A picture containing icon

Description automatically generated

System Description

**For Cloud-Connected Toaster Oven**

**Document Version:** 1.0

**Date:** 05/18/2022

**Contents**

[1 OVERVIEW 4](#_Toc387220239)

[1.1 Product Description 4](#_Toc387220240)

[1.2 Data Model 5](#_Toc387220241)

[1.3 Technology 5](#_Toc387220243)

[1.4 Development Tools 6](#_Toc387220244)

[1.5 Services 6](#_Toc387220245)

[2 Support details 8](#_Toc387220248)

[2.1 Third Party 8](#_Toc387220250)

[2.2 Standard Tasks 8](#_Toc387220251)

[3 Cloud Details 10](#_Cloud_Details)

[3.1 General Setup 10](#_General_Setup)

[3.2 Pub/Sub Setup 10](#_IoT_Core_Setup)

[3.3 IoT Core Setup 10](#_IoT_Core_Setup)

[3.4 Google Actions Setup 14](#_Google_Actions_Setup)

[3.5 Firestore Database 16](#_Firestore_Database)

[3.6 Firebase NodeJS Setup 21](#_Firebase_NodeJS_Setup)

[3.7 Cloud Functions 33](#_Functions_in_General)

[3.8 Fulfillment 34](#_Fulfillment)

[3.9 Function Deployment 35](#_Function_Deployment)

[3.10 Function to IoT Core 36](#_Function_to_IoT)

[3.11 IoT Core to Firestore 37](#_IoT_Core_to_1)

[3.12 IoT Core to Device 37](#_IoT_Core_to)

[3.13 OAuth and Account Linking 37](#_Device_to_IoT)

[4 Embedded Details 42](#_Cloud_Details)

[4.1 ESP32 Documentation 42](#_General_Setup)

[4.2 UART Changes 42](#_General_Setup)

[4.3 Com Module Changes 43](#_Google_Actions_Setup)

[4.4 CloudCommandParser 44](#_Firestore_Database)

[4.5 Control State Machine 44](#_Firestore_Database)

[4.6 Wi-Fi State Machine 45](#_Firestore_Database)

[4.7 Temperature Reading 45](#_Firestore_Database)

[4.8 Hardware Documentation 46](#_Firestore_Database)

[5 MOBILE APP Details 48](#_Cloud_Details)

[5.1 General Setup 48](#_General_Setup)

[5.2 App Structure 5](#_General_Setup)1

[5.3 Package Usage](#_General_Setup) 52

[5.4 Testing 5](#_Google_Actions_Setup)4

# 1 Overview

## Product Description

This goal of this product was to create a cloud-connected toaster oven. Given a base model toaster oven, additional embedded software and hardware components were added to provide online connection capabilities while interoperating with the existing hardware. Additionally, the necessary cloud infrastructure was deployed using the Google Cloud Platform to store the data and logic required to communicate with the hardware using the Internet. Finally, a custom mobile application was developed to allow the user a means of wirelessly controlling their device. Alternatively, a custom Google Action was created, adding support for Google Assistant interactions.

Diagram

Description automatically generated

## Data Model

Diagram, schematic

Description automatically generated

Further details regarding these components are described in future sections. At a high level, users can use with either the Google Assistant or their mobile app to interact with their device, or a webapp when linking their account with the Google Assistant. These three points will then utilize functions hosted on Cloud Functions to both modify the Firestore database and send requests to IoT Core. IoT Core acts as an MQTT Broker and sends configuration updates and commands to the physical device. Alternatively, the user may modify their physical device directly, in which case the ESP32 will send either a state update or telemetry back to IoT Core and the specified Pub/Sub topic. In turn, a cloud function will detect this new data and update Firestore accordingly. The Google Assistant and mobile app will then be able to query Firestore and update their displays for the user. As a result of both systems, bidirectional communication is achieved.

## Technology

#### Embedded

The following technologies and standards were used to create the embedded software:

* C/C++
* UART Protocol
* Bluetooth Low Energy (BLE)
* MQTT Protocol
* Matter Standard

#### Cloud

The following technologies and standards were used to create the cloud software:

* JavaScript
* NodeJS
* JSON
* MQTT Protocol
* Matter Standard

#### Mobile App

The following technologies and standards were used to create the mobile app software:

* Flutter
* Dart

## Development Tools

The following IDEs were used and are recommended to be used for further development:

* Arduino IDE: Used to develop code for the ESP32.
* Keil μVision: Used to develop code for the connected toaster oven on the Nuvoton board.
* Visual Studio Code: Used to develop the account linking endpoint and cloud functions which manage communication between the mobile app and the toaster oven through the cloud.
* Android Studio: Used to develop the mobile application.

GitLab was chosen to manage requirements, track time, assign tasks, host our remote repository, and execute our CI/CD pipeline. Our current pipelines run only in the main branch or on merge requests. The .gitlab-ci.yml file is where all the CI/CD logic is stored for GitLab. This file sources the .cicd-files/firebase-cicd.yml file. The jobs in firebase-cicd.yml build, test, and deploy firebase functions (Cloud\_Code/Firebase/functions) and the web app (Cloud\_Code/Firebase/web).

## Services

The Cloud team made extensive use of Google’s services to create the cloud infrastructure. They are outlined here and described in further detail in Section 2.1.

Our cloud services were deployed using the following:

* Firebase Development Platform
  + Authentication: Stores users who login to our service with their Google accounts. Firebase automatically stores their corresponding user ID and an ID Token to link the user with their device(s) and to authorize the Google Assistant.
  + Firestore: Stores device and user information in a NoSQL real-time database. Stores the most recent device state information.
  + Hosting: Hosts a webapp that allows users to link their account with Google, thus enabling use of the Google Assistant.
  + Functions: The Firebase Functions API allowed us to trigger HTTP functions in response to changes in our Firestore database. This is useful for responding to requests from the mobile app or changes from the device sent to the cloud.
* Google Cloud Platform
  + IoT Core: Used to communicate directly with the device. There are two directions of communication, each with two types of messaging. First, when communicating from the cloud to the device, you can send Configuration updates or Commands. Otherwise, when receiving data from the device, the toaster oven can send data as a State update or Telemetry data. Both directions use the MQTT protocol and can be configured with a desired Quality of Service (QoS) level.
  + Pub/Sub: Receives state updates from the device. State updates and Telemetry data is routed to a custom Pub/Sub topic when it is received. Then, a Cloud Function will execute, which updates our Firestore database with the relevant changes.
  + Cloud Functions: Hosts our HTTP functions that our cloud infrastructure uses to manage device state and communication.
  + Secret Manager: Securely stores secret information and API keys that our cloud infrastructure relies on, such as the client ID and client secret used with the Google Assistant, or the JWT secret used to sign and verify our JWTs used in authentication.
* Google Actions Console
  + Account Linking: Stores a link to our account linking endpoint where the user can authorize the Google Assistant to interact with their cloud-connected toaster oven.
  + Fulfillment: Stores a link to the entry point of our Google Assistant fulfillment services. These are the HTTP functions that execute when the user speaks to their Google Assistant, and queries/updates their device accordingly.

# Support Details

## Third Party

#### Cloud

The following third-party libraries were used by the Cloud team:

* [Actions on Google](https://www.npmjs.com/package/actions-on-google): Used to initialize a Smart Home instance required to execute fulfillment intents with the Google Assistant. See the Fulfillment section for more information.
* [Google APIs](https://www.npmjs.com/package/googleapis): Used the Homegraph API to implement Report State and Request Sync functionality with the Google Assistant.
* [Firebase](https://www.npmjs.com/package/firebase): Used to interact with our Firebase services. See Firebase NodeJS Setup for more information.
* [JSON Web Token](https://www.npmjs.com/package/jsonwebtoken): Used to create, sign, and verify JWTs used to verify the user for account linking and fulfillment with the Google Assistant. See the OAuth and Account Linking section for more information.
* [Firebase UI React Components](https://www.npmjs.com/package/react-firebaseui): Used with the account linking endpoint. Provides a styled “Sign in with Google” button that displays a configurable popup for the user to sign in with their Google account. This button also automatically links with Firebase Authentication, creating a new user upon a successful sign in.
* [Chai](https://www.npmjs.com/package/chai): Used as the assertion library for JavaScript unit tests.
* [Sinon](https://www.npmjs.com/package/sinon): Used as the mocking framework for JavaScript unit tests.

#### Embedded

The following third-party libraries were used by the Embedded team:

* [ArduinoJson](https://arduinojson.org/): In Arduino IDE to serialize and deserialize JSON messages.
* [EEPROM:](https://docs.arduino.cc/learn/built-in-libraries/eeprom) In Arduino IDE to write to ESP32 memory.
* [ESP32\_BLE\_Arduino:](https://github.com/nkolban/ESP32_BLE_Arduino) In Arduino IDE to use BLE.

#### App

The following third-party libraries were used by the App team:

* [cupertino\_icons](https://pub.dev/packages/cupertino_icons): For Google Sign-In and other icons.
* [google\_sign\_in](https://pub.dev/packages/google_sign_in): Enables Google Sign-In popup.
* [googleapis](https://pub.dev/packages/googleapis): Supporting Package for Google Sign-In and Firebase Packages.
* [connectivity\_plus](https://pub.dev/packages/connectivity_plus): Used for toaster connection via bluetooth.
* [network\_info\_plus](https://pub.dev/packages/network_info_plus): Used to retreive connected network info to send to toaster.
* [firebase\_core](https://pub.dev/packages/firebase_core): Core dependency for Firebase\_Auth.
* [firebase\_auth](https://pub.dev/packages/firebase_auth): Uses Firebase Authentication to sign in a user as associate their smart device with them
* [cloud\_firestore](https://pub.dev/packages/cloud_firestore): Dependency for Cloud\_Functions
* [font\_awesome\_flutter](https://pub.dev/packages/font_awesome_flutter): [TODO]
* provider: Allows for providers that can transfer information from view to view.
* [flutter\_blue](https://pub.dev/packages/flutter_blue): Bluetooth package for flutter.
* [permission\_handler](https://pub.dev/packages/permission_handler): [TODO]
* [cloud\_functions](https://pub.dev/packages/cloud_functions): Allows for calling Firebase Cloud Functions.
* [http](https://pub.dev/packages/http): Allows communication with HTTP endpoints.
* [json\_annotation](https://pub.dev/packages/json_annotation): Generates JSON Serializable classes (.dart & .g.dart files)

## Standard Tasks

### 2.2.1 Connection Certification: Root Certificate

As cybersecurity protocols recommend, Google’s root certificates expire and are updated every so often. Their public announcement in the winter of 2021 (found [here](https://security.googleblog.com/2021/03/google-https-and-device-compatibility.html)) states that their current certificate will expire in on January 28th of 2028. It is uncertain whether the update at that time will cause an issue with the ESP32’s connection to Google Cloud IoT Core, as Google mentions that users should not notice changes due to them handling the carryover on their end. However, this project is not an official Google product so there may be action required to meet their eventual encryption update. The root certificate is placed in the ‘ciotc\_config’ file in the ESP32 code.

# Cloud Details

## General Setup

The following sections describe in detail the steps to follow to setup a new IoT device or to maintain our existing setup. In general, the following steps need to be taken. First, a new project must be created in the Google Cloud Platform (GCP). Then, within that project, IoT Core must be setup with a new registry to store a collection of devices. Then, if using the Google Assistant, a new project can be created in Google Actions Console and linked to the original GCP project. Likewise, a Firebase project must be created and linked to the original GCP project, which will manage user and device data, as well as communication with devices. Next, the Firestore database must be setup to store user and device data. Additionally, a local JavaScript environment with NodeJS can now be setup for developing the necessary cloud functions. If using the Google Assistant, Firebase Authentication and Hosting can also be used to host the account linking webapp.

## Pub/Sub Setup

### Introduction

The Publish/Subscribe model is very useful for exchanging messages between applications or devices quickly and reliably. A more in-depth look can be found [here](https://cloud.google.com/pubsub/docs/overview). In general, Pub/Sub works by having producers of data publish a message containing data to a Topic. A subscriber can then receive and consume the messages sent to that topic by creating a Subscription tied to it.

### Setup

A minimum of one topic is required for a Registry (section 3.3.2 just below). You can create this topic ahead of time in Pub/Sub or later when creating the registry. Two topics are recommended however, one for sending normal telemetry messages and one for state and configuration messages. All necessary topics and subscriptions are already created and setup in this project.

## 3.3 IoT Core Setup

### 3.3.1 Introduction

The IoT Core section of Google Cloud Platform is where devices, registries, and gateways are created and monitored. For the scope of this project the gateways are not utilized but are available for handling communication of low-powered devices.

Graphical user interface, application

Description automatically generated

The *Registry details* section is where the configuration and details of the registry which manages pub/sub communication lives, the *Devices* section is where devices can be created and managed, and the *Monitoring* section is where various metrics are visualized to track the health of the devices and communications system.

### 3.3.2 Registry Setup

Each registry in IoT Core is required to have a default telemetry topic. This is the topic that telemetry will be sent to if no subfolder is specified, or if the specified subfolder does not exist. A default topic for receiving state updates can also be set but is not required. To do this, navigate to the **Registry details** page in IoT Core and then select **EDIT REGISTRY** fromalong the top toolbar. Once you’ve done that you will see the options pictured below.

Graphical user interface

Description automatically generated

Clicking the “SHOW ADVANCED OPTIONS” text next to the down arrow at the bottom of this page will present the option to set a device state topic. Another useful area of the registry can also be found on this page, the **Additional topics** section.

If your project has different types of information required to be sent and you want them to be sent to separate topics, you can specify subfolders in the registry. Your device will specify a subfolder in its publishTelemetry function and route the data to whichever topic is connected to that subfolder. If the subfolder does not exist, it will be routed to the default telemetry topic.

To add subfolders in the registry you must first link additional topics to the registry. You can create a new topic either in the **Pub/Sub** of the Google Cloud Platform or create it when adding a subfolder. Click the **Add** button. This will prompt you to add an additional topic to the registry, and you can create a new topic here or select an existing one. You then name a subfolder to link to that topic (subfolder names must be unique: a topic can have multiple subfolders pointing to it, but a subfolder can only point to a single topic) and click **Update** at the bottom of the page. Now, in your device code, you can pass the name of the subfolder to the publishTelemetry function, and the data will be passed to the topic connected to that subfolder rather than the default.

### 3.3.3 Devices

In the Devices section of IoT Core, you can see all the devices that are connected to each registry. The base page will show you the list of devices and their Device IDs, if communication is allowed, the last time that a connection to IoT Core was made by the device, and how it manages its cloud logging.

Graphical user interface, application

Description automatically generated

When creating a new device, all that needs to be supplied is a Device ID, and optional (but recommended) authentication. The authentication consists of a public/private key pair (more information about key pairs found [here](https://www.ibm.com/docs/en/ztpf/1.1.0.14?topic=concepts-public-key-cryptography)), which will need to be made ahead of time. Information on how to generate key pairs can be found by Google at [this link](https://cloud.google.com/iot/docs/how-tos/credentials/keys) and can be done in Google Cloud page on a browser through their Cloud Shell, or in the command line of a system with OpenSSL installed. Back in IoT Core, the device will need to have the public key provided, and the paired private key will be stored on the physical device.

Many useful things can be seen and done within a specific device on this page. If you select a specific device you will be greeted by the following page…

**Graphical user interface, text, application, email

Description automatically generated**

Here we can see some useful details about the device, and the Authentication page will allow you to update or add a new/additional key authentication to the device. Perhaps the most useful tab though is the Configuration & State section. Clicking into this tab will allow you to see both the configuration history and the state history (each can be toggled on or off). Config updates are messages sent to the physical device, and state updates are messages sent from the device to the cloud.

Graphical user interface, text, application, email

Description automatically generated

You are also able to use the **Update Config** and **Send Command** tools found along the upper toolbar on the device page. The update config option will allow you to send a message to the device from the browser. These are asynchronous messages that will be delivered at least once and can only be sent once a second. If the device is offline when the message is sent, it will be held and delivered once the device comes online. The commands work like normal telemetry messages, meaning that they will be dropped if the device isn’t online when sent, and can handle up to 100 messages a second. These are useful for testing and troubleshooting devices.

## 3.4 Google Actions Setup

### 3.4.1 Introduction

The Google Actions Console is where settings related to Google Assistant support can be configured. At a glance, this is the location to update the invocation name of the device, setup the account linking process, and configure the fulfillment endpoint. The console also offers a testing suite for the device. To begin, select the Spectrum-Brands-Main-Project. This project should be automatically linked to the primary project hosted on the Google Cloud Platform.

A picture containing graphical user interface

Description automatically generated

### 3.4.2 Quick Setup

The Quick Setup portion of the project should already be completed with the necessary information to get started with Google Assistant support.

Background pattern

Description automatically generated with low confidence

First, the name of the Action can be specified. This is the name that users will speak to inform their Assistant to modify their smart toaster oven once it is connected. By default, this is “Spectrum Brands Smart Toaster Oven” however this can be changed as necessary.

Graphical user interface, text, application, email

Description automatically generated

This is also where you can set the necessary information for account linking. This process is described further in the OAuth documentation, but at a high level, these parameters allow the user to link their Google account with their Assistant, which authorizes it to interact with their smart toaster oven. The Client ID and Client Secret are strings specified by you and are used in the account linking endpoint and token endpoint to verify that a request to the endpoint is authentic. The authorization URL is the link to the account linking endpoint hosted with Firebase Hosting. This is a simple webapp that lets users log in with their Google Account to confirm that they wish to link their account. If they do, they are automatically redirected to the token endpoint. The token endpoint is hosted on Cloud Functions. See the OAuth and Account Linking documentation for more information.

Graphical user interface, text, application, email

Description automatically generated

### 3.4.3 Build Your Action

The Build Your Action portion of the project includes the preconfigured link to the fulfillment endpoint. This is the entry point of all the code that executes in response to user input to the Google Assistant.

Background pattern

Description automatically generated with low confidence

The fulfillment URL is hosted by Cloud Functions on the Google Cloud Platform.

Graphical user interface, text, application, email

Description automatically generated

Google Actions also provides a test suite for the device. See [here](https://developers.google.com/assistant/smarthome/develop/testing) for more information.

## 3.5 Firestore Database

### 3.5.1 Creating a Firebase Project

If you have an existing Firebase project, continue to step 2.

On <https://console.firebase.google.com> click add a project.

Graphical user interface, application

Description automatically generated

Then enter a project name, or select an existing Google Cloud Platform project, if you have one.

Google analytics should be enabled, and pick the default account for Firebase.  
Graphical user interface, text, application, email

Description automatically generated

### 3.5.2 Creating a Firestore Database

If you already have Firestore Database setup, skip to step 3.

Go to “Firestore Database”, then click “Create database”

Graphical user interface

Description automatically generated

Leave the default settings, production mode and with the region closest to you.

Now we see the Firestore Database!

Background pattern

Description automatically generated

### 3.5.3 Firestore Documents & Collections

If you want to skip to the document structure we use in the smart toaster project, go to step 4.

In Firestore, data is stored in documents, which are contained in collections. A document can contain collections as well.

Data is schema-less in Firestore, which means it is easy to add/remove data. And you cannot exactly enforce that certain properties are present.

To demonstrate collections and documents, we will create some sample data in the context of a chat application.

First, create a users collection, and a document with an automatic ID. And give it some properties, for example “Name=Max”

Graphical user interface, application, website

Description automatically generated

It should initially look like this (with a different ID)Graphical user interface, application

Description automatically generated

Next, create a new collection called “rooms”, with a blank document with an auto generated ID, and a name property of “Room 1”

It should initially look like this (with a different ID)

Graphical user interface, application

Description automatically generated

Next, create a sub collection in Room 1, called “messages”. The initial document will again have an auto generated id, and two properties:

* “text” that has the text of a message, “My first message!”
* “sender” that has the ID of the user we created earlier, “cACSxuj4dswglOVnw4Rb” for me.

This is what it should look like now

Graphical user interface, application

Description automatically generated

We have a “users” collection, that contains different users. A “rooms” collection, that contains chat rooms. And each room has a “messages” collection that contains all messages in the room. Each message has a sender, and the text of the message.

### 3.5.4 Smart Toaster Oven Structure in Firestore

The Firestore database for the Smart Toaster project has four main collections. Three collections contain information about the devices, “deviceConfiguration”, “deviceProperties”, and “deviceState”. Each device has a document in each collection that shares the device’s id.

The device configuration collection stores the ID of the device’s owner, and the device’s current configuration, which is set by either the Google home/Google actions or our custom mobile app. It is used for outgoing requests to the device. Configuration includes the target temperature, is the toaster cooking, what food/recipe to use, etc. The device properties collection stores metadata about the device, like its name, location, type, and the ID of the device’s owner. The device state collection stores the ID of the device’s owner, and the current state of the device, including if the device is online. The device’s state is determined by what our toaster sends to IoT Core and is used for incoming communication. State includes information like current temperature, heating, fan, time remaining on a timer, etc.

The last collection is a collection of users, where each document is the user’s ID. This collection stores information about the user, such as an optional refresh token, which is used to access their devices via Google Assistant if the account linking process has been completed.

## 3.6 Firebase NodeJS Setup

### 3.6.1 Introduction

In the creation of a Google Smart Device, there is a necessity to host multiple aspects in the cloud. These include our functions for transferring data between device and the cloud, along with our authentication web app for connecting users to their devices. The solution to this we chose was to use Google’s Firebase. This is likely the easiest to set up hosting method; however, with extended use, charges may occur. In this document, we will go over how to set up Firebase for a Google Cloud project utilizing Node.js.

### 3.6.2 Prerequisites

* Node.js installed: <https://nodejs.org/en/download/>
* Existing Google Cloud Project: <https://console.cloud.google.com/cloud-resource-manager>

### 3.6.3 Firebase Setup

If you already have Firebase set up for your project, you should still go through this section to ensure all steps are completed. To start, go to [https://Firebase.google.com/](https://firebase.google.com/) and select Get Started.

Graphical user interface, application, Word

Description automatically generated

Select Add Project and chose your Google Cloud project you just created. Your project should be visible in a dropdown in the Enter your project name text field. Select continue.

Graphical user interface, text, application, chat or text message

Description automatically generated

When prompted with Enable Google Analytics, it is your choice to choose this or not.

A picture containing text

Description automatically generated

Select Add Firebase and wait a few seconds for it to load. This should open the Firebase dashboard for your project.

Graphical user interface, application, website

Description automatically generated

The first step in the dashboard is to set up a payment plan. To utilize Firebase’s function hosting, you must upgrade from the free version to the Blaze Pay as you go plan. There will initially be no cost, but with extend use, charges may occur. Select the Spark plan button and add payment info for the Blaze plan.



Graphical user interface, text, application

Description automatically generated

Once payment has been set up, we must set up individual parts of the Firebase. For initial set up with Node.js, we need only set up Firestore Database and Functions, but other aspects like Authentication and Hosting will likely be needed for the creation of a Google Smart Device.

Select the Firestore Database tab on the right.

Graphical user interface, application

Description automatically generated

Select Create Database and chose start in production mode.

Graphical user interface, text, application, chat or text message

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Select your proper location for server hosting (likely US-central) and enable Firestore.

Graphical user interface, text, application, email

Description automatically generated

Next, select Functions tab on the right.

Graphical user interface, application

Description automatically generated

Select Get Started and chose Finish in set up menu.

Graphical user interface, text, application, chat or text message

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Firebase should now be set up, and we can begin work with Node.js.

### 3.6.4 Firebase Setup with NodeJS

To begin, ensure you have Node.js installed following the link in the prerequisites section.

Open the Node.js command prompt and navigate to an empty folder you wish to set up your project in.

Text

Description automatically generated

Once in this directory, we should initialize Firebase. First, you must log into your Google Account. Enter command *firebase login*. Select yes or no to Firebase collection usage data, and then you will be redirected to a Google login web page on a browser. Login and be sure to select allow on the final screen.

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Now being logged in, we will go through the setup process for the project in this directory. First, enter the command *firebase init*. Confirm the directory and select ready to proceed.

A screenshot of a computer screen

Description automatically generated with medium confidence

Next, you will be asked which features you wish to set up. Likely you will not need all features, but in this example, we will confirm the use of all. Select <a> to toggle all and <enter> to proceed.

A picture containing graphical user interface

Description automatically generated

Select use an existing project and select your Google Cloud project.





Select yes for initializing the Realtime Database.



As we did earlier, select the same location for where the database will be hosted (likely us-central).



For all generated files, select default names.

Text

Description automatically generated

For setting up functions, we chose to use JavaScript, so other tutorial will use JavaScript. However, if you chose to use TypeScript, select that instead here.

Text

Description automatically generated

It is recommended you use ESLint to catch errors when deploying your code to Firebase.



Select yes to install npm dependencies.



Select the default public directory.

Text

Description automatically generated

Select no for single-page app configuration.



The next step is to set up automatic builds and deploys through git. We recommend setting this up, as you can automatically deploy your functions on all pushes that change their code. However, we are going to skip this in this tutorial.



Select default name of storage rules file.

Text

Description automatically generated

The next step is emulator set up. Emulators are useful for debugging various aspects of Firebase. It is recommended initializing all emulators correlating to all features of Firebase you are using. These can be initialized later, as well.

Text

Description automatically generated

Use all default ports for the emulators.

Text

Description automatically generated

Lastly, select default name for remote config file, and Firebase will be successfully initialized.

Text

Description automatically generated

The last command we need to run for our set up within this directory *is npm install -g Firebase-tools*. This will install dependencies needed for deployment of code within this directory.

At this point, your directory should have a similar structure to these 2 images. Within the functions folder is where the creation of your deployable functions will take place.

Table

Description automatically generated

Within functions directory:

Text

Description automatically generated

## 3.7 Cloud Functions

### 3.7.1 Introduction

All functions are created in JavaScript and hosted on Cloud Functions on the Google Cloud Platform. There are a couple of different function types. The HTTP functions use a HTTP endpoint to call them and perform their respective duties. The Pub/Sub functions utilize the topics setup in IoT Core to receive or send data between a physical device and the Firestore database. Document functions are created to watch Firestore documents and perform an action based on the events specified.

Functions must be created within the index.js file found within the functions directory. To keep index.js clean, we export our functions to multiple different files, one for each function. To view our code for the functions used on the Toaster Oven, in our repository, go to Cloud\_Code\Firebase\functions and view index.js. In this file, you will see imports for each function that show which directory each function’s file is in.

### 3.7.2 Functions

#### Create Device Function

The create Device function is an HTTP function that is called when a new toaster oven is made within the Cloud system. The function grabs the device Id off the device and creates a private and public key pair. When the key pair is created, it will try to create a device using the device Id and public key. The new device is than stored in Firestore.

#### Register Device Function

The register Device function is an HTTP function that is called when a new toaster oven is being registered with a new user. The function will take the device Id off the toaster along with the public key and check to make sure the device exists within the system. If the device does exist, it will attempt to assign the device to the user in cloud system.

#### Updating Firestore

The device state function is a Pub/Sub topic that handles device state updates. It watches IoT Core to check if there are any incoming device states. If there is, it will update the Firestore database to reflect the changes received from the device.

#### Relaying the Device State

If any changes are made to the device configuration within the Firestore data base, on device update function will relay the information to IoT Core. Once IoT Core receives this update it will push it to the device.

#### Checking Online State

The function online state function checks if the respective device is responsive. If the device is in a CONNECT state it will be updated as an online state in the Firestore database otherwise if it is in a DISCONNECT state it will be updated as an offline state in the Firestore database.

#### Fulfillment

See the Fulfillment documentation.

#### Sync and Report State

Untested functions that implement the required request sync and report state functionality for the Google Assistant. See [here](https://developers.google.com/assistant/smarthome/develop/request-sync) and [here](https://developers.google.com/assistant/smarthome/develop/report-state) for more information.

#### Token

See the OAuth and Account Linking documentation.

#### Device Getters

There are several helper functions that exist to facilitate communication between the mobile app and Firestore. These include getDeviceState, getDeviceConfig, and getDeviceProperties.

## 3.8 Fulfillment

### 3.8.1 Introduction

Fulfillment is used in the intents for a Google Smart Device. It will process a request based on the action that is taken upon it, such as a SYNC request. Fulfillment is necessary for the Google Assistant to control the user’s device. It does this by modifying specific traits updating our Firestore database accordingly. The traits are well defined in the Google documentation and the main limiting factor to the functionalities the toaster can perform through Google Assistant.

### 3.8.2 Functionality

The functionality of our fulfillment consists of a lot of moving parts. Files (fulfillment-contoller.js, device-factory.js, device-helper.js, firestore-helper.js, device-model.js, toaster.js, execute.js, query.js, sync.js, fulfillment.js). The main premise is to get accessed through fulfillment.js. Then firestore-helper.js and fulfillment-controller.js does a lot of heavy lifting. It is also split up in this fashion to allow for viable all-around testing.

### 3.8.3 Intents

There are several intents that google assistant recognizes and uses for controlling the toaster. These intents are SYNC, QUERY, EXECUTE, and DISCONNECT. Additionally, the google assistant uses an access token from our firebase to authorize the user. If the user provides an invalid bearer token, fulfillment will respond with a 401 unauthorized error.

#### SYNC

The SYNC request responds with the all the devices the user owns and their capabilities. More details can be found in the [Google SYNC documentation here.](https://developers.google.com/assistant/smarthome/reference/intent/sync)

#### QUERY

The QUERY intent requests the current state of all the devices provided to fulfillment. Fulfillment will then respond with the current state of the device including if the device is online and reachable. More details can be found in the [Google QUERY documentation here.](https://developers.google.com/assistant/smarthome/reference/intent/query)

#### EXECUTE

The EXECUTE intent sends commands to the device as defined in the individual traits of the devices. To add additional commands additional traits must be added to device. [A list of possible traits can be found here](https://developers.google.com/assistant/smarthome/traits). The google assistant will then send the execution commands as defined in the [Google EXECUTE documentation here.](https://developers.google.com/assistant/smarthome/reference/intent/execute)

#### DISCONNECT

The DISCONNECT intent is used to remove a users device from the google assistant. Additional information can be found in the [Google DISCONNECT documentation here.](https://developers.google.com/assistant/smarthome/reference/intent/disconnect)

## 3.9 Function Deployment

After completing one or any number of functions, the next step is to deploy these so Firebase can host them. This is a very simple process that can be repeated. Any time you wish to deploy a new function or make a change to a function, you must make sure you are also deploying all currently existing functions, or they may be deleted from Firebase. This just means you would have to deploy those functions again.

Once your functions are ready for deployment, in Node.js command prompt, enter the file where your index.js file exists, likely your functions folder.



Text

Description automatically generated

Likely you already have, but in case you have not done so yet, be sure to install Firebase tools. *npm install -g Firebase-tools*





Lastly, to deploy, simply run the command *Firebase deploy --only functions*

Text

Description automatically generated

Before deploying, first ESLint will check for errors and report them to you in the command prompt, stopping the deploy. You will need to fix these issues before attempting to redeploy, running the same command as before.

If your functions should deploy successfully, you should now be able to view them within your Firebase Functions tab. The image below should the functions page on Firebase along with a few of the functions deployed for the Toaster Oven.

Graphical user interface, text, application

Description automatically generated

## 3.10 Function to IoT Core

For Firestore database updates to be sent to IoT core, a cloud function must be setup to listen for updates at some place in the Firestore database. Cloud functions that monitor the Firestore database can listen for different event types.

Event Types:

* onCreate – triggered when a document is written to for the first time
* onUpdate – triggered when an existing document has any value changed
* onDelete – triggered when a document with data is deleted
* onWrite – triggered when a document is created, updated, or deleted

Along with an event type for the cloud function to list to, it also needs a specified document or collection to listen for events at. This is called the document path and is formatted as “CollectionName/Document”. It’s important to note that functions only respond to document changes and cannot monitor specific fields or collections on a document. However, document paths can take advantage of wildcards for identifying document paths. For example,

* deviceState/Embedded\_Demo : monitors a specific document, Embedded\_Demo
* deviceState/{deviceId} : monitors all documents in the deviceState collection

## 3.11 IoT Core to Firestore

The following process is used for incoming data sent from the device to IoT Core. To communicate with the Firestore database, a Pub/Sub topic must be established in the IoT Core registry setup for the toaster oven. Whenever the device detects a state change, it sends that data to the ESP32, which in turn, uses the MQTT protocol to send the device’s current state to IoT Core as a JSON message. IoT Core will automatically redirect the message to the default Pub/Sub topic, or a specific one if specified on the ESP32. Next, the Cloud Function “deviceState” will detect the newly published data in the specified Pub/Sub topic and execute. This function will parse the published data and use it to modify the Firestore database accordingly.

## 3.12 IoT Core to Device

The following process is used for outgoing data sent from the cloud to the device. For IoT Core to communicate to the device, the device must be subscribed to a configuration topic. For the toaster oven, the MQTT protocol was used to communicate to the device. This was done by sending the device a JSON message. Whenever IoT Core receives a new configuration update or command, it pushes the JSON object to the device. The device never receives an older configuration update than the most recent version. Example configuration data can be seen below.

|  |
| --- |
| **Example config data:**  {“command”: “GREENLED”} |

## 3.13 OAuth and Account Linking

### 3.13.1 Introduction

The OAuth server implemented for this project allows users to link their Google account with Google, which enables them to use their Google Assistant to interact with their Spectrum Brands smart toaster oven. A valid account link must be established before use of a Google Assistant is permitted, because it officially authorizes Google to send commands to the user’s device on their behalf. The process is comprised of two primary parts: the authorization endpoint and the token exchange endpoint.

### 3.13.2 The Authorization Endpoint

The first step of the account linking process is completed at the authorization endpoint. This is a webpage, hosted using Firebase Hosting, where the user can sign into their Google account and confirm if they wish to link their account with Google. If the user begins the account linking process through their Google Assistant, the webpage will be opened automatically on their phone for them. Google will also include a few parameters in the HTTP request to the endpoint.

Graphical user interface, website

Description automatically generated

*Initial screen of the webpage used for the authorization endpoint.*

Google provides a series of recommended design guidelines for the authorization endpoint to follow. First, Google’s Privacy Policy should be displayed if users would like to read through it before linking. Next, a clear call to action must be displayed, for example, an “Agree and link” button. Lastly, the user must have the ability to either cancel the account linking process, or, if they have already linked their account, to unlink their account. The “Cancel” button will navigate the user to the homepage, prompting them to sign in again to continue. Otherwise, the user may unlink their account by following the provided link to their Google account settings, where they can unlink from there.

Graphical user interface, text

Description automatically generated

*Confirmation screen of the authorization endpoint.*

Recall that Google automatically sends some URL search parameters in the HTTP request to the authorization endpoint. Upon successfully linking a user account, the OAuth server uses these parameters to continue the authorization process. Specifically, it must do the following:

* Ensure that *client\_id* matches the Client ID specified on the Google Actions Console.
* Ensure that *redirect\_uri* is from a verified source, as detailed in Additional Resource 1.
* Ensure that *response\_type* is “code”.

|  |
| --- |
| **Example HTTP request:**  GET https://authserver.com/  ?client\_id=GOOGLE\_CLIENT\_ID  &redirect\_uri=REDIRECT\_URI  &state=STATE\_STRING  &response\_type=code |

Once these conditions are met, the authorization endpoint will generate an authorization code. This is used in the next step of the OAuth process, which utilizes the authorization code flow. In this OAuth server, the authorization code is the user’s ID Token from Firebase Authentication. This is determined by Firebase, and both uniquely identifies the user and can retrieve their user ID.

Finally, after this is done, the authorization endpoint will automatically redirect the user to the token exchange endpoint. Starting with the *redirect\_uri* URL search parameter provided in the initial HTTP request, append the newly generated authorization code and the *state* URL search parameter from the initial HTTP request. Send the user to this URL. At this point, the authorization endpoint is complete and is now in the token exchange part of the full authorization process.

|  |
| --- |
| **Example HTTP response:**    https://redirect.com/r/PROJECT\_ID  ?code=AUTHORIZATION\_CODE  &state=STATE\_STRING |

### 3.13.3 The Token Exchange Endpoint

The token exchange endpoint is the final step of the authorization process and is comprised of two distinct flows itself. The first is the authorization code flow, which exchanges an authorization code for a refresh token and an access token and is only called when the user is redirected to this endpoint after agreeing to link their account. The second is the refresh token flow, which is called in subsequent attempts to authorize the user so long as their account remains linked. It exchanges a refresh token for an access token.

#### Authorization Code Flow

Recall that the authorization endpoint was responsible for generating an authorization code after the user linked their account. This code is short-lived and should not be used to authorize the Google Assistant for subsequent requests. Therefore, it must be exchanged for a persistent refresh token. This refresh token can then be used to generate access tokens which authorize the Google Assistant. Upon receiving an HTTP request from Google, the process is as follows:

* Ensure that both the *client\_id* and *client\_secret* parameters match the Client ID and Client Secret, respectively, specified on the Google Actions Console.
* Ensure that the *code* is valid.

|  |
| --- |
| **Example HTTP request:**    POST /token HTTP/1.1  client\_id=GOOGLE\_CLIENT\_ID  &client\_secret=GOOGLE\_CLIENT\_SECRET  &grant\_type=authorization\_code  &code=AUTHORIZATION\_CODE |

If these conditions cannot be satisfied, the endpoint must return an HTTP 400 response along with the JSON {“error”: “invalid\_grant”}. Otherwise, a refresh token and access token can be successfully generated. Google recommends that refresh tokens should be any string value that uniquely identifies the user and should not be guessable. Therefore, our OAuth server uses JSON Web Tokens (JWTs) as the refresh and access tokens.

Per the JWT guidelines, we utilize the subject (sub) and audience (aud) fields of a JSON object to represent key values necessary for refresh token validation. Namely, the subject is the user ID of the user in Firebase Authentication, which can be retrieved from the authorization code included in the request. This was generated in the previous endpoint, which is the user’s ID token from Firebase Authentication, and is why we can retrieve the corresponding user ID. The audience is the aforementioned Client ID, so we can verify the origin of a refresh token request in the other flow, described below.

After signing the JWT using a predetermined JWT Secret, we have generated the user’s refresh token. We can then use this new token to generate an access token right away as well. It is also important to update the user’s Firestore record with their new refresh token, so that we can verify it when receiving new refresh token requests.

Finally, we generate the HTTP response. It must contain a *token\_type* of “Bearer”, the refresh token, the access token, and an expiration time of the access token (1 hour). It is sent back to Google with the status code HTTP 200.

|  |
| --- |
| **Example HTTP response:**  {  "token\_type": "Bearer",  "access\_token": "ACCESS\_TOKEN",  "refresh\_token": "REFRESH\_TOKEN",  "expires\_in": 3600  } |

#### Refresh Token Flow

The process for exchanging a refresh token is similar but includes additional checks to ensure that the refresh token in the request and the user’s Firestore database match, and that the audience in the refresh token and Client ID match. Just like with the authorization code flow, upon receiving an HTTP request from Google, the OAuth server ensures that both the *client\_id* and *client\_secret* parameters match the Client ID and Client Secret, respectively, specified on the Google Actions Console, otherwise an invalid grant error is returned with code HTTP 400.

|  |
| --- |
| **Example HTTP request:**    POST /token HTTP/1.1    client\_id=GOOGLE\_CLIENT\_ID  &client\_secret=GOOGLE\_CLIENT\_SECRET  &grant\_type=refresh\_token  &refresh\_token=REFRESH\_TOKEN |

At this point, the server retrieves the refresh token from the corresponding URL search parameter. It then decodes it to retrieve the audience and subject previously encoded in the authorization code flow. The server then verifies that the audience matches the Client ID specified in the Google Actions Console. It also gets the corresponding user document from Firestore by using the subject, also set in the authorization code flow. If the refresh token from the HTTP request and the refresh token stored in Firestore match, then the server can generate an access token.

This time, the response must contain a *token\_type* of “Bearer”, the access token, and an expiration time of the access token (1 hour). It is sent back to Google with the status code HTTP 200.

|  |
| --- |
| **Example HTTP response:**  {  "token\_type": "Bearer",  "access\_token": "ACCESS\_TOKEN",  "expires\_in": 3600  } |

### 3.13.4 Additional Resources

1. [Google Smart Home Docs - Implement Account Linking](https://developers.google.com/assistant/smarthome/develop/implement-oauth)
2. [JWT - Introduction](https://jwt.io/introduction)
3. [jsonwebtoken Node.js Library - GitHub](https://github.com/auth0/node-jsonwebtoken)
4. [Verifying ID Tokens - Firebase Authentication](https://firebase.google.com/docs/auth/admin/verify-id-tokens)

# Embedded Details

## ESP32 Documentation

The ESP32 is the access point between the toaster and the cloud. Using Arduino, the team created a script that the ESP32 follows. The toaster will receive various data from the cloud and vice versa. The ESP32 will receive a message from IoT Core in the form of a JSON which will be deserialized using the ArduinoJson library. This will be sent to the Nuvoton in the format of  

**"Command tempToSet TimeToCook CookingMode \n”.**

Each parameter is separated by a space.

Whenever the ESP32 recieves data from the Nuvoton, it will receive it in the format of   
“C:\_:\_:\_:\_:\_:\_:: *the sum of the 6 spaces*”.

The three spaces are typically timeRemaining, currentOvenTemp, ovenCookingMode, ovenSetTemp, isOvenOn, and IsOvenStarted in that order. The sum is used to make sure that the entire protocol was received correctly.

A Bluetooth Low Energy (BLE) server is established on the ESP32 and is used to communicate between the app and ESP32. The app writes the SSID of the WIFI network that the phone is using, and a network password that the user enters to a characteristic on the BLE server, which the ESP32 can then parse and use to obtain the data. Once the ESP32 obtains the WIFI information, it can use this to connect to WIFI. In this way, the ESP32 can connect to different or changing WIFI networks.

#### Additional Resources

1. [Arduino JSON](https://arduinojson.org/)

## UART Changes

The toaster oven code given by Spectrum Brand was originally intended to work with I2C. This would allow for two-way communication using only one UART port for debugging. Halfway through implementation, the team will use UART as the communication between the Nuvoton and the ESP32 as well as the communication between the Nuvoton and the command parser. The biggest problem with enacting this plan was that the code that was given by Spectrum Brand was based around I2C and the team had to implement a 2nd UART line so that the Nuvoton can communicate with both the ESP32 and the command parser.     
   
All the files that are listed here have a very simple change. Each file now either calls UART0 or UART1. UART0 is used in tandem with the command parser for debugging purposes. UART1 on the other hand will be used by the CloudCommandParser and Com to communicate with the ESP32.

Files:

CommandParser.c

CommandParser.h

BSP

BSPConfig

* UART0 uses pins MFP\_P30 and MFP\_P31
* UART1 uses pins MFP\_P02 and MFP\_P03

If the need for more UART lines arises, the process of adding new lines is not difficult. The BSPConfig file have the all the pins the board has access to.  Here, a set of pins can be dedicated as a line for a new UART line.

## Com Module Changes

### 4.3.1 Changes

Since we changed our communication protocol from I2C to UART, a couple of changes were needed to utilize the UART connection. Firstly, Com Module was only responsible for reading from UART1 which was the connection between the ESP32 and the Nuvoton Board.

The original state machine used to support an I2C connection has been reduced significantly. Specifically, we have 4 other states beside a basic initialization state.

Text

Description automatically generated

Figure 1: Com States

COM\_STATE\_IDLE is responsible for both determining if a dirty flag has been sent for the input/output of data and setting a dirty flag if a timer is elapsed. Every time the com process is called, it switches between the COM\_ID that it uses for the action it will take.

Text

Description automatically generated

Figure 2: Different COM ID’s

If a dirty flag is found on the corresponding COM\_ID that the Com process is using, the state machine will transition to next state COM\_STATE\_TRANSACTION\_START. This state will clear the buffers that are use to write/read data. Then based on the COM\_ID, the state machine will either enter the COM\_STATE\_RX\_PROCESS or COM\_STATE\_TX\_PROCESS. RX\_PROCESS will be responsible for reading in the data and parsing it into the ReadDataStruct struct. TX process will be responsible for sending data located within the WriteDataStruct struct. For both TX\_PROCESS and RX\_PROCESS, after they are completed, they will transition back to COM\_STATE\_IDLE.

To manually send a message using the Com Module, ComValueSet must first be called by passing the WriteDataStruct loaded with the data along with relevant COM\_ID. After the data is loaded in, the user must call ComSend which is responsible for setting the dirty flag identifying that the data is ready to be read from.

## 4.4 CloudCommandParser

The CloudCommandParser is heavily based off the command parser provided to us from spectrum brands. A few changes have occurred specifically.

Cloud Command Parser is responsible for reading data from the Com module using the ComValueGet function to obtain a ReadDataStruct loaded with relevant information. The Cloud Command Parser utilizes this data to determine if a certain command has been called or a cooking mode has been set. Any parameters needed for certain functionalities can be pulled from the previously mentioned ReadDataStruct.

CloudCommandParserProcess is responsible for determining if a dirty flag has been set by the com module for the COM\_READ\_DATA COM ID.

The parsing of the UART RX was originally handled within the Cloud Command Parser but was later moved to the com module to reduce the dependency on the communication protocol used by the Nuvoton board.

## 4.5 Control State Machine

The control state machine is in charge of interpreting the parsed commands and executing toaster functions accordingly. This module will turn on the oven when a cooking command is received and carry out any other functions directly related to cooking, such as the timer or fan control.

## 4.6 Wi-Fi State Machine

The Wi-Fi state machine is responsible for monitoring the status of the toaster oven’s connection to the Wi-Fi via the use of the ESP32 microcontrollers and the use of Bluetooth Low Energy (BLE) code. Some functions of this module include monitoring when the ESP32 has successfully connected to the Wi-Fi, checking when the ESP32 loses connection, or asking for the user’s credentials via SSID and password for the ESP32 to be granted access to the Wi-Fi.

## 4.7 Temperature Reading

The toaster oven has two methods of reading temperature: the probe, which is intended to detect the temperature inside of food, and the NTC, which is intended to read the internal temperature of the oven. To read the temperature, each source gives an ADC (analog-to-digital converter) value, which the program checks against a table in order to get a temperature value. When wiring up the temperature probe, a 4.7kΩ resistor was used. The probe temperature table is as follows:

|  |  |
| --- | --- |
| Probe ADC Value | Temperature in Degrees Fahrenheit |
| 258 | 400.1 |
| 302 | 387.1 |
| 346 | 373.0 |
| 409 | 360.0 |
| 588 | 329.4 |
| 766 | 303.3 |
| 877 | 293.5 |
| 1182 | 262.0 |
| 1376 | 248.4 |
| 1554 | 235.6 |
| 1760 | 220.6 |
| 2010 | 207.0 |
| 2210 | 195.6 |
| 2431 | 182.3 |
| 2650 | 170.4 |
| 2818 | 159.4 |
| 2990 | 149.9 |
| 3175 | 135.3 |
| 3296 | 124.3 |
| 3413 | 113.5 |
| 3650 | 88.5 |
| 3773 | 72.5 |

The NTC temperature table is as follows:

|  |  |
| --- | --- |
| NTC ADC Value | Temperature in Degrees Fahrenheit |
| 2050 | 480.0 |
| 2692 | 377.0 |
| 2737 | 365.0 |
| 2826 | 340.0 |
| 2902 | 318.0 |
| 2949 | 289.0 |
| 2976 | 267.0 |
| 3007 | 251.0 |
| 3028 | 239.0 |
| 3075 | 217.0 |
| 3117 | 191.0 |
| 3122 | 181.0 |
| 3141 | 151.0 |
| 3146 | 141.0 |
| 3157 | 118.0 |
| 3162 | 98.0 |

The NTC temperature table is currently not in its final state, as the ADC value will differ depending on whether the Nuvoton board is plugged into a computer or not. This temperature table was created based on ADC readings from when the Nuvoton board is plugged into a computer, so this temperature table should be recreated with ADC readings from when it’s not plugged in.

## 4.8 Hardware Documentation

### 4.8.1 Pin Changes

|  |  |
| --- | --- |
| Old | New |
| P02 - PIN\_NOT\_USED\_P02 (GPIO, Output, LOW, No Pullup)  #define P02\_OPERATING\_MODE OPERATING\_MODE\_OUTPUT  #define P02\_INITIAL\_VALUE LOW  #define P02\_PULL\_UP\_MODE PULL\_UP\_MODE\_DISABLED | P02 - PIN\_UART2\_RX (RX, Input, LOW, Pullup)  #define P02\_OPERATING\_MODE OPERATING\_MODE\_INPUT  #define P02\_INITIAL\_VALUE LOW  #define P02\_PULL\_UP\_MODE PULL\_UP\_MODE\_ENABLED |
| P03 - PIN\_NOT\_USED\_P03 (GPIO, Output, LOW, No Pullup)  #define P03\_OPERATING\_MODE OPERATING\_MODE\_OUTPUT  #define P03\_INITIAL\_VALUE LOW  #define P03\_PULL\_UP\_MODE PULL\_UP\_MODE\_DISABLED | P03 - PIN\_UART2\_TX (TX, Output, HIGH, No Pullup)  #define P03\_OPERATING\_MODE OPERATING\_MODE\_OUTPUT  #define P03\_INITIAL\_VALUE HIGH  #define P03\_PULL\_UP\_MODE PULL\_UP\_MODE\_DISABLED |
| P05 - PIN\_HEATER\_BOTTOM (GPIO, Output, LOW, No Pullup) | P05 - PIN\_NOT\_USED\_P05 (GPIO, Output, LOW, No Pullup) |
| P06 - PIN\_NOT\_USED\_P06 (GPIO, Output, LOW, No Pullup) | P06 - PIN\_LCD\_SEG0 (GPIO, Output, LOW, No Pullup) |
| P07- PIN\_NOT\_USED\_P07 (GPIO, Output, LOW, No Pullup) | P07 - PIN\_LCD\_SEG1 (GPIO, Output, LOW, No Pullup) |
| P10 - PIN\_BUZZER (GPIO, Output, LOW, No Pullup)  #define P10\_OPERATING\_MODE OPERATING\_MODE\_OUTPUT  #define P10\_INITIAL\_VALUE LOW  #define P10\_PULL\_UP\_MODE PULL\_UP\_MODE\_DISABLED | P10 - PIN\_UART1\_RX (GPIO, Output, LOW, No Pullup)  #define P10\_OPERATING\_MODE OPERATING\_MODE\_INPUT  #define P10\_INITIAL\_VALUE LOW  #define P10\_PULL\_UP\_MODE PULL\_UP\_MODE\_ENABLED |
| P11 - PIN\_EN1\_B (GPIO, Input, LOW, Pullup)  #define P11\_OPERATING\_MODE OPERATING\_MODE\_INPUT  #define P11\_INITIAL\_VALUE LOW  #define P11\_PULL\_UP\_MODE PULL\_UP\_MODE\_ENABLED | P11 - PIN\_UART\_TX (GPIO, Output, LOW, No Pullup)  #define P11\_OPERATING\_MODE OPERATING\_MODE\_OUTPUT  #define P11\_INITIAL\_VALUE HIGH  #define P11\_PULL\_UP\_MODE PULL\_UP\_MODE\_DISABLED |
| P14 - PIN\_SCL (SCL, Input, LOW, No Pullup)  #define P14\_OPERATING\_MODE OPERATING\_MODE\_INPUT | P14 - PIN\_NOT\_USED\_P14 (GPIO, Output, LOW, No Pullup)  #define P14\_OPERATING\_MODE OPERATING\_MODE\_OUTPUT |
| P16 - PIN\_UART\_TX (TX, Output, HIGH, No Pullup)  #define P16\_INITIAL\_VALUE HIGH | P16 - PIN\_NOT\_USED\_P16 (GPIO, Output, LOW, No Pullup)  #define P16\_INITIAL\_VALUE LOW |
| P17 - PIN\_UART\_RX (RX, Input, LOW, Pullup)  #define P17\_OPERATING\_MODE OPERATING\_MODE\_INPUT  #define P17\_PULL\_UP\_MODE PULL\_UP\_MODE\_ENABLED | P17 - PIN\_LCD\_BACKLIGHT (GPIO, Output, LOW, No Pullup)  #define P17\_OPERATING\_MODE OPERATING\_MODE\_OUTPUT  #define P17\_PULL\_UP\_MODE PULL\_UP\_MODE\_DISABLED |
| P20 - PIN\_NOT\_USED\_P20 (GPIO, Output, LOW, No Pullup) | P20 - PIN\_LCD\_SEG4 (GPIO, Output, LOW, No Pullup) |
| P21 - PIN\_NOT\_USED\_P21 (GPIO, Output, LOW, No Pullup) | P21 - PIN\_LCD\_SEG5 (GPIO, Output, LOW, No Pullup) |
| P22 - PIN\_NOT\_USED\_P22 (GPIO, Output, LOW, No Pullup) | P22 - PIN\_LCD\_COM3 (GPIO, Output, LOW, No Pullup) |
| P23 - PIN\_NOT\_USED\_P23 (GPIO, Output, LOW, No Pullup) | P23 - PIN\_LCD\_COM2 (GPIO, Output, LOW, No Pullup) |
| P24 - PIN\_NOT\_USED\_P24 (GPIO, Output, LOW, No Pullup) | P24 - PIN\_LCD\_COM1 (GPIO, Output, LOW, No Pullup) |
| P25 - PIN\_NOT\_USED\_P25 (GPIO, Output, LOW, No Pullup) | P25 - PIN\_LCD\_COM0 (GPIO, Output, LOW, No Pullup) |
| P26 - PIN\_NOT\_USED\_P26 (GPIO, Output, LOW, No Pullup) | P26 - PIN\_LCD\_SEG6 (GPIO, Output, LOW, No Pullup) |
| P27 - PIN\_NOT\_USED\_P27 (GPIO, Output, LOW, No Pullup) | P27 - PIN\_LCD\_SEG7 (GPIO, Output, LOW, No Pullup) |
| // P31 - PIN\_UART\_TX (TX, Output, High, No Pullup) | // P31 - PIN\_UART\_TX & PIN\_EN2\_A (TX, Output, High, No Pullup) |
| P34 - PIN\_NOT\_USED\_P34 (GPIO, Output, LOW, No Pullup) | P34 - PIN\_LCD\_SEG2 (GPIO, Output, LOW, No Pullup) |
| P35 - PIN\_NOT\_USED\_P35 (GPIO, Output, LOW, No Pullup) | P35 - PIN\_LCD\_SEG3 (GPIO, Output, LOW, No Pullup) |
| P42 - PIN\_NOT\_USED\_P42 (GPIO, Output, LOW, No Pullup) | P42 - PIN\_LCD\_COM7 (GPIO, Output, LOW, No Pullup) |
| P43 - PIN\_NOT\_USED\_P43 (GPIO, Output, LOW, No Pullup) | P43 - PIN\_LCD\_COM6 (GPIO, Output, LOW, No Pullup) |
| P44 - PIN\_NOT\_USED\_P44 (GPIO, Output, LOW, No Pullup) | P44 - PIN\_LCD\_COM5 (GPIO, Output, LOW, No Pullup) |
| P45 - PIN\_NOT\_USED\_P45 (GPIO, Output, LOW, No Pullup) | P45 - PIN\_LCD\_COM4 (GPIO, Output, LOW, No Pullup) |
| P64 - PIN\_NOT\_USED\_P64 (GPIO, Output, LOW, No Pullup) | P64 - PIN\_LCD\_SEG8 (GPIO, Output, LOW, No Pullup) |
| P65 - PIN\_NOT\_USED\_P65 (GPIO, Output, LOW, No Pullup) | P65 - PIN\_LCD\_SEG9 (GPIO, Output, LOW, No Pullup) |

# Mobile App Details

## 5.1 General Setup (Android Development Only)

### 5.1.1 Development Environment Setup

Required Software for Installation on 64-Bit Windows:

[Android Studio](https://developer.android.com/studio) (Alternative IDEs such as VS Code and IntelliJ work well too)  
[Flutter SDK](https://docs.flutter.dev/get-started/install/windows)

Within the IDE, navigate to plugins:

In Android Studio:

**File -> Settings -> Plugins:**

Then, install the Flutter and the Dart plugins to allow for syntax highlighting.

**Appearance & Behavior -> System Settings -> Android SDK:**

The SDK versions installed during development: 12.0 (S), 11.0 (R), 10.0 (Q)

NOTE: Selected the Android SDK Location above.

**Languages & Frameworks -> Dart:**

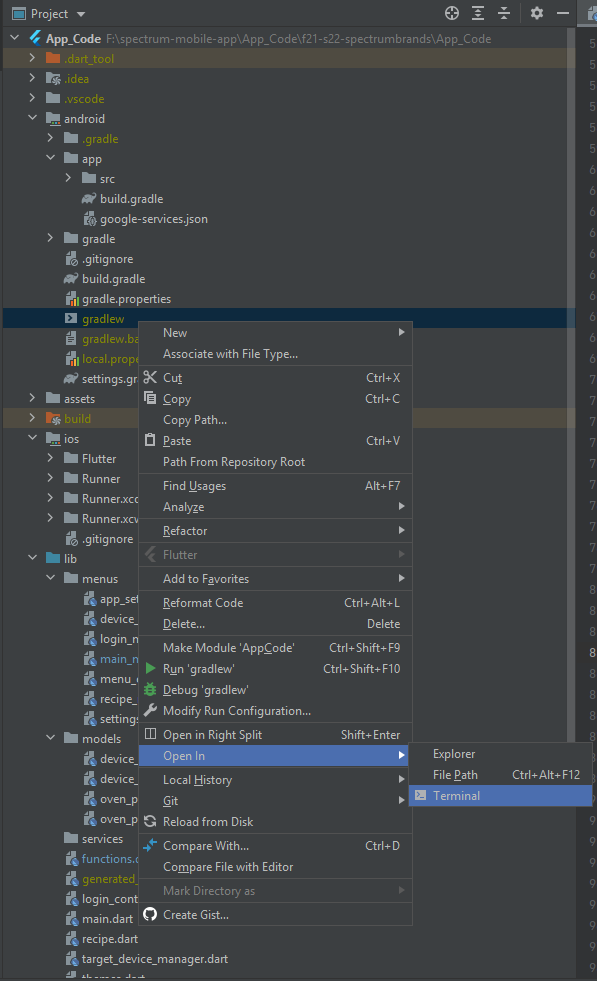
Target the Flutter SDK that was installed, will look like:

flutter\_windows\_x.x.x-stable\flutter\bin\cache\dart-sdk

Register device in Firebase:

With the project open (App\_Code is the project folder within the repo):

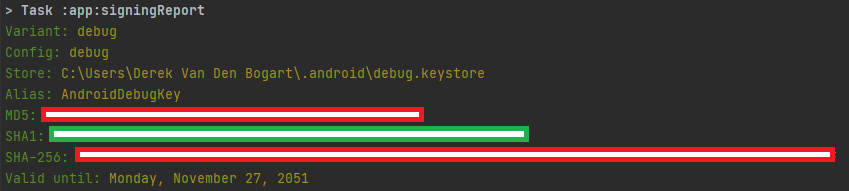
Navigate to App\_Code/android/gradlew. Right click and open in the terminal.



Type the command:

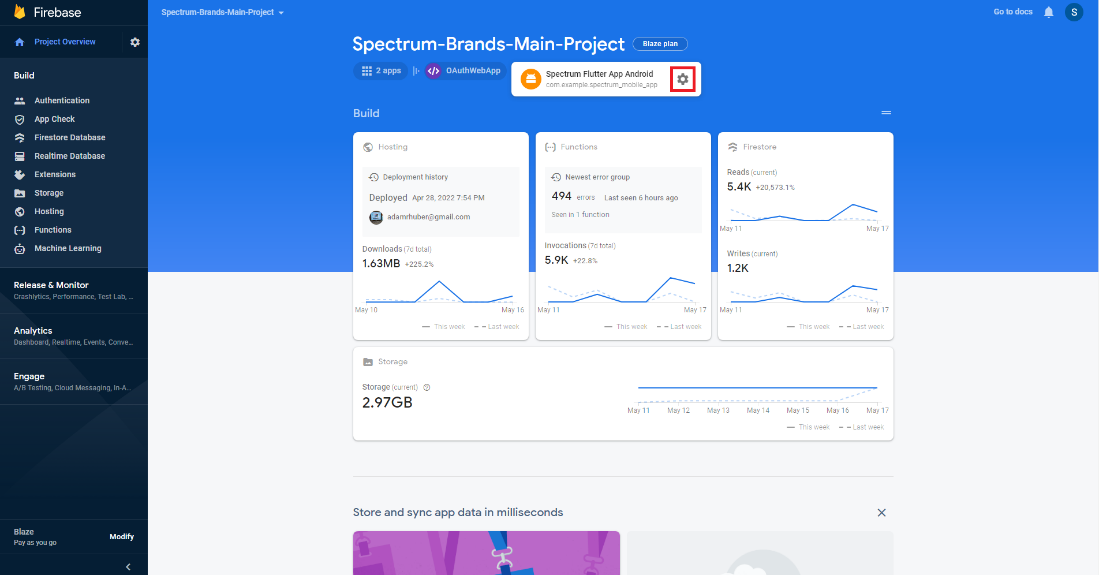
gradlew signingReport

Then, find Variant: debug, Config: debug and select the SHA-1 key.

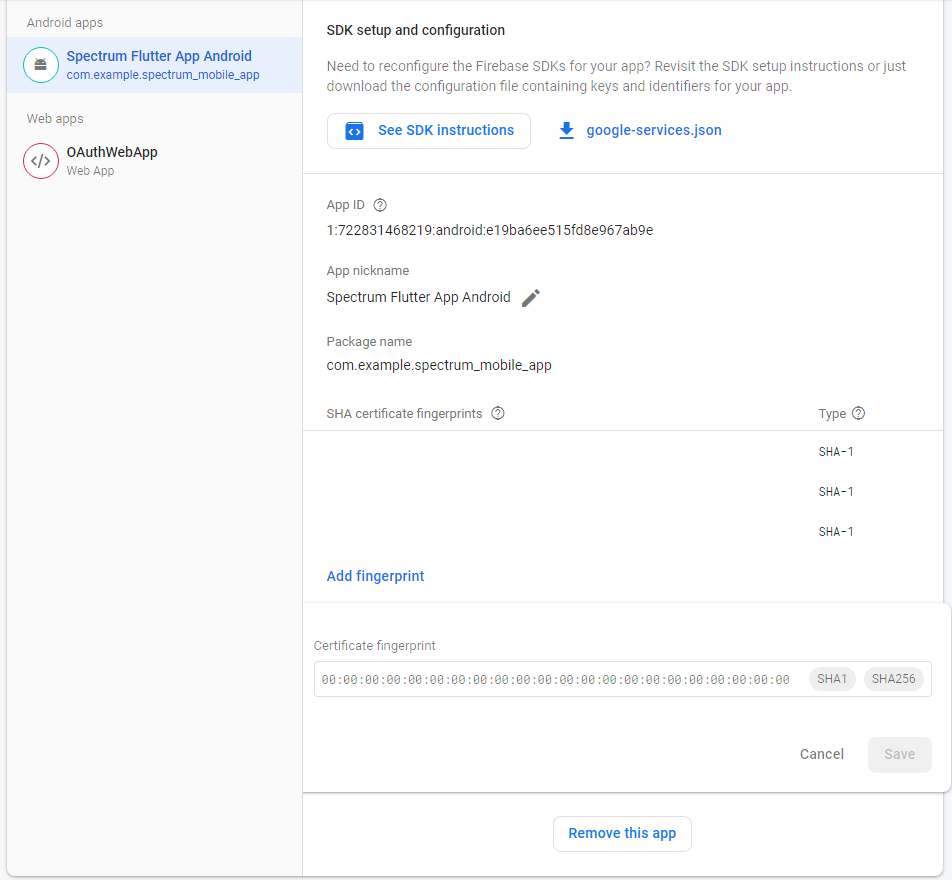


Next, navigate to the project [Firebase](https://console.firebase.google.com/?):

Select Spectrum-brands-main-project and open the Project settings:



Scroll down and add the SHA-1 Fingerprint:



Environment is now setup for development.

### 5.1.2 Setting Up Physical Device

Our development only included Android.

On an Android Mobile device:

Open Settings and navigate to About Phone -> Software Information -> Build Number:

Tap build number 7 times to enable developer mode.

The, still in settings, navigate to Developer options -> USB debugging

Enable USB debugging.

To run the app on phone: Android Studio, Intellij, and VS Code automatically detect your device. Then, run the program (or debug) from the IDE and gradle will build an Android apk and run it on the phone.

### 5.1.3 Setting Up Emulator Device

Within Android Studio:

**Tools -> AVD Manager**

Create Virtual Device... Then select the type of device and the image for it.

Select an image the correlates to your installed Android SDKs

Finish the Device Creation Wizard

Then run the device from the actions column in the main window of AVD Manager, IDE will detect this device the same way it detects physical devices.

## 5.2 App Structure

Menus:

* **App\_setup\_menu.dart:** Stateless view that decides which menu to show user during setup
* **Device\_registration\_menu.dart:** Builds objects for bluetooth connection and device registration
* **Login\_menu.dart:** Shows user a login button for Google Sign-In
* **Main\_menu.dart:** Menu with functions, time, and temperature, handles http calls
  + NOTE: These calls should be moved, but you already know that
* **Menu\_export.dart:** This file exports the menu files to be imported as a one-liner
* **Recipe\_menu.dart:** Not implemented
* **Settings\_menu.dart:** Contains app settings like theme and push notification options

Models:

* **Device\_parameters.dart:** JSON Serializable class for reading firebase responses
  + **Device\_parameters.g.dart:** Supporting class for device\_parameters.dart JSON
* **Oven\_parameters.dart:** JSON Serializable class for reading firebase responses
  + **Oven\_parameters.g.dart:** Supporting class for overn\_parameters.dart JSON

Other:

* **Login\_controller.dart:** Contains methods for google sign-in and user management
* **Main.dart:** Start point for the application
* **Recipe.dart:** Not implemented
* **Target\_device\_manager.dart:** Bluetooth connectivity and interaction class (See below)
* **Themes.dart:** Handles UI themes for the application (See below)

## 5.3 Package Usage

### 5.3.1 Bluetooth Communication Package Usage

Overview:

This file is responsible for connecting the phone to the Toaster via Bluetooth and sending the toaster necessary WIFI credentials. The file uses the Flutterblue Library and requires the phone to have WIFI and location permissions allowed from the phone

Private Variables Summary

* + String SERVICE\_UUID
    - This is the Unique Identifier for the service from the target device
  + String CHARACTERISTIC\_UUID
    - This is the Unique Identifier for the characteristic from the target device that is written to
  + String READFROM\_UUID
    - This is the characteristic that is read from the target device
  + String TARGET\_DEVICE\_NAME
    - This set to the name of the device required to be connected, in this case “Spectrum Brands Toaster”

Method Summary

* + ScanAndConnect(ssid, pass)
    - Scans for the target device defined by the TARGET\_DEVICE\_NAME, and if found, attempts connection and sends the WIFI credentials
      * Param String ssid, SSID of WIFI needed to be connected to
      * Param String pass, Password of WIFI needed to be connected to
      * Returns void
  + writeData(String data)
    - Writes param data to characteristic found on target device
      * Param String Data
      * Returns void
  + readData()
    - Reads data from the characteristic found on target device
      * Returns void
  + IsConnected()
    - Returns whether or not there is a connection between the app and the target device
      * Returns Bool
  + dispose()
    - Removes the connection and disconnects from target device

Private Method Summary

* + \_connectToTargetDevice(ssid, pass)
    - Connects the device and writes the SSID and Password to the device, allowing it to connect.
      * Param String ssid, SSID of WIFI needed to be connected to
      * Param String pass, Password of WIFI needed to be connected to
      * Returns void
  + \_disconnectFromTargetDevice()
    - Deprecated method
  + \_discoverServices()
    - Discovers any services defined by the SERVICE\_UUID and reads data from them
      * Returns void

### 5.3.3 Themes

Overview:

The Themes file is used to manage the themes used within the application. The system allows for the switching of themes and the saving/loading of themes. The file is responsible for the contents of the themes and the available themes.

Themes Enum:

The Themes Enum is used to establish the name/number of themes available for the app. The current enumerated values are as follows: Spectrum, Light, Dark.

Theme Summary:

Currently, there are three themes available: Light, Dark, and Spectrum. Each theme has four colors associated with it: Primary, Secondary, Tertiary, and Quaternary. The Primary color is used for any background elements. The Secondary color is used for any text elements. The Tertiary color is used for buttons and visual elements that need to pop out of the background but be distinct from the text. The Quaternary color is used for any accents that are desired for the application page.

Private Variable Summary:

* \_myTheme
  + Used to track the current theme of the application. Default set to Dark theme.

Method Summary:

* loadTheme()
  + Used to load the theme from the saved cache to \_myTheme
    - Returns type Bool true upon completion
* saveTheme(Themes theme)
  + Used to save the given theme to the saved cache
    - Returns type void upon completion
* getTheme()
  + Returns the value of \_myTheme
    - Returns type Theme
* getPrimaryColor(Themes theme)
  + Returns the primary color of the given theme
    - Returns type Color
* getSecondaryColor(Themes theme)
  + Returns the secondary color of the given theme
    - Returns type Color
* getTertiaryColor(Themes theme)
  + Returns the tertiary color of the given theme
    - Returns type Color
* getQuaternaryColor(Themes theme)
  + Returns the quaternary color of the given theme
    - Returns type Color

## 5.4 Testing

Tests for the methods in Themes.dart can be found in Themes\_Test.dart, all tests contain the expected values. If changes are made to the colors within the themes, the tests need to be updated to ensure matching values. The remaining classes do not currently have test cases.