Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

Cognisent

Project Engineering

Year 4

Ross Molloy (G00359442)

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2021/2022

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technological University.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

____

Acknowledgements

I would like to thank my project supervisor, Brian O’Shea, for his assistance and guidance throughout the duration of my final year project.

I would also like to thank my Project Engineering coordinators Paul Lennon, Michelle Lynch, and Niall O’Keeffe for their feedback and tutelage on project work throughout the year.

Table of Contents

1 Summary

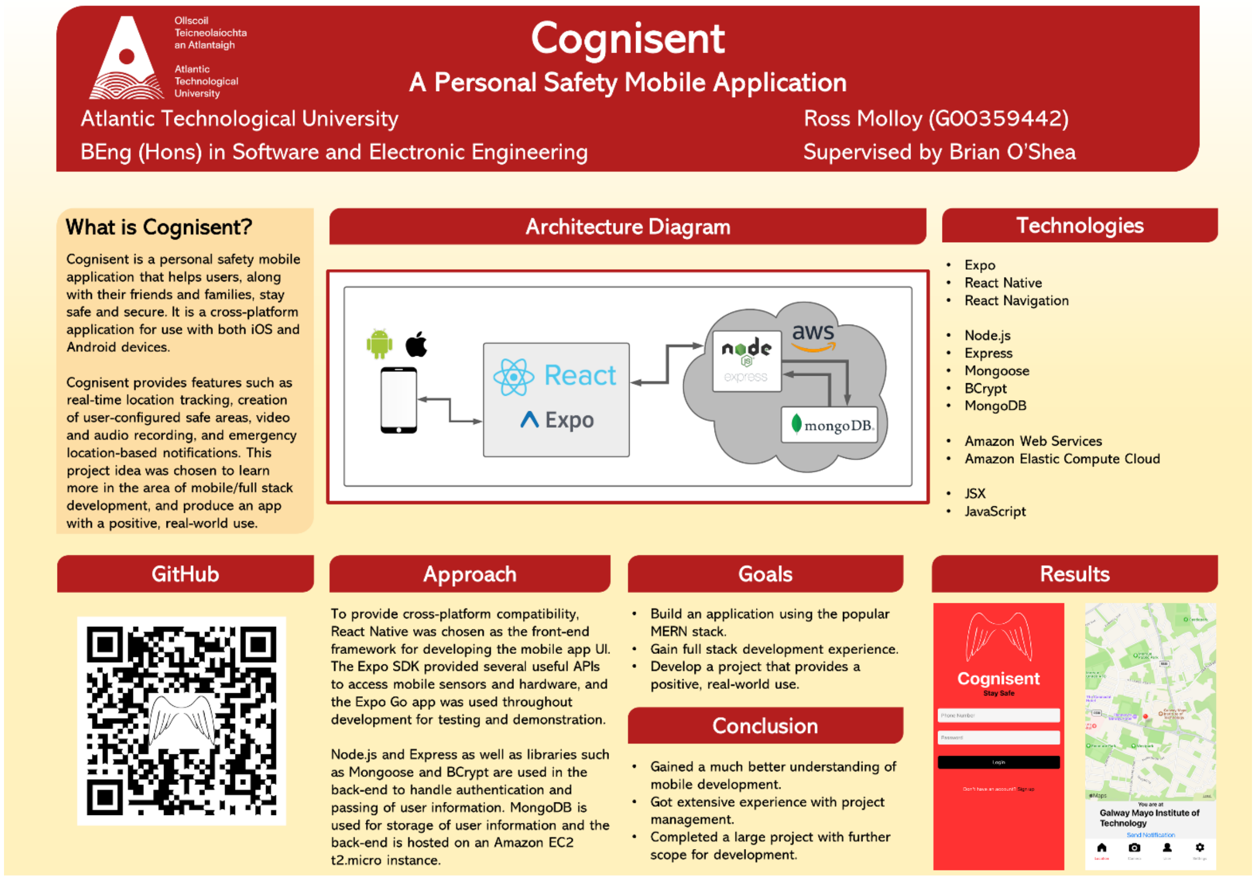
For my final year project, I wanted to develop an application that could help solve important, relevant real-world problems and assist people in a positive, reassuring way.

Cognisent is a cross-platform personal safety and emergency response mobile application. It includes features such as real-time location tracking, user-configured safe areas, emergency location-based push notifications, as well as video/audio recording. Users can create and login to an account within the app and set up a network of contacts. Upon signup, the user is tasked with setting up a 'safe area'. Within the app, the user's location is tracked, and notifications can be sent at any time with the current location information attached. The user can also record a video using the phone's built-in camera and microphone. Each user has their own profile page within the app and can update their information along with their safe area in the app's settings.

For project management, Jira was used for tracking work throughout the duration of the project, along with OneNote which was used for logging of information/work on a weekly basis. The application has been developed using the MERN stack (MongoDB, Express, React Native, Node.js). Technologies such as Expo, React Navigation, Mongoose, and Amazon EC2 were also vital components for development and delivery of the project. React Native, Expo, and React Navigation are used in the front-end, while Node.js, Express, Mongoose are used in the back-end, with the back-end being hosted on an EC2 t2.micro instance. MongoDB is used for storage of user information.

In completing this project, I developed and enhanced my skills in project planning, management, and documentation. I improved my software engineering skills through working with technologies I had little or no experience with before and also through developing an application from scratch with a sizeable codebase. I also greatly improved my knowledge of the mobile app development process.

2 Poster



Ollscoil Teicneolaíochta an Atlantaigh
Atlantic Technological University

Cognisent

A Personal Safety Mobile Application

Atlantic Technological University
BEng (Hons) in Software and Electronic Engineering

Ross Molloy (G00359442)
Supervised by Brian O'Shea

What is Cognisent?

Cognisent is a personal safety mobile application that helps users, along with their friends and families, stay safe and secure. It is a cross-platform application for use with both iOS and Android devices.

Cognisent provides features such as real-time location tracking, creation of user-configured safe areas, video and audio recording, and emergency location-based notifications. This project idea was chosen to learn more in the area of mobile/full stack development, and produce an app with a positive, real-world use.

Architecture Diagram

Technologies

- Expo
- React Native
- React Navigation

- Node.js
- Express
- Mongoose
- BCrypt
- MongoDB

- Amazon Web Services
- Amazon Elastic Compute Cloud

- JSX
- JavaScript

GitHub

Approach

To provide cross-platform compatibility, React Native was chosen as the front-end framework for developing the mobile app UI. The Expo SDK provided several useful APIs to access mobile sensors and hardware, and the Expo Go app was used throughout development for testing and demonstration.

Node.js and Express as well as libraries such as Mongoose and BCrypt are used in the back-end to handle authentication and passing of user information. MongoDB is used for storage of user information and the back-end is hosted on an Amazon EC2 t2.micro instance.

Goals

- Build an application using the popular MERN stack.
- Gain full stack development experience.
- Develop a project that provides a positive, real-world use.

Conclusion

- Gained a much better understanding of mobile development.
- Got extensive experience with project management.
- Completed a large project with further scope for development.

Results

Figure 2.1 - Poster

3 Introduction

One of the primary goals of my final year project was to develop an application that could help make a positive effect on important real-world issues. Following recent events in the news around the time of my project commencement surrounding the issue of women's safety, as well as my own personal experiences regarding the safety of the elderly and those suffering from dementia, I decided to build an app targeting the issue of personal safety and emergency response.

Considering the qualitative and quantitative power of hardware and sensors built into modern smartphones, as well as the portability and widespread use of mobile phones in today's world, it made sense to build my project for use with mobile devices. Alongside those considerations, developing a mobile application would provide a valuable, challenging, and exciting learning experience for me as I had little to no experience developing on mobile platforms going into my final year.

In addition, this project would provide a challenge in developing a sizeable codebase from scratch encompassing many different technologies and development practices across the full stack. Having worked mainly in Java, C, and C++ in college, on Java microservices during my third-year work placement, and not having had as much experience with popular front-end JavaScript frameworks such as React, it made sense to build a project largely focused in that area.

4 Tools and Technology

In researching and planning this project, it was essential that the right tools and technologies were chosen for implementing all the desired features in the application, using a modern, relevant stack, and enacting a professional workflow.

4.1 React Native

React Native is a JavaScript UI framework used to develop applications for Android and iOS [1]. It allows developers to use native platform elements while writing platform-agnostic JSX, a unique mixture of JavaScript and XML-esque markup. It's very similar to React, but targeting mobile platforms as opposed to the web, while providing cross-platform compatibility unlike Objective-C (iOS) and Java (Android) and access to native UI unlike hybrid mobile app development frameworks.

4.2 React Navigation

React Navigation is a library that provides routing and navigation capabilities to React Native applications [2]. It handles the presentation and transition of screens within the app. Screens are stored in a navigation structure within the app and transitions/animations are configured to match OS default behaviour.

4.3 Expo

Both the Expo Go application and Expo SDK elements were used extensively throughout the project. Expo is a set of tools and services built around React Native that provide access to mobile hardware and streamlines testing of applications [3]. Generally, building React Native projects with native iOS code requires a macOS machine, but with Expo, it is possible to test and debug on a Windows machine making it an essential aspect of my project's development and testing process.

4.4 Node.js

Node.js is a JavaScript runtime environment allowing the execution of JavaScript code outside of a web browser [4]. As my front-end was built with React Native, Node.js seemed the obvious choice for building my back-end by allowing me to write JavaScript (and JSX) across both the client and server-side of the application.

4.5 Express

Express is a back-end Node.js web application framework [5]. Express handles routing, or how the app's endpoints respond to requests from the client. Using Express, I was able to define responses from endpoints for user authentication (login, signup), as well as fetching and updating of user information.

4.6 Mongoose

Mongoose is a tool that provides object modelling with MongoDB. With the use of its schemas, I was able to easily define and validate the structure and layout of objects being stored in my database [6].

4.7 MongoDB

MongoDB is a document-based NoSQL database program [7]. MongoDB was chosen for my database as I wanted to use a more modern NoSQL database as opposed to a traditional, more rigid relational database. MongoDB is a popular choice as evidenced by its use in the MERN and MEAN development stacks, so it made sense to use alongside React Native, Express, and Node.js in my project.

4.8 Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2) is an AWS service that allows users to deploy services to the cloud by running their applications on virtual computers [8]. In the context of my application, I use EC2 to run my back-end in the cloud on a t2.micro instance. This refers to the computing power of a virtual computing environment, t2.micro being one of the cheaper, least intensive options.

5 Project Architecture

This is the architecture diagram for my project.

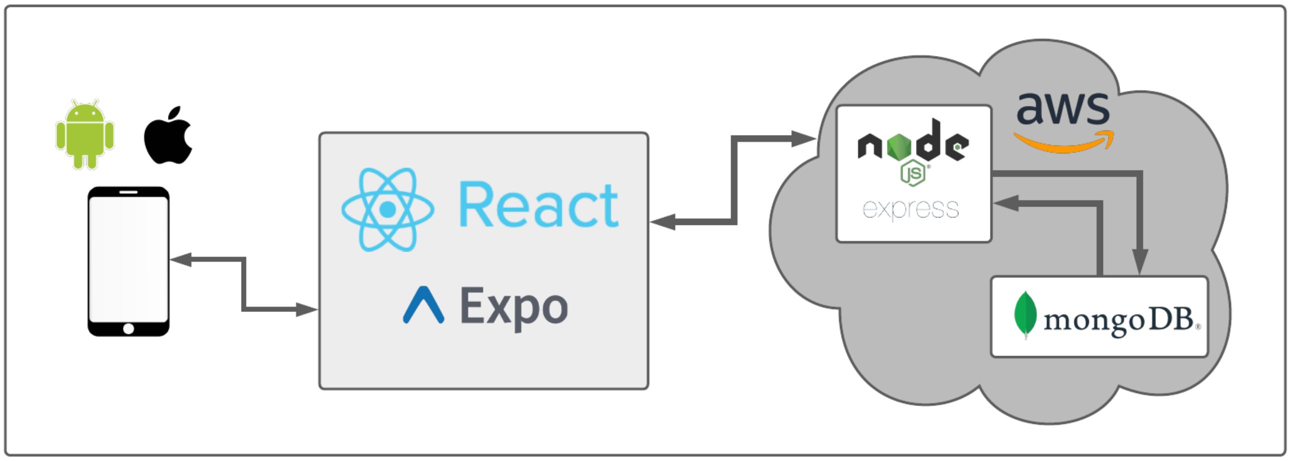
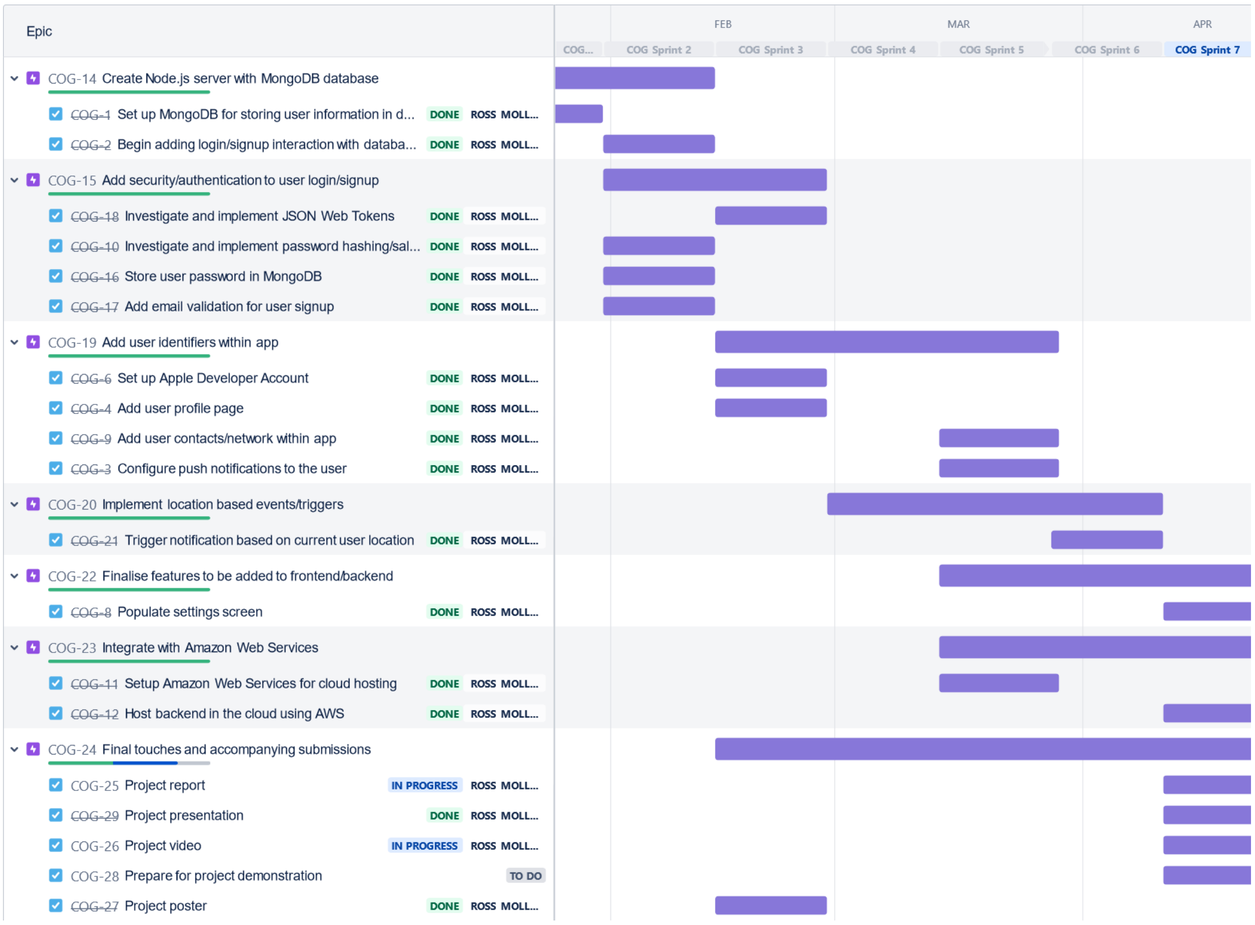


Figure 5.1 - Architecture Diagram

6 Project Plan

I used Jira as a project management tool. I updated it every week when managing sprints and upon completion of project work. I created epics (large user stories that can be broken down into smaller pieces of work [9]) for features and each epic contained one or more individual tasks. These tasks were pulled into sprints and worked on throughout the course of two weeks before the end of a sprint and beginning of a new one.



Epic		
COG-14 Create Node.js server with MongoDB database		
~~COG-1~~ Set up MongoDB for storing user information in d...	DONE	ROSS MOLL...
~~COG-2~~ Begin adding login/signup interaction with databa...	DONE	ROSS MOLL...
COG-15 Add security/authentication to user login/signup		
~~COG-18~~ Investigate and implement JSON Web Tokens	DONE	ROSS MOLL...
~~COG-10~~ Investigate and implement password hashing/sal...	DONE	ROSS MOLL...
~~COG-16~~ Store user password in MongoDB	DONE	ROSS MOLL...
~~COG-17~~ Add email validation for user signup	DONE	ROSS MOLL...
COG-19 Add user identifiers within app		
~~COG-6~~ Set up Apple Developer Account	DONE	ROSS MOLL...
~~COG-4~~ Add user profile page	DONE	ROSS MOLL...
~~COG-9~~ Add user contacts/network within app	DONE	ROSS MOLL...
~~COG-3~~ Configure push notifications to the user	DONE	ROSS MOLL...
COG-20 Implement location based events/triggers		
~~COG-21~~ Trigger notification based on current user location	DONE	ROSS MOLL...
COG-22 Finalise features to be added to frontend/backend		
~~COG-8~~ Populate settings screen	DONE	ROSS MOLL...
COG-23 Integrate with Amazon Web Services		
~~COG-11~~ Setup Amazon Web Services for cloud hosting	DONE	ROSS MOLL...
~~COG-12~~ Host backend in the cloud using AWS	DONE	ROSS MOLL...
COG-24 Final touches and accompanying submissions		
COG-25 Project report	IN PROGRESS	ROSS MOLL...
~~COG-29~~ Project presentation	DONE	ROSS MOLL...
COG-26 Project video	IN PROGRESS	ROSS MOLL...
COG-28 Prepare for project demonstration	TO DO	
~~COG-27~~ Project poster	DONE	ROSS MOLL...

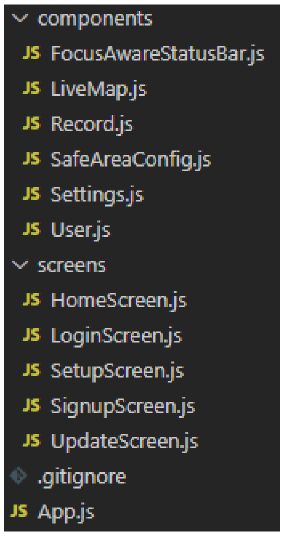
Figure 6.1 - Jira Project Roadmap

7 Project Code

This section will delve into the important code/software elements of the project and will be split into two sections, one describing the front-end and the other describing the back-end.

7.1 Frontend

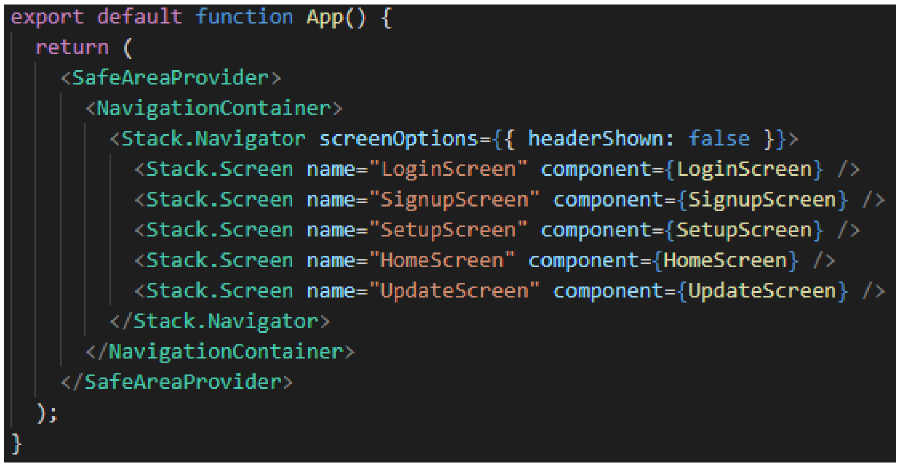
There are three primary sections into which the front-end code can be split. There is App.js, which contains the core app structure, there are the screens which make up the navigation structure of the app, and there are the components which are the independent and reusable parts of the code which make up the screens.



```
components
  FocusAwareStatusBar.js
  LiveMap.js
  Record.js
  SafeAreaConfig.js
  Settings.js
  User.js
screens
  HomeScreen.js
  LoginScreen.js
  SetupScreen.js
  SignupScreen.js
  UpdateScreen.js
.gitignore
App.js
```

Figure 7.1.1 – Frontend Structure

In App.js, each screen is stored inside a Stack.Navigator [10].

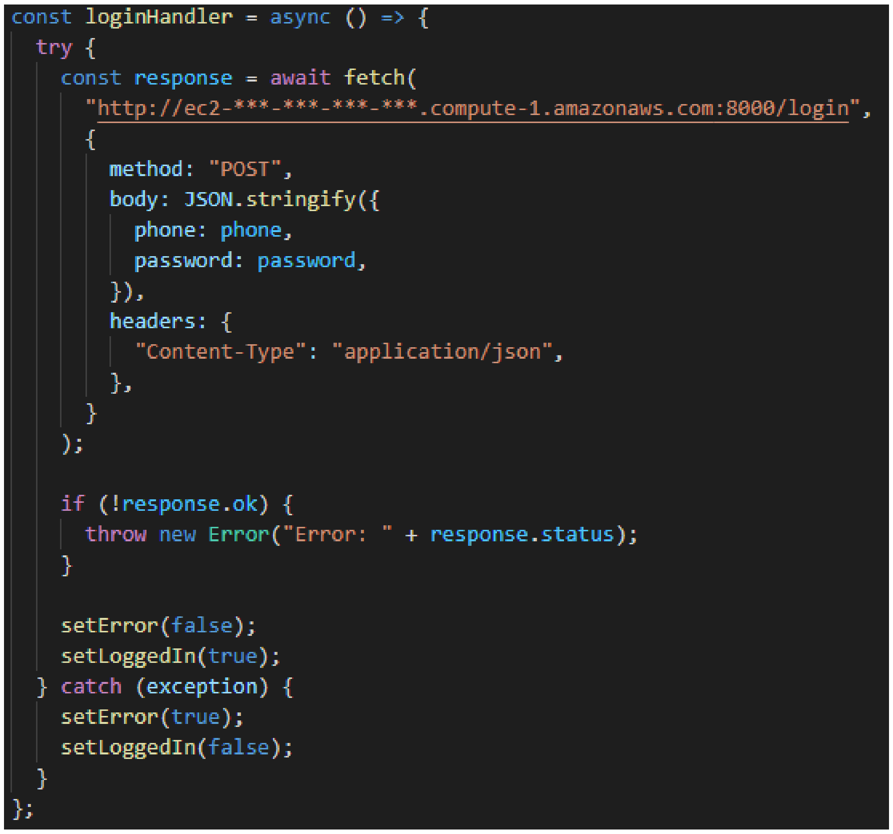


```
export default function App() {
  return (
    <SafeAreaProvider>
      <NavigationContainer>
        <Stack.Navigator screenOptions={{ headerShown: false }}>
          <Stack.Screen name="LoginScreen" component={LoginScreen} />
          <Stack.Screen name="SignupScreen" component={SignupScreen} />
          <Stack.Screen name="SetupScreen" component={SetupScreen} />
          <Stack.Screen name="HomeScreen" component={HomeScreen} />
          <Stack.Screen name="UpdateScreen" component={UpdateScreen} />
        </Stack.Navigator>
      </NavigationContainer>
    </SafeAreaProvider>
  );
}
```

Figure 7.1.2 - App.js

Each of these screens make up the app and are navigated by the user. The login screen is the first screen the user will come upon when starting up the app and navigation can easily be made between the login and signup screens without having to sign into an account or create one.

Inside LoginScreen.js, a loginHandler() function is defined.

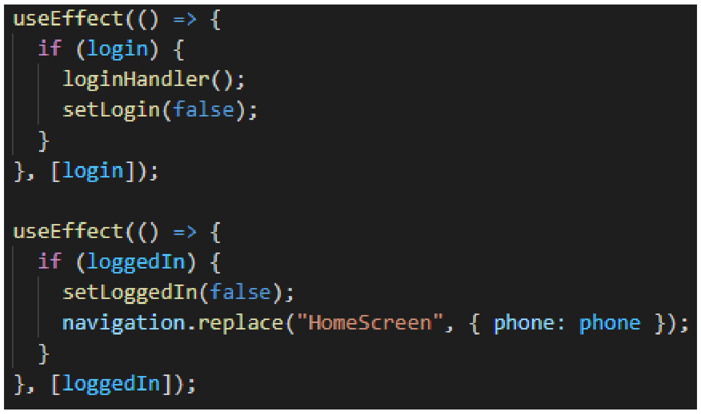


```
const loginHandler = async () => {
  try {
    const response = await fetch(
      "http://ec2-***-***-***-***.compute-1.amazonaws.com:8000/login",
      {
        method: "POST",
        body: JSON.stringify({
          phone: phone,
          password: password,
        }),
        headers: {
          "Content-Type": "application/json",
        },
      }
    );

    if (!response.ok) {
      throw new Error("Error: " + response.status);
    }

    setError(false);
    setLoggedIn(true);
  } catch (exception) {
    setError(true);
    setLoggedIn(false);
  }
};
```

Figure 7.1.3 - Login Handler

This is an asynchronous function, meaning it does not hold up the execution of other code while it is being run. A POST request is sent to the EC2 instance's HTTP address with the input phone number and password in the body of the request and the function waits for this to complete. Upon completion, it checks if the response is 200 OK and, using the State hook [11], it updates the Boolean value of a couple state variables, one to track if there's an error, and another to track if the login was successful.

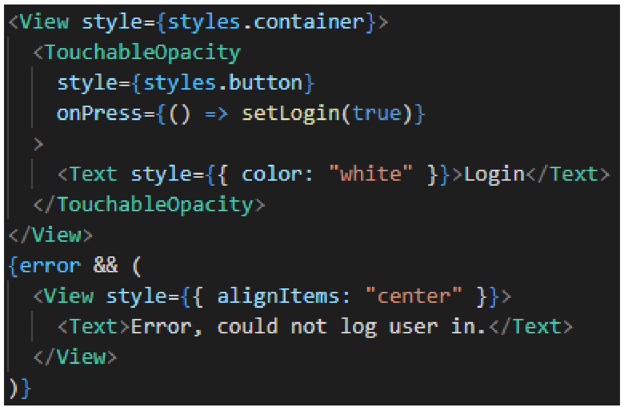
This function, and most throughout the application, are triggered by the Effect hook. The Effect hook lets the developer perform side effects in function components [12]. An example, as used in LoginScreen, is shown below:



```
useEffect(() => {
  if (login) {
    loginHandler();
    setLogin(false);
  }
}, [login]);

useEffect(() => {
  if (loggedIn) {
    setLoggedIn(false);
    navigation.replace("HomeScreen", { phone: phone });
  }
}, [loggedIn]);
```

Figure 7.1.4 - Effect Hooks

The first useEffect there is triggered by an update of the ‘login’ variable, which is updated every time the login button is pressed.



```
<View style={styles.container}>
  <TouchableOpacity
    style={styles.button}
    onPress={() => setLogin(true)}
  >
    <Text style={{ color: "white" }}>Login</Text>
  </TouchableOpacity>
</View>
{error && (
  <View style={{ alignItems: "center" }}>
    <Text>Error, could not log user in.</Text>
  </View>
)}
```

Figure 7.1.5 - Login Button

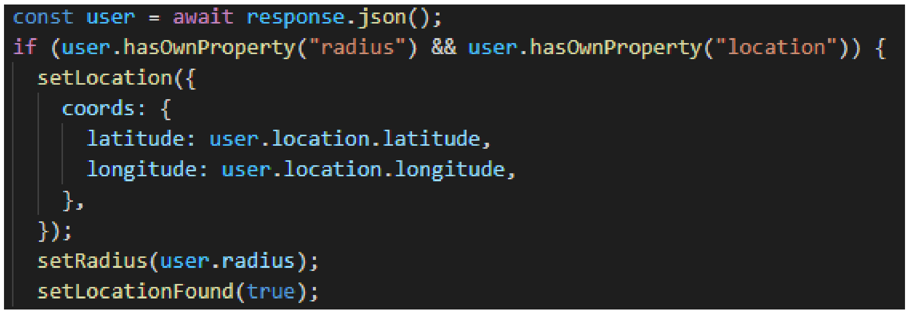
The specific value is checked to prevent side effects from being run the first time the screen is rendered, only when the value is set to true, in both useEffect functions. Upon successful login, ‘navigation.replace’ is called to navigate to the Home Screen (and replace the current route with a new one).

The user also has the option to navigate to the signup screen if they do not have an account.

In each screen/component, numerous imported components from the React Native or Expo libraries (among others) are returned to provide the UI. A snippet is shown above for the login button and error message UI elements. StyleSheet is used for formatting, i.e. to specify the colour of an element, add padding, specify width/height, etc.

SignupScreen is similar to LoginScreen in its makeup, with a signupHandler() function handling POST requests to an endpoint defined in the back-end and triggered upon the press of a button. Both screens have numerous text inputs for inputting user information.

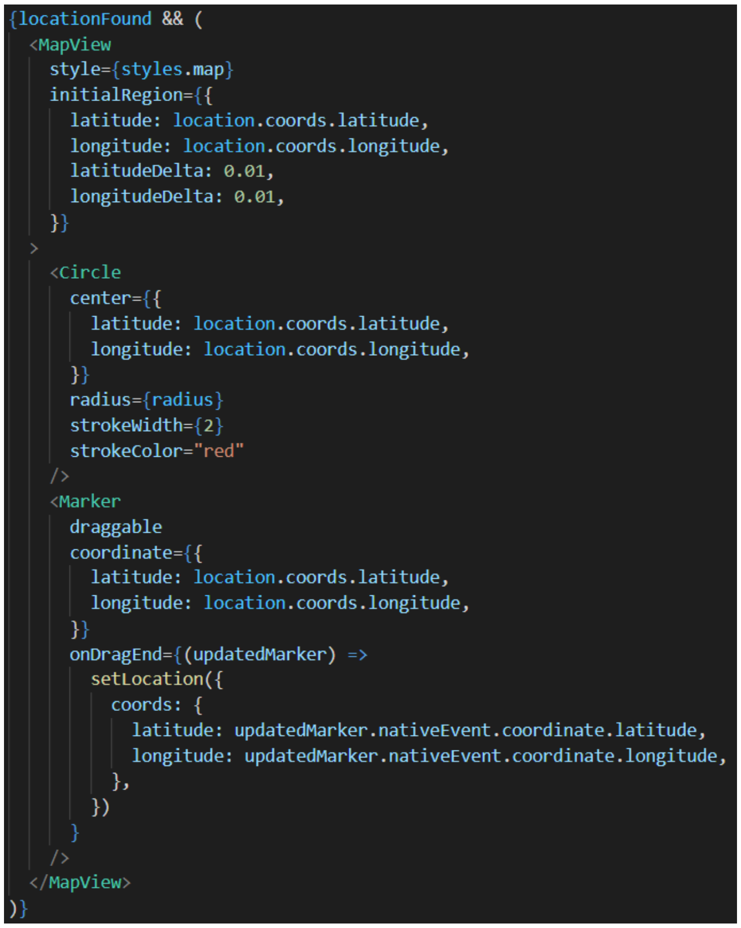
Upon signup, the user is taken to SetupScreen. The input phone number is passed as a prop, which is then passed as a prop to the SafeAreaConfig component. This component controls how the user safe area is configured. Inside SafeAreaConfig, there is a function named ‘getSafeArea’ which is run the first time this component is rendered to see if the user already has a safe area set up. A GET request is sent to retrieve information about the user and the response is stored in a ‘user’ object.



```
const user = await response.json();
if (user.hasOwnProperty("radius") && user.hasOwnProperty("location")) {
  setLocation({
    coords: {
      latitude: user.location.latitude,
      longitude: user.location.longitude,
    },
  });
  setRadius(user.radius);
  setLocationFound(true);
```

Figure 7.1.6 - Safe Area Check

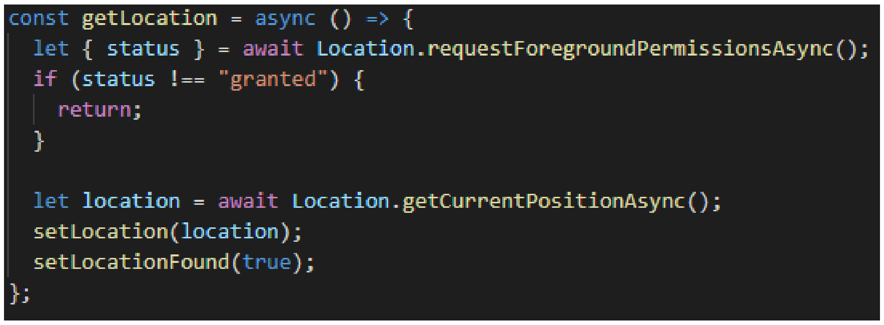
If ‘radius’ and ‘location’ properties are present, location and radius fields are updated and the map is updated to display these coordinates.



```
{locationFound && (
  <MapView
    style={styles.map}
    initialRegion={{
      latitude: location.coords.latitude,
      longitude: location.coords.longitude,
      latitudeDelta: 0.01,
      longitudeDelta: 0.01,
    }}
  >
    <Circle
      center={{
        latitude: location.coords.latitude,
        longitude: location.coords.longitude,
      }}
      radius={radius}
      strokeWidth={2}
      strokeColor="red"
    />
    <Marker
      draggable
      coordinate={{
        latitude: location.coords.latitude,
        longitude: location.coords.longitude,
      }}
      onDragEnd={(updatedMarker) =>
        setLocation({
          coords: {
            latitude: updatedMarker.nativeEvent.coordinate.latitude,
            longitude: updatedMarker.nativeEvent.coordinate.longitude,
          },
        })
      }
    />
  </MapView>
)}
```

Figure 7.1.7 - MapView

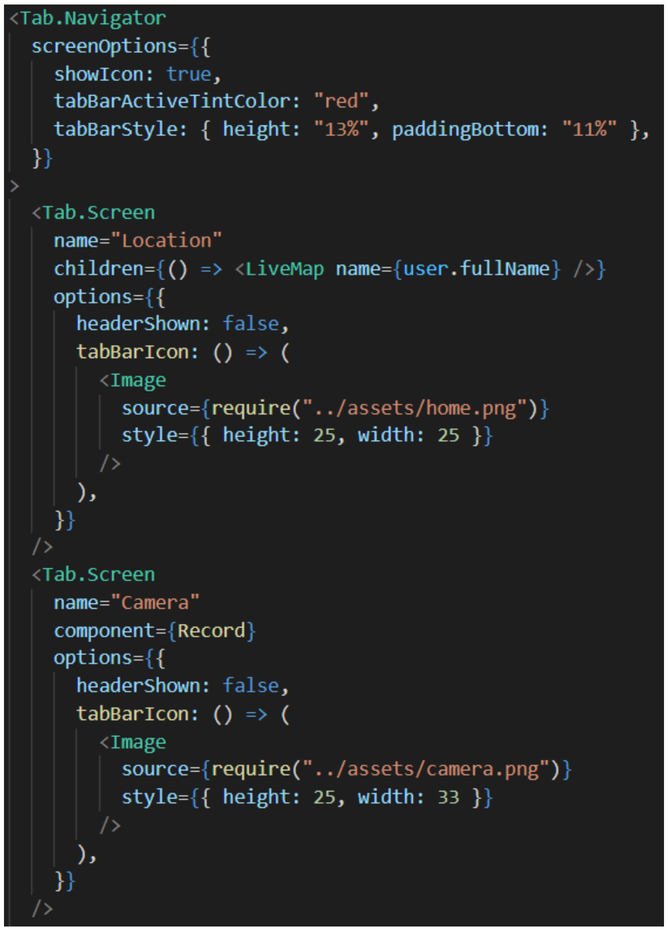
The radius can be manipulated by a slider below the MapView and the marker at the centre of the safe area can be moved to change the safe area if already configured, or when creating one during user account setup. If one hasn't already been configured, GPS location foreground permissions will be requested and, if granted, the current user location will be retrieved.



```
const getLocation = async () => {
  let { status } = await Location.requestForegroundPermissionsAsync();
  if (status !== "granted") {
    return;
  }

  let location = await Location.getCurrentPositionAsync();
  setLocation(location);
  setLocationFound(true);
};
```

Figure 7.1.8 - Location Retrieval

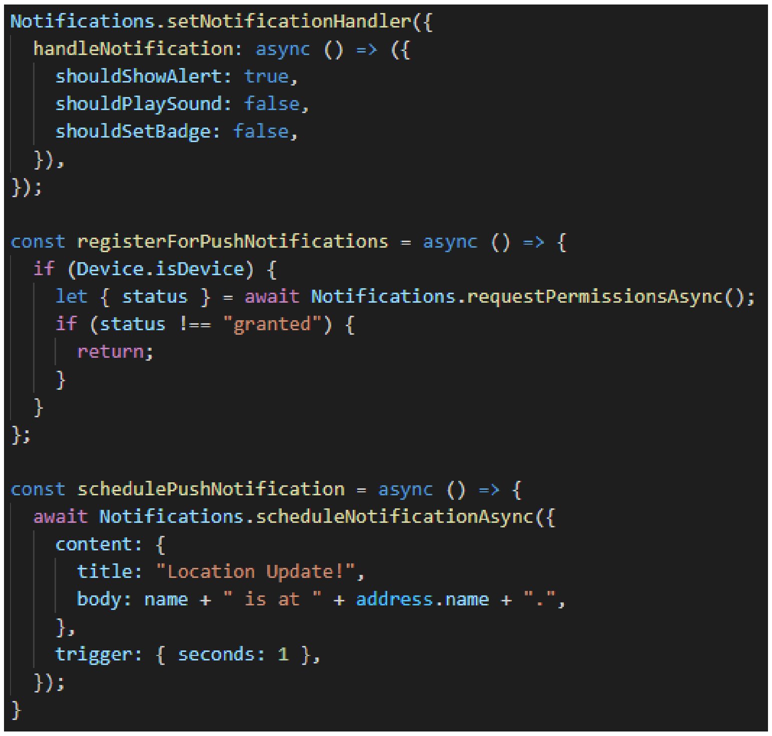
Upon configuring a new safe area and submitting it, a PATCH request is sent to update the user information and store the safe area coordinates. Following this, the user is sent to HomeScreen. Inside HomeScreen, user information is retrieved from MongoDB again and stored in a user object. HomeScreen contains a tab navigator allowing the user to navigate between different components or 'tabs' at the bottom of the screen [13]. Each of these is defined as a 'Tab.Screen'.



```
<Tab.Navigator
  screenOptions={{
    showIcon: true,
    tabBarActiveTintColor: "red",
    tabBarStyle: { height: "13%", paddingBottom: "11%" },
  }}
>
  <Tab.Screen
    name="Location"
    children={() => <LiveMap name={user.fullName} />}
    options={{
      headerShown: false,
      tabBarIcon: () => (
        <Image
          source={require("../assets/home.png")}
          style={{ height: 25, width: 25 }}
        />
      ),
    }}
  />
  <Tab.Screen
    name="Camera"
    component={Record}
    options={{
      headerShown: false,
      tabBarIcon: () => (
        <Image
          source={require("../assets/camera.png")}
          style={{ height: 25, width: 33 }}
        />
      ),
    }}
  />
```

Figure 7.1.9 - Tab Navigation

There is a tab for each of the LiveMap, Record, User, and Settings components. The LiveMap component shows the current real-time location of the user and provides a button for the user to trigger a push notification with their location information attached.

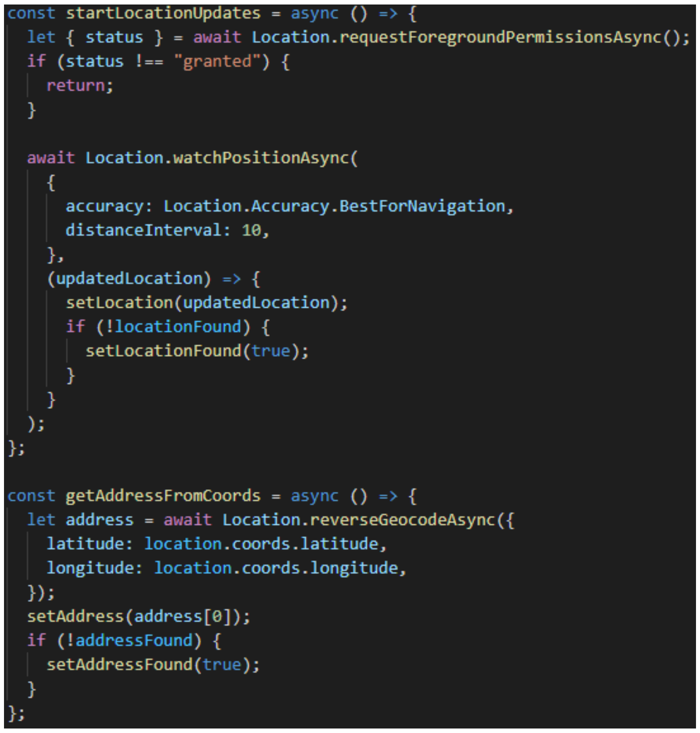


```
Notifications.setNotificationHandler({
  handleNotification: async () => ({
    shouldShowAlert: true,
    shouldPlaySound: false,
    shouldSetBadge: false,
  }),
});

const registerForPushNotifications = async () => {
  if (Device.isDevice) {
    let { status } = await Notifications.requestPermissionsAsync();
    if (status !== "granted") {
      return;
    }
  }
};

const schedulePushNotification = async () => {
  await Notifications.scheduleNotificationAsync({
    content: {
      title: "Location Update!",
      body: name + " is at " + address.name + ".",
    },
    trigger: { seconds: 1 },
  });
}
```

Figure 7.1.10 - Push Notification Setup

Notifications are configured to show an alert and two functions are defined to request permissions for push notifications and schedule a push notification respectively. The content of a push notification is defined in the latter and contains the user's name and location in the body of it.

Access to GPS location is also needed here and functions are defined to watch the user's location and update it regularly, as well as to get the address in English from the coordinates (latitude and longitude) for display purposes.

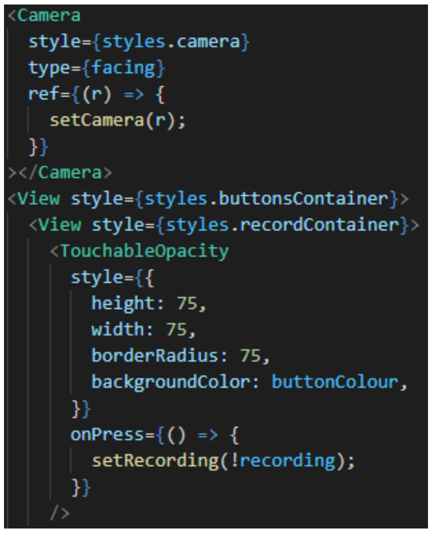


```
const startLocationUpdates = async () => {
  let { status } = await Location.requestForegroundPermissionsAsync();
  if (status !== "granted") {
    return;
  }

  await Location.watchPositionAsync(
    {
      accuracy: Location.Accuracy.BestForNavigation,
      distanceInterval: 10,
    },
    (updatedLocation) => {
      setLocation(updatedLocation);
      if (!locationFound) {
        setLocationFound(true);
      }
    }
  );
};

const getAddressFromCoords = async () => {
  let address = await Location.reverseGeocodeAsync({
    latitude: location.coords.latitude,
    longitude: location.coords.longitude,
  });
  setAddress(address[0]);
  if (!addressFound) {
    setAddressFound(true);
  }
};
```

Figure 7.1.11 – Updating Location

As with SafeAreaConfig, a MapView component is displayed in the UI and the user's location is marked on the map and updated every time the location state variable is updated. Below the map, a button to send a push notification is displayed.

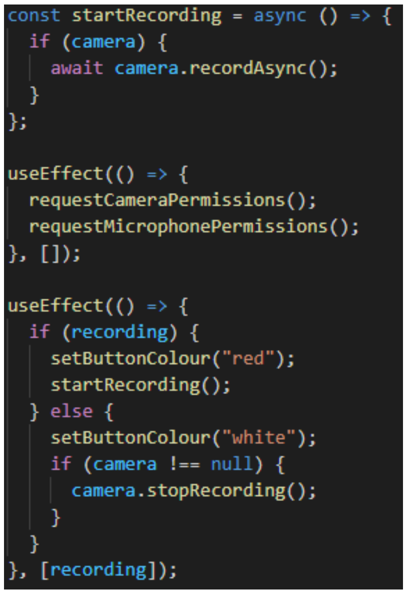
The Record component allows the user to record a video. As with location and push notifications, permissions need to be granted to use the camera and microphone. After these have been granted, the user can record videos using the Camera API from Expo.



```
<Camera
  style={styles.camera}
  type={facing}
  ref={(r) => {
    setCamera(r);
  }}
></Camera>
<View style={styles.buttonsContainer}>
  <View style={styles.recordContainer}>
    <TouchableOpacity
      style={{
        height: 75,
        width: 75,
        borderRadius: 75,
        backgroundColor: buttonColour,
      }}
      onPress={() => {
        setRecording(!recording);
      }}
    />
```

Figure 7.1.12 – Camera UI

To use Camera methods, a reference to the component needs to be created. In this piece of code, it's stored in a state variable named 'camera'. The record button is a TouchableOpacity that sets a state variable whenever it's pressed.



```
const startRecording = async () => {
  if (camera) {
    await camera.recordAsync();
  }
};

useEffect(() => {
  requestCameraPermissions();
  requestMicrophonePermissions();
}, []);

useEffect(() => {
  if (recording) {
    setButtonColour("red");
    startRecording();
  } else {
    setButtonColour("white");
    if (camera !== null) {
      camera.stopRecording();
    }
  }
}, [recording]);
```

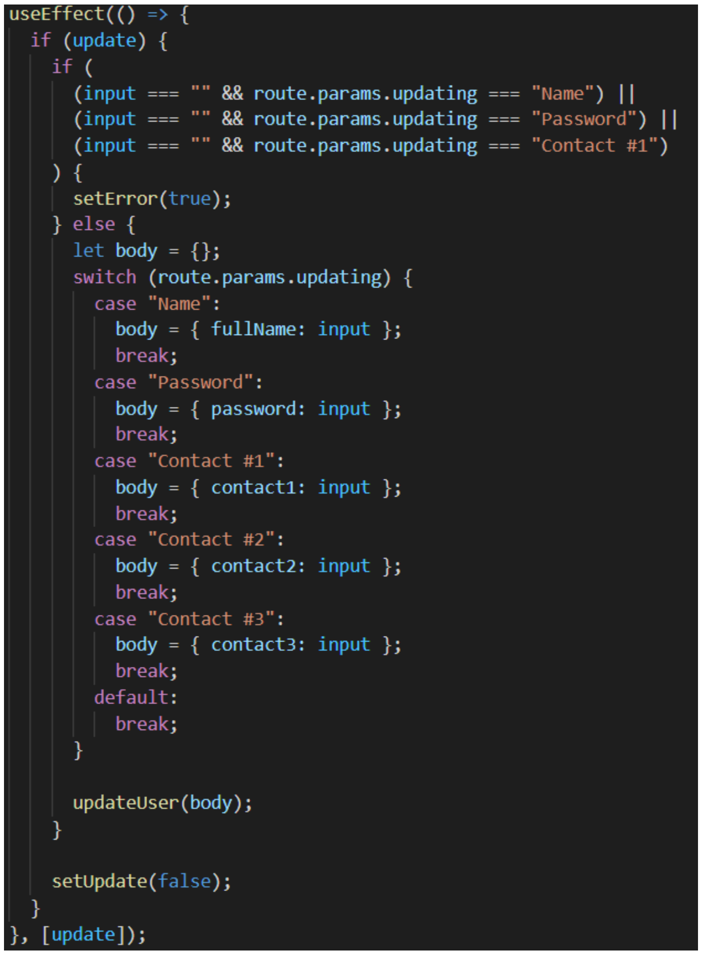
Figure 7.1.13 – Camera Recording Functionality

In the above piece of code, a function is defined named 'startRecording' which calls an asynchronous Camera method to start recording a video. This function is called every time the record button is pressed.

The User component contains information about the user currently logged in. It takes a user object as a prop and displays the user's name, phone number, and contacts they have linked to their account along with a profile picture. These are mostly displayed with Text components.

The Settings component contains numerous buttons which link to different screens. The first set of buttons, or TouchableOpacitys, when pressed will navigate the user to UpdateScreen. Each of these buttons are for updating a certain piece of user information, either the user's name, password, or individual contacts within the app. The relevant identifier is passed as a prop to UpdateScreen in order to display what the user is updating and send a correct PATCH request to the back-end.

Inside UpdateScreen, there is a switch statement that checks the value of the prop (under 'route.params.updating'). Depending on what the value of it is, it will generate the correct PATCH request body which is then passed to the updateUser() function.



```
useEffect(() => {
  if (update) {
    if (
      (input === "" && route.params.updating === "Name") ||
      (input === "" && route.params.updating === "Password") ||
      (input === "" && route.params.updating === "Contact #1")
    ) {
      setError(true);
    } else {
      let body = {};
      switch (route.params.updating) {
        case "Name":
          body = { fullName: input };
          break;
        case "Password":
          body = { password: input };
          break;
        case "Contact #1":
          body = { contact1: input };
          break;
        case "Contact #2":
          body = { contact2: input };
          break;
        case "Contact #3":
          body = { contact3: input };
          break;
        default:
          break;
      }

      updateUser(body);
    }

    setUpdate(false);
  }
}, [update]);
```

Figure 7.1.14 – UpdateScreen Switch Statement

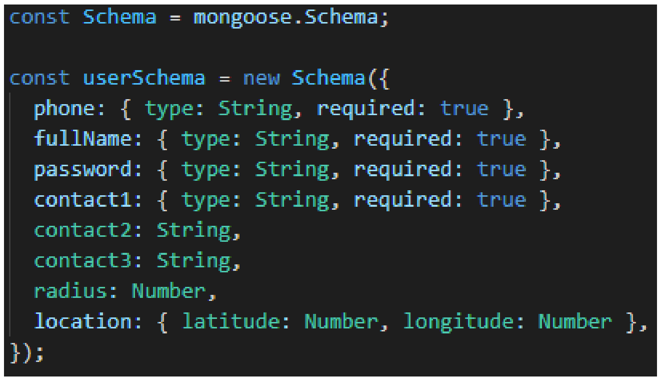
Apart from that, UpdateScreen contains a Text header, TextInput, and TouchableOpacity for inputting the updated field.

Back in Settings, the user is also given the choice to revisit SetupScreen to update their safe area, or they can log out if they wish. Pressing the latter will navigate them back to LoginScreen.

7.2 Backend

In the back-end, there are four primary endpoints: '/login', '/signup', '/user/:phone', and '/update/:phone'.

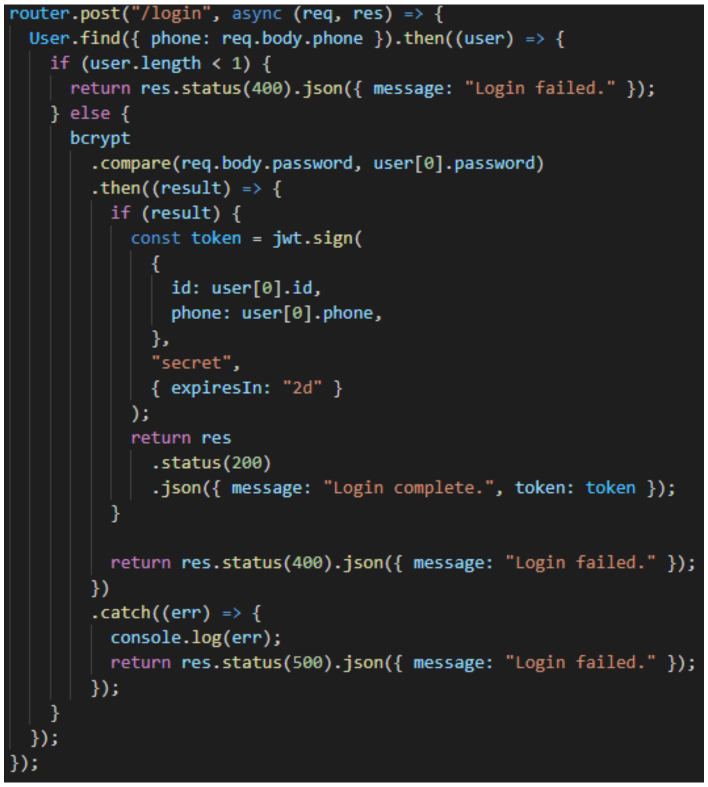
Using Mongoose schemas, the shape of documents can be defined. Each user schema is required to have a phone number, name, password, and primary contact. They can also have a second and third contact, as well as a location and radius which are referenced when reconfiguring a safe area.



```
const Schema = mongoose.Schema;

const userSchema = new Schema({
  phone: { type: String, required: true },
  fullName: { type: String, required: true },
  password: { type: String, required: true },
  contact1: { type: String, required: true },
  contact2: String,
  contact3: String,
  radius: Number,
  location: { latitude: Number, longitude: Number },
});
```

Figure 7.2.1 – User Schema

The /login endpoint is hit every time a user tries to login.

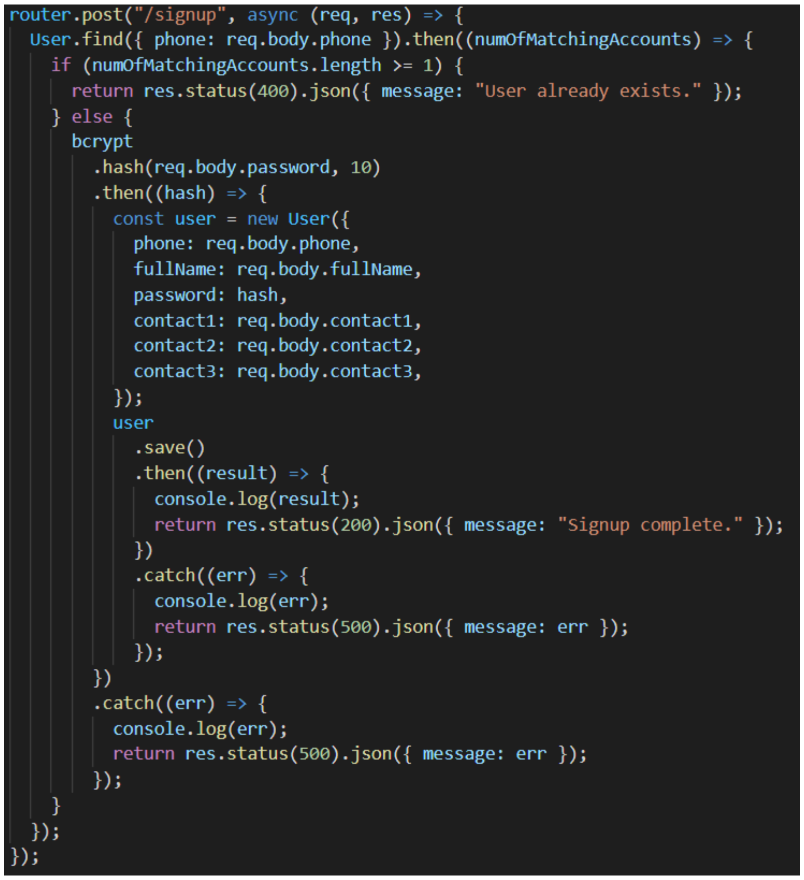


```
router.post("/login", async (req, res) => {
  User.find({ phone: req.body.phone }).then((user) => {
    if (user.length < 1) {
      return res.status(400).json({ message: "Login failed." });
    } else {
      bcrypt
        .compare(req.body.password, user[0].password)
        .then((result) => {
          if (result) {
            const token = jwt.sign(
              {
                id: user[0].id,
                phone: user[0].phone,
              },
              "secret",
              { expiresIn: "2d" }
            );
            return res
              .status(200)
              .json({ message: "Login complete.", token: token });
          }

          return res.status(400).json({ message: "Login failed." });
        })
        .catch((err) => {
          console.log(err);
          return res.status(500).json({ message: "Login failed." });
        });
    }
  });
});
```

Figure 7.2.2 – Login Endpoint

A User is searched for using the phone number embedded in the body of the request. The bcrypt library is used in the back-end for hashing of passwords [14]. Using bcrypt, a salt can be generated and used when hashing the plaintext password passed from the client. Here, bcrypt.compare() is used to check whether the input password matches the hash stored in the database. If it does, a JSON Web Token is signed (using the jsonwebtoken library [15]) and sent in the response to the client along with a success message. Otherwise, an error message is sent.

The /signup endpoint handles user signup.

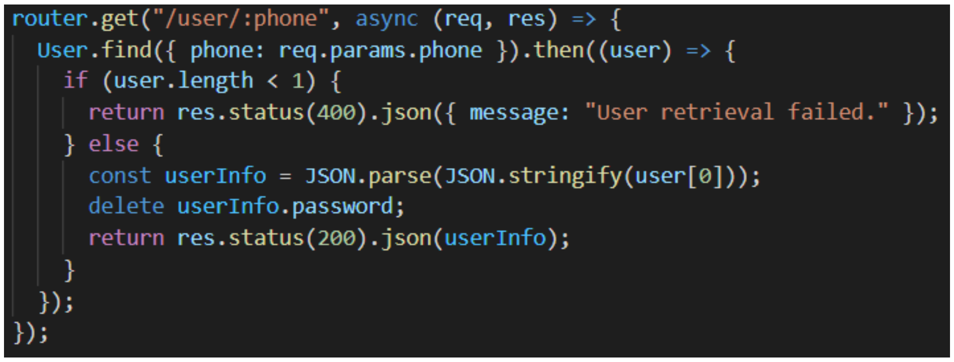


```
router.post("/signup", async (req, res) => {
  User.find({ phone: req.body.phone }).then((numOfMatchingAccounts) => {
    if (numOfMatchingAccounts.length >= 1) {
      return res.status(400).json({ message: "User already exists." });
    } else {
      bcrypt
        .hash(req.body.password, 10)
        .then((hash) => {
          const user = new User({
            phone: req.body.phone,
            fullName: req.body.fullName,
            password: hash,
            contact1: req.body.contact1,
            contact2: req.body.contact2,
            contact3: req.body.contact3,
          });
          user
            .save()
            .then((result) => {
              console.log(result);
              return res.status(200).json({ message: "Signup complete." });
            })
            .catch((err) => {
              console.log(err);
              return res.status(500).json({ message: err });
            });
        })
        .catch((err) => {
          console.log(err);
          return res.status(500).json({ message: err });
        });
    }
  });
});
```

Figure 7.2.3 – Signup Endpoint

The code here is similar to the login endpoint, though now it's checking if one or more users with the phone number already exist rather than checking if none exist, and now it's using bcrypt.hash() to hash the plaintext password with ten salt rounds before storing it in the database. After that's completed, the user is saved, and a 200 OK response is sent.

/user/:phone is a GET endpoint that returns the user matching a specified phone number.

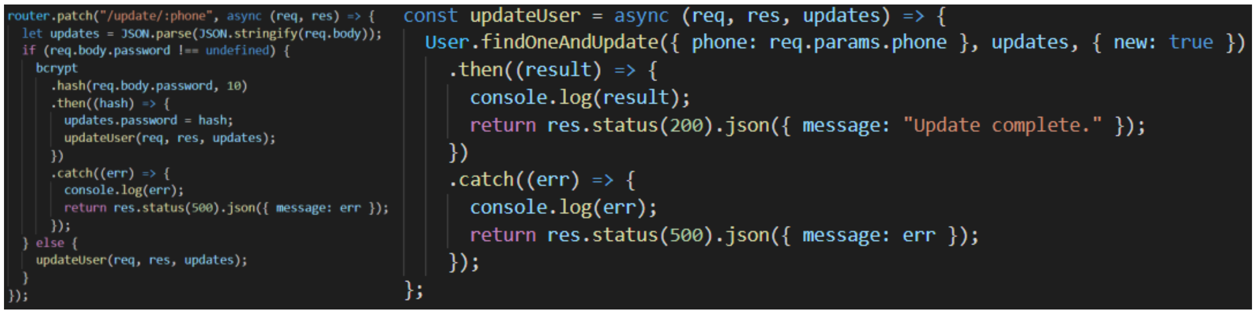


```
router.get("/user/:phone", async (req, res) => {
  User.find({ phone: req.params.phone }).then((user) => {
    if (user.length < 1) {
      return res.status(400).json({ message: "User retrieval failed." });
    } else {
      const userInfo = JSON.parse(JSON.stringify(user[0]));
      delete userInfo.password;
      return res.status(200).json(userInfo);
    }
  });
});
```

Figure 7.2.4 – GET User Endpoint

The user object is sent as part of the response, with the password excluded for security reasons.

Finally, there is the /update/:phone PATCH endpoint that is hit whenever a user's information needs to be updated.



```
router.patch("/update/:phone", async (req, res) => {
  let updates = JSON.parse(JSON.stringify(req.body));
  if (req.body.password !== undefined) {
    bcrypt
      .hash(req.body.password, 10)
      .then((hash) => {
        updates.password = hash;
        updateUser(req, res, updates);
      })
      .catch((err) => {
        console.log(err);
        return res.status(500).json({ message: err });
      });
  } else {
    updateUser(req, res, updates);
  }
});
```

```
const updateUser = async (req, res, updates) => {
  User.findOneAndUpdate({ phone: req.params.phone }, updates, { new: true })
    .then((result) => {
      console.log(result);
      return res.status(200).json({ message: "Update complete." });
    })
    .catch((err) => {
      console.log(err);
      return res.status(500).json({ message: err });
    });
};
```

Figure 7.2.5 – Update Endpoint

The updateUser() function is called either when the password is not being updated and not sent in the client request, or after the updated plaintext password has been hashed. It finds the user and updates its fields using the Mongoose findOneAndUpdate() function.

8 Ethics

While considering this project idea, among others, it was important to me to put some amount of scrutiny into the ethics of the subject matter.

Regarding Cognisent, I believe the largest ethical consideration to be had is surrounding the issue of surveillance and location tracking. While of course these are completely optional with location tracking performed at the user's discretion, and it being there to serve a solely ethical purpose, it is still worth considering whether the presence of it at the forefront of the application might have an impact on normalising the idea or presence of digital surveillance.

I think, in this case, any concerns there may be about this would be strongly outweighed by the benefits of a fully-fledged personal safety app where a user chooses to share their location. In this day and age, we are connected at all times to those around us and constantly sharing information about ourselves knowingly or unknowingly. In that sense, when location tracking is so commonplace even in applications where it's rarely or never truly needed, it would not be something I would be overly concerned about in an app that is largely built around it.

9 Conclusion

In conclusion, I believe I achieved many of the goals I had set out at the commencement of this project by developing a working, demonstratable prototype of a personal safety and emergency response mobile application.

I learned an extensive amount about full stack and mobile development, the tools and techniques used in those areas, and gained extensive experience with project management tools and delivering a large project in a professional manner.

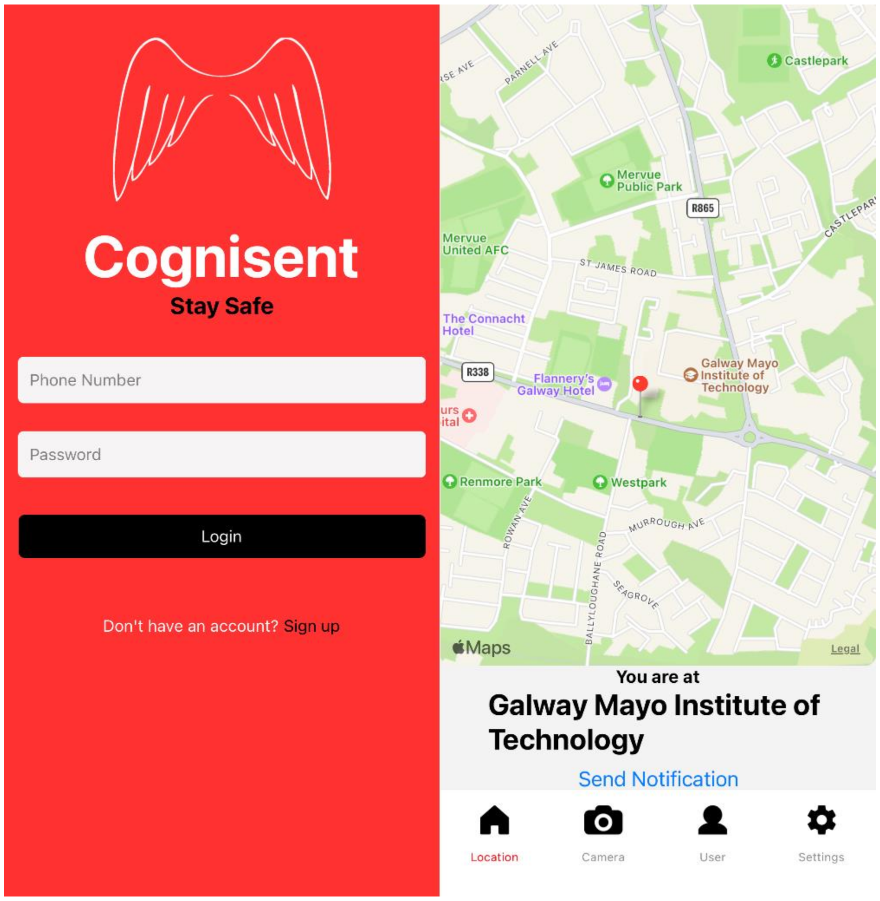


Figure 9.1 – Login & Home Screens

I think this project idea has endless potential and could be developed even further. I hope to continue research in this area in future and expect my learnings from this project to be valuable in my career after college.

10 References

[1] O'Reilly Media, "Chapter 1. What Is React Native?," [Online]. Available: https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html. [Accessed 10 April 2022].

[2] React Navigation, "GitHub - react-navigation/react-navigation," [Online]. Available: https://github.com/react-navigation/react-navigation. [Accessed 10 April 2022].

[3] Expo, "Introduction to Expo - Expo Documentation," [Online]. Available: https://docs.expo.dev/. [Accessed 10 April 2022].

[4] Node.js, "Introduction to Node.js," [Online]. Available: https://nodejs.dev/learn. [Accessed 12 April 2022].

[5] Express, "Express - Node.js web application framework," [Online]. Available: https://expressjs.com/. [Accessed 12 April 2022].

[6] MongoDB, "MongoDB & Mongoose: Compatibility and Comparison," [Online]. Available: https://www.mongodb.com/developer/article/mongoose-versus-nodejs-driver/. [Accessed 12 April 2022].

[7] TechTarget, "What is MongoDB?," [Online]. Available: https://www.techtarget.com/searchdatamanagement/definition/MongoDB. [Accessed 12 April 2022].

[8] AWS, "What is Amazon EC2?," [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. [Accessed 12 April 2022].

[9] Atlassian, " Agile epics: definition, examples, and templates," [Online]. Available: https://www.atlassian.com/agile/project-management/epics. [Accessed 15 April 2022].

[10] React Navigation, "Stack Navigator," [Online]. Available: https://reactnavigation.org/docs/stack-navigator/. [Accessed 16 April 2022].

[11] React, "Using the State Hook," [Online]. Available: https://reactjs.org/docs/hooks-state.html. [Accessed 16 April 2022].

[12] React, "Using the Effect Hook," [Online]. Available: https://reactjs.org/docs/hooks-effect.html. [Accessed 16 April 2022].

[13] React Navigation, "Tab navigation," [Online]. Available: https://reactnavigation.org/docs/tab-based-navigation/. [Accessed 16 April 2022].

[14] npm, "bcrypt - npm," [Online]. Available: https://www.npmjs.com/package/bcrypt. [Accessed 20 April 2022].

[15] npm, "jsonwebtoken - npm," [Online]. Available: https://www.npmjs.com/package/jsonwebtoken. [Accessed 20 April 2022].