UNIVERSITI MALAYA

ACADEMIC SESSION 2023/2024

SEMESTER 1

LECTURER: DR ONG

COMPANY NAME: SAFEWHERE

Phase 3b: Final Report

WIA2007 Mobile Application Development - TEAM OSY11

SGD and Selected Targets: SDG 16: Peace, Justice and Strong Institutions

Group Member	**Matric No.**
Amaan Geelani Syed (Project Leader)	S2191704
Ahmed Ibrahim Adem Hamed	S2173079
Alyssa Atmasava	S2171319
Muntaha Alsadoon	S2165149
Abdullahi Ibrahim	S2181462

Table of Contents

1. Introduction to the SDG and selected target

Achieving the Sustainable Development Goals (SDG 16) requires societies to be safe, just, and peaceful. Everyone has the right to feel safe while going about their daily lives, free from any sort of violence. As of 2021 the Malaysian CSOs in the SDG Alliance *(1)*, have given a 56.71% score overall with the lowest score being SDG 16 at 42%.

That is why as a group of students we acknowledged the importance of having an app to improve the safety of streets and public spaces. In our app, we also addressed key targets of SDG 16 such as reducing violence, protecting vulnerable groups, promoting justice, and ensuring participatory decision-making. Our app aims to allow people to report incidents in real time, allowing law enforcement to respond quickly and effectively. By doing so, we hope to create a safer and more secure society that everyone deserves.

1.1 Functional Requirements

Modules	**Functionalities**	**How it could achieve selected targets**
Safety Notification	1. **Safety alerts notification** for users, such as crime reports wherever the user goes. 2. **Notification history** 3. **Customizable notifications** (types of notifications they receive, set preferences for specific alert categories, and specify preferred communication channels (e.g., SMS, push notifications, email).	By notifying users of safety concerns or crime reports in the vicinity we can **significantly reduce all forms of violence and related death rates everywhere (Target 16.1)** to allow users to be more aware/prevent them from going into places with numerous alerts.
Maps	1. **Danger hotspots** to know what areas on a map of my neighborhood are currently reported as dangerous 2. **Parentally controlled geofencing** to define safe areas and receive notifications when their children leave these zones.	The geofencing mode functionality specifically for children could help achieve **Target 16.2 (End abuse, exploitation, trafficking and all forms of violence against and torture of children)** as parents are able to

	3. **Route planning** for the safest route.	monitor their childs whereabouts and define a specific safe zone for their child to be in. Hence achieving **Target 16.1 (Significantly reduce all forms of violence and related death rates everywhere)** as well along with the first and third functionality.
SOS button	1. **Emergency relay button** to alert family/friends and emergency services when I am in danger 2. **Automatic GPS location sharing** to predefined contacts 3. **Emergency call initiation** to emergency services **4. Panic Mode** enable users to press a "panic" button every few seconds when they are in a critical situation so that we can track their current response before activating emergency actions such as alerting emergency services & contacts	The SOS button is able to help achieve **Target 16.1 (Significantly reduce all forms of violence and related death rates everywhere)** as it provides immediate access to emergency assistance which can prevent situations from escalating to life-threatening levels.
Information Module	1. **Legal aid / Law information** (updates to local laws). **2. Crime reports and statistics.** **3. Safety Tips/ FAQs/**	The information module helps achieve **Target 16.3 (Promote the rule of law at the national and international levels and ensure equal access to justice for all) and Target 16.10 (Ensure public access to information and protect fundamental freedoms, in accordance with national legislation and international agreements)** as it provides an easy access for users to law information.
Report	**1. Crime specifications** to enable users to choose the type of incident/crime they want to report in their surroundings **2. Incident Reporting** **3. Cancel Report**	The report module helps achieve **Target 16.7 (Ensure responsive, inclusive, participatory and representative decision-making at all levels)** allowing all users to make a report easily.

2. Non-functional requirements

2.1 Security

To fulfill our security requirements, we must first identify the possible security risks that an application could have. In the context of our application, Safewhere, the main aspect under security risk is the data. To address this, all of our application's data is stored on the Firebase back-end platform. Firebase authentication automatically encrypts the data stored in its servers, and not even app administrators are able to view user sensitive details such as user passwords. Additionally, we are able to restrict the data access of each module of our application to be limited only to the data necessary for its functionality, allowing us to reduce the risk of possible data leakage.

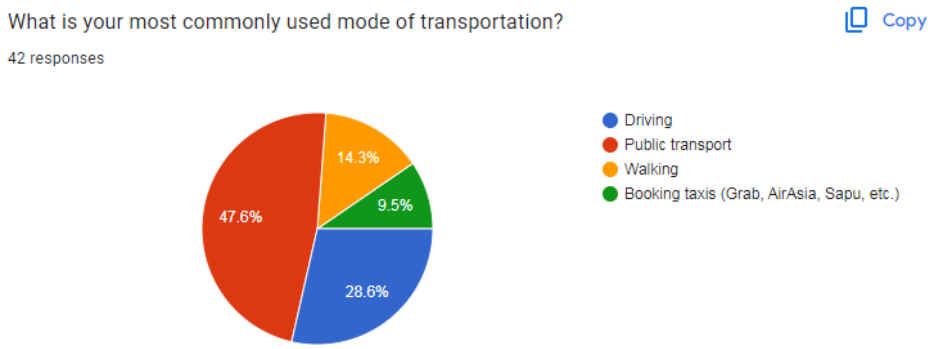
2.2 Portability

Portability is defined as the system's ability to work in different environments and how well its elements may be accessed or interacted with from two different environments. Our considerations of portability focuses on making sure our application is able to run on all kinds of hardware devices (provided that the software is Android). This is ensured by our use of constraint layouts on all of our activities and fragments, allowing for the elements in a page to adjust themselves by the relative size of the screen it's displayed on. Additionally, the use of the Firebase back-end platform ensures that every user can use our application on multiple devices while still retaining the same user experience. This is because all report, hotspot, geofencing, and user information for each user are loaded on log-in. One final consideration is that the development of our app has a minimum SDK of API 24, i.e Android 7.0, making our app

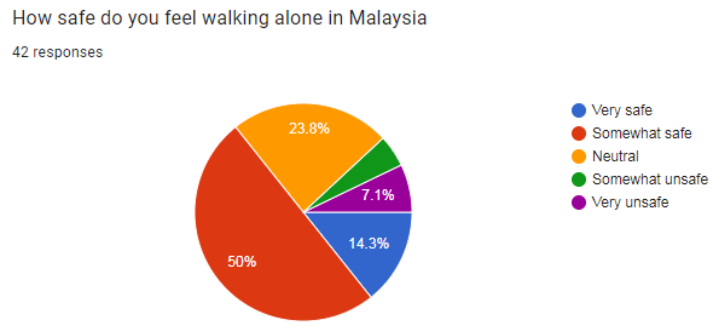
compatible with all newer versions up to Android 14 (up to 96.3% of devices), without losing out on any features available only in the later versions.

3. Survey Findings

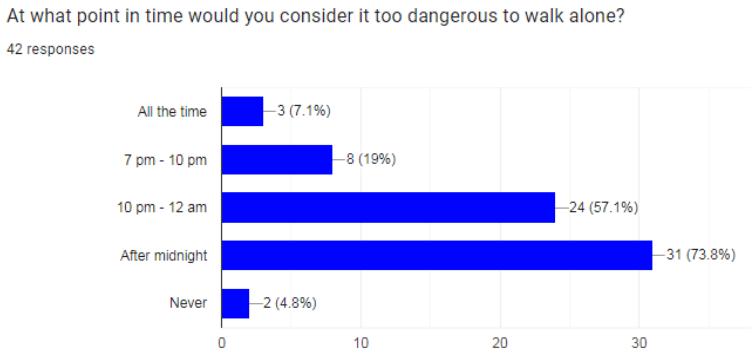
A survey was conducted by our team to ascertain the usefulness of the modules and features we chose to include in our mobile app, but it also served to give us an insight on what other features our targeted users might find useful. From the survey, the vast majority of our users are students between the age of 18 and 22.



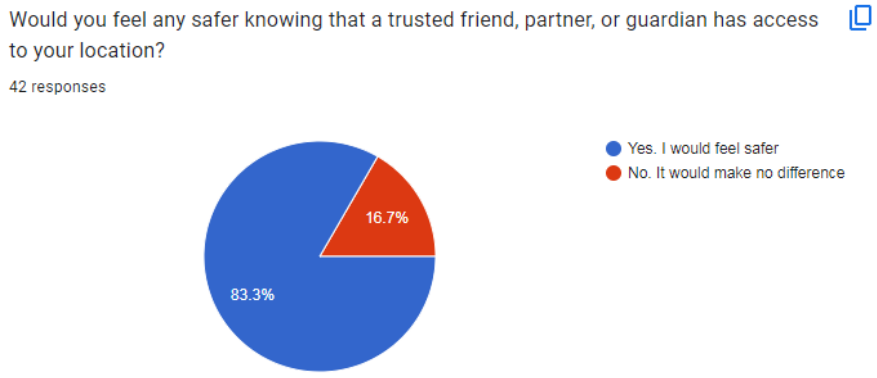
From the above chart we can see that only around 28.6% of students drive their own vehicles, while the remaining 71.4% of students are divided between public transport, walking, and booking taxis. Therefore 71.4% then turns out to be the percentage of students who could make use of our app, as it is this group of people that have their safety more at risk during their commutes.



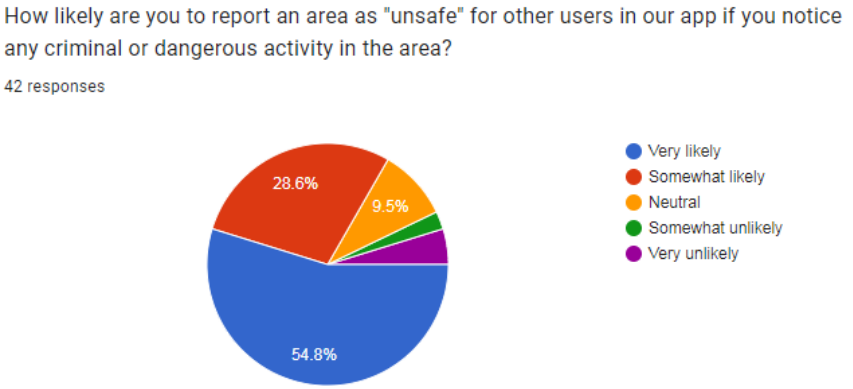
The majority of respondents feel somewhat safe walking alone in Malaysia, with the second biggest majority being those feeling neutral, and the third being very safe. This data goes to show that under normal circumstances, students generally feel safe while walking in Malaysia. With this information, we pivot our focus then into the rarer use cases where people's safety isn't guaranteed. With implementing features such as the Danger Hotspot and Route Planning, along with features included in the SOS Button module such as panic mode, we are able to allow our users to maintain that sense of safety even throughout these rarer, more dangerous circumstances.



From the data presented above, we find that safety while walking alone is a time dependent matter. Although the previous question gives us the impression that walking in Malaysia is generally a safe option, this data now shows us that that is not always the case, giving us more confidence in the potential usefulness of our application.



From our data, a staggering majority of our survey takers believe that they would feel much safer sharing their location with a friend, partner, or guardian. This gives us the confidence in knowing that our potential users will make use of the location sharing feature of our SOS button module.



From the above data point, we can infer that possible future users of our application are likely to be active when it comes to reporting any crimes or dangerous activity they witnessed. This allows us to easily update our crowdfunded Danger Hotspots data that shows which areas are susceptible to dangerous activities at what times, as well as helping us update our crime statistics in our information hub.

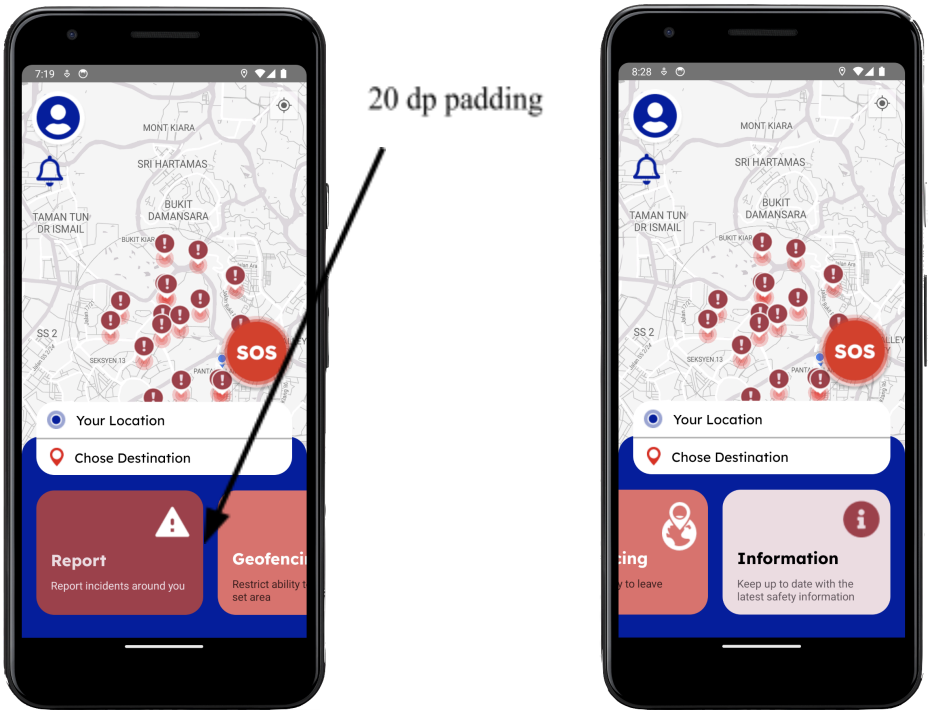
4. User Interface & App Flow

4.1 User Interface Design Principles

Layout and typography

In our UI design for our app we have implemented the recommended size for touch targets which is at least 48dp x 48dp. This allows all buttons in our app to be easily seen and clicked by a wide group of users.

Target Spacing



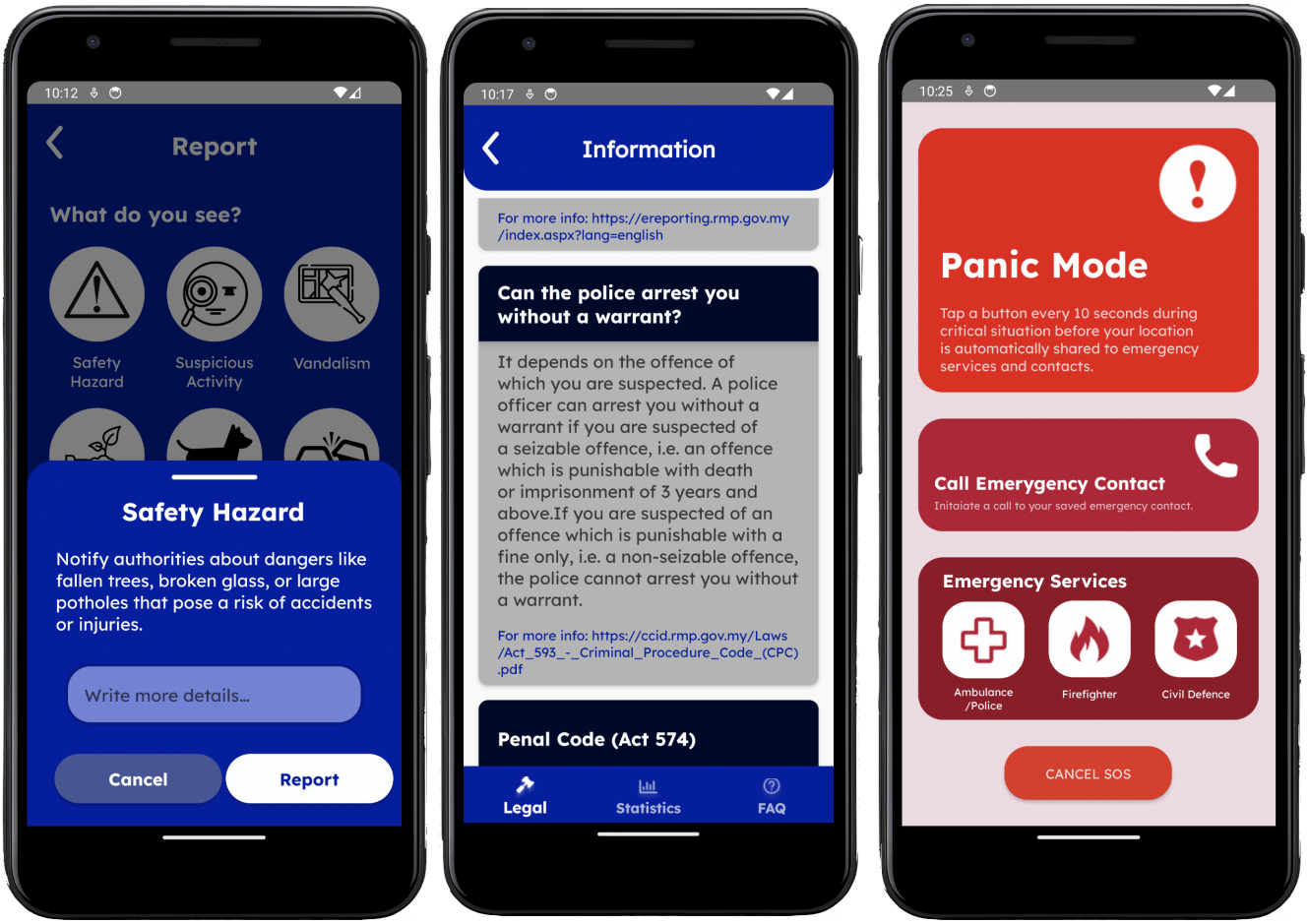
In our main home page we have designed our main features on the bottom with a 20dp padding between each button as seen above promoting balanced information density and usability.

Focus Order

As you can see in the UI of the main home page above the maps feature has the biggest UI as compared to the rest of the features as it is the main feature of our app. This allows users to focus on the maps feature first before anything else when opening the SafeWhere app. The SOS module is placed on top of the map, with a bright red contrast with the “SOS” text for quick focus and easy access for emergency situations. On the bottom part of the home page layout, you can access the 3 other main features which are the report, geofencing and information modules. It is placed based on the priority of each module with report being the most important out of all of them for quick access and information having the least priority.

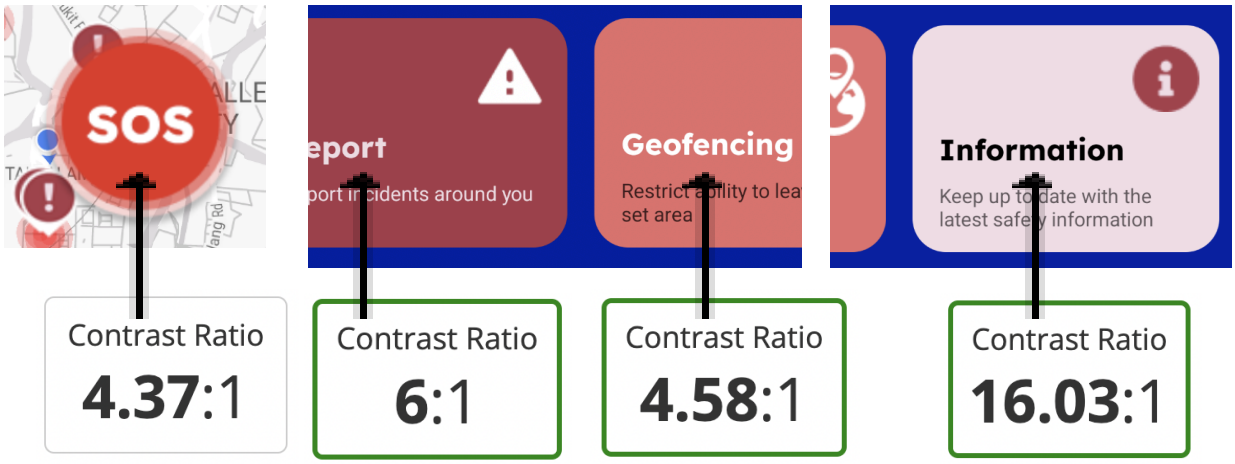
Writing

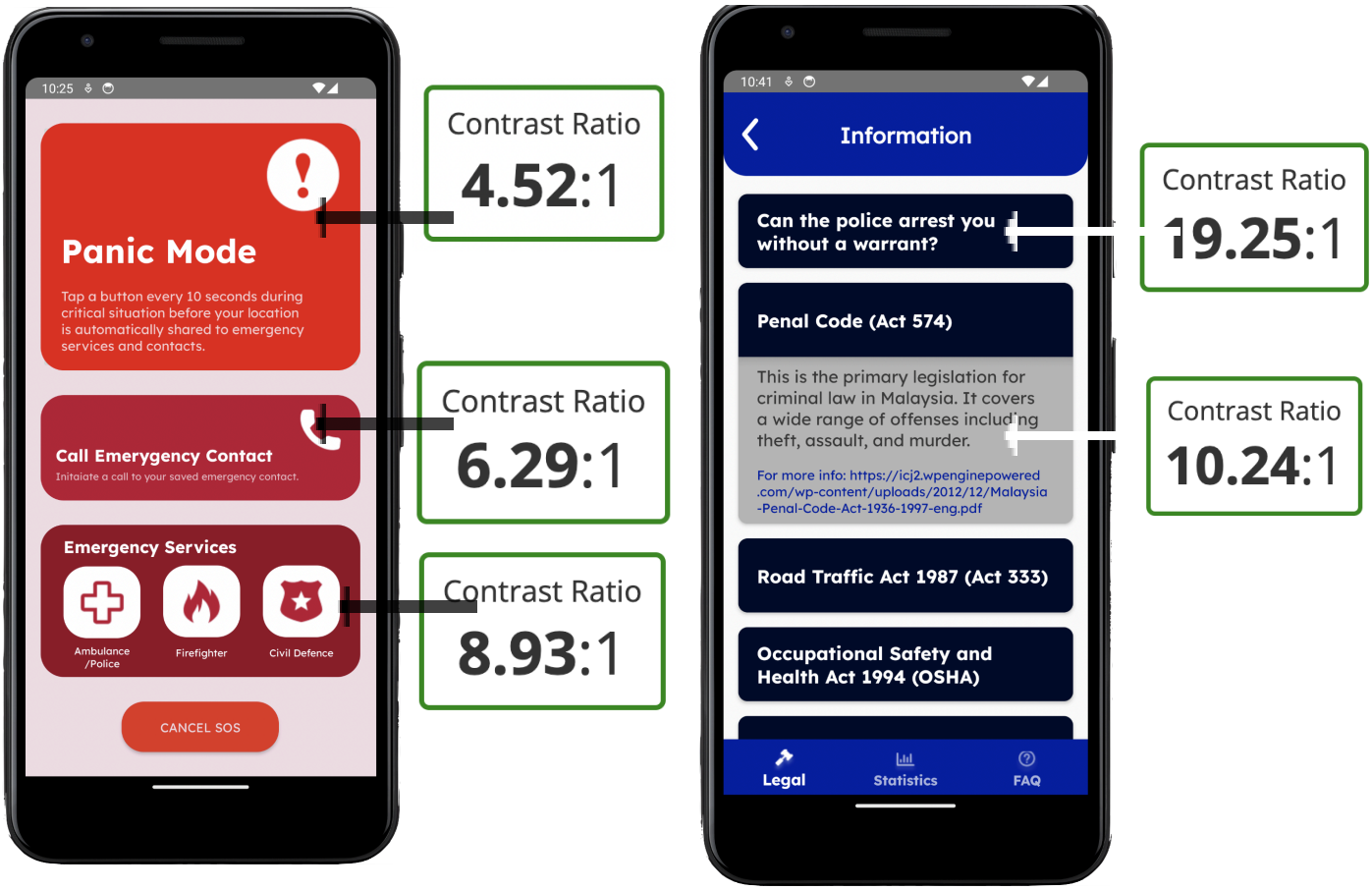
In the main home page, below the maps module, we have added captions below the title for the 3 buttons which are the report, geofencing and information buttons. Also keeping in mind the essential and non-essential elements. Keeping the essential elements bigger than the non-essential elements.



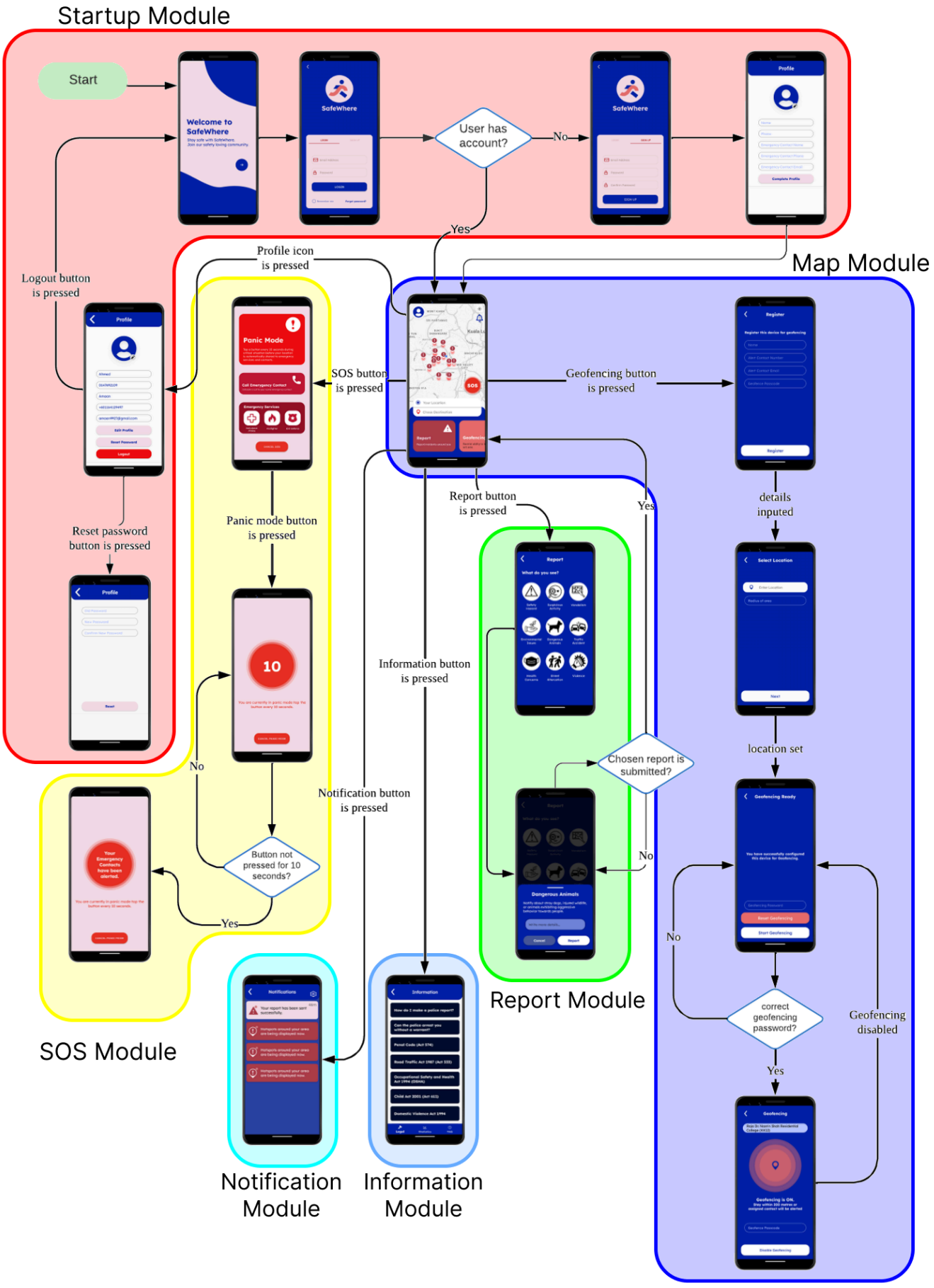
Color and Contrast

All our text and non-text elements have achieved the recommended W3C (World Wide Web Consortium) contrast ratio which is at least 3:1, as shown by the few examples below.





4.2 App Flow Diagram



5. System Implementation

5.1 Development environment

For our development process, we utilized a combination of design and development tools to ensure efficient collaboration and high-quality outcomes.

Design Phase:
During the design phase of our project, we employed Figma, a design collaboration environment. Figma allowed our team to collaboratively design the user interface and establish the app's overall flow and structure. Figma Design Collaboration Environment.

Implementation Phase:
Once the design phase was complete, we transitioned to the implementation phase using Android Studio Giraffe, along with Git/GitHub for version control and collaboration.

Collaboration with Git/GitHub: To ensure smooth collaboration and code management, we relied on Git and GitHub. Git allowed us to track changes, collaborate seamlessly, and manage multiple code branches. Our project's GitHub repository, which can be accessed via the Link To Public Github Repository, served as a central hub for our codebase, enabling team members to contribute, review, and merge code changes efficiently.

XML version: 1.0 **Java version:** 1.8 / Java 8
Our SDK Version: Our Target SDK was 34 and min SDK was 24.
Name Space: com.cyk29.safewhere
Database and Authentication: Firebase
Dependencies used:

```
dependencies {
    // Firebase BOM (Bill of Materials) for managing Firebase dependencies versions
    implementation platform('com.google.firebase:firebase-bom:32.7.0')

    // Firebase Authentication for managing user sign-in and authentication
    implementation 'com.google.firebase:firebase-auth:22.3.0'

    // Firebase Realtime Database for storing and syncing data in real-time
    implementation 'com.google.firebase:firebase-database:20.3.0'

    // Google Maps Services for displaying and interacting with maps
    implementation 'com.google.android.gms:play-services-maps:18.2.0'

    // Google Location Services for accessing the user's location
    implementation 'com.google.android.gms:play-services-location:21.0.1'
```

```
    // Utilities for Google Maps, such as marker clustering, heatmaps, etc.
    implementation 'com.google.maps.android:android-maps-utils:2.2.0'

    // Google Places API for rich place information and autocomplete functionality
    implementation 'com.google.android.libraries.places:places:3.3.0'

    // AppCompat library for backward-compatible versions of Android UI components
    implementation 'android.appcompat:appcompat:1.6.1'

    // Material Components for implementing Material Design UI components
    implementation 'com.google.android.material:material:1.11.0'

    // ConstraintLayout for complex UI layouts
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'

    // CardView for creating cards UI component
    implementation 'androidx.cardview:cardview:1.0.0'

    // Navigation components for easier navigation and UI management in the app
    implementation 'androidx.navigation:navigation-fragment:2.7.6'
    implementation 'androidx.navigation:navigation-ui:2.7.6'

    // JavaMail API for sending, receiving, and processing emails
    implementation 'com.sun.mail:android-mail:1.6.4'

    // WorkManager for managing background tasks in an efficient way
    implementation "androidx.work:work-runtime:2.9.0"

    // JUnit for unit testing
    androidTestImplementation 'junit:junit:4.13.2'
}
```

5.2 Coding style and convention

- We decided on a common code style for all Activities and Fragments and made the **template shown below**. Where each method has a specific function and each necessary method is called in the part of the Activity or Fragment lifecycle where it is needed.

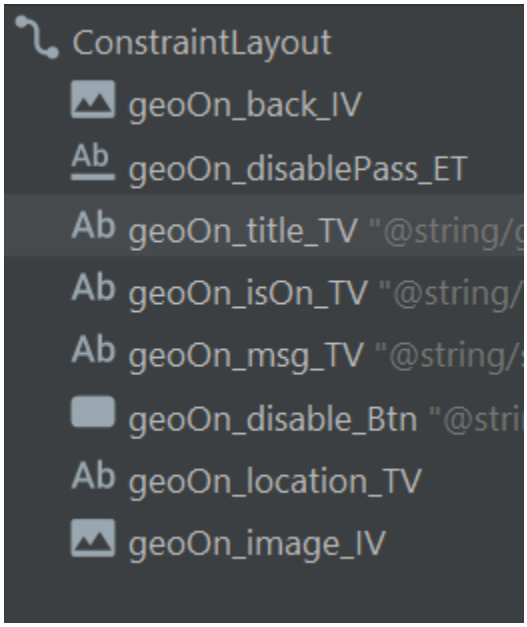
```
package com.cyk29.safewhere.modulename;

import . . .
```

```java
/**
 * Purpose of this class
 */
public class MyActivityName extends AppCompatActivity {
   private static final String TAG = "MyActivityName";

   // Global variables
   private Button nextButton;

   @Override
   protected void onResume() {
       super.onResume();
       method1UsedInOnResume();
       method2UsedInOnResume();
   }

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_my_activity_name);
       initializeUI();
       setUI();
       method1UsedInOnCreate();
       method2UsedInOnCreate();
   }

   /**
    * Docs explaining the method
    */
   private void method2UsedInOnResume(){
       // code and other method calls
   }
   private void method1UsedInOnResume(){}
   private void initializeUI(){}
   private void setUI(){}
   private void method1UsedInOnCreate(){}
   private void method2UsedInOnCreate(){}
}
```

- All **variable names** in the java were to follow the standard java Camel Case naming convention.
- **Reusable fragment**s were made which also follow reusability principles, for example: “HotSpotFragment” uses the same class and just changes the values based on the

specific marker the user clicks and loads the data for the specific report from the database.

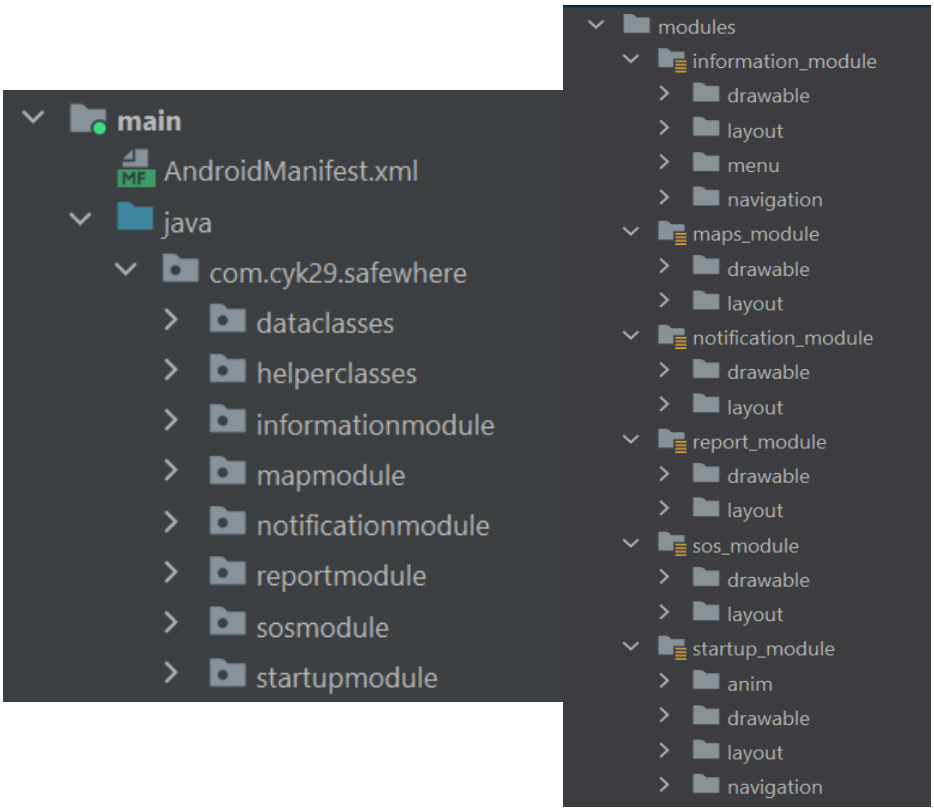
- **Utility classes** were created to make objects storing important information like User class, Report class, InformationItem class etc.
- **Helper classes** were made to ensure frequently reused code was not written more than once
- All **view IDs** followed a convention set by us of "fragmentName_purpose_Type". Example from fragment_geofencing_on is shown below



-

5.3 Source code

Our code was organized and divided into modules and subpackages to ensure separate modules were distinguished and people wouldn't interfere with other people's code.



Below is a flow chart showing all the different parts of our code and how they were divided into separate modules. Our app has 6 main modules: **startup, map, sos, report, information & notification**. These modules are helped by the **helperclasses** and **dataclasses** which act as utility classes.

- **Data Classes:** This package contains classes which describe various objects in our system, they are mainly used for storage of data and easily uploading and downloading data from the firebase database. This package contains the **User, Report, InfoItem, NotificationItem** and the **GeofencingInfo** classes. These contain various variables, constructors, setter and getter which help us manipulate data for various scenarios.
- **Helper Classes:** This package contains utility classes which are used by some or all of the modules. These Class contain reusable code which is often useful to call. The classes in this package are:
    - **EmailHelper class** - Which helps us send emails from any module using our company email and predesigned templates**.**
    - **GeofenceHelper class** - Which helps us set up all the necessary background tasks for the Google Maps Geofence API.
    - **HotSpotHelper class** - Which helps us retrieve all the reports from the database and display hotspot markers on the map whether they are for the local location or during route planning.
    - **NotificationHelper class**- Which helps us send high priority notifications when needed.
    - **ToastHelper class** - Which helps us display our redesigned toast messages.


com.cyk29.safewhere
.startupmodule
Java Package
startupmodule
IntroFragment
StartupActivity
loginsignup
LoginTabFragment
SignUpLoginFragment
SignUpTabFragment
ViewPageAdapter
profile
ProfileFragment
ProfileMainActivity
ResetPasswordFragment
Layout Files
startup_module
anim
drawable
layout
activity_main.xml
activity_profile.xml
fragment_intro.xml
fragment_login_tab.xml
fragment_profile.xml
fragment_reset_password.xml
fragment_sign_up.xml
fragment_sign_up_tab.xml
navigation
.mapmodule
Java Package
mapmodule
DestinationSelectFragment
HomeFragment
HotSpotFragment
MapsActivity
OnRouteFragment
geofencing
GeofenceBroadcastReceiver
GeofencingFragment
GeofencingOnFragment
Layout Files
maps_module
drawable
layout
activity_maps.xml
fragment_destination_select.xml
fragment_geofencing.xml
fragment_geofencing_on.xml
fragment_home.xml
fragment_hot_stop.xml
fragment_on_route.xml
.sosmodule
Java Package
sosmodule
PanicModeFragment
SosActivity
Layout Files
sos_module
drawable
layout
activity_sos.xml
fragment_panicmode.xml
.reportmodule
Java Package
reportmodule
MakeReportFragment
ReportMainActivity
Layout Files
report_module
drawable
layout
activity_report_main.xml
fragment_make_report.xml
.informationmodule
Java Package
informationmodule
FAQFragment
InfoAdapter
LegalInfoFragment
MainInformationActivity
StatisticsFragment
Layout Files
information_module
drawable
layout
activity_main_information.xml
fragment_f_a_q.xml
fragment_legal_info.xml
fragment_statistics.xml
info_item.xml
menu
navigation
.notificationmodule
Java Package
notificationmodule
NotificationMainActivity
Layout Files
notification_module
drawable
layout
activity_notification_main.xml
item_notification.xml
spinner_dropdown_item.xml
spinner_item_fixed_text.xml
.dataclasses
Java Package
helperclasses
EmailHelper
GeofenceHelper
HotSpotHelper
NotificationHelper
ToastHelper
.helperclasses
Java Package
dataclas es
GeofencingInfo
InfoItem
NotificationItem
Report
User

- **Startup Module:** This is the first module the user interacts with. At the core of the module is the **StartupActivity**. The activity initializes with the **IntroFragment**, which provides a welcome message. Upon completion of the introductory content, users are directed to the **SignupLoginFragment**. This fragment houses two tab fragments: **LoginTabFragment** and **SignUpTabFragment**. These fragments implement Firebase authentication to securely manage user sign-ins and registrations. For new users opting to sign up, the **SignUpTabFragment** collects necessary credentials and, upon successful registration, redirects to the ProfileActivity. Here, users are prompted to complete their profile details, which is facilitated through a specially designed user interface.

  Below is a snippet from **LoginTabFragment** to show how it works.

```
public void onViewCreated(View view, @Nullable Bundle savedInstanceState) {
   emailET = view.findViewById(R.id.loginTab_email_ET);
   passwordET = view.findViewById(R.id.loginTab_password_ET);
   setUpForgotPassword(view);
   setUpRememberMe(view);
   setUpLoginButton(view);
}
```

As you can see it contains 3 main methods.

```
/**
* Sets up the login button functionality and related actions.
*
* @param view The fragment's root view.
*/
private void setUpLoginButton(View view) {
   Button loginBtn = view.findViewById(R.id.loginTab_login_Btn);
   loginBtn.setOnClickListener(v -> handleLoginButtonClick());
}

/**
* Handles the click event for the login button.
*/
private void handleLoginButtonClick() {
   String email = emailET.getText().toString().trim();
   String password = passwordET.getText().toString().trim();
   if (email.isEmpty() || password.isEmpty()) {
       ToastHelper.make(getContext(), "Please fill in all fields", Toast.LENGTH_SHORT);
   } else {
       firebaseLogin(email, password);
       if (radioButton.isChecked()) {
           saveCredentials(email, password);
       }
   }
}
```

Here is how the login to firebase works.

```
/**
* Performs login using Firebase Authentication.
*
* @param email    The user's email.
* @param password The user's password.
*/
public void firebaseLogin(String email, String password) {
   mAuth.signInWithEmailAndPassword(email, password)
           .addOnCompleteListener(requireActivity(), task -> {
               if (task.isSuccessful()) {
                   startActivity(new Intent(getContext(), MapsActivity.class));
                   ToastHelper.make(getContext(), "Welcome Back!", Toast.LENGTH_SHORT);
```

```
            } else {
                ToastHelper.make(getContext(), "Authentication failed.", Toast.LENGTH_SHORT);
            }
        });
}
```

Within the **ProfileActivity**, there are two fragments that allow users to manage their profiles:
ProfileFragment: Where users can view their profile information, with options to edit or log out.
ResetPasswordFragment: Provides users with the functionality to change their password should they need to.
The ProfileFragment is equipped with buttons that allow users to log out, edit their profile, or reset their password, integrating essential user account management features within a single interface

- **Map Module:** Map module contains the **MapsActivity,** which serves as the main activity of this module. **MapsActivity** employs a Fragment Container View (FCV) to display various fragments that handle different functionalities depending on the user's needs and actions.

  Navigation
  Upon entry, the **HomeFragment** is presented, acting as a gateway to the different features of the app. It offers navigation to other modules and houses several buttons that enable users to access extended functionalities.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
   View view = inflater.inflate(R.layout.fragment_home, container, false);
   initializeUI(view);
   setUI();
   return view;
}
```

  This fragment mainly only contains buttons to navigate to other fragments and activities from other modules. This is done using the initializeUI() and setUI() methods.

  Destination Selection
  The **DestinationSelectFragment** utilizes the Google Places API to assist users in selecting a destination. This is an integral part of the user journey, facilitating the search and choice of locations within the app.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
   View view = inflater.inflate(R.layout.fragment_destination_select, container, false);
   if (!Places.isInitialized()) {
       Places.initialize(requireActivity().getApplicationContext(), getString(R.string.api_key));
   }
   setupAutocompleteFragment();
   setupStartButton(view);
   return view;
}
```

This Fragment first initializes the places api if it's not already initialized and then it sets up the autocomplete fragment which gives the users the places information using the setupAutocompleteFragment() method call. This fragment also sets up a start button to start the route using the setupStartButton() call.

Route Navigation

After the destination is selected, users transition to the **OnRouteFragment**. Here, the app calculates and displays the route from the user's current location to the chosen destination. All this is done on a separate thread using the **ExecutorService** class. Additionally, the **OnRouteFragment** provides access to SOS and report modules, further enhancing user safety and engagement. An "end route" button is available to reset the map and return the user to the **HomeFragment**.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
   View view = inflater.inflate(R.layout.fragment_on_route, container, false);
   initializeUI(view);
   setupRoute();
   return view;
}
```

This fragment contains the method initializeUI() which is used to set up the end button, sos button and the report button. It also calls the setupRoute() method which sets up the route in the map.

```
/**
* Sets up the route by fetching directions and updating the UI accordingly.
*/
private void setupRoute() {
   ((MapsActivity) requireActivity()).backBtn.setVisibility(View.GONE);
   LatLng origin = ((MapsActivity) requireActivity()).getUserLatLng();
   executeGetDirectionsTask(origin, destinationLatLng);
}
```

The setup route retrieves the UserLocation from the MapsActivity, and also disables the back button to prevent any errors. It then calleds the executeGetDirectionsTask() method.

```
/**
* Starts the getDirections task asynchronously  to get directions from the origin
* to the destination using the Google Directions API.
* Executor service is used to run the task on a separate thread.
* @param origin      Starting point of the route.
* @param destination Destination point of the route.
*/
private void executeGetDirectionsTask(LatLng origin, LatLng destination) {
   ExecutorService executorService = Executors.newSingleThreadExecutor();
   executorService.submit(() -> {
       List<LatLng> decodedPath = getDirections(origin, destination);
       if(decodedPath == null) {
                requireActivity().runOnUiThread(() -> ToastHelper.make(requireContext(),"Error Fetching Directions",
Toast.LENGTH_LONG));
           return;
       }
                                                  (requireActivity()).runOnUiThread(()       ->       ((MapsActivity)
requireActivity()).setPolylineOptionsForRoute(decodedPath));
   });
}
```

This method calls the getDirections() method on a separate thread from the ui thread as it may take a long time. Once the directions are retrieved a polyline is made on the map to show the route.

```
/**
* Fetches the directions from Google Maps Directions API.
* This method builds a URL with the origin, destination, and API key, and makes an HTTP request to the API.
* Which returns a decoded string of directions.
* If everything is error-free, the string is decoded into a list of LatLng points.
* @param origin      The starting point of the route (latitude and longitude).
* @param destination The ending point of the route (latitude and longitude).
* @return A list of LatLng points representing the path of the route.
*         Returns null if there is an error in fetching or parsing the data.
*/
private List<LatLng> getDirections(LatLng origin, LatLng destination) {. . .}

/**
* Converts the JSON response from the Google Maps Directions API into a list of LatLng points.
* This method parses the JSON string, extracts the polyline points, and decodes them into LatLng points.
*
* @param routeJson The JSON string returned by the Google Maps Directions API.
* @return A list of LatLng points representing the path of the route.
*         Returns null if there is an error in parsing the JSON string.
*/
private List<LatLng> convertJsonToLatLngList(String routeJson){. . .}

/**
* Decodes an encoded polyline string into a list of LatLng points.
* The encoding algorithm is a compact form of representing a list of coordinates as a single string.
*
* @param encoded The encoded polyline string.
* @return A list of LatLng objects representing the decoded polyline points.
*/
private List<LatLng> decodePolyline(String encoded) {. . .}
```

After that these 3 methods call each other to return a List<LatLng> for the polyline.

Hotspot Visualization

- The **MapsActivity** incorporates a **HotSpotFragment**, which uses a custom helper class to place the user's last known location on the map and visualize reported incidents within a 2500-meter radius as hotspot markers. The **HotSpotFragment** is a **BottomSheetDialogFragment** which shows the Hotspot description, up votes, downvotes, username and other details when each hotspot is clicked. This feature aims to inform users about their surroundings in real-time.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
   View view = inflater.inflate(R.layout.fragment_hot_stop, container, false);
   initializeUI(view);
   setDangerDescriptionText();
   setUserDescriptionText();
   return view;
}
```

This fragment calls on the initializeUI() method to setup the ui, the setDangerDescriptionText() and setUserDescriptionText() to retrieve the data from the database for that particular report and then set it accordingly

Geofencing Capabilities

- Users can set up a geofence via the **GeofencingFragment**, which, with the assistance of a **GeofenceHelper** class, activates geofencing functionality. The application allows users to enable or disable geofencing as needed in the **GeofencingOnFragment**, and upon activation, a marker is added on the map to indicate the geofence's boundary. All Alerts from the Geofence API are handled in the **GeofenceBroadcastReceiver** Class

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
   View view = inflater.inflate(R.layout.fragment_geofencing, container, false);
   initializeUI(view);
   initializeFirebaseAndPlaces();
   setupAutocompleteFragment();
   setupButtonListeners();
   return view;
}
```

In the **GeofencingFragment** the UI is initialized using the initializeUI() method. Then the Firebase and Places APIs are initialized with the initializeFirebaseAndPlaces() method. Then the autocomplete fragment for selecting the places is configured using the setupAutoCompleteFragment() method call. And the setupButtonListeners() is used to setup the various buttons.

```
@Override
public void onReceive(Context context, Intent intent) {
   GeofencingEvent geofencingEvent = GeofencingEvent.fromIntent(intent);
   if (geofencingEvent == null || geofencingEvent.hasError()) {
       Log.d(TAG, "onReceive: Geofencing event error or null");
       return;
   }
   // Extracting information from the intent
   String name = intent.getStringExtra("Name");
   String email = intent.getStringExtra("Email");
   String phone = intent.getStringExtra("Phone");
   handleGeofenceEvent(context, geofencingEvent, name, email, phone);
}
```

In the **GeofenceBroadcastReceiver** the Geofencing event is handled, first the user name, the alert email and phone are extracted from the intent. Then they are put in as parameters to the handleGeofencingEvent() along with the geofencingEvent itself and the context.

```
/**
 * Processes a geofencing event by sending notifications and recording the event
in Firebase.
 *
 * @param context           The context in which this handler is operating.
 * @param geofencingEvent   The geofencing event to be processed.
 * @param name              Recipient's name for personalized notifications.
 * @param email             Recipient's email for email notifications.
 * @param phone             Recipient's phone number for SMS notifications.
 */
private void handleGeofenceEvent(Context context, GeofencingEvent geofencingEvent,
String name, String email, String phone) {
   Date date = new Date(System.currentTimeMillis());
   sendNotification(context);
```

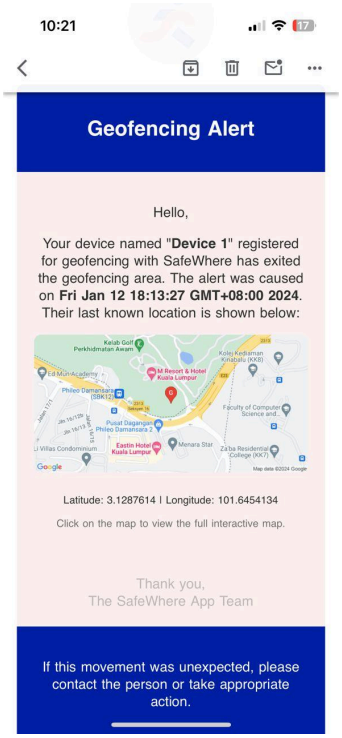
```
    sendEmailAndSms(context, geofencingEvent, name, email, phone, date);
    saveNotificationToFirebase(date);
}
```

In the handleGeofencingEvent() method a notification is sent to the user indicating that they have violated their geofence using the sendNotifcation() method which uses the **NotificationHelper** class.

After that an update is sent to the alert contact email and sms using the sendEmailAndSms() method which uses the **EmailHelper** class.

A user notification is also saved into the user database using the sendNotificationToFirebase().

Below is an example of the email sent using a custom-made template.



- **SOS Module:** The SOS Module is a critical safety feature of our mobile application, designed to provide users with immediate access to emergency services and panic mode. It is accessible via the HomeFragment or the OnRouteFragment of the Map Module.

  Primary Interface

  The primary interface within this module is the **SosActivity**, which presents the user with five essential buttons for various emergency scenarios.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_sos);
    getUserLocation();
    getUserInfo();
    initializeUI();
}
```

At the start of the activity the user location and user info is retrieved using methods of the same name. After that the UI is set and Initialized using the initializeUI() method.

Activate Panic Mode: The first button within **SosActivity** transitions the user to the **PanicModeFragment**. This mode is crucial for situations where the user feels threatened or requires urgent assistance.

```
panic.setOnClickListener(v -> getSupportFragmentManager().beginTransaction()
        .replace(R.id.sos_FCV, new PanicModeFragment())
        .addToBackStack(null)
        .commit());
```

A simple fragment transaction transitions the user to the **PanicModeFragment**.

Emergency Contact Communication: The next button directly redirects users to the phone application with a predefined emergency contact number ready to dial.
Emergency Services Communication: The subsequent three buttons are similarly programmed to dial three different emergency contacts, enabling users to reach out for help with a single tap.

```
/**
* Initiates a phone call to the specified phone number.
*
* @param number The phone number to call.
*/
public void call(String number){
   Intent dialIntent = new Intent(Intent.ACTION_DIAL, Uri.parse("tel:" + number));
   startActivity(dialIntent);
}
```

The call features can easily be done using the call() method. The Emergency contact details are retrieved from the database and the other 2 are pre-saved into the application.

```
/**
* Retrieves user information from the Firebase Realtime Database.
*/
private void getUserInfo(){
   String uid = FirebaseAuth.getInstance().getUid();
   if (uid != null) {
       DatabaseReference databaseReference = FirebaseDatabase.getInstance().getReference("users").child(uid);
       databaseReference.addListenerForSingleValueEvent(new ValueEventListener() {
           @Override
           public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
               currentUser = dataSnapshot.getValue(User.class);
           }
           @Override
           public void onCancelled(@NonNull DatabaseError databaseError) {
           }
       });
   }
}
```

The user into is retrieved using the getUserInfo method();

Panic Mode

The **PanicModeFragment** features a 10-second countdown timer, which serves as a last-chance mechanism for users to prevent false alarms. The timer can be reset by

tapping the screen. If the timer expires without user intervention, the application automatically notifies the user's emergency contacts via email and SMS.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
   View view = inflater.inflate(R.layout.fragment_panicmode, container, false);
   initializeUI(view);
   timer();
   setUI();
   return view;
}
```

The **PanicModeFragment** starts off by initializing the UI and then starting the timer using the timer() call and then setting up the UI accordingly using the setUI() call.

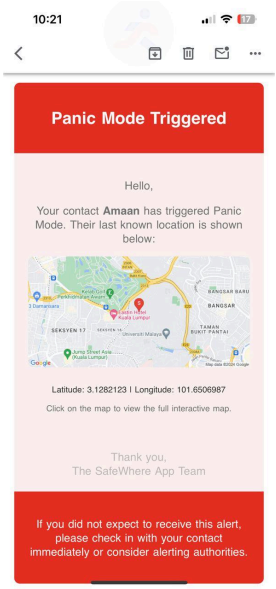
```
/**
* Handles actions to be taken when the countdown timer finishes.
*/
private void countFinished(){
   continueButton.setEnabled(false);
   AppCompatActivity parentActivity = (AppCompatActivity) getActivity();
   if (parentActivity != null) {
       if (parentActivity instanceof SosActivity) {
           SosActivity myActivity = (SosActivity) parentActivity;
           myActivity.alertContacts();
       }
   }
   countdownTextView.setText(R.string.your_emergency_contacts_have_been_alerted);
   countdownTextView.setTextSize(25.0f);
   countdownTextView.postDelayed(new Runnable() {
       @Override
       public void run() {
           if (isGrey) {
               countdownTextView.setTextColor(Color.WHITE);
           } else {
               countdownTextView.setTextColor(Color.GRAY);
           }
           isGrey = !isGrey;
           countdownTextView.postDelayed(this, 500);       }
   }, 0);
}
```

Once the timer is let finish the countFinished() method is called which calls the alertContact() method from the parent activity. And also displays the text on the screen and uses the postDelayed() method to show a flashing text.

```
/**
* Alerts emergency contacts by sending SMS messages and emails.
*/
public void alertContacts() {
   getUserInfo();
   if (currentUser != null) {
       String emergencyContactName = currentUser.getEcName();
       String emergencyContactPhone = currentUser.getEcPhone();
       String emergencyContactEmail = currentUser.getEcEmail();
       Location userLocation = getUserLocation();
       String subject = "Emergency Alert about your friend " + currentUser.getName() + "!";
        String body = "Dear "+emergencyContactName+",\nYour friend " + currentUser.getName() + " may be in danger, as
they initiated SOS in their SafeWhere App! Their last known location is: " ;
       String htmlBody;
       if (userLocation != null) {
           body += getGoogleMapsUrl(userLocation.getLatitude(), userLocation.getLongitude())+".";
                                       htmlBody    =    loadEmailTemplateWithValues(this,    currentUser.getName(),
Double.toString(userLocation.getLatitude()), Double.toString(userLocation.getLongitude()));
       } else {
           body += "Unknown.";
```

```
            htmlBody = loadEmailTemplateWithValues(this, currentUser.getName(), null, null);
        }
        sendSMS(emergencyContactPhone, body);
        EmailHelper emailHelper = new EmailHelper();
        emailHelper.sendEmail(emergencyContactEmail, subject, htmlBody);
    }
}
```

The alertContacts() method gets the userInfo and then also checks the location info. Then it creates the messages and the template for the email. At last is sends the email and sms using the **EmailHelper** class and the **sendSMS()** method respectively.
Below is an example of the email received by the user.



- **Report Module:** The Report Module is an integral part of our mobile application that facilitates user reporting of incidents or concerns. This module can be easily accessed through the HomeFragment or OnRouteFragment of the Map Module.

  Flow

  The entry point to the Report Module is the **ReportMainActivity**, which presents the user with nine distinct buttons, each corresponding to a different type of report.

```
/**
* Displays a BottomSheetDialogFragment for making a report based on the selected type.
*
* @param type The type of incident to report.
*/
private void makeOverlay(String type){
    MakeReportFragment makeReportFragment = new MakeReportFragment(type);
    dimBackground();
    makeReportFragment.show(getSupportFragmentManager(), "makeReportFragment");
}
```

The makeOverlay Method is called with the report type as a parameter. This method dims the background and shows the MakeReportFragment with the corresponding report description.

Reporting Process

Upon selecting the appropriate report type, the **MakeReportFragment** is invoked. This fragment is designed as a **BottomSheetDialogFragment**, providing a user-friendly overlay that enhances the overall user experience. Within this fragment, the user is shown a predefined description for the selected report type.

```
/**
* Sets up the user interface and attaches click listeners to buttons.
*/
private void setUI(){
   report.setOnClickListener(v -> {
       sendReport();
       requireActivity().finish();
   });
   cancel.setOnClickListener(v -> dismiss());
   switch (type) {. . .}
}
```

The setUI() sets the button and what they do. The report button calls on the sendReport() method and then finishes the report activity once done. The cancel button just dismissed the fragment.
The switch case statement with the report type as input set the title and description text according the the type of the report chosen by the user.

User Interaction

Users have the option to provide additional details about the incident in the **MakeReportFragment**. Two primary actions are available:

Cancel: Allows users to dismiss the report if they decide not to proceed.
Send: By pressing the send button, the report, along with any additional user-provided information, is submitted to the database.

Backend Integration

Once the send action is triggered, the report is transmitted and securely stored within the Firebase database. This action is also picked up by nearby users using the app on their phone and a HotSpot marker is placed on the marker.

```
/**
* Send a report to the Firebase Realtime Database.
*/
private void sendReport() {
   uid = FirebaseAuth.getInstance().getUid();
   String type = title.getText().toString();
   String userDescription = userDesc.getText().toString();
   String date = String.valueOf(System.currentTimeMillis());
   String latitude;
   String longitude;
   if(currentLocation == null){
       ToastHelper.make(getContext(), "Wait for location Data", Toast.LENGTH_SHORT);
       getLastLocation();
       return;
   } else {
       latitude = String.valueOf(currentLocation.getLatitude());
       longitude = String.valueOf(currentLocation.getLongitude());
   }
   DatabaseReference ref = FirebaseDatabase.getInstance().getReference("reports");
```

```
    String reportID = ref.push().getKey();
    Report report = new Report(reportID, uid, userName, type, userDescription, date, latitude, longitude);
    if (reportID != null) {
        ref.child(reportID).setValue(report);
    }
    sendNotification(uid,date);
    ToastHelper.make(getContext(), "Report Sent", Toast.LENGTH_SHORT);
}
```

The sendReport() method is called when the user presses the send button. String method first collects all the required information from the FirebaseAuth, the Views and the Location.
Then once it verifies the location is not null. It creates a database reference to the “reports” child and pushes to get the unique ID. Once that is done a new **Report (DataClass)** object is made with the given values and it is saved into the database. A confirmation notification is added to the notifications list and a confirmation toast is also shown using the **ToastHelper (HelperClass).**

- **Information Module:** The Information Module is a vital component of our mobile application, serving as a knowledge hub for users. It is accessible through the **HomeFragment** of the Map Module. The entry point to this module is the **MainInformationActivity**.

  User Interface
  MainInformationActivity incorporates a bottom navigation menu that facilitates easy switching between different types of information. This navigation menu includes three key options: Legal, Statistics, and FAQ. Selecting any of these options updates the Fragment Container View within the main activity to display the relevant fragment:

  LegalInfoFragment: Contains legal information relevant to the user.
  StatisticsFragment: Displays statistical data, possibly related to app usage or other pertinent information.
  FAQFragment: Provides answers to frequently asked questions by the app users.

```
/**
* Initialize UI components and set up navigation.
*/
private void initializeUI() {
    ImageView btnBack = findViewById(R.id.info_back_Btn);
    btnBack.setOnClickListener(v -> finish());

    NavHostFragment host = (NavHostFragment) getSupportFragmentManager().findFragmentById(R.id.info_FCV);
    NavController navController = Objects.requireNonNull(host).getNavController();

    setupBottomNavMenu(navController);
    formatMenuUI();
}
```

The initializeUI() method handles the setting up of the Navigation between fragments and the back buttons and also formats the MenuUI to be more clear and visible using the formatMenuUI() method call.

Functional Design

Each fragment within the Information Module operates uniformly. They all utilize a **RecyclerView** with a custom adapter known as **InfoAdapter**, which works in conjunction with a custom-designed card view layout defined in **info_item.xml**. This setup pulls information from the database and populates the **RecyclerView** with interactive elements.

```
/**
* Initializes the UI components of the fragment.
* @param view The view inflated for the fragment.
*/
private void initializeUI(View view) {
   recyclerView = view.findViewById(R.id.recyclerView);
   adapter = new InfoAdapter(legalInfoList);
}

/**
* Sets up the UI components such as the RecyclerView.
*/
private void setUI() {
   recyclerView.setLayoutManager(new LinearLayoutManager(getContext()));
   recyclerView.setAdapter(adapter);
}
```

The initializeUI() and the setUI() methods in each fragment handle the setting up of the recycler view with the custom adapter made called “InfoAdapter”.

Interaction

Each item within the **RecyclerView** is interactive. Upon tapping an info item, it expands to reveal detailed descriptions and, when applicable, a hyperlink for users to obtain further information. This interactive design ensures users can navigate through the information efficiently and find what they need with minimal effort.

```
/**
* Sets up the click listener for the item view.
*/
private void setupItemViewClickListener() {
   itemView.setOnClickListener(v -> toggleDetailsVisibility());
}

/**
* Toggles the visibility of layoutDetails with animation.
*/
private void toggleDetailsVisibility() {
   ViewGroup parent = (ViewGroup) itemView.getParent();
   TransitionManager.beginDelayedTransition(parent, new ChangeBounds());
   layoutDetails.setVisibility(layoutDetails.getVisibility() == View.GONE ? View.VISIBLE : View.GONE);
}
```

The setupItemViewClickListener() and the toggleDetailsVisibility() methods make sure the interaction with each info item works properly by expanding and shrinking the CardView.

Backend Integration

The module's backend is designed to fetch and display dynamic content from the database, ensuring that the information presented is always current and relevant.

```
/**
* Fetches data from Firebase and updates the RecyclerView adapter.
*/
private void fetchDataFromFirebase() {
```

```
    DatabaseReference databaseReference = FirebaseDatabase.getInstance().getReference("legalInfo");
    databaseReference.addValueEventListener(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
            legalInfoList.clear();
            legalInfoList = new ArrayList<>();
            for (DataSnapshot snapshot : dataSnapshot.getChildren()) {
                InfoItem infoItem = snapshot.getValue(InfoItem.class);
                legalInfoList.add(infoItem);
            }
            adapter.updateData(legalInfoList);
        }

        @Override
        public void onCancelled(@NonNull DatabaseError databaseError) {
            Log.e(TAG, "onCancelled: " + databaseError.getMessage());
        }
    });
}
```

The fetchDataFromFirebase() method is used to retrieve data from the database and update the **RecyclerView** accordingly. It basically creates a database reference to the "legalInfo", "statsInfo" or "FAQInfo" depending on which fragment it is in. Then it sets up an addValueEventListener which updates the list of infoItem. A custom DataClass **InfoItem** is used to store information about each information item.

- **Notification Module:** This module can be accessed through the **HomeFragment** in the Map Module. The main interface for this module is the **NotificationMainActivity**, which serves to display a variety of notifications to the user.

  Interface Overview
  Upon accessing the Notification Module, users are greeted by **NotificationMainActivity**, which features a clean and intuitive layout.

  <u>Functionality</u>
  NotificationMainActivity employs a **RecyclerView** to list the notifications, which is populated by a custom adapter named **NotificationAdapter**. The notifications themselves are defined by item_notification.xml for each individual entry.

```
/**
* Initializes the user interface components, including the RecyclerView and adapter.
*/
private void initializeUI() {
    RecyclerView notificationRV = findViewById(R.id.notif_RV);
    notificationRV.setLayoutManager(new LinearLayoutManager(this));
    notificationItems = new ArrayList<>();
    adapter = new NotificationAdapter(notificationItems);
    notificationRV.setAdapter(adapter);
}
```

The initializeUI() method sets up the RecyclerView, the Adapter and the list of the **NotificationItems (DataClass).**

<u>Dynamic Content Loading</u>
Notifications are fetched from a Firebase database and are filtered based on user selection through a spinner with dropdown options defined in

spinner_dropdown_item.xml. Users can filter the view to show all notifications or sort them by categories such as Danger, Report, or Geofencing.

```
/**
* Sets up the filter spinner by configuring its adapter and item selection handling.
*/
private void setupFilterSpinner() {
   Spinner filterSpinner = findViewById(R.id.notfi_filter_Spinner);
   ArrayAdapter<CharSequence> filterAdapter = ArrayAdapter.createFromResource(this,
           R.array.notification_filter_options, R.layout.spinner_item_fixed_text);
   filterAdapter.setDropDownViewResource(R.layout.spinner_dropdown_item);
   filterSpinner.setAdapter(filterAdapter);
   filterSpinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
       @Override
       public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
           String selectedFilter = parent.getItemAtPosition(position).toString();
           switch (selectedFilter) {
               case "All":
                   loadNotificationItems("All");
                   break;
               case "Danger":
                   loadNotificationItems("dangerzone");
                   break;
               case "Report":
                   loadNotificationItems("report");
                   break;
               case "Geofencing":
                   loadNotificationItems("geofencing");
                   break;
               default:
                   break;
           }
       }
       @Override
       public void onNothingSelected(AdapterView<?> parent) {
           // Do nothing
       }
   });
}
```

The setupFilterSpinner() method sets up the Spinner and makes sure it show to notification according to the filter.

Backend Interaction

The activity interacts with the Firebase backend to dynamically load and display notifications relevant to the user.

```
/**
* Loads notification items from Firebase based on the selected filter type.
*
* @param selectedType The type of notifications to load ("All," "Danger," "Report," or "Geofencing").
*/
private void loadNotificationItems(String selectedType) {
   String uid = Objects.requireNonNull(FirebaseAuth.getInstance().getCurrentUser()).getUid();
   DatabaseReference myRef = FirebaseDatabase.getInstance().getReference("users")
           .child(uid).child("notifications");
   myRef.addListenerForSingleValueEvent(new ValueEventListener() {
       @Override
       public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
           notificationItems.clear();
           for (DataSnapshot snapshot : dataSnapshot.getChildren()) {
               NotificationItem item = snapshot.getValue(NotificationItem.class);
               if (item != null) {
                   if (selectedType.equals(item.getType()) || selectedType.equals("All")) {
                       notificationItems.add(item);
                   }
```

```
                }
            }
            sortNotifications(notificationItems);
            adapter.notifyDataSetChanged();
        }
        @Override
        public void onCancelled(@NonNull DatabaseError databaseError) {
            Log.d(TAG, "onCancelled: " + databaseError.getMessage());
        }
    });
}
```

The loadNotificationItems() method is used to create a database reference which checks the notifications of the current user and iterates through them, and depending on the selected type of notifications adds the notifications to the list. It then updates the adapter which shows the updated list in the recycler view.

6. Testing functional and non-functional features

6.1 Functional Features Testing

- **Notification Module:** Tests were designed to ensure timely delivery of notifications under various network conditions. Manual testing involved assessing the clarity and accuracy of the notification messages. Also tasks which would trigger the notification module were carried out manually to see the accuracy of timely notifications and credibility.

- **Maps:** The maps functionality was tested for accuracy in routing, ease of use in navigation, and the correct rendering of map data across different devices. Reports were also issued to see how the hotspots are updated on the map which also proved to be quite accurate. Live location on the app was also in place alongside making sure heatpoints are issued on the exact location of users on the app to make sure far away reports/hotspots are not issued.

- **SOS Button:** Stress testing was performed to ensure the SOS feature triggered the appropriate response without fail, even under heavy load. Test scenarios included the button's responsiveness and subsequent actions after a fixed time of unresponsiveness. Load testing was also performed to determine how the application behaves under normal and peak loads. The emergency contact email function was also tested to make sure it works in all conditions provided availability of network.

- **Information Module:** Links to relevant websites regarding various information and content were clicked to and tested to make sure they lead to the desired website. Information was also cross checked to make sure they are accurate and up to date. Guidelines provided regarding the app were also tested to make sure steps are correct and fully understandable.

- **Report Module:** In this feature security assessments were carried out to identify potential vulnerabilities and to ensure that data protection measures were effective. Tests were also carried out to ensure fake reports can easily be identified.

6.2 Non-Functional Requirement Testing

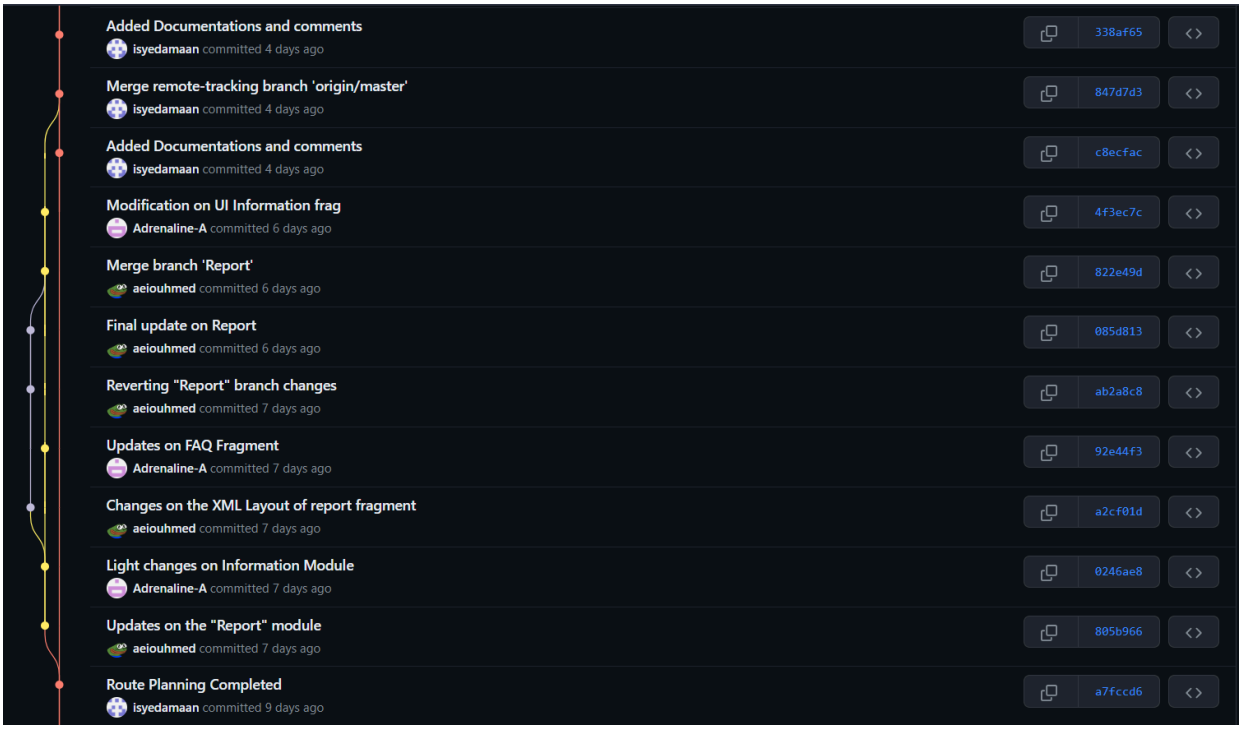
- **Security:** For this requirement, we tried testing the app and trying to extract the user passwords, API Keys and other User/ Non-User Sensitive data. Due to the robustness of the firebase database and authentication security, and also putting in logical rules for database access as well as encryption of the password and the API Keys made sure that all the safety requirements of our app have been reached.

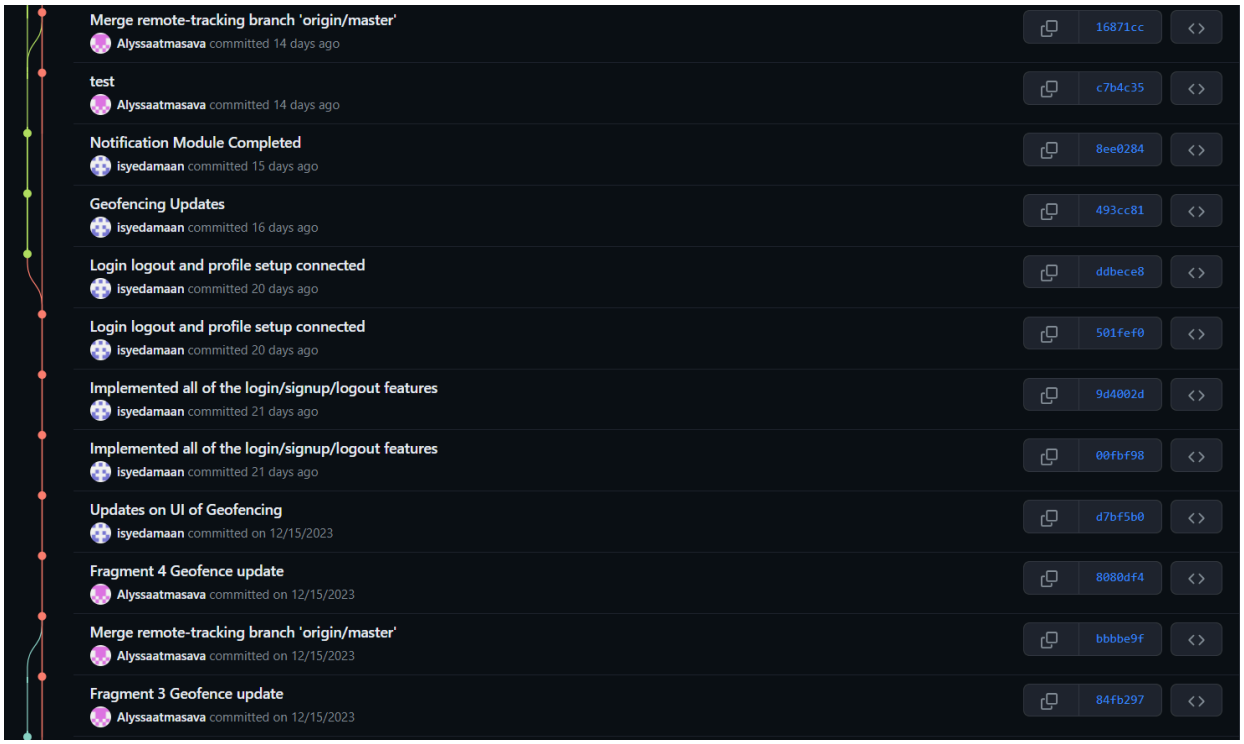
- **Portability:** For this requirement, we tested the app of various different types of devices with various different screen sizes and hardware. Thorough testing was done to make sure all our UI was visible and usable on all devices and all the features worked normally provided all permissions were given. The database also proved reliable providing users the same experience on all devices.

7. Task Distribution

Name	Contributions
Amaan Geelani Syed	Phase 1, Phase 2, Startup module logic and backend, Map Module (Complete), Database Implementation, Final Report
Alyssa Atmasava	Phase 1, Phase 2, UI Design (Figma), Startup Module Layout, SOS Module (Complete), Final Report
Ahmed Ibrahim Adem Hamed	Phase 1, Phase 2, Report Module (Complete) , Final Report
Abdullahi Ibrahim	Phase 1, Phase 2, Information Module (Complete), Final Report
Muntaha Alsadoon	Phase 1, Phase 2, UI Design (Figma) , Notification Module (Complete)

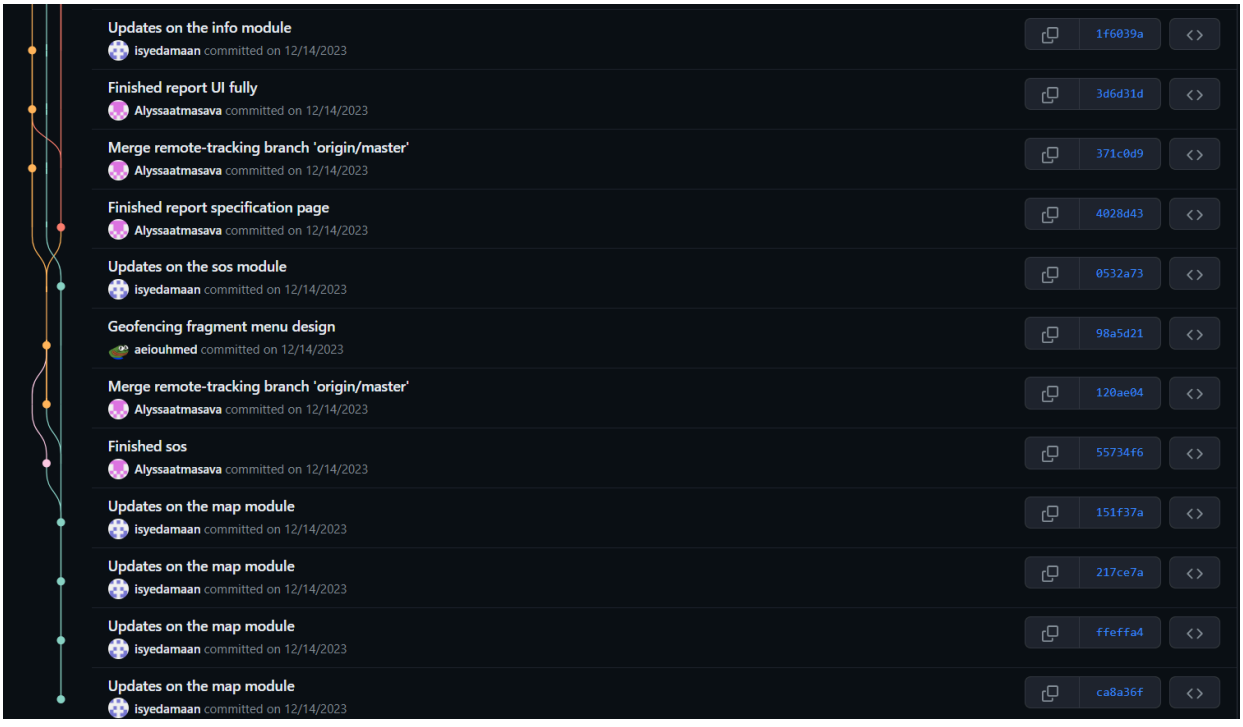
7.1 Git commit logs

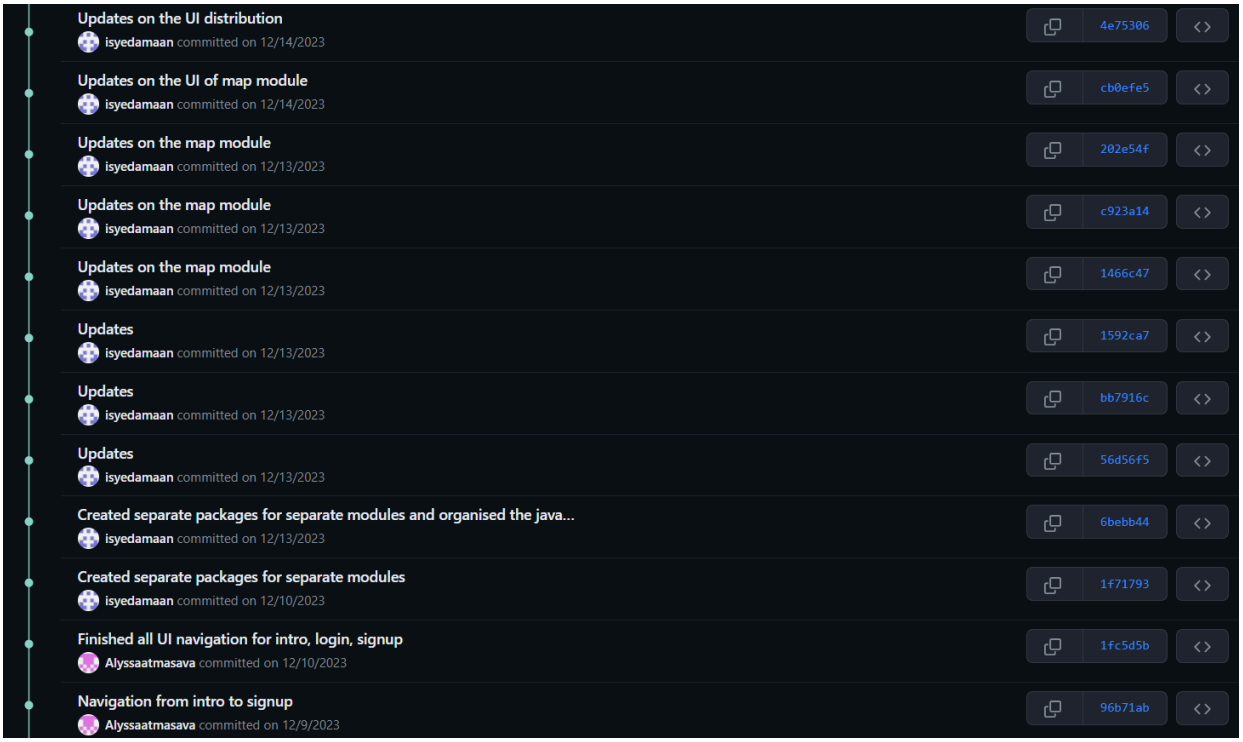


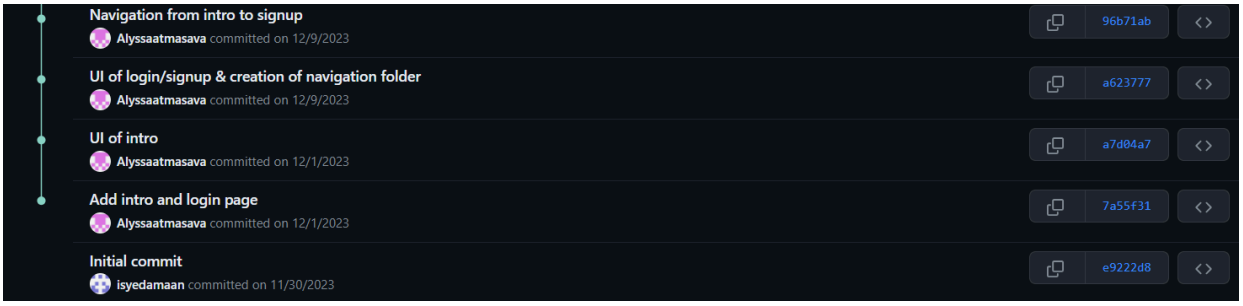

Merge remote-tracking branch 'origin/master'
Alyssaatmasava committed 14 days ago
16871cc
test
Alyssaatmasava committed 14 days ago
c7b4c35
Notification Module Completed
isyedamaan committed 15 days ago
8ee0284
Geofencing Updates
isyedamaan committed 16 days ago
493cc81
Login logout and profile setup connected
isyedamaan committed 20 days ago
ddbece8
Login logout and profile setup connected
isyedamaan committed 20 days ago
501fef0
Implemented all of the login/signup/logout features
isyedamaan committed 21 days ago
9d4002d
Implemented all of the login/signup/logout features
isyedamaan committed 21 days ago
00fbf98
Updates on UI of Geofencing
isyedamaan committed on 12/15/2023
d7bf5b0
Fragment 4 Geofence update
Alyssaatmasava committed on 12/15/2023
8080df4
Merge remote-tracking branch 'origin/master'
Alyssaatmasava committed on 12/15/2023
bbbbe9f
Fragment 3 Geofence update
Alyssaatmasava committed on 12/15/2023
84fb297


Updates on UI of Geofencing
isyedamaan committed on 12/15/2023
2f61bec
Merge remote-tracking branch 'origin/master'
isyedamaan committed on 12/15/2023
e6d91ef
Fragment 3 Geofence
Alyssaatmasava committed on 12/15/2023
d3b336c
Merge remote-tracking branch 'origin/master'
isyedamaan committed on 12/15/2023
da34945
Fragment 2 Geofence
Alyssaatmasava committed on 12/15/2023
03c48cd
Merge remote-tracking branch 'origin/master'
isyedamaan committed on 12/15/2023
200cfc8
Fragment 1 Geofence
Alyssaatmasava committed on 12/15/2023
b5a495f
Merge remote-tracking branch 'origin/master'
isyedamaan committed on 12/15/2023
a20e77e
Merge remote-tracking branch 'origin/master'
Alyssaatmasava committed on 12/15/2023
e73581d
Finished Information Module
Alyssaatmasava committed on 12/15/2023
266db13
Updates on profile
isyedamaan committed on 12/15/2023
bfdfc51


Fixed merge conflicts and committed geofencing xml
aeiouhmed committed on 12/15/2023
f0336e6
Fixed merge conflicts and committed geofencing xml
aeiouhmed committed on 12/15/2023
fa4ce36
Geofencing merged with rest of program
aeiouhmed committed on 12/15/2023
29d73ac
Merge remote-tracking branch 'origin/master'
aeiouhmed committed on 12/15/2023
35b7ea9
Updates on the notification module
isyedamaan committed on 12/15/2023
eb75b2a
Updates on the notification module
isyedamaan committed on 12/15/2023
412a349
Updates on the report module
isyedamaan committed on 12/15/2023
c6969eb
updates
Alyssaatmasava committed on 12/14/2023
56d625c
Finished profile UI
Alyssaatmasava committed on 12/14/2023
672cded
Finished notification ui
Mimi077 committed on 12/14/2023
e564b23
Updates on the info module
isyedamaan committed on 12/14/2023
5efdf2c
Merge remote-tracking branch 'origin/master'
isyedamaan committed on 12/14/2023
31ed7be


Updates on the info module
isyedamaan committed on 12/14/2023
1f6039a
Finished report UI fully
Alyssaatmasava committed on 12/14/2023
3d6d31d
Merge remote-tracking branch 'origin/master'
Alyssaatmasava committed on 12/14/2023
371c0d9
Finished report specification page
Alyssaatmasava committed on 12/14/2023
4028d43
Updates on the sos module
isyedamaan committed on 12/14/2023
0532a73
Geofencing fragment menu design
aeiouhmed committed on 12/14/2023
98a5d21
Merge remote-tracking branch 'origin/master'
Alyssaatmasava committed on 12/14/2023
120ae04
Finished sos
Alyssaatmasava committed on 12/14/2023
55734f6
Updates on the map module
isyedamaan committed on 12/14/2023
151f37a
Updates on the map module
isyedamaan committed on 12/14/2023
217ce7a
Updates on the map module
isyedamaan committed on 12/14/2023
ffeffa4
Updates on the map module
isyedamaan committed on 12/14/2023
ca8a36f





9. References

(1) *Malaysia CSO SDG Alliance - 2021 People's Scorecard on Sustainable Development Goals Report.*

(2) *Google Forms survey findings -* *Personal Safety App Survey (Responses)*