<Pollaris>

Architecture and Design Specification

Version 1.0

04/04/2024

Table of Contents

1. Introduction

Pollaris is a live web-based polling application that facilitates classroom engagement and participation between instructors and their students.

1.1 Purpose of Specification

This document serves to describe the current architectures and patterns which compose Pollaris's systems communications and framework. This document is updated to reflect changes as the application evolves developmentally. Pollaris is ideally used within a classroom setting between one instructor and multiple students, with multiple devices communicating with one another through an external database and central server.

1.2 Scope

Pollaris is primarily designed for use within a classroom environment. An instructor can create quizzes on their own device in advance, and students can take these quizzes whenever the instructor shares the quiz's name and ID. Students can join a quiz through their own device and respond to questions at the pace dictated by the instructor. If a class is not in-person, Pollaris can still be hosted virtually if the instructor were to share their screen with their students.

1.3 Document Conventions

Any bolded **text** in this document is used to highlight important or central points in a section or paragraph. Italicized *text* is used to label diagrams which can be referenced in any part of the document.

2. General Overview and Design

2.1 High-Level Overview

While each component of Pollaris implements different design and architectural patterns, the overall design is the **Model-View-Controller** Pattern. Pollaris's system architecture is segmented into the three components of the MVC. The Model is the Firebase database that Pollaris utilize to maintain and manipulate user data. Firebase provides an interface that captures Pollaris's functionality, stores user data and for administrative queries.

Distinctly, the View is Pollaris's user interface that connects the client to the features that Pollaris offers. The user interfaces and features they offer differ between two user classes or roles (Instructors and Students). Both classes can request and submit data which can be processed and stored. Upon request, the user interface displays the responses to requests made by users. The user interface also requests updates from the database to ensure the most recent data is rendered.

The Controller is the live server that hosts Pollaris's web files. The server selects which response is directed to the user via the interface and to the database. It also maps HTTP requests to data updates. It coordinates the communications between the first two blocks and manages changes and requests on both.

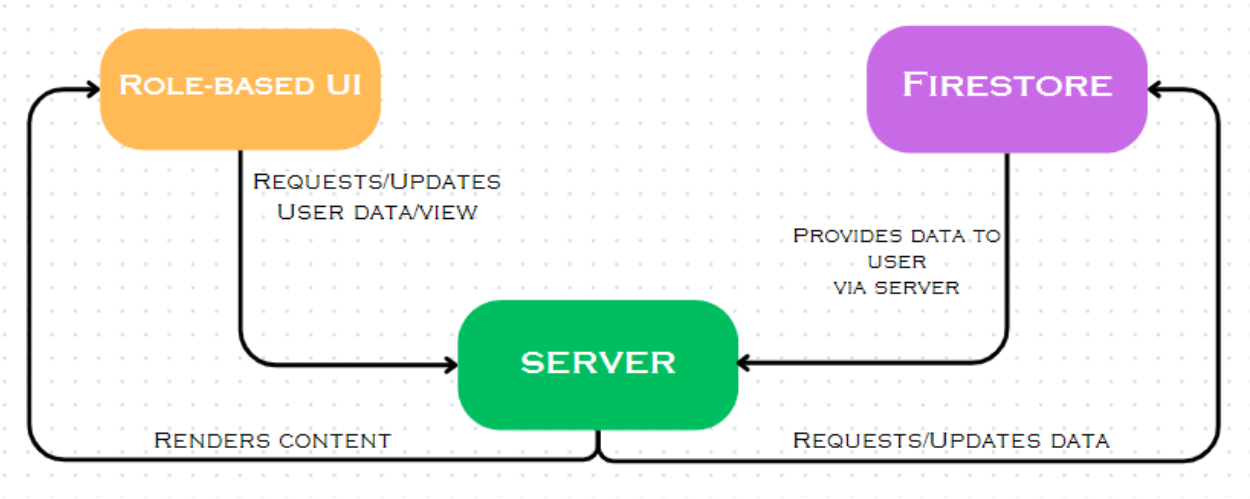

Figure 1: Model-View-Controller Diagram of Pollaris

2.2 Architectural Styles

As a system, Pollaris implements the **Model-View-Controller** system design as described in 2.1 (Fig. 1). However, some individual components reflect other architecture styles within the MVC. Pollaris's NoSQL database, Firestore, implements an object-oriented architecture. Specifically, it stores data as a tree object which is readily reusable and can be traversed optimally.

The client-server architecture is also expressed as different user classes can access Pollaris's server simultaneously. A usual setup between the user classes hosted on Pollaris have a one-to-many relationship, that is, one Instructor to many Students. Pollaris accommodates a number of one-to-many interactions between multiple Instructors in their respective sessions, thereby, enforcing its client-server architecture.

Furthermore, to ensure configuration management and version control, Pollaris adopts a repository architecture pattern. Github is used to maintain different versions of all of Pollaris's configuration items.

2.3 Key Architectural Decisions

- Object-oriented architecture, that is, the tree-like storage structure of Firebase NoSQL database, reduces the time complexity of traversal and retrieval of user data as opposed to using array-like SQL databases (Fig. 4).
- The Observer pattern between the different user classes (Instructor and Student) allows administrators to manage user permissions. For example, the Student UI does not provide permission to switch to the next question during a live session.
- The Repository pattern provides a storage and maintenance system for administrators of Pollaris's system to provide the most stable version to its users while maintaining other versions of the system.

3. Architectural and Design Diagrams

3.1 Block Diagram

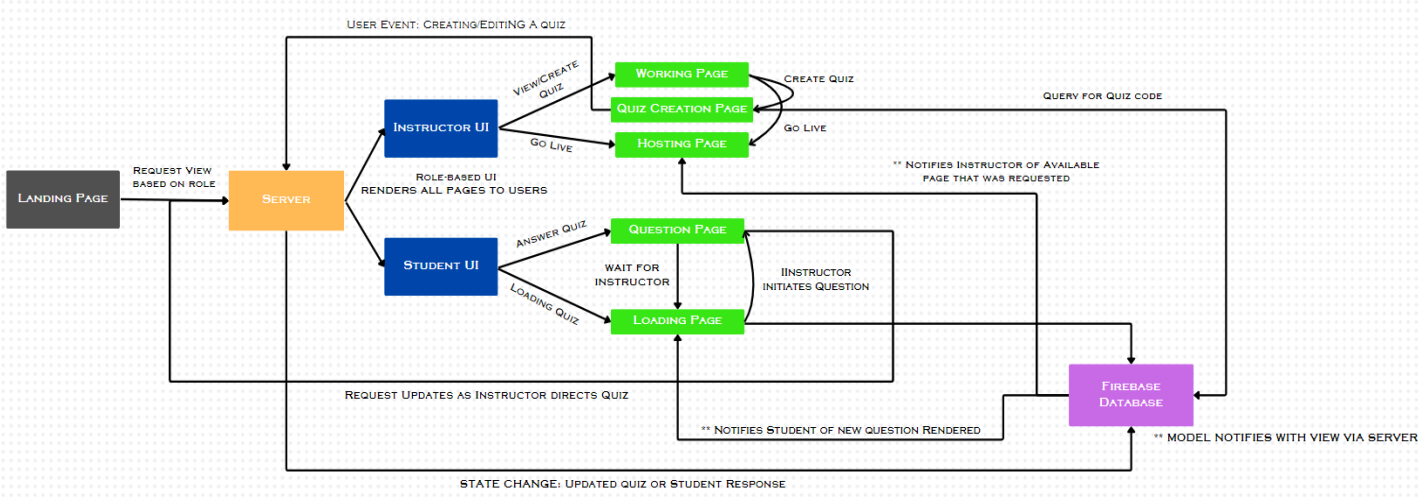

Figure 2: Data Flow and Navigation of Pollaris

Figure 2 outlines the data flow and navigation of Pollaris, enforcing the overall MVC architecture that Pollaris implements. During use, all users begin at the landing page. Based on the selected role, the server directs them to the appropriate interface.

In the Instructor UI, users can sign in and login once registered, view existing quizzes in the working page, create new quizzes in the quiz creation page, and host quizzes. Firebase stores information related to the Instructor view, such as account credentials and quiz information. Each quiz is associated with a unique 5-digit code.

In the Student UI, users enter their first and last name as well as the unique quiz code. When the quiz is being hosted by an Instructor, Students will be sent to a loading page and wait for the Instructor to begin the quiz. A question page displays a question and its answer choices based on

the information extracted from Firebase. After a student picks an answer, they are sent back to the loading page until the Instructor moves on to the next question. This process repeats until all of the questions have been answered.

3.2 Component Diagram

Descriptions of each figure can be found in sections 4 and 5.

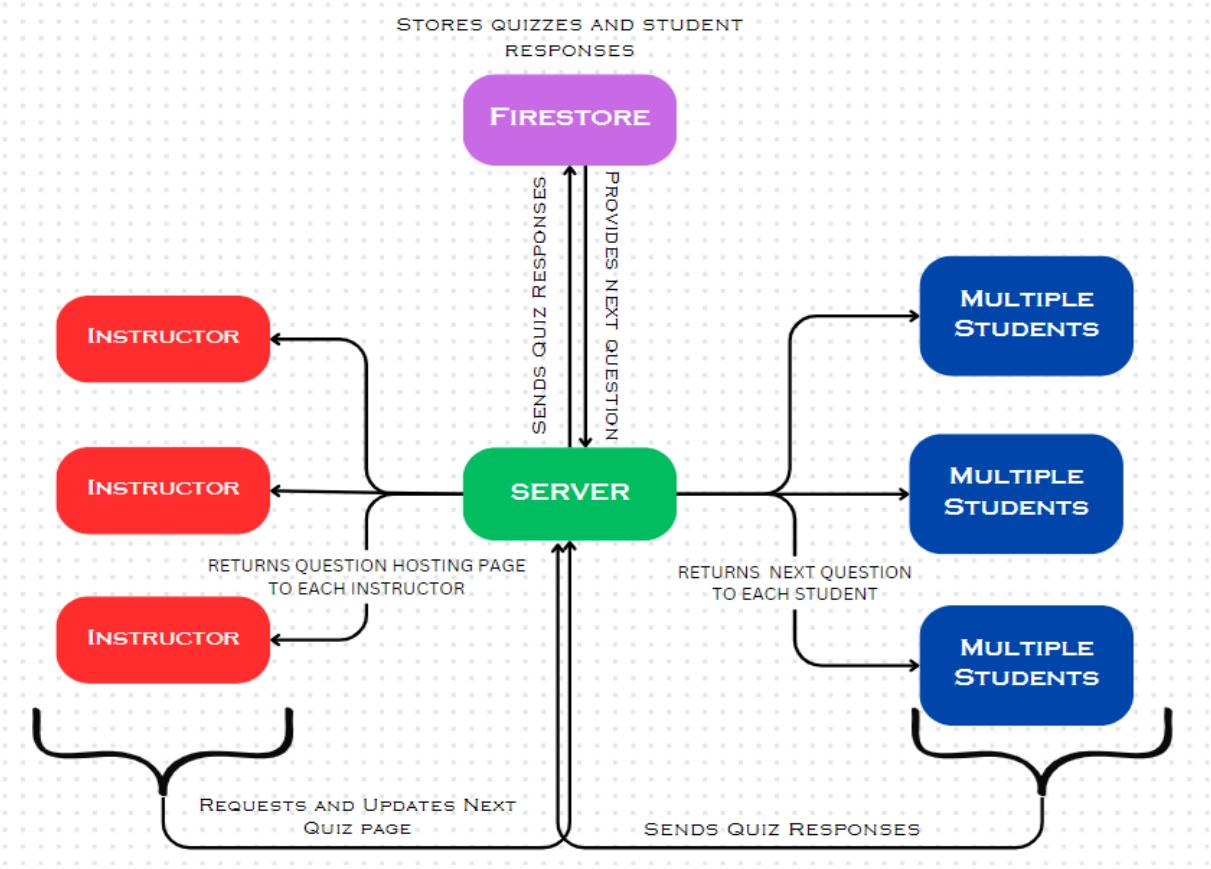

Figure 3: Client-Server Architecture of Pollaris

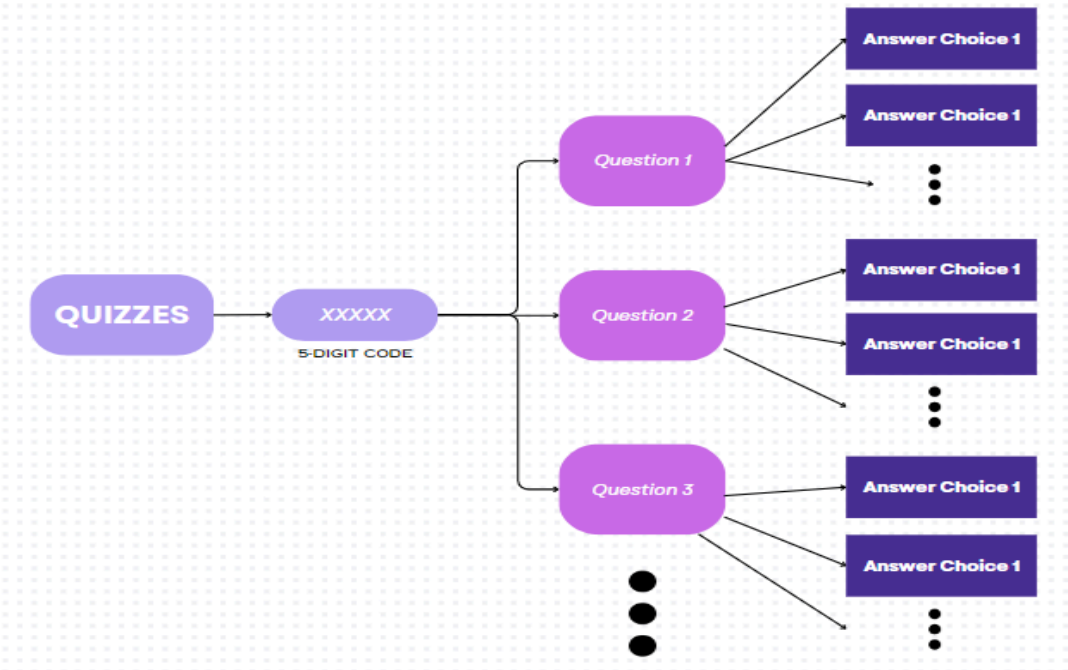

Figure 4: Object-Oriented Architecture of Firestore, Pollaris's Database

3.3 Sequence Diagram

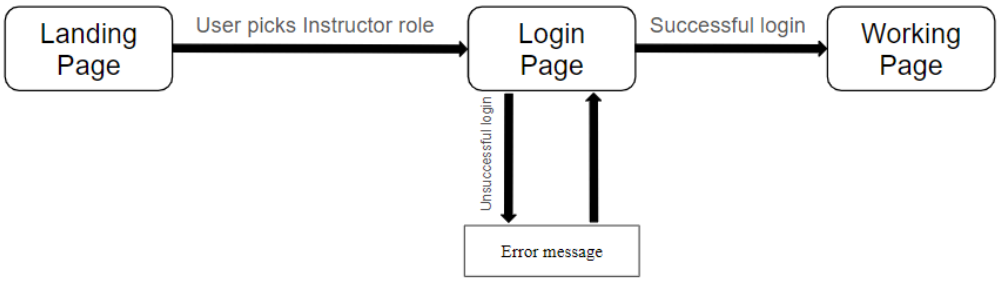

Figure 5: Use-case for Instructor View

Figure 5 showcases a common use-case for a user who wants to operate in the Instructor view. On the Landing Page, the user clicks on the Instructor option. From there, they can log in to their account if they already have one. If the credentials don't match any existing accounts, they will

be given an error message. Further issues can be resolved by going to the sign in page or clicking on ‘Forgot Password?’ If the login is successful, the user is taken to the Working Page. From here, they’re now able to view and create quizzes.

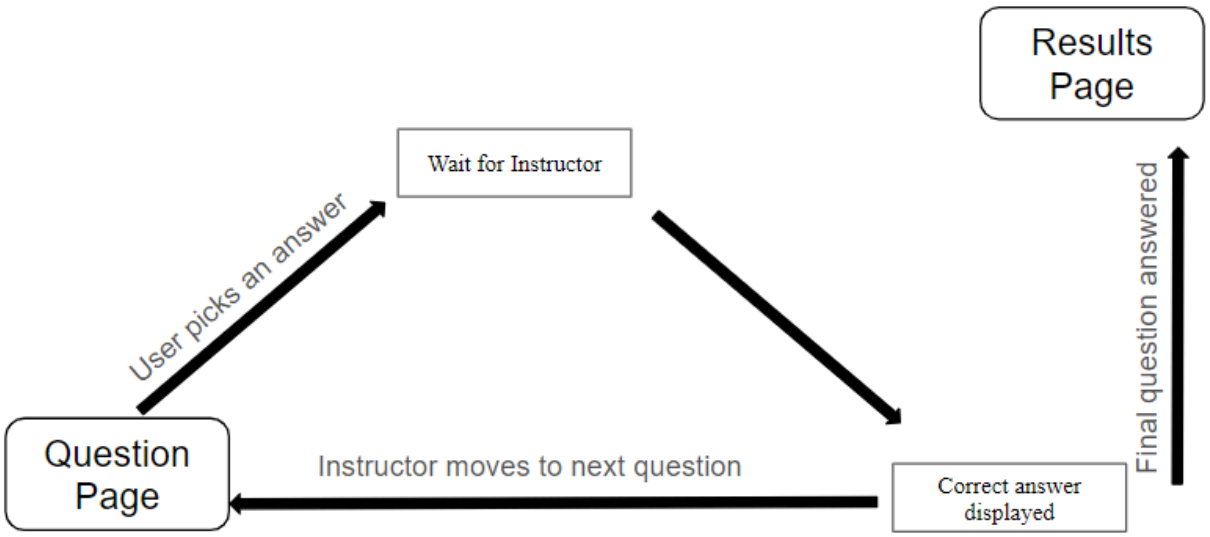

Figure 6: Use-case for Student View

Figure 6 displays the most common use-case for a user in the Student view: the process of doing a poll/quiz. When the quiz starts, the first question is displayed on the user’s screen. After picking an answer, the user must wait for the Instructor to progress. When the Instructor decides to move on, all users under the Student view will be unable to change their answer for that question, and the correct answer is displayed. If there are more questions, the Instructor progresses to the next question, where the question page is then displayed again with a new question. This process repeats until the final question has been answered, in which case, after the final answer is displayed, Students will be taken to a results page that displays their grade.

4. Component Descriptions

4.1 User Interface Component

The UI Component displays Pollaris's web files to users based on user roles and enables interactions between both user classes. It comprises HTML, CSS, and JavaScript code that determines how the user interface looks, feels, and functions. The UI Component shows the poll questions, choices, and voting buttons to users. It also manages user actions like picking an option and giving a vote. Moreso, it could show instant changes in poll outcomes to the users.

4.2 Database Component

The Database Component, Firebase Realtime Database, keeps and controls user data from authentication, published quizzes and polling data during a live session such as poll questions, available choices, and number of votes. It offers an organized storage method for data that enables fast retrieval and handling of polling information via the tree data structure. Since user data is structured this way, CRUD (Create, Read, Update, Delete) operations can be applied to handle poll information and isolate user sessions. Real-time data synchronization and backup via Firebase guarantees that all connected clients get updates instantly.

4.3 Server Component

The Server Component includes a backend structure that handles the movement of information in Pollaris. It is the middle agent between the User Interface Component that users see and the Database Component where data is stored. The Server Component is responsible for managing all user authentication processes. It guarantees that only users who have been given authorization

can use Pollaris by controlling user sign-up, login, logout, and password reset features securely. It confirms that the user's inputs are accurate, avoiding any data inconsistency and stopping harmful or incorrect inputs from affecting the application's functionality or safety.

The Server Component applies access control rules to keep in check user permissions and limits entry to some characteristics or information according to the roles or privileges of a user. It also handles the backend logic. This is where the main functional parts of the polling application are put into action. It deals with creating, changing, and removing polls. Also, it manages how data moves between the User Interface Component and the Database Component to guarantee that everything stays consistent and dependable.

Apart from managing user interactions and data operations, the Server Component also handles other backend tasks like notifying users about poll outcomes and working on analytics to understand how users behave based on their results. One technology that is being used is the Firebase Cloud Functions. This is a service that allows code to run on the backend in response to events that are triggered by features of Firebase and HTTPS requests. This aids our server-side logic in the Pollaris like authentication, checking data validity, or working with database actions. The actual Server Component is placed on a Linux server, which offers stable and protected surroundings for operating backend services.

Servers working on Linux provide adaptability, and expandability along with strong security elements that make them perfect to host web applications and handle backend jobs efficiently. As an extra point, Linux servers can work with different kinds of web server software such as Apache or Nginx. In our case, we are utilizing Apache, which is responsible for accepting HTTP requests from visitors and sending them back the requested information in the form of web

pages. They also support a multitude of programming languages like Node.js and Python, JavaScript which gives us flexibility.

5. Architecture and Design Patterns

The database architecture harnesses the power of Firebase as a cloud-based NoSQL database, complemented by architecture and design patterns such as Model-View-Controller, Repository, Client-Server, Observer, and Iterator. This holistic approach enables us to build scalable, real-time applications with robust data management capabilities, ensuring optimal performance and user satisfaction.

5.1 Model-View-Controller (MVC)

Within the architecture, we adopt a Model-View-Controller (MVC) design pattern to organize our codebase and separate concerns effectively. The Model component encapsulates the logic for interacting with Firebase, handling data retrieval, storage, and manipulation. By centralizing these operations within the Model, we promote code reusability, maintainability, and scalability, facilitating easier management of data-related functionalities.

5.2 Repository

In Pollaris, utilizing a repository-type architecture ensures that the system's design revolves around a clear separation of different modules. Specifically, the application comprises several key components working together to deliver a seamless user experience. Each component is stored in the Github repository so that the correct versions are assembled together when integrating Polaris's system. This architecture promotes scalability, maintainability, and testability, allowing for easy adaptation to evolving requirements and facilitating the integration of different component implementations. Overall, by adopting a repository-type architecture,

Pollaris achieves an essential foundation for efficient configuration management and version control.

5.3 Client-Server

Due to its availability as a web application, the client-server architecture pattern is utilized to organize multiple interactions between user classes. A central server houses Pollaris's files and is responsible for requesting new data from the database based on user requests. The clients (Instructors and Students) request new pages and data from the server and from the database through the server respectively (Fig. 3).

5.4 Observer Pattern

Furthermore, our database implementation incorporates the Observer design pattern to enable efficient data synchronization and real-time updates across multiple clients. This pattern establishes a dependency relationship between Firebase's data sources and the corresponding observers (user classes) within our application. Both Students and Instructors operate Pollaris, however, with different views.

For example, during a live quiz, the Student UI displays only the rendered question, the answer choices for that question and radio buttons offering the option to choose the desired option. The Instructor UI offers the "Next" button to move to the next question and so, has a similar view to the Student UI but without the buttons to choose. The Instructor can also choose to display the generated poll which is not available on the Student UI. As a result, changes made to the database and requests by users are promptly reflected in both user classes' UI, ensuring a seamless and responsive scope experience.

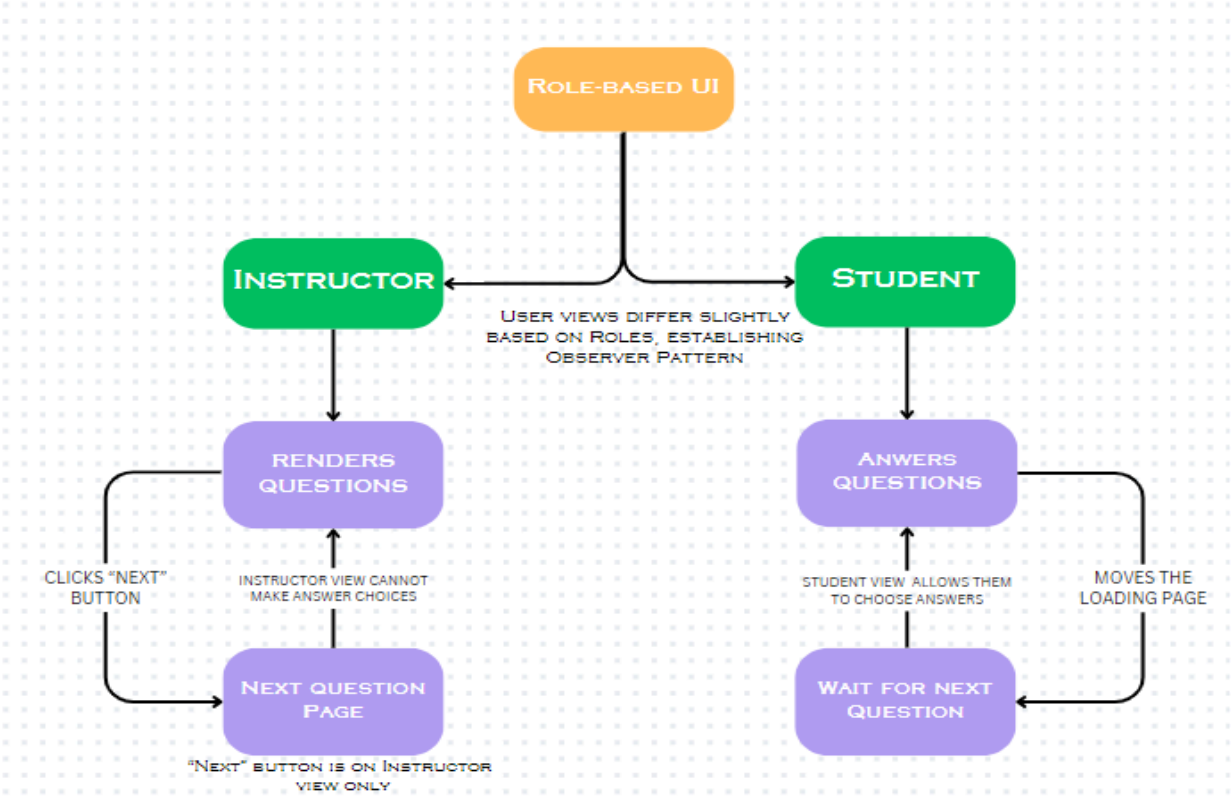

Figure 7: Observer Pattern of Pollaris

5.5 Iterator Pattern

In addition, the database architecture incorporates the Iterator pattern, particularly when dealing with tree-like data structures. By employing iterators, traversal and manipulation of hierarchical data stored within Firebase is facilitated and maintained. This approach enables us to efficiently navigate through complex data structures, retrieve data in a structured manner, and perform operations such as filtering and iteration seamlessly.

Appendix A: Acronyms and Glossary

Term	Acronym	Definition
<HTTP>	*Hypertext Transfer Protocol*	*This web protocol governs user interactions and navigations of a web user via links*
<UI>	*User Interface*	*Describes the view, interactions and experiences of a web page's user*
<HTML>	*Hypertext Markup Language*	*Markup language for generating web elements*
<CSS>	*Cascading Style Sheets*	*Styling language for designing web pages*

Table 1: Acronyms and Glossary