Contents

- **Pruthvi Shrikaanth, King Edward VI Camp Hill School for Boys**

Secure Defender - a security system to arrange the guarding and defence of buildings in need of security.

Analysis

Physical security of buildings is a problem that is often overlooked in the UK. It is very uncommon to see guards or any security around buildings except for the most important government offices. However, this approach of trusting that no trouble can occur on British soil has been proven unsuccessful time and again. From the 2011 London riots to the 2022 Leicester attacks on homes and places of worship, on both occasions police struggled to contain mass gatherings and disperse crowds, often being attacked or otherwise harmed themselves in the process. One of the reasons for this is that by the time police respond, large crowds have already assembled - for example, in many crowd crushes around the world, no amount of police that arrive after a crowd has formed can do anything other than waiting for it to disperse - attempting to do so themselves would cause a stampede and large-scale death.

A way that I could solve this problem is by designing a security system - many people are hesitant to adapt adequate physical security systems including guards for maintaining safety around and within their buildings - this is because it is not only expensive but also difficult to manage. I could address both of these issues by creating a simple, cheap and relatively independent system that manages physical security arrangements for buildings - it will include live tracking of guards, a feature to automatically call nearby guards for support and place information on the guards' whereabouts on a map that is visible to a security overseer. Additional security measures can be implemented on the overseeing side such as password/passcode access to the system.

Computational methods

- I can make use of abstraction to simplify the projection of security personnel onto a plan of the area - my software need not be aware of decoration in the walls or excessive furniture. It only has to know the basic shape of the building or area that is to be guarded, so that when personnel are projected onto it, it is simple to tell where they are. Another way I can use abstraction is in simplifying how the system represents personnel - I only need a unique identifier and their current location to project them onto our plan - other information is not so useful and can be discarded. These uses of abstraction mean that less data needs to be transferred between devices from the personnel and a central controlling/monitoring device. The advantages of this include a lower security risk, as even if data is intercepted, its uses would be

limited since the software only transfers what is necessary. It also means that the central computer can receive data faster and process it more quickly - this is very important for a system that works in real-time.

- Technologies I use will include geolocation and network communication. Geolocation is so that the current location of personnel can be fetched; network communication is so that this can be sent to a central computer.
- The computational approach to this problem of security would be more maths-related than how a human would manage such a system. This is because the computer is able to convert location coordinates into a place on an area where it projects markers for personnel, while a human would take much longer plotting such data on a graph - this would be even more problematic as this data is being consistently updated. The advantage of how a computer handles this issue is that it visually displays where personnel are, making it easier to understand a security picture.
- Decomposition can be used to split this problem into a few main components. One of these is the basic framework of our software - the network model it will be using. This enables personnel to communicate with a central computer, a bit like a star network. Another component is how I can convert location coordinates of personnel into a projection on a map. The final component is our alerting system, which also relies on my network model to send location information - this part can be developed independently of the map-projection component as both work from another distinct component, which is the network model. An advantage of decomposition (splitting the problem into smaller, more manageable parts) is that I now have an idea of how I should start development - I need to start by designing the network model. The other components are also much easier to design and implement as they can be built separately.
- Decomposition leads to modular code and design - a way of building software that splits distinct parts of a program by their purpose so that my code can be more easily read and modified by others (and me in the future). For example, my alerting system will be coded and stored separately from my mapping system. I can also write different code depending on if the device is the central controller or that of the personnel, while only using the parts that are necessary.

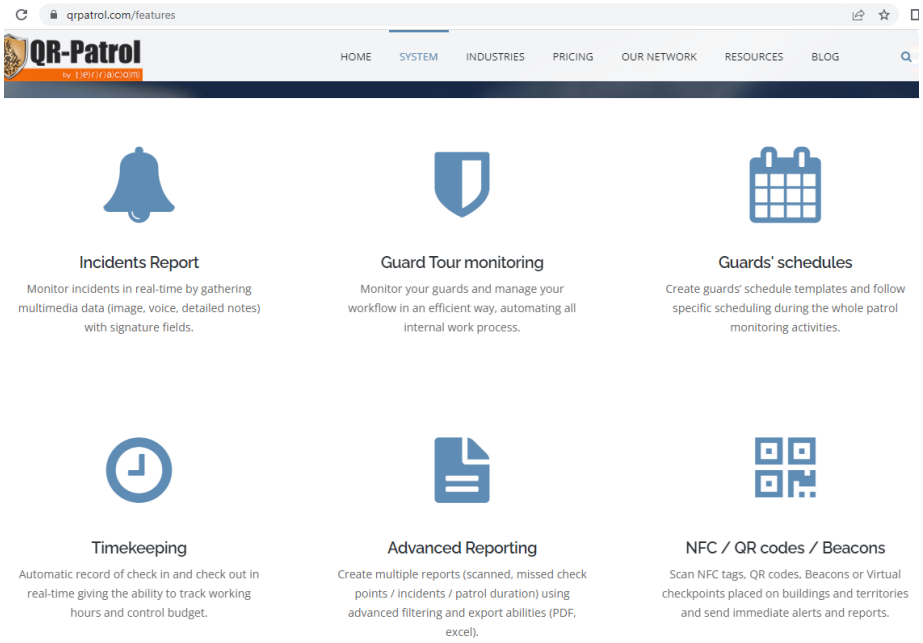
Stakeholders

1. Warehouse/factory manager - they manage big trading entities' storage spaces, holding much bulk product, material and potentially dangerous substances. They would want to use this software to appropriately maintain security around the warehouse and prevent any thieves from stealing products. This simplifies the security management process and reduces the possibility of any security breaches from the administrator side (i.e. lower chance of dishonesty from a security manager if he exerts less control).
2. Head of defence, army captains and government institutions (eg: Parliament, the Prime Minister) - by nature, these people and entities have access to important information - strong security that protects from external and internal attackers is a need, and this can be managed more effectively by a system that isn't as easily fooled as a human. These people would use my solution to safeguard themselves and any sensitive information they may possess - it would benefit them by simplifying the management process, and the amount of control that security controllers (in the case of larger buildings) have could be adjusted for security needs.
3. These are the main stakeholders I will target with my software: hospital CEOs, leaders of places of worship, school headteachers - they could employ security in these places as they can be sensitive targets to any sort of attack, therefore it would be beneficial to have a security system in place. These people would make use of my security system to prevent low-level antisocial incidents from taking place, but also for evacuating personnel and civilians in the case of a major attack or disaster. My security system would help them by not only allowing automatic calls for backup when needed but also coordinating how this backup is provided. The following are my user group who I will interview so that I can choose in which direction to head when building my software.
    a. I can consult a safeguarder (Mr Dipak Mistry) of a place of worship near me to see how useful a security system for a place of worship would be.
    b. I can also consult Dr Shrikaanth Krishnamurthy, who works in numerous hospitals and would understand the need for security and in what form it should be presented.
4. Everyday population - ordinary people hold stake in that their safety can be better guaranteed by a security system controlling who enters and exits public buildings. They would not set up my software or manage it, rather being users in the sense that they benefit from its effects.

Research and Ideas

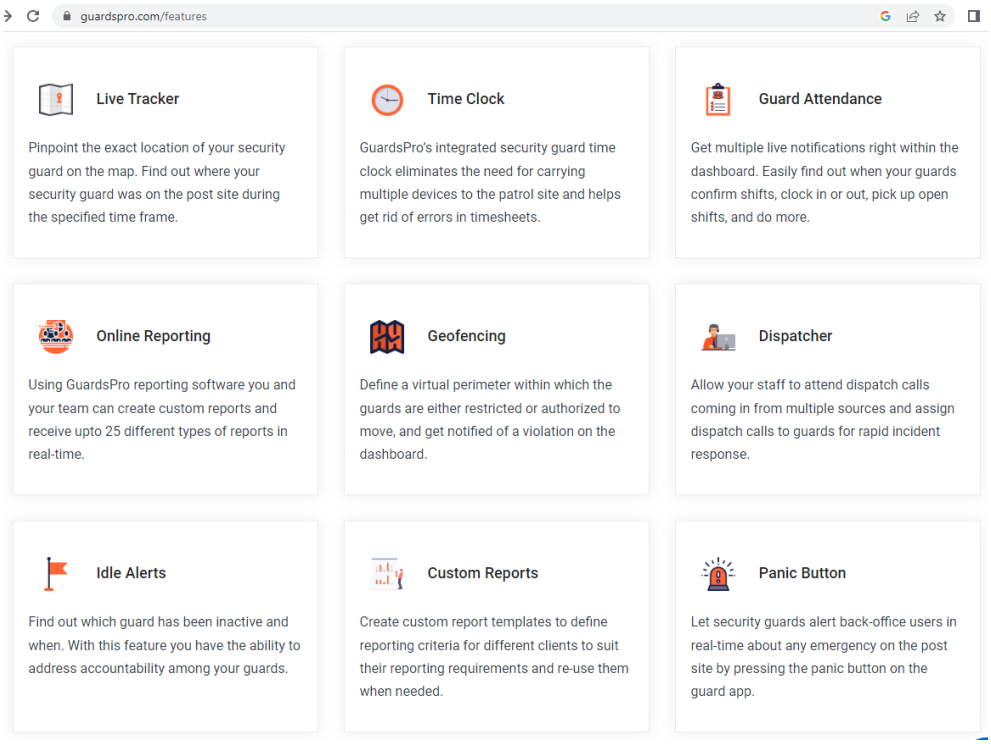
Existing solutions:

a. QR-Patrol, a real-time guard monitoring system which works by a system of checkpoints and registration. Guards scan codes at checkpoints periodically for the system to show that the areas they should scan are being patrolled. An email is sent for each time a checkpoint is missed, and SOS alerts are sent by guards in case there is a problem. Advantages of this system include the ability to send out instant alerts, which is a feature that I want to include in my system. Patrol routes can also be remotely configured. However, setting this system up takes a long time and is difficult to change as beacons or codes must be physically moved each time a patrol route is changed, unless the same landmarks are being used in a different order or assigned to someone else. If a patrol is missed, only an email is sent, which cannot be seen in a time-critical situation - combined with this, the system does not track guards, only checkpoints. Resultantly, if a guard abandons his post then there is no way of knowing this until a checkpoint is missed and the email is sent and acknowledged - this time delay is an important concern that I want to fix by tracking the live location of guards, rather than relying on checkpoints.



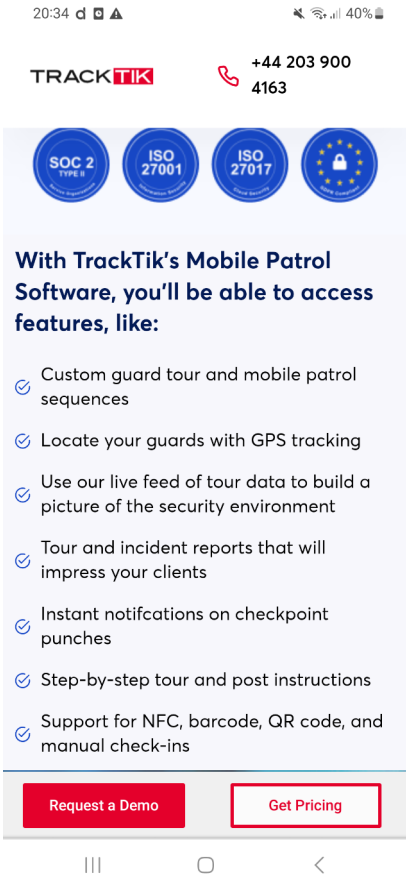
b. GuardsPro is a system to manage security guards that is quite feature-rich, and presents itself as a mobile app for guards, and a web application for managers. It offers live-tracking, a panic button, patrol history and geofencing (notifying if a guard strays outside of his patrol area) among other numerous features. It comes in with a built-in communication system for guards and security managers to communicate with each other. It is advantageous in that it does not rely on a preset system of beacons, rather tracking guards itself. However, it is an extremely cluttered app due to so many features, and is also

quite pricey - it seems reasonable at 10 USD per user per month, but this scales up quickly since there may be tens of hundreds of users (guards) since the same people may not guard every day or all of the time. Many of the features also cost extra per user per month, making this a less affordable solution for those who want basic security for smaller buildings such as places of worship.

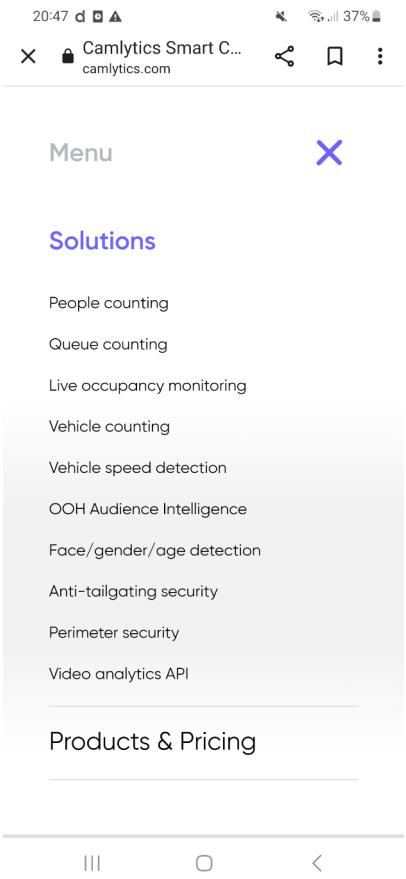


c. Tracktik is a piece of software that specialises in managing guard patrols. It offers GPS tracking, a checkpoint-registration system and incident reporting. It is advantageous in that it offers basic necessary features to build a real-time picture of the security of an area, without having to generate numerous reports or wait for the scanning of any checkpoint. However, a problem in it lies where it does not do anything other than produce data for the security manager setting up the software. It cannot automatically request for backup or handle security situations without someone monitoring it and doing it themselves. This causes a problem if the monitoring computer fails or the

manager is unavailable, possibly rendering the system useless in the moment.



d. An alternative option that works quite differently is Camlytics - rather than managing guard patrols or even employing guards, it makes use of cameras and artificial intelligence to detect people, vehicles and motion. This is advantageous in that it can be used to notify security advisors of any trespassing at times when nobody must be present in an area. It also learns as it does its job, meaning that it is always improving due to its functioning on artificial intelligence. However, it is not able to differentiate between employees and intruders, nor is it able to provide a response to any security incidents - it is an incomplete software in this sense as it has to work in tandem with other pieces of software to provide these features.



Preliminary conclusions and requirements:

- I will go for a guard-tracking approach rather than a camera-recognition approach as it is important to differentiate between authorised personnel and intruders - this is done much more easily by a human than a camera using artificial intelligence.
- Live-tracking of personnel is a must to ensure that an appropriate real-time response can be provided in the case of any security incident. It also ensures that if any personnel are misbehaving then this can be swiftly detected and appropriately dealt with.
- My system should have relative independence and be able to handle problems by itself to an extent - this solves one of the main problems with existing security software; a security overseer does not need to command everything, especially with minor incidents that may occur often on large campuses.
- It must be simple to operate so that it can be implemented almost immediately - users should not require training to use my software.
- It needs to be designed appropriately with fast response times - it has to run smoothly as security incidents are time-critical.

Interviews and Transcript

Questions for Dr Shrikaanth Krishnamurthy:

These questions are designed to first get an idea of any current security arrangements in a hospital setting and how secure they are (1). They are also designed to see how useful some of my key features would be (2, 3).

1. How do you feel about the security at your workplace?
   I think security is okay but I think it's not foolproof. It is obviously dependent on people to look after (the security), sometimes if they go out for a cigarette, for example, or if they are not around on that day then the security does come down.
2. Would you find a way to instantly call for help or backup useful?
   I think that would be really helpful, yes.
3. What other features would you look for in a security system?
   I suppose to monitor people, having circuit TV or cameras to capture people if they walk in and out - that would be quite helpful to install as well.

Questions for Mr Dipak Mistry:

The purposes of these questions were to see how security is employed in a contrasting setting to that which the previous interview focussed on (1), and to gather an idea of how my software would be used and how it could be designed (2, 3).

1. In your opinion, how safe is this building in terms of security?
   Very secure. It's got quite heavy-duty doors, very secure doors. It's got a number of locks on each side, it has CCTV as well and it's got an alarm system where the police are informed if it goes off pretty much straight away. There are cameras around all sides of the building. On top of that, it has double glazing with locks on each window so they're hard to open.
2. How do you think a software that manages secure guarding of the area would improve security?
   There haven't been many incidents but it'd be good to have (a software) to keep the place safe, especially when there are people living here to ensure their safety. When events are going on, we want to make sure the public are secure as well.
3. What features would you look for in a security software that works alongside guards?
   An alerting system so that if there is an intruder then someone can be on-site within 5 minutes - either the alarm company, a third-party security company or even the police. We wouldn't want to be coming here ourselves if there's an intruder.

Proposed Solution

As a result of these answers to my interview questions, I have decided to stick with my idea of a guard-managing security system and include the following key features:

- Live-location tracking of personnel to show up on the central computer. This is the main feature of my software - to visually display how a place's security is being managed. This is the simplest, best and most easy-to-understand way to implement this, as discussed in the computational methods section and realised through my research.
- Alert functionality to alert the central computer - the central computer can forward this to other personnel as backup, while the security manager can also call for additional support if necessary. This section can be modified to take photos at the time of an alert, automatically call other external services such as emergency, or anything else that a client would specifically need - additional functionality can be decided when I analyse how useful it would be in practice and whether it is wanted/needed.
- The ability to send updates from the central computer to personnel - this ensures that specific instructions can be sent to personnel, either to be used as a part of other features of my software or for the security manager to manually type in an instruction and send it to a specific guard.
- It should be designed to work with other programs - as Dr Shrikaanth Krishnamurthy mentioned in my interview with him, it would be useful to maintain security when personnel leave the premises or when non-security personnel leave the premises - this could be done by counting people with a camera, like Camlytics. However, since I am going for a guard-tracking approach, my main focus cannot be on implementing security through cameras, rather through eyes. This is why I want my software not to be entirely closed-off - clients should have the ability to use it in conjunction with other software that adapts to their needs if my software doesn't fulfil all of them.

Limitations

1. Economical limitations include limiting which equipment I can use - I cannot buy dedicated equipment for location-fetching from satellites. Therefore, I will have to use GPS - this in turn will limit how accurate personnels' locations will be, including height-wise. This will influence my requirements and success criteria.
2. A communicational limitation is also a result of economic limitations - I will have to use the public Internet to transmit data as I cannot set up my own network or use dedicated devices that form a separate network, which unfortunately weakens the security of my system as it makes data interception easier.
3. Another limitation is of devices - since I don't possess a MacBook or anything of the like, I cannot develop any sort of app for iPhones, which will be quite limiting in terms of the testing phase - if I build my software for the personnel as a mobile application, which is the next-best solution after using dedicated equipment, then I will effectively eliminating anyone who does not possess an Android device from testing my software. An alternate option would be to build the software as a web application, but this is also problematic - I would have to host it on the Web, which costs money; the only free option that is suitable to my project is Github, but this reveals the source code publicly; obscuring this removes the ability to test my software. As I cannot publish my source code for this project since the project will be assessed, I have no choice but to develop it as an Android app for the purposes of this project.
    - Note: this applies only to the part of the software that works on the personnels' devices - the part for the central computer can be developed as a web app. Since it is only being tested on a single computer, I will not need to transfer the code very often and therefore there is no need to host it.

Requirements

1. I will need access to a web-based data-hosting service to transfer data between personnel and the central computer - I will use Google Firebase as it is free for small projects and feature-ready.
2. I will need access to Internet-ready GPS-enabled Android devices for testing purposes - I have a few at home that I can sideload my personnel-side software on.
3. I will need access to a computer that I can render websites and access the Internet from - I have one at home.
4. I will also need some knowledge of programming for the Web and for Android - I have this knowledge (I can use HTML, CSS and JS on the Web while combining this with the Firebase API; I will use Dart (Flutter) for building the android application, also in combination with the Firebase API).

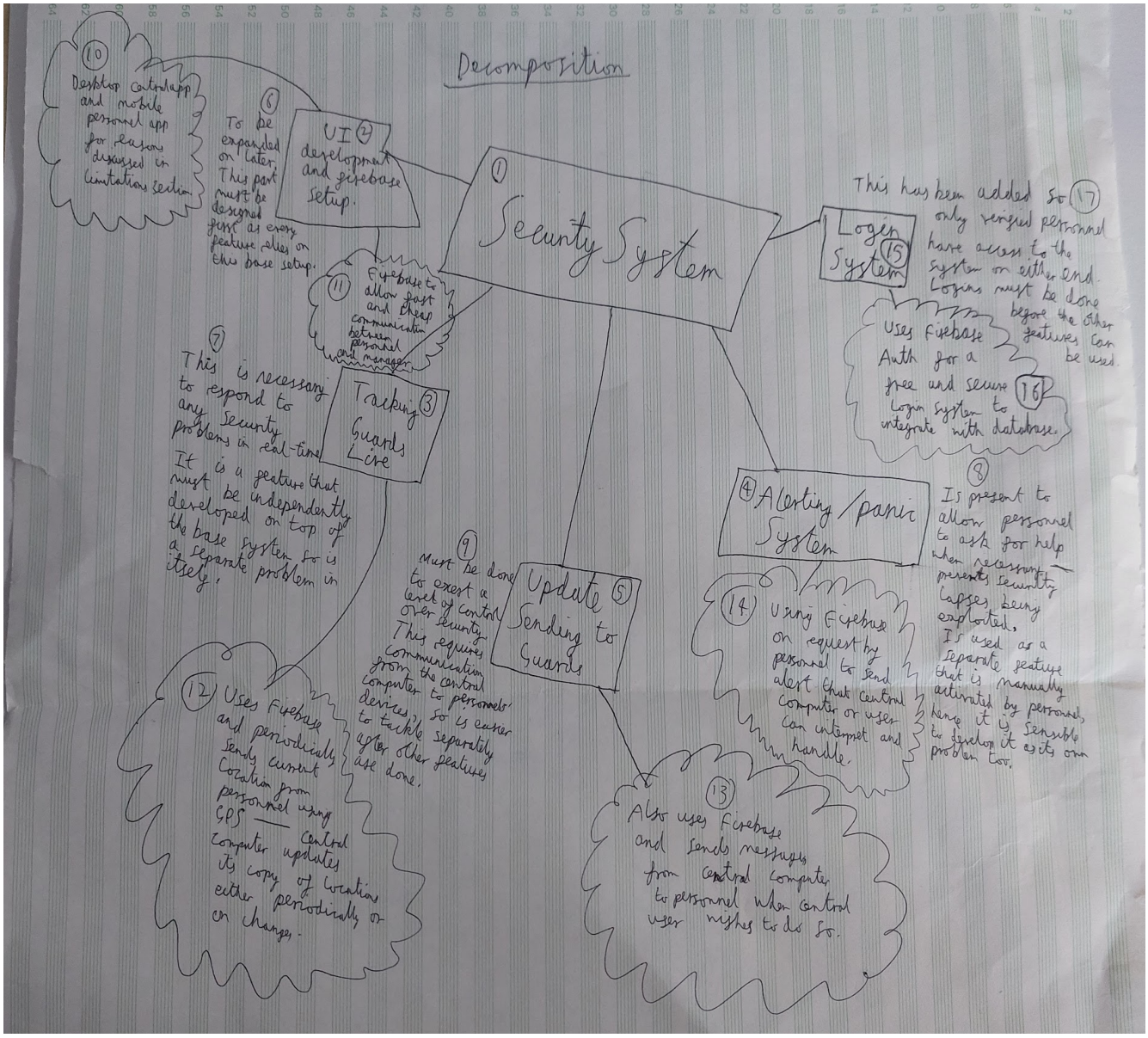
Success criteria

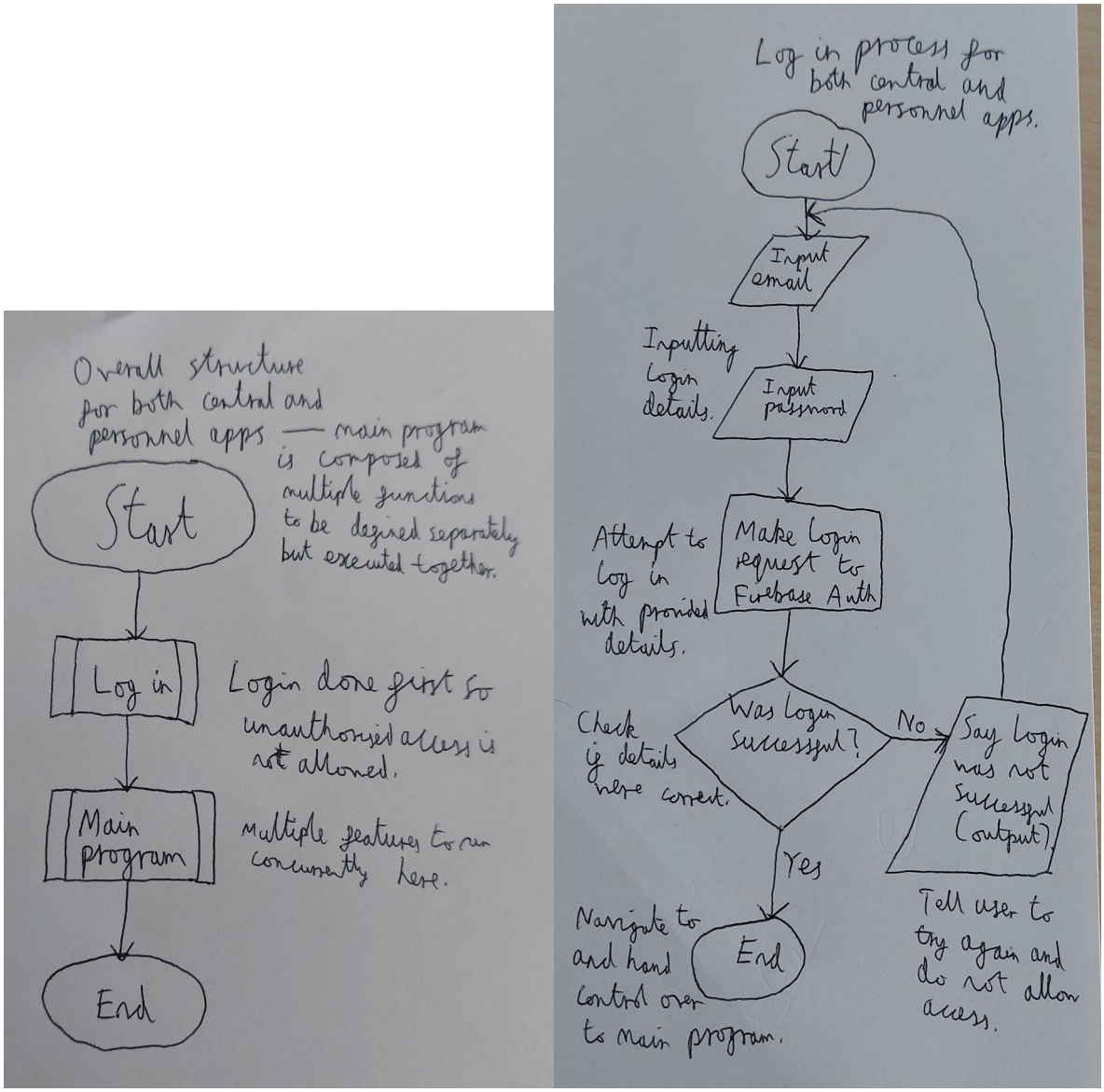
1. The personnels' locations must update at least once every 5 seconds - this requirement is to ensure that their whereabouts are known from sufficiently recent times so that the location projections are close to real-time. I will not make this requirement too rigorous as I am not using dedicated devices or networks to transmit data - therefore, I will be limited by signal strength and network traffic on the public Internet.
2. The personnels' locations must be accurate within a 5 metre radius - this requirement is so that live-location tracking gives a rough but accurate picture of the security situation in a place. My choice of 5 metres was influenced by the fact that GPS is only accurate from 30 cm up to 500 cm when it is functioning correctly, therefore I want to ensure that if there are location problems, I don't pick up a weak GPS tracker in a testing device as a bug in my own code.
3. The alert system must at least be able to send an alert to the central system within 5 seconds - this ensures that even if it doesn't automatically notify nearby personnel or another security service such as the Police, the security manager at the central computer (or the central system itself) can do so.

Design

Decomposition

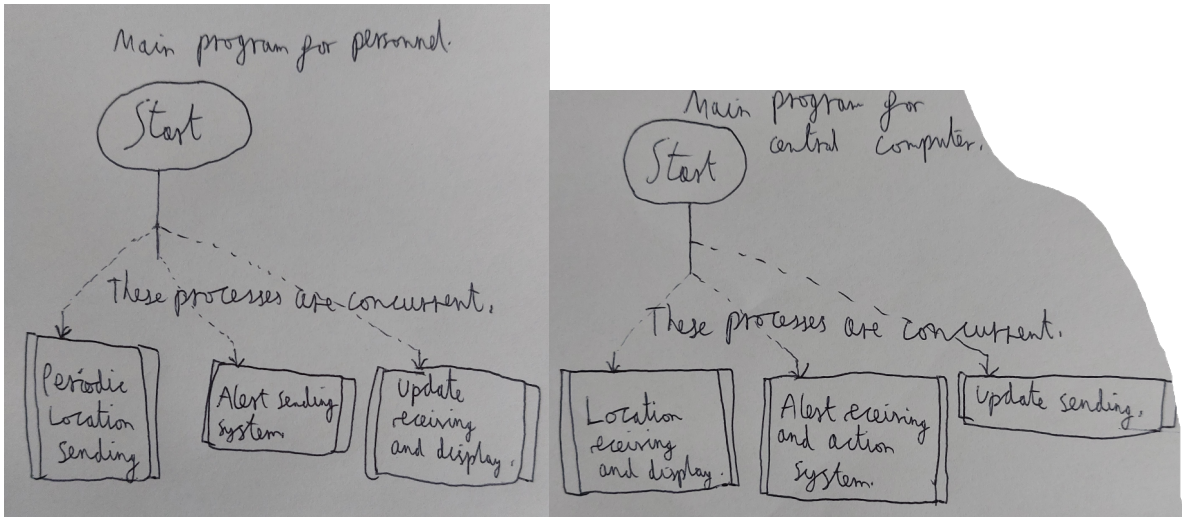
I have split my project into 2 apps - one for the personnel and one for the central computer. I made this choice because of the natural many-to-one relationship between personnel and a security manager - the system should reflect this relationship. Both do fundamentally different things and building one common app to fit all of them would sacrifice the organisation and readability of code - it is therefore a better option to build 2 apps that are distinct and serve more specific purposes.



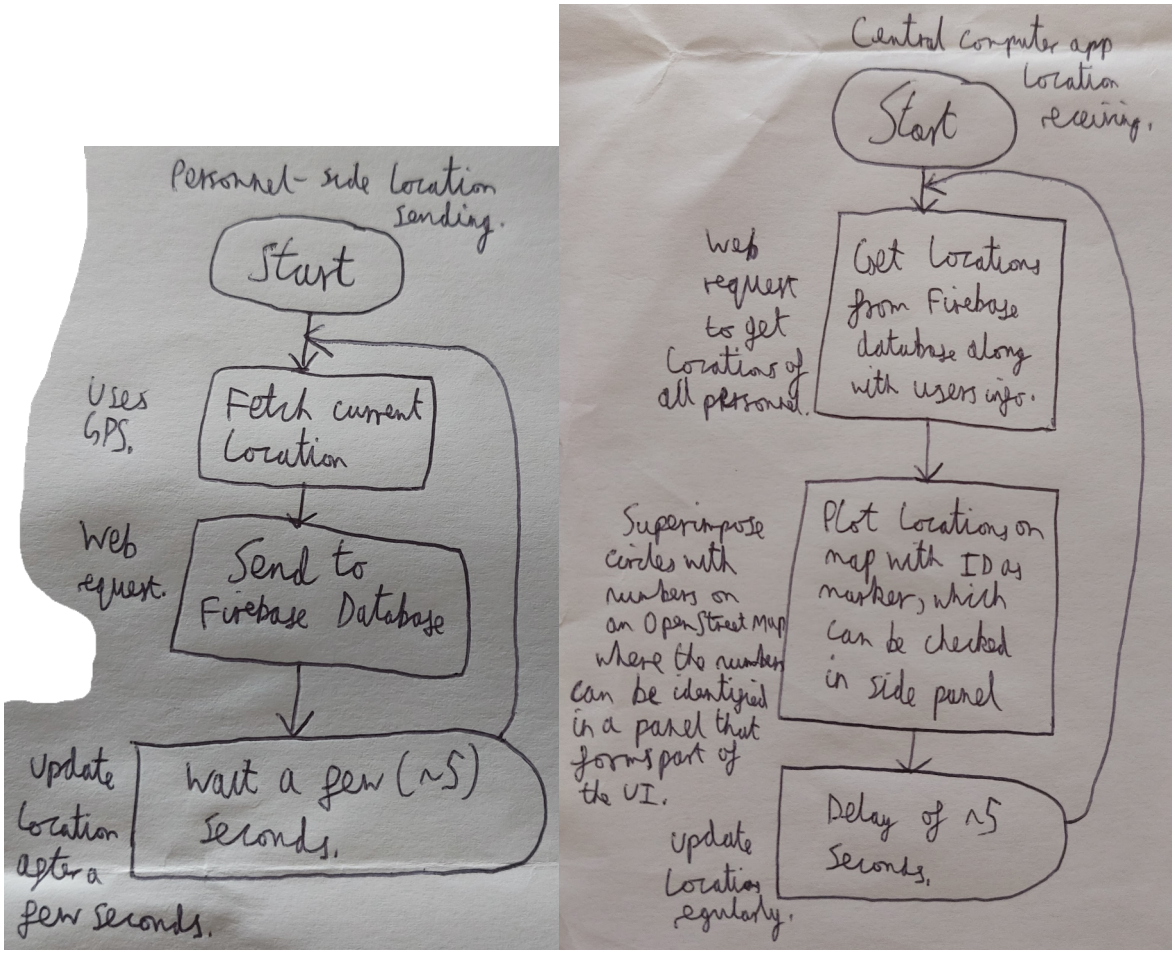


Left: Main structure for both apps, right: login systems for both apps

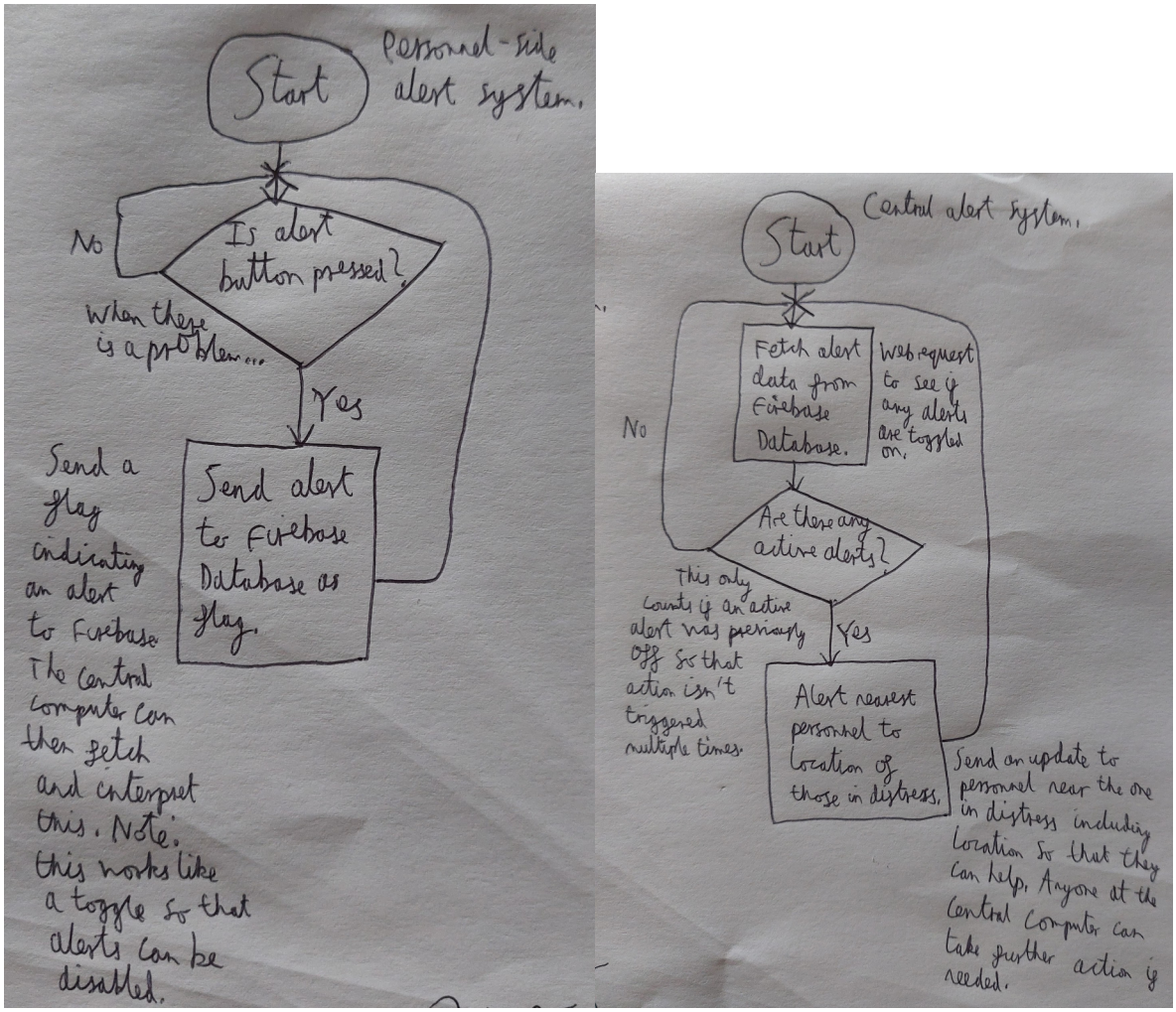
The choice of using a login system before proceeding to the rest of the app was due to security concerns - this design ensures that if one has not logged in (therefore his identity is unverified) then he cannot access the system from either side, ensuring that nobody can infiltrate the security system.



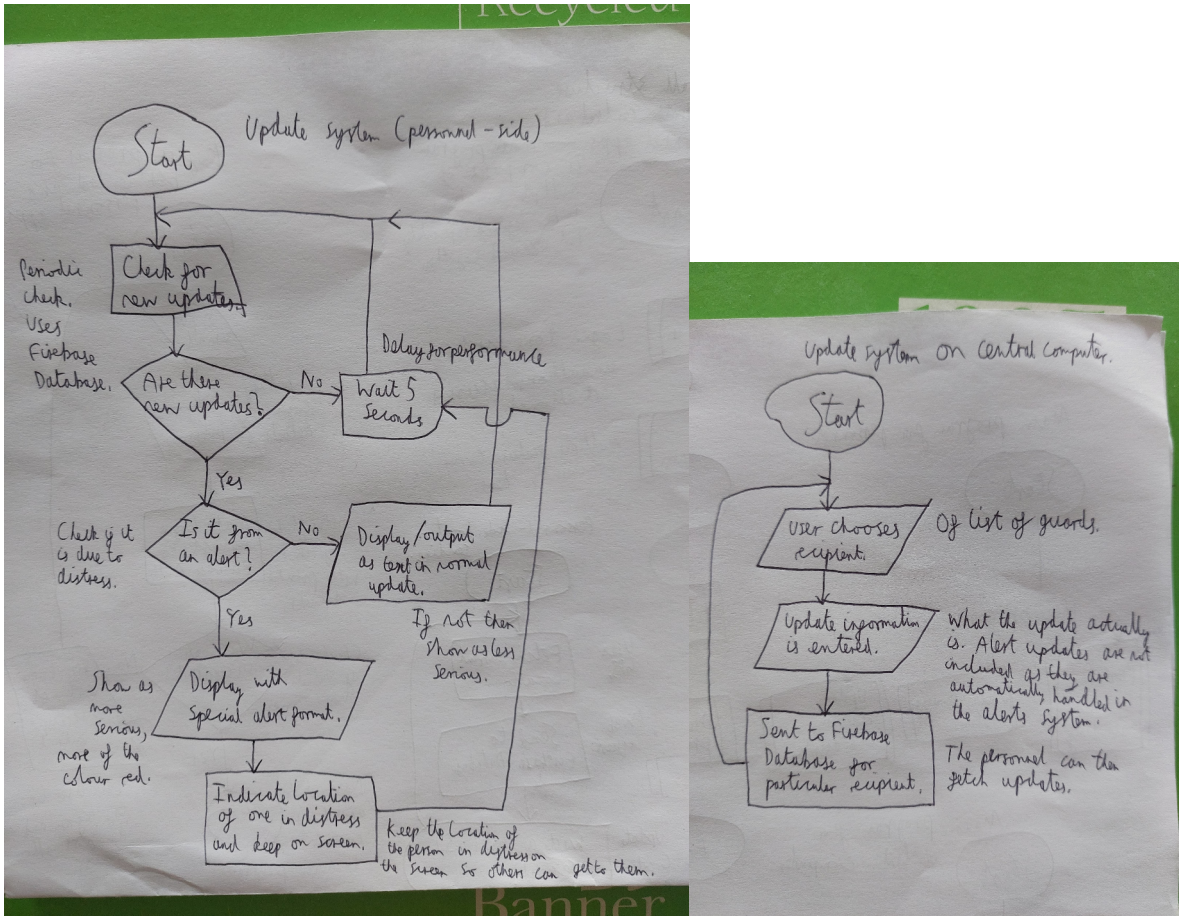
Main program internal structure (left: personnel app, right: central app)



Location management (left: personnel, right: central)



Alerting system (left: personnel, right: central)

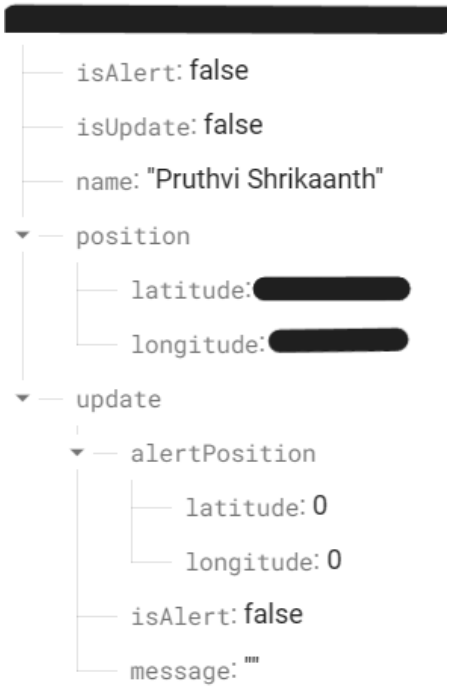


Updates system (left: personnel, right: central)

User and Database System

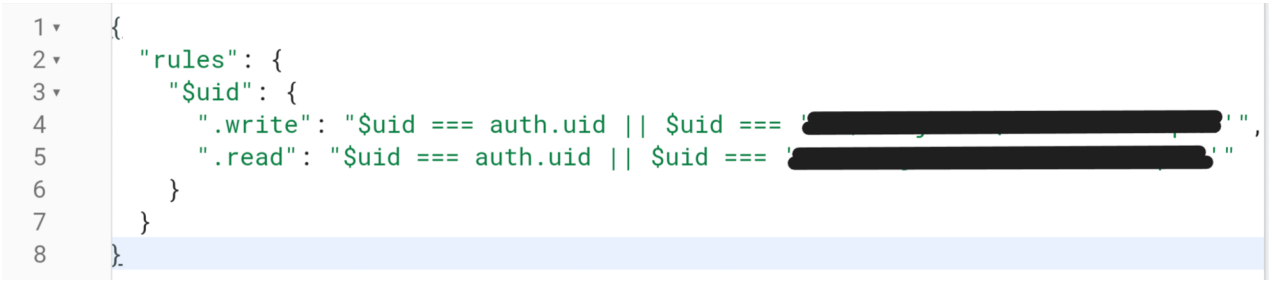
For managing users, I will use Firebase Authentication as it is well-developed, reputable and secure. It is also simple to use and integrate with Firebase Realtime Database, therefore it is the most suitable option for my project, which uses both. For each user, Firebase Authentication will only store their email address, a salted hash of their password and their user ID, which it generates on creation of that user.

For storing data, I will use Firebase Realtime Database as it is really fast, which is important for a security system which works similar to a real-time system such that guaranteed fast response times are required. It is also reliable, which is important as security systems may be used 24/7 in some cases. Contrary to its name, I will be using it to manage data in JSON format as this format is much more accessible and easier to work with using Javascript and other Object-Oriented languages such as Dart (used with Flutter apps). My data structure will look like this for each user (all under the root):



The alphanumeric string represents the user ID, which is known from when the user logs in with Firebase Authentication. i*sUpdate* indicates whether there is a currently active update or alert for that user. *position* stores their latitude and longitude, which is updated every few seconds. *update* stores the current/previous update and also whether it is an alert or just a regular update. *latitude* and *longitude* are stored as numbers because this also simplifies calculating distance between different longitudes and latitudes (to be used when alerting other personnel to someone in distress). They are stored inside *position* as they will always be used together. I am storing *isUpdate* separately from the updates themselves so that bandwidth is not wasted fetching previous updates when they are not active. *message* is stored as a

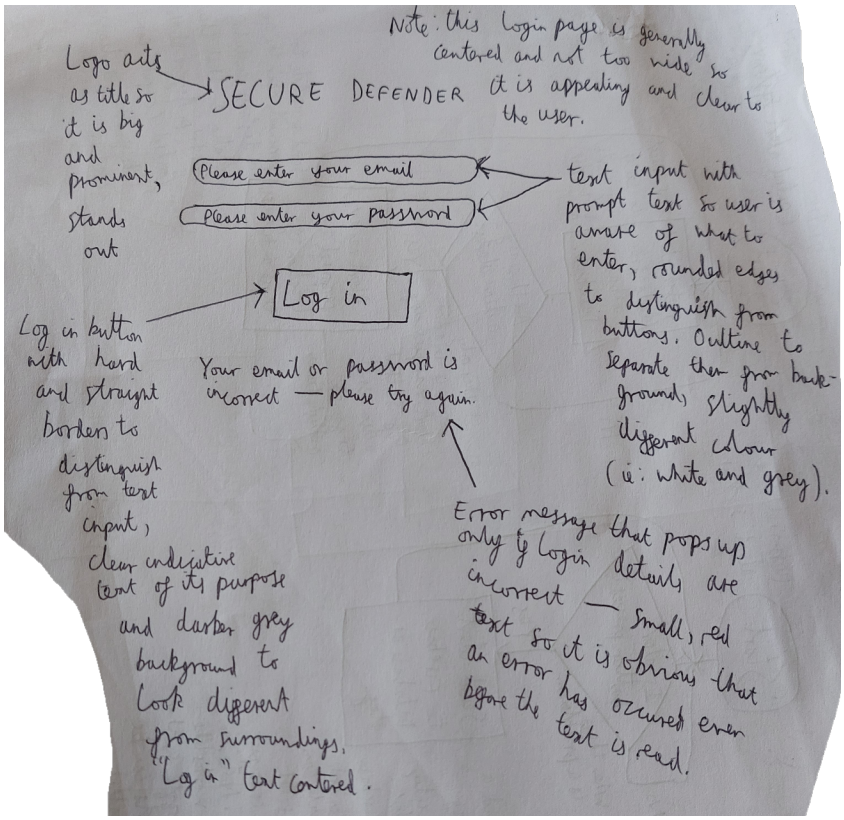
string because it needs to be customised for a user, while *isAlert* (on the inside of the update object, changed when an alert is received) is stored together with *message* as at this point it is already given that there is an active update, hence fetching both of these is necessary. *isUpdate* and *isAlert* are boolean values as they are only to indicate if something is or is not - they only need to take 2 values. The field *name* holds the full name of the user so that they can be identified by the central security staff easily - a suitable data type for storing a name is a string. The *alertPosition* object inside the *update* object is present so that if the update is an alert then the position of personnel in distress can be stored too. The purpose of the outer *isAlert* (first entry in the user object, changed when an alert is sent) is quite unique as it is there to indicate if the current user is in trouble - this is stored distinctly as the mobile app needs to be able to distinguish alerts of other users from alerts sent by itself (that particular instance).



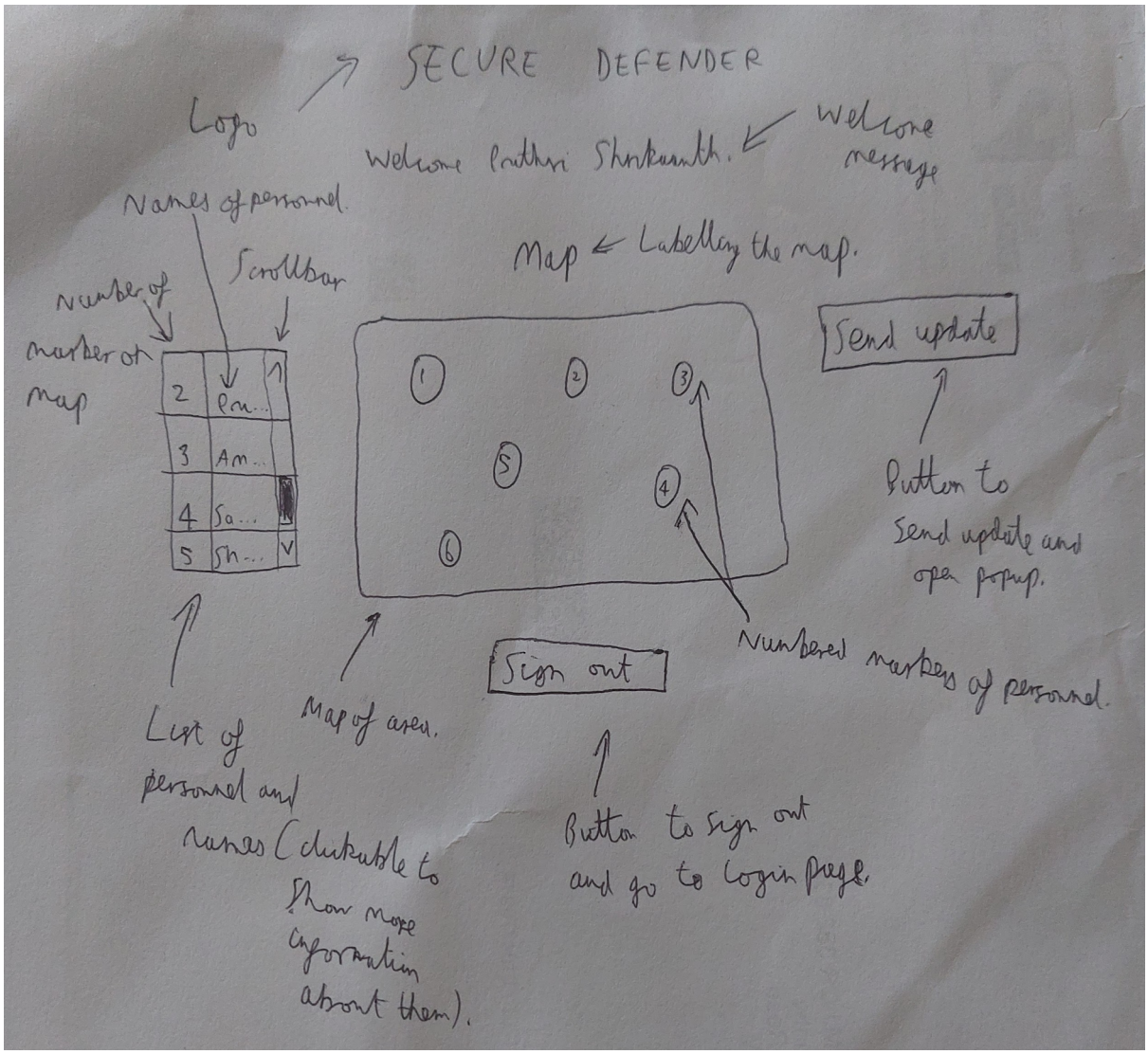
The rules above are rules for accessing the Firebase Realtime Database - since each user's data is stored in an object that is accessed by their user id, I set rules for each user object generally using *"$uid"*. I set *write* and *read* permissions to be the same as there is no need for a distinction since only the central computer will have to have absolute access to everything. Access is allowed if either one of two conditions are fulfilled: the user is accessing their own object (*$uid* === *auth.uid*) or the logged in user is a recognised administrator who would be manning the central computer (*$uid* === *'##########################'*). I have built the rules like this to ensure utmost security of the data such that even if the API key is leaked, user data cannot be read or written to/over by anyone who is not authorised to do so.

User Interface and Data Structures

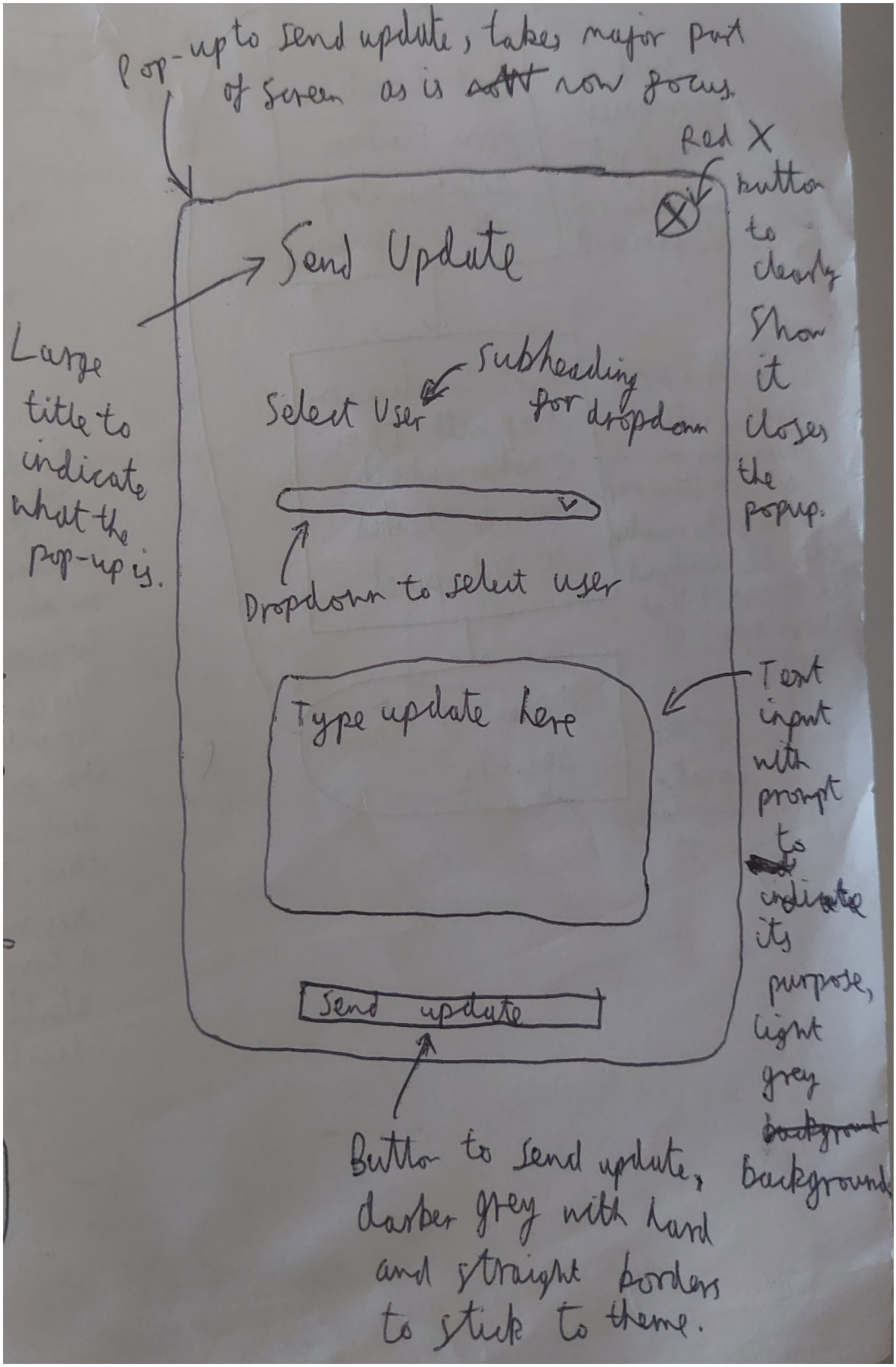
The general User Interface design for my software is intended to be simple to use - this is one of the key distinctions about my software. A number of the other solutions I found in my research had many features and were not very obvious to navigate - they may even require training for one to use them proficiently. I want to fix this issue by making my software much more self-explanatory and clear. One way I could do this is by suitably labelling features with what they do - for example, the map could be labelled as a “map of security personnel”. The toggle to send an alert should also be shown obviously as a toggle, for example with a slider. If it is mistaken for a button then this may cause users to disable their alerts before the central computer reads them. Another way I will make my software self-explanatory and clear is by using sensible and contrasting colour designs - for example, some shades of red and green don’t contrast well in some people’s eyes and green and yellow are also quite similar and hard to distinguish when overlaid. The reason for this is that the software should be accessible to people with colour defects and blindness too, although they are more likely to be working at the central computer than as a guard. Another way to use colour is to make my software look professional - I won’t include too many colours as that would make my software seem unorganised. I will also judiciously choose how to use colours; for example, an alert should be in red or even grey but never in green. This is because red is perceived as a sign of danger, while green is perceived as a sign to continue (much like traffic lights).



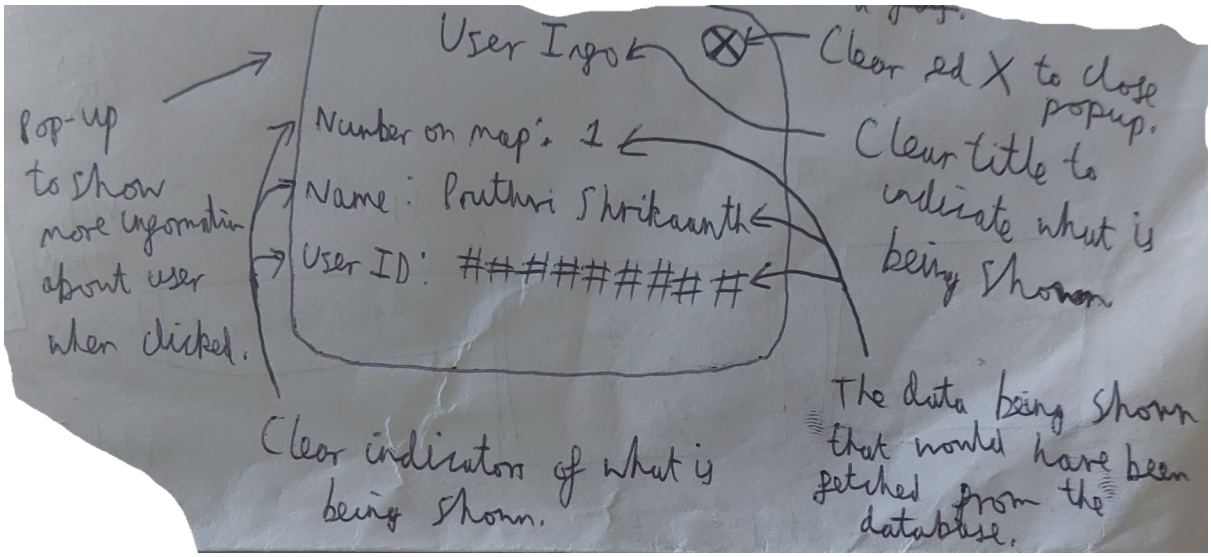
Login UI for both web and mobile apps.



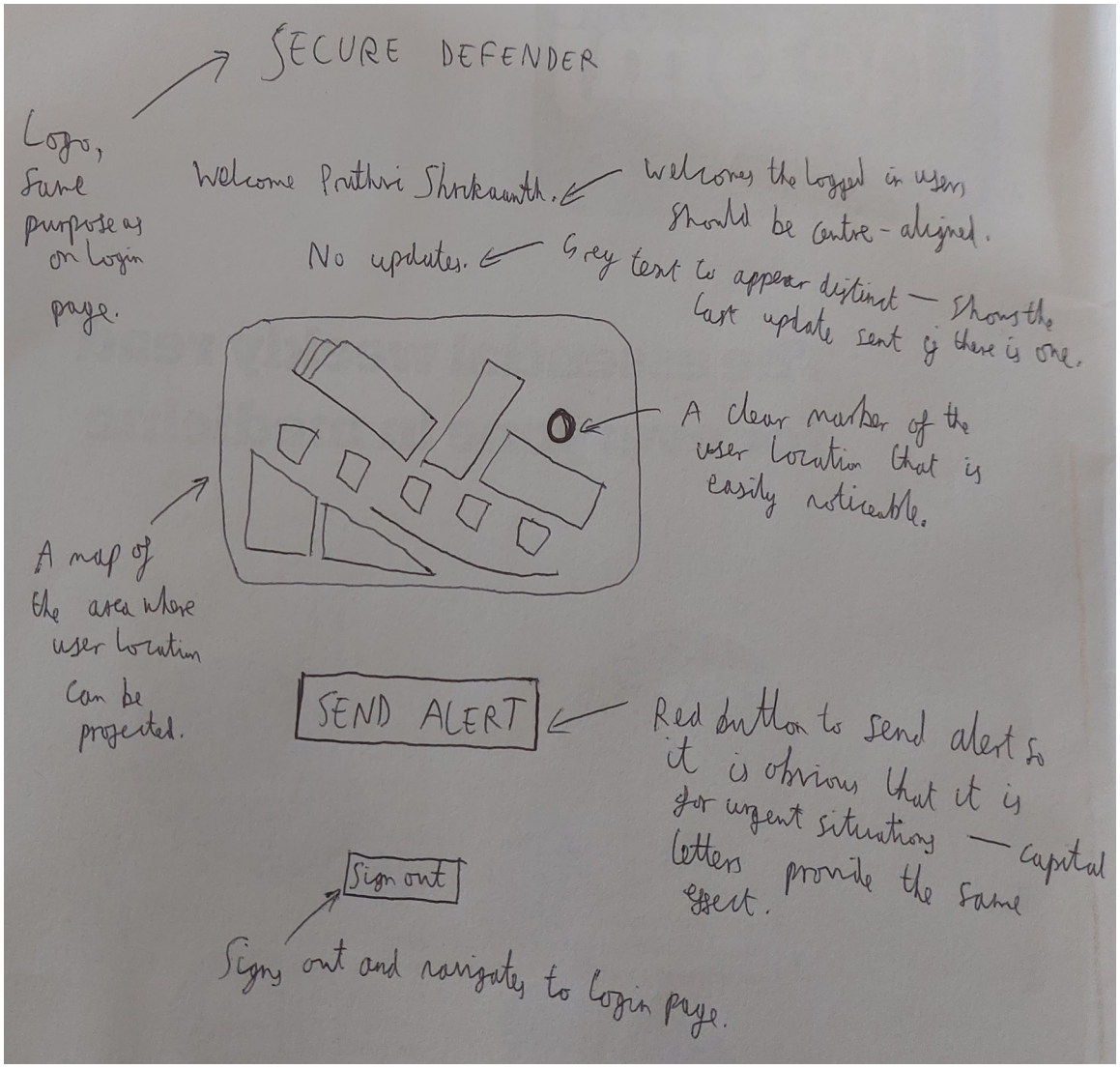
Main UI for web app.



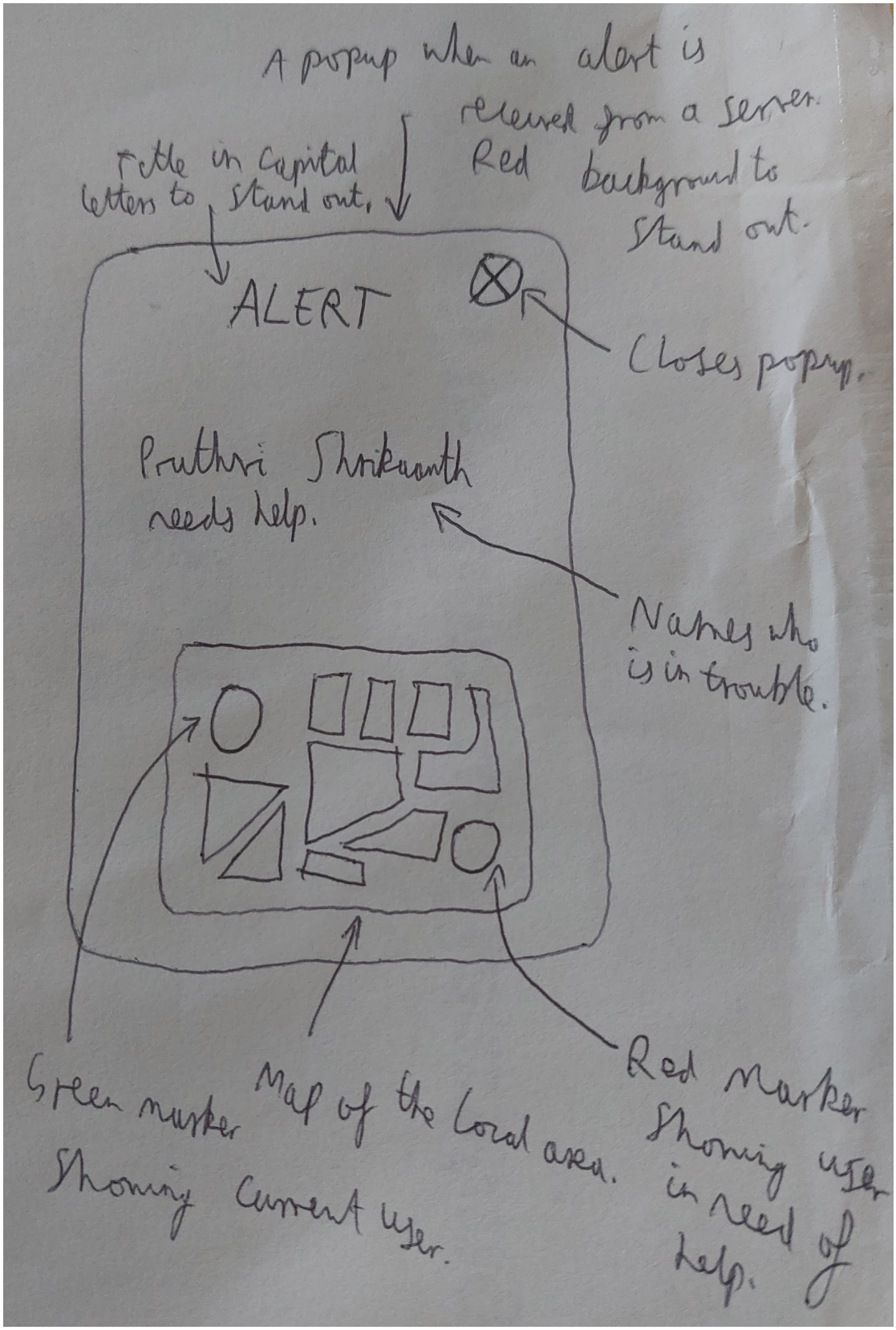
Send update popup on web app.



User info popup on web app.



Main UI on mobile app.



Alert received popup on mobile app.

Some common features:

- The popups are to be just that - they don't function as a separate page to navigate to and instead appear on the screen when triggered by a user action or change in the database - they always include a red button in the top-right corner containing an *X* as this is typically used to close windows and indicates to the inexperienced user that it is used to close the popup.
- The logo - it will appear on each separate page of navigation as a constant part of the general theme of my software - its look and feel will remain relatively consistent so it is easier to navigate and use. The logo itself will look professional and simplistic, clearly indicating its purpose and the name of the app.
- Text inputs will have slightly-rounded edges and prompt text inside them to indicate what is to be entered in them.
- Buttons will have straight edges to show them as distinct from text inputs and other features.
- Capital letters will be used only when showing something more important, such as an alert or the button to trigger one.
- The most common colours for background will be white and shades of grey - this is to keep the UI easy to read and simplistic.
- Red will only be used to indicate negative actions or signal something negative; this includes buttons to close popups and to send alerts as well as red text to indicate a failed login.
- Green is used minimally and is used to contrast from red - for example when showing the current user and the one in trouble on the map in the alert popup. The shades of green and red will be chosen to contrast from each other so that they don't confuse people who are less able to see colour.
- Other than red text, the vast majority of text will be black. This maintains simplicity and the theme of the software.
- Both apps will be generally centre-aligned so that they look organised and make efficient use of space, while keeping content in line with itself.
- Buttons, text inputs and other features will not be placed too close to each other nor will they be abnormally-sized: this is to ensure that nothing is accidentally pressed; this can cause a serious problem, especially if the alert button is unintentionally pressed and raises a false alarm (although this should cause minimal issues as this button will be designed to appear clearly as a toggle).

Input validation:

- Since Google Firebase handles input validation too, there is minimal need for me to secure my text inputs from any sort of code injection.
- However, it is convenient for the user if I add a few basic checks in, such as:
    - Ensuring that no text input is empty.
    - Ensuring that the email field for both login pages contains an @ and . symbol.

Data structures:

1. Login on both apps:
    a. Email address is a string as all email addresses can be stored in only this format.
    b. Password is also a string as it is to be entered by the user as a string of characters.
    c. The user information such as user ID and email address altogether will also be stored in an object, which is returned by Firebase Authentication when the user logs in - this will have uses such as accessing the user object in Firebase Realtime Database which is referenced by the user ID, which is stored in the user object.
2. Map rendering on web app:
    a. The map itself is a class that is created from the imported MapBoxGL library.
    b. Each marker on the map is stored in a list because the number of markers cannot be coded in - it is not constant between different runs.
    c. Each marker itself is an object containing an integer for its number, string for the name of that person and another string for their user ID - on top of this, the marker contains another object called position which contains the longitude and latitude of the marker, each as signed floating point numbers since the sign indicates the direction and a decimal number is enough to indicate specifically a location.
3. Update sending on web app:
    a. The pop-up is displayed in a Div tag.
    b. The list of pointers is used to generate a list of personnel (with each item as a string to be displayed and one chosen by the user), where the chosen person is also stored in another string.
    c. The update text will be stored as another string and the app will formulate a request to send to Firebase Realtime Database to send this update.
4. Alert handling on web app:
    a. The active alerts are detected and a string is composed containing the name of the user who sent it, which can be used as the update message for the alert.
    b. A function using Baudhayana's theorem (Pythagoras' theorem) is used to calculate the distance to the personnel closest to the one in distress, and these personnel are chosen to receive the alert.
    c. An object containing the alert text and location is sent, as well as the isAlert and isUpdate flags in Realtime Database being marked as true.
5. Map rendering on mobile app:

    a. The map itself will be rendered using the Google Maps Flutter package as it is officially supported by Google and has plentiful documentation. It also provides all of the functionality I need and renders very quickly.
    b. The user location object will be stored on its own and will contain only the latitude and longitude, since it does not need to identify it from other users. Longitude and latitude will again be stored as signed floating point numbers.
    c. In the case of an alert popup, the location of the user in trouble will also be stored as an object - however, this will also be stored separately rather than in a list, therefore it is easy to distinguish.
6. Alert sending on mobile app:
    a. The user's current location is sent as an object containing longitude and latitude (both signed floating point numbers) to Firebase Realtime Database.
7. Alert rendering on mobile app:
    a. Displayed as a pop-up Widget.
    b. As well as the map that is rendered with markers, a string containing the alert message is stored and displayed in the pop-up.

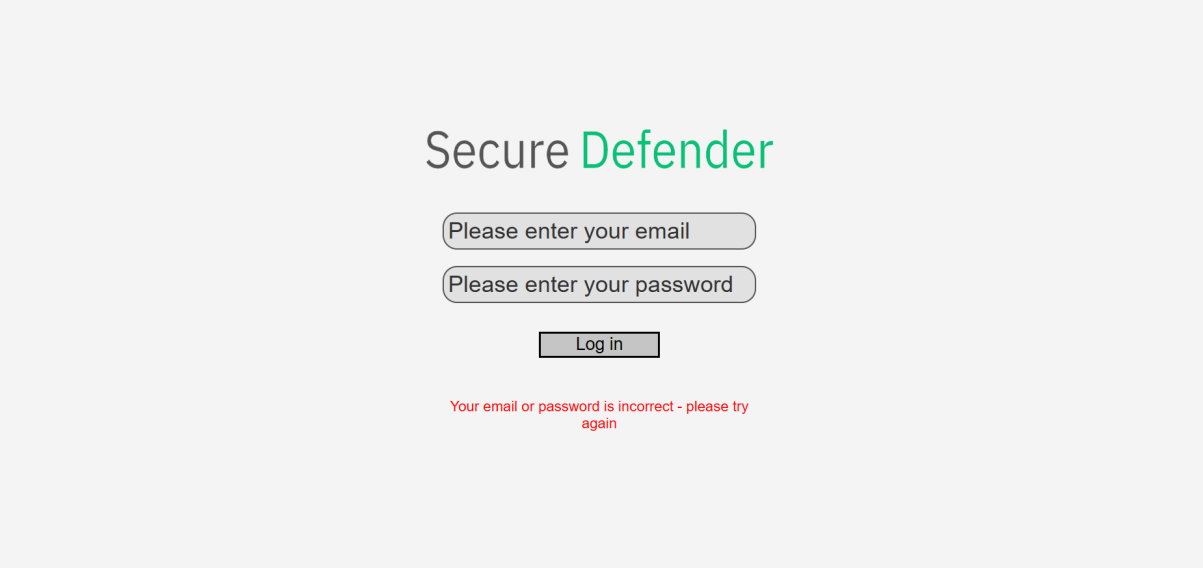
General Test Plan

- Iterative testing will be done on each function and module as it is being built - this is to ensure that the code is being continuously tested so that any errors are fixed as soon as they're introduced, as well as simplifying the search for the source of a particular problem.
- This will include entirely distinct tests for both apps, as well as separate tests for testing that the login system is working, that the map is rendering, that general updates are working properly and that updates are working properly. These are the main features of the app and will be tested as a whole as well as in part. Logging will be used to monitor what is contained in data structures.
- Terminal testing will be done on both apps as a whole to test against my success criteria - for this, I will make use of multiple devices to simulate a real-life security setup as well as logging in my code in case any significant errors occur and interrupt the test. I will only test against the final success criteria evaluatively when there are no significant errors hampering the use of my apps.
- More detailed test plans will be shown before the development of each part of my software and when development is complete so that I can take into account any changes in my design that may occur to fix errors or as a part of iterative development, designing my final tests accordingly.

Development

Web - Login

I started the Web version of the project by making the first page, called *login.html*, in HTML This was relatively simple, only adding a page title with a logo and specifying the general layout of the page. This page contains the logo of my project, two input fields for an email and password, a login button and an initially hidden message that is displayed when the login fails. I configured how it looks by using CSS in a file called *loginStyle.css*. I initially aligned everything to be central horizontally and vertically, then configured the sizes of everything on my page. I did this in a way that is relative to the size of the webpage window - this is to ensure that it renders correctly on a number of different computer browsers. I did not optimise it for mobile devices as this part of the application is meant to be used on a desktop. I spaced all of the elements on my page so they are distinct, and styled them with different colours, shapes and borders such that they set a style for the rest of my app. I used sensible colours to indicate neutral and error messages, while keeping prompts for the button and input fields clear and indicative of their functionality and purpose. My login page looks almost identical to how I wanted it to appear in my plan.



I used Javascript in tandem with the Firebase Authentication Web API to integrate my app with my login system, which is directly linked to my database that I will use for this app. In the script *login.js*, I set up Firebase Authentication and then wrote a function to sign in. I then bound this to the login button. I also set up a condition where if the user was already logged in then I would navigate them to the main page immediately. This was when I ran into a problem and had to make a decision - I would have had to import and configure the app over and over again for different scripts if I had kept my app structure like this. I therefore decided to create another file called *setup.js* where I moved the Firebase initialisation so that it only has to be done once. I then imported the object *app* in *login.js*, which contained the initialised Firebase configuration object. I also initialised Firebase Authentication into a variable

called *auth*. I used a variable *signInButton* to reference the sign in button from *login.html*, then added a *keydown* event listener to call the *signIn* function. This is an asynchronous function which first puts the entered email and password into variables of the same names, then attempts to sign in, passing *auth*, *email* and *password*. On success, this returns an object *response* which has a child **user** that is stored in local storage under the same name. The page URL is then changed to navigate to *main.html*. If the login fails, I fetch the error message element into a variable called *errorMessage* and then set *errorMessage.style.display* to *block*. This makes the error message visible to indicate that a problem has occurred with logging in. I also implemented an if condition which checks if an object of the name *user* exists in local storage. If so, it navigates the user to *main.html*.

In *setup.js*, I imported *initializeApp* from Firebase and set a constant called *firebaseConfig* which contained details about how to access my authentication system and database. I then exported a constant *app* which contains an object returned by when *initializeApp* is called with *firebaseConfig* as an argument. I have kept comments in the file *loginStyle.css* as it would be a more appropriate and efficient way of describing its functionality, and this code with its comments can be found in the code index at the end of this document.

To test the code functionality, I set out a test plan with a range of inputs. I will show my test table below.

Input email	**Input password**	**Expected result**	**Actual result**
[empty]	[empty]	Login failure and error message shown	Login failure and error message shown
Incorrect email	Incorrect password	Login failure and error message shown	Login failure and error message shown
Correct email	Correct password	Login success and navigation to main page	Login success and navigation to main page
[none] Reload main page		User stays logged in	User stays logged in
[none] Navigate to login page while still logged in		User is sent to main page	User is sent to main page

First, I opened the website and tried logging in with empty email and password fields.Then, I tried logging in with an incorrect email. Following this, I entered an incorrect password. I then logged in with the correct email and password and

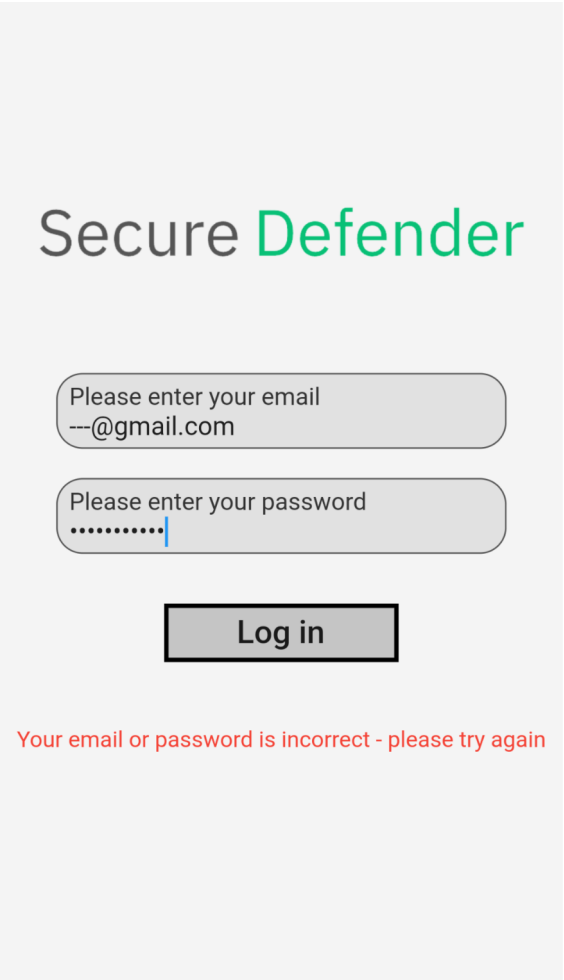
reloaded the page to ensure I stayed logged in. I then signed out. As all tests were passed, this section of code works successfully.

Android - Login

I started the Android version of my project by first setting up my development environment - this involved setting up a Flutter project using Dart and setting up my Firebase API for this project. After this was done, I was ready to start building my login page. I initially did this by setting up a navigation backbone in *main.dart*. I created a page in a file called *home.dart* that will function as the main page for my app. I then set up Firebase Authentication in *main.dart* to check if the user is signed in - if so, the app will start on the landing page, else it will do so on the login page.

I then created my actual login page. This was initially done as part of *main.dart*, but I later moved it into its own file called *loginpage.dart* to ensure that my code was modular and easy to read. To create the login page, I used a *StatefulWidget* which is a feature in Flutter that allows you to create dynamic components such as a login page, that do not stay the same all the time. I created a *LoginPage* class that extends this class. I then created two variables, *email* and *password* and set them as empty strings, to be controlled by text inputs. I also created a boolean value called *showErrorMessage* which toggles whether the sign in failure message shows on screen when incorrect login details are entered. I then created an asynchronous function called *signIn* that tries to sign in with the currently entered login details, setting the *email* and *password* variables to empty strings if successful (this is to ensure they are not stored unnecessarily for security reasons) and navigating to the main page. If signing in fails, *showErrorMessage* is set to *true* and I also used the *setState* method to update the page so that the error message is visible to the user.

After this, I set up the actual layout for the login page. I aligned everything horizontally and vertically centrally, first displaying the logo. Then I added in both text inputs and my sign in button. After testing that these worked, I researched how to control their styling attributes so that I could make them look as they did in my initial plan, and to the same theme as my Web version of the app. For the text inputs, I surrounded them with a container and removed their default borders so that I could add my own, more customisable borders. I also styled the hint text and internal styling with the use of an instance of the *InputDecoration* class. For my sign in button, I just created a container inside it and styled that container, which contained some centre-aligned text that I also set to an appropriate size. After all of these widgets, I put the error text inside an instance of the *Visibility* class so that I could control when it was visible with the *showErrorMessage* variable. I then set the size of the text to be small and set its colour as red. I finally added spacing between each of the elements with the use of padding so that the UI was clear and didn’t seem too cramped.



While finishing off my login page, I realised that its readability was not the best as it was quite a large file, especially considering that I repeated almost exactly the same code when creating both text inputs. This was when I decided to make another class within the same file called *CustomInput* that took one argument in its constructor called *isPassword*, that indicates whether the input is a password (if not then it is an email). I used this to control which variable to change when the text input was edited (*email* or *password*). I also used it to hide the inputted text (cover it with dots) when a password was being typed in. One problem that arose when creating this custom input was that it could not access the *email* and *password* variables directly as they only existed in an instance of the *LoginPage* class. I fixed this by making them static - this meant that I could access them from anywhere and that there was only one copy of them existent at any time - this was suitable as there would be only one instance of the *LoginPage* class in use at any given time.

I then noticed quite a peculiar error being thrown when I signed in on the app - it read “!navigator._debuglocked’: is not true.”. After some playing around and investigation, I figured that it was due to a number of reasons, that I fixed one at a time. In this process, I also fixed any warnings that I was getting at compile-time, which helped to prevent any future problems with my code. First, I made sure to

check whether the *BuildContext* was mounted, as I did some research on its purpose and found that if it changes while navigating between pages it can break functionality of the app. I then made sure to only navigate after the renderer had bound successfully to all components (ie: nothing was still loading on screen while the screen was changing). This fixed navigation when the keyboard was already down, but when the keyboard was still active and the sign in button was pressed, as it went down by default, this counted as a change in rendering and broke my navigation system. My fix for this was to invoke a method on the device that hid the keyboard before any navigation was carried out.

To test the code functionality, I set out a test plan with a range of inputs. I will show my test table below.

Input email		**Input password**	**Expected result**	**Actual result**
[empty]		[empty]	Login failure and error message shown	Login failure and error message shown
Incorrect email	or	Incorrect password	Login failure and error message shown	Login failure and error message shown
Correct email		Correct password	Login success and navigation to main page	Login success and navigation to main page
[none] Reopen app			User stays logged in	User stays logged in
[none] Log out			User is sent back to login page	User is sent back to login page

First, I opened the app and tried logging in with empty email and password fields.Then, I tried logging in with an incorrect email. Following this, I entered an incorrect password. I then logged in with the correct email and password and restarted the app to ensure I stayed logged in. I then signed out. As all tests were passed, this section of code works successfully.

Web - Main Page (design with basic functionality)

I started by building the page *main.html* visually through HTML and CSS. I initially included a dedicated stylesheet *mainStyle.css* and script *main.js* for the page, although both were almost empty at this point. I then added my icon and page title, moving on to the body. Here, I split my page into three sections and included my logo and a basic welcome message in the first. In the second one, I created three subsections for the table, map and update-sending button. I left these empty for now and finished my third section, which contained a sign-out button. I then copied the button style from *loginStyle.css* and applied it to both buttons on my page.

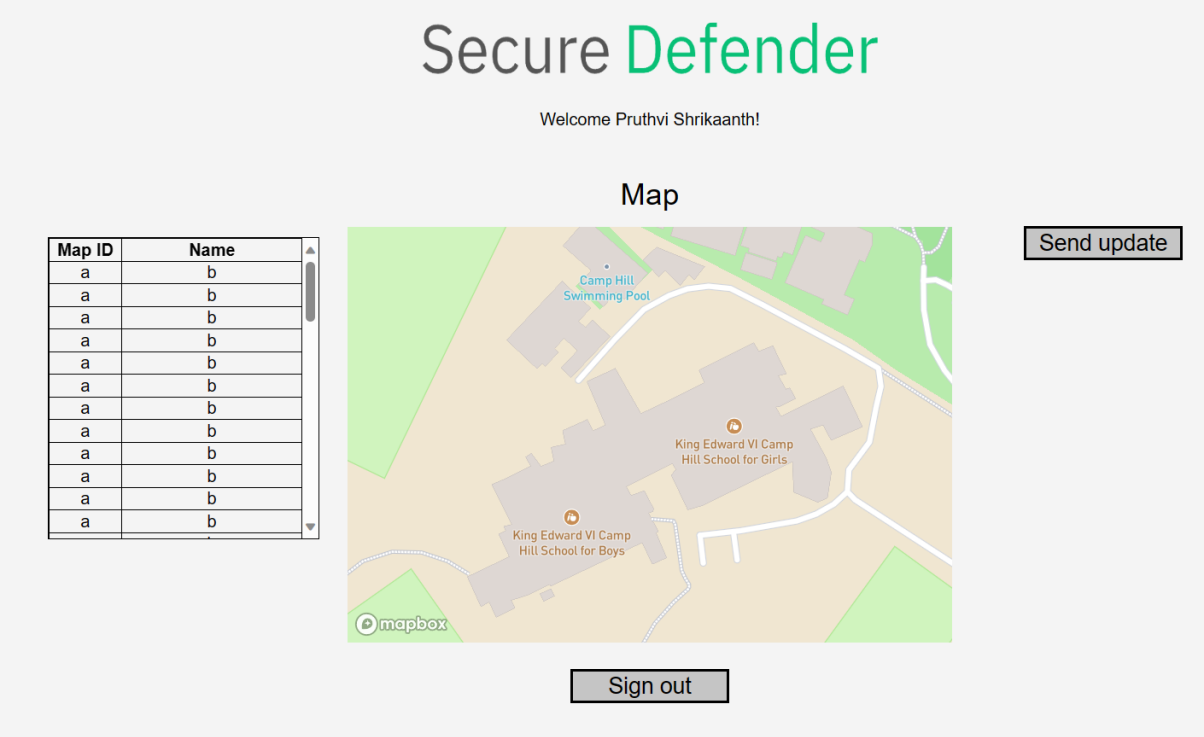
After this was complete, I customised the welcome message to show the name of the user signed in. After checking that the user was indeed logged in, I fetched the *user* object from local storage and extracted the user ID. I used this to access the user's dedicated folder in the Firebase Database; if the query was successful and the folder existed, I fetched the contents of the folder and extracted the name of the user. To display this in *main.html*, I selected the welcome message element by its ID and modified its contents to append the name of the user after "Welcome ".

Then, I added functionality to my sign-in button. I did this by writing some javascript code in *main.js*. I created a function called *signOut* which waited for Firebase to sign out the user and then continued to remove the *user* object from local storage if successful, then navigating the user to the *login.html* page. I then selected the sign-out button by its ID and added an event handler for the *click* event, binding it to the *signOut* function.

Now, I created the map with mapbox. To achieve this, I created another javascript file called *mapRender.js* and included it in *main.html* after creating the *div* for the map. In it, I loaded my access token for mapbox and created a new map. I created it with the following parameters: element ID *map* to change the *div* I created to a map, a particular style to make the map most useful for its purpose of security patrolling, a centre to ensure the map was focussed on a specific location, a zoom level to ensure that the map was sensibly in focus and a final parameter to disable an unnecessary info button that was popping up on the map. These parameters were enclosed in an object to conform to the argument requirements that mapbox uses. I finally sized the map accordingly in *mainStyle.css*.

I also set out the layout of my table using CSS. Initially, I created the label row and then I dumped a lot of testing data that filled both columns for the number of the person on the map and their name with "a" and "b". This was because I needed to test how the table would behave when data overflows its bounds. I ran into a problem here - when I initially set the *max-height* of the table to a finite amount in *mainStyle.css*, it did not limit the table from overflowing. This was something that I fixed by setting the table's *display* style attribute to *block* and setting only a height

rather than a maximum height. I then assigned the overflow behaviour to scroll so that the extra data was not cut off. I added to how useful the table would be by ensuring that the table column labels stayed on screen even when the user scrolled down. Then, I adjusted the width of the columns within the table so that the name column was wider (since name strings are usually longer than small numbers). I then ensured that a clear border was drawn between each item in the table. This led to a minor problem - since I was applying the border to not only the entire table but to each row and column, there were two sets of borders between items in the table. I fixed this by using a style attribute called *border-collapse*. This ensures that double-borders collapse into one single border that is much clearer and a more sensible choice.



There was minimal testing on this page, its only purpose being to test functionality that I had added; I did not add much in this stage as the main aim here was to set out a visual structure for the main UI.

Action	Expected result	Actual result
Reload page	User stays logged in and on this page with their name loading in the welcome message.	Exactly as expected.
Pressing sign out button	User is signed out and navigated to the login page.	Exactly as expected.
Scrolling through table	Title bar of the table stays in place while elements scroll through.	Exactly as expected.
Dragging map	Map should move with the mouse.	Exactly as expected.
Zooming into/out of map.	Map should change size.	Exactly as expected.

I first reloaded the webpage to ensure I stayed signed in. I then tried scrolling through the table. After this, I tried dragging the map and zooming into and out of it. Finally, I pressed the sign out button to ensure that I was sent to the login page. As all tests were passed, this section of code works successfully.

Android - Main Page (design with basic functionality)

To design the main page on the mobile application for my project, I worked entirely in *home.dart*. I started by inserting my logo with appropriate styling that is consistent with how my logo appears on the login page. I then spaced it appropriately so that it has some margin above and below it, then adjusted the layout of the page as a whole so that everything is aligned centrally vertically and horizontally. I followed this by creating two text placeholders - one for the welcome message and one for the update text.

After this, I created a sign out button and copied the styling from the login button in *loginpage.dart*. I placed it after all other widgets. I bound this to a function called *signOut*, which signs out with *FirebaseAuth* and then navigates to *loginpage.dart*. I then created two variables to store the welcome message and update text so that it can be programmatically changed. I haven't worked on the updates system yet so the update string will be constant for now. To initially set the welcome message, I created a reference to the name field under the user ID in the database. I then created an asynchronous function called *setWelcomeMessage* which fetches a snapshot of this reference. I checked if the snapshot was successfully fetched, storing its value if that was the case. I then used string interpolation to edit the welcome message so that it included the user's name in it. Since the welcome message updates only once when the page is loading, I fetched the name of the user only once and did not need to make use of state to change what was displayed on the page.

Now, I set up the *google_maps_flutter* library and added my Google Maps API key to my Android manifest file so that I could successfully use this library. Between the update text and sign out button, I inserted a map, binding it to a map controller so that the user could interact with it (ie: scrolling, zooming, rotating). I set its initial position and zoom in the form of parameters. To ensure that the map didn't overflow the display, I wrapped it in a *SizedBox* and specified a finite height of 400 pixels to it. This was to ensure that the map was reasonably large while also not overflowing the display on most modern devices.

To fetch and plot the user's current location on the map, I used the *geolocator* library which is simply a tool to request location access and fetch device location. I used a stream to continuously listen for changes in the user location so I did not have to constantly fetch it multiple times and abuse the location service. I then stored the updated location in a position object that stored the longitude and latitude of the device. I ran into a problem with fetching the location of the user - I had insufficient permissions to access it, which I fixed by doing two things. The first was specifying that I needed access to the user's precise location in my Android manifest file.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

The next thing I did was to check if permission for location access had been granted by the user. If this was the case, I went ahead and fetched the user's location - else, I requested permission to do so and then proceeded. To allow this code to be called in my constructor, I split it into two functions - one asynchronous function called *locationUpdates* to ensure necessary permissions were granted to my app and another called *doLocationUpdate* where the actual location-fetching stream was run. I called *locationUpdates* in my constructor which in turn called *doLocationUpdate* after ensuring that necessary permissions were granted.

```
35    static Position? currentPosition; // variable to store current position
36    final LocationSettings locationSettings = const LocationSettings(accuracy: LocationAccuracy.best); // use best location accuracy
37
38    locationUpdates() async {
39      LocationPermission perms = await Geolocator.checkPermission(); // check if location can be fetched
40      if (perms != LocationPermission.always && perms != LocationPermission.whileInUse) { // if not able to fetch location
41        perms = await Geolocator.requestPermission(); // request permission
42        doLocationUpdate(); // fetch location
43      } else {
44        doLocationUpdate(); // fetch location if permissions are already there
45      }
46    }
```

To plot the user's fetched location on my map, I used a type of marker called a circle - I created a circle with ID 0 (as it is the only circle on this map) centred around the user's fetched location. It centred around the map centre until the user's location was first fetched to ensure that a marker was always visible. I made its radius 2.5 metres to ensure that the maximum radius of the user's displayed position did not exceed our requirements and to show to a sensible accuracy the position of the user. I thinned the borders of the circle so that its fill shows more clearly and filled it in as green, contrary to what I chose in my design. I did this because the current user's location marker should be displayed in green in the alert popup and I want to keep my app's design consistent. On top of this, maintaining a solid fill colour for the circle aids its visibility on the map. At this point, my code was mostly functional but there was still a minor problem - the location would not update after being fetched for the first time. I knew that this was not a problem in the location being fetched itself as I had used a stream to do that - therefore, it was a problem caused by the page not being updated. I fixed this by inserting an empty *setState* function in the *doLocationUpdate* function after the location was updated. The *setState* function triggers a widget update in all changed widgets whenever called; the effect of adding it here was that the map markers were updated whenever the user's location was updated, effectively causing the circle to follow the user around the map.

```
47    doLocationUpdate() {
48      StreamSubscription<Position> positionStream = Geolocator.getPositionStream(locationSettings: locationSettings).listen((Position? position) { // get live location
49        _HomeState.currentPosition = position; // store position
50        setState(() {
51
52        }); // update map
53      });
54    }
55
56    _HomeState() { // constructor
57      setWelcomeMessage(); // update welcome message (add name into it)
58      locationUpdates(); // start live updating of user location
59    }
```

I tested all interactive aspects of this page and also repeated a couple of previous tests to ensure that the integration of this page with the whole app was not

negatively affected. There is still some functionality to be added that will be tested as part of other unit tests (ie: updates).

Action	**Expected result**	**Actual result**
Sign in button from from **loginpage.dart** pressed	After navigation to main page, user's name loads in welcome message and map loads.	Exactly as expected
App restart	User remains logged in with their name in welcome message and map successfully loads.	Exactly as expected.
Swipe on map	Map scrolls.	Exactly as expected.
Pinch on map	Map zooms in and out.	Exactly as expected
Rotate fingers on map	Map rotates.	Exactly as expected.
Physically move around	Location marker on map updates and moves too.	Exactly as expected.
Sign out button pressed	User is signed out and sent to **loginpage.dart**.	Exactly as expected.

I tested all features in exactly the same order as in my test plan. As all tests were passed, this section of code works successfully.

Location Rendering (Android and Web)

Minimal work needed to be done on my Android app for this feature as it only had to write the same location that was already fetched onto the database. I did this by creating a function *updateDatabaseLocation* with one parameter *position*, which is the new position fetched from the device Geolocator. I called this function in *doLocationUpdate* after updating the map in the app. It first created a reference to the position attribute of the user object in the database. It then updated the data at this location with *position*.

I then worked on the Web app, adding functionality into *mapRender.js*. I started by getting the database and its root reference, then initialising Firebase Authentication so that I could get the user's ID. I then set up my data structures - an empty object called *markers* that would store the markers for each user after fetching/updating their positions, an empty array called *userList* that would store the list of user IDs. To initially create markers I created a function *getList* that was to be called at the end of *mapRender.js*. I fetched all data from the root reference for the database and then converted it into value form with *snapshot.val()*. After this, I deleted the element of fetched data which had a key which was the currently logged in user's ID. This was to avoid trying to display the location of the admin user, which was unnecessary. I then filled *userList* with all of the keys of the fetched data, which I accessed with *Object.keys(data)*. I then set up a *for* loop to loop through each element in *userList* by index. It singled out their user ID, latitude, and longitude.

```
let getList = function() {
        get(rootRef).then((snapshot)=>{
                let data = snapshot.val();
                delete data[auth.currentUser.uid]; // don't attempt to display location of admin user
                userList = Object.keys(data);
                for(let i = 0; i < userList.length; i++) {
                        let userId = userList[i];
                        let lat = data[userId].position.latitude;
                        let long = data[userId].position.longitude;
```

I then passed these three variables and a fourth (**i**, the index) as parameters to a function called *createMarker* (this was done for each user). This function created a marker as a *div* element and styled it with appropriate height, width and colour (green), setting it up to align its contents in the centre both vertically and horizontally. I then created a *p* element to append as a child to the marker. Its text, **i**, was to indicate uniquely each marker on the map. After creating the marker, I attached it to a *mapboxgl* marker so that it functioned appropriately on the map. I then added the marker to the *markers* object under a key which was the user ID of the user whose location the marker would contain. I now set the longitude and latitude of the marker, then adding it to the map. Finally, I used an *onValue* function to listen for changes in the database for that particular user, re-fetching and setting the new longitude and latitude to the marker each time the user's object in the database changed.

To allow the administrator to identify the personnel uniquely, I filled in the table. To do this, I created an additional list called *nameList*. I Then fetched the name of each user along with their longitude and latitude, appending it to the list such that its position in the list is the same as the position of the user ID in *userList*. After looping through each user from the fetched data, I called a function called *fillTable*. In this function, I selected the table from the HTML document by its ID and looped through each name in *nameList* by its index. For each name, I created a *tr* element (table row) with two *td* elements (table data) to fill both columns. The first simply contained the index of the name in *nameList* as text, while the second contained the name. I added the class *mapID* to the first column and *mapName* to the second; this was to size each column appropriately in terms of width. After appending both *td* elements to my *tr* element, I set the *cursor* attribute of the style of the row to “pointer”. I also jumped momentarily to the function *createMarker* to make the marker appear clickable too. This makes the mouse cursor appear differently over the table row and marker such that it appears clickable, indicating clearly to the user that this is possible. I then appended the row to the table.

To allow the table rows and markers to be clickable, I added a *click* event to both that called a function *openInfoPopup*, passing parameters as *index*, *name* and *uid* (user ID). This function created a *div* called *popup* and appended it to the HTML body, changing the colour of the page background to grey it out such that it was clear that a popup was being displayed. I then set the style of the popup to keep its background consistent with my application theme and set its content alignment, height and width. I also made it fit with my application theme by rounding its borders and setting an appropriate border thickness. I split the popup into two sections - the title bar and the content block. In the title bar, I created a close button and set it to float on the right of the screen, then created the title itself which displays the text “User Info”. I created the button first as it had to float to the right of the content that would be created in its container afterwards. After styling the button and text, I added an event listener to the button that listened for the *click* event. When the button is clicked, the page’s background colour is changed to the normal colour and the popup is removed from the page, then deleted. In the content block, I placed some multiline text to display the number of the user on the map, the user’s name and the user ID. These were passed into this (*openInfoPopup*) function as parameters.

```
let contentBlock = document.createElement('div');
contentBlock.style.textAlign = 'left';
let textDiv = document.createElement('div');
textDiv.innerText = `Number on map: ${index}\nName: ${name}\nUser ID: ${uid}`;
textDiv.style.margin = '3vh';
textDiv.style.lineHeight = '4vh';
contentBlock.appendChild(textDiv);
```

To test this functionality, I loaded up the Android version of my application and moved those devices around to ensure their locations were updating on the

administrator application too. I then clicked on the table entries and markers themselves to ensure that popups were displaying correctly.

Action	**Expected result**	**Actual result**
Open **main.html**.	Markers appear on map and move around as personnel move around. Table fills with entries for each user.	As expected.
Click on row in table.	Background of page greys and popup appears displaying user's number on marker, user's name and user's ID.	As expected.
Click on marker on map.	Background of page greys and popup appears displaying user's number on marker, user's name and user's ID.	As expected.
Click on close button on popup.	Background of page sets back to normal and popup closes.	As expected.

All tests were done in the order specified in the table (the final test was done additionally after the second as it was necessary to continue testing). As all tests were passed, this section of code works successfully.

Alert and Update System - Android

Since these functionalities are interlinked and depend on each other (particularly alerts depending on the update system), I built these together and then tested both units. I started by building the mobile application functionality in its entirety. This consisted of three main parts - a listener to listen for updates, a setter for changing the alert status of the user on the database and an alert handler that opens the alert handling popup when an alert from another user is received. For the update listener, I set the default update message as “No updates” and created a function called *listenForUpdates* which I called in the constructor for the page class. This function used the *onValue* function from Firebase Database to listen for changes in the database in the *isUpdate* value. I checked if the new value was true and if it was, I proceeded to fetch a snapshot of *isAlert* (the database value inside *update*, not the one on the outside). I updated *updateMessage* to the new message that I also fetched and set the state to update the GUI. I then updated the database to set *isUpdate* to false again so that this function was not repeatedly called. Then, I added a condition to check whether the fetched value for *isAlert* was true.

```
listenForUpdates() {
  isUpdateRef.onValue.listen((DatabaseEvent event) async {
    final isUpdate = event.snapshot.value;
    if(isUpdate == true) {
      final isReceivedAlertSnapshot = await isReceivedAlertRef.get();
      final isReceivedAlert = isReceivedAlertSnapshot.value;
      final snapshot = await updateTextRef.get();
      setState(() {
        updateMessage = snapshot.value.toString();
      });
      if(isReceivedAlert==true) {
        handleReceivedAlert();
      }
      rootRef.update({
        "isUpdate": false
      });
    }
  });
}
```

This condition was added to define alert-handling behaviour and I called a function called *handleReceivedAlert* if the condition evaluated to true. In this function, I set default values for the latitude and longitude of the user in need of help that were to be later modified. I also created a parallel map centre and controller to make the map that I was going to create interactive but distinct from the first map. Then, I created and showed a popup with a title that said “ALERT” (in uppercase letters to show its importance) and an *IconButton* that showed an “X” to which closed the popup. I then created some text to show the updates message so that the user could

see the name of who was in trouble (which would be included in this message). After this, I created a *GoogleMap* and bound it to its controller and initial centre, setting its zoom to the same value of 17.5 as the other map used. To add markers, I again added two *circles* - one for the current user that functioned in the same way as in the previous map and the other for the user in distress, which functioned similarly but with a different set of coordinates that would be updated whenever the other user's location updated. To update it in the Android application, I called *onValue* on the alert position inside the *update* object of the database of the currently logged in user. This listener updated the variables storing the position of the user in distress whenever they changed on the database and set the state to update the map, using *setNewState* to distinguish setting the state of the popup from that of the main page.

```
alertPositionRef.onValue.listen((DatabaseEvent event) {
  var otherLong = event.snapshot.child('longitude').value;
  var otherLat = event.snapshot.child('latitude').value;
  otherUserLongitude = otherLong;
  otherUserLatitude = otherLat;
  setNewState(() {});
});
```

On the main page, to change the alert status of the current user on the database, I added a button between the map and the sign-out button. This button was initially labelled as "SEND ALERT" in uppercase letters to signify its importance and in the colour red to indicate that it was for emergency purposes. I set these two properties as variables to allow them to be changed when the code was running. I created this button in the same button style that my Android and Web applications were both built in to ensure that it was clearly a button and my applications were consistently designed. Whenever this button was pressed, I called a function called *toggleAlert*. This function was asynchronous to allow the current alert status (whether the current user was alerting, *isAlert* as the direct child of the user ID) to be fetched from the database before proceeding. After fetching this status, I created a variable *alertToPush* that stored its opposite (ie: if the status was false, *alertToPush* would be true as I wanted the alert to be toggled on and vice-versa). I also set the alert button colour to yellow-orange if an alert was being sent and back to red if it was being disabled. I used *setState* to update the button text and colour (the text would be set to "Alert on!" whenever alert sending was enabled) and then updated the database to write the new value of *isAlert* (which would be *alertToPush*).

```
toggleAlert() async {
  final snapshot = await isAlertRef.get();
  final snapVal = snapshot.value;
  bool alertToPush;
  if(snapVal==true) {
    alertToPush = false;
    setState(() {
      alertButtonColor = const Color(0xFFFF0000);
      alertButtonText = 'SEND ALERT';
    });
  } else {
    alertToPush = true;
    setState(() {
      alertButtonColor = const Color(0xFFFFCC00);
      alertButtonText = 'Alert on!';
    });
  }
  rootRef.update({
    'isAlert': alertToPush
  });

}
```

Alert and Update System - Web

I started this side of the alert and update system by changing the createMarker function. I used *onValue* to check for changes to the (outer) *isAlert* attribute of the database (all within the current user object). If *isAlert* was true, I called *handleAlert* with parameters for the user ID and their place in the *userList* array. If *isAlert* was false, I called *handleNoAlert* with a single parameter which was the user ID. In *handleAlert*, I set up some placeholder variables for the user ID of the closest user to the one alerting and the distance between these two users. I stored the current longitude and latitude of the alerting user to make comparisons with every other user in *userList* without fetching the position of the alerting user for each comparison (as comparisons are done quickly, it would be inefficient to do this). To store this, I fetched it from the object stored under the user ID in *markers*. To make comparisons, I compared the longitude and latitude of the alerting user with every other user. To determine the distance between the users, I used Baudhayana's theorem, from which the formula $a^2 + b^2 = c^2$ was derived. Although the Earth is round, this application is designed to work on a small enough scale that the curvature of the Earth can be ignored when calculating the distance between two users, since they would be sufficiently close to each other that it makes negligible difference to assume that they lie on the same plane. If the distance calculated between these users was smaller than the smallest distance stored already or if the smallest distance was zero (I created this outside the loop and initialised it as zero) then this distance was stored as the shortest distance and the user that the alerting user was compared to was stored as the closest user to the alerting user. I then realised that I needed a way to store which users were sending alerts, so I quickly jumped to the section of code when I was creating *userList* and also created an

object called *isAlertingList*. I then jumped back and created an object with the alerting user ID as the key and the user ID of the closest user as the child. I added this object into *isAlertingList*. I then updated the closest user's object in the database to send an update which had the name of the alerting user in it, stating that they were in trouble. I accessed the name of the alerting user by using their place in *userList* to access their name from *nameList*. I also marked to the closest user that an update was sent and that it was an alert. I then used an *onValue* listener, listening to the alerting user's location in the database to update the alerting user's location in the closest user's object in the database. This meant that the closest user on the client-side application could access the distressed user's location. This was the *handleAlert* function complete; I then built the *handleNoAlert* function. This sent a new update to the closest user of the passed user ID that changed the update text to "No updates.", marked that a new update has been sent and that it is not an alert. Finally, it deleted the previously alerting user from *isAlertingList*.

To send text updates, I created and called a function called *setupUpdateSender* at the end of the *fillTable* function outside of the loop. This added an event listener to the update sending button, which would call another function called *openUpdatePopup* when clicked. To make this button appear clickable, I changed the cursor style when this button was hovered over by adding the line "cursor: pointer;" inside "button { … }" in *mainStyle.css*. In *openUpdatePopup*, I changed the background colour of the main page to grey it out so it was clear that a popup was being displayed. I then created the popup and layered it on top of everything else, also aligning it centrally in both axes. I set an appropriate height and width, changing its background colour and borders to suit the style of my application. I created a title bar which contained a close button and a title. I created the close button first so that I could float it to the right and style it as I did in the user info popup. I then created the title itself, which said "Send Update". I sized it up to appear like a title. Now, the title bar was complete and I created a content section. This started with a dropdown title that simply said "Select User". It was followed by a dropdown that visually displayed the names of the users and internally used their indices (places) in *nameList* to decide who was selected. After creating the dropdown, I created a *textarea* for multiline text and fixed its size, sizing it to appropriate dimensions and giving it a placeholder that read "Type update here". After this, I created a submit button that selected the user ID of the selected user from the dropdown list, which stored the index of the user in *nameList* which was consistent with where the user ID was stored in *userList*. After styling the submit button to be consistent with my application style and other buttons, I added a *click* event listener that updated the update message in the database for the selected user after fetching it from the *textarea* and marking that an update was sent (while also marking that it was not an alert). I then restored the background colour of the main page, removed and destroyed the popup.

Alert and Update System - Testing

Before testing my system's final stage, I had to go back to setting up my Firebase Database and change the permission configuration. This was because I was now trying to read from and write directly to the database, sometimes accessing it from its root than through the child user. This would not have worked with the current permission configuration as Firebase Database automatically assumes that a lack of permissions specified evaluates to false, so if I tried to access a user object through its parent (the root), the lack of defined permissions would mean that permission is denied. I found this out when I opened my project before I was able to run any tests on it, so initially all tests (which I have specified a little below) failed and the following error message was produced.

```
▶Uncaught (in promise) Error: Permission denied   Reference_impl.ts:821
    at Reference_impl.ts:821:16
```

To fix this, I added permission for the administrator user (the user with a particular user ID) to read and write to children through the root object. When doing this, I noticed a security vulnerability in my permissions due to a simple mistake I had made writing them - I was checking if the user object being accessed was that of the administrator rather than if the currently logged-in user was the administrator! I fixed this by replacing *$uid* with *auth.uid* in the second condition for *.read* and *.write* actions within a user object.

```
{
  "rules": {
    ".read": "auth.uid === '[redacted]'",
    ".write": "auth.uid === '[redacted]'",
    "$uid": {
      ".write": "$uid === auth.uid || auth.uid === '[redacted]'",
      ".read": "$uid === auth.uid || auth.uid === '[redacted]'"
    }
  }
}
```

To test my update and alert system, I not only needed the administrator side of the application running but also three clients - I needed the first as I had to check if updates were being sent and I needed the others to ensure that alerts were being processed correctly, including that they needed to be sent to the physically closest other user. The second client was necessary for alerts to be received, while the third was necessary to show that it was indeed the most proximate other user that was receiving alerts from the first, who was sending them. In all of my further testing, *User One* refers to the administrator while all other users are clients. My test tables are below - I split them accordingly so that the alerting and updating features were tested separately.

Update System Testing:

Action (done on desktop application)	**Expected Result (shown on desktop application unless otherwise specified)**	**Actual Result (Run 1 | Run 2)**
Send Update button pressed	Update sending popup pops up, background greyed out and popup displayed on top of the page.	Failed | As expected.
Send Update popup close button pressed	Popup closes, background colour of page reverted.	Failed | As expected.
User selected from dropdown in popup	Dropdown works interactively and a single user can be selected.	Failed | As expected.
Update message typed in *textarea*	The text updates in the *textarea*, which should not be resizable.	Failed | As expected.
Submit button pressed	Update sent to correct user with correct text (must automatically update on mobile application and be visible to the user), popup closes with page background colour reverted.	Failed | As expected.

All tests were done in the order specified in the table (the first test was repeated as I had to re-open the update sending popup). As all tests were now passed, this section of code works successfully.

Alert System Testing:

Action (done on mobile application)	**Expected Result (shown on mobile application)**	**Actual Result**
Send alert button pressed to enable alerting	Most proximate user is notified that the alerting user is in trouble, popup showing name of user in trouble and their location on a map is displayed and updated.	As expected.
Interact with map on received alert popup.	It should be scrollable, zoomable and rotatable.	As expected.
Close received alert popup.	The popup should close with no problems.	As expected.
Press button again to disable alerting	The previously notified user should receive an update automatically that says “No updates.” to indicate that the previously alerting user is no longer in trouble.	As expected.

All tests were done in the order specified in the table (the first test was repeated as I had to re-send the alert popup). As all tests were now passed, this section of code works successfully.

Evaluation

Full Application Testing

To test the whole application, I initially logged in to all devices that I was using - one laptop and three mobiles. I tested the updates system in action and also the alert system, not only to ensure that it functions correctly and as expected in a real world scenario but also responds quickly enough that the alert function is feasible. I observed the performance of my application against my success criteria and noted it down in a table below, as well as a test table for the functionality I am using itself. The test lasted around 10 minutes overall.

Examining performance of application against success criteria:

Success Criteria	**Satisfied?**
User locations update at least every 5 seconds.	Yes, whenever users had a relatively unblocked line of sight with GPS satellites.
User locations are accurate within a 5 metre radius.	Yes, whenever users had a relatively unblocked line of sight with GPS satellites.
Alerts are sent to the central system within 5 seconds (and processed and sent back).	Yes, whenever users remain connected to the Internet.

Testing primary functionality in real life scenario:

Action	Expected Result	Actual Result
Central system sends an update to User 3.	User 3 receives an update that is visible as update text.	As expected, except that this happened after the alert was sent.
User 2 sends an alert.	The closest user (intended to be User 4) receives the alert popup and finds User 2 by their location.	Not as expected but functioning correctly. User 3 ended up being closer and received the alert instead. User Interface slightly malfunctioned but functionality being tested performed as expected.

Analysis of Application Usability

My application has kept a consistent design and theme throughout that extends across both Web and Android platforms, with clear indication of functionality that enables users to use my application with minimal training without cluttering the User Interface. It has deviated minimally, if at all, from my proposed solution and my development roadmap has worked successfully.

Given that the GPS signal is strong and Internet connection is present (so that the central system can access devices' locations), live-location tracking functionality is working as expected and is very easy to interpret, with a clear map of numbered pins displayed with a table of numbers corresponding to users next to the map. This feature of the application is extremely usable, while the mobile-side application has the same advantage - it only needs to display the current user's location so it uses a single pin; this leaves no room for confusion as to whose location that pin represents. I ensured that this advantage was maintained when displaying an alert dialog by keeping the colour of the current user's pin the same - connotations of red with danger help to indicate that the red pin corresponds to the other user, whose name is displayed very clearly in the text above the map on the alert dialog, stating that they are in danger.

When an Internet connection is present (as my application relies on an Internet connection being present, since the lack of one caused problems in my final testing, I will elaborate more on this in the later section for improvements), alerts are sent to the nearest user almost immediately due to my efficient code that uses simple mathematical calculations to find the distance between two users. This, in combination with minimal bandwidth usage and the use of already fetched values means that the Web application can process alerts very quickly, allowing the Android application to function seamlessly as it hands over more intensive processing to the Web application. This server-client design addresses the problem of data security too, as the locations of other users are not unnecessarily sent to each other, minimising the chances of data interception in transit.

The modular design of my application makes it easy to modify functionality based on the needs of a client - for example, if an organisation wants a device to call Emergency Services automatically when the *Send Alert* button is pressed, I only need to modify the *toggleAlert* function in *home.dart* in my Android Application to also call a library function that rings Emergency Services if the alert is being enabled. Therefore, my application has been designed to work with other applications and functionality, as I do not need to write much extra code and only call pre-built functions from libraries of other applications to let them handle their functionality.

I exceeded the expectations of my proposed solution in terms of the updating system as I not only implemented it exactly as I intended to on the surface (the administrator

can manually type in instructions and send them to specific users), I also designed it to be independent from the alerting system. This meant that although I began to develop these two features together, I could test them separately. I did this because the backend handling of updates (sending and receiving) was quite simple and did not require much code - it would not increase the efficiency much to reuse such a small update function than retype it as part of a separate alert handling function.

Achievement of Success Criteria

My success criteria were by and large achieved in their whole. The personnels' locations update around every 5 seconds. However, when the GPS signal was lost, this sometimes slowed down. This slowdown was also due to me testing my application on some older devices, which used less refined GPS systems. This was something that I had little control over, but I managed to keep my code running efficiently so that the User Interface updated quickly and unnecessary data was not fetched from my database. This meant that I only fetched updated locations when they actually updated, but this was done immediately, leaving no time to waste. Another part that I had little control over was fetching the location from the device itself - the *Geolocator* class had control over this and I kept the time delay between location updates as small as possible. As long as the devices that my Android application runs on are not extremely old, this time delay should cause no problem to the functionality of my application.

My second success criterion was also satisfied, as I cross-checked the apparent location of users from the app with the actual location of the users and their devices. The location displayed was well within 5 metres of the actual location, and only drifted closer to 5 metres when the GPS signal was lost (or in the case of the desktop application, when devices lost Internet connection due to being outdoors, they had to wait to update the central system with their new locations). I also used circles of 2.5 metre radius to display the location of users so that the perceived locations could also be no more than 5 metres away from each other.

My third success criterion would have been strictly speaking, achieved as the central system did indeed receive, process and finish handling alerts in far below 5 seconds from the time the alerting device connected to the Internet. However, since the Wi-Fi connection of the alerting device was patchy, it took a longer duration of time for the alert to be processed. This, however, indicated a strength of my app - an alert cannot be cancelled before it finishes being processed and sent. This ensures that users who panic and spam the alerting button still have their alerts sent instead of cancelling them before they finish processing. The *Send Alert* button, when pressed, in my previous tests, sent alerts very quickly to the central system. However, the central system does not change the User Interface in any way to indicate that an alert has been received. This inhibits the administrator's ability to take action when

an alert is received and this means that my application does not utilise the fruits of meeting my third success criterion as much as it should.

A problem I encountered that I was not testing for was the User Interface breaking when a popup displayed for User Three - this device was older and therefore had a smaller screen, which my application was not designed for or previously tested on. However, all functionality was still intact and the display overflow would be invisible with debugging information turned off, not causing an issue if I were to release this application commercially.

Improvements and Meeting Remaining Success Criteria

- By designing my mobile application such that the measurements of items in the User Interface, I can ensure that the application functions correctly on devices with smaller screens, so that an error like the one on the device with User Three logged in is much less likely to occur.
- By ensuring that all devices are connected to the Internet using Mobile Networks rather than Wi-Fi, I can ensure that *all success criteria* are more strictly met since no device would lose connectivity and fail to transmit its current and accurate location or alert to the central system.
- The desktop application should also visually show and update if someone is in trouble - this could be done by fetching names dynamically from the alerting user list and showing them in a section on the UI. This allows the administrator to take further action when their personnel are in trouble in case it is necessary. Upon an organisation's request, I could set up a handler that automatically contacts authorities such as the Police when a user sends an alert - this is in line with my interviewee Mr Dipak Mistry's suggestion, as it is not necessarily safe for other personnel to rush to provide aid in a dangerous situation. It also furthers my achievement of *success criterion 3* as the administrator at the central computer (desktop application) can now contact the Police themselves. The alerting system could be further improved by also adding functionality to let users in trouble send messages to the administrator, enabling them to clarify the seriousness of any issues. Not only is it possible to use this to ensure the right type of support is provided in a problematic situation, it is also a good feature to implement for use even when not alerting to enable personnel to communicate with the administrator.
- The device that is initially sent an alert popup when a user near them is alerting should have the popup auto-close as well as having the update text change to indicate that the other user is no longer in distress. This makes it clearer that the user is no longer in need of help - this could be made clear as intentional functionality by introducing a small popup (the kind that isn't interactive, rather flashing on the screen for a few seconds, also called a *toast*) that says that the user is no longer in trouble. This can be done by

using the *onValue* function in the client applications to see when the alert is no longer live or when the update text is again set to “No Updates.”.

- When a user is alerting, the UI should show who is coming to help them on their map too so that they know how far away help is - this only needs an object for the other user’s position, which *alertPosition* could be reused for. This requires some modifications on the mobile application, such as to check if this user is alerting before showing the other circle. The other change in the mobile application would be to have another circle that can be added to the list of circles on the map or removed when needed. The Desktop application will need to have functionality to handle and send both user’s positions to each other; this can be added to our alert handling functionality. There is a possible functionality clash when an alerting user is marked as the closest to another user (such that they are alerting and being alerted), but in most situations, these users will be marked as each other’s closest so that they only show each other’s positions regardless of the fact that they are both alerting.
- To fix the unlikely but possible issue of clashes from different user’s positions being stored in the same *alertPosition*, I could create another object in the database - however, a better solution would be to use a more live-updating user position tracking system on the desktop application. This ensures that if the closest user to the alerting user changes then the now closest user is alerted. This need not cancel the alert that was sent to the previously closest user as there may be situations in which more than one other individual is needed to provide assistance.
- The alerting system could alert multiple users instead of just one, such that help is provided in sufficient volume such that it is enough but not overwhelming. The particular number of users that are alerted depends on the organisation that uses my application - for example, my interviewees’ organisations would need a different number of users alerted as there would be differing numbers of personnel and increased or decreased mobility in different physical situations (for example, a building with many different rooms that do not display on maps would result in personnel taking longer to provide help to someone in distress).
- In my database, for each user, I would change the name of the *isAlert* key on the outside of the *update* object to *isAlerting* to make it more indicative of its purpose and distinguish from *isAlert* inside the *update* object. This would require some code refactoring in my applications but only a little to ensure that my database references are correct, while massively increasing the readability of my code. It also means that it is more clear what I am referring to when talking about my code.

Maintenance - Ease and Issues

- Maintaining my application is generally pretty straightforward as I have commented all of my code in exceptional detail and anyone could understand what it does or add new functionality.
- I built my application modularly so that it is easy to slot new functionality in or improve and remove old functionality without disrupting any other functionality.
- I reused variables and code where possible to ensure there was little overflow of memory in my application - I also didn't overdo this code reduction as there comes a point where it is more costly to try and pack code when it need not be reused enough - for example, in my desktop application, there are only two popups, therefore I need not create any sort of class for a popup since this is unlikely to significantly reduce my code size or make it more readable.
- It is easy to allow multiple firms to use my application - as it is built in a way that I can add any additional functionality that they may want to use, I can also change database information for each firm to ensure that they cannot access each others' data and effectively isolate data that much not be shared across firms.
- One issue with maintaining my application is when scaling up users for my clients - due to concerns about security, I have not allowed the desktop user to create and remove user accounts from the system. This is something I would have to do each time a client wants to add or change personnel on their system. I can enable this by creating a separate interface solely for managing users - users of this app are likely to have more than one person managing security.

Limitations of my Application and how to deal with them

- One limitation of my application is how applicable it is in areas with bad coverage and obstacles in the line of sight between the device and GPS satellites. This causes communication to slow down as it takes much longer to send any data. However, I have built my application in a way that it minimises how much data needs to be transmitted over the Internet, so the slowdown of communication will not greatly inhibit my application's performance. The GPS line of sight problem is something I can adapt around - as geolocation technology improves, I can upgrade my application to use these better technologies so that my application still functions As expected. I could also use the device's in-built motion sensors to detect how much it has moved so that it functions independently of GPS. With current technology, however, this technology is not accurate enough for me to use in my application. Currently, my application is unsuitable for environments such as caves where GPS is

not easily accessible. The intended use of my application is in urban settings, so this will not cause a problem to the vast majority of users.

- Another limitation of my application is use within multi-level buildings. As the mapping system I'm using does not have an in-built feature for displaying the height of a location, users on multiple different floors may appear to be in the same location. This can be worked around by stationing personnel in different areas of buildings to reduce confusion about how many personnel are on each floor. Personnels' knowledge of who they will be stationed near is also instrumental in making this work. In future versions of my application, on client's request, I could use altitude that is also fetched with latitude and longitude to calculate which floor a user is on and colour-code their pin accordingly. For this, I would need some data on the internal structure of buildings that are to be guarded.
- Another limitation of my application is how usable it is on older devices - these have less accurate GPS systems built into them. This means that my application performs better on devices released in the last three years. Older devices are also of lower resolution, so screen items appear larger in relation to the screen size, possibly causing overflow errors. This limitation can be minimised by basing the size of the User Interface on the size of the screen.
- The loss of connection to the Internet is a problem that appeared in my final test since I connected all of my testing devices to Wifi. This limitation can be easily overcome by placing SIM cards in all devices used by personnel so that they are connected to the Internet at all times, either by Mobile Network or Wi-Fi.
- A final limitation is related to personnel behaviour - if there are many of them, it is easy to confuse personnel with each other due to the abundance of markers. This can be solved by efficiently placing personnel to provide adequate security; a good security setup will not have overly concentrated security anywhere as this would be redundant and inefficient. If personnel are running, their location may not be accurate within a 5 metre radius. However, it is unusual for personnel to run unless they are in distress, and in that case they could send an alert to share their location with someone else. This means that when the two meet, a concentration of personnel can indicate an issue in an area. The 5 metre radius of location accuracy becomes less significant where multiple personnel can define an area between them, so this problem is minimised.

Code Index

Web

login.html:

```html
<!DOCTYPE html>
<html>
    <head><!--Include the icon and stylesheet, then set the title of the page-->
        <link rel="icon" href="./assets/logo.ico">
        <link rel="stylesheet" href="./loginStyle.css">
        <title>Secure Defender - Log in</title>
    </head>
    <body>
        <script src="./login.js" type="module"></script><!--This script enables the page to function and login-->
        <img src="./assets/logo.png" /><!--This is the logo for my project-->
        <form id="loginForm"><!--This encloses the login functionality in a form-->
            <input class="textInput" type="email" name="email" placeholder="Please enter your email" id="email" /><!--A text input where the email address should be entered-->
            <br />
            <input class="textInput" type="password" name="password" placeholder="Please enter your password" id="password" /><!--Another text input where the password shpuld be entered but hidden when being typed-->
            <br />
            <input class="button" type="button" value="Log in" id="signIn" /><!--This button has a custom functionality to login to Firebase and return the user token or display the error message if the login fails - this works by calling a function in the login.js file -->
        </form>
        <p id="errorMessage">Your email or password is incorrect - please try again</p><!--This error message is initially hidden but shown if an error occurs when trying to log in-->
    </body>
</html>
```

login.js:

```javascript
import {app } from './setup.js'; // this import is for the setup data
import * as FirebaseAuth from "https://www.gstatic.com/firebasejs/10.4.0/firebase-auth.js";

const auth = FirebaseAuth.getAuth(); // set up Firebase Authentication to be able to login

if(localStorage.getItem('user')) { // if already logged in then navigate directly to the main page
    window.location.replace('./main.html');
};

let signIn = async function() { // defining the signIn function to be called when the button on the login page to sign in is pressed
    let email = document.getElementById('email').value;
    let password = document.getElementById('password').value; // just fetched both email and password
    await FirebaseAuth.signInWithEmailAndPassword(auth,email,password).then((response)=>{ // try signing in
        localStorage.setItem('user',JSON.stringify(response.user)); // store the user data in localStorage to stay signed in
        window.location.replace('./main.html');  // navigate to the main page
    }).catch((error)=>{ // if login fails
        let errorMessage = document.getElementById('errorMessage');
        errorMessage.style.display = 'block'; // show the error message
    });
};

let signInButton = document.getElementById('signIn');
signInButton.addEventListener('click',signIn); // execute the signIn function when the sign in button is clicked
let loginForm = document.getElementById('loginForm');
loginForm.addEventListener('keydown',(event)=>{
    if(event.key=='Enter') {
        signIn(); // also try to sign in if the Enter key is pressed while any elements such as text inputs are highlighted (ie: when the user finishes typing their password and presses Enter)
    };
});
```

loginStyle.css:

```css
body { /* Align everything to be central, set a default set of fonts to be used, set a minimum layout height and the background colour */
    max-width: max-content;
    margin: auto;
    display: flex;
    flex-direction: column;
    justify-content: center;
    min-height: 100vh;
    text-align: center;
    font-family: Arial, Helvetica, sans-serif;
    background-color: #f4f4f4;
}

img { /* Appropriately size the logo and space it away from the rest of the elements */
    margin-bottom: 30px;
    width: 30vw;
    height: 10vh;
}

input { /* Some initial spacing between the form elements */
    margin: 10px;
}

.textInput { /* Class textInput includes the two input fields and will space them appropriately, sizing them and colouring them to a unique style that is easy to visually understand */
    padding: 0px 5px 0px 5px;
    width: 25vw;
    height: 6vh;
    font-size: 4vh;
    border-radius: 2.5vh;
    border-style: solid;
    border-color: #555555;
    background-color: #e1e1e1;
}

::placeholder { /* Changes the colour of the placeholder text in the text inputs */
    color: #333333;
}
```

```css

.button { /* Spaces and sizes the submit button, giving it a unique style and colouring it to be indicative of its purpose */
    margin: 25px;
    height: 4.5vh;
    width: 10vw;
    font-size: 3vh;
    color: #000000;
    border-radius: 0px;
    border-width: 3px;
    border-style: solid;
    border-color: black;
    background-color: #c5c5c5;
    cursor: pointer;
}

p { /* Align the error message and wrap it onto the next line if it is too wide, size it appropriately, colour it red and initially hide it */
    margin-top: 25px;
    max-width: 25vw;
    align-self: center;
    font-size: 2.5vh;
    color: red;
    display: none;
}
```

main.html:

```html
<!DOCTYPE html>
<html>
    <head><!--Include the icon and stylesheet, then set the title of the page-->
        <link rel="icon" href="./assets/logo.ico">
        <link rel="stylesheet" href="./mainStyle.css">
        <script src='https://api.mapbox.com/mapbox-gl-js/v3.0.0/mapbox-gl.js'></script><!--Include mapbox api-->
        <link href='https://api.mapbox.com/mapbox-gl-js/v3.0.0/mapbox-gl.css' rel='stylesheet' /><!--Include mapbox styling-->
        <title>Secure Defender</title>
    </head>
    <body id="body">
        <script src="main.js" type="module"></script><!--scripts for this page-->
        <div id="topBar">
            <img id="logo" src="./assets/logo.png" /><!--This is the logo for my project-->
            <p id="welcomeMessage">Welcome!</p><!--Welcome message for user which will be modified by script-->
        </div>
        <div id="mainBody">
            <div id="personnelList">
                <table id="userTable"><!--A table of personnel that are involved in patrols-->
                    <tr id="titleRow">
                        <th class="titleRow mapID">Map ID</th><!--The number of the person on the map-->
                        <th class="titleRow mapName">Name</th><!--The name of the person whose number is such-->
                    </tr>
                </table>
            </div>
            <div id="mapContainer">
                <p id="mapTitle">Map</p><!--Title labelling map-->
                <div id="map"></div><!--Will be modified in script to show map-->
            </div>
            <script src='./mapRender.js' type="module"></script><!--Script to render map-->
            <div id="sendUpdate">
                <input class="button" type="button" value="Send update" id="sendUpdateButton" /><!--Button to send update-->
            </div>
        </div>
        <div id="bottomBar">
            <input class="button" type="button" value="Sign out" id="signOutButton" /><!--Button to sign out-->
        </div>
    </body>
</html>
```

main.js:

```javascript
import { app } from './setup.js'; // this import is for the setup data
import { getDatabase, ref, get } from "https://www.gstatic.com/firebasejs/10.4.0/firebase-database.js"; // import functions needed to work with database
import * as FirebaseAuth from "https://www.gstatic.com/firebasejs/10.4.0/firebase-auth.js";

if(!localStorage.getItem('user')) { // if user is not logged in
    window.location.replace('./login.html'); // take them to login page
} else { // contiue execution of code

    const auth = FirebaseAuth.getAuth(); // setup Firebase Authentication

    let user = JSON.parse(localStorage.getItem('user')); // fetch and store user object
    let uid = user.uid; // isolate user id
    let db = getDatabase(); // set up database
    const userRef = ref(db,uid); // get reference to user's folder within database
    get(userRef).then((snapshot)=>{ // get user data stored in database once
        if(snapshot.exists()) { // if user data was successfully found
            let name = snapshot.val().name; // isolate name of user
            let welcomeMessage = document.getElementById('welcomeMessage'); // access welcome message from main HTML
            welcomeMessage.innerText = `${welcomeMessage.innerText.substring(0,welcomeMessage.innerText.length-1)} ${name}!`; //  modify it to include user's name
        } else { // if user data was not found
            return // do nothing
        }
    }).catch((error)=>{ // if request was unsuccessful
        return // do nothing
    });

    let signOut = async function() { // defining sign out function
        await FirebaseAuth.signOut(auth).then((response)=>{ // wait for signing out
            localStorage.removeItem('user'); // remove user object from local storage
            window.location.replace('./login.html'); // navigate to login page
        }).catch((error)=>{
            return // do nothing if failure to sign out
        });
    };

    let signOutButton = document.getElementById('signOutButton'); // select sign out button
    signOutButton.addEventListener('click',signOut); // associate clicking this button with signing out
};
```

mainStyle.css:

```css
body { /* Align everything to be central, set a default set of fonts to be used, set a minimum layout height and the background colour */
    max-width: max-content;
    margin: auto;
    display: flex;
    flex-direction: column;
    justify-content: center;
    min-height: 100vh;
    text-align: center;
    font-family: Arial, Helvetica, sans-serif;
    background-color: #f4f4f4;
}

#logo { /* Appropriately size the logo */
    width: 30vw;
    height: 10vh;
}

.button { /* Spaces and sizes the sign out button, giving it a unique style and colouring it to be indicative of its purpose */
    margin: 25px;
    height: 4.5vh;
    width: 10vw;
    font-size: 3vh;
    color: #000000;
    border-radius: 0px;
    border-width: 3px;
    border-style: solid;
    border-color: black;
    background-color: #c5c5c5;
    cursor: pointer;
}

#topBar { /* Setting layout of page */
    min-height: 15vh;
}

#mainBody { /* Set arrangement of items and layout of page */
    display: flex;
    flex-direction: row;
    justify-content: space-evenly;
    min-height: 50vh;
    min-width: 130vh;
}

#bottomBar { /* Setting layout of page */
    min-height: 10vh;
}

#map { /* Set height and width of map */
    width: 80vh;
    height: 55vh;
}

#personnelList { /* Set width of table container */
    width: 40vh;
}

#sendUpdate { /* Set width of button container to be the same as that of the table container to ensure map is centered */
    width: 40vh;
}

#mapTitle { /* Text styling for title of map called "Map" */
    margin-bottom: 2vh;
    font-size: 4vh;
}

#sendUpdate { /* push send update button below map title */
    margin-top: 7vh;
}

table, th, td { /* Set borders of table to show and merge into one (no double borders), stop text touching borders */
    border: 1px solid black;
    border-collapse: collapse;
    margin: 2px;
}

table { /* Set dimensions of table and enable scrolling when names fill up maximum height */
    margin-top: 12vh;
    display: block;
```

```
        height: 40vh;
        width: 36vh;
        overflow-y: scroll;
    }

    #titleRow { /* Ensure that table does not move out of place, also ensure that labels for columns stay visible */
        position: sticky;
        top: 0;
        table-layout: fixed;
    }

    .mapID { /* Set width of first column of table */
        width: 10vh;
    }

    .mapName { /* Set width of second column of table */
        width: 26vh;
    }
```

mapRender.js:

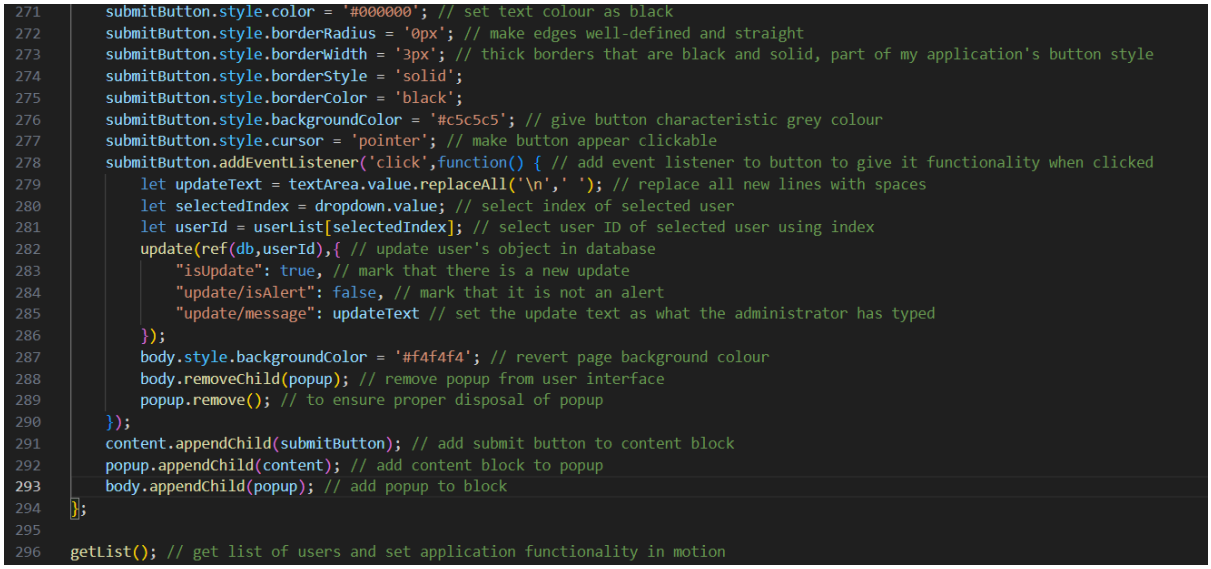
```
import { app } from './setup.js'; // this import is for the setup data
import { getDatabase, ref, get, onValue, update } from "https://www.gstatic.com/firebasejs/10.4.0/firebase-database.js"; // import functions needed to work with database
import * as FirebaseAuth from "https://www.gstatic.com/firebasejs/10.4.0/firebase-auth.js"; // import functions related to Firebase Authentication

mapboxgl.accessToken = 'pk.eyJ1IjoiYXBvb3J2ZWxvdXMiLCJhIjoiY2ttZnlyMDgzMzlwNTJ4a240cmEzcG0xNyJ9.-nSyL0Gy2nifDibXJg4fTA'; // access token for mapbox api
const map = new mapboxgl.Map({
    container: 'map', // container ID
    style: 'mapbox://styles/mapbox/streets-v12', // style URL
    center: [-1.9036, 52.4310], // starting position [lng, lat]
    zoom: 16.5, // starting zoom
    attributionControl: false // hide extra info button on map
});

const db = getDatabase(); // setup database
const rootRef = ref(db,'/'); // setup root reference for database
const auth = FirebaseAuth.getAuth(); // setup authentication

var markers = {}; // markers to be displayed on map, accessed by user ID as key
var userList = []; // list of user IDs in order initialised
var nameList = []; // list of user's names in same order as above
var isAlertingList = {}; // users who are marked as closest to users alerting (whose user IDs are used as keys)

let getList = function() { // function to get list of users
    get(rootRef).then((snapshot)=>{ // fetch all users in database
        let data = snapshot.val(); // process fetched data and organise into object
        delete data[auth.currentUser.uid]; // don't attempt to display location of admin user, remove it from list of users
        userList = Object.keys(data); // get list of user IDs from data object where user IDs were top-level children of the root
        for(let i = 0; i < userList.length; i++) { // loop through every user
            let userId = userList[i]; // isolate user ID
            let lat = data[userId].position.latitude; // isolate latitude of user
            let long = data[userId].position.longitude; // isolate longitude of user
            let name = data[userId].name; // isolate name of user
            nameList[i] = name; // add name of user into list of names
            createMarker(userId,lat,long,i); // create a marker for this user
        };
        fillTable(); // fill the table of users
    });
};

```

```
let createMarker = function(userId,lat,long,i) { // function to create a marker for a user
    let markerEl = document.createElement('div'); // create container use as marker
    markerEl.style.width = "3vh"; // style marker to appropriate size
    markerEl.style.height = "3vh";
    markerEl.style.borderRadius = '1.5vh'; // make marker stand out while also allowing fill colour to be visible
    markerEl.style.backgroundColor = 'lightgreen';
    markerEl.style.textAlign = 'center'; // align everything in the marker centrally in both vertical and horizontal directions
    markerEl.style.justifyContent = 'center';
    markerEl.style.display = 'flex';
    markerEl.style.alignItems = 'center';
    markerEl.style.cursor = 'pointer';
    let markerElText = document.createElement('p'); // create container for text inside marker
    markerElText.innerText = i; // assign a number to the user
    markerEl.appendChild(markerElText); // place text inside container
    markerEl.addEventListener('click',function() { // add functionality when marker is clicked
        openInfoPopup(i,nameList[i],userId); // open a popup to show more information about the user clicked on
    });
    markers[userId] = new mapboxgl.Marker({element:markerEl}); // use container as marker
    markers[userId].setLngLat([long,lat]); // set position of marker
    markers[userId].addTo(map); // add marker to map
    onValue(ref(db,userId),(snapshot)=>{ // add listener for user
        let data = snapshot.val(); // when user data changes in database, fetch new data
        let lati = data.position.latitude; // isolate updated latitude
        let longi = data.position.longitude; // isolate updated longitude
        markers[userId].setLngLat([longi,lati]); // set new position of marker for user
    });

    onValue(ref(db,`${userId}/isAlert`),(snapshot)=>{ // listen for users sending alerts
        let data = snapshot.val(); // process value stored in outer isAlert
        if(data==true) { // if alert is being sent
            handleAlert(userId,i); // handle the sent alert
        } else if(data==false && isAlertingList[userId]) { // if alert is being cancelled and user was previously alerting
            handleNoAlert(userId); // handle alert cancellation
        };
    });
};

let fillTable = function() { // function to fill table of users
    let table = document.getElementById('userTable'); // select table
    for(let i = 0; i < nameList.length; i++) { // loop through list of names
        let row = document.createElement('tr'); // create row for data entry
        let mapId = document.createElement('td'); // create item in first column
        mapId.innerText = i; // set its contents to the number of the user on the map
        mapId.classList.add('mapID'); // assign class to allow this column to be of appropriate width
        let mapName = document.createElement('td'); // create item in second column
        mapName.innerText = nameList[i]; // set its contents to the name of the user
        mapName.classList.add('mapName'); // assign class to allow this column to be of appropriate width
        row.appendChild(mapId); // add first column item into row
        row.appendChild(mapName); // add second column item into row
        row.style.cursor = 'pointer'; // make row appear clickable
        row.addEventListener('click',function() { // add event listener to show more information about user when the row is clicked
            openInfoPopup(i,nameList[i],userList[i]); // open popup to show more information about user
        });
        table.appendChild(row); // add row to table
    };
    setupUpdateSender(); // setup update sending functionality
}

let openInfoPopup = function(index,name,uid) { // function to open info popup
    let body = document.getElementById('body'); // select main page body
    body.style.backgroundColor = '#d4d4d4'; // grey out page background
    let popup = document.createElement('div'); // create container to use as popup
    popup.style.position = 'fixed'; // keep positon fixed
    popup.style.zIndex = 2; // place popup over other items
    popup.style.height = '22vh'; // set appropriate height and width
    popup.style.width = '60vh';
    popup.style.alignSelf = 'center'; // align itself centrally in both directions
    popup.style.justifySelf = 'center';
    popup.style.backgroundColor = '#f4f4f4'; // set its background to offwhite
    popup.style.borderRadius = '5vh'; // curve its corners
    let titleBar = document.createElement('div'); // create title bar
    let closeButton = document.createElement('div'); // create close button
    closeButton.style.float = 'right'; // move close button to right of title bar
    closeButton.style.width = '3vh'; // set width and height equally
    closeButton.style.height = '3vh';
    closeButton.style.borderRadius = '1.65vh'; // make button round
    closeButton.style.marginRight = '2vh'; // keep button within popup to avoid overflow
    closeButton.innerText = '\u2715'; // display an X inside the button
```

```
    closeButton.style.color = 'red'; // make the button red
    closeButton.style.fontWeight = '900'; // make the X thicker
    closeButton.style.border = '0.3vh solid red'; // thicken the button by making solid border in the same colour as the button
    closeButton.style.cursor = 'pointer'; // make close button appear clickable
    closeButton.addEventListener('click',function() { // add functionality when close button is clicked
        body.style.backgroundColor = '#f4f4f4'; // restore page background colour
        body.removeChild(popup); // remove popup from user interface
        popup.remove(); // to ensure proper disposal of popup internally and prevent memory leaks
    });
    titleBar.appendChild(closeButton); // add close button into title bar
    let title = document.createElement('div'); // create element for title itself
    title.innerText = 'User Info'; // add text to title
    title.style.margin = '2vh'; // allow title to visually stay inside popup
    title.style.marginRight = '0vh'; // push title away from left but take no extra space on the right to allow the close button to remain in place
    title.style.marginLeft = '5vh';
    title.style.fontSize = '3vh'; // set larger font size for title
    titleBar.appendChild(title); // add title to title bar
    popup.appendChild(titleBar); // add title bar to popup
    let contentBlock = document.createElement('div'); // add context block
    contentBlock.style.textAlign = 'left'; // align contents of content block to the left
    let textDiv = document.createElement('div'); // create a text container to display user information
    textDiv.innerText = `Number on map: ${index}\nName: ${name}\nUser ID: ${uid}`; // display map number, name and user ID of clicked user
    textDiv.style.margin = '3vh'; // detatch text from popup walls
    textDiv.style.lineHeight = '4vh'; // space out lines of text vertically
    contentBlock.appendChild(textDiv); // add text to content block
    popup.appendChild(contentBlock); // add content block to popup
    body.appendChild(popup); // add popup to page
}

let handleAlert = function(userId,i) { // function to handle alerts
    let closestUserId = userId; // initialise user ID of most proximate user
    let closestDistance = 0; // initialise distance to closest user
    let currentUserLngLat = markers[userId].getLngLat().toArray(); // convert latitude-longitude pair of alerting user's position to array of two items
    let long = currentUserLngLat[0]; // isolate longitude
    let lat = currentUserLngLat[1]; // isolate latitude
    for(let i = 0; i < userList.length; i++) { // loop through list of users
        let otherUserId = userList[i]; // get user ID
        if(otherUserId == userId) { // if the selected user is the alerting user then skip to next user
            continue;
        };
        let otherUserLngLat = markers[otherUserId].getLngLat().toArray(); // get latitude-longitude pair of selected user's position in array form
        let oLong = otherUserLngLat[0]; // isolate longitude
        let oLat = otherUserLngLat[1]; // isolate latitude
        let distance = Math.sqrt(((oLong-long)*(oLong-long))+((oLat-lat)*(oLat-lat))); // get straight line distance, works for small distances as angle change is negligible
        if(distance < closestDistance || closestUserId == userId) { // if this is first user being checked or if this is the closest user of the ones that have been checked
            closestUserId = otherUserId; // set selected user as closest
            closestDistance = distance; // set new shortest distance to compare to
        };
    };
    isAlertingList[userId] = closestUserId; // add alerting user : closest user pair to alerts
    update(ref(db,closestUserId),{ // update user object of closest user
        "update/isAlert": true, // mark that there is a new alert
        "update/message": `${nameList[i]} is in trouble!`, // include name of distressed user in update text
        "isUpdate": true // mark that there is a new update
    });
    onValue(ref(db,`${userId}/position`),(snapshot)=>{ // add listener for distressed user's position
        if(isAlertingList[userId]) { // if alert for this user is still active
            let data = snapshot.val(); // interpret data
            let lati = data.latitude; // isolate latitude
            let longi = data.longitude; // isolate longitude
            update(ref(db,`${closestUserId}/update/alertPosition`),{ // update alertPosition in closest user's user object in database
                "latitude": lati, // update latitude
                "longitude": longi // update longitude
            });
        };
    });
};

let handleNoAlert = function(userId) { // function to handle cancelled alert
    update(ref(db,isAlertingList[userId]),{ // update database to set inner isAlert of other user as false, reset their update text and mark that a new update has been sent
        "update/isAlert": false,
        "update/message": "No updates.",
        "isUpdate": true
    });
    delete isAlertingList[userId]; // remove this user pair entry from the list of active alerts
```

```
};

let setupUpdateSender = function() { // setup update sending mechanism
    let sendUpdateButton = document.getElementById('sendUpdateButton'); // select update sending button
    sendUpdateButton.addEventListener('click',function() { // add event listener to give functionality when button clicked
        openUpdatePopup(); // open an update sending popup
    });
};

let openUpdatePopup = function() { // function to create and open an update sending popup
    let body = document.getElementById('body'); // select main page body
    body.style.backgroundColor = '#d4d4d4'; // grey out page background
    let popup = document.createElement('div'); // create popup
    popup.style.position = 'fixed'; // keep popup in place
    popup.style.zIndex = 2; // place popup on top of page
    popup.style.height = '80vh'; // give appropriate height and width
    popup.style.width = '60vh';
    popup.style.alignSelf = 'center'; // align popup centrally in both directions
    popup.style.justifySelf = 'center';
    popup.style.backgroundColor = '#f4f4f4'; // give offwhite background colour
    popup.style.borderRadius = '5vh'; // curve corners of popup
    let titleBar = document.createElement('div'); // create title bar
    let closeButton = document.createElement('div'); // create close button
    closeButton.style.float = 'right'; // move button to right
    closeButton.style.width = '3vh'; // give same height and width
    closeButton.style.height = '3vh';
    closeButton.style.borderRadius = '1.65vh'; // make round
    closeButton.style.marginRight = '2vh'; // detatch from right wall of popup
    closeButton.innerText = '\u2715'; // show X in button
    closeButton.style.color = 'red'; // make button red
    closeButton.style.fontWeight = '900'; // thicken X
    closeButton.style.border = '0.3vh solid red'; // add border to thicken button
    closeButton.style.cursor = 'pointer'; // make button appear clickable
    closeButton.addEventListener('click',function() { // add functionality when close button is pressed
        body.style.backgroundColor = '#f4f4f4'; // revert page background colour
        body.removeChild(popup); // remove popup from page
        popup.remove(); // to ensure proper disposal of popup
    });
    titleBar.appendChild(closeButton); // add close button to title bar
```

```
    let title = document.createElement('div'); // create title
    title.innerText = 'Send Update'; // set title text
    title.style.margin = '2vh'; // detatch from top wall
    title.style.marginRight = '0vh'; // take up no extra space on the right
    title.style.marginLeft = '5vh'; // detatch from left wall
    title.style.fontSize = '3vh'; // make title text stand out by being bigger
    titleBar.appendChild(title); // add title to title bar
    titleBar.style.marginBottom = '5vh'; // space content below away from title bar
    popup.appendChild(titleBar); // add title bar to popup
    let content = document.createElement('div'); // create content block
    let dropdownTitle = document.createElement('p'); // add label for dropdown
    dropdownTitle.innerHTML = 'Select User'; // set label text
    content.appendChild(dropdownTitle); // add dropdown label to content block
    let dropdown = document.createElement('select'); // create dropdown
    dropdown.style.backgroundColor = '#ffffff' // set dropdown background colour to white
    dropdown.style.width = '40vh'; // make dropdown wide enough to fit names
    dropdown.style.marginBottom = '5vh'; // give dropdown some space below it
    for(let i = 0; i < nameList.length; i++) { // loop through list of names
        let dropdownItem = document.createElement('option'); // add entry for name
        dropdownItem.setAttribute('value',i); // give the dropdown's internal ID for this entry as the user's ID on the map
        dropdownItem.innerHTML = nameList[i]; // set the name of the user as the displayed text for the entry
        dropdown.appendChild(dropdownItem); // add entry to the dropdown
    };
    content.appendChild(dropdown); // add dropdown to content block
    let textArea = document.createElement('textarea'); // create multiline text input
    textArea.style.width = '40vh'; // set large area to input an update
    textArea.style.height = '40vh';
    textArea.style.resize = 'none'; // don't allow the size of the text input to be changed
    textArea.style.borderWidth = '0.4vh'; // make borders mildly thick
    textArea.style.borderRadius = '2vh'; // curve borders a little
    textArea.style.padding = '1vh'; // detatch typed text from walls of text area
    textArea.style.marginBottom = '5vh'; // add space below text area
    textArea.setAttribute('placeholder','Type update here'); // add hint text to indicate what the input is for
    content.appendChild(textArea); // add text area to content block
    let submitButton = document.createElement('button'); // create button to confirm sending update to user
    submitButton.innerHTML = 'Send update'; // set button text
    submitButton.style.height = '4.5vh'; // set sensible height and width that depends on screen dimensions
    submitButton.style.width = '10vw';
    submitButton.style.fontSize = '3vh'; // set larger text to indicate importance of button
```



```
    submitButton.style.color = '#000000'; // set text colour as black
    submitButton.style.borderRadius = '0px'; // make edges well-defined and straight
    submitButton.style.borderWidth = '3px'; // thick borders that are black and solid, part of my application's button style
    submitButton.style.borderStyle = 'solid';
    submitButton.style.borderColor = 'black';
    submitButton.style.backgroundColor = '#c5c5c5'; // give button characteristic grey colour
    submitButton.style.cursor = 'pointer'; // make button appear clickable
    submitButton.addEventListener('click',function() { // add event listener to button to give it functionality when clicked
        let updateText = textArea.value.replaceAll('\n',' '); // replace all new lines with spaces
        let selectedIndex = dropdown.value; // select index of selected user
        let userId = userList[selectedIndex]; // select user ID of selected user using index
        update(ref(db,userId),{ // update user's object in database
            "isUpdate": true, // mark that there is a new update
            "update/isAlert": false, // mark that it is not an alert
            "update/message": updateText // set the update text as what the administrator has typed
        });
        body.style.backgroundColor = '#f4f4f4'; // revert page background colour
        body.removeChild(popup); // remove popup from user interface
        popup.remove(); // to ensure proper disposal of popup
    });
    content.appendChild(submitButton); // add submit button to content block
    popup.appendChild(content); // add content block to popup
    body.appendChild(popup); // add popup to block
};

getList(); // get list of users and set application functionality in motion
```

setup.js is not included due to it containing sensitive information about my database access credentials. That is the only purpose this file serves.

Android

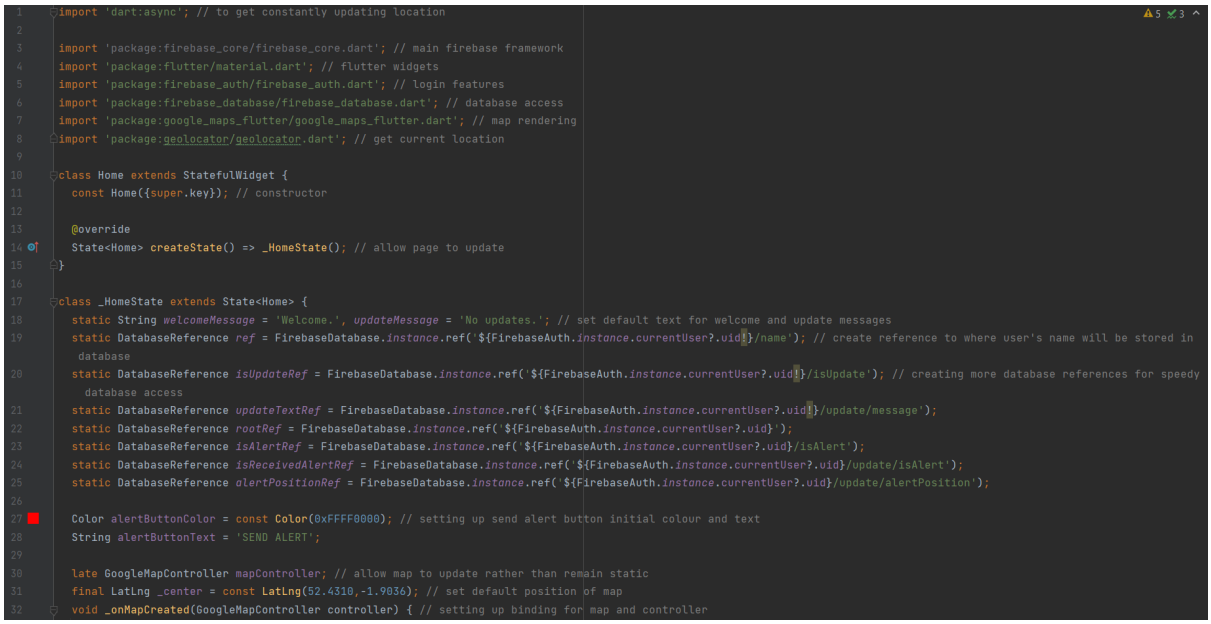
main.dart:

```dart
import 'package:flutter/material.dart'; // basic flutter widgets
import 'package:firebase_core/firebase_core.dart'; // setting up firebase
import 'firebase_options.dart'; // my firebase configuration
import 'package:firebase_auth/firebase_auth.dart'; // setting up firebase authentication
import 'home.dart'; // landing page after login
import 'loginpage.dart'; // login page

String initialRoute = 'login'; // to navigate to login page by default

void main() async {
  WidgetsFlutterBinding.ensureInitialized(); // ensure widgets are ready before setting up firebase
  await Firebase.initializeApp(
      options: DefaultFirebaseOptions.currentPlatform
  ); // set up firebase connection
  FirebaseAuth.instance
      .authStateChanges() // check if user is already signed in
      .listen((User? user) {
    if (user != null) {
      initialRoute = 'home'; // set main page as landing page if user is signed in
    }
  });
  runApp(const Root()); // start app
}

class Root extends StatelessWidget { // main app
  const Root({super.key}); // constructor

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false, // don't show debug label in corner of ap
      initialRoute: initialRoute, // set up initial page depending on whether user is logged in
      routes: {
        'login': (context) => const LoginPage(),
        'home': (context) => const Home() // different pages of app
      },
    );  // MaterialApp
  }
}
```

loginpage.dart:

```dart
import 'package:flutter/material.dart'; // basic flutter widgets
import 'package:flutter/rendering.dart'; // used to get around navigation error
import 'package:flutter/services.dart'; // also used to get around navigation error
import 'package:firebase_auth/firebase_auth.dart'; // use firebase authentication to sign in

class LoginPage extends StatefulWidget { // creating login page
  const LoginPage({super.key}); // constructor

  @override
  State<LoginPage> createState() => _LoginPageState(); // setting up state so page can be updated
}

class _LoginPageState extends State<LoginPage> { // main body

  static String email = '', password = ''; // set up email and password variables, static to be updated when typed in from custom class
  bool showErrorMessage = false; // whether to show failure to log in message

  void signIn() async { // function to sign in called when log in button is pressed
    SystemChannels.textInput.invokeMethod('TextInput.hide'); // hides keyboard - this was causing a navigation issue when signing in
    try { // don't crash app on failure
      await FirebaseAuth.instance.signInWithEmailAndPassword(email: email, password: password); // attempt to sign in
      email = '';
      password = ''; // clear variables for security
      if(mounted) { // check if context exists and is updated to avoid errors
        RendererBinding.instance.addPostFrameCallback((_) {Navigator.pushReplacementNamed(context, 'home');}); // ensure that landing page is ready to render and then navigate
          to landing page
      }
    } on FirebaseAuthException catch(e) { // if signing in fails
      showErrorMessage = true; // show the error message
      setState(() {}); // update the UI
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: const Color(0xFFF4F4F4), // set background colour
      body: Center( // center items horizontally
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center, // center items vertically
          children: <Widget>[
            Image.asset('./assets/logo.png',width: MediaQuery.of(context).size.width*0.9), // show Secure Defender logo
            const Padding(padding: EdgeInsets.only(top: 70)), // gap below logo
            const CustomInput(isPassword: false), // email input
            const Padding(padding: EdgeInsets.only(top: 20)), // gap between inputs
            const CustomInput(isPassword: true), // password input
            const Padding(padding: EdgeInsets.only(top: 30)), // gap above sign in button
            MaterialButton(onPressed: ()=>{signIn()}, child: Container( // create sign in button
                decoration: BoxDecoration(
                    border: Border.all(color:Colors.black,width:3), // black border
                    color: const Color(0xFFC5C5C5) // grey background
                ),  // BoxDecoration
                width: 160,
                height: 40, // set dimensions
                child: const Center(child: Text('Log in',style: TextStyle(fontSize: 20))) // button text
            )),  // Container, MaterialButton
            const Padding(padding: EdgeInsets.only(top: 40)), // gap below button
            Visibility(
              visible: showErrorMessage, // to allow error message to be initially hidden
              child: const Center(child: Text('Your email or password is incorrect - please try again',style: TextStyle( // error message
                  fontSize: 14,
                  color: Colors.red // make text small and red
              ))),  // TextStyle, Text, Center
            )  // Visibility
          ],  // <Widget>[]
        ),  // Column
      ),  // Center
    );  // Scaffold
  }
}

class CustomInput extends StatelessWidget { // custom class for text input to avoid code repetition
  final bool isPassword; // whether the input is for a password or not (in which case it is treated as an email input)
  const CustomInput({super.key,required this.isPassword}); // constructor with required field indicating whether input is for password or email

  @override
  Widget build(BuildContext context) {
    return Container(
      width: MediaQuery.of(context).size.width*0.8,
      height: MediaQuery.of(context).size.width*0.9*0.15, // specify size of text field
      decoration: BoxDecoration(
          color: const Color(0xFFE1E1E1), // grey background
          border: Border.all(color: const Color(0xFF555555)), // different grey border
          borderRadius: BorderRadius.circular(18) // curved borders
      ),  // BoxDecoration
      child: TextField(onChanged: (text)=>{isPassword?_LoginPageState.password=text:_LoginPageState.email=text}, obscureText: isPassword, decoration: InputDecoration( //
       update either email or password depending on which text field, show dots if password is being entered
          labelText: 'Please enter your ${isPassword?"password":"email"}', // placeholder text to be displayed inside text input
          border: InputBorder.none, // no borders here as custom borders are used
          contentPadding: const EdgeInsets.all(8), // add space between custom borders and text inside
          labelStyle: const TextStyle(color: Color(0xFF333333),fontSize: 20) // make placeholder text a contrasting colour and fairly big
      )),  // InputDecoration, TextField
    );  // Container
  }
}
```

home.dart:



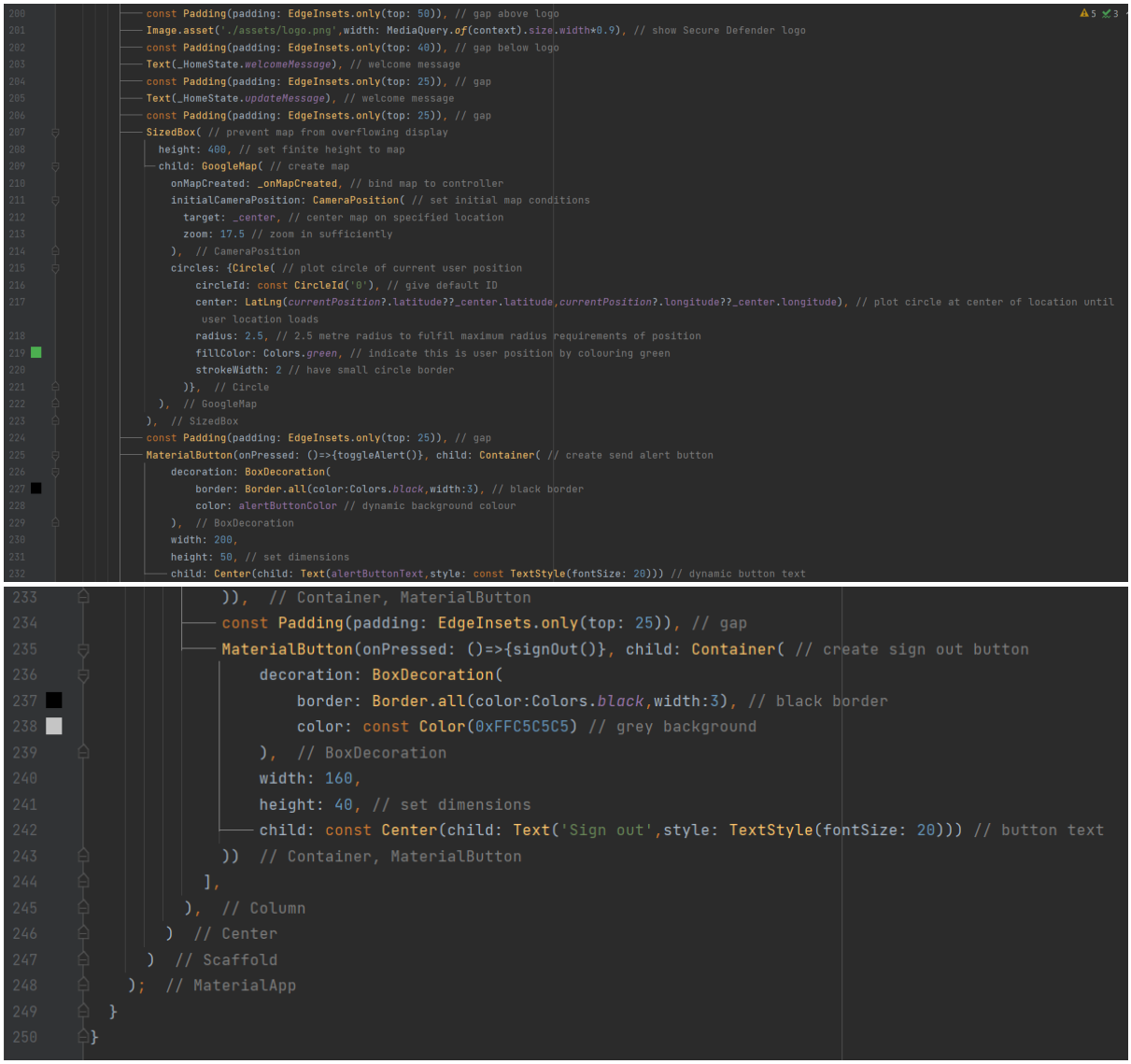
```dart
import 'dart:async'; // to get constantly updating location

import 'package:firebase_core/firebase_core.dart'; // main firebase framework
import 'package:flutter/material.dart'; // flutter widgets
import 'package:firebase_auth/firebase_auth.dart'; // login features
import 'package:firebase_database/firebase_database.dart'; // database access
import 'package:google_maps_flutter/google_maps_flutter.dart'; // map rendering
import 'package:geolocator/geolocator.dart'; // get current location

class Home extends StatefulWidget {
  const Home({super.key}); // constructor

  @override
  State<Home> createState() => _HomeState(); // allow page to update
}

class _HomeState extends State<Home> {
  static String welcomeMessage = 'Welcome.', updateMessage = 'No updates.'; // set default text for welcome and update messages
  static DatabaseReference ref = FirebaseDatabase.instance.ref('${FirebaseAuth.instance.currentUser?.uid!}/name'); // create reference to where user's name will be stored in database
  static DatabaseReference isUpdateRef = FirebaseDatabase.instance.ref('${FirebaseAuth.instance.currentUser?.uid!}/isUpdate'); // creating more database references for speedy database access
  static DatabaseReference updateTextRef = FirebaseDatabase.instance.ref('${FirebaseAuth.instance.currentUser?.uid!}/update/message');
  static DatabaseReference rootRef = FirebaseDatabase.instance.ref('${FirebaseAuth.instance.currentUser?.uid}');
  static DatabaseReference isAlertRef = FirebaseDatabase.instance.ref('${FirebaseAuth.instance.currentUser?.uid}/isAlert');
  static DatabaseReference isReceivedAlertRef = FirebaseDatabase.instance.ref('${FirebaseAuth.instance.currentUser?.uid}/update/isAlert');
  static DatabaseReference alertPositionRef = FirebaseDatabase.instance.ref('${FirebaseAuth.instance.currentUser?.uid}/update/alertPosition');

  Color alertButtonColor = const Color(0xFFFF0000); // setting up send alert button initial colour and text
  String alertButtonText = 'SEND ALERT';

  late GoogleMapController mapController; // allow map to update rather than remain static
  final LatLng _center = const LatLng(52.4310,-1.9036); // set default position of map
  void _onMapCreated(GoogleMapController controller) { // setting up binding for map and controller
    mapController = controller; // binding map with controller
  }
  late GoogleMapController secondMapController; // duplicating map above to create another for received alert popup - allows maps to be controlled independently
  final LatLng _center2 = const LatLng(52.4310,-1.9036);
  void _onMap2Created(GoogleMapController controller) {
    secondMapController = controller;
  }

  setWelcomeMessage() async { // add name into welcome message
    final snapshot = await ref.get(); // wait for name of user to be fetched
    if(snapshot.exists) { // if fetch was successful
      var name = snapshot.value; // store name
      _HomeState.welcomeMessage = 'Welcome $name!'; // add name into welcome message
    }
  }

  static Position? currentPosition; // variable to store current position
  final LocationSettings locationSettings = const LocationSettings(accuracy: LocationAccuracy.best); // use best location accuracy

  locationUpdates() async { // setup location fetching
    LocationPermission perms = await Geolocator.checkPermission(); // check if location can be fetched
    if (perms != LocationPermission.always && perms != LocationPermission.whileInUse) { // if not able to fetch location
      perms = await Geolocator.requestPermission(); // request permission
      doLocationUpdate(); // fetch location
    } else {
      doLocationUpdate(); // fetch location if permissions are already there
    }
  }
  doLocationUpdate() { // start location change listener to update location on database
    StreamSubscription<Position> positionStream = Geolocator.getPositionStream(locationSettings: locationSettings).listen((Position? position) { // get live location
      setState(() {
        currentPosition = position; // store position
      }); // update map
      updateDatabaseLocation(position); // update database with new location
```

```
  });
}
updateDatabaseLocation(position) { // update database with location
  DatabaseReference locationRef = FirebaseDatabase.instance.ref('${FirebaseAuth.instance.currentUser?.uid}/position'); // create reference to write to
  locationRef.update({ // update database with new latitude and longitude
    "latitude": position.latitude,
    "longitude": position.longitude
  });
}

listenForUpdates() { // listen for updates sent from central server
  isUpdateRef.onValue.listen((DatabaseEvent event) async { // when a new update is published in the database
    final isUpdate = event.snapshot.value; // check if isUpdate in the database is true
    if(isUpdate == true) { // if this is the case
      final isReceivedAlertSnapshot = await isReceivedAlertRef.get(); // fetch value of inner isAlert in the database
      final isReceivedAlert = isReceivedAlertSnapshot.value; // store value of inner isAlert
      final snapshot = await updateTextRef.get(); // get text of new update
      setState(() {
        updateMessage = snapshot.value.toString(); // update user interface with new text
      });
      if(isReceivedAlert==true) { // if update was also an alert
        handleReceivedAlert(); // handle this received alert
      }
      rootRef.update({
        "isUpdate": false // set isUpdate in the database to false after handling the update to allow new updates to be sent
      });
    }
  });
}

handleReceivedAlert() { // handle the received alert
  Object? otherUserLongitude = -1.9036, otherUserLatitude = 52.4310; // constants that are the same as the map centre to set initial position on map
  showDialog(context: context, builder: (BuildContext context) { // show the dialog that I am about to create
    return AlertDialog( // create dialog
      insetPadding: const EdgeInsets.all(10), // detatch from screen edges
```

```
      title: Row( // title row                                                                                    ⚠5 ✓3 ^
        children: [
          const Text('ALERT'), // alert title
          const Spacer(), // pushes following items to the right and takes up all remaining space
          IconButton(onPressed: () => Navigator.pop(context), icon: const Icon(Icons.close_rounded)) // create close button that closes dialog when clicked
        ]
      ),  // Row
      content: StatefulBuilder( // allow dynamic changes to the dialog and allow it to be created
        builder: (BuildContext context, StateSetter setNewState) { // use a different function to set the state inside the dialog so it is not confused with setting the
         state of the main page
          alertPositionRef.onValue.listen((DatabaseEvent event) { // listen for changes of the position of the alerting user
            var otherLong = event.snapshot.child('longitude').value;
            var otherLat = event.snapshot.child('latitude').value; // set longitude and latitude of other user from database to our app
            otherUserLongitude = otherLong;
            otherUserLatitude = otherLat; // used an extra step to allow the type of the variables to be initialised before writing them into the variables that I will use
             to display the other user's circle
            setNewState(() {}); // update dialog user interface
          });
          return Column( // show rest of dialog
            children: [
              Text(updateMessage), // show update text in dialog so that name of user in trouble is displayed
              const Padding(padding: EdgeInsets.only(top: 25)), // gap
              SizedBox( // prevent map from overflowing display
                height: 500, // set finite height to map
                width: 400, // set width to fit in dialog
                child: GoogleMap( // create map
                  onMapCreated: _onMap2Created, // bind map to controller
                  initialCameraPosition: CameraPosition( // set initial map conditions
                      target: _center2, // center map on specified location
                      zoom: 17.5 // zoom in sufficiently
                  ),  // CameraPosition
                  circles: {
                    Circle( // plot circle of current user position
                      circleId: const CircleId('0'), // give default ID
```

```
                    center: LatLng(currentPosition?.latitude??_center.latitude,currentPosition?.longitude??_center.longitude), // plot circle at center of location until
                     user location loads
                    radius: 2.5, // 2.5 metre radius to fulfil maximum radius requirements of position
                    fillColor: Colors.green, // indicate this is user position by colouring green
                    strokeWidth: 2 // have small circle border
                  ),  // Circle
                  Circle( // plot circle of other user position
                    circleId: const CircleId('1'), // give other user different ID
                    center: LatLng(otherUserLatitude as double,otherUserLongitude as double), // convert longitude and latitude to decimals before using them to plot the
                     circle
                    radius: 2.5, // 2.5 metre radius to fulfil maximum radius requirements of position
                    fillColor: Colors.red, // indicate this is other user's position by colouring red
                    strokeWidth: 2 // have small circle border
                  )  // Circle
                },
              ),  // GoogleMap
            )  // SizedBox
          ],
        );  // Column
      },
    ),  // StatefulBuilder
  );});  // AlertDialog
}

_HomeState() { // constructor
  setWelcomeMessage(); // update welcome message (add user's name into it)
  locationUpdates(); // start live updating of user location
  listenForUpdates(); // start listening for updates
}

signOut() { // function to sign out
  FirebaseAuth.instance.signOut(); // sign out
  Navigator.pushReplacementNamed(context, 'login'); // navigate to login page
```

```
  }

  toggleAlert() async { // functionality for sending and cancelling alerts
    final snapshot = await isAlertRef.get(); // get current alerting status (outer isAlert)
    final snapVal = snapshot.value; // interpret value of fetched data
    bool alertToPush; // will be the opposite of what was fetched as it inverts the alerting status, as a toggle should
    if(snapVal==true) {
      alertToPush = false; // inversion
      setState(() { // update user interface
        alertButtonColor = const Color(0xFFFF0000); // cancelling alert and changing button accordingly
        alertButtonText = 'SEND ALERT';
      });
    } else {
      alertToPush = true; // inversion
      setState(() { // update user interface
        alertButtonColor = const Color(0xFFFFCC00); // enabling alert and changing button accordingly
        alertButtonText = 'Alert on!';
      });
    }
    rootRef.update({ // update database to store new alert status
      'isAlert': alertToPush // (outer isAlert)
    });

  }

  @override
  Widget build(BuildContext context) { // main page
    return MaterialApp(
      debugShowCheckedModeBanner: false, // hide debug banner
      home: Scaffold(
        backgroundColor: const Color(0xFFF4F4F4), // set background colour
        body: Center( // align items horizontally centrally
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center, // align items vertically centrally
            children: [
```



```
const Padding(padding: EdgeInsets.only(top: 50)), // gap above logo
Image.asset('./assets/logo.png',width: MediaQuery.of(context).size.width*0.9), // show Secure Defender logo
const Padding(padding: EdgeInsets.only(top: 40)), // gap below logo
Text(_HomeState.welcomeMessage), // welcome message
const Padding(padding: EdgeInsets.only(top: 25)), // gap
Text(_HomeState.updateMessage), // welcome message
const Padding(padding: EdgeInsets.only(top: 25)), // gap
SizedBox( // prevent map from overflowing display
  height: 400, // set finite height to map
  child: GoogleMap( // create map
    onMapCreated: _onMapCreated, // bind map to controller
    initialCameraPosition: CameraPosition( // set initial map conditions
      target: _center, // center map on specified location
      zoom: 17.5 // zoom in sufficiently
    ), // CameraPosition
    circles: {Circle( // plot circle of current user position
        circleId: const CircleId('0'), // give default ID
        center: LatLng(currentPosition?.latitude??_center.latitude,currentPosition?.longitude??_center.longitude), // plot circle at center of location until
         user location loads
        radius: 2.5, // 2.5 metre radius to fulfil maximum radius requirements of position
        fillColor: Colors.green, // indicate this is user position by colouring green
        strokeWidth: 2 // have small circle border
      )},  // Circle
  ),  // GoogleMap
),  // SizedBox
const Padding(padding: EdgeInsets.only(top: 25)), // gap
MaterialButton(onPressed: ()=>{toggleAlert()}, child: Container( // create send alert button
    decoration: BoxDecoration(
        border: Border.all(color:Colors.black,width:3), // black border
        color: alertButtonColor // dynamic background colour
    ),  // BoxDecoration
    width: 200,
    height: 50, // set dimensions
    child: Center(child: Text(alertButtonText,style: const TextStyle(fontSize: 20))) // dynamic button text
```



```
        )),  // Container, MaterialButton
        const Padding(padding: EdgeInsets.only(top: 25)), // gap
        MaterialButton(onPressed: ()=>{signOut()}, child: Container( // create sign out button
            decoration: BoxDecoration(
                border: Border.all(color:Colors.black,width:3), // black border
                color: const Color(0xFFC5C5C5) // grey background
            ),  // BoxDecoration
            width: 160,
            height: 40, // set dimensions
            child: const Center(child: Text('Sign out',style: TextStyle(fontSize: 20))) // button text
        ))  // Container, MaterialButton
      ],
    ),  // Column
  )  // Center
 )  // Scaffold
);  // MaterialApp
}
}
```

firebase_options.dart is not included due to it containing sensitive information about my database access credentials. That is the only purpose this file serves.