     - **questionsArray**: An array state that holds the question objects, each with their respective selected answers and other properties.

- **checkAnswerBtn**: A boolean state that controls the visibility of the button for checking answers.
- **correctAnswersCount**: A number state that tracks how many answers the user got correct.
- **isGameOver**: A boolean state indicating if the game has concluded.
- **showConfetti**: A boolean state that triggers the confetti effect upon a successful game.
- **isLoading**: A boolean state that indicates whether the questions are still being fetched.
- **pageHeight**: A number state that tracks the height of the page, useful for adjusting the confetti display.

- **Effects**
  - **Window Resize and Scroll Effect**: This effect listens for window resizing and scrolling events, updating the pageHeight state accordingly.
  - **Fetch Questions**: This effect runs once on component mount, calling getQuestions with the gameOptions. It checks if questions are fetched; if not, it triggers the error handling. If questions are found, it sets them into the questionsArray with additional properties (like id and selectedAnswer), and updates the loading state.
  - **Check Answers Effect**: This effect runs whenever questionsArray changes, calculating the number of correct answers when all questions have been answered.
  - **Confetti Timer**: This effect triggers a timer to hide the confetti after a set duration when the showConfetti state is true.

- **Event Handlers/Functions**
  - **generateQuestionStyles**: This function generates CSS styles dynamically to create a staggered animation effect for rendering questions. Each question is given a unique animation delay based on its order.
  - **handleSelectAnswer**: This function updates the selected answer for a specific question when the user makes a choice, preventing changes if the game is over.
  - **checkAnswers**: This function checks if all questions have been answered and updates the game state accordingly. It sets isGameOver to true, triggers confetti if all answers are correct, and marks answers for display.
  - **resetGame**: This function resets the game state, preparing for a new round of questions by calling handleGameStart and resetting other relevant states.

- **questionTotal**
  - questionTotal is a derived constant that represents the total number of questions that the user has selected to answer in the quiz.
  - It is set using the value from gameOptions.questionno, which comes from the parent component (App.jsx) where the user specifies how many questions they want to answer.

- **allQuestionsAnswered**
  - allQuestionsAnswered is a boolean that checks whether all questions in the questionsArray have been answered by the user.
  - This variable is calculated using the every method on questionsArray, which iterates over each question object and checks if the selectedAnswer property is not an empty string ("").
  - If all questions have a selected answer, allQuestionsAnswered will be true; otherwise, it will be false.
- **Render Logic**
  - The component conditionally renders the confetti effect based on the showConfetti state.
  - It displays a loading message until the questions are fully fetched.
  - The main section of the component renders each question using the Question component, passing necessary props for rendering.
  - At the bottom, it displays the user's score if the game is over and includes a button to either check answers or reset the game.

3. **Question.jsx**:
   The Question component renders a single quiz question along with its answer options, allowing users to select their answers. It highlights correct and incorrect answers based on user interaction, and visually indicates whether the selected answer is right or wrong after submission. The component uses html-entities package to decode HTML entities to display and shows icons for correct and incorrect answers.

   - **Imports**
     - **CSS Styles**: The component imports its styles from Question.css to handle the visual layout.
     - **Dependencies**:
     - **nanoid**: This is used to generate unique keys for each answer button.
     - **html-entities**: It decodes HTML entities in the question and answers to ensure they are displayed correctly.
     - **Icons**: tickIcon and crossIcon are imported to visually represent correct and incorrect answers.
   - **Props**
     The component receives several props, including:
     - **id**: The unique identifier for the question.
     - **num**: The question number for display.
     - **question**: The actual text of the question.
     - **correctAnswer**: The correct answer for the question.
     - **incorrectAnswers**: An array of incorrect answer options.
     - **selectedAnswer**: The user's selected answer.
     - **showAnswer**: A boolean that indicates whether the correct answers should be displayed.

- **handleSelectAnswer**: A function to handle the selection of an answer.

- **Rendering Incorrect Answers**
  - **Mapping**: The component maps through props.incorrectAnswers to create a button for each incorrect answer.
  - **Conditional Classes**: Each button gets a dynamic class name based on whether it was selected or if it is shown as incorrect when showAnswer is true.
- **Rendering Correct Answer**
  - Similar to incorrect answers, a button for the correct answer is created. It also gets conditional styling based on the user's selection and whether the answer should be revealed.
- **Combining and Sorting Answers**
  - All answer elements, including both correct and incorrect answers, are pushed into an array, which is then sorted alphabetically based on the answer text to provide a consistent order of presentation.
- **Rendering the Question**
  - The main rendering of the component consists of:
    * An article element that contains the question text.
    * A div for the answer buttons.
    * An optional section that displays an icon (tick or cross) indicating whether the selected answer was correct or incorrect, depending on the showAnswer prop.
- **User Interactions**
  - **User Interaction**: Users can select an answer by clicking the corresponding button. The component captures this action through the handleSelectAnswer prop function.
  - **Visual Feedback**: The component visually communicates the correctness of answers post-submission, enhancing user experience and engagement during the quiz.

4. **getQuestions.jsx**:
   The getQuestions function is an asynchronous utility that fetches quiz questions from the Open Trivia Database API based on user-defined game options. It dynamically constructs the API URL with parameters like category, difficulty, type, and the number of questions, depending on the values provided in the gameOptions object. The function returns an array of quiz questions based on the response from the API.

   - **Function Parameters**
     The function accepts a single parameter, gameOptions, which is an object containing the following properties:
     - **category**: The ID of the category of questions (string).
     - **difficulty**: The difficulty level, such as "easy," "medium," or "hard" (string).

  - **type**: The type of questions, such as "multiple choice" or "true/false" (string).
  - **questionno**: The number of questions requested (number).

- **Query Parameter Variables**
  For each property of gameOptions, the function initializes a query parameter variable. If the corresponding property is not an empty string, it is added to the respective query parameter variable as a URL parameter. If the user has set the options to "Any" then the questions can have varying categories, difficulty and type.
  - **categoryQueryParam**: If category is specified, it is set as &category = categoryID and append to the URL.
  - **difficultyQueryParam**: Set as &difficulty = difficultyLevel, append to the URL if difficulty is provided.
  - **typeQueryParam**: Set as &type = questionType, append to the URL if type is specified.
  - **questionnoParam**: This defaults to 1, but if questionno is provided, it replaces this default and appended to the URL.
- **API URL Construction**
  - The base API URL is built by appending the parameters created above to the apiUrl string, which queries the Open Trivia Database API.

```
let apiUrl = `https://opentdb.com/api.php?
    amount=${questionnoParam}\
${categoryQueryParam}${difficultyQueryParam
    }${typeQueryParam}`;
```

- **Fetch Request and JSON Parsing**
  - The function uses fetch to make an HTTP request to apiUrl.
  - It then awaits the response, converts it to JSON, and accesses the results field, which contains an array of questions.
- **Return Value**
  - The function returns an array of quiz question objects retrieved from the API. Each object contains details about a question, such as the question text, correct answer, and incorrect answers.
- **Usage Scenario**
  - This function is typically called in a component responsible for managing the state of the game, such as QuestionList, to retrieve questions based on user-selected options before rendering them in the quiz.

3.4 Back-end

This project does not include a back-end, the architecture can be easily extended to incorporate server-side functionality using Node.js and Express. To properly set up we must choose a technology stack, define API endpoints, selecting a database and ensure secure and efficient data management.

Chapter 4

Implementation

4.1 Landing Page

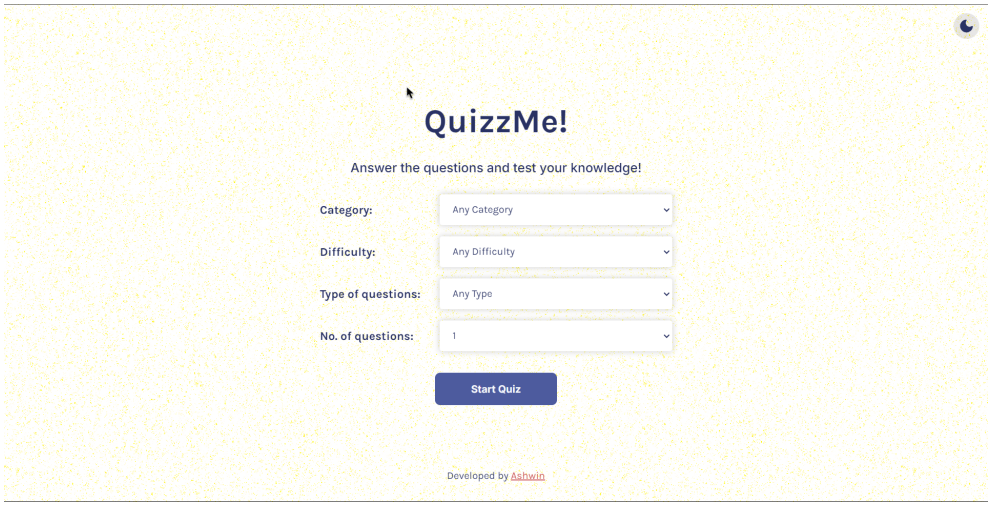

Figure 4.1: Landing Page

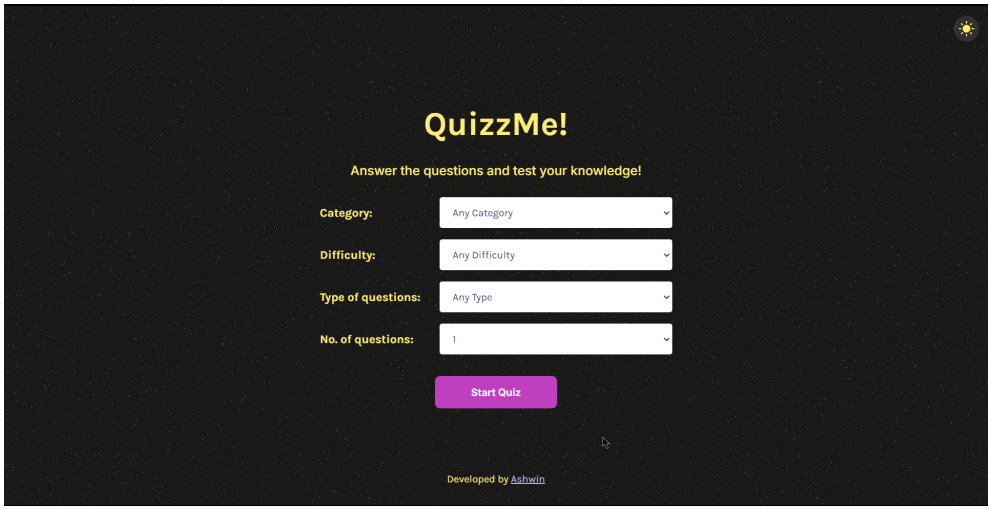

Figure 4.2: Landing Page - Dark Mode

The above given figures 4.1 and 4.2 show the landing page for the project, here the users are greeted with a form where they are able to select and customize the categories, difficulty, type and number of questions they wish to answer as part of the quiz. They also show the dark and light themes of the project which can be switched using the toggle at top-right corner.

4.2 Customizing Questions

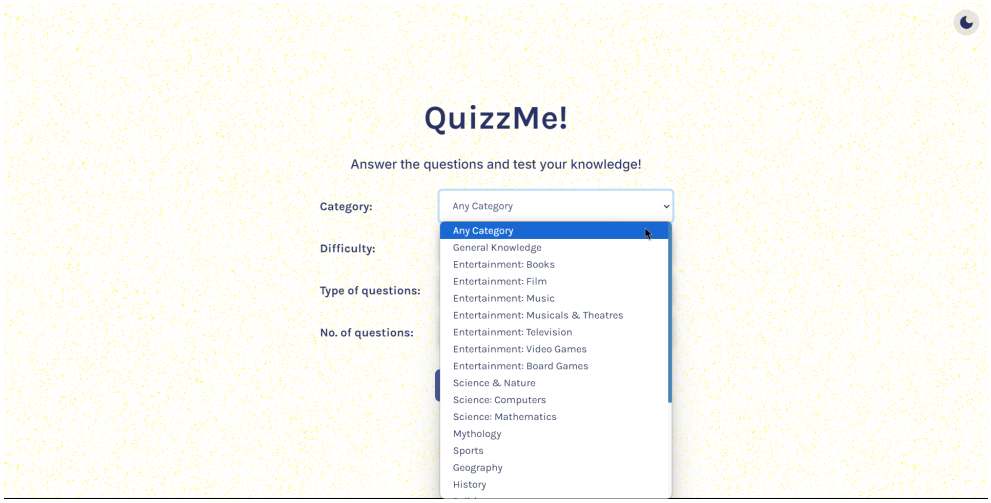

Figure 4.3: Options - Category

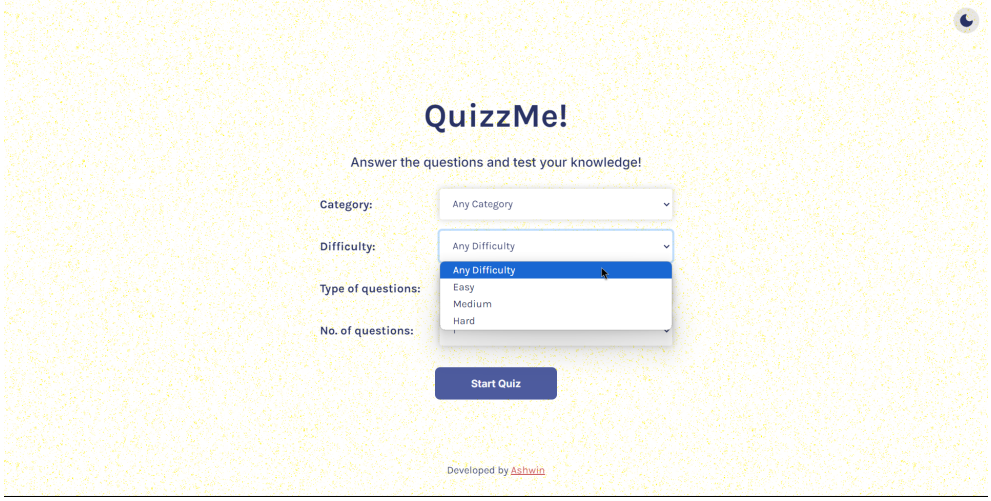

Figure 4.4: Options - Difficulty

In the figures 4.3,4.4,4.5 and 4.6 we can see the different options that are available to the users in terms of categories, difficulty level that the users can set, if the options are left at any then the questions will be from different topics randomly and of various difficulty levels.

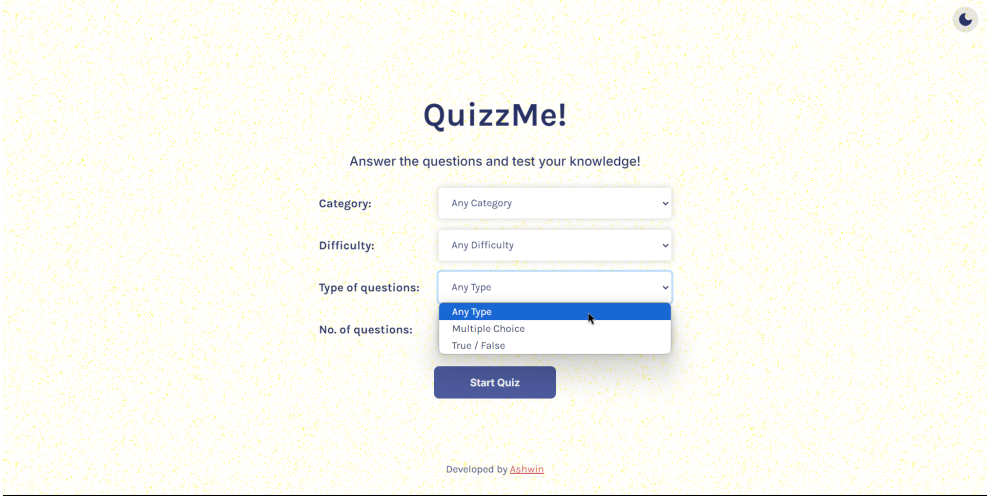

Figure 4.5: Options - Type

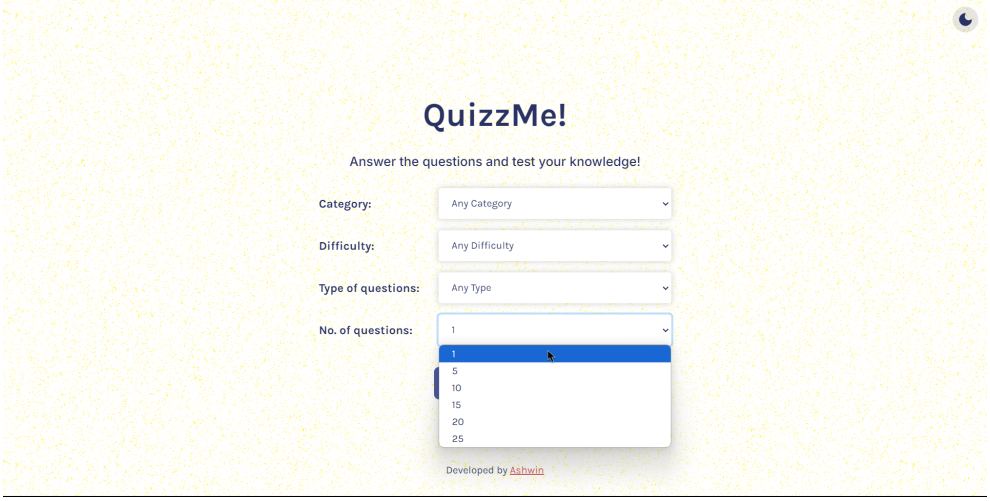

Figure 4.6: Options - QuestionNo

In the figures 4.5 and 4.6, the type of questions and number of questions can be selected from the options present by the users and along with earlier options for category and type are used in order to source the questions from the Open Trivia database. After selecting the the options clicking the "Start Quiz" button shows the questions page.

4.3 Questions Page

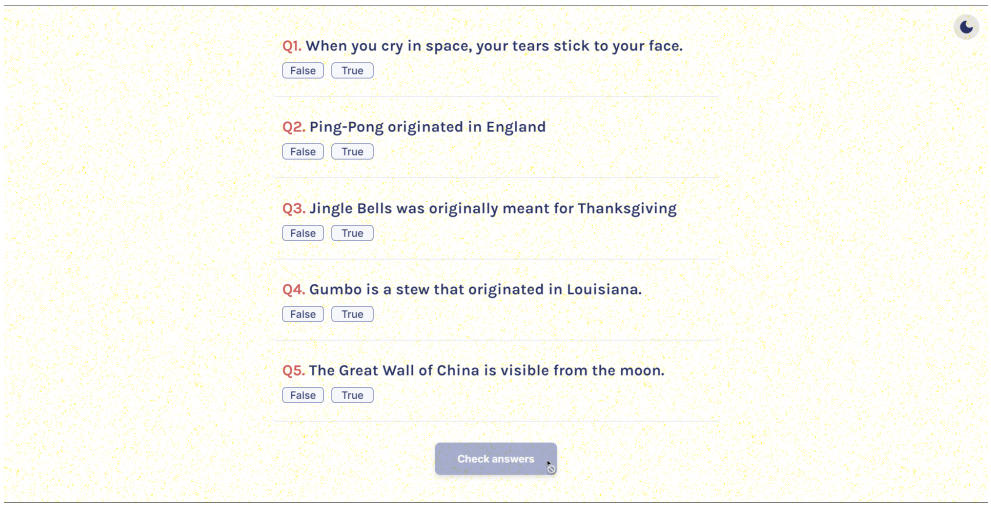

Figure 4.7: Questions Page

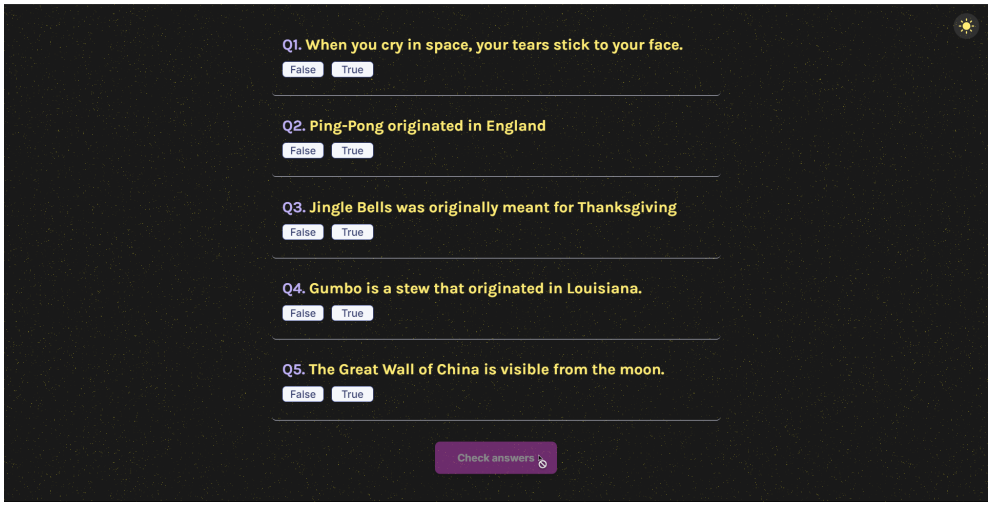

Figure 4.8: Questions Page - Dark Mode

In the Questions page we can see the the questions according to the customizations from the user and the user is able to select the answers.

In the below figures 4.9,4.10 and 4.11 we can see the answers selected and the score reveal page which is shown after the user selects the answers for all the questions, then the "Check answers" button is enabled which when clicked reveals the score and "Play again" button to restart the game.

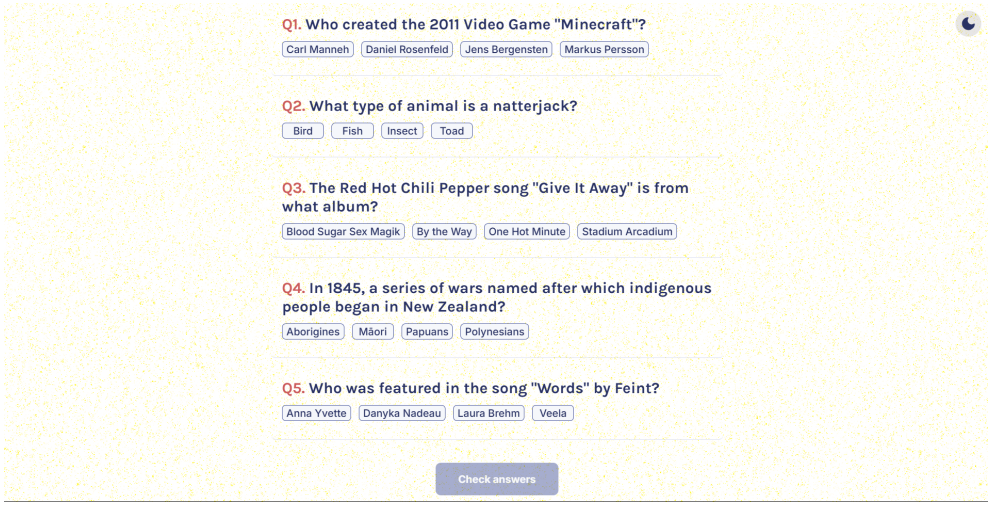

Figure 4.9: Questions Page - Any Options

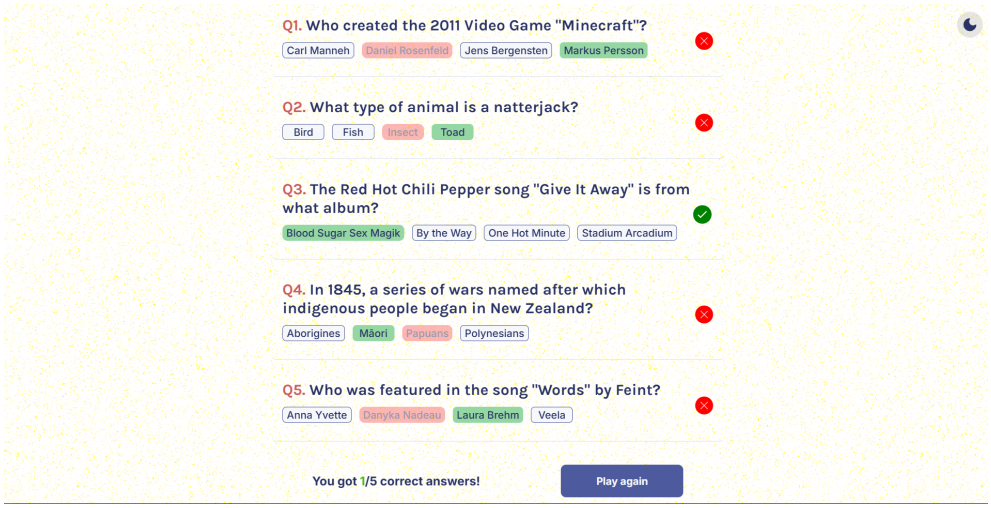

Figure 4.10: Questions Page - Any Options Answers

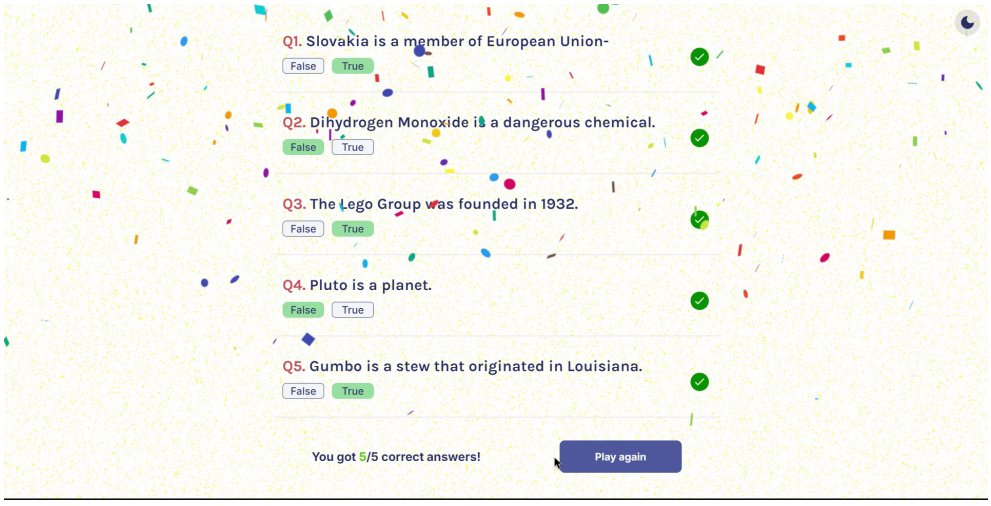

Figure 4.11: Questions Page - Perfect Answers

Chapter 5

Responsiveness

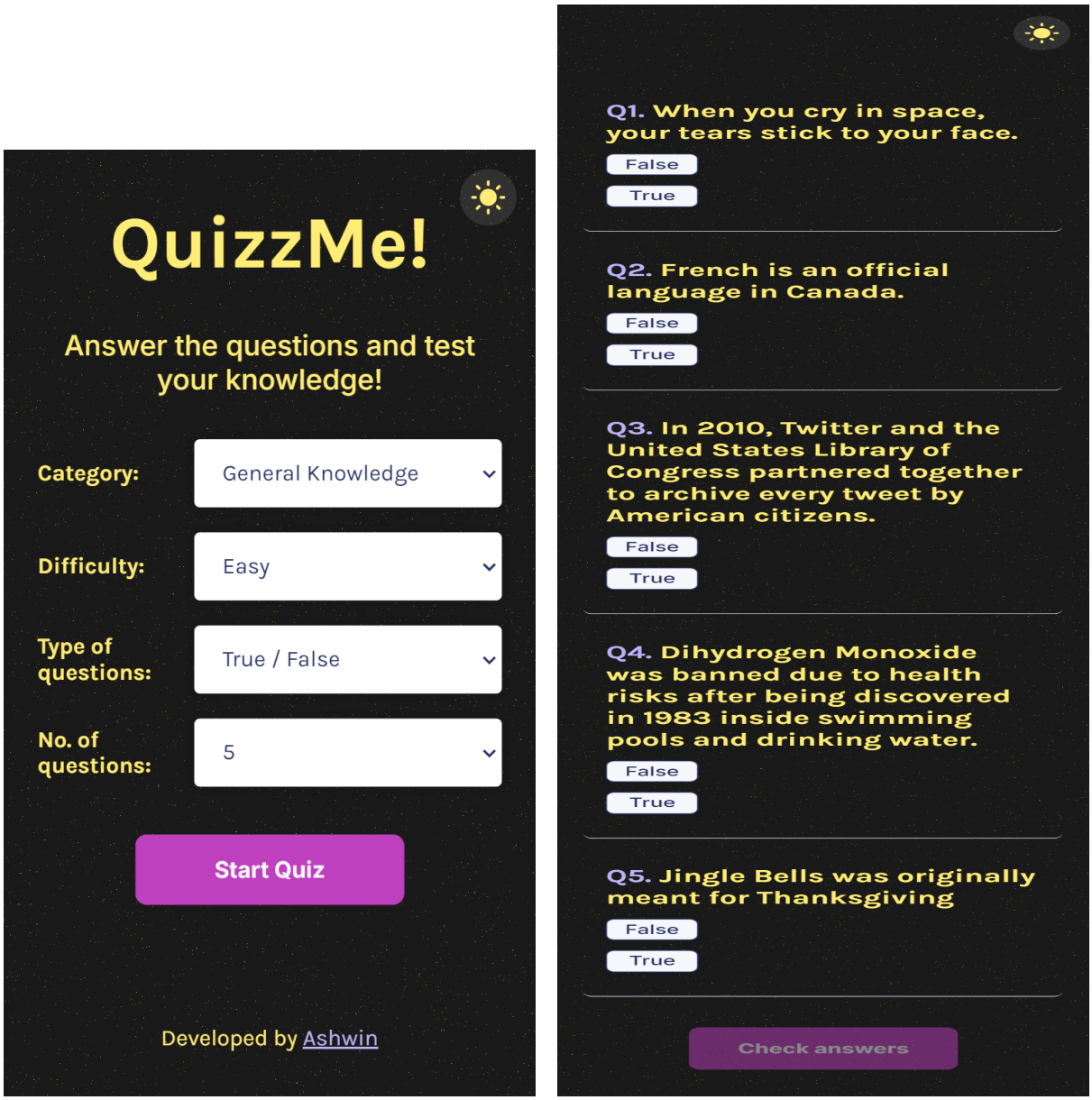

Figure 5.1: Mobile Screen Size Responsiveness

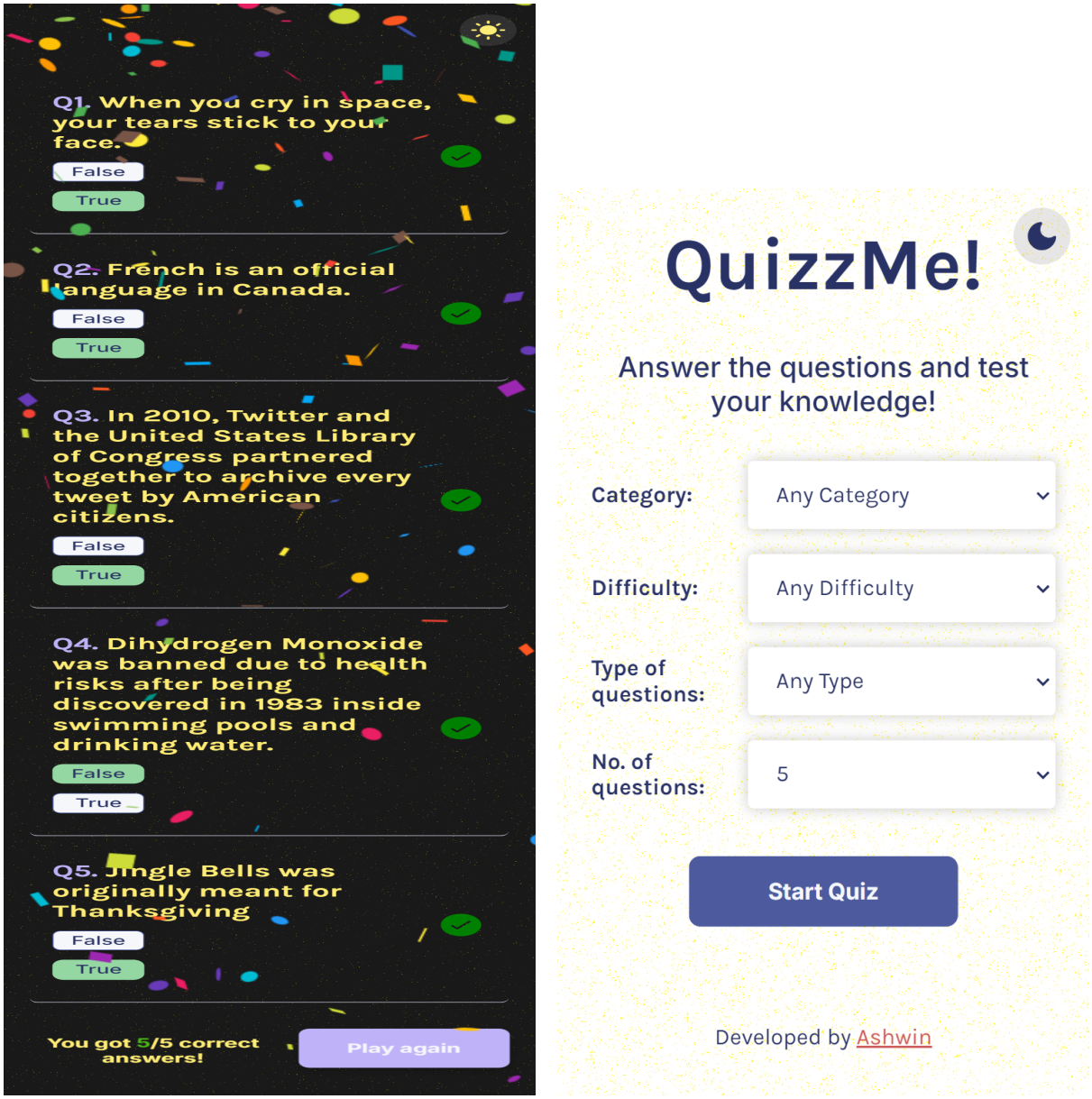

Figure 5.2: Mobile Screen Size Responsiveness Contd

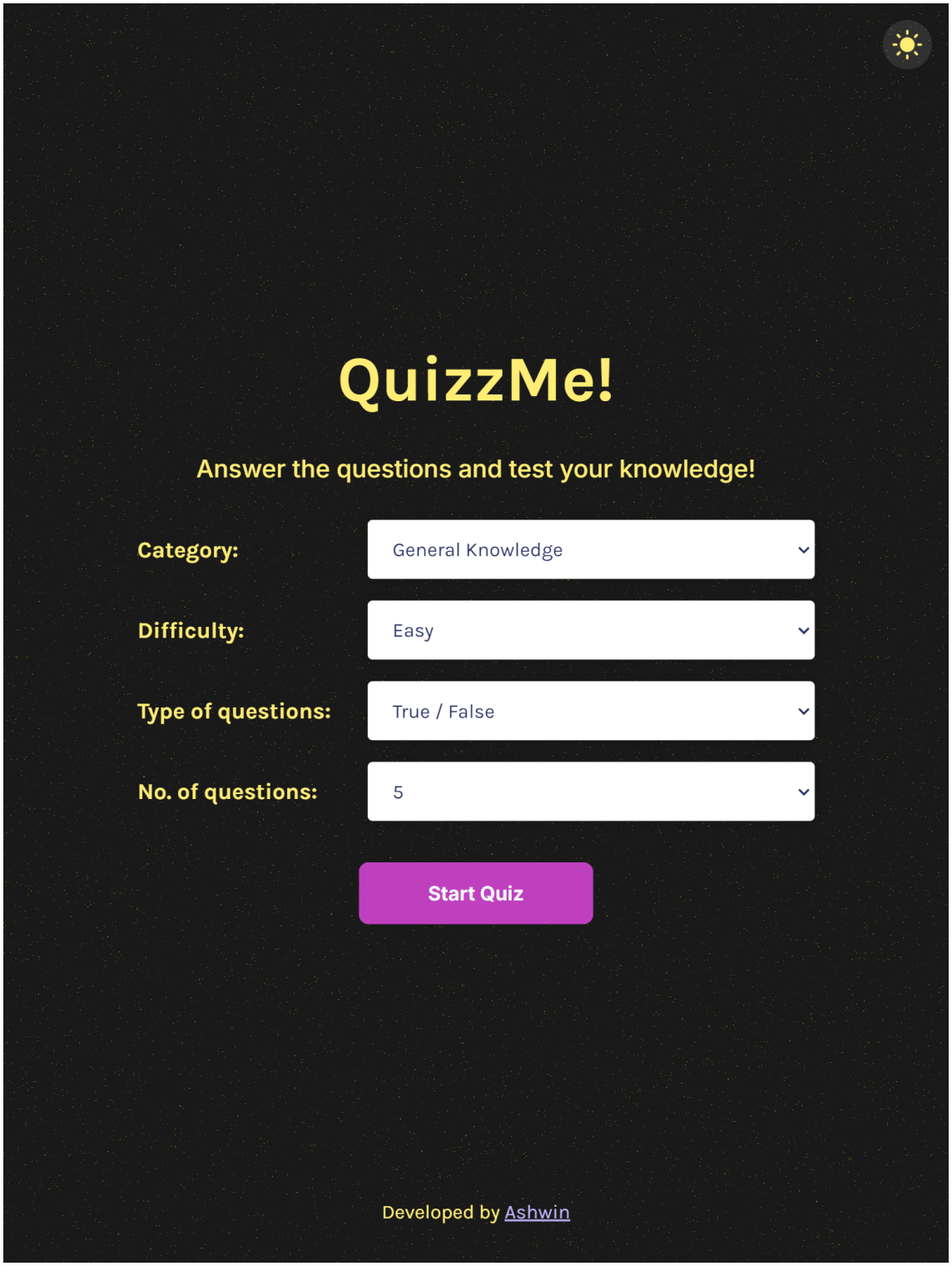

Figure 5.3: Tablet Screen Size Responsiveness

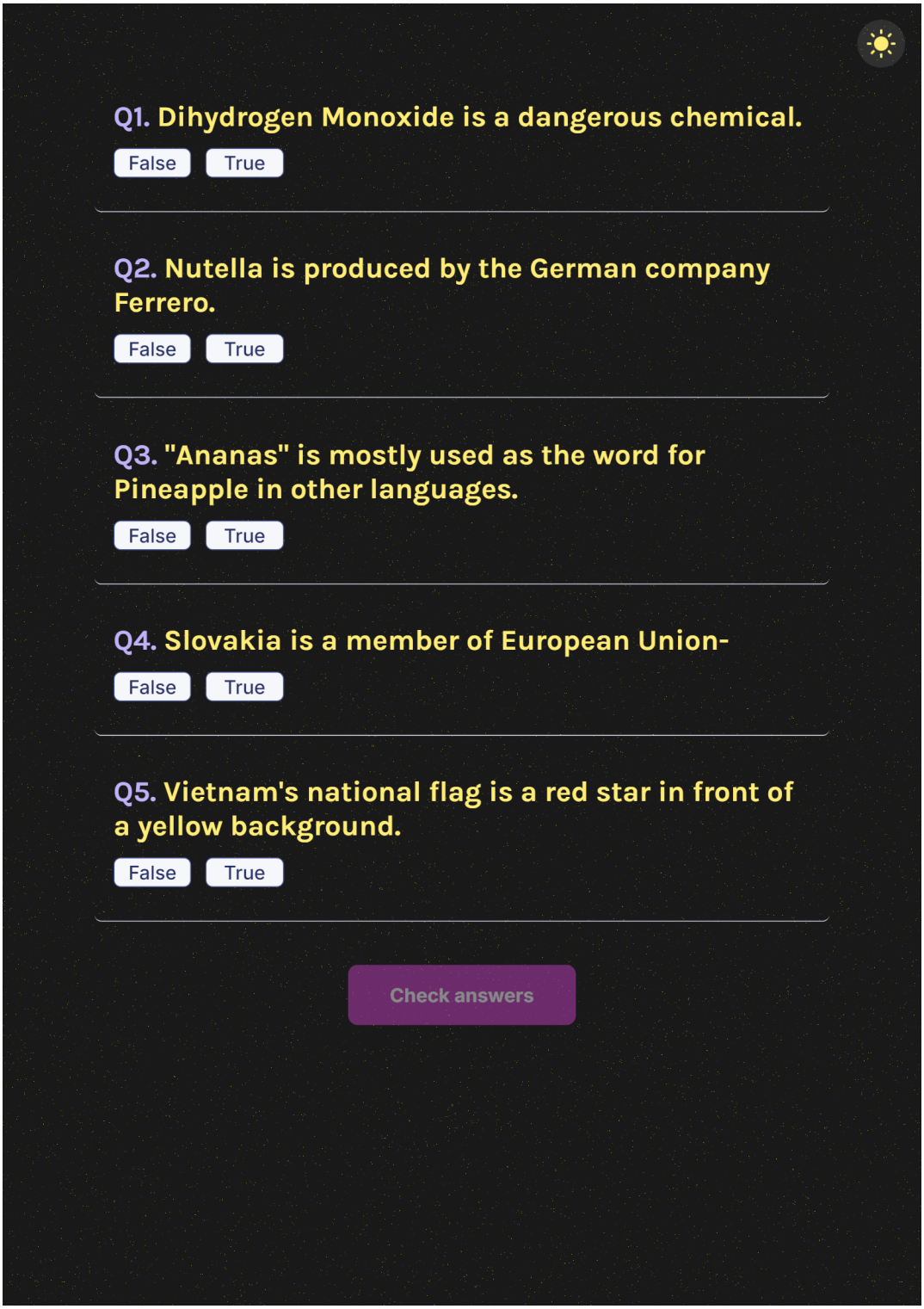

Figure 5.4: Tablet Screen Size Responsiveness Contd

Chapter 6

Challenges Faced

The following challenges were encountered during development:

- Main challenge was to make the site responsive to various screen sizes.
- Understanding the Open Trivia DB API and how to modify the API URL to customize the questions to the user's needs.
- Adding animations using "keyframes" as well as transitions to improve the aesthetics and user experience of the website.
- Refactoring the code after ensuring proper functioning was a good challenge in a bid to simplify the codebase.
- Learning more about React hooks and understanding their working properly as it's an essential part for this project.

Chapter 7

Future Improvements

Future enhancements as well as improvements planned:

- Implement react router for a better routing and navigation experience.
- Adding a timer which on expiring finishes the quiz even if questions are not fully answered and displays the score and message.
- Improving user experience and better animations and transitions.
- Changing the current questions layout to make it show one question at a time and buttons for previous and next questions as well.
- To add a back-end to the website and explore the features and improvements as a full-stack project if possible.
- Adding a leader-board for most questions answered correctly for the users.

Chapter 8

Summary

The QuizzMe! application is an interactive web-based platform designed to engage users in trivia quizzes across various topics. Built using React, the application allows users to customize their quiz experience by selecting options such as category, difficulty, and question type. It retrieves questions from the Open Trivia Database API, providing a dynamic and varied quiz experience. Users can select their answers, receive immediate feedback, and enjoy celebratory animations upon achieving perfect scores. The application features a responsive design that adapts to different screen sizes, ensuring accessibility across devices.

The project aims to enhance user interaction by providing a fun and educational way to test knowledge. Future enhancements could include user authentication for personalized experiences, enabling users to track their progress and save their results. Additionally, the implementation of a back-end system using Node.js and a database like MongoDB or PostgreSQL could facilitate better data management and storage. Overall, this project combines an engaging user interface with robust functionality, offering a promising platform for trivia enthusiasts.

Chapter 9

Acknowledgments

I would like to express my gratitude to:

- My mentor Lekha Savale Ma'am for her guidance.
- My fellow mates Sarbajit, and Zeeshan for their inputs and support in reviewing and providing feedback for the project.
- The open-source community for providing resources and libraries that facilitated this project.

Chapter 10

References

For further reading and reference, please consult the following resources:

- React Docs
- CSS Tutorials and References
- JavaScript Docs
- Open Trivia API Docs
- NPM Packages