Blue HomeLab App

Software Guide

*By Shayna Rosado*

*Last updated: 4/21/20*

TABLE OF CONTENTS

**Part 1: Introduction**

* 1. What is the Blue HomeLab?
  2. What are the components of the Blue HomeLab system?
  3. Setting up
     1. Git and accessing code on GitHub
        1. Git Installation
        2. Pushing code
        3. Downloading the code
        4. Installing project dependencies locally
     2. React Native
        1. Installation
        2. Running the app
     3. Visual Studio Code

**Part 2: Firebase**

* 1. Why Firebase?
  2. Firebase Console
  3. Cloud Firestore
  4. Cloud Messaging
  5. Cloud Functions
  6. App Distribution
     1. Building the app (Android)
     2. Distributing the app

**Part 3: Project Files**

* 1. package.json
  2. /\_\_tests\_\_
  3. / .vscode
  4. /android
  5. /components
     1. Firebase config files
     2. /functions
     3. index.js
     4. package.json
     5. QualityPage.js
  6. /ios
  7. /node\_modules
  8. /styles
  9. Miscellaneous/Configuration files
  10. Regular Expression error fix

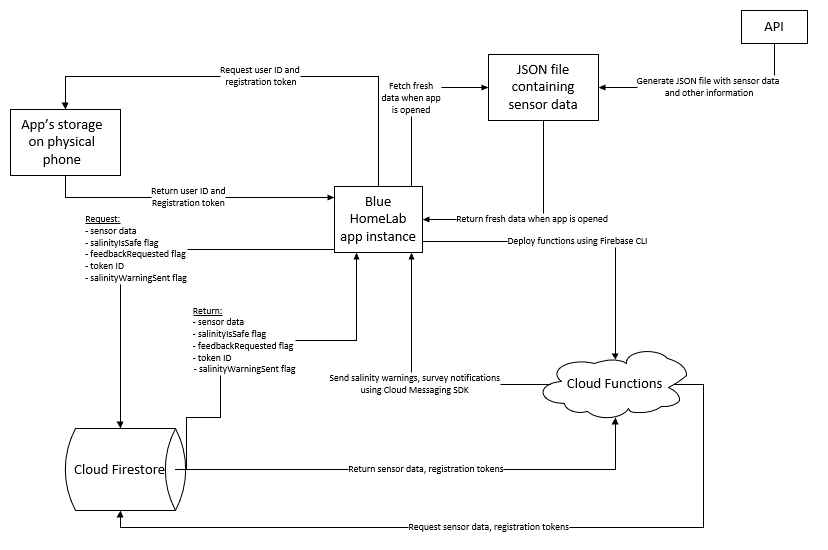
**PART 1: INTRODUCTION**

* 1. *What is the Blue HomeLab?*

The Blue HomeLab is a mobile application which gives people the ability to access real-time information about water quality. Why is the need for this technology so important? The answer to that question is that our bodies need water more than anything else. As humans, we can survive on food for weeks, but without access to water, we won’t last more than a few days.

When a contamination occurs in a given water supply, the exact nature of the contamination, including what chemicals are involved, and when the contamination occurred, should be transparent for water consumers. Currently, however, it takes weeks for water to be brought to a lab, tested, and evaluated—yet the technology already exists for water to be monitored in real time. By using sensors to measure properties of water, such as salinity, in conjunction with the Blue HomeLab, people can stay healthy and informed about what exactly is in their drinking water.

* 1. *What are the components of the Blue HomeLab system?*



The Blue HomeLab is more than just an app; it is an entire system comprised of different parts. There are also parts which are external to the system which the BlueHomeLab interacts with. These parts include the database, whose input is data collected from every sensor owned by the Blue CoLab, and the API which hosts this data at an endpoint on the Internet. Currently, the sensor data related to the body of water being monitored by the HomeLab app is hosted at:

<http://vulcan.seidenberg.pace.edu/~lkeeley/temp/db.json>

From there, the data is transferred to the app’s “backend,” Firebase. Google’s Firebase service provides us with storage, background functions which can be run at a regular set interval. Integration of the Firebase SDK in the Blue HomeLab also enables notifications to be sent to user devices. Firebase Cloud Functions check the sensor data being hosted at the API endpoint, pull that data into the Firebase Firestore, and then decide what to do next. If the salinity value is out of range, Clound Functions will execute code to fire off a notification, while using device registration tokens stored in the Firestore to “address” these notifications.

Firebase is responsible for distributing data to the client app. When Blue HomeLab is opened on a device, many things happen. First, the app communicates with Firebase’s servers to read sensor data from the Firestore. Next, the app initializes a unique ID for the user. This is implemented in the code using AsyncStorage. Because the same value is stored in the app’s local storage, the app will always have the same ID assigned to it, unless the app is deleted from the user’s phone and later downloaded again. The app also verifies the user registration token assigned by Firebase using AsyncStorage. If Google updates the token, the client app will then update its token value stored in AsyncStorage.

Keep reading for the details! To jump right into the main code, see files *~/components/QualityPage.js* (client app main code) and *~/components/functions/index.js* (Cloud Functions).

* 1. *Setting up*

The first thing you will need to get started is React Native. React Native is an open-source mobile application framework created by Facebook. It is useful because the idea is that you write JavaScript code in one place, while the native code is abstracted away (until you need it, of course), enabling your app to be developed quickly and offering cross-platform compatibility. Please follow the installation documentation EXTREMELY carefully—skipping over one step could break things and keep you from testing the app! Also, make sure you read and follow the instructions for setting up the React Native CLI (not Expo). While Expo is a really awesome tool and helps gets apps up and running even faster, a lot of libraries used for this project are not compatible with Expo, which is why the project was “ejected” from Expo early on during development.

<https://reactnative.dev/docs/environment-setup>

By the end, you should have:

* Node, Java 8, and Python2 via Chocolatey
* Android Studio & Android SDK
* A real Android device or an emulator

If you want to experiment and start your own app from scratch, use the command “*npx react-native init YourProjectName”*. To read documentation covering the basics of React Native, visit:

<https://reactnative.dev/docs/tutorial>

* + 1. *Git and accessing code on GitHub*

In order to access the code for this project, you will need to be added to view the private repository on GitHub. Once you are added, the repository is located at:

<https://github.com/shaynajrosado/BlueHomeLab/>

* + - 1. *Git Installation*

Next you should install Git. Git is essential, no matter where the repository is located, for version control and keeping track of changes made to the code. Git can be downloaded by following the directions at:

<https://git-scm.com/download/win>

* + - 1. *Downloading the code*

First, create a folder on your computer where you want the project files to exist. Next, open up a File Explorer window and navigate to that folder. Right-click within the empty folder and select “Git Bash here.” Initialize git by using the command “git init” inside of the project root folder. Next, type the command “git clone <https://github.com/shaynajrosado/BlueHomeLab/>” to get a copy of the repository on your computer. To create a local branch to keep track of your local changes, enter “git checkout -b yourBranchNameHere.” Now, you can begin editing code!

* + - 1. *Pushing new code*

To push code to GitHub, first type the command “git add .” to track any new files you added to the tracking system. Next, to make a commit, use the command “git commit -m ‘Your commit message here’.” Finally, enter “git push” to push the code to GitHub’s servers.

* + - 1. *Installing project dependencies locally*

By now, you should have React Native installed. To install other imported modules, open a Window’s Command Prompt (run as Admin where possible), navigate to the root project folder and use command “npm install” to install all project dependencies. This command goes through package.json (also located in the root directory), locates required packages, and installs them.

* + 1. *Starting the development server*

Next, we will start the npm development server. To do this, navigate to the root folder in Command Prompt and enter the command “npm start.”

* + 1. *Running the app*

Once the npm server command says “Loading dependency graph…done” then we can run the app on the emulator or on the physical device. If you are using an emulator, start it now ad make sure it is fully loaded before proceeding. Next, type “npx react-native run-android.” Wait for the app to build, and then it will be launched on the phone.

* + 1. *Visual Studio Code*

I recommend using VS Code to edit and debug the code in the app. VS Code has a lot of useful extensions, such as React Native Tools for setting breakpoints and debugging code. Personally, I like to log values to the console at various places in the code, but stepping through code is useful when developing in any language. Do what you are most comfortable with.

VS code also has a Git tab which allows you to compare and contrast changes from the last commit on your branch. This is also helpful when dealing with merge conflicts.

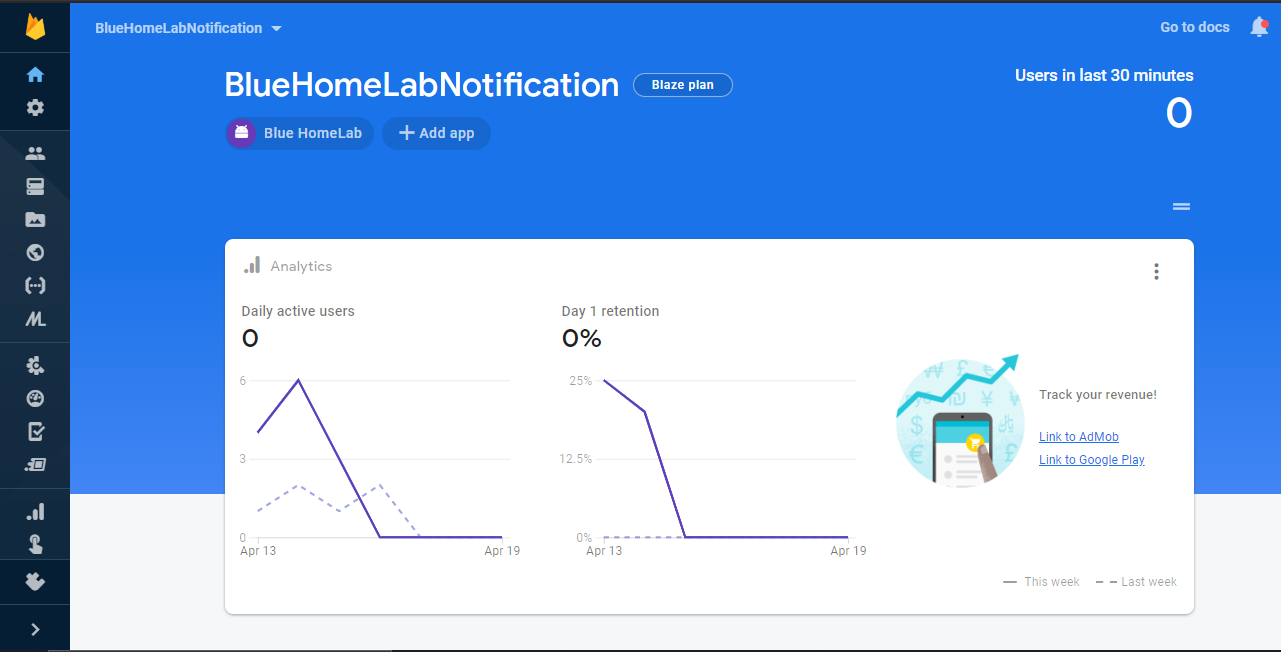
**PART 2: FIREBASE**

* 1. *Why Firebase?*

In order to store data and run functions “behind-the-scenes,” we need Firebase. Firebase, made by Google, is an extension of the popular Google Cloud Services. Firebase provides the HomeLab with a centralized database, notification server, distribution center, and analytics hub all rolled into one.

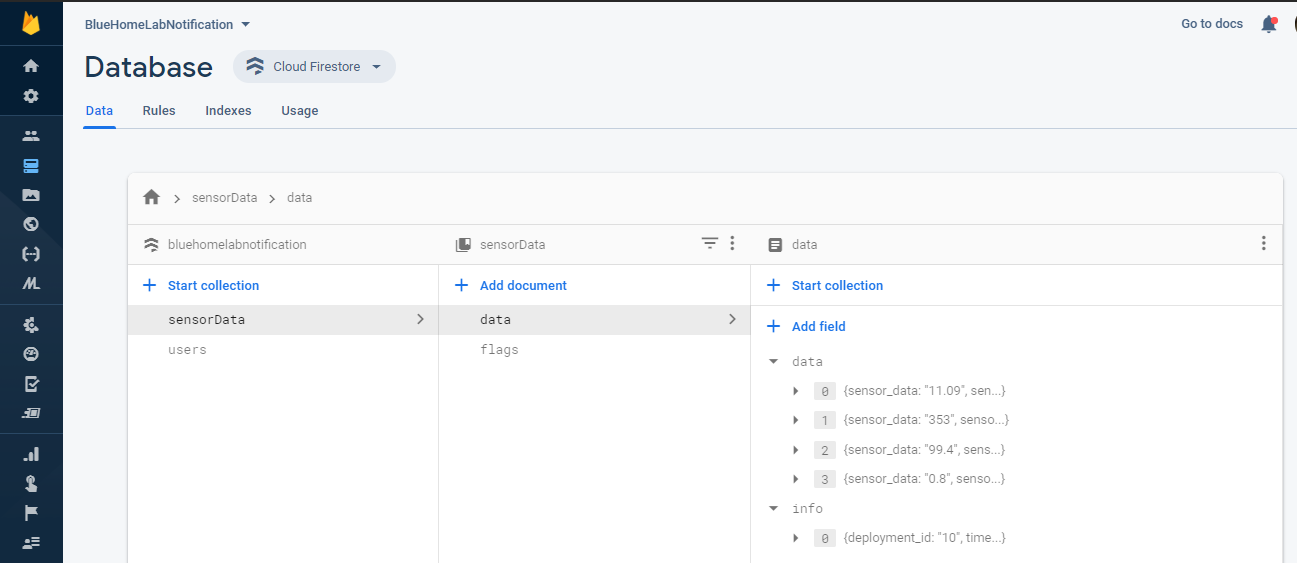
* 1. *Firebase Console*

The Firebase Console is where you can find an overview of all information collected by Google about a given app or project, displayed in various cards. While the Analytics card is shown below, other cards display overviews of data linked to the other Firebase Services, which we will go through next.



* 1. *Cloud Firestore*

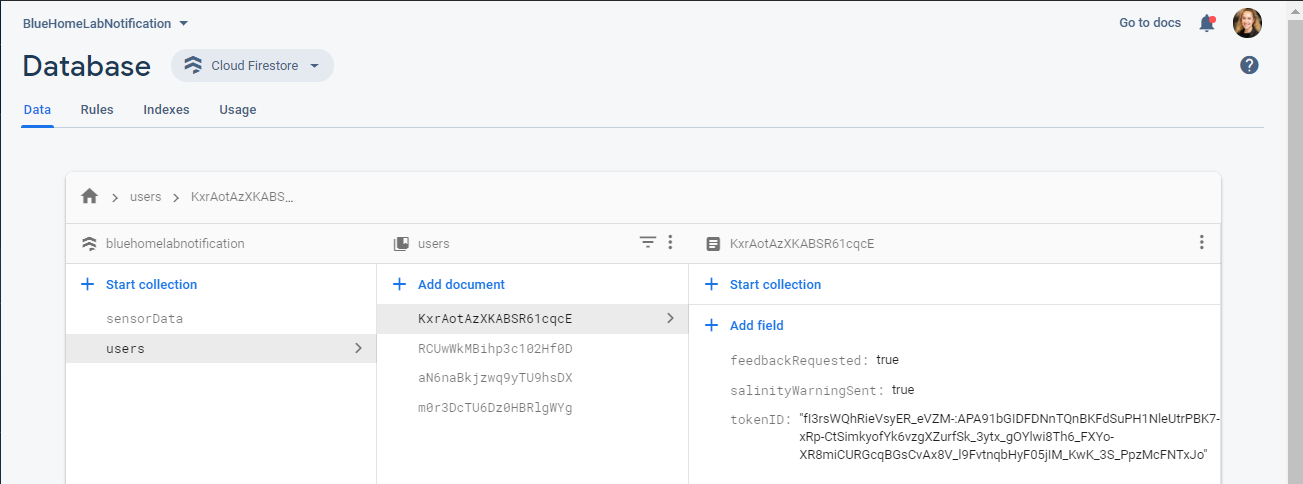
The Cloud Firestore is the app’s “backend” NoSQL database. Full CRUD (create, read, update, delete) operations can be made by code in either the client app or in Cloud Functions. To learn how to write code to do this, visit <https://firebase.google.com/docs/firestore/quickstart>.



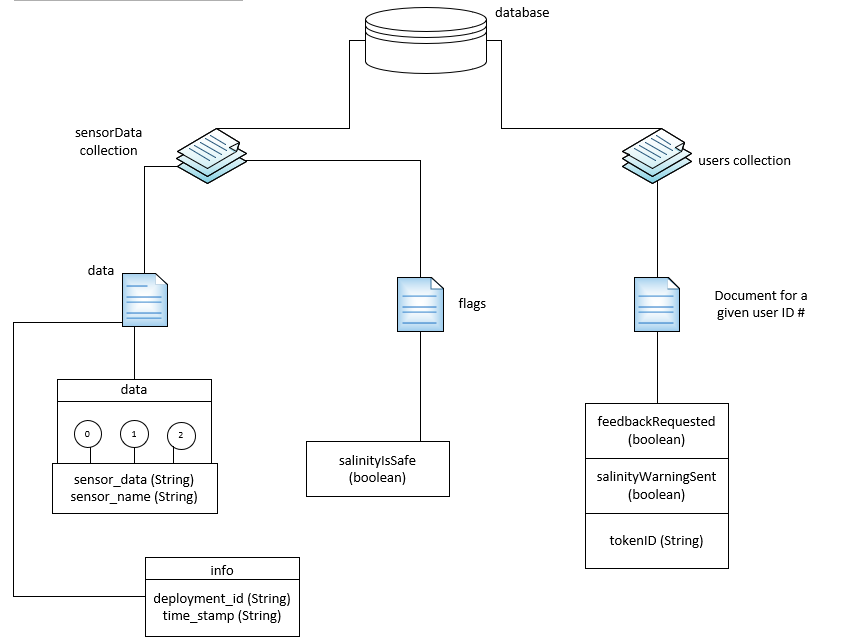
As shown above, there are two collections in the database: sensorData and users. The sensorData collection contains two documents: data and flags. The data document has the fields data and info. Objects in the data field represent different sensors and their values. In the info field, there is a single object denoting the deployment\_id and time\_stamp key-pair values as denoted by the JSON file which was parsed by a cloud function.

In sensorData/flags, there is one field: salinityIsSafe. This flag is updated by a Cloud Function, run at a regular interval, once the salinity value enters or exits an acceptable range of values. For instance, a salinity value of 0.6 would cause salinityIsSafe to change to false, but then later on, if the value changes back to a value that is less than 0.5, salinityIsSafe will be true again. The logic for this is in the code of file ~/components/functions/index.js in the checkValue() method. If the salinity has reached an unsafe value, sendNotification() in the same file will be called. salinityIsSafe is also checked by client apps when rendering the color for the salinity quality indicator. If salinityIsSafe == false, then the indicator will be red, while if salinityIsSafe == true, then it will be green. For the code, see ~/components/QualityPage.js.

Next is the users collection. Right now, the users collection looks like this:



The middle column shows the user IDs generated and assigned to the user the first time they open the app on their phone. Each user ID is assigned three fields: feedbackRequested, salinityWarningSent, and tokenID. feedbackRequested denotes whether or not the user successfully received the salinity warning and if a link to the notification user feedback survey will be displayed in the app. The salinityWarningSent flag denotes whether or not a user has received a salinity notification for a given contamination. It is used in conjunction with feedbackRequested when determining if a notification feedback survey needs to be sent.



* 1. *Cloud Messaging*

The Cloud Messaging SDK enables notifications to be sent from Cloud Functions to users’ phones. To send a message, you’ll need the device’s tokenID as stored in Firestore. The code which retrieves and updates this value is in the getToken() method of ~/components/QualityPage.js. For more information, see <https://firebase.google.com/docs/cloud-messaging/android/first-message>.

* 1. *Cloud Functions*

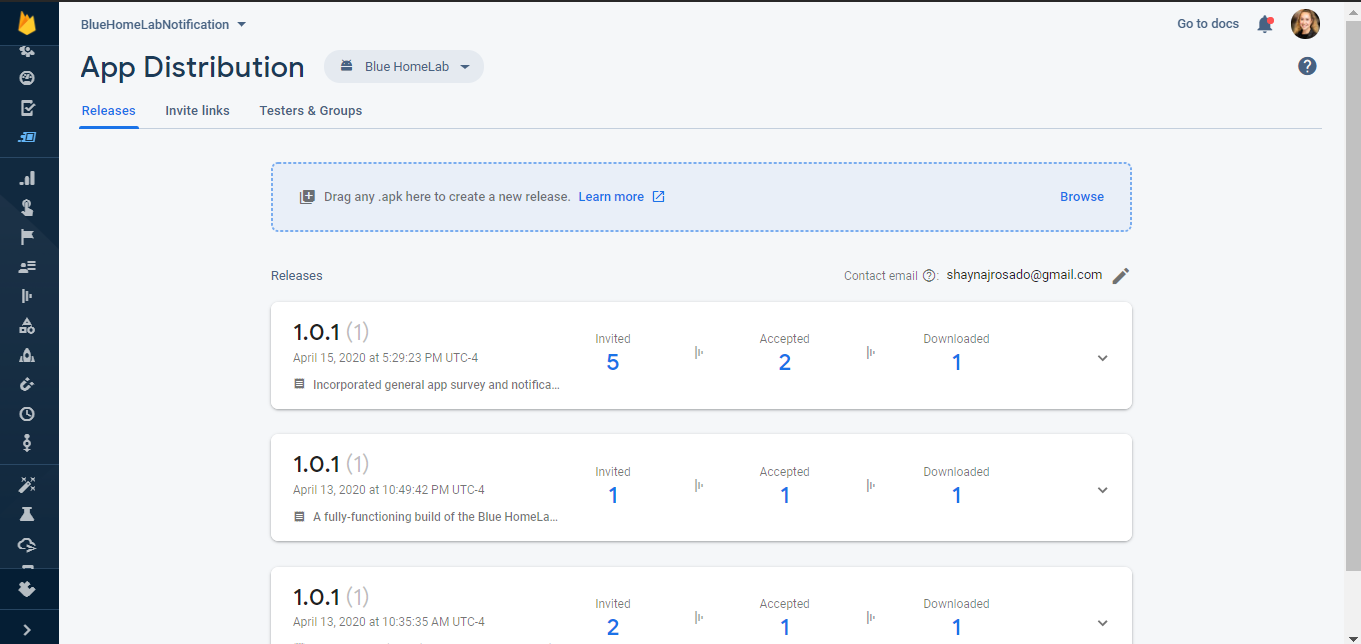
Cloud Functions enable code to be executed via Google’s servers—external to app code execution. This allows us to collect user data, scan in sensor data, determine if the salinity value is out of range, and send out notifications for both the salinity warning and prompt to return to the app and complete the salinity notification survey. File ~/components/functions/index.js contains all code for Cloud Functions. Additionally, the Firebase CLI (which provides tools for managing, viewing, and deploying to Firebase projects) is installed in ~/components/functions. From ~/components/, command “firebase” can be run via Windows Command Prompt or the terminal in VS Code to deploy new or updated code to Google’s Cloud Function servers. Some common uses of the “firebase” command:

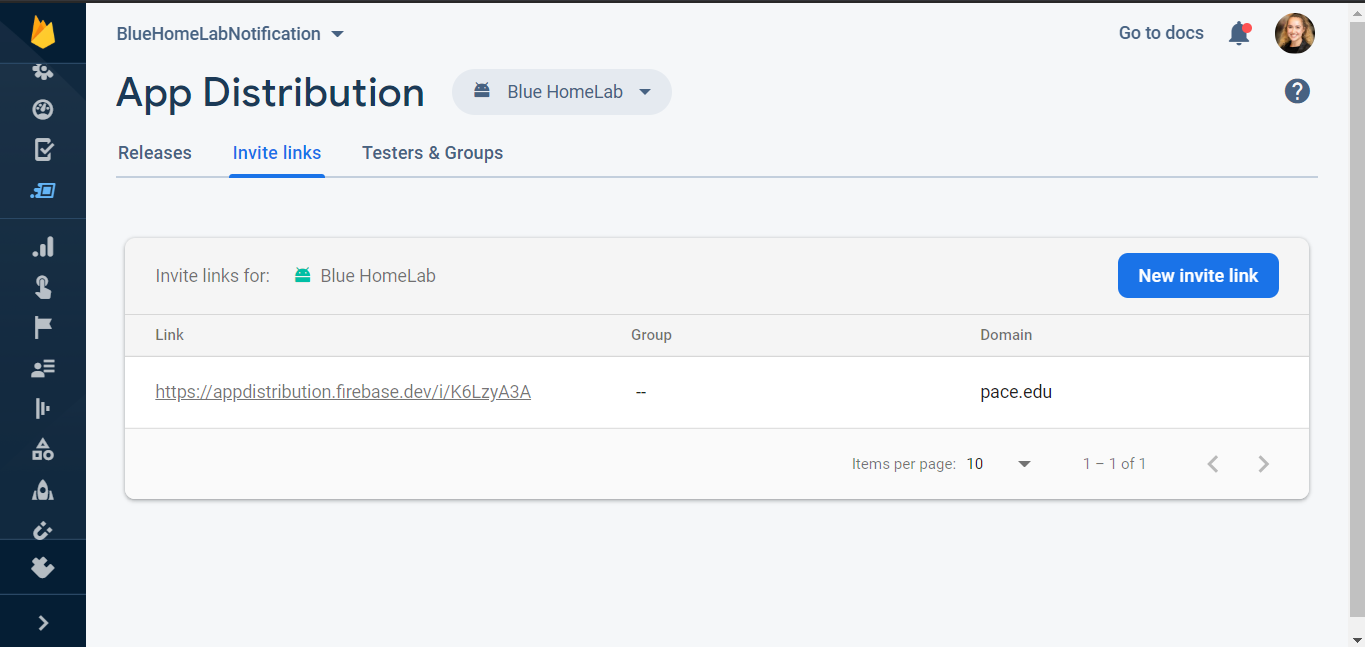
* firebase deploy = publish functions to Firebase
* firebase init = set up project directory with Firebase products
* firebase use = view a list of currently defined projects
  1. *App Distribution*
     1. *Building the app (Android)*

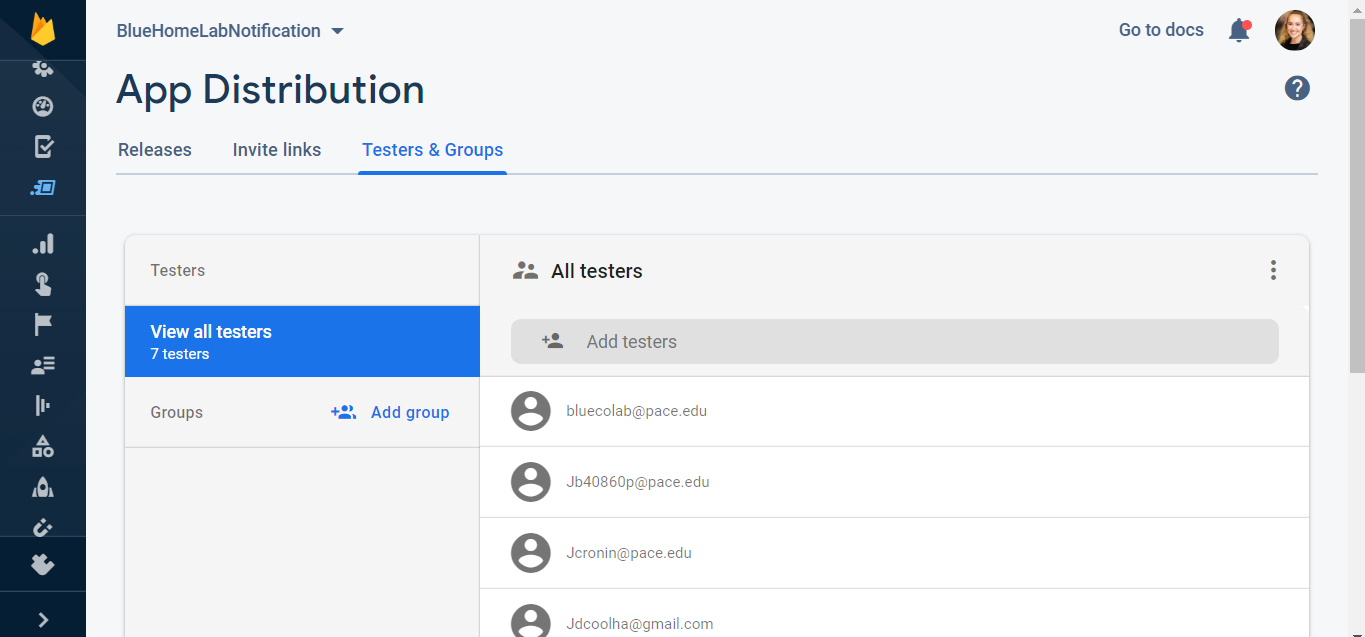
To generate a build for the app, navigate to ~/android/ and run Windows Command Prompt as an Administrator. Run command “gradlew assembleRelease.” This generates a .apk file and may take some time. If you want a .aab file (for instance, this is required when publishing to the Google Play store), run “gradlew bundleRelease”). When the command is finished, navigate to ~\android\app\build\outputs\apk\release. You should see file app-release.apk; this is a distributable file form of the app. Opening this file on an Android device should install the app. If the app fails to install, there is likely an issue related to the signing of your app bundle. For more information, visit <https://developer.android.com/studio/publish/app-signing>.

* + 1. *Firebase Distribution*

The good thing about having a .apk file is that it can be easily distributed to select users via Firebase!



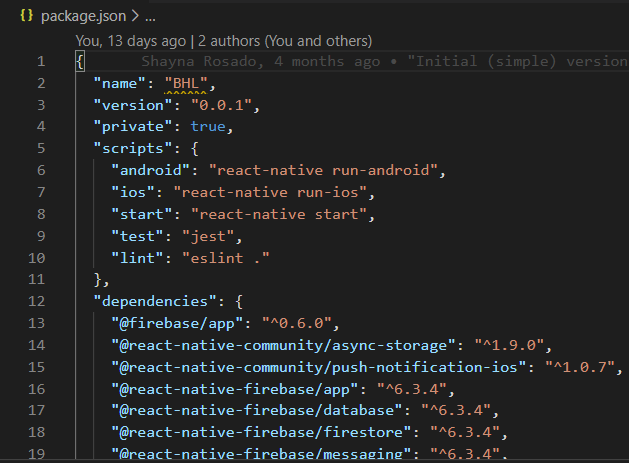




**PART 3: PROJECT FILES**

* 1. *package.json*

This file contains important data for npm, which manages the imports of the projects. The name of the app and the version number can be set in this file. Also, scripts are defined for npm commands. Running “npm start” from the command line, for instance, starts the development server, enabling fast-refresh (app will re-render every time code is updated & saved in Visual Studio code) and the printing of anything logged to the console.



* 1. */\_\_tests\_\_*

This is a space for unit tests to be kept.

* 1. */ .vscode*

This folder contains VS code data.

* 1. */android*

This folder contains the Android version of the app and other necessary data for the app to build. If you want to use Android Studio to rebuild the project or run the code in an emulator, you will need to open the project from this folder (not the root directory!) for things to work properly. Gradle configuration files, which are used to build Android apps, can be found here. Any builds will also be generated within this folder (see Section 2.6.1).

* 1. */components*

This folder contains the main code for the client app in QualityPage.js, and files for Firebase, both of which we will cover shortly.

* 1. *Firebase config files*

|  |  |
| --- | --- |
| File name | Description |
| .firebaserc | Stores project aliases |
| firebase.json | Project configuration |

For more information, see <https://firebase.google.com/docs/cli>.

* 1. */functions*

This folder is mainly for Cloud Functions. package.json lists the dependencies for Cloud Functions. Running npm install in the root directory of the project installs these dependencies in the ~/components/functions/node\_modules folder. The main file to be concerned with is index.js, which we will cover next.

* 1. *index.js*

Index.js contains the all Cloud Functions code. After editing this file, it is important to run “firebase deploy” from ~/components/functions to update the code on Google’s servers, otherwise, any new code after the most recent deploy won’t execute. This file contains an object containing the configuration for the app (information for Google). Below is an overview of the methods in this file:

|  |  |
| --- | --- |
| Function name | Description |
| requestFeedBack\*\* | Determines whether users have received a salinity warning, and if so, sends out a second notification prompting the user to return to the app and fill out the notification survey. |
| evaluateSensorData\*\* | Fetches sensor data and determines whether or not a notification should be sent |
| placeData(data) | Places sensor data into Firestore. |
| checkValue(data) | Verifies whether the salinity value is in/out of range. |
| getRegistrationTokens() | Retrieves all registration tokens to send out a notification. After a notification is sent, a second one will deploy, notifying users that it was just a test & prompts the user to return to the app and complete the user feedback survey. |
| sendNotification(registrationList) | Sends a salinity warning to all registered users. |
| sendSecondNotification(registrationList) | Sends a second message prompting the user to return to the app to access the Notification Survey. |

\*\* = these are the scheduled functions which are exported to Google’s servers. Their .schedule() method takes a string describing how often they should run. Examples:

- ‘every 2 minutes’

- ‘every 3 hours’

- ‘\* \* \* \* \*’ (unix-cron notation for every minute, every hour, every day)

- ‘0 \*/6 \* \* \*’ (unix-cron notation for every hour at :00, every 6 hours, every day)

For interactive practice with unix-cron notation, visit <https://crontab.guru/>.

* 1. *package.json*

Package.json lists all dependencies needed by the app to function. When “npm install” is run from the root directory, all packages listed in this file are installed in /node\_modules.

* 1. *QualityPage.js*

This is like the “main” method of the client app. It contains the code for setting the user’s ID, registration token, getting sensor data, setting the state of the app, and dynamically rendering the app’s display.

When the app is opened on a phone, the constructor executes first. This is where values are declared and initialized. These important variables include registrationToken, senderID (links code to Firebase notifications), and most importantly, the state. The state object holds critical variables, including the userId, the salinityIsSafe flag (a local instance mirroring the salinityIsSafe value from Firebase), the displaySurveyLink flag from Firebase, the overallQuality of water, and the sensor data. After the constructor, componentDidMount is called.

Inside componentDidMount, setToken() is first executed. This method gets the registration token from Firebase using:



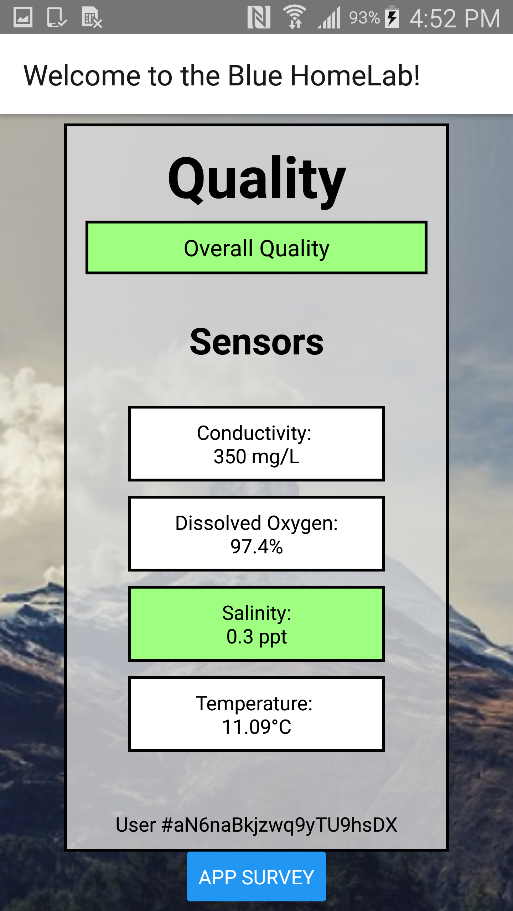
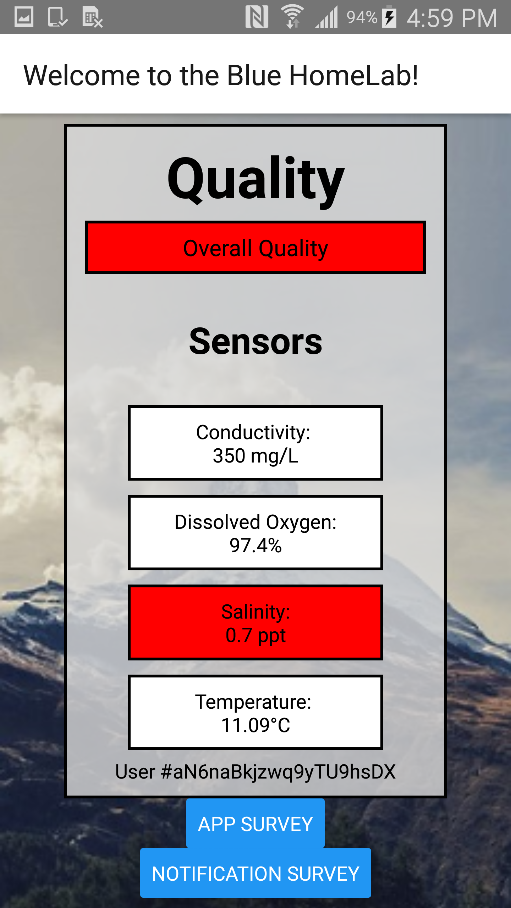
Then, this value is stored in AsyncStorage (the app’s local storage on the device). After this, the user ID is initialized. If the user already has a user ID in AsyncStorage under the ‘USER\_ID’ key, it is stored in the app’s state and used to look up user-specific information from Firebase, like whether or not to display the Notification Survey after an alert is sent out. If there is no user ID in AsyncStorage (this is the person’s first time using the app), then a 20-character randomized ID is generated in makeID(20), and later stored in AsyncStorage for later use.

Still in the setToken() method, next the app checks if there is an entry for the user in Firestore. If there isn’t, one is created at this point. If there is, the ID number of the existing user document in Firestore is printed, and the document is then updated with the latest Firebase registration token obtained from the code snippet shown above.

After exiting the setToken() method, execution returns to componentDidMount. Then, getData() is called, which retrieves the sensor data from the sensorData collection in the Firestore. Doing this allows the app to get fresh data every time it is opened (in addition to the Cloud Functions code running around the clock). Next, the salinityIsSafe flag is retrieved from Firestore (by calling getSalinityFlag()) and saved in the app for the purposes of determining the style for the Salinity (and Overall Quality) indicators later on in the code of this file. Similarly, getSurveyFlag() stores the feedbackRequested flag from Firestore in the QualityPage.js displaySurveyLink variable.

Since at the time of writing, salinity is the only factor affecting the Overall Quality indicator, if salinity is safe, overall quality is safe. So, finally, the code determines the overall quality of water and saves it in the state of the app.

Down at the bottom of the file is the render() function. This function defines what will be displayed given the state of the app. If salinity is high, display a red color, for instance. This logic comes first, and then there is the return statement, which returns the tagged components which will display the selected parts of the app.

* 1. */ios*

This folder contains resources for the IOS version of the app.

* 1. */node\_modules*

This folder contains all external modules (outlined in ~/package.json) required for the app to function.

* 1. */styles*

This folder contains styles.js, which is essentially a JavaScript object containing a CSS StyleSheet for the app.

* 1. *Miscellaneous/Configuration files*

You don’t really need to touch these files. However, it is useful to know what the main ones are for:

|  |  |
| --- | --- |
| File Name | Description |
| .buckconfig | Build configuration for Buck |
| .gitignore | Files which won’t be tracked by version control |
| .prettierrc.js | Formatting settings |
| app.js | Stores the name and display name for the app. The name is used for the package name, too. |
| Index.js | Registers the top-level component of the app, App.js |
| App.js | Top-level container for the app |
| Background.png | Background image for the app. |
| Regularexpressionfix.txt | See Section 3.11 |
| ScreenContainer.js | Code for setting up each/more screen(s) of the app |

* 1. *Regular Expression error fix*

There is a common issue with React Native where you will run “npm start” (usually for the first time) and you will get an error about a regular expression. The fix for this issue can be found in ~/regularexpressionfix.txt.