Stephanie Connell

Final Paper

April 24th, 2020

Senior Project

Migrating an Existing Mobile Application to a Cloud Database

Introduction

In Systems Analysis and Software Design during Spring 2019, I worked with the class on the Volition mobile application for Recovery Enhancement Solutions (RES), a nonprofit. This app intends to help people in recovery from drug addiction. I anticipated being involved with all aspects of the application but only worked on front-end development and UI design. To learn more about the full range of software development, I intend to complete a back-end development task to improve the application. Specifically, the local database designed and implemented during the class will be replaced with a cloud database.

To begin the project, it was best to decide the order in which to complete the tasks. An abbreviated version of that project description is included in the following paragraphs.

Since it had been some time since I had interacted with the application, I decided that the first step should be determining the current state of the project. In studying it, I may find that modifications to the app are necessary before implementing a cloud database. Additionally, I will need to determine how the current database will be migrated to a cloud database. The steps for doing so will be described in a migration plan.

In the next month, we will need to choose a cloud database product that will best suit the application's needs. This will involve completing a trade study, which compares all of the products or services that a company offers based on weighted categories and how each item fulfills a category. Once a product is chosen, the actual implementation can begin, following the previously written migration plan. That will be the bulk of the project.

Finally, the cloud database implementation will be completed and tested extensively. At this point in the project, hopefully all that will need to be completed is putting the finishing touches on the database implementation. Of course, migrating to a cloud database is not the only improvement this app needs before it can be released to the public. A structure will be put in place so upgrades to the database can be made easily, and so other people may be able to improve the project in the future.

Luckily, I remembered most of the work accomplished once I began interacting with the application again. Interacting with a nearly fully functional app on one branch of our Git repository, I was able to see what had been accomplished and where the app fell short. We had met all of the expectations of our professor during the class, but the app was still far from being released. One notable flaw was that the app only takes the user to the create profile screen on startup; there is no logic to check if a user already has created a profile. A classmate attempted to implement that functionality, but their work crashes on startup, possibly due to the fact that Google changes its code frequently.

Contrary to the initial plan, Brady Sheehan from RES and I decided, based on the services Google offers, to complete the trade study first. A trade study involves looking at what services are offered, deciding on requirements that must be fulfilled for the app, and assigning

weights to each requirement. Then, the services and how well they fulfill the requirements are compared through numerical assessment. Google offers two cloud database services for mobile applications: Realtime Database and Firestore. Although the documentation provided by Google strongly encourages new applications to use Firestore, it was still beneficial to do a trade study. For the study, I determined most of the categories included although Elizabeth LaRue and Brady from RES had some suggestions. The importance and weights were decided by me with Brady confirming their accuracy. The content of the study can be found in tables in the Appendix. Since it received much higher scores, we chose to use Firestore for the cloud database.

The next step was to determine in what order the steps to create this new database would be completed. This was formalized in a migration plan, some of which is included below. Brady and Elizabeth also requested that I include an estimate of how long it will take to implement the new database.

The first step in implementing this database is choosing one group of Room classes already implemented in the app and converting them to Firestore classes. One group is any set of classes that has at least an Entity and a Data Access Object (DAO). This may take a while, as I don't have previous experience using this technology. In working with one set of classes, I will be able to see what works and what doesn't and test it more easily. After successfully converting that class, I'll work on the other classes that fit the above definition.

Next, I'll work on the classes in the app that don't represent data in the database but interact with it. There are a few Activities that use data provided by the sets of Room

classes. The calls to those classes will need to be replaced with the equivalent Firestore call. Finally, as much extensive testing as the remaining time allows will be completed.

I expect implementing this cloud database will take me the rest of the time in my senior project, meaning about two and a half months.

DemographicData

In order to learn how to translate the existing Room classes into Firestore classes, I chose to work with the `DemographicData` set of classes. The first step was to create a class to represent a `DemographicData` object. Of course, demographic data isn't a real-world object like Firestore objects are meant to represent, but I thought it easiest to stick with the structure previously defined for the Room database. Writing the `DemographicData` class was not hard, since it simply needed to contain the same fields as the `DemographicDataEntity` class, and seemingly, the same getters and setters. The only main difference I found is that Firestore does not require `fetchID`s. However, determining if I had written the class correctly was made difficult by the fact that I did not yet have access to the RES Google Cloud Platform account. Despite this setback, I continued to work on the class.

In the example app that I viewed, Friendly Eats, the Google developers that wrote it placed all user interface-related functionality into Adapter classes. Assuming this to be the correct way to implement this, I began writing an Adapter class for `DemographicData`. I found that any calls in Activities to the database that return data should all happen in an Adapter class. Looking through the existing Activities, I made a list of the usages of `DemographicData`. In addition to creating classes for `DemographicData`, I realized that I needed a general `FirestoreAdapter` to interact with the database. As I worked on the Adapter for

DemographicData, I ran into some significant issues with Dates and Spinners. A Spinner is what Android calls a drop-down list. The main issue was that no easy way existed to get the data from a DatePicker or Spinner from anywhere other than the Activity that interacts with those UI elements.

Finally gaining access to the correct Google Cloud Platform account, I was able to add Firestore to the application properly and interact with the Firebase console more meaningfully. Doing more research, I discovered that Timestamp is what Firestore uses to store dates, not Date. This resolved the Date-related issue, but the Spinner issue still remained.

Realizing that I could not resolve the Spinner issue without having the code within an Activity, I decided to interact with the data in the Activities instead. This would allow me to use more of the code that my classmates had written in the past instead of having to write entirely new classes myself. I removed the DemographicDataAdapter class and began working with ProfileActivity. The logic to get data from the text boxes and other UI elements already existed, I just needed to learn how to put that into the database. Looking at the documentation, I discovered that the best way to write data to a document is to accumulate it throughout the Activity and then write it all to a document at once. I believed the best way to put data into a document in the database was to first put the data in a HashMap and then create a document for it using something similar to the code below:

```
Map<String, Object> data = new Map<>();

  if (parent.getId() == R.id.gender_spinner) {
   if (pos == 0 && spinnerCount > 1) {
          onNothingSelected(parent);
   }
   final String gender = (String) parent.getItemAtPosition(pos);
          data.put("gender", gender);
  }
```

```
db.collection("DemographicData").document().set(data);
```

Using the Firebase console, I was able to see that inserting data in that manner was successful. A screenshot of sample data is included in the appendix.

Reading data became the next task. The Friendly Eats app seemed to write and read data based on a Firebase Authentication user ID. I initially wrote that into `DemographicData` but decided against using it since doing so would require more research and implementation and that is not the current focus of this project. I chose to name and read from documents based on the name the user entered, which likely will need to be changed in the future for increased security. I was yet unable to see if this code could successfully read data from the database since that functionality relates to editing a user's profile. In order to edit a profile, a user must first create a profile, take a questionnaire inquiring about drug use habits, and be able to view the home screen, where there is an option to edit their profile. Since the `Questionnaire` class did not yet use Firestore, I could not use this series of activities to test the code I had just written. The app would crash upon reaching the `Questionnaire` class.

Not remembering that writing tests was the usual next step, I began working on translating the `Questionnaire` class to Firestore in order to test reading from `DemographicData` documents. After beginning that, Brady communicated with me that he was coming to campus. I met with him, and he reminded me of the usual software development process. Testing the code that I had just written should be the next step. Remembering that my classmates had already written tests related to `ProfileActivity`, I reverted the few changes I had eagerly made to the `Questionnaire` class and instead began working on those tests as Brady had instructed.

Two tests already existed that would meaningfully test the code I had just written in `DemographicData` and `ProfileActivity`: `CreateProfileActivityTest` and `EditProfileActivityTest`. There were two other tests that are also related to `ProfileActivity`, but they involve how that activity relates to other classes that I had not worked on yet. Looking over those tests, I decided that they would test the new database sufficiently, and I would not have to write new tests. I chose to begin with `CreateProfileActivityTest`. While working on this project, I had noticed that Android Studio was telling me that the `DemographicData` class was not being used. Working on the test, I realized why this was the case: I wasn't adding data to the database in the correct way.

In order to utilize the custom class I had written to represent this data, I had to create a constructor with all of the fields in the class. The data is still accumulated throughout `ProfileActivity` as it was before, but instead of being stored in a `HashMap`, it is used to create a `DemographicData` object, which is then passed to the database. This constructor is included below.

```
DemographicData demographicData = new DemographicData(patientName,
lastClean, lastUseReport, dateOfBirth, gender, isPersonInRecovery,
useHeroin, useOpiateOrSynth, useAlcohol, useCrackOrCocaine,
useMarijuana, useMethamphetamine, useBenzo, useNonBenzoTranq,
useBarbituratesOrHypno, useInhalants, useOther, disorderAlcohol,
disorderOpioid);

db.collection("DemographicData").document(patientName)
.set(demographicData)
    .addOnSuccessListener(new OnSuccessListener<Void>() {
      @Override
      public void onSuccess(Void aVoid) {
        Log.d(TAG, "DocumentSnapshot successfully written!");
      }
    })
    .addOnFailureListener(new OnFailureListener() {
      @Override
```

```
    public void onFailure(@NonNull Exception e) {
      Log.w(TAG, "Error writing document", e);
    }
  });
```

After making those changes, I resumed rewriting `CreateProfileActivityTest`. Not many changes had to be made since Android Studio's Espresso Test Recorder interacts with the `Activities` automatically. Espresso works by loading an `Activity` that needs to be tested and performing the same touches as a human would. One notable change was in removing the `setTestMode` method from ProfileActivity. The Room database required a temporary test database to be set in order to not make permanent changes to the actual database. With Firestore, the same database can be used for testing and production, so the method was no longer necessary. After making the necessary changes, I tried to run the test. I immediately received Espresso-related errors, mainly saying that it could not find the correct UI element to tap on. The code that my classmates had previously written no longer worked.

I decided that the best course of action would be to rerecord the test. I made a note of what the old test did by looking at the code. The list is included below.

- enter a name into the name text box (Robert Test)
- tap the date of birth date picker and enter September 9th, 1989
- choose Other from the gender spinner
- choose Person in Recovery/Seeking Recovery
- chose Other Opiates and Synthetics
- change the above choice to Other
- type Other drug in the text box
- choose Opioid Use Disorder from the substance use disorder spinner
- change that choice to Alcohol Use Disorder
- tap the clean date date picker and enter January 1st, 2020
- tap the record answers button

Once I rerecorded the test, it functioned correctly. One out of two tests were complete.

Now changing my focus to `EditProfileActivityTest`, I realized that I had not completely finished rewriting that part of `ProfileActivity`. In order to differentiate between creating and editing a profile, the `Activity` has a `boolean` variable named `editMode`. When set to true, the user has indicated elsewhere in the app that they want to edit their profile, and different code in the Activity is executed. The main issue became finding the document in the database since documents are named by the user's name. Since I had already created a `DemographicData` object in the class, I decided to try getting the user's name through that object. I realized that that would only work in certain cases, as the app would eventually need to know the name of the user who is currently using the app. That user would not create a profile each time they opened the app, so that object would not be created every time. This fix was good enough for now, however.

`EditProfileActivityTest` also needed to be rerecorded in the same way `CreateProfileActivityTest` was. After doing so, I discovered another problem: two databases can't exist in the same app at once. I thought about going against Brady's recommendation and rewriting everything first, but I eventually discovered an easier solution. Commenting out all of the Room-related code allowed me to run the tests that I had rewritten. Doing so allowed me to get one of the test methods in `EditProfileActivityTest` to pass, but there were still two other test methods that failed.

I eventually found the reason those two tests failed: the data wasn't being displayed in the UI elements after being read from the database. I attempted to debug that but then realized the possibility that I had removed something important when taking out all of the Room-related code. I looked at the code on the dev branch of Bitbucket, which is commercial software used to maintain git repositories, and discovered that I had removed all of the logic that displayed that

data in the UI elements, thinking that it was only necessary for Room. After returning that code to `ProfileActivity`, two of the test methods in `CreateProfileActivityTest` passed. All that remained was the final method.

The `DatePickers` weren't functioning correctly again. Thinking that it was an Espresso problem, I tried rerecording the test again. That did not fix it. I eventually discovered that, instead of having three separate test methods within `CreateProfileActivityTest`, I had to take the code from within the one that was not passing and put it within the method that retrieves data from the database so that it would not be looking for data that had not yet been retrieved. Some of that code has been included below.

```
  @Test
public void createProfileActivityTest() {
  // Test setting a name in the "name" EditText
  ViewInteraction appCompatEditText = onView(
      allOf(withId(R.id.name),
          childAtPosition(
              allOf(withId(R.id.RelativeLayout01),
                  childAtPosition(
                      withId(R.id.LinearLayout01),
                      0)),
              1)));
   appCompatEditText.perform(scrollTo(), replaceText("Robert Test"),
 closeSoftKeyboard());
  .
  .
  .
  // Test tapping the "Record Answers" button
  ViewInteraction appCompatButton4 = onView(
  allOf(withId(R.id.record_button), withText("Record Answers"),
         childAtPosition(
           allOf(withId(R.id.RelativeLayout01),
             childAtPosition(
                withId(R.id.LinearLayout01),
                0)),
         12)));
  appCompatButton4.perform(scrollTo(), click());
}
```

Finally, all three of the tests passed!

The next step in the software development process is to open a pull request so Brady can review my code and ensure that it works. Software development using Git typically divides each task into a branch. For example, I created a branch for the code involving translating the `DemographicData` class to Firestore. In order to consider that task complete, it must be of sufficient quality to be merged into the main branch, which is called dev. Brady will review my code, ensuring that it works and that it meets some standard of quality. Then, hopefully, he will approve it, and I can merge it into dev so everyone else working on this project can work with the code that I've written.

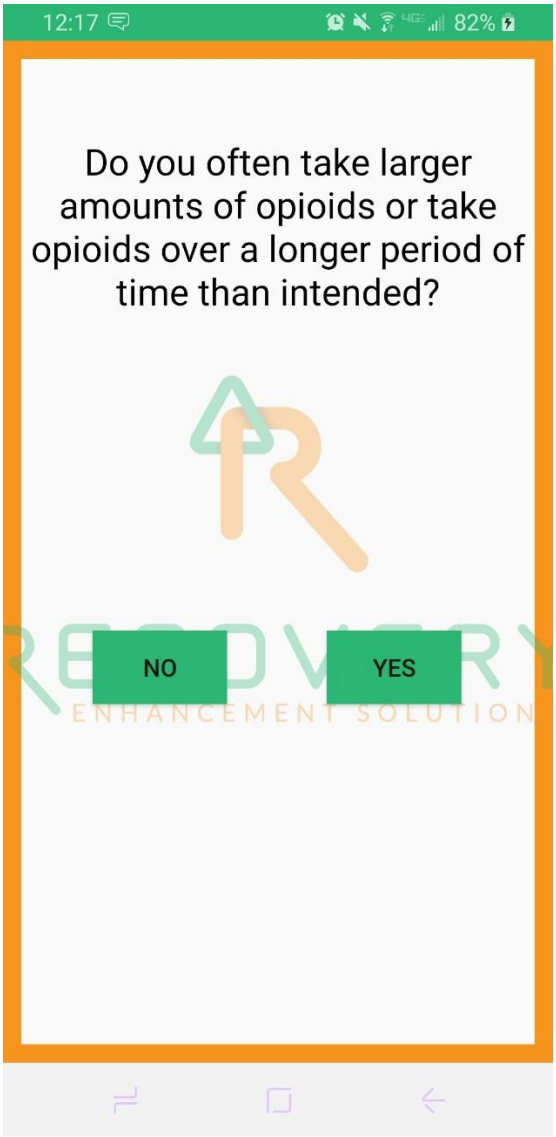
Once Brady began reviewing my pull request, he brought up the important question of why Room-related classes and code still existed in the project. I told him that, in order to get the tests to pass, I had to comment out a lot of the Room-related code as two databases can't exist in one app at the same time. We came to the decision to just leave this pull request open until I had completed the entire project. I could branch from my branch to work on the next part, `Questionnaire`. Since dev, the main branch, would not contain the code I had written for `DemographicData`, I needed to create new branches from the `DemographicData` branch.

Questionnaire

Since I had mistakenly rewritten the `Questionnaire` class earlier in this project, I was able to move on to figuring out how it should be used in the rest of the app immediately after creating a branch to contain that code. A Room database typically uses an object called a `ViewModel` to provide a layer of abstraction between the database and the user interface. A `ViewModel` also contains functionality relating to the visual aspects of an app. For example, if a

user inputs data and then rotates the phone so it's in a horizontal position, that data will still be displayed on the screen. Since Firestore doesn't require `ViewModels`, I was simply able to remove all references to the `DemographicData` `ViewModel` in the rest of the app. The data did not disappear when the phone was rotated. `Questionnaire` was not so simple.

While looking over the code that my classmates had written last year, I discovered that they had implemented functionality to determine what question of the `Questionnaire` a user is viewing. The questionnaire is presented as a series of pages, each one containing a question, a yes button, and a no button. One of the pages of this questionnaire is included below.



When a user taps either the yes button or the no button, the next question appears on the screen. If they tap the back button, that should return them to the previous question. In order to get that functionality to work properly, they made use of the private `ViewModel` variable called `displayState`, which keeps track of the number of screens a user has viewed. This functionality did not seem to be easily replicated without using a `ViewModel`, so I chose to leave that code as it is even though that is not recommended.

The next step was rewriting `QuestionnaireActivity`. Making changes to that class was easy since I was able to model it after `ProfileActivity`. However, `Questionnaire` data was not being stored in the database. Since I made the decision to name documents based on the user's name, I needed to pass that `String` to each `Activity` that needed to use it. I hadn't realized that the next `Activity` that gets called after `QuestionnaireActivity` is `QuestionnaireConfirmActivity`. Passing the `String` value to that `Activity` instead fixed the problem: `Questionnaire` data was now being stored in the database.

To complete this part of the project, all that remained was rewriting the tests. The first test involved ensuring that a user could select an answer for each question and advance to the next one. Like the `ProfileActivity` tests, this was achieved using the Espresso Test Recorder. However, this test did not pass. I quickly recognized that the problem was that whoever either recorded or wrote it thought that there were twelve questions. I confirmed this by looking at the code on the dev branch in Bitbucket. Removing the last piece of code that attempted to answer a twelfth question that didn't exist fixed it.

The second and final test involved testing the `ViewModel` to ensure that it communicated with the database correctly. Since the app no longer needed that functionality, I simply rerecorded the test to ensure that the function of the back button provided by the `ViewModel` works. Once getting both tests to pass, I asked Brady if I should open another pull request for this branch even though we can't merge it. He said yes, and, with that, the `Questionnaire` part of this project was completed.

Global Variables

While discussing the project with Dr. Jackson and expressing the difficulties that I was having passing the `patientName` variable to each `Activity` to retrieve the correct document from the database, he suggested looking into storing it in a global variable instead. I had initially decided on passing a value to an `Activity` through an `Intent` since that was what my classmates had done last spring. However, this sometimes involved passing a value from one `Activity` to another only to get that value to a third `Activity` in the chain. This was tedious, and I welcomed the idea of a better way to do this. Looking at the documentation, I recognized that Android has a class called `SharedPreferences`, which stores a small amount of data that then can be accessed by the entire app. Rewriting the classes I had worked on already to use `SharedPreferences` instead of `Intent`s was easy and made retrieving data from the database much simpler as well. No longer would I have to be concerned with which `Activity` a piece of data was coming from; I could simply place `patientName` in `SharedPreferences` once when it was entered by the user in `ProfileActivity` and access it from any `Activity` anywhere in the app.

ViewSeverityLevel

Looking at the remaining list of classes, I originally chose to move on to the next complete set of Room classes, `MedicationChoice`. Realizing that `ViewSeverityLevel` was next in the app's workflow, I chose to work on that class instead. Rewriting that class was very easy since it simply retrieves the `severityLevel String` from the user's `Questionnaire` document and displays it. Looking next at the test, I found it entertaining since it simply has to open the `Activity` to ensure that the data is being displayed. Brady and I had previously discussed simply merging with the `DemographicData` branch rather than trying to merge with

`dev` until all of the merge conflicts on the `DemographicData` branch could be resolved. I completed this part of the project by merging with myself.

MedicationChoice

The most unusual part of `MedicationChoice` was that its `Entity` did not have the same getters and setters that the other `Entities` possessed. Writing those was not difficult, but it took some time. Like `ViewSeverityLevelActivity`, `MedicationChoiceActivity` also only displays one screen. This one was not as simple, though, as it receives data from the user pressing one of two buttons. The test was also not complicated: it just taps the buttons. Rewriting this class and the test took me about half an hour.

However, `MedicationChoice` also has a related `MedicationDosageActivity` class. This `Activity` allows a user to select how many doses of a medication they would like to take daily if they did not choose to abstain from medication on the previous screen. This class did not require writing a `MedicationDosage` object since the field for the number of doses already exists in the `MedicationChoice` document. I originally believed the correct way to update that field was simply to call the related setter in `MedicationChoice`, but this was not updating the field in the database. I reviewed the documentation and realized that there is an `update` method meant for updating a single field of a document. Changing the code to correctly update the document, I was able to see in the Firestore console the new value for the medication dosage.

TreatmentPlan

One of the two remaining sets of classes to translate was `TreatmentPlan`. Again, I chose which class to work on next according to the workflow. Rewriting `TreatmentPlan` was very

similar to rewriting `DemographicData`: create a `TreatmentPlan` class, assign all of the important data to variables representing each piece of data in `TreatmentPlanActivity`, and use those to create a `TreatmentPlan` object to insert into the database. While writing the class was easy, I ran into unusual errors with the related tests. It seemed as though `SharedPreferences` wasn't working for this class, even though it worked with the others. After looking at how the other classes used `SharedPreferences`, I realized that I had forgotten the class reference before the call to `getSharedPreferences`(). Instead of the code reading `getSharedPreferences`(…), it should have read `TreatmentPlanActivity`.this.getSharedPreferences(…). Changing that one line of code fixed it, but the test still caused errors. I quickly recognized that whoever had written the test was using different arbitrary test data than what I had in the Firestore database. Editing the validation test code to reflect my data allowed the test to pass. I completed this class by merging with the `DemographicData` branch and moved onto the next set.

UserActivities

This set of classes seemed deceptively easy. I followed the usual steps: created a class to replace `UserActivitiesEntity`, edited `ActivityActivity` appropriately, and ensured that the tests passed. One unique aspect of this class was that `ActivityActivity` doesn't use the `UserActivities` class; it only gets data from `TreatmentPlan`. The tests passed, and I chose to work on the `HomeActivity` screen next instead of moving onto the `Activities` related to `UserActivities`.

The `HomeActivity` screen simply retrieves the `lastClean` field from `DemographicData`, finds the number of days between that date and today, and displays that number on the screen. Rewriting that and the corresponding test was easy. Working on the `Activities` related to

`UserActivities` was not. While working on `PlanActivity`, I realized that the database would need to store a list of activities that the user has completed. This seemed to be best implemented using nested documents, but I did not have enough time remaining to work on this project to learn how to write that code. Unfortunately, the project would have to be left unfinished.

Conclusion

Most of the app now has a functioning cloud database. Deciding which of the two databases to use was easy after completing a trade study. Learning how to use the database we had decided on, Firestore, took a significant amount of time but made translating the later classes easier. Not all of the classes were similar to `DemographicData`, but their quirks were not difficult to handle. However, learning how to work with each unique set of classes took more time than I was expecting, and I did not get to translate all of the classes to Firestore. Despite that, working on this project was rewarding, and I feel proud of what I have accomplished.

Appendix

Category	Weight	Criteria	Tools
Support for iOS and Android	Very important	Supports both iOS and Android operating systems.	Software Development Kits for both operating systems
Security and HIPAA compliance	Very important	Protects data in a way that complies with HIPAA requirements.	Firebase Authentication and Realtime Database Security Rules
Easy to learn and use effectively	Moderately important	Has tutorials and extensive documentation.	Code examples
Ease of migration	Very important	Minimal changes need to be made to the existing SQL-based database structure.	Must be structured as a JSON tree
Realtime data syncing	Moderately important	Allows a user to view new data immediately.	Data synchronization even when offline
Scalability	Moderately important	The database grows as the app gains more users.	Requires sharding once a certain limit is reached
Offline support	Moderately important	Data can be stored, accessed, and updated while offline. Data updated while offline is uploaded to the database as soon as the connection returns.	Disk Persistence
Support for complex queries	Less important	Provides ways to sort and apply multiple filters to one type of data.	Limited to one sort method per query, but can have multiple filters
National or international access	Moderately important	Anyone in the US or other countries can retrieve data quickly. (Anyone can download the app, but people who are farther way from where the data is stored might experience a significant delay.)	Single-region availability
Unlimited writes	Moderately important	Large amounts of data may eventually need to be written to the database at one time.	Only limited in size: 16 MB per write

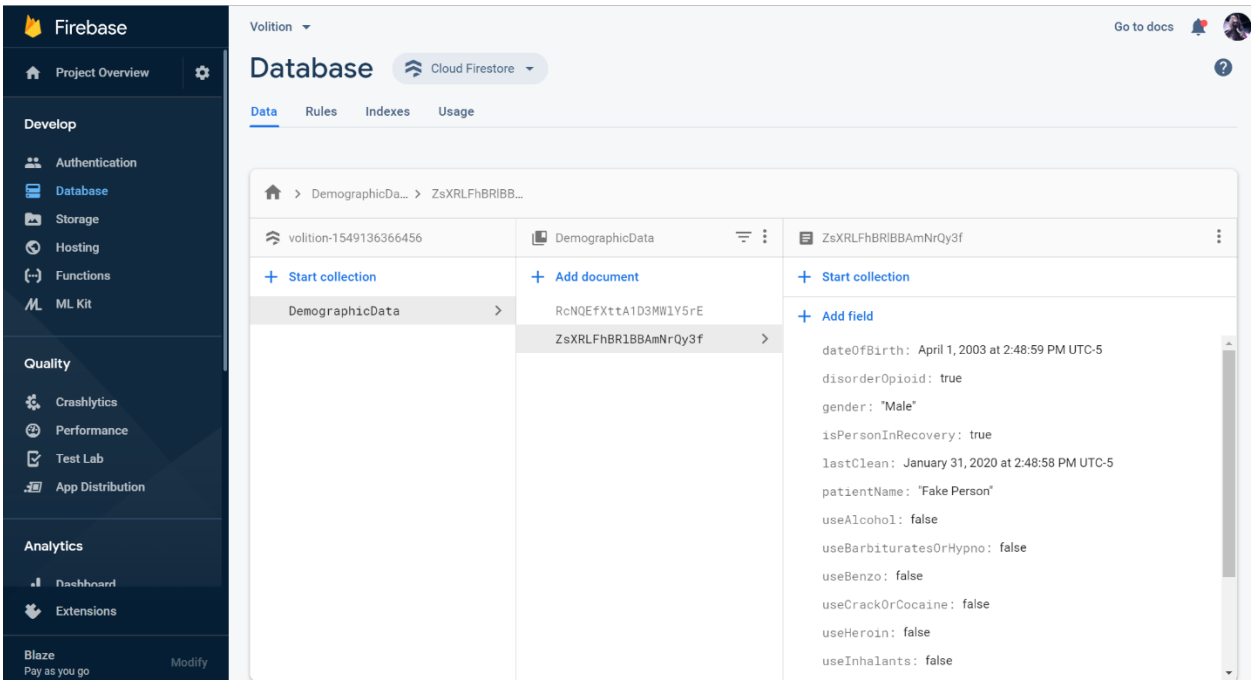
Trade study table for Google Realtime Database without numerical weights.

Category	Weight	Criteria	Tools
Support for iOS and Android	15	Supports both iOS and Android operating systems.	Software Development Kits for both operating systems
Security and HIPAA compliance	15	Protects data in a way that complies with HIPAA requirements.	Firebase Authentication and Cloud Firestore Security Rules
Easy to learn and use effectively	8.3333333	Has tutorials and extensive documentation.	Codelabs and API Reference
Ease of migration	15	Minimal changes need to be made to the existing SQL-based database structure.	Flexible data model
Realtime data syncing	8.3333333	Allows a user to view new data immediately.	Java classes and methods that watch for changes
Scalability	8.3333333	The database grows as the app gains more users.	Google Cloud Platform infrastructure
Offline support	8.3333333	Data can be stored, accessed, and updated while offline. Data updated while offline is uploaded to the database as soon as the connection returns.	Built-in data caching
Support for complex queries	5	Provides ways to sort and apply multiple filters to one type of data.	Chained filters, combinations of filtering and sorting, and default indexing
National or international access	8.3333333	Anyone in the US or other countries can retrieve data quickly. (Anyone can download the app, but people who are farther way from where the data is stored might experience a significant delay.)	Multi-region data replication
Unlimited writes	8.3333333	Large amounts of data may eventually need to be written to the database at one time.	Maximum of 10,000 writes per second
Sum:	100		

Trade study table for Google Firestore with numerical weights.

Category	Firestore	Realtime Database	Winner
Support for iOS and Android	9	9	tie
Security and HIPAA compliance	9	9	tie
Easy to learn and use effectively	8	2	Firestore
Ease of migration	4	1	Firestore
Realtime data syncing	8	8	tie
Scalability	8	5	Firestore
Offline support	8	5	Firestore
Support for complex queries	7	6	Firestore
National or international access	8	2	Firestore
Unlimited writes	5	5	tie
Cost	9	4	Firestore
Sums:	83	56	

Final comparison trade study table, showing Firestore as the winner.



Sample demographic data shown in the Firebase console.

Works Cited

"Cloud Firestore Android Codelab." *Codelabs*, Google, codelabs.developers.google.com/codelabs/firestore-android/#0.

Friendly Eats. Github, 10 Feb. 2020, github.com/firebase/friendlyeats-android.

"Save Data in a Local Database Using Room." *Documentation*, Android Developers, 27 Dec. 2019, developer.android.com/training/data-storage/room.

"Timestamp." *Documentation*, Firebase, firebase.google.com/docs/reference/android/com/google/firebase/Timestamp.