Quiz App

W. Sparks, A. Mccuan

Introduction

As we have recently become more reliant on computer systems to facilitate academic ventures it is clear that new learning systems may help instructors teach their students more effectively. Due to the need for instructors to give quizzes, assignments and allow students to work in groups these systems should be built as a distributed system. Since not all students have access to the same quality of hardware the system should focus on providing an experience that runs equally well on both high end and budget hardware.

This means that the developers should have most of the processing and overall work for the system be done by a server, while the client's device should perform as little as is reasonably possible. This is the idea of the thin client as presented by force point [1], which makes the application available to more users. In addition, many lifelong educators believe that short, informal quizzes help assess how well students retain information from their lectures and propose that these can be used to both increase student exam scores and improve the lectures of the instructor [2]. Our idea was to construct a quiz application using these ideas in an attempt to build a system that allowed instructors to check their students' understanding. By checking their students' understanding they are able to find any gaps in their students' knowledge and adjust their course material accordingly.

This system should conform to multiple different criteria. First most it should be accessible so that all students can reasonably be able to utilize it effectively. Second it needs to be a lightweight application so that even devices with limited computational ability would be able to run it efficiently. Lastly it should utilize a distributed structure that allows multiple clients to access and alter data on a remote server.

Related Work or Literature

There were a few articles we read which allowed us to determine the direction to take when formulating our project idea. One topic that we wanted to research was what ways an instructor could determine which topics their students were struggling with. We found that many authors, such as Monica Fuglei from Resilient Teacher, support the idea that informal assessments help instructors achieve this goal. In her article *How Teachers Use Student Data to Improve Instruction* she addresses this issue directly writing that "Information gleaned from this process allows for quick modification to the next class's plan and identifies learning gaps long before they show up in a summative assessment or become an issue in standardized testing" [2]. By using these informal assessments regularly both in their lectures and online, instructors can continue to improve their courses materials. By improving their course materials, they will be able to teach more effectively and help their students to succeed.

Without checking the students' knowledge, their lack of understanding would have gone unnoticed until the next formal assessment. By that time, the student will have likely fallen far behind and may not be able to effectively recover. By making sure that their students understand the foundational knowledge required to understand a topic before moving on to more complex material they will improve the overall effectiveness of their teaching. These improvements will not simply benefit the current students, they will also benefit all future students taught by the instructor.

Another related topic that we studied was the idea of a database listener and how it may be able to be implemented in a distributed application. One source of information we found on this topic was the manual for the Oracle database titled *Listening for Changes and Modifying Data*. The author of this manual explains that database listeners wait for changes to occur to the database data, and once this happens the programmer can have actions associated with a particular change to execute [3]. For our application this could be used to update a page on the user interface once the associated data was changed on the server. This was our first idea when attempting to find a way to have our application update dynamically based on changes to the database data.

The last topic that we studied was similar both to the idea of database listeners, and also to the publish subscribe model: web sockets (Fig. 1).

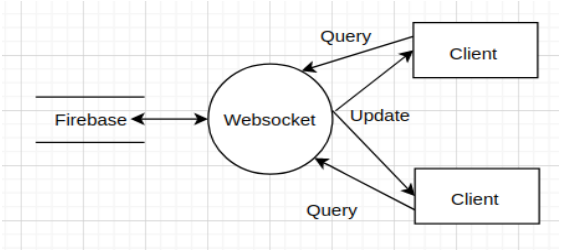

Fig. 1: Web socket Diagram

A document titled *The Web Socket Protocol* written by Ian Fette and Alexander Melnikov of the Internet Engineering Task Force was our main source of information. This document explains how a web socket is a protocol that is built over the existing tcp protocol. Its main purpose is to allow a two-way connection between a client and a server without having to have many http connections at any given time. In addition, Melnikov and Fette explain that this also side steps the issue when using multiple http connections of constantly having to poll the server to check for updates. The main point that made this applicable to our application is that it allows for real time updates to be delivered to clients [4]. This seems similar to the idea of the publish subscribe model which we studied in class. In this case the server would be like the publisher, and the client would be analogous to the subscriber. We decided to pursue this particular topic because it utilized preexisting protocols and seems to be supported by most programming languages. In addition it is supported by many databases and back end servers.

Problem Statement

Many test taking applications exist such as those embedded in the Canvas, Blackboard and Moodle learning systems that are used at CSUB. Each of these works well to give standard exams, but they are not set up to perform informal quizzes, so they are not as effective at checking students' understanding. Although the instructor can gain some insight into what their students have learned, it may be better to informally evaluate students immediately after the associated lecture. To solve this issue, we wanted to create an informal quiz taking application that would work well for both in class, and completely virtual quizzes. This application should be lightweight, efficient, and intuitive for users to navigate.

Technical Approach

Since many students may not have easy access to a computer, but likely have a smartphone, we decided to build a phone application. This serves the purpose of making our application available to more students and thus more useful to the instructor. In formulating our technical approach, we had to decide whether to build our application for iOS or Google's android. In researching which OS would be best to develop this application for, we came across the cross-platform options for mobile phone development such as Flutter, created by Google, and React native. While both have merit and would have allowed us to complete our project, we went with Flutter as it easily integrates with other Google products such as Firebase and its associated databases.

To set up the groundwork for our application we needed some way to store the quizzes data. There are many possible options to implement a database with a phone application. One that we have used in the past is the sqlite database framework. This is a database that is relational and is natively used on all mobile phones that are currently in use [5]. Although this is a good option for a local database, it is a serverless design so we could not use it in our application.

This led us to pursue other options. Most of the options that we found that were able to be integrated with our development environment utilized the nosql paradigm for database design. We decided to utilize Firebase's cloud firestore for the database for our application as it allowed us to easily implement it with our idea. Since our selected database is nosql we had to design our database specifically for this paradigm. First, we identified what exactly needed to be stored on the server, and built the design shown in Fig. 2.

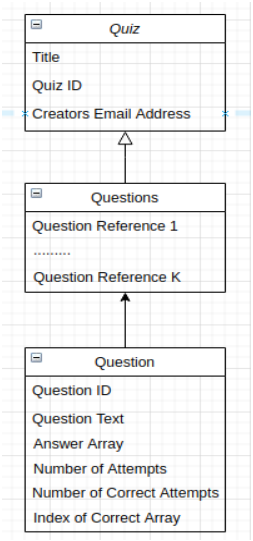

Fig. 2: Firebase database design

At the topmost section of the figure, we have the quiz collection. This quiz collection is filled with references to quiz documents. Each of these quiz documents contains all of the data for a quiz in the database. The specific data that it contains is the quiz's ID, information about its creator, and its title This is analogous to a design in a typical sql database. What breaks from the typical format of a sql design is that each quiz document has a subcollection of question documents. Much like the quiz collection, the questions subcollection holds references to all of the questions that are associated with the quiz. Each question document holds the information needed to produce statistics for the instructor and present the quiz questions and answers to the students. The attributes that it contains are: a question ID, the questions text, an array of answers, the index of the correct answer, the number of attempts to answer the question and the number of times the question was answered correctly. Both of the number of attempts and the number of correct answer fields are used to produce statistics for the instructor.

After designing the database, we built a user interface that was sectioned into two views, one for the instructor and another for the student. The instructor view (Fig. 3) focused on presenting real time statistics about a quiz that is currently taking place, while the student view presents the quiz to the students (Fig. 4).

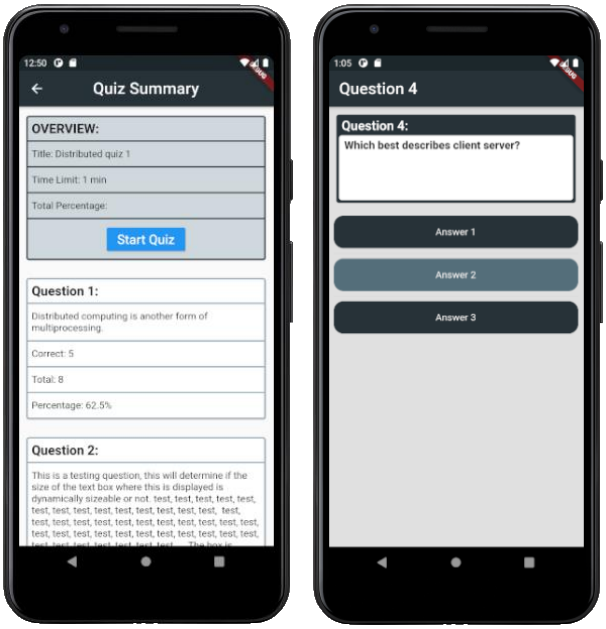

Fig. 3 (Left Image): Instructor Quiz Summary
Fig. 4 (Right Image): Student Quiz View

Both views utilize the previously defined web sockets, which are available using the correct methods for the Firebase API. The method that we utilized for our application was the snapshot() method in Firebase to retrieve data from the database. This method which is defined in the Firebase documentation functions as being similar to a database listener, so once the database is updated the new data is automatically sent to the users device [6]. By performing the query we are establishing a connection between the device and the firebase server. This is serviceable, as it allows the instructor view to receive information from the server about the students performance on a quiz in real time. In addition, the student devices will also get an update if any of the information that was queried from their devices is altered. For example, if a quiz's question has a typo and the question is updated on Firebase the updated data will be sent to the device.

This is great as now we can have data sent directly to the connected devices, but we need to actually process the data and present it to the user. The way that we accomplished this was by defining the query as a stream object and using that as a parameter to the build function for each of the pertinent pages. Specifically, we utilized the

StreamBuilder class as defined in the fluter documentation for Firebase, which allows us to build a page using dynamic data from a database [7]. By using this class along with the snapshot method, we can define the page in such a way that it will be re-rendered when the client's device receives an update from Firebase via the websocket. The main functionality of this Streambuilder class in our implementation of our project idea is that it allows the page to be re-rendered anytime the stream data changes. This was useful for most pages of our user interface design that utilized any dynamic data, whether supplied by the user or the database.

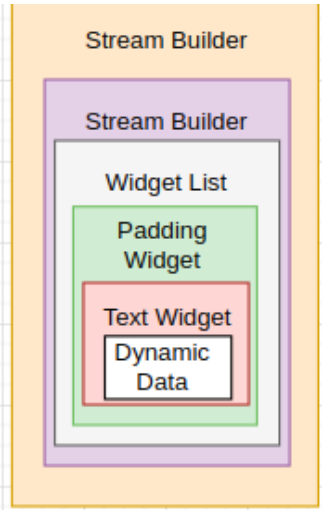

Fig. 5: Illustration of the basic structure of a user interface page in our application.

Here the child of the padding widget is another widget named Text which is given the ${data[index]['q_text']} argument. This argument is the data that was retrieved from the database. Since it is dynamic data, and is inside of a StreamBuilder widget, the user's device will be sent an update via the websocket when the document this information was from is changed. Since the document that the dynamic data was retrieved from was defined as a stream variable in the StreamBuilder widgets definition, the page will re-render when it receives the update from the web socket. One characteristic to note is that in Fig. 5 we have a nested stream builder. This is needed because we need to utilize information both from the parent collection (quiz) and its subcollection (questions), and we want the elements of the page that utilize this data to be re-rendered dynamically when the database is altered. Another way we could have built the application is by using the single interior Stream Builder. This is possible since the quiz data that is handled by the exterior StreamBuilder class is less likely to change when compared to the question data, which changes every time a question is answered. It is important to note that Fig. 5 is an extreme simplification. Inside of the interior StreamBuilder there are dozens of widgets that are utilized by the application but are not relevant to the CMPS 3640 course material.

Related to this is how we dealt with the issue of synchronizing students' devices that are connected via a websocket to the database server. For one, to take a quiz the instructor must login and select the given quiz. If students try to take this quiz without the instructor being logged in and the quiz not being selected, they are sent to a waiting room which gives basic information about the quiz (Fig. 4).

Once logged in, the instructor can then start the quiz by pressing the "start quiz" button at the top of the page (Fig 3). When this button is pressed it updates a field on the database that indicates that the quiz is now live. When this change takes place, an update is sent to all connected student devices from the database to the users devices using the web socket. When they receive this update, the application will navigate all connected users from the waiting room to the first page of the quiz (sample question page in Fig. 4).

If new users try and take the quiz while it is valid, they will immediately be connected to the server, and can start the quiz. Ending the quiz is similar. There is a time limit associated with each quiz, and once the instructor presses the start quiz button it starts a timer. Once the timer elapses the flag that indicates if the quiz is valid is set to zero on the database. This causes an update to be sent to the users though the web socket, which once it is received it automatically navigates the student from whatever quiz page, they are on to the waiting room (Fig.6). This serves the purpose of synchronizing multiple users. Although it is not going to perfectly synchronize the students, it works well for the sort of informal quizzes our application is built for.

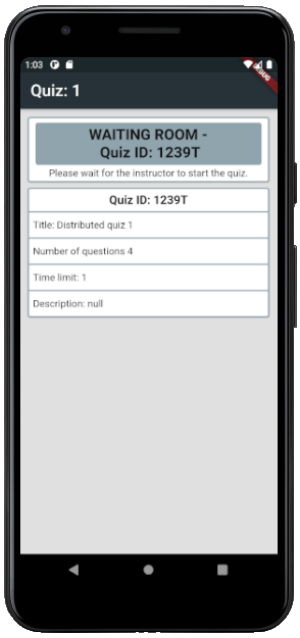

Fig. 6: Student waiting room

Another important interaction is how the statistics for the instructor view are generated. When each user answers a question the backend of the application increments one or more fields associated with the answered question on the database. For instance, if a student answered question one correctly, the backend of the application would increment both the “number of correct attempts” and the “number of attempts” attributes in the associated question document. This “correct index in array” field in the question document on the database and checks to see if the selected answer has the same index. If it does, we call the FieldValue.increment function on the “numCorrect” attribute. This has the same function as a mutex as it only allows a single user to alter the numCorrect field of this particular quiz document at any given time. In addition, whether or not this answer is correct it calls the increment function on the numAttempts field as well to indicate that the user has answered the question. Each time either of these fields are incremented on the database an update is sent to the instructor, which causes the quiz summary page for the instructor to be updated (see Fig. 3).

The obvious issue that needed to be tackled with this implementation is keeping multiple users from updating any given attribute at the same time. This issue was solved by utilizing an increment function on the server which works similar to a MuTex. Instead of having each user wait until the attribute is ready to be changed each individually calls the increment function which resides on the server and the application moves to the next page. This allows the user’s device to continue working rather than having to wait, or continuously check to see if the field is available for alteration.

The server itself performs each increment sequentially in the order in which the method was called by the students. It works similar to a standard queue in which each function call is sequentially placed in this queue. It is then emptied in the standard first in first out order which ensures that no two clients will attempt to alter the same piece of data at a given time. When the fields associated with the quiz’s statistics are incremented an update is sent from the server to the instructor's device, and once it is received the instructor view page (Fig. 3) is updated with the newly received data.

It starts two timer instances of the Timer class. The topmost timer is used to display a timer indicating the number of minutes left on the quiz. The second timer is used to determine if the time for the quiz has elapsed. If so, it gets a reference to the quiz from the database and sets its valid attribute to zero to end the quiz. Since the students are connected to the websocket they will receive an update and their UI will update when the quiz document is updated. The function at the bottom sets the current state of the instructors view of the application to 1. This causes the instructors U.I. to be updated.

Results and Analysis

We were able to complete our project with reasonable success by the end of the semester. The application is fully functional and since it uses web sockets through Firebase it is scalable and can be utilized by a large number of users at once. In addition, the web sockets allow for the use of real time data in the application. It is easy to navigate, and the simplicity of the user interface makes it intuitive for people to use. In testing the application, we found that the time it took for the server to return data to the user was negligible when the user had a stable internet connection.

As with any implementation of an idea there are a few ways that it can be improved. In an analysis of this application, we found that a few features were

lacking. One is that the sharing of quizzes between instructors is not permitted. As stated in the previous section, for a quiz to be marked as valid the creator of the quiz must login and set the start time of the quiz. This makes it impossible for instructors to share their quizzes with other educators. In addition, if a user loses internet connection at the exact time when they are navigating to the next page, the page will be empty as the device will not receive the information from the database. This can be fixed by storing the quiz data in the cache of the user's device, so even if they lose connectivity they will still be able to complete the assessment.

A result of our application using websockets to synchronize the users, the synchronization is not exact. The variance we saw when testing our application was that in some instances this can be greater than five seconds. This is likely due to the use of tcp by the websocket protocol.

Conclusion

During this assignment we were able to create a working mobile application, apply some of the knowledge that we gained from CMPS 3640 and see the ideas presented in class used in practice. In doing so we learned more about the inner workings of distributed systems and the difficulties of implementing one such as how to synchronize multiple clients.

We built our application by utilizing the mobile application framework, Flutter, along with a database server accessed through Google's Firebase. Flutter allowed us to create the user interface using the Dart programming language, while the Firebase system allowed us to use websockets to send updates to our users in real time. These web sockets also allowed us to solve the issue of synchronizing multiple clients by allowing us to utilize a timer which would update a flag on the database and send the flag out to all connected students. The most important of which was the StreamBuilder widget.

This was utilized with queries which utilized the snapshot function which allowed the application to open a websocket connection with the server. When the data was altered on the database by any user, the pages which utilized that data were sent an update via the websocket. To ensure that only one user could alter these sections we utilized an increment function which operated similar to a mutex. This increment function ran server side and allowed us to maintain the integrity of our data. These methods allowed us to create a complete and useful application for our CMPS 3640 project.

In addition, by utilizing both Firebase and Flutter we gained valuable experience utilizing frameworks which are frequently used in industry. Having experience with multiple frameworks and languages is particularly important when pursuing a career in computer science. How the websocket was used in this project was also important. It is being used in more and more projects in many different areas of computer science beyond simple web applications.Since many entry level positions (especially in bakersfield) are web based this knowledge will help us stand out from other applicants when we are job hunting. As students who are both graduating this academic year gaining experience with frameworks which are used in industry will be extremely helpful to landing an entry level position.

References

[1] Cyber Edu, *What is a Thin Client?*, Force Point, Accessed on : Oct. 26, 2020. [Online]. Available: https://www.forcepoint.com/cyber-edu/thin-client

[2] M. Fuglei, *How Teachers Use Student Data to Improve Instruction.* Resilient Educator, Accessed on : Oct. 27, 2020. [Online]. Available: https://resilienteducator.com/classroom-resources/how-teachers-use-student-data-to-improve-instruction/

[3] Oracle Help Center, *Listening for Changes and Modifying Data*, Oracle Inc. Accessed on : Oct. 30, 2020. [Online]. Available: https://docs.oracle.com/cd/E15357_01/coh.360/e15831/datachanges.htm#COHTU221

[4] I. Fette, A. Melnikov, *The WebSocket Protocol*, Internet Engineering Task Force, Accessed on : Nov. 1, 2020. [Online]. Available: https://www.hjp.at/doc/rfc/rfc6455.html

[5] SQLite, *What is SQLite?*, Accessed on : Nov. 3, 2020. [Online]. Available: https://www.sqlite.org/index.html

[6] Firebase Documentation, *Get realtime updates with Cloud Firestore*, Firebase, Accessed on : Nov. 12, 2020. [Online]. Available: https://firebase.google.com/docs/firestore/query-data/listen

[7] Flutter Documentation, *StreamBuilder<T> class*, Flutter, Accessed on : Nov. 1, 2020. [Online]. Available: https://api.flutter.dev/flutter/widgets/StreamBuilder-class.html

Code

Note: Our presentation includes a demo of the code near the end. The link to the presentation is here: https://youtu.be/XNB9G8-cvL0

The code for the project is attached as a complete zip file. Our work is mostly in the lib folder, the rest is set up by the Fluter framework. Since this is a Flutter project the size is relatively large. To run the project, you can install the built SDK that can be found on my public html on odin. This is the best option as it can be difficult to configure Flutter with android studio. All you need to do is enter the URL: http://cs.csub.edu/~amccuan/app-debug.apk into a web browser on an android device (either an emulator or an actual phone) to download the SDK. From that point if it doesn't install automatically click on it from the downloads page. You may or may not have to allow the installation of applications from untrusted sources in the settings of the device to be able to install it. Note: You need at least two devices to properly test the application as you need one to manage the instructor view, and another to manage the student view. Also, there is only a single quiz for testing. The email for the instructor view is test@test.edu, and the quiz id is 40558.

The other option is to unzip the project folder and open the project in android studio. Then install flutter 1.22.2 and download the flutter and dart plugins for android studio. After completing this step, you will need to configure the flutter SDK option for the project, giving it the install location of flutter on your system. You will also need to enable dart support for the project in the android studio settings. After this you need to run the command "flutter pub get" from the command line from the program directory (or the command line interface built into android studio). This will download all the files for the flutter dependencies. Then you can build+run the application using an android device in android studio.