**Report for Logic App Development: Coursework.**

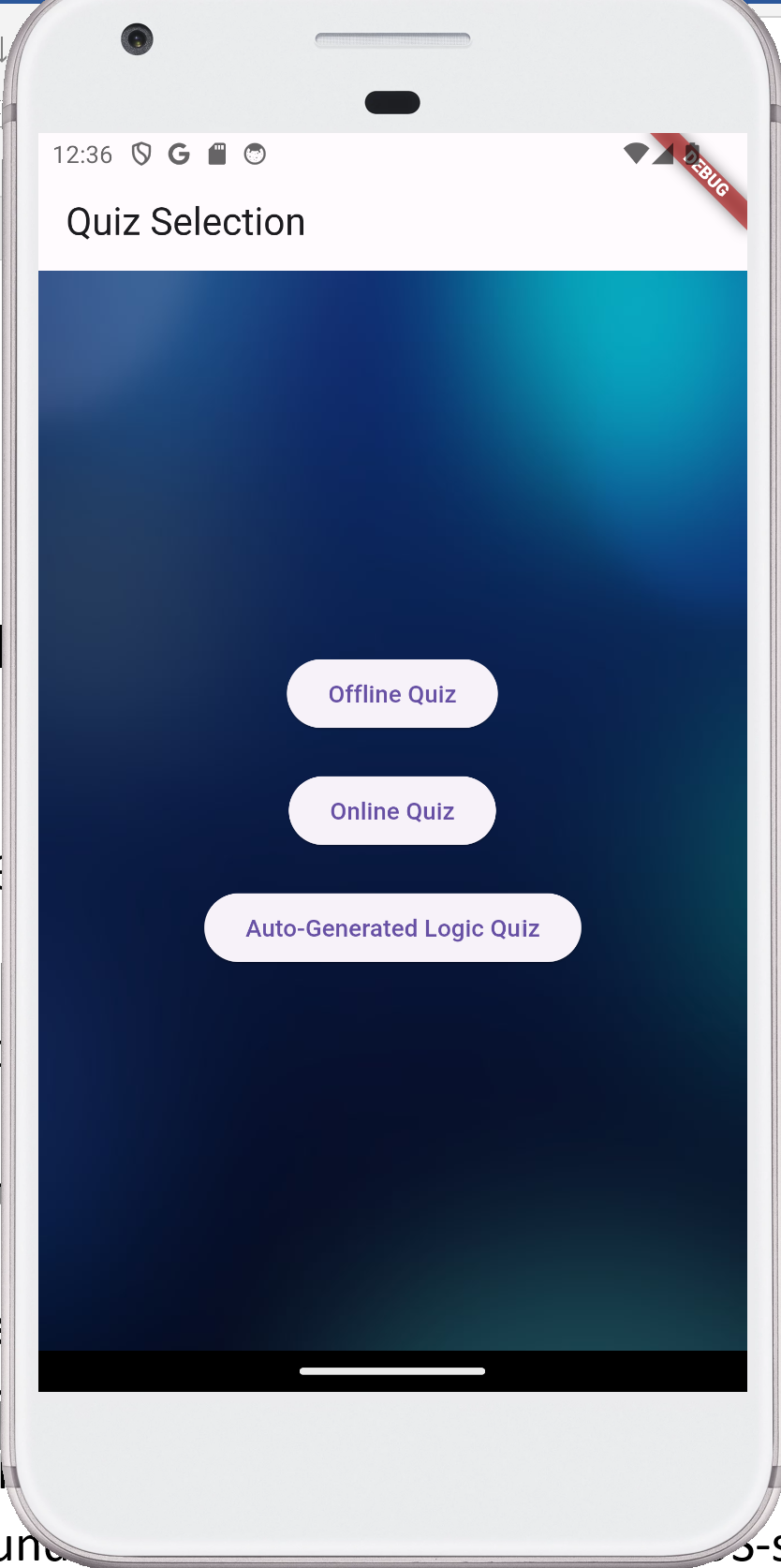
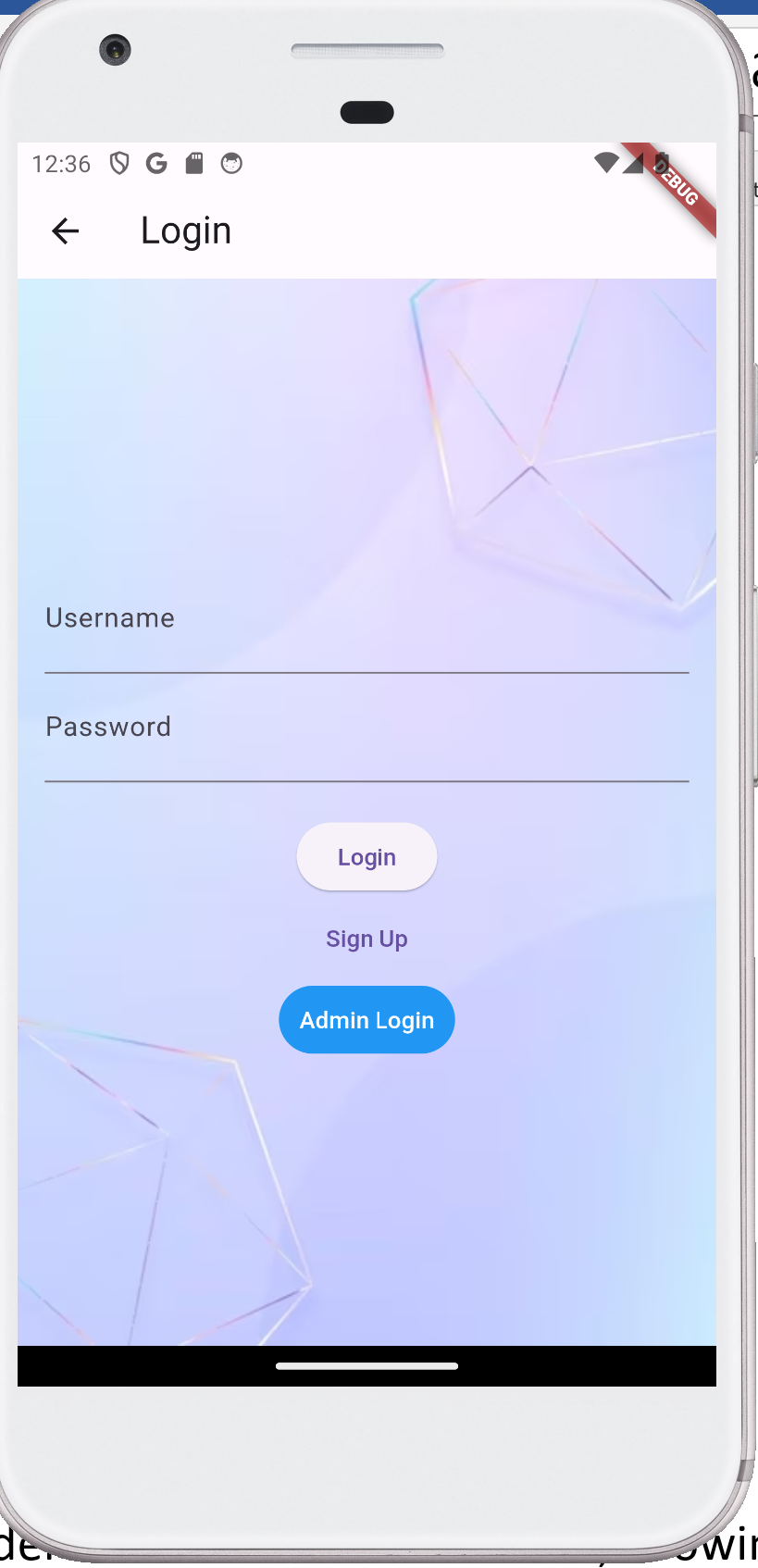
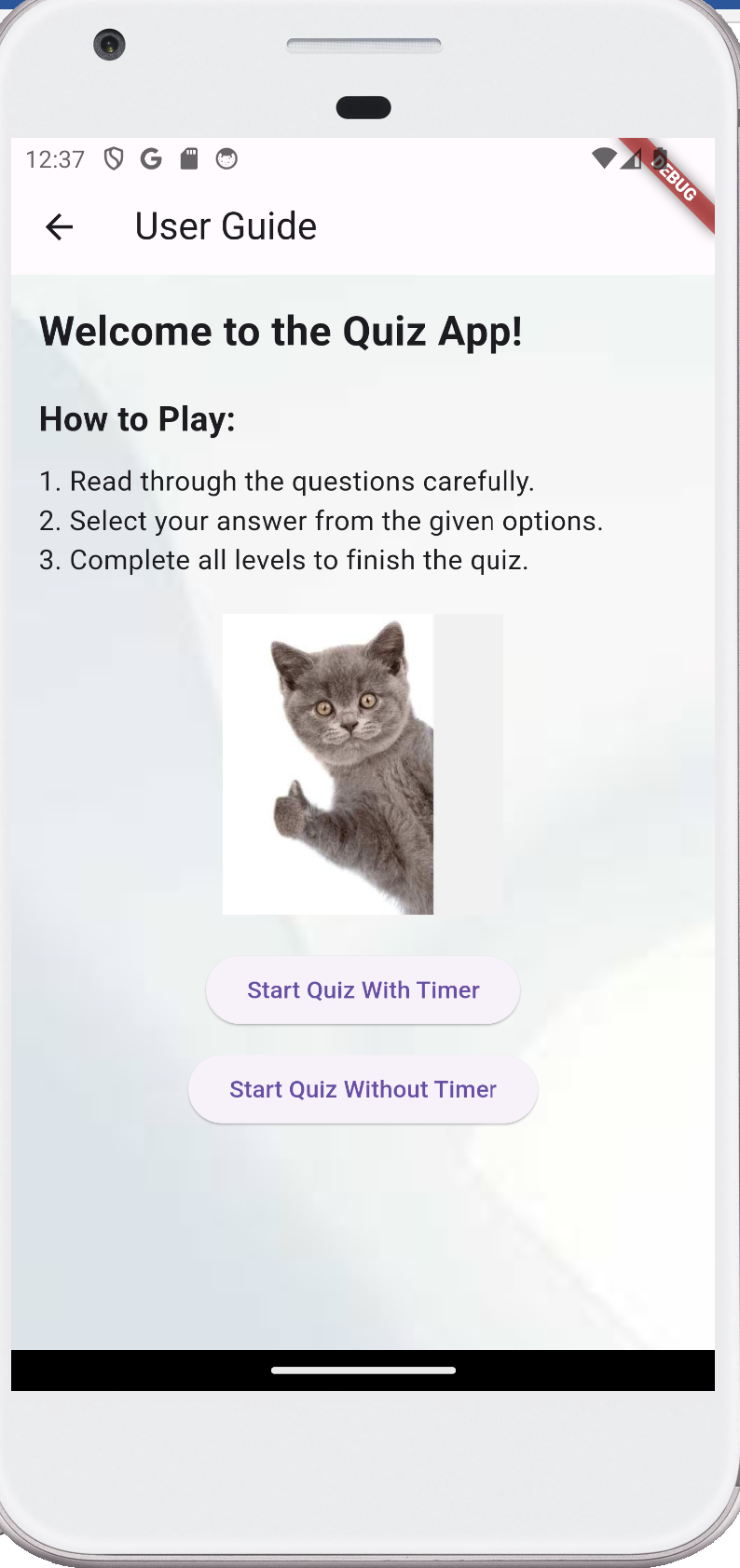
**Name:** Deb Kumar Roy

**Matriculation Number:** 3643231

1. **Introduction**:

The Flutter-based application provides users with a variety of online and offline quizzes themed around the concepts of Formal and Propositional logic. The application can be used on both the Web and on Android.





* 1. **Licenses applied for the development**
* Flutter SDK: Licensed under the BSD 3-Clause License, allowing flexible use in projects.
* Flutter Riverpod: Uses the MIT License, offering unrestricted freedom with proper credit.
* Cupertino Icons: Likely under the MIT License, provides iOS-style icons within Flutter's ecosystem.
* HTTP Package: Facilitates HTTP requests, licensed under the BSD 3-Clause License.
* Flutter Lints: Assists with coding standards, typically under the BSD License.

1. **Implemented Features**

The application is divided into three parts which are Offline Quiz, Online Quiz, and Auto-Generated Logic Quiz.

The first page the user will get into after launching the application is the “QuizSelectionScreen” where the user can select which kind of quiz they want to start as mentioned above.

* 1. **Features for Offline Quiz**
* If the user selects “Offline Quiz” then the user will reach the “welcome\_screen”. The welcome screen has a login mechanism.
* The login ID and password for the welcome screen are hard coded and are **“asdfgh”** and password **“asdfgh”**. At the bottom of the welcome screen, this user ID and password are mentioned.
* After logging in the user will reach the “UserGuideScreen”. This page contains information on how to play and it also has two buttons “Start Quiz With Timer” and “Start Quiz Without Timer”
* The “**Start Quiz With Timer**” is clicked by the user, the user will enter the quiz screen and will start with a quiz that has three levels and each level has 10 questions to be answered.
* After Completion of each level the users are provided with an award and the user needs to answer 8 questions correctly to progress to the next level. If not then they get a try again message in a dialog widget and they go back to the initial stage to try again.
* For every question at this stage the user needs to answer in time otherwise the question is going to get skipped.
* The questions for this Timed Quiz are static and stored in the code itself and it has its corresponding question model.
* When an option is clicked, if the option is correct, it turns green otherwise it turns red and shows the user the right answer by turning the right answer to green at the same time. This is achieved by incorporating an option widget.
* If the user selects “**Start Quiz Without Timer**” from the “UserGuideScreen” the user reaches the quiz page of questions without timer.
* This particular quiz has 10 hardcoded questions without any timer and at the end, the user is shown how many questions they have correctly answered.
* This Particular quiz does not show the correct answers to the user when a wrong answer is clicked. If the option is correct, it turns green otherwise it turns red.
  1. **Features for Online Quiz**
* When the user selects the “Online Quiz” option from the “QuizSelectionScreen”, the user ends up on the Login page.
* The Login page has a few features, a placeholder for entering the Username and Password, a Sign-Up button, and Admin Login.
* To Participate in this quiz the user has to Sign up before. When the “Sign Up” button is clicked the user reaches the signup page where the user has to include their username and password to sign up.
* This signup data is stored in the MongoDB server and now the user can use this password and username to log in for the quiz.
* The online quiz provides the user with 10 propositional logic questions which have to be answered in a certain timeframe and at the end the user is shown how many questions they correctly answered.
* All 10 questions are fetched from MongoDB and presented to the user on the online quiz page.
* When an option is clicked, if the option is correct, it turns green otherwise it turns red and shows the user the right answer by turning the right answer to green at the same time. This is achieved by incorporating an option widget.
* Now we have the Admin Login in the Login Page if the admin clicks in and will reach the admin dashboard. The admin dashboard contains the user names and the number of correct answers.
* The Admin login is hardcoded for now to username: “Admin” and password: “12345678”.
* The data (username and the number of correct answers) for each user is stored in MongoDB and fetched and displayed for the admin.
  1. **Features for “Auto-Generated Logic Quiz”**
* Now when the user selects “Auto-Generated Logic Quiz” from the “QuizSelectionScreen” the user will reach the “TaskGeneratorQuizSelectionPage” page which has 5 options, TaskGenerator Quiz 1, TaskGenerator Quiz 2, TaskGenerator Quiz 3, TaskGenerator Quiz 4, TaskGenerator Quiz 5.
* Each task generator has different types of questions as below:

|  |  |
| --- | --- |
| **Name of Task Generator** | **Type of Questions** |
| TaskGenerator Quiz 1 | Formal Logic Quiz |
| TaskGenerator Quiz 2 | Syllogism Logic Quiz |
| TaskGenerator Quiz 3 | Deductive Logic Quiz |
| TaskGenerator Quiz 4 | Inductive Logic Quiz |
| TaskGenerator Quiz 5 | Pattern Logic Quiz |

* By clicking on each the user will get different quizzes but the basic features included for each quiz are the same.
* Every question is a true or False question and when the user answers a question and clicks on next the user will be shown if the answer is correct or incorrect by a message at the bottom of the page.
* At the end of the quiz the user just gets to know that the quiz is complete and they can restart the quiz.
* The task generators have a fixed structure of questions defined in them and the task generator creates different combinations to provide the questions to the users.
  1. **Partially functional**

The last 4 task generators are functioning partially the user can still get questions on the screen and answer them and all the other features are included but the number of questions generated is not enough by the generator.

1. **Setting up the project**

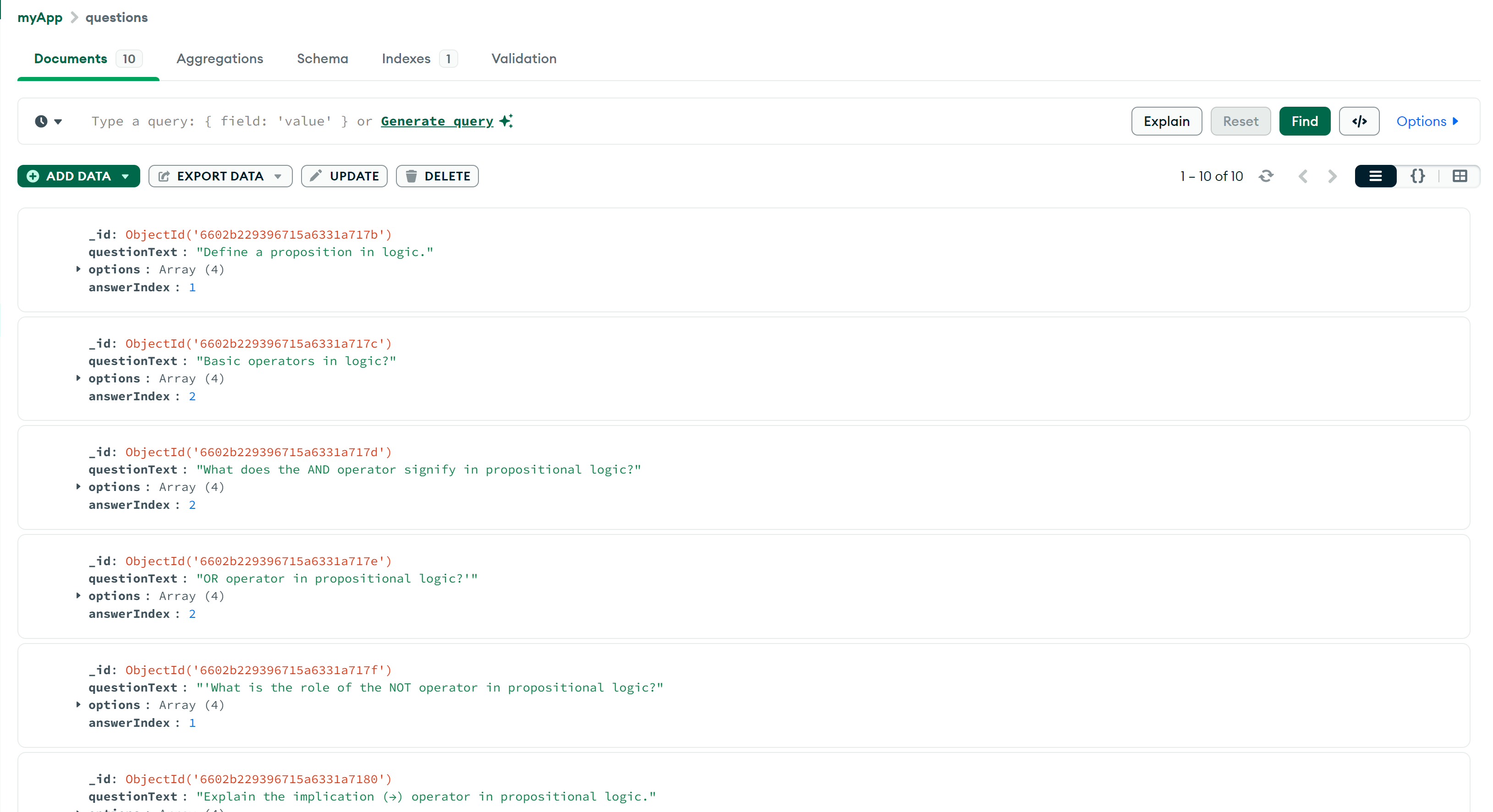
The code for this application is done in VSCode using the Flutter framework. The front end is done in Dart and the server side is done in Go. The database used for online functionality and storing data is MongoDB. “http” package, a Dart package providing a simplified client for HTTP requests, which is used to make various types of HTTP requests, including GET, POST, PUT, and DELETE, with ease. To include the http package in the project, I added it as a dependency in the pubspec.yaml file.

One of the key components of the application’s architecture is the state management system. The package was included in our project by adding the dependency to the pubspec.yaml file

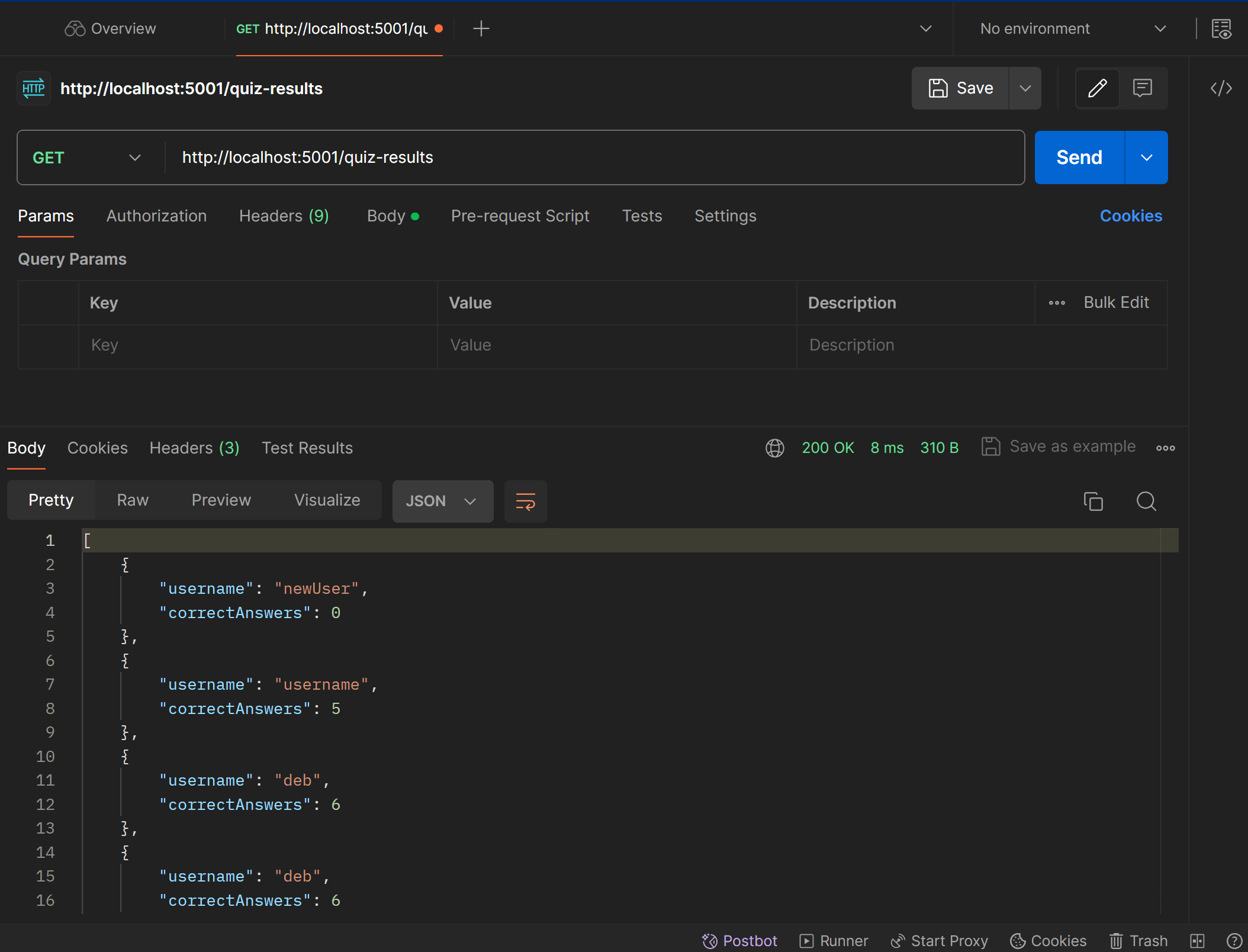
The application uses some background and other images which are present under the asset folder and also mentioned in the pubspec.yaml file.

For running the code on mobile I have generated an apk and for the laptop, I have used a Pixel XL API 34, an android-x64 emulator which I added after installing Android Studio and the required SDK with it. Along with the Android emulator the application can also be run on the web but because the endpoints mentioned in the front end for the online quiz are specifically for Android which is “10.0.2.2”, only the offline quiz part and the task generator quiz can be run.

If the dart code is run alone then only the front end and the offline part of the quiz work. To enable the online functionality MongoDB needs to be switched on and the two server programs need to run which are in Go.

The server needs to be connected and seeded with the questions directly by running main.go and if the server already has questions seeded in them and the admin wants to change or edit them then the question list in the main needs to be edited or changed accordingly and database is needed to be seeded again so that it reflects the new changes. This happens by running “go run main. go seed” after which the database should look like below:  


After the server codes are running it’s a good idea to check in Postman the endpoints also before running the flutter code. For example, I am showing one endpoint(http://localhost:5001/quiz-results) below:



1. **Program Structure and Details**

**4.1 Frontend:**

**4.1.1. Offline Quiz Structure.**

The app uses Riverpod for easy updates and growth, focuses on making quizzes fun and user-friendly, and keeps things organized for simple upkeep and adding new features.

**Architectural Decisions**

**Modular Design:** The app is structured into distinct screens (QuizSelectionScreen, WelcomeScreen, QuizPage), each responsible for a specific part of the quiz flow. This separation enhances maintainability and allows for easy expansion or modification of individual components.

**State Management with Riverpod:** Riverpod is chosen for state management to facilitate a reactive and decoupled architecture. The use of StateNotifierProvider for the quiz state (quizProvider) ensures that the quiz logic is cleanly separated from the UI, allowing the app to respond dynamically to state changes across different levels of the quiz.

**Custom Widgets for Reusability:** The app defines widgets like OptionWidget to encapsulate the presentation and behavior of quiz options. This approach promotes code reuse and simplifies the UI code in the main quiz screen, making it easier to manage and update.

**Timer Integration for Question Time Limits:** The TimerController class manages time constraints on quiz questions, adding a layer of challenge and engagement. It is tightly integrated with the quiz state, demonstrating a strategic decision to enhance user interaction through timed questions.

**Implementation**

**Quiz Selection:** The QuizSelectionScreen provides users with options to navigate between different quiz modes (offline, online, auto-generated), leveraging Flutter's navigation capabilities. This screen acts as a central hub for accessing various parts of the app.

**Welcome and Authentication:** The WelcomeScreen implements a simple login mechanism for the offline quiz, illustrating how user authentication could be managed within the app. Although basic, this feature sets the groundwork for more complex authentication flows.

**Quiz Interaction:** In the QuizPage, users interact with the quiz questions and options. The implementation includes logic to track the current question, select and evaluate answers, and navigate through questions based on user input and timer expiration. This screen is where the core quiz functionality comes to life, supported by the underlying state management and timer logic.

**Level Progression and Feedback:** The app uses LevelProgressionUtil to give you quick feedback when you finish a level. It decides what happens next based on how well you did. This tool helps keep the app easy to manage and use again in different parts.

**4.1.2. Online Quiz Structure.**

**Architectural Decisions**

**Use of Flutter Riverpod:**

Riverpod's Provider and FutureProvider to fetch quiz questions from the backend and manage the quiz state. This approach ensures that the app's UI remains responsive and up-to-date with the latest state.

**Integration of a Custom Timer:**

To enhance the quiz-taking experience by introducing time constraints for each question, adding a layer of challenge and engagement for the user. A TimerController was implemented to manage countdown timers for questions. This controller is responsible for initiating the countdown at the beginning of each question and handling timeout events, such as automatically advancing to the next question or marking the current one as incorrect after the time expires.

**Modular Screen Architecture:**

To keep the codebase organized, maintainable, and scalable. By breaking down the app into smaller, focused screens, easily manage, update, and extend individual parts of the app without affecting others.Screens like QuizSelectionScreen, LoginPage, SignupPage, and QuizScreen were created as separate widgets. This modularization facilitates code reuse and simplifies the navigation flow within the app.

**Quiz API Service Integration:**

To dynamically fetch quiz content from a backend service, allowing for a constantly updating and expanding question bank without the need to update the app. A QuizApiService class was introduced to handle HTTP requests to the backend, fetching quiz questions and submitting quiz results. This service uses Dart's http package to make network requests and process responses.

**Error Handling:**

Implementing error handling within network request functions and using Flutter's SnackBar, AlertDialog, and FutureBuilder widgets to communicate feedback and errors to users.

**4.1.3. Task Generator Structure**

**Architectural Decisions:**

**Modular Design for Quiz Variants:**

To accommodate various types of logic quizzes, such as deductive reasoning, inductive reasoning, and pattern recognition, to structure the app in a way that allows easy addition of new quiz types without significant rework. Separate Dart files (autotask.dart, autotask2.dart, etc.) were created for different quiz themes, encapsulating specific logic and UI for each quiz type. This modular approach enhances maintainability and scalability.

**Asynchronous Question Generation:**

Generating questions based on logic expressions can be computationally intensive, especially when ensuring uniqueness and diversity in the questions. Performing this operation synchronously could lead to UI freezes. Utilizing Dart's Future and async/await patterns, question generation is performed asynchronously (generateQuestionsAsync). The compute function offloads the computation to a separate isolate, ensuring the main UI thread remains responsive.

**State Management with Flutter Foundation:**

Effective state management is important for tracking user progress through the quizzes, including current questions, selected answers, and quiz completion status. By using Flutter's StatefulWidget and managing the state within each quiz page (AutoTaskPage, AutoTaskPage2, etc.), ensuring a reactive UI that updates according to user interactions.

1. **Testing**

Testing is done in different parts as below:

**Frontend Testing:**

* Input Field Validation: For each component, validated the input fields to ensure accurate capture and handling of user data.
* Button Functionality: Confirmed that button interactions trigger the correct application logic.
* Authentication and Form Submission: Each component's unique submission logic is verified to either navigate to the next screen or present error feedback correctly.
* Riverpod Providers Testing: Verify that the Riverpod providers correctly fetch and supply the quiz questions, utilizing mock responses to simulate API calls.
* Timer Functionality: Test the TimerController's ability to count down, reset, and stop under various conditions, ensuring accurate timing for each quiz question.
* User Interaction: Simulate user selection of quiz options and validate the UI response, including selection feedback and transition to the next question upon selection.
* State Management: Test the component's state to accurately track the current question index, selected option, and the count of correct answers.
* Visual Feedback: Test the widget visually responds as expected to different states: selected, correct, incorrect, and unselected.
* Completion Logic: Validate that the completion dialog appears as expected after the last question and that results are correctly submitted, including handling of success and error states.
* Edge Cases validation: Validate the working of start and end questions for all types of quizzes.
* The frontend inputs are tested by adding print statements and then checked in the console while running the app on the web.

**Server-side Testing:**

* Database Seeding: Ensure the seedDB function correctly clears the existing collection and inserts the predefined set of questions without errors.
* User Registration and Login: Validate the encryption of passwords, successful user creation, and login functionalities. Verify that passwords are correctly hashed and that login rejects invalid credentials.
* Quiz Results Submission: Test the submission process for quiz results, ensuring data is accurately recorded in the database.
* Quiz Results Retrieval: Ensure the correct retrieval of quiz results, focusing on data accuracy and authorization checks if applicable.
* HTTP Endpoint: Validate the questions endpoint's behaviour under various scenarios, including successful data retrieval and handling unsupported HTTP methods. The endpoints are tested using Postman.
* Data Model: Test the serialization and deserialization of the Question struct to and from BSON, ensuring the accuracy and integrity of data exchanged with MongoDB.
* Error Handling: Assess the application's response to database connection issues or unexpected errors during database operations.

**Task Generator Testing:**

* Dynamic Question Generation: Utilizes a combination of boolean variables and logical operators to construct diverse logic expressions, ensuring a wide range of question types.
* Variable Complexity: Incorporates four boolean variables (p, q, r, s) and selects one of six structures randomly to create questions. This variety challenges users to think critically about the application of logical operators.
* Uniqueness Check: Before adding a new question to the set, it verifies uniqueness to avoid repetition, enhancing the learning experience with a broad set of questions.
* Flexibility in Question Volume: Allows for the specification of the number of questions to generate, accommodating quizzes of varying lengths and intensities.

1. **Packages**

**Riverpod Package:**

**Offline Quiz:** Riverpod is employed in the app to manage quiz states, handle level progression, and facilitate reactive UI updates based on the user's progress and interactions.

State Management for Quiz Progress: The quizProvider is a StateNotifierProvider that maintains the state of the quiz, including the current level, completed levels, and awards earned. This centralized state management facilitates communication between different parts of the app, ensuring that the UI reflects the current state accurately and reacts to changes dynamically.

Level Progression Handling: The LevelProgressionUtil statically checks and handles level completion based on the user's performance. It uses the WidgetRef ref argument to interact with the quiz's state, reading and updating it through the quizProvider. This approach decouples the level progression logic from UI widgets, promoting reusability and testability.

Conditional Navigation and Dialogs: Within the QuizPage and QuizPageWithoutTimer, Riverpod's ConsumerStatefulWidget and WidgetRef are employed to read and watch the quiz state. Based on the user's actions and the timer, it conditionally navigates between quiz levels or shows completion dialogs. This pattern enables a reactive UI that responds to state changes, such as advancing levels or completing the quiz.

**Online Quiz: Riverpod is used for managing its state, including a special service to get quiz questions and a timer to keep track of how long you have for each one. It smoothly handles logging in and out, keeping the app's look and feel in sync with whether you're signed in or not.**  
  
QuizApiService: Handles fetching questions from a backend service. It makes an HTTP request to a specified baseUrl and parses the JSON response into a list of Question objects. This service is provided to the application via the quizApiProvider.

questionsProvider: A FutureProvider that uses QuizApiService to asynchronously load quiz questions. It's consumed within the QuizScreen to display questions and manage quiz flow.

QuizScreen: The main UI component where users interact with the quiz. It leverages TimerController for timing each question and listens to questionsProvider to load and display questions. User selections, question navigation, and quiz completion logic are managed here.

TimerController: Manages the countdown for each question. It invokes a callback when the time is up, prompting the quiz to proceed to the next question or end the quiz.

AuthService: This class likely contains the methods for handling authentication operations such as logging in and logging out with a backend service. While its implementation isn't shown here, it's assumed to have a login method that returns a User object upon successful authentication and a logout method that clears any authentication tokens or user data.

AuthState: Extends StateNotifier with a nullable User state, meaning the state can either hold a User object (when the user is logged in) or be null (when logged out). It uses AuthService to perform the actual authentication logic. The login and logout methods manage the state based on the authentication status.

authStateProvider: A StateNotifierProvider that provides an instance of AuthState. This provider is used to access and modify the authentication state throughout the app. It's the bridge between the UI and the state logic, allowing widgets to react dynamically to changes in the user's authentication status.

**HTTP package:** **The HTTP package in Flutter is used to perform network requests to RESTful APIs, allowing to fetch, send, and manipulate data over the internet.**

**Fetching Quiz Questions**

To retrieve a list of quiz questions from a backend service. The QuizApiService makes a GET request to a specified endpoint ($baseUrl/questions). Upon a successful response (HTTP status code 200), it decodes the JSON response into a list of Question objects, which are then used to populate the quiz.

**User Authentication (Login and Signup)**

To authenticate users by verifying their credentials (login) or registering new users (signup) through communication with a backend service. For login, AuthService sends a POST request to the login endpoint with the username and password in the request body. If the response is successful, it decodes the user data from the response and returns a User object.

For signup, a similar POST request is made to the register endpoint with the new user's credentials. Success is indicated by a specific HTTP status code.

**Submitting Quiz Results**

To submit the results of a quiz attempt to the backend for record-keeping or further processing. A POST request is made to a submission endpoint with the quiz results encoded in the request body. The success of this operation can be managed based on the HTTP response status code.

**Admin Dashboard - Fetching Quiz Results**

Display quiz results in the admin dashboard, requiring fetching the data from the backend.

A GET request fetches the results, which are then decoded from JSON and displayed in the UI.

1. **Sources**

* <https://github.com/abuanwar072/Quiz-App-Flutter>
* <https://github.com/souravpalitrana/Taskly>
* https://github.com/golang/go/blob/master/src/database/sql/sql\_test.go

These sources are used as a reference for the app.

1. **Conclusion**

Building this quiz app with Flutter has been an exciting journey. By using Flutter, I created an app where users can practice and learn about logic through different quizzes, both online and offline. The app is carefully put together, from choosing the right tools to making sure it runs smoothly and teaches effectively.

I have used <https://github.com/abuanwar072/Quiz-App-Flutter> as a reference and built the app step-by-step which is also uploaded in the GitLab but this particular app uses getx which should not be used so after getting an idea of how the server handling and the online part’s working I started to build the app from scratch using Riverpod and http.

I have used JavaScript before to handle the server creation and communication so I implemented a server in JavaScript before to seed the DB and test my front end and then started replacing it with Go. I have uploaded this also in the GitLab.

This can be found in “quizz\_app\_master\_offline – 7” in Gitlab.

This was a good opportunity for me to learn, experiment and implement.