Technical Specification

Project Title: Sonic
Student 1: Jason Henderson; 19309916
Student 2: Conor Joyce; 19425804
Supervisor: Stephen Blott
Date Completed: 04/03/2022

Table of Contents

1. Introduction **2**
1.1 Overview/ Motivation 2
1.2 Glossary 2
1.3 Research 3

2. High Level Design **4**
2.1 NFC Scan Sequence Diagram 4
2.2 User Registering on App 5
2.3 User Logging in on App 6
2.4 Covid Notify Sequence Diagram 7
2.5 Data flow diagram 8

3. System Architecture **9**

4. Implementation **10**
4.1 Backend 10
Tools: 10
4.2 Frontend 11
Tools: 11
4.3 Scanner 12

5. Problems Solved **12**
5.1 NFC Understanding 12
5.2 Databases 12
5.3 Framework Choice 12
5.4 Testing 12
5.5 Mobile Application 12
5.6 Learning New Things 12
5.7 Error Handling 13

References **13**

1. Introduction

1.1 Overview/ Motivation

Sonic is an NFC-enabled Covid contact tracing solution for businesses, venues and other busy locations. Sonic is an end-to-end system that emphasises ease of use and can be used in tandem with existing contact tracing solutions such as the HSE's Covid Tracker app or the NHS's equivalent. These existing solutions use a smartphone's Bluetooth connection for contact tracing between devices, while Sonic uses NFC-enabled cards that can be scanned on entry and exit of buildings/venues and has a companion app to notify the system of a user having tested positive.

The name "Sonic" originates from the video game character of the same name, whose unique attribute is his speed, which is also a prominent feature of the Sonic system.

Sonic consists of a Go REST API backend, a relational PostgreSQL database, an NFC scanner script written in Python and a React-Native/Typescript frontend companion app.

Users will have an NFC "Covid Card", similar to a Leap Card, that they will use to "tap in" or "tap out" of a building/venue. The scanner device - a Raspberry Pi is used in our demo, for ease of development - will send a POST request to the backend REST API that will store a log of this user entry/exit in Sonic's PostgreSQL database.

If a user tests positive for Covid-19, they can use the Sonic companion app and press the "I Have Covid" button, which will send a POST request to the backend REST API also, that will scan through the PostgreSQL database to find any close contacts of the user within the last 3 days, and then send each of these close contacts a push notification via the Expo Notifications API.

1.2 Glossary

Nfc
Near-field communication is a set of communication protocols for communication between two electronic devices over a distance of 4 cm or less.
Backend Server
In simple terms, a backend server is a server that we don't see that is a running server.
Frontend
The frontend refers to the client-side of things, e.g the index page of an online shop
Database
A database is used to store information.
PostgreSQL (Postgres)
PostgreSQL, also known as Postgres, is a free and open-source relational database management system emphasising extensibility and SQL compliance.
Raspberry Pi
Raspberry Pi is a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation in association with Broadcom.
Go (Golang)

Go is a statically typed, compiled programming language.

React Native

React Native is an open-source UI software framework created by Meta Platforms, Inc. It is used to develop applications for Android, Android TV, iOS, macOS, tvOS, Web, Windows and UWP by enabling developers to use the React framework along with native platform capabilities.

Expo

Expo is a framework and a platform for universal React applications.

TypeScript

TypeScript is a programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript and adds optional static typing to the language.

RESTful API

A REST API (also known as RESTful API) is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for representational state transfer.

HTTP

The Hypertext Transfer Protocol is an application layer protocol in the Internet protocol suite model for distributed, collaborative, hypermedia information systems.

GET request

The HTTP GET request method is used to request a resource from the server.

POST request

POST is a request method supported by HTTP used by the World Wide Web. By design, the POST request method requests that a web server accept the data enclosed in the body of the request message, most likely for storing it. It is often used when uploading a file or when submitting a completed web form.

JSON

JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays. It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers.

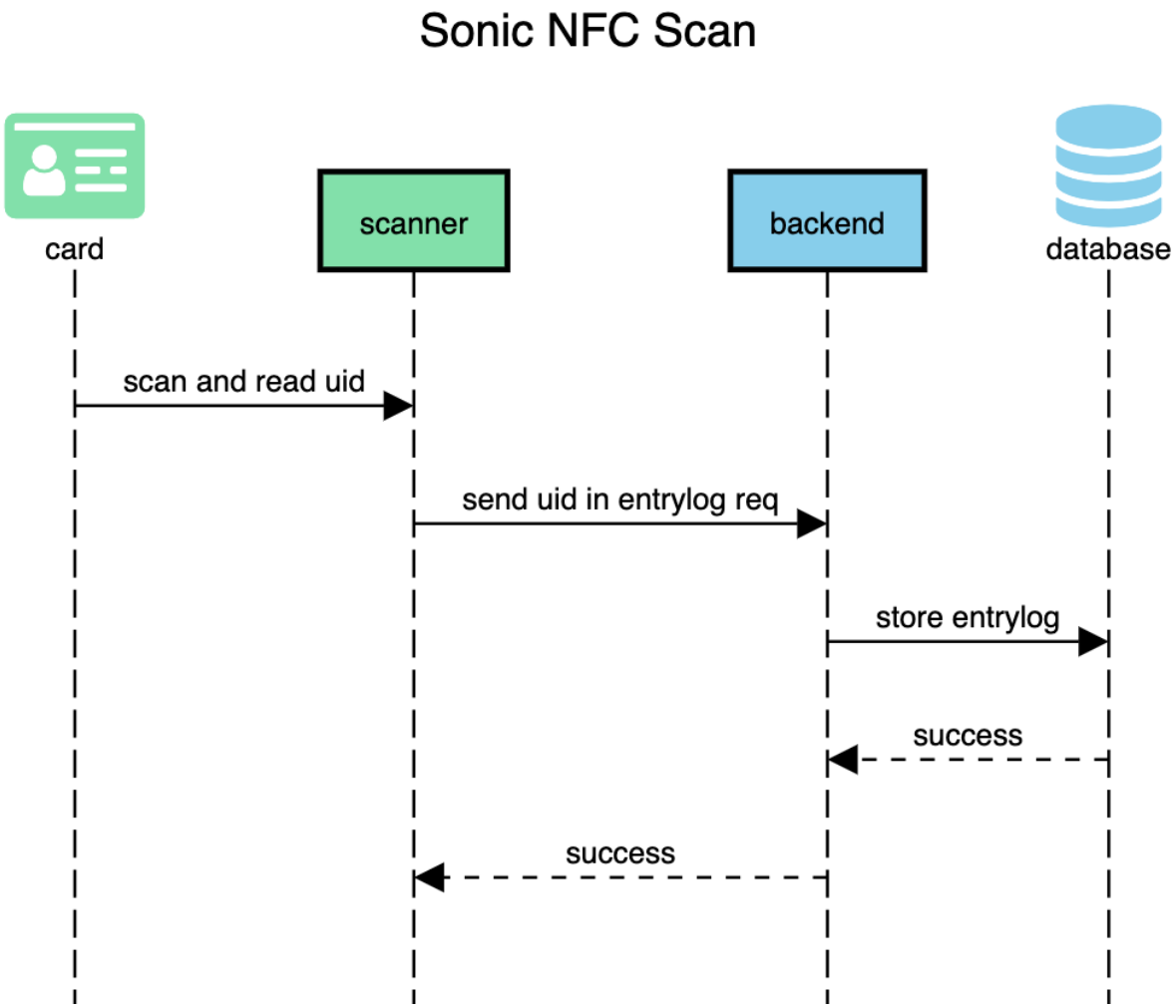
1.3 Research

To start work on this project it was necessary to carry out sufficient research on what tools we were going to use and what skills we needed to complete the key features we originally planned on doing. Early research required deciding on a suitable framework and database for our backend. We chose the Chi framework for Go and decided to use Postgres as our database. The reasoning behind these decisions was that these solutions seemed the most ideal for problems we would need to solve.

Moving on we needed to learn how to scan an NFC card and obtain the data on it. Once the scanning of NFC card's research was complete we could move on to research on our frontend application. We had decided early on we would make an application in React Native but in carrying out further research, we built on to that idea and built a React Native application using the Expo CLI tool. This enabled us to obtain a frontend project with features from the getgo and already containing a boilerplate template.

2. High Level Design

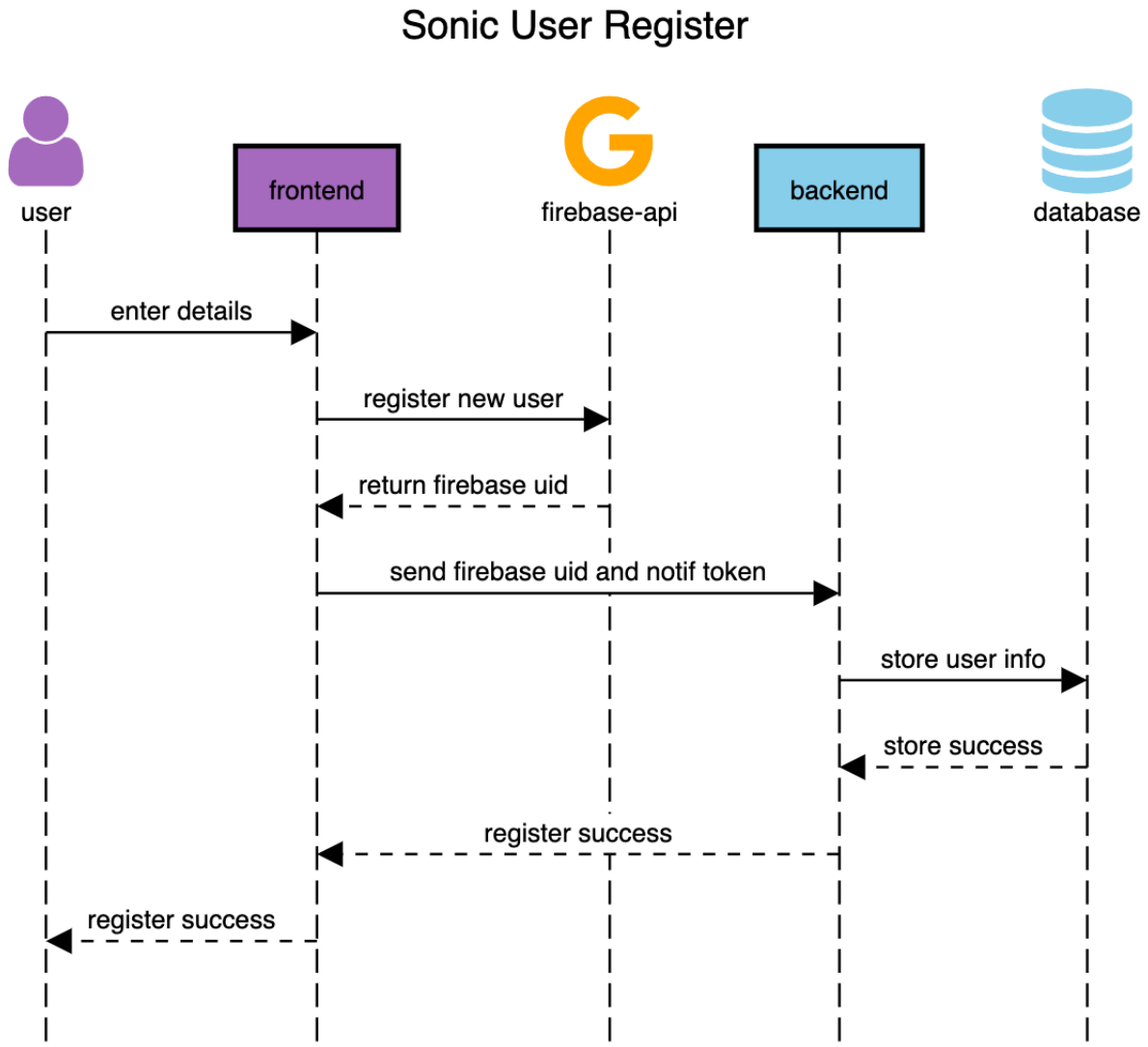
2.1 NFC Scan Sequence Diagram



This sequence diagram represents when a user scans their card on the scanner upon entering or exiting a facility.

- The user scans the car
- Card data containing user uid's is sent to our backend
- Backend communicates with the Postgres database and stores data
- Database returns response to the backend
- Backend returns response to the scanner

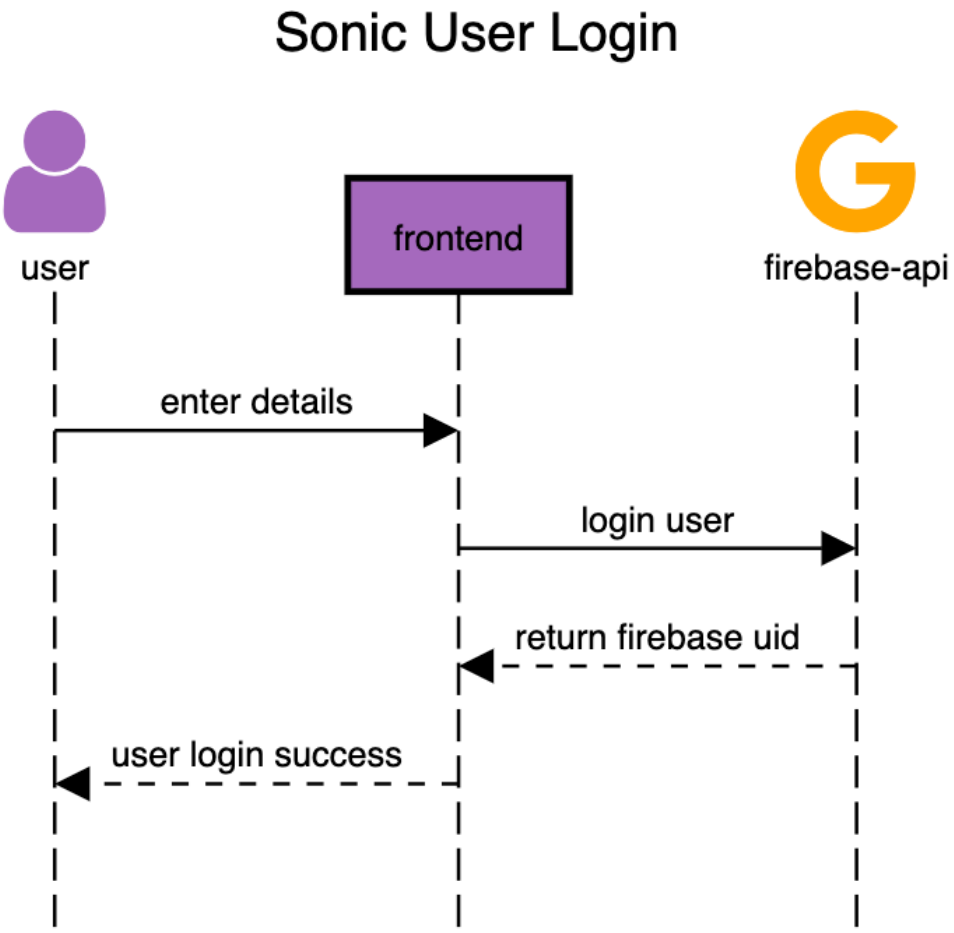
2.2 User Registering on App



This sequence diagram represents a user registering/signing up on our front end application.

- User enters email and password details on the app
- Application registers the new user using Firebase and sends Firebase uid along with notification token to backend
- Firebase returns uid to the app
- Backend stores user info in the database
- Success response from database returned to backend
- Registration success returned from backend to front end
- Front end returns registration success to the user

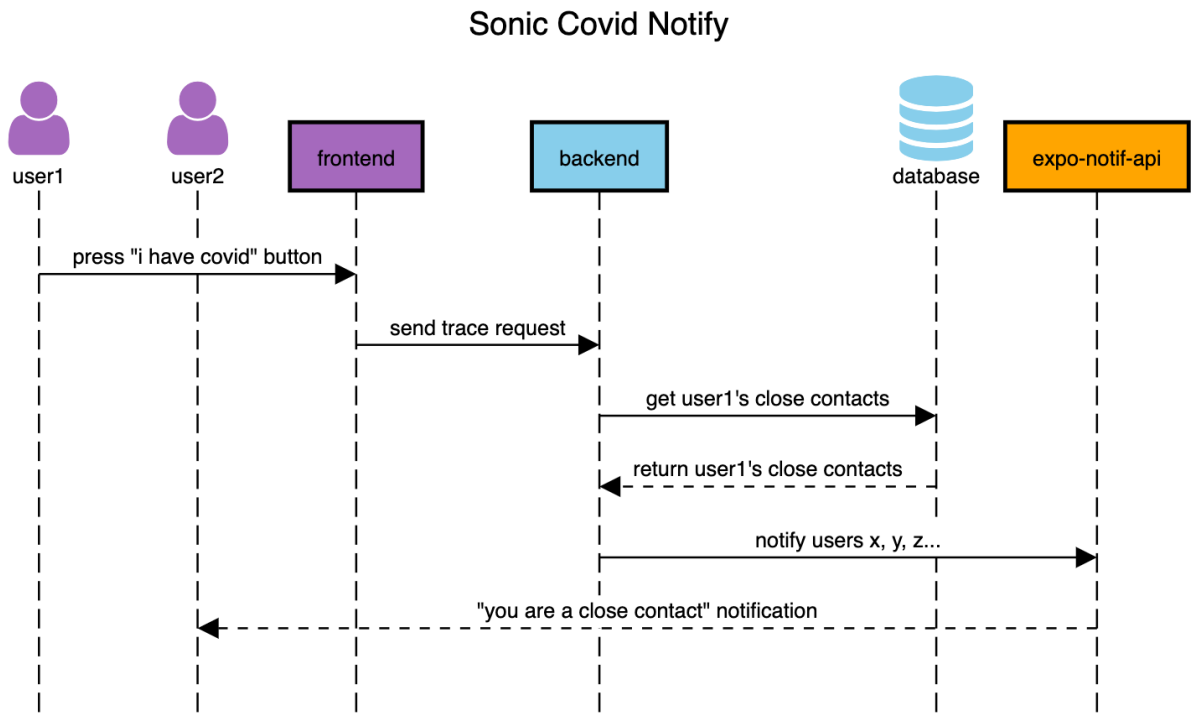
2.3 User Logging in on App



This sequence diagram represents an already registered user logging in on our front end application.

- User enters login details on the app
- Frontend uses Firebase to login user
- Firebase returns firebase uid to frontend
- Login success returned from frontend to user

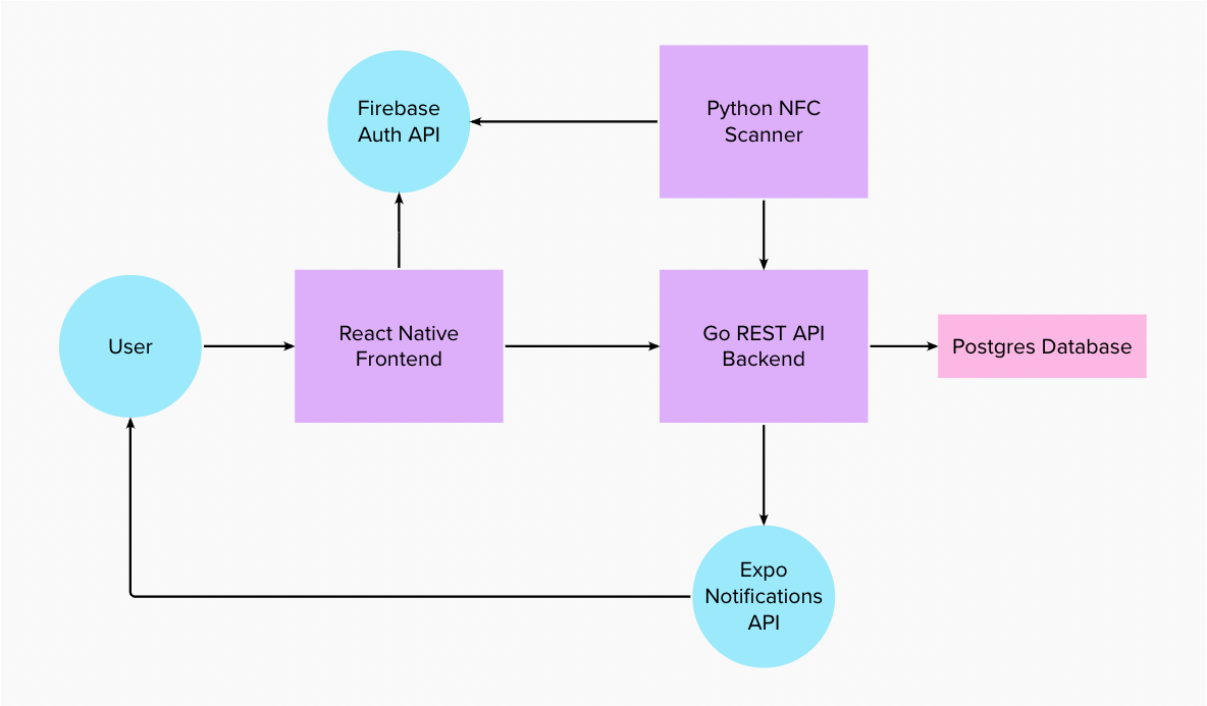
2.4 Covid Notify Sequence Diagram



The last sequence diagram represents the notification process to close contacts of a user who has selected that they have Covid.

- User1 presses the “I Have Covid” button on the front end app
- Front end sends a trace POST request to the backend trace endpoint
- Backend gets user1’s close contacts from the database
- Database returns user1’s close contacts to the backend
- Backend uses close contact data to contact the expo notification API to request to send notifications
- Expo notification API contacts close contacts of user1

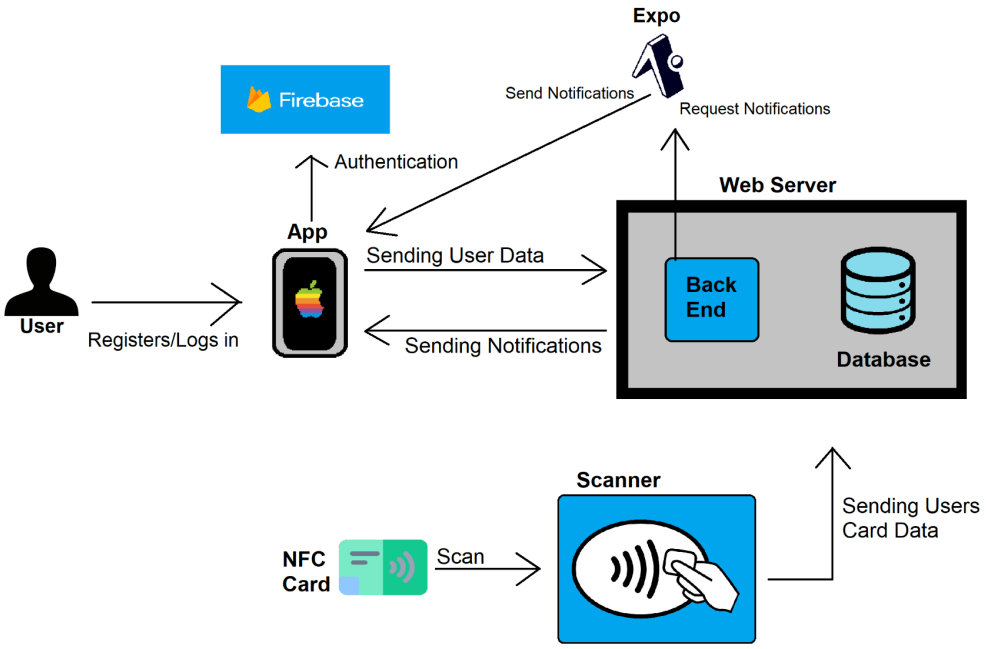
2.5 Data flow diagram



This data flow diagram represents the interactions and flow of information between the different components of our system.
It also shows the use of external APIs within the project, as such:

- Both the frontend and the scanner send requests to the Firebase Authentication API
- The backend sends requests to the Expo Notifications API

3. System Architecture



The Sonic system contains/utilises these components:

- A **backend server** that consists of a Go RESTful API that implements the go-chi routing framework, and a PostgreSQL relational database that stores all information necessary for the system to operate.
  These are deployed via Docker and docker-compose to a homelab server maintained by one of us. The aliveness of the backend can be found at this address: https://sonic.cawnj.dev/health
- A React Native **frontend mobile application** written in Typescript and hosted on Expo. This application is used to send a request to the backend when a user tests positive for Covid and as an interface for push notifications via the Expo Notifications API.
- An **NFC card scanner** that is connected to a Raspberry Pi, will scan a user's card on entry/exit to a building/venue and send the user's id (which is stored on the card) to the backend server for logging.
- The **Firebase Authentication API** is used for authentication within the frontend mobile application and for differentiating between users anonymously via their Firebase UID.
- The **Expo Notifications API** is used to send push notifications to any close contacts detected by the backend on a contract trace event.

4. Implementation

4.1 Backend

The backend consists of a server written in Go. This server is the backbone of the Sonic project and is where all the heavy lifting and logic takes place. Go contains in-built functions and methods for hosting a web server but we decided to use the go-chi routing framework alongside this. go-chi is a lightweight, idiomatic and composable router for building Go HTTP services, and resulted in a more simplified, readable and manageable routing method for our API implementation.

PostgreSQL was chosen over other database solutions as we realised very early on that a relational database would suit the needs of the project best, as we would be performing more than basic queries for the contact tracing function of the backend. Also, Go has good support for relational databases such as PostgreSQL and alongside go-chi's Bind() method, we could map a JSON object to a struct type that represents what the underlying row in the database looks like.

For each endpoint, a handler function was written. These handler functions contained any logic required solely within the backend, and/or calls methods on our Database struct type that would perform queries for us. The handler functions also implement a lot of error handling techniques for each edge case, which standards for writing Go code really emphasise - e.g. most functions will return the requested data alongside an error object, which will be 'nil' in the case where nothing goes wrong, else it will contain data regarding the error which can then be handled easily upon function return.

The structure of the Go backend is a "main.go" file that imports other subpackages such as db, handler and models to serve the API.
The backend is deployed via a docker image build of the "sonic-server" and containers created with docker-compose, which can be accessed publicly via an existing reverse proxy instance on our homelab server which also includes DDoS protection via Cloudflare.

Tools:

- Go
  Dependencies:
  - go-chi: for routing HTTP requests
  - godotenv: for use of .env files storing sensitive info
  - pq: for interactions with the postgres database
  - exponent-server-sdk-golang: for sending requests to the expo notif API
  - apitest: for unit tests of the API
- PostgreSQL
  Dependencies:
  - postgis: adds a geometry type that can store coordinates
- Make
- Docker
- docker-compose

4.2 Frontend

The frontend application was created using Expo and React Native. Expo allows the app to be published online and run on its own Expo Go app meaning the Sonic app can be accessed remotely anywhere as long as there is an established network connection. An advantage of using React Native with Expo was that during the initialising of the project, options for pre-existing boiler-plate templates were offered upon setup. This allowed for an initial setup with a pre-existing app, some sample screens and navigation already implemented on top of which we could add our own features.

After the initial setup of the project was completed, all major components of the app were modified in the screens directory for the pages that the user was going to physically see within the app. Other changes such as modifications to themed components to enable light/dark mode and changes to the index navigation file to ensure all navigation worked correctly were also made.

Use of the Firebase SDK was used on the project to enable seamless authentication for users upon registering or logging in to the Sonic app. Once a user has signed in, a POST request to the backend's /register endpoint is made containing the user's unique firebase uid and expo notification token. The backend then stores this information in the PostgreSQL database.

If a user has Covid they can press the "I Have Covid" button on the home screen which brings them to a confirmation page where the user confirms this action, as to prevent misclicks of the button sending contact trace requests to the backend. If the user confirms and presses the "Continue" button, a POST request to the backend's /trace endpoint is sent containing the user's uid. From there the backend will find all the given users' close contacts and use the Expo Notification API to send all of the close contacts a push notification on their device.

Tools:

- Node
  Dependencies:
  - eslint: for linting
  - prettier: for linting
  - firebase: for communication with the firebase API
- Typescript
- React Native
  Dependencies:
  - react-native-elements: for card styled usage
- Expo
  Dependencies:
  - expo-device: for logging the current device being used
  - expo-notifications: for sending push notifications to users
  - expo-permissions: for allowing notifications to be sent

4.3 Scanner

The scanner is used for scanning users' cards that contain their user id and sending a POST request to our backend REST API containing this data. The scanner has two capabilities, scanning the cards and sending data to our server, or formatting the cards, e.g reading/writing data to the card. Both these functionalities for the scanner are written in Python.

The reading/writing to the cards is merely so that we can manually put data like user ids on each card so that when they are scanned for data to be sent to the backend, requests can be sent correctly with the sufficient data required in request bodies.

The scanning operation involves the user scanning upon entry and exit of the given building that the scanner is located. When the card is scanned, the scanner retrieves the user's id from the card and sends it in a POST request to the backend's /entrylog endpoint alongside the current time and location id to map their entry and exit times.

An admin can write a given user's id to a card with the "write_firebase_uid.py" script, which simply interacts with the Firebase API to grab the relevant user id by inputting the user's email address and writes that user id to the card.

Tools:

- Python
  Dependencies:
  - nfc: to enable interactions with the scanner
  - ndef: for encoding and decoding messages in standardised NDEF format
  - firebase_admin: for communicating with Firebase auth API as project admin

5. Problems Solved

5.1 NFC Understanding

5.2 Databases

In the beginning, deciding on what database we should use to store all data was difficult - for example, whether to use a NoSQL or relational database. We decided to use a relational database as we would be performing more than basic queries across the data for contact tracing functionalities on the backend. Therefore PostgreSQL was chosen to be our database as it seemed best suited for our project's goals.

5.3 Framework Choice

We knew we wanted our backend to be written primarily in Go as it is well suited for creating a fast and flexible REST API, and did some research on if there were any frameworks or modules that would assist in creating this. In early research, we were strung between using

either the Chi or Gin routing frameworks to simplify the routing process while minimising bloat. The main aspect in deciding to use Chi over Gin was that Chi is more minimal and felt like using an extension for the native Go HTTP modules rather than a more complex framework-like experience with Gin.

5.4 Testing

Testing was quite a challenge for us during the project and took some time to get to a state that we were comfortable with. All of the heavy-lifting of the project is done in the backend so this is where we focused our efforts in terms of testing due to time constraints. Unit tests were a clear choice for testing the API routes to ensure everything was working correctly, and we used the “apitest” module to assist with this, allowing us to match expected responses against certain types of data, or against patterns with regex.

For overall system testing we created a bash script - found in /src/e2e-test.sh - that would perform all the same HTTP requests that would be performed in regular system use:

- User registration
- Scanner sending entrylog requests for users entering/exiting a building
- User pressing the button in the app confirming they have Covid

Since most of the logic happens in the backend, we decided that performing a system test this way still covers a lot in terms of the scope of the project and what we thought was important to focus our efforts on in regards to implementing this within the given timeframe.

5.5 Mobile Application

In knowing early on that we would carry out the application development in React Native, we didn’t know how we would publish our app to be run on our physical mobile devices to demonstrate key features. In carrying out research we learned we could initialise a React Native project bundled with Expo via their CLI which made development easier and allowed us to publish our app to run on a device via their “Expo Go” mobile app. This also allowed us to invite other users to the project to test our app and give us feedback on the UI and UX.

5.6 Learning New Things

To complete this project at a high level, it required both of us to adapt to each other's strong points and learn new things. The creation of our backend was almost completely new to myself (Jason) as it was the first time I encountered the language Go but it was one of Conor’s strongest languages.
We both had experience with MySQL but using Postgres alongside using the chi module for Go were both completely new to us. Docker was another thing we had to learn more about in order to dockerize our server and have it running from a container, this feature was pushed heavily by Conor as he suggested that dockerizing the app would help ease deployment during the development of the project and to make the API public.
We both sharpened our knowledge of NFC and the NDEF message standard in the creation of the scanner in Python.
The creation of our frontend was done mostly by myself as I had more experience with HTML, CSS and JavaScript, and these skills transferred well to developing in React Native.

Conor also assisted with certain aspects of the frontend, especially with getting the requests working properly and we both learned a lot about TypeScript and React.

5.7 Error Handling

Error handling was a big focus in the Go backend since Go as a language emphasises good error handling as most functions do return an Error object. A lot of testing was involved in figuring out different edge cases and implementing flexible error handling statements that can handle each of these.

References

https://github.com/go-chi/chi
https://www.postgresql.org/
https://docs.expo.dev/
https://reactnative.dev/
https://docs.expo.dev/versions/latest/sdk/notifications/