

Final Project Report

Quartermaster

TU856

BSc in Computer Science

**Ethan Holohan**

**C20322553**

**Colette Kirwan**

School of Computer Science

Technological University, Dublin

**11/04/25**

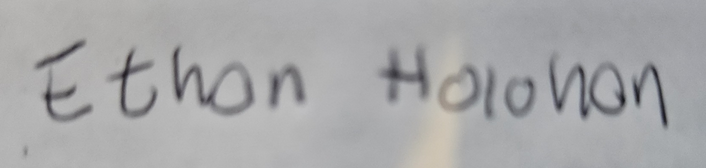
Abstract

Airsoft is a popular and growing hobby, with new people playing and trying the hobby every year. However, Airsoft is a complex game, with each site having its own sets of rules and games, and as a result, issues arise from players not fully knowing the rules and site layouts, leading to rule-breaking and potential injury.  
  
This application aims to help both newer players and experienced ones better understand the rules of each site and the games played there, as well as help staff enforce rules and maintain safety for players.  
  
The way this app intends to do this is by making sure that all the information that players need for each game is available at all times during the game by adding in a tracking feature that allows players to see where each other are for gameplay and for staff to see where all players are on the site, to help rule enforcement by using geofencing technology.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Ethan Holohan**

**11/11/2024**

Acknowledgements

I would like to thank my project supervisor Colette Kirwan for all the help and stellar guidance they offered on this report.

I would also like to thank the FYP course co-ordinator Jack O’Neill for all his work organising the FYP course this year.

I would also like to thank the Players at the Airsoft site I work with and my boss John for helping with testing and requirements gathering.

Table of Contents

[1. Introduction 7](#_Toc119925914)

[1.1. Project Background 7](#_Toc119925915)

[1.2. Project Description 7](#_Toc119925916)

[1.3. Project Aims and Objectives 7](#_Toc119925917)

[1.4. Project Scope 7](#_Toc119925918)

[1.5. Thesis Roadmap 7](#_Toc119925919)

[2. Literature Review 8](#_Toc119925920)

[2.1. Introduction 8](#_Toc119925921)

[2.2. Alternative Existing Solutions to Your Problem 8](#_Toc119925922)

[2.3. Researched Technologies 8](#_Toc119925923)

[2.4. Other Research you’ve done 8](#_Toc119925924)

[2.5. Existing Final Year Projects 8](#_Toc119925925)

[2.6. Conclusions 8](#_Toc119925926)

[3. System Design 9](#_Toc119925927)

[3.1. Introduction 9](#_Toc119925928)

[3.2. Software Methodology 9](#_Toc119925929)

[3.3. Overview of System 9](#_Toc119925930)

[3.4. Other Sections 9](#_Toc119925931)

[3.5. Conclusions 9](#_Toc119925932)

[5. Prototype Development 11](#_Toc119925938)

[5.1. Introduction 11](#_Toc119925939)

[5.2. Prototype Development 11](#_Toc119925940)

[5.3. Other Sections 11](#_Toc119925941)

[5.4. Conclusions 11](#_Toc119925942)

[6. Issues and Future Work 12](#_Toc119925943)

[6.1. Introduction 12](#_Toc119925944)

[6.2. Issues and Risks 12](#_Toc119925945)

[6.3. Plans and Future Work 12](#_Toc119925946)

[6.3.1. GANTT Chart 12](#_Toc119925947)

[Bibliography 13](#_Toc119925948)

# 1. Introduction

## Project Background

Airsoft is a sport/hobby that is played all around the world, typically played an airsoft site, and it is very similar to paintball, with two teams fighting over an objective decided upon by game marshals, whose responsibility it is to make sure players are safe and that they are following the rules of the game. However, this is quite a challenge. Many players don't know the site well, and some sites are extensive. Many players get lost or confused throughout the game, which leads to a bad experience, and since there are not that many marshals compared to the number of players, it is hard to keep track of all players, which can lead to safety issues.  
Additionally, although Airsoft has been around for a while, there has been very little digital integration until recent years.  
  
This app aims to address these issues by giving marshals more tools to monitor the status of the game and its players and to improve player experience by easing navigation and providing more information about the rules of the current game and general safety rules aswell.

## Project Description

Quartermaster is a mobile application designed for Airsoft players and Airsoft Marshals. Its purpose is to give airsoft players more information so they can play the game better with less confusion over rules and to help marshals make sure that players remain safe throughout the game day. It intends to do this using new technology on smartphones to track users on the site and display accurate site maps of each site.

## Project Aims and Objectives

The main aims of the quartermaster project are:

* To enhance players understanding of the game and reduce confusion
* To provide a way of improving team coordination and encouraging team play
* To ensure that players always have a way of communicating with a marshal in an emergency.
* To Provide players with a easy method of tracking their stats, such as number of re spawns used and average win rate.

To Ensure that these goals are achieved robust requirements gathering will be carried out in addition to Live User testing, multiple field tests will be carried out using the app and if users report that the app achieves its main goals well, then I will know that these goals has been achieved.

The main objectives of Quartermaster are:

* To implement a method of users being able to contact marshals across the site
* To have all the information about a game and the site being easily accessible through the application
* To have a way of tracking players stats through geofencing and GPS tracking.

To ensure that these objectives are achieved, a robust set of requirements gathered from requirements gathering will be created, to have a set success criteria for this application.

## Project Scope

Quartermaster's Scope is focused on enhancing the Airsoft experience for both players and marshals, however for this project it is also important to understand what this app doesn't intend to do

In Scope:

* this app is intended to act as a tracking app during an Airsoft event
* this app is intended to help people navigate in a large Airsoft event
* this app is intended to help people understand the rules and objectives of Airsoft
* this app is intended to help enforcement of these rules

Out of Scope:

* this app is not intended to get rid of marshals or their role in the Airsoft experience
* this app is not intended to be used as a tracking app outside of the Airsoft arena

## Thesis Roadmap

In the rest of this report i will go through:

Chapter 2 – Literature Review: This chapter is concerned firstly with the project background and any existing solutions in order to get a good idea of what direction the project should go. then it will go into detail on research done on various technologies and related studies, it also looks at some previous projects for inspiration, and finally how I created my set of requirements for the application.

Chapter 3 – System Design: This chapter is concerned with how the app was designed using the set of requirements to create a good framework of how to create this app.

Chapter 4 – Software Development: This chapter is concerned with the development of the application, firstly in the prototype stage and secondly in the full fledged application, what changes were made etc.

Chapter 5 – Testing and Evaluation: This chapter is concerned with the testing of various stages of development of my application including unit, system, integration and end user testing.

Chapter 6 – Conclusions and future work: This chapter is concerned with going through the reflections on the project, the work done, the work to be done and the challenges and success encountered in development.

# Literature Review

## 2.1. Introduction

The Purpose of this chapter is to mainly determine a base set of user requirements, this is achieved by doing research into the following areas, firstly exploring existing solutions to the problem and examining any strengths or weakness that they had. Secondly by researching technologies that are going to be used in the creation of the app and deciding on the best fits for it, finally by gathering requirements from the potential end users of the application using interviews and surveys. Then using this research to construct a set of requirements to be fulfilled.

## 2.2. Alternative Existing Solutions to My Problem

In this section two existing solutions to the problem are examined, first by looking at their strengths and where they went right, and then by looking at their weakness and where they failed, this is so we can use their good ideas in the project while avoiding the pitfalls they have encountered.

**Ares Alpha**

Ares Alpha is a pre-existing cross-platform application that allows users to create or join a session created by a user. Then, that user can form teams and assign players to teams and squads within those teams. Users can see where everyone on their squad is located via GPS functionality and can see marks created on the map by their squad leader, and this is to facilitate the co-ordination of Milsim Airsoft games.  
The main strength of this app is its ability to assist team management and its ability to show where squad mates are at all times. This is due to the complex team management system which helps split up players into squads for the Milsim with squad leaders able to see the names and who is in the team allowing for more effortless co-ordination out in the field. Secondly, its ability to show where squad mates are at all times is another strength, as in Milsim-type games, players get lost quickly due to the large field.  
The main weakness of this app is it is hard to understand and use. When trying to create games on the app, nowhere does it explain how to do it, in fact you cannot create full games on the app at all, it needs a desktop computer for this, which is not ideal for short form games made on the fly. The UI is also challenging to use, and it is hard to understand what is happening; the app has many features buried under confusing options menus. Another weakness is its unreliability; it only supports specific versions of Android, and as many as half of the players could not connect to the app on the game day due to these issues, in addition joining session was difficult due to the fact that you could only join via QR code which many peoples phones had issues scanning aswell.

**Airsoft Force Tracking**

Airsoft Force Tracking is a pre-existing application that allows users to create or join a session produced by an admin; then, players can join a team and see where their team members are, put down markers on the map, and see what game is being played.  
The main strength of this app is the ability of the players to interact with the map simply by placing markers on the map to coordinate their team more effectively. Creating new sessions on the app is simple, allowing it to be used easily on a skirmish day. Another strength of the app is the ability to download after the game, the paths, and tracking of players to see where players went during the game; this would be very helpful to the marshal, allowing them to improve game modes via data gained from this.  
  
The main weakness of this app is that it has no safety features. There are no ways for players to contact the marshals in case someone gets injured in the field, which is very important in larger Milsim games where the field is large, and the marshals are spread out over the entire area. This could lead to people being seriously hurt with no contact with a marshal or team-mate, a worst-case scenario in Milsim games.  
I've learned from analysing these previous solutions that while some reasonable solutions and ideas from these apps tend to be missing essential parts of the whole solution or have serious reliability issues encountered often throughout. When developing my solution, I need to remember where these apps went right, how to include them in my project, where they went wrong, and how to avoid these apps' issues.

**Conclusions**

From examining these two apps some features that will be included in Quartermaster from these apps are, the continuous tracking of team mates from ARES ALPHA as it increases co-ordination between players and allows marshals to monitor the entire field from where ever they are on it.

From Airsoft Force Tracking a similar session system will be used with marshals creating a session and players joining it with a simple code as it is a simple and easy to follow method for doing this and doesn't lead to the complexity and reliability issues Ares alpha had with the Qr codes.

## 2.3. Technologies researched

**Introduction**

in this section we will cover technologies researched and other domain research.

**Native App vs Web app**

When creating this app, there are two routes the project could go down: either a native app using Java or Koltin with Android Studio, or I could create a web app using react.js. Both have their advantages over each other.  
A web app is easily multiplatform because web apps run on the web and do not need a local installation. They are very easy to make compatible with a wide range of devices with only a single codebase. Secondly, Web apps have a more straightforward development process.  
A native app has a lot more functionality, with greater access to sensors on the phone, most notably for this project, GPS, and has better performance versus web apps, which is important in my app for preserving battery life over extended periods in Milsim games[1]  
For the reasons listed above, in my project, Quatermaster will use a native app because my app needs access to accurate GPS to work correctly, and the improved performance will make the battery last longer in the field.

**Java vs Koltin**

Java is a popular programming language for developing native apps on an Android system; it is used on over 3 billion devices and has extensive support on nearly all Android devices; it also has extensive online documentation due to its many years in use.  
Koltin is a newer Android-focused programming language for developing apps using Android Studio. It has many new features compared to Java, such as Null Safety and a lot less boilerplate code. It is compatible with Java, being able to use the same libraries designed for Java, and can run on devices that have Java installed [2]  
Due to these factors, I will use Koltin over Java in my project. Even though I am more familiar with Java overall, the cross-compatibility and the ability to use the same libraries as Java with the additions of new features and less code needed makes me feel like Koltin is the best choice for this project

**No SQL vs SQL**

There are two directions that I can go for a database in this project: using No SQL with Firebase or using a MySQL database with GCP.  
No SQL is a database that stores data differently from standard SQL databases; instead of storing data using rows and tables in a relational model like SQL, it instead uses other ways based on the model: either document, key-value, wide-column, or graph.  
The advantages of No SQL databases are that they are pretty flexible and can deal better with high user loads; there are also different types of No SQL databases, as said before, that can be used individually or together in a multi-model database; this gives No SQL its flexibility[3].  
The advantages of an SQL database are that it is more secure than a regular NO SQL database and is more likely to be used by the government or a secure industry as it meets the criteria of specific standards. They are also better for transactional databases and enterprise resource planning systems.[4]  
Due to these factors, I will be going with a No SQL database with Firebase over an SQL one, mainly due to performance, as my database will be accessed and updated often. However, I must monitor my usage limits and ensure that my app does not use too much data.

**Nearby Connections API**

The nearby connections API is an API for Android that allows nearby devices to connect to other nearby devices regardless of internet connection by allowing devices to perform NDP and establish direct offline wireless links to exchange data [5].  
This can be used in my app by allowing devices with outdated data to be able to exchange data with a nearby device that has the updated data, allowing it to receive and send data, allowing phones that are out of connection to be able to still communicate with marshals in an emergency and still receive GPS data for player location [6].

**Maps**

In my app, the Map that tracks and shows other players' locations is vital to the project's functionality. From research, there are a few different libraries that could be used for the project; these are the Google Maps API and the Map box API; both of these APIs offer similar services; another option to consider is just directly using Open Street Maps through a library like OSM droid.  
Map Box is an alternate mapping solution to larger providers such as Google Maps. It provides many customization options and has a slick modern design to its maps; it uses the data from OpenStreetMaps and has a good SDK and API and a generous free tier.  
The first advantage of Map Box is that it offers more customization than Google. It can edit every map layer, which could help add different map styles for accessibility and personal preference in the app. Also, Map Box is cheaper to use, with it only charging once there are 25000+ users and 28,000 web loads per month; Google, however, is $7 per 1,000 loads, which is expensive. [7]  
Google Maps API Is an API offering from Google that allows developers to integrate Google Maps into their products; it is a well-built piece of software with plenty of data and easy integration with Android native apps through its API; it has extensive route-finding technology, being one of the most convenient ways to find and get to a place.[8]  
The advantage of Google Maps is that it has much more map data and features, with satellite images and street view being some of the services that Map lacks. However, this is irrelevant to the project, though having satellite images would be nice.  
Open Street Maps is an open-source mapping project that uses volunteers to map out the world; it is entirely customizable due to its open-source nature and offers itself as an alternative to other more authoritative mapping services[9]  
The advantages of Open Street Map over the rest of the options are, firstly, it is entirely free to use and will not have any overhead costs. It is also highly customizable, with many layers of styles to choose from when making the app; one of the weaknesses of directly using OpenStreetMap is that the OSM droid library, which now is no longer being updated any more, was last updated on August 19, 2024.  
From the research, Open Street Maps will be used as it is a cheaper, more customizable SDK that offers a good library (OSM Droid) of functions for Android; even though it is no longer updated, it has not gotten old enough yet for me to not want to use it.

## 2.4. Other Research

**Cross platform considerations**

For this app, there are two platforms I was considering developing on, either iOS or Android. Both have an almost equal market share in Ireland, and creating an app for both platforms would be too much work for the project's scope, so a platform had to be chosen from research, Android became the obvious choice. To develop in iOS, I need to use a VM running macOS or a Mac, which is not ideal for development. Additionally, I have a lot more experience with Android development, so Android was chosen.[10]

**GPS**

GPS, or global positioning system, is one of the major technologies that has enabled many of the phones to be helpful. GPS works by having satellites in the GNSS network continuously broadcasting their location. Once the phone has the location of four of these satellites, it can locate its position anywhere in the world, to a degree of accuracy, typically 5-16 feet, influenced by the phone's surroundings; generally, the more of the visible sky, the better.[11]  
GPS will be a cornerstone of the project. A good amount of the app's functionality will be based around it; one challenge I will need to overcome regarding GPS is the accuracy issue since Airsoft often happens in forests or inside buildings, and accuracy will be affected.

**Geofencing**

Geofencing is a technology that allows GPS-enabled devices to do something when they enter or leave a geo fenced area. These areas can be as large as they want and can be any polygonal shape. Due to GPS not being entirely accurate, it can give many false positives or negatives when near the edges of a geo fenced area.[12]  
Geofencing is used for many things, and it is commonly used in marketing, giving location-based adverts to better appeal to a user and get more value out of their advertisement. [13].  
In the project, It is necessary to implement areas that are "out of bounds" and not in play for either gameplay or safety reasons. Most of the time, newer players will go into these areas due to a lack of experience on the site, so from research, geofencing would be the way to do this.

**GDPR Considerations**

GDPR is a set of rules and regulations implemented by the EU in 2018 that greatly impacted how data was handled worldwide; it is hugely important for the project since the app gathers sensitive information and must abide by these rules.

The data protection principles can be broken up into 7 points in broad strokes.

1. Lawfulness, fairness, and transparency — Processing must be lawful, fair, and transparent to the data subject.
2. Purpose limitation — You must process data for the legitimate purposes specified explicitly to the data subject when you collected it.
3. Data minimization — You should collect and process only as much data as necessary for specified purposes.
4. Accuracy — You must keep personal data accurate and up to date.
5. Storage limitation — You may only store personally identifying data for as long as necessary for the specified purpose.
6. Integrity and confidentiality — Processing must be done in such a way as to ensure appropriate security, integrity, and confidentiality (e.g., by using encryption).
7. Accountability — The data controller is responsible for demonstrating GDPR compliance with these principles.

[14]

The main concerns with the app are data minimization and integrity. The reason is that the app will collect sensitive information like location, and storing identifiable personal data in case of a data breach is a bad idea. Secondly, having good data security is just a good best practice, so sticking to this as a principle will be beneficial. The rest of the principles will also be kept in mind as well.

**Battery Life**

Quartermaster will be using GPS as a core feature of its design. However, GPS and location-based services do have some drawbacks. One of the major ones, especially for this project, is the increased power drain that will limit battery life on the phone that the application is running on; this is a problem as most Airsoft games can last anywhere from a few hours to multiple days, so battery life must be considered.  
In indoor areas, battery consumption from GPS can be increased by as much as 75% from the phone using more battery to find a better signal strength[15]. One way to keep battery life costs down is to use less accurate location-finding methods, such as coarse network location, draining only a tenth of what using fine location may take using GPS[16].  
Hopefully, this will help offset issues of using this app in areas with bad signals; it might be more inaccurate, a balance will have to be struck between accuracy in tracking and a workable battery life.

**Requirements Gathering**

Requirements gathering is an important stage in the development of any application. Requirements gathering consists of creating requirements by getting user feedback and wants for the potential application.  
  
There are two different types of users this app intends to assist:  
Marshals are the people who create and run the games; their job is to enforce rules and look after the safety of the players.  
Players: the people who participate in and play the game.

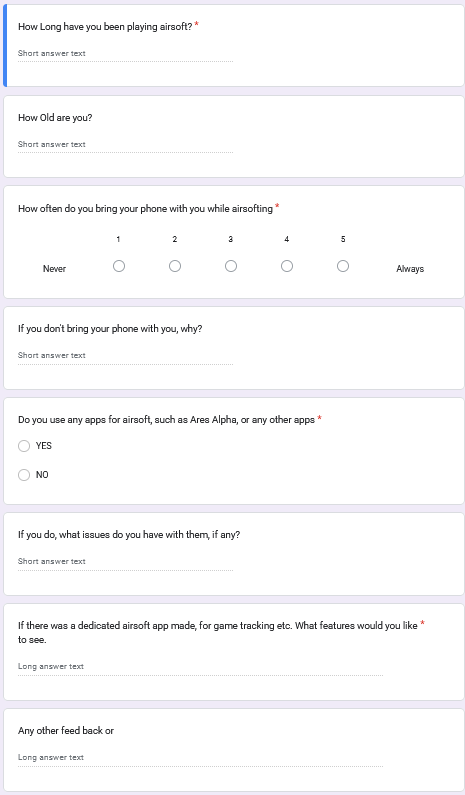
**Interviews**

To get some requirements for the marshals, I drew on my experience from working as one and interviewed fellow marshals to get a good idea of what was needed.  
I asked them questions along the lines of:  
“What would this app need for you to use it?”  
“What problem do you encounter on the field?”  
“What features would you like to see?”  
“What do you think should be avoided?”  
These questions were chosen to get a simple list of requirements and actionable features.  
Moreover, these responses were used to construct some requirements:  
Must-Have: SOS feature in case someone is injured   
Should Haves: Safe zone markers, out-of-bounds areas  
Could Haves: Emergency contact information(phone numbers)  
Avoid Complexity and barriers to entry for players

**Surveys**

To get some requirements for the players, i drew on my own experience of being a player, as well as creating a survey and using the responses to help with creating my requirements.

These questions were chosen as to ascertain whether or not that the app would be viable, via the phone question. And to see if there are any features that players want in this application.

Figure 1: survey

**Table 1 - Responses:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| q1 | q2 | q3 | q4 | q5 | q6 | q7 |
| 6 | 15 | 1 | In Case of losing it | N |  | Map of Site |
| 6 | 51 | 5 |  | N | Most apps are bad | Team Mate tracking |
| 13 | 25 | 5 |  | N |  | Sign in numbers |
| 25 | 56 | 1 | Losing it / it gets hit | N |  | Real time updates |
| 14 | 60 | 1 | Dont want contact | N |  | comradely |
| 10 | 58 | 5 |  | N |  | Topics on games terrain etc, a league system |

(Full chart available in Airsoft App Research (Responses).ods)

**Response Discussion**

from the responses received (in table above) we can see a few things, firstly that 50% of respondents didn't usually bring their phone out when they played airsoft for a few reasons Firstly the main reason is is that they didn't want their phone getting damaged secondly main reason is that don't want contact on the field.  
  
Another point noted is that most of the respondents do not have any experience with any other airsoft app either Ares alpha or any other application, this is due to as one respondent said “most of them are bad”, so Quartermaster has to deliver on a better experience than the other apps.

Finally the most requested features from the survey all have to do with information, either to do with teammates location or map information about the site and games.

## 2.5. Existing Final Year Projects

**Introduction**

In this part of the project we will go through a two previous final year projects and will examine their strengths and their weaknesses and how they may apply to this project.

**Anseo!**

**Author: Jonathan Hew**

Anseo is a web app that is designed to be used by lecturers to keep attendance to their lectures in a more convenient way than a paper sign-in sheet; it does this by creating a session and allowing students to join it only if they are physically present within a geo fenced area.  
Jonathan used a three-tier model for the project with a presentation layer, application layer, and data layer; the user opens the web app, which uses react for its function, and then connects using node.js to a PostgreSQL server to store data.  
They also used the Agile development method of feature-driven development for their project; their reason was because of the numerous features of the project; this is similar to the project and the many features I will also have to make, so they might be a good example of what to follow.  
Anseo! It uses technology similar to the app, such as geo-fencing and GPS technology. At the same time, one user hosts a central session that all other users connect to, so some lessons should be taken from this project due to its similarities.

**Nitelite**

**Author: Sean Breen**

Nitelite is a native mobile app designed to help people stay safe in nightlife environments; it intends to do this by creating an app that allows people to keep track of their friends in a busy nightlife environment and, if needed, request urgent help from friends in a dangerous situation.  
They also had a 3-tier system, with a React native front end, a Django back end, and a SQL database.  
They also used an agile development methodology, though they did state that it would be hard to use in a single-person project due to the nature of agile. Sticking to the six key principles would be ideal to ensure future best practices.  
Nitelite interested me because it used Bluetooth beacons to get accurate positions inside a building, which could be helpful for my app.

**Conclusions**

From these projects I have examined above, I have learned some important lessons; firstly, the agile development method is quite popular, and for good reason: it leads to better outcomes for the project overall. Secondly, both projects gave me a good idea of what a finished project looks like and my requirements to achieve a similar result to the above projects.

## 2.6. Conclusions

The conclusion of this literature review conducted for my final year project is, firstly, the research into different tech away from the stuff I would usually use led to some fascinating and beneficial insight into some of the tech I am going to use for this app and made me change my mind on confident choices I had already made up in my head about the development of this app.  
Secondly, there are some existing solutions from my examination of previous solutions. However, both of those examined have flaws I also need to overcome and strengths I should try to put into my project, and it proved that what I wanted to do was feasible.  
Thirdly, reading how previous students approached similar problems showed me some examples of what was expected in terms of complexity from this project and ideas on how to move forward with it.

**Requirements**

From what I have learned, I have built a set of requirements using the ideas gathered from above and using some requirements gathering from the expected user base using interviews and a survey.

See Requirements in table 2 below.

**Table 2 - Requirements**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Req ID | Requirement Desc | Category | Priority | Source | Criteria | Notes |
| UR-001 | Realtime Tracking of players and teammates | Functional | High | Survey | Tracking works well | Completed in Prototype |
| UR-002 | Stats Tracking of players | Functional | Medium | Survey | Be able to track your win loss and other stats |  |
| UR-003 | Accurate Map of site | Functional | High | Survey | Have the ability to create an accurate map of the site |  |
| UR-004 | Reliability of functionality | Non-functional | High | Interview | Have all the functions work reliably | This is a common issue with other apps |
| UR-004.1 | Tracking Accuracy | Non-functional | High | Interview | Have tracking be accurate to 5-10 feet |  |
| UR-004.2 | Session Reliability | Non-functional | High | Interview | Have session be easy to join and not close randomly |  |
| UR-005 | Safety SOS features | Functional | High | Interview | Have a way for players to contact marshals in case of an emergency |  |
| UR-006 | Be able to tell how many people are at the site/ are booked in remotely | Functional | Low | Survey | Have the ability to see different sites and how many people are planning to be there | Would require a lot of work and a whole site system, do if time permits but otherwise ignore |

# 3. System Design

## 3.1. Introduction

System design is a vital part of the development process, as it sets guidelines and the design for the project in place to ensure that the project is done to a good standard and in a reasonable time frame. In this chapter, we will review some software methodologies going through how they work and then choosing an appropriate methodology to use for this project and the reasons why.

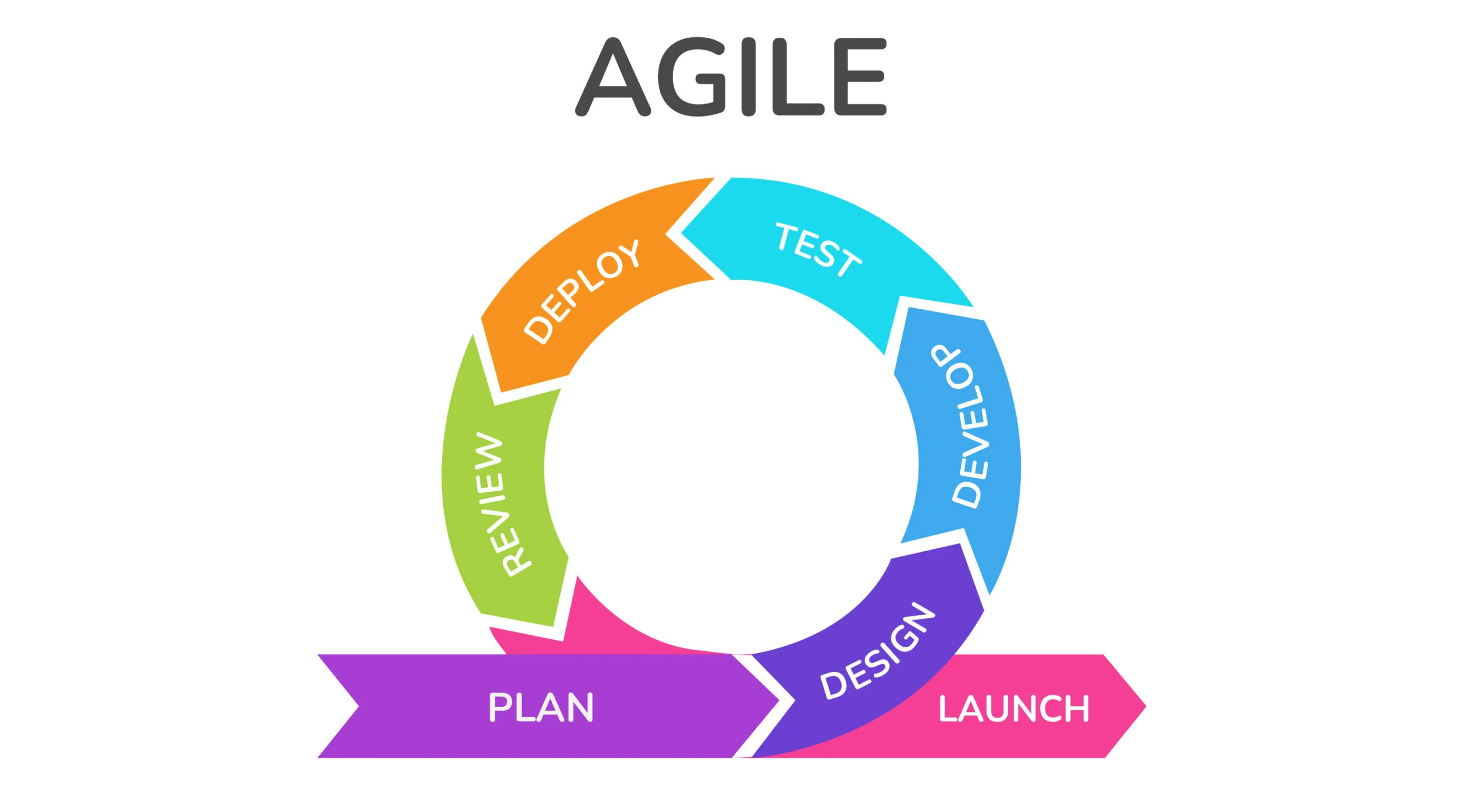
Then we will go through the system overview, examining each part of the designed system and why it will be that way, this incudes going through our use cases made from the requirements, our Database Design and finally our UI design any why it is that way.

## 3.2. Software Methodology

In my research I found 2 software methodologies that I thought where suitable for this project

**Agile development methodology**

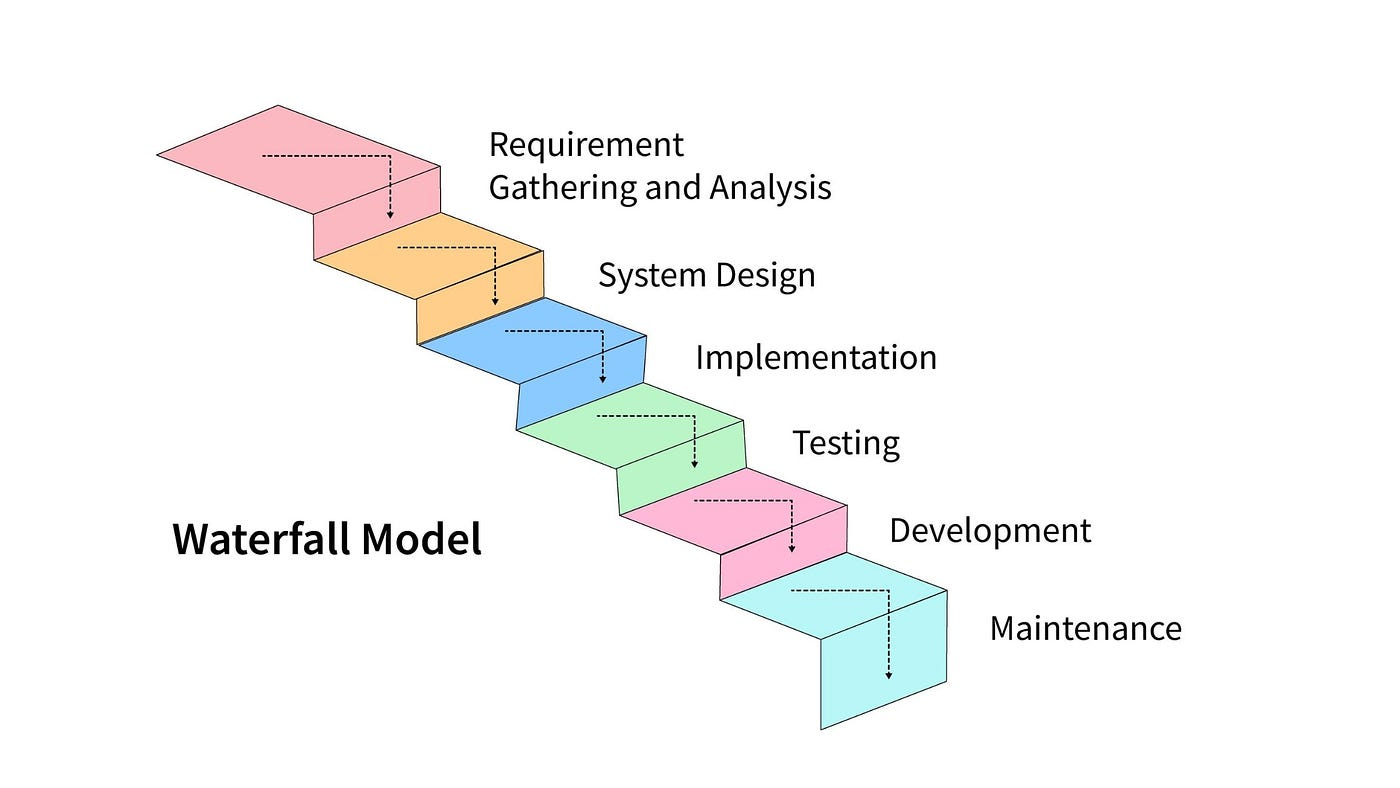
The agile methodology is a methodology framework that is based on breaking down the project into multiple stages and iterating on the development cycle as shown above, and it is designed to be reactive and be able to change based on changing requirements and take in shareholder or project owner feedback and incorporate it into the project with minimal issues, there are a few ways to implement an agile approach, examples include scrum and kanban[17][18]

Figure 2: example of a agile development cycle

This approach appeals to me due to my circumstances around this project, i am a solo developer for this so a lot of the collaborative benefits of agile will be lost however since i work at an Airsoft site it puts me in a good position to receive feedback on each iteration of the project.

**Waterfall development methodology**

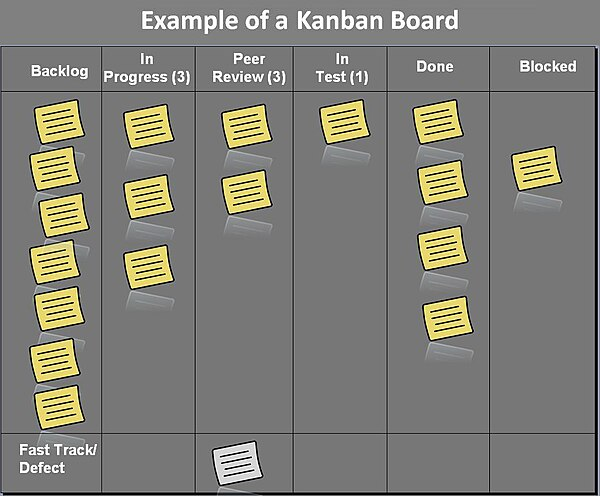
The waterfall development methodology is a linear sequential life cycle model, where planning, designing, development, and launching are handled in sequential order; due to this highly structured approach, the waterfall methodology is highly inflexible and does not react well to changing requirements. In the figure below you can see how it is a sequential series of steps with no going back later and changing previous design.

Figure 3: an example of a waterfall development cycle

However, since this project is a solo-created project, its highly structured nature, and easy-to-use nature might be beneficial to the project, and its focus on getting the project done rather than re-iterating over and over could help get the project done quicker[19]

**Kanban Methodology**

The Kanban method is a framework for implementing agile software development by visualizing all work to be done and limiting the work in progress. It does this by breaking down the work to be done into individual items and then organizing them on a board and sorting them by the categories “to-do,” “Progress,” “in Review,” “Finished,” and “Blocked” This is all done to minimize work wasted and ensure that projects are finished on time, instead than using fixed sprints a more continuous flow to development with each item being worked on to completion then moved onto the next stage. Below you can see an example of a kanban board.

Figure 4: a example of a kanban board

**The Chosen Method: Agile Kanban**

In the end i decided that using an agile Kanban method would be good for this project, i feel that with the ability to be able to regularly test my app on a weekly basis that an agile method would produce good results, as i will be able get feedback on each sprint and actually use that feedback to improve the project.

The main reason i am going to be using the kanban method is because it is very good at visualisation, with each task being represented on a physical board as a card, it will help me keep track of what work i need to do and what work i have already done. Additionally a way to visualise the progress i will be making will help me stay motivated and productive for this project.

## 3.3. Overview of System

**Overview**

Quartermaster will follow a simple 2 tier architecture with the front end native application communicating directly with the back end server, the reason this architecture has been chosen is due to its simple and quick to make nature, additionally the complexity of an N tier architecture is not required for the scope of this project

**Front End**

The Front end of the Quartermaster application will be an android native app that uses Kotlin, a new object oriented programming language, designed to be interoperable with Java and Java Libraries., The applications UI will be created using Jetpack Compose, which is the recommended toolkit for making UI in native android apps as it is significantly easier to use and allows for far more interactivity within the UI.

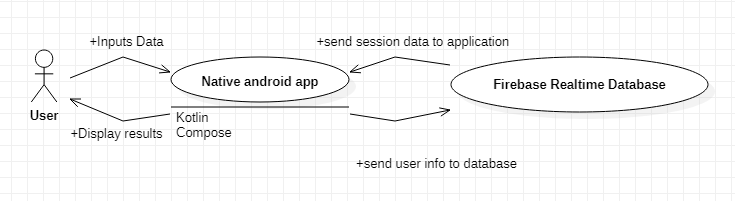
OSMdroid will be used to create any maps screens with the OpenStreetMaps map set, and will include an editor allowing people to create their own maps for the application.

**Back end**

The Back end of the application will use Firebase to host the data for the application on a real-time database, Firebase was chosen as it has an affordable pricing model, and it is designed to be used with mobile applications similar to this project.

**Real-time database**

Real-time database is going to be the data storage solution used. it is a NoSql Database and it stores its data in JSON format. Its being used as it is able to be updated and changed very quickly, which is essential for this projects functionality. in addition its integration with other google cloud tools like cloud functions allows for additional functionality, In also since we are using Firebase, all the hosting and setup is handled by Firebase itself, saving time.

Figure 5: basic diagram of planned architecture

**Database Design**

for my database i need to have a few different linked objects such a users and sessions and sites and games etc, in order to do this in a planned fashion a database diagram was created as a visual representation of what the final database will look like.

**Sessions JSON**

Sessions: {

session:{

Session\_ID: 123456 – this is also the join code

Game\_ID: 123 – links to a game

Site\_ID: 123 – links to a site

Users:{

user\_123:{

User\_ID: 123

role: admin – what role they will be

status: ok – their status in the game

team: red – their team

username: Kozi – user inputed name

notification\_token: token – token to send notifactions to the users phone

location:{ - stores last known location.

longitude: 53

latitude: -6

}

}

}

more users...

}

}

}

**Sites JSON**

sites:{

site:{

name: test

site\_id: 123 – site id for access from session

users:[ - users that have access to the site

“user\_123”

]

brief: safety rules etc

games:{

game:{

name: tdm

game\_id: 123 – game id for access from session

brief: game objective and rules

markers:[ - contains info about each marker

colour: Red

name: marker

icon: flag

team: red

points:[

{

longitude : 60

latitude : -1

}

],

more markers...

]

}

}

markers:[

same as markers in games object

]

}

more sites...

}

Purpose of each part:

**Sessions:**

The sessions JSON contains all of the data related to each session, each session has a 6 digit ID which is both its unique identifier and also the join code to each session. Inside each session is a Users object which contains all the users that are in the session, as well as a game ID and a site ID which tells the users in the session what game and site to load from the sites JSON.

Each user will have their unique ID, along with their role, team, username, current status and current location in the user object, these will be used to display the user correctly on the map. The status will be used to indicate if the player is in danger so it can be marked on the map.

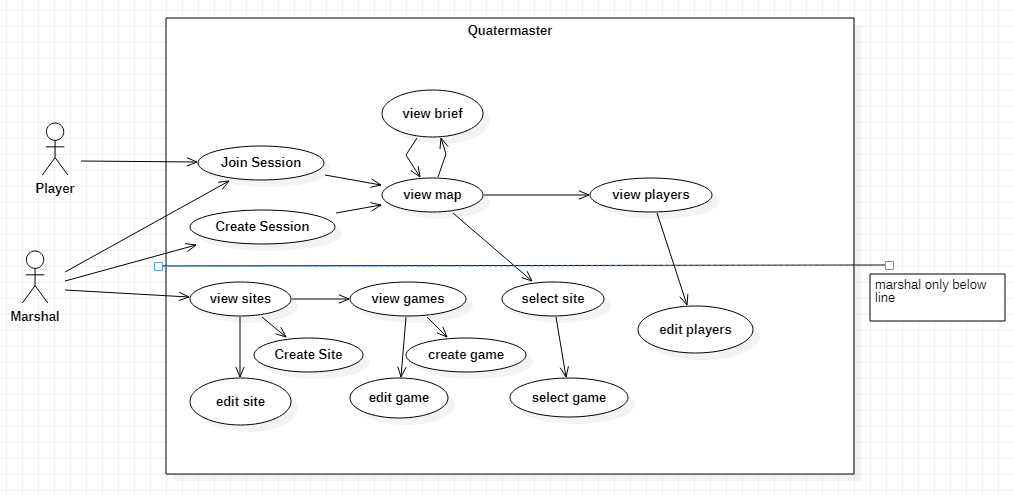
**Sites:**

The Sites JSON contains all of the data related to each site and the games within that site, each site will have a brief for information like safety rules and rules that stay the same between games and a list of markers for objects that will never move in between games, while each game will have the markers specific for each game and the rules for just that game, this is to reduce the amount of time that marshals spend creating map layouts so they can reuse the site map each time.

Each site and game will be loaded using a unique id each will have that will be saved to the session when the game/site is selected by the marshal.

**Use Cases**

In the app, there are two different types of user players and marshals; players only need the app to be able to join sessions, and marshals act as admins. They create the sessions and the games and manage the players, and they also need to be able to join other marshals' sessions, as there can be more than one marshal out there at once. The diagram below (figure 6) depicts a high-level diagram of the app's use case and, following that, a more in-depth explanation of each.

Figure 6: Use case

**Use Case Diagrams**

In this part I will elaborate on the use cases set out previously

|  |  |  |
| --- | --- | --- |
| **UC-01** | **Create Session** | |
| **Dependencies** |  | |
| **Description** | User selects create session | |
| **Preconditions** | The user has opened the app and isn’t already in a session | |
| **Ordinary Sequence** | **Step** | **Action** |
| **1** | User selects the Create session button |
| **2** | System connects to the Firebase server |
| **3** | System creates a new session with a unique ID |
| **4** | System creates a new user in that session and moves the user to the Map screen |
| **Postcondition** | The user is in a session and can have other users join it | |
| **Exceptions** | **Step** | **Action** |
| **2** | if the system cannot connect, an exception will be thrown |
| **Comments** |  | |

|  |  |  |
| --- | --- | --- |
| **UC-02** | **Join Session** | |
| **Dependencies** |  | |
| **Description** | User joins a session | |
| **Preconditions** | The User has opened the app and isn’t already in a session, a session exists | |
| **Ordinary Sequence** | **Step** | **Action** |
| **1** | User inputs a session code |
| **2** | User presses the join session button |
| **3** | System connects to the Firebase |
| **4** | System checks to see if there is a session that exists with that code |
| **5** | If the session exists it adds them to it and puts them in the map screen |
| **Postcondition** | The user is in a session | |
| **Exceptions** | **Step** | **Action** |
| **3** | Cannot connect, throw an error |
| **5** | Session doesn’t exist throw an error |
| **Comments** |  | |

|  |  |  |
| --- | --- | --- |
| **UC-03** | **Create Game mode** | |
| **Dependencies** | **UC-05** | |
| **Description** | User Creates a game mode. | |
| **Preconditions** | The user has opened the app and has navigated to the site they want to create the game mode in | |
| **Ordinary Sequence** | **Step** | **Action** |
| **1** | User Presses Create Gamemode Button |
| **2** | System opens dialogue that requests input for name |
| **3** | User inputs a name |
| **4** | System connects to firebase and creates a new game in the current site |
| **Postcondition** | Gamemode is created and user can now edit it | |
| **Exceptions** | **Step** | **Action** |
| **4** | System cannot connect, throw error |
| **Comments** |  | |

|  |  |  |
| --- | --- | --- |
| **UC-04** | **Call SOS** | |
| **Dependencies** | **UC-01/UC-02** | |
| **Description** | Selects to send a SOS signal. | |
| **Preconditions** | User is in a session | |
| **Ordinary Sequence** | **Step** | **Action** |
| **1** | User needs help, either emergency or not and presses the SOS button |
| **2** | System connects to firebase and updates their status to SOS |
| **3** | FCM sends out notification to all admins |
| **4** | Users location is marked on all admins maps |
| **Postcondition** | User is marked on map and admin have notification informing them | |
| **Exceptions** | **Step** | **Action** |
| **2** | System cannot connect to firebase throw error |
| **Comments** |  | |

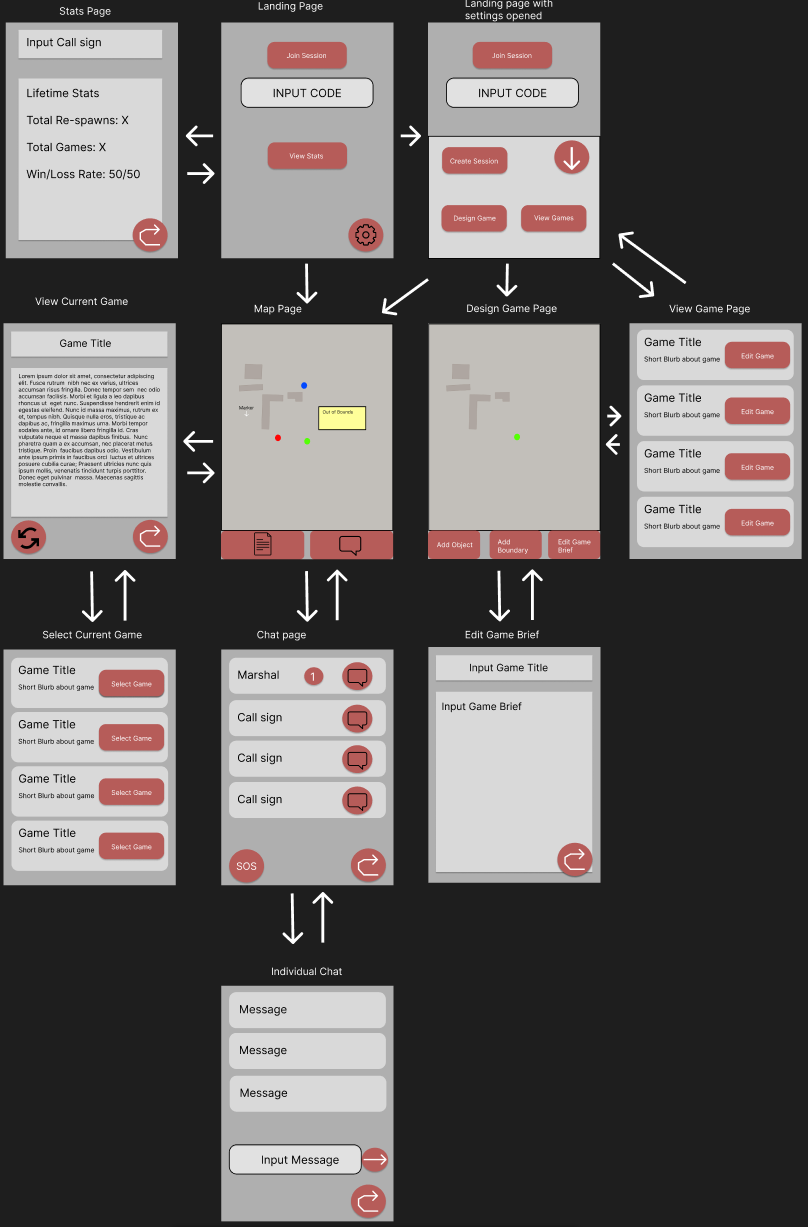
|  |  |  |
| --- | --- | --- |
| **UC-05** | **Create Site** | |
| **Dependencies** |  | |
| **Description** | Marshal Creates Site | |
| **Preconditions** | User has navigated to site list page | |
| **Ordinary Sequence** | **Step** | **Action** |
| **1** | User presses the create site button |
| **2** | System opens dialogue that requests input for site name |
| **3** | User inputs a valid site name |
| **4** | System connects to firebase and creates a new site |
| **Postcondition** | Site is created and is able to be accessed | |
| **Exceptions** | **Step** | **Action** |
| **3** | User inputs invalid site name, refuse input and try again |
| **4** | System cannot connect, throw error |
| **Comments** |  | |

**UI Design**

One of the most important aspects of the app is UI, since Quartermaster is going to be used in high stress situations, its important that the UI is well designed and is easy to use

**Prototype UI design**

During the prototyping stage of development, a few designs were tried out for the UI, below in figure 5 you can see the first mock-up of the UI design, however this was not implemented for the prototype in time, and it was scrapped after some features were dropped and others added.

Figure 7: UI Mockup of Pages in App

**Final UI design**

Figure 8: phone holder

in order for quartermaster to reach its full potential a good simple to UI was necessary, and the previous mock-up prototype UI was too cluttered and complex to fulfil that vision so when going into the developing the full application, more UI design and testing was done in-order to land on an optimal solution, in this mock-up, the UI was simplified and labels where added to each of the navigation options, a lighter grey colour was chosen as it is easier on the eyes in most light conditions. (See figure 9)

In this design more simplicity was emphasised in order to make the design easier to navigate, having icons for the buttons from the materials 3 library to ensure consistency, also a landscape view will also have to be made due to the fact that alot of users use chest mounted phone holders (see Figure 8)while playing airsoft so a landscape is a must, however there where still issues.

Figure 9: UI mockup for Map Page

In order to evaluate this design a Heuristic Evaluation was carried out []

**Heuristic Evaluation**

**Visibility of System Status:**

User is told what tab they are in via the toolbar, however contrast on selected button is low

Solution: Increase Contrast for active button

**Match Between System and the Real world:**

Map symbols in the current design are not consistent and need improvement

Solution: Design a new set of symbols, base them of common video game or military symbology.

**User Control and Freedom:**

users can freely switch between info and map, however the only way for the user to leave the session is via closing the app.

**Consistency and Standards:**

Uses material 3 icons as standard design to keep similarities with other applications

**Error Prevention:**

no warning when pressing the SoS button, may cause accidental presses

Solution: add in a dialogue to make sure user is sure.

**Recognition rather that recall:**

no issue that can be found.

**Flexibility and efficiency of use:**

No customisation option available for individual users

Solution: add in a way for users to customise their interface.

**Aesthetic and minimalist Design:**

No issue, design is minimalist

**Help users recognize diagnose and recover from errors:**

Not enough feedback when error occur such as lost connections

Solution: add in error message

**Help and documentation:**

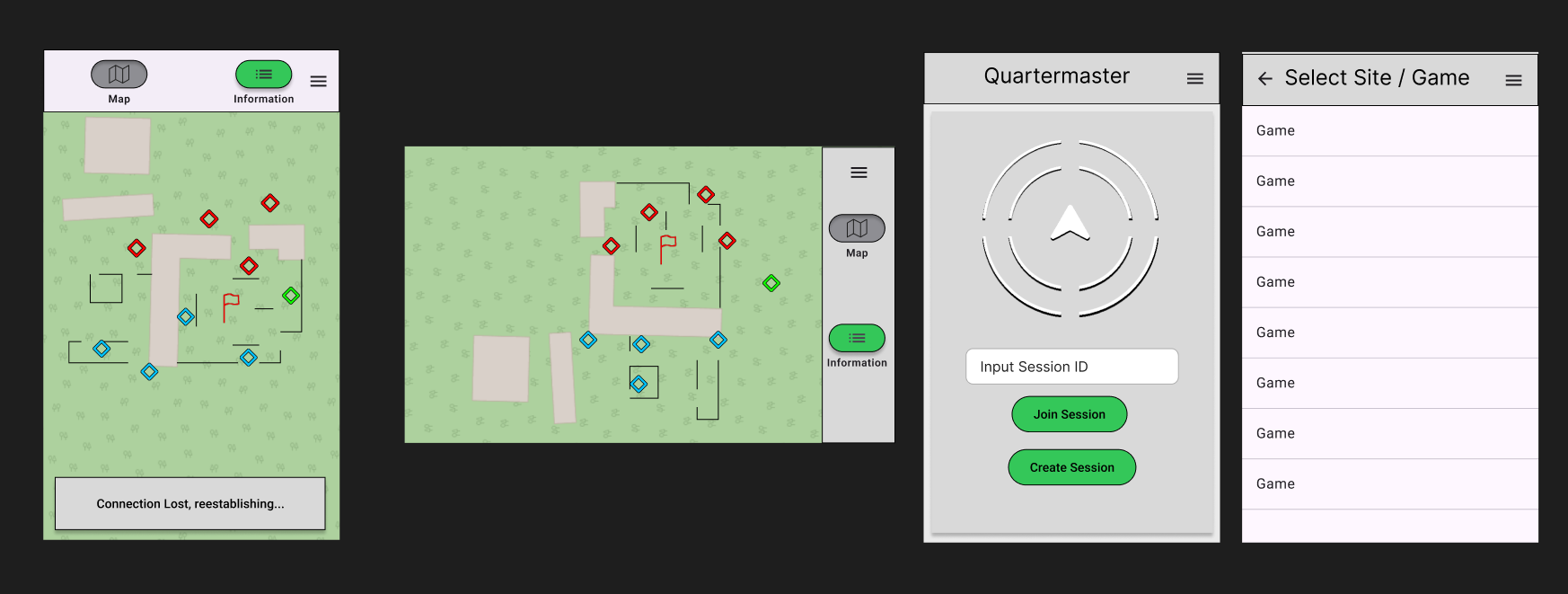
No Available documentation on how to use application at all.

Solution: add a help page in that allows new users to learn a basic understanding of the application.

As we can see from the evaluation that there are plenty of issues that need to be solved in this current design, therefore the original design was iterated on to create this:

**Second Design**

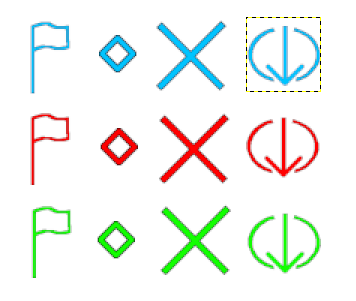
in the second design (shown above in figure 10) some improvements where made, firstly intractable buttons where given a green colour which contrasts nicely with the grey background and allows user to identify what to interact with better, secondly the main join/create session page had a lot of white space removed and a logo added to increase visual clarity, in addition feed back was added to a lot of the functions so if something stopped working the user would be informed.

Figure 10: second design

Outlines where also added to a lot of elements as they help improve visual clarity and accessibility[]

**Map Icons**

It was decided that map icons and user icons must be as easy to spot and identity as possible, when designing the icons the NATO joint military Symbology was used as a base, due to the fact its designed to be easy to identify on a map, and supports the Nielson's principle of matching the system with the real world as these symbols are designed to be similar to real ones used in actual military mapping and ones used in video games, which should be familiar to an average airsoft player. See figure 10 for the icons made.

Figure 11: icon set created

## 3.5. Other Sections

**Balancing battery life with location updates**

Due to Quartermasters intended design is that the app is to be used throughout the course of a game-day (4-24 hours) battery life is very important. However a lot of the technologies I use in this app tend to consume a lot of battery, especially since i want the location tracking and geofencing to be running while the app is in the background or the app is closed.

One way i will try to be managing this is to only have location be updated every 15 seconds to minimize the amount of battery that is used by the application, this will save battery at the cost of live information.[20]

Another way i can reduce battery consumption is via lowering the accuracy of the GPS as GPS will take up more power to get a better signal in bad connection areas, however i am loath to do this as i feel that it would reduce the usefulness of my app.

In the end i decided to have the GPS precision as high as it can at all times, with the addition of a 15 second break between each update, as this keeps precision high and also keeps battery drainage to a minimum.

## 3.6. Conclusions

From the design work covered in this chapter the possible methodologies for this project where covered and then a chosen methodology, in this case Agile Kanban was chosen due to its visualisation of progress and the flexibility of the system, secondly the design work for Quartermaster was covered, including architecture design, database design and UI design, as well as the creation of use cases which will be fulfilled during development.

# 4. Project Development

## 4.1. Introduction

In this chapter the development process of the of the project will be covered, including how the project was managed and controlled. each feature developed will be covered along with how they fulfil the various use cases and requirements, as well as covering any technologies used to develop them.

**Project Management**

In order to manage the project and ensure that any mishaps would result in minimal progress lost, git hub was used to manage the project with all code being commit there on a regular basis in case any damage or corruption accidentally happened via a power outage or something similar.

Commits would be made when there was any change to the code and at the end of each workday.

## 4.2. Software Development

**Introduction**

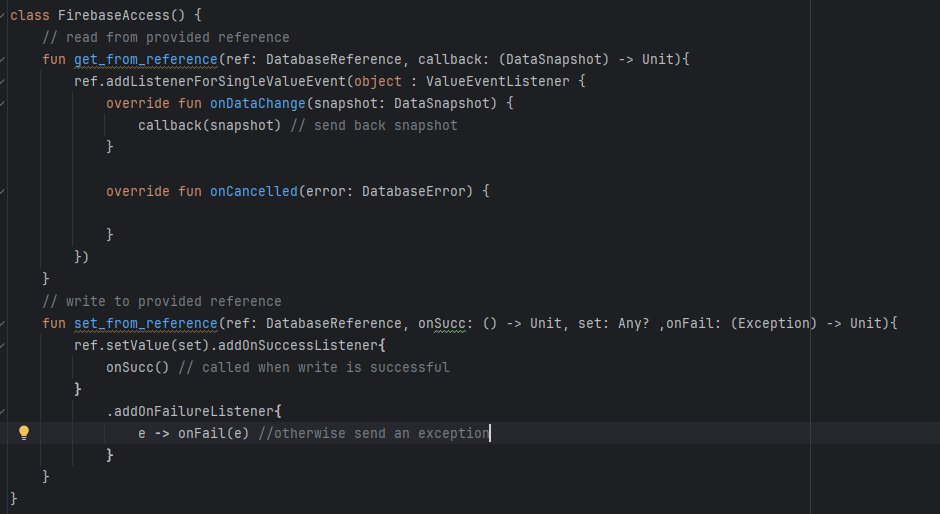
The development of Quartermaster was split up into multiple individual items, each of these items consist of a feature to be developed. Then each of these features was put on the Kanban board and worked on in order of priority as some features needed to be complete in order for others to work at all. Firstly we will discuss how the back end was setup, including setting up firebase, how firebase was accessed from the application and how that data was stored temporally on the phone. Then the rest of the features will be covered including the session system, the game map screen, the editor and the UI.

**Firebase Setup**

The first step in developing quartermaster was the first set up the Firebase real time database so that the application could communicate with it. This is necessary as nearly every other feature depends on firebase to work.

In order to set the database firstly a firebase project was created, then the application was added to the project so it could access the information inside the project, finally the Firebase Real-time Database SDK was added to the application, this gives us all the tools to read and write to the real-time database from the application.

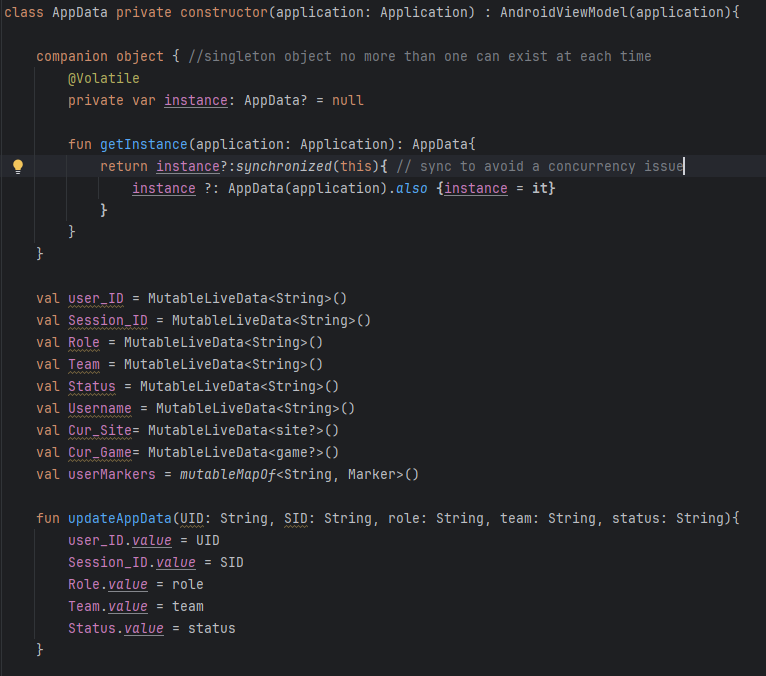
Then a class was setup to organize reading and writing behaviour for the database, so that the behaviour when accessing the database is consistent. (shown in figure 12), it accesses the database by being passed a reference to a part of it, references can either already exist and be used to overwrite old data or be new to create new data. If get\_from\_reference is called it will return a snapshot of that reference, if it exists and since we use objects to create the data on the database such as user or game, these snapshots can be mapped back to that object. This is facilitated by making sure all the objects have a zero argument constructor so they can be mapped back with snapshot.getValue.

Figure 12

**App Data**

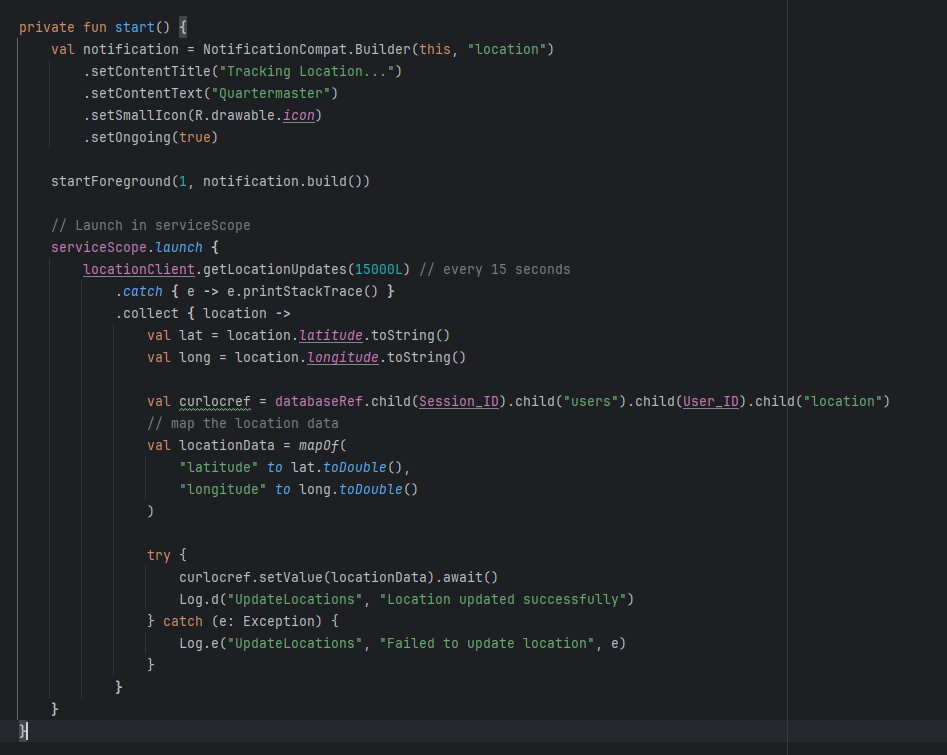
The purpose of the appdata object was to act as temporary storage for data that only needed to be stored for the applications lifecycle and not stored permanently on the device.

Early on in development it became clear that local temporary storage of some data would be needed to ensure functionality even when the App lost connection to the database, the approach to solving this was the creation of a singleton object that would contain all needed data within itself. AppData is that object, its singleton as creating more that one would lead to issues with accessing the data and storing it, this is done by using a companion object, which is an object that are called on the class itself rather than on each instance of the class, this allows the get Instance function to be implemented which either creates a new userData class if one hasn't been created yet in the application or returns the already existing one.

Figure 13: code snippet of AppData

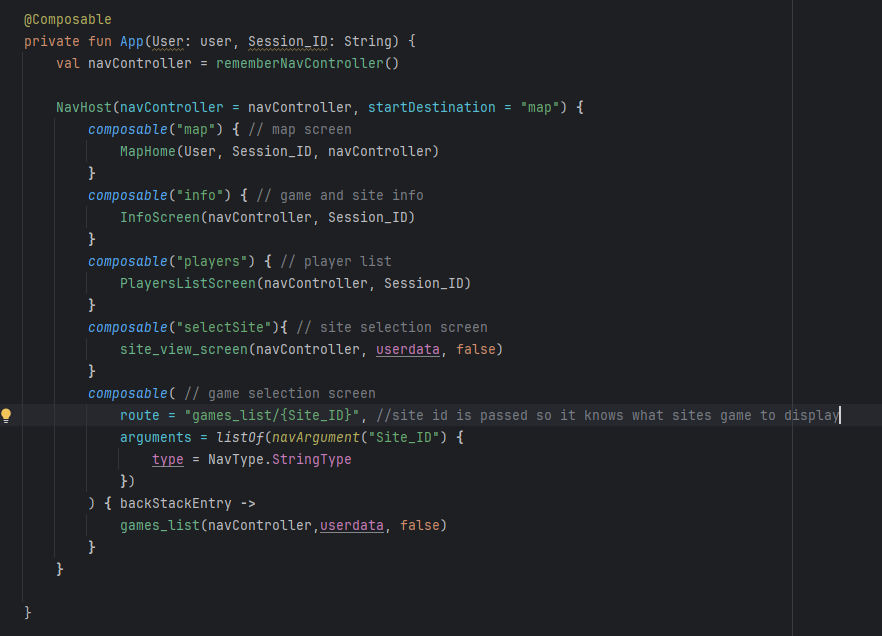
**Location Service**

The purpose of the Location Service is to track users locations when the user is using the app even if its in the background. Location service was developed to replace the old method of setting the users location in the database, the old method in the prototype had a major issue where it would stop updating once the application was in the background, so all the logic governing the updating was moved into location service and it was turned into a foreground service, so it will keep running and updating the database even if the app is in the background, to ensure that the user knows they are being tracked this is setup as a foreground service as is recommended by android developer documentation.[21] The way location service works is by creating a thread in the service scope which allows it to run in the background while the app is closed as a service. It then uses the Location Client to get the location of the device which is taken using the android location library. Finally once it has the location it sends it to the firebase after mapping the data to what location is stored like in the database.[22] Finally in order to have the service run we need to let android know that our service is a service and what type of service it is this is done through the android manifest by declaring it as a service there.

Figure 14: code snippet of location service

**Navigation**

In Android the use of multiple activities or fragments is no longer needed in most applications to display different screens, instead the use of a Nav controller and Nav host allows for far better implementation of multiple navigable screens, with each separate screen being a compose-able function which is then navigated to from any composable.

Figure 15: nav host for the main activity

**Create Session**

Create Session’s purpose is to allow the user to create a session with a join code that other users can join.

Before work could be started on anything else the session system had to be setup, this was the focus of the prototype but a few things had to be reworked, firstly querying the database was moved from doing it within the function itself and changed to using the Firebase access object, in addition in order to accommodate sending notifications later in development the messaging token was also added to the user object. Finally it was moved outside of the activity and the navigate logic is handled in the callback from the firebase access object once the data is uploaded.

Create session works by creating a user and a session object loading the needed data in from the app data object for the user and creating the session id, then this is uploaded to the database and stored under sites, the user who creates the session is by default the admin. It then moves to the map screen when complete. The notification token is created by getting the latest token off of the firebase messaging singleton object provided by firebase messaging library. This token is unique to each phone.

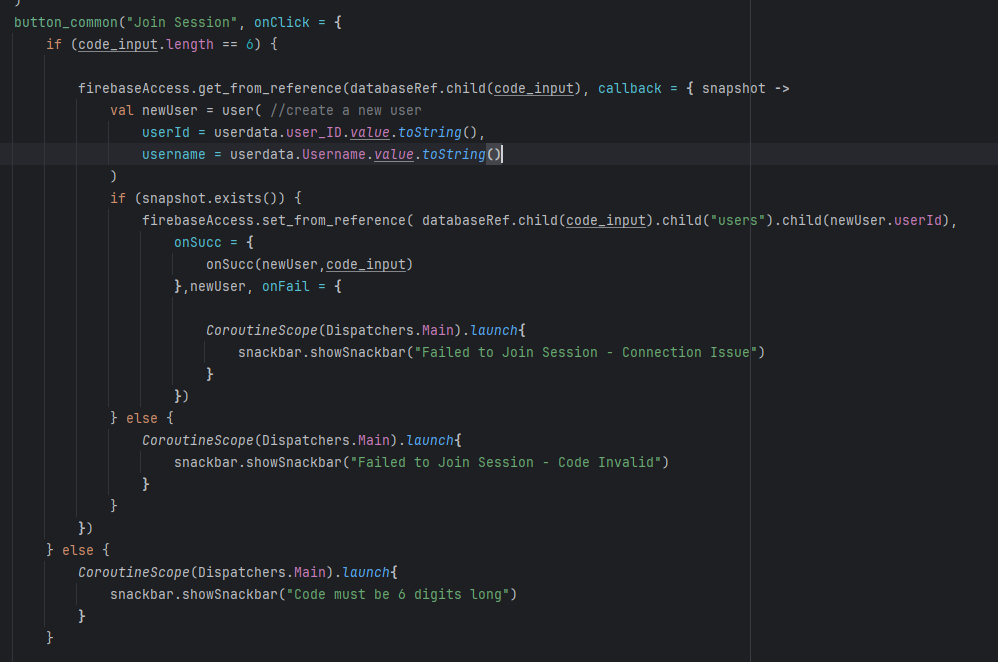
Figure 16: create session function

**Join Session**

Join Session’s purpose is to allow a user to join a session using the code created by join session.

Join Session was also reworked similarly to create session with it now using the firebase access object to handle accessing the database and being moved outside the activity with navigation to the map being handled in a call back.

Join session works by getting a snapshot of the database and seeing if the code inside the text input when the join session button was pressed matches any of the sessions IDs, if any match, it will add that user to that session by creating a new user in the users section of the session. It does this this by using the firebase access object to get a snapshot from the reference, the reference is constructed using the session code as each session is named its pin number. If the retrieval fails either due to a bad PIN number or no connection, it throws up a snackbar informing the user why its failed.

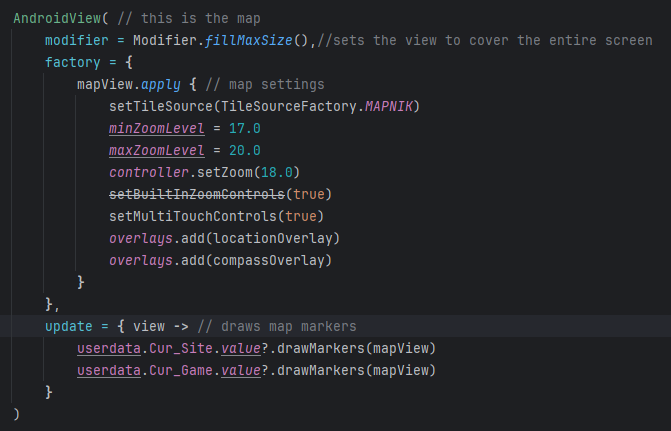
Figure 17: Snippet of code from join session

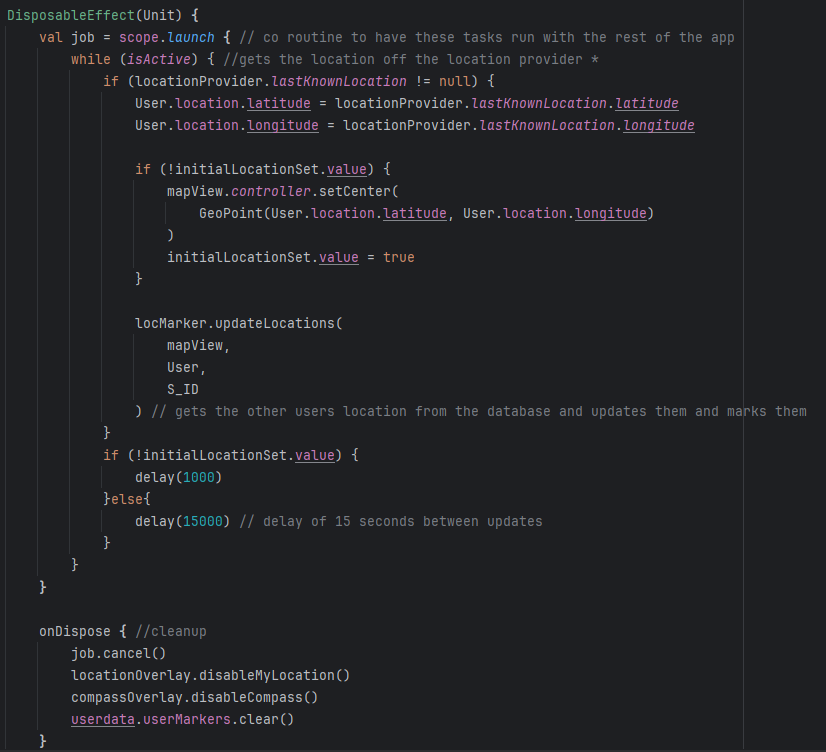
**Game Map Screen**

The game map screens purpose is to display the users current location and any markers other users nearby in order to do this a OSMdroid map view was used, in order to do this a few things had to be setup firstly the OSMdroid library had to be installed for this to work then the user agent had to be configured other wise it might result in request from my app to the Osmdroid map being blocked.

Then once this was all setup the OSMdroid map view could be created, in order to display markers on the map overlays had to be added to the map view, these overlays are how OSMdroid draws additional information on top of the map such as the current user location, markers and the compass.

In addition the map was setup so every time it was loaded it will zoom to the current location of the user, this was done by waiting until the location provider got the latest location and if the location hadn't been set yet zooming over to that point on the world map. In order to display player markers continuously a thread was setup to launch when the map view is created, this co routine will then continuously update the location markers of other players, with a 15 second delay between each update.

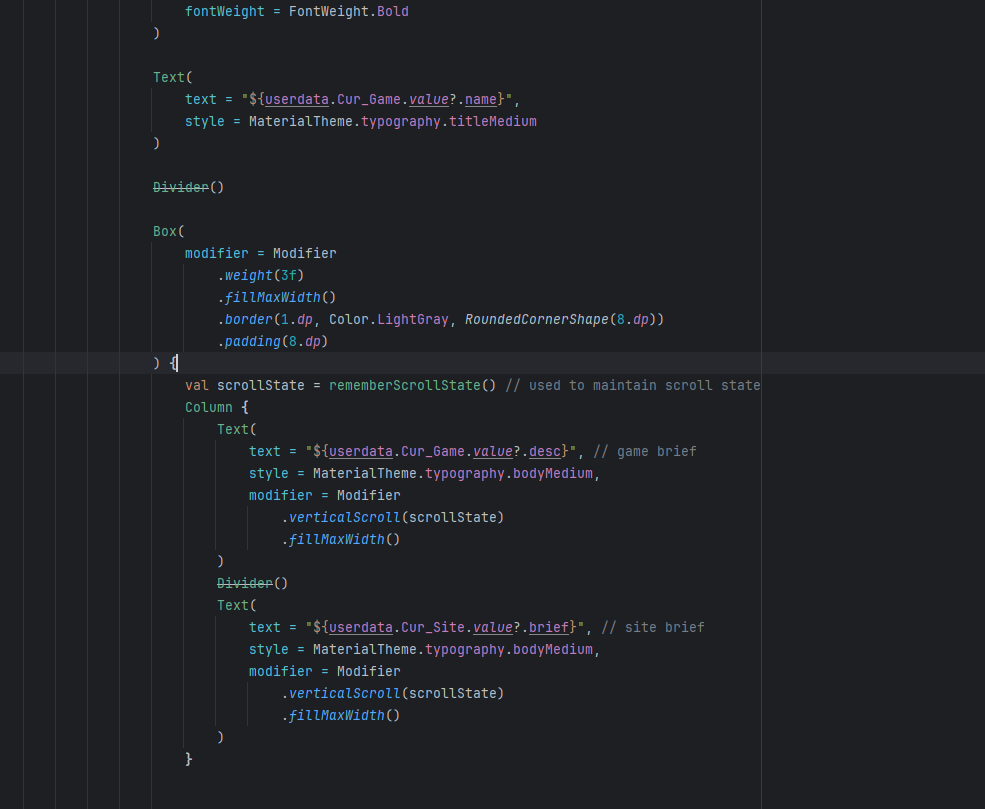


Figure 18: co routine and android view

**Game Info Screen**

The purpose of the game info screen is to display all of the text based info for the currently loaded site and game, such as the sites safety brief and the games current objective explained.

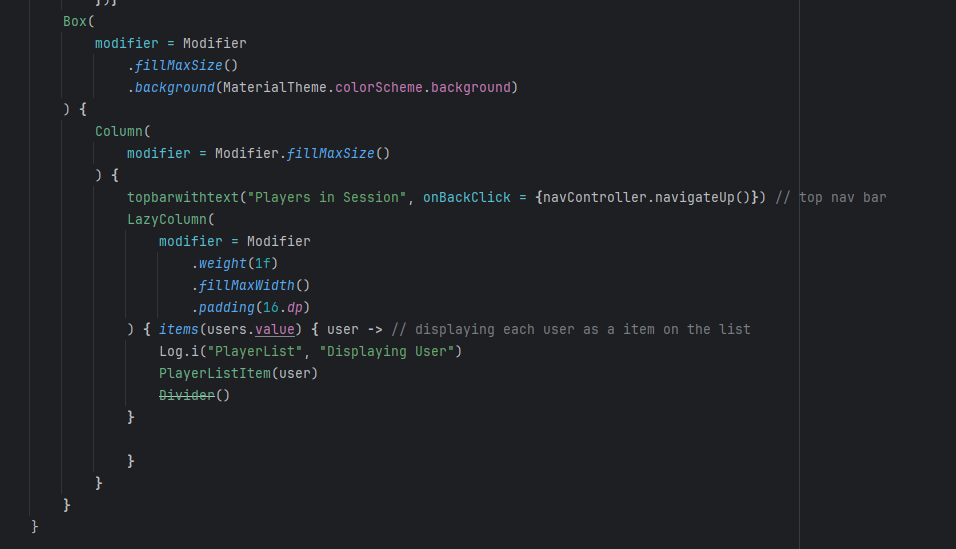
This works by using the App Data and getting the local stored game and site, before it gets this however it uses the App Data's update app data function to get updated data from the database to make sure that the latest info is available, it then displays the data in two separate scrollable text boxes, it does this by having the text boxes have the vertical scroll modifier. Scrollstate is used to control the current scroll of the text boxes and is persited through redraws of the compose by using remember scroll state. Remember is a compose feature that allows values to persist through states changes and redraws of the compose.

Figure 19: game info screen code snippet

**Player Management**

The Purpose of the Player management system is to allow marshals to manage the players within the session, such as changing their teams, giving players admin privileges and resetting their status after an emergency has been resolved, it also gives the players a list of who is in the game.

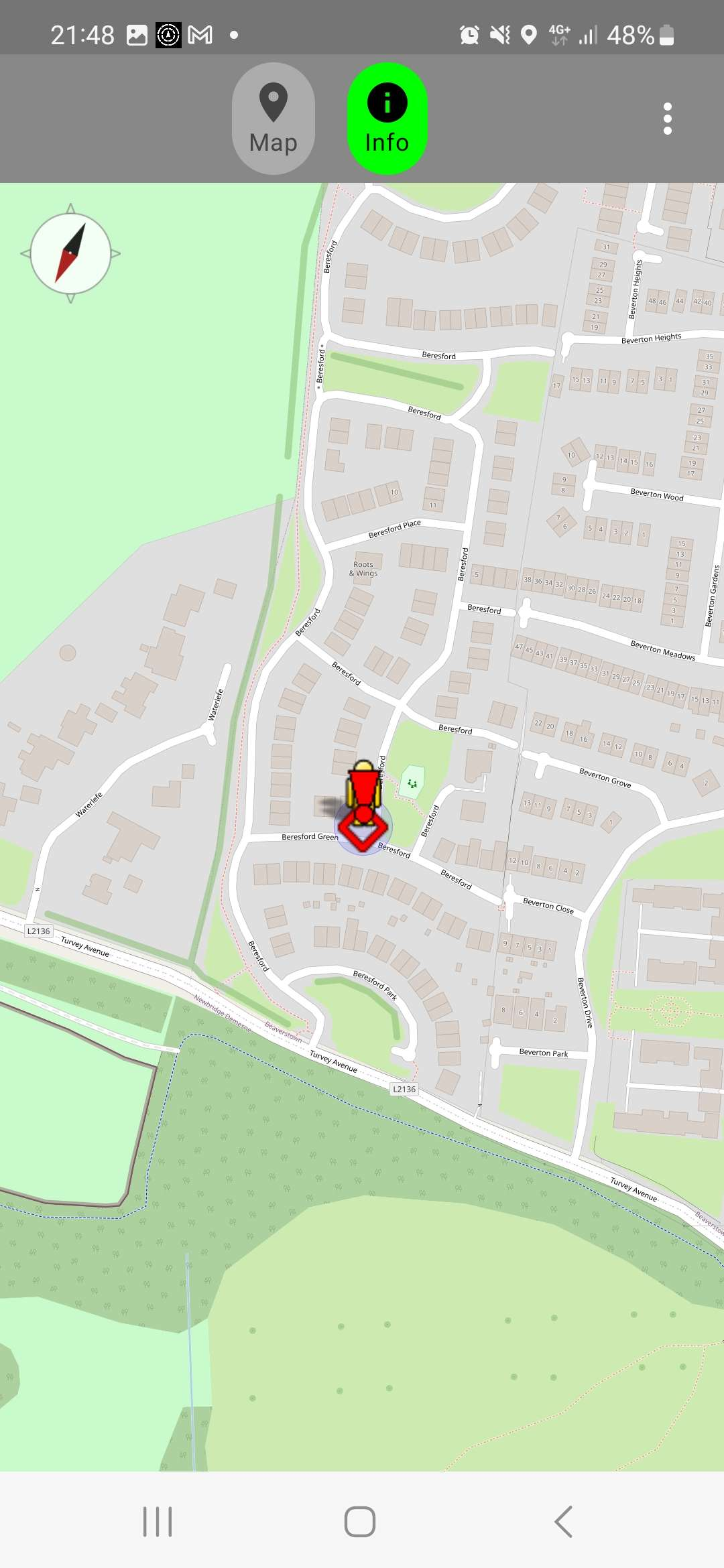
It works by firstly retrieving the player data from the Firebase from the current session, it then loads this data into a list of users that we will use with a Lazy List to display them as a scrollable list, Lazy list is composes implementation of a recycler list. In order to get the lazy list to display all the data we want and all the admin options desired in each row, A new compose-able called PlayerListItem was created this composable consists of all the data that is going to be displayed by the list and includes a drop down menu, that only shows for admins that allows them to manage each individual player. Using Lazy List and compose was a significantly better approach then using XML and recycler lists as it allowed me to customise how the list was processed and displayed in order to get the effect I wanted.

Figure 20: player list code snippet

**SOS and Requesting AID**

The SOS features goal is simple, there needs to be a way for users to be able to indicate that they need help or are in an emergency, in order to do this a few things had to be implemented, Firstly Cloud functions had to be set up with cloud messaging in order to be able to send notifications to admins even if they had their phone closed secondly there needs to be a way that when the user indicates during the stress that will mark it on the map so everyone can see that they need help.

For the cloud functions the function to send a notification would be triggered by a Https request containing the session id and the intended message sent from the application, the cloud function then processes this request, gets all the admins notification tokens from the database and then uses firebase cloud messaging to send a notification to each of those users.

****

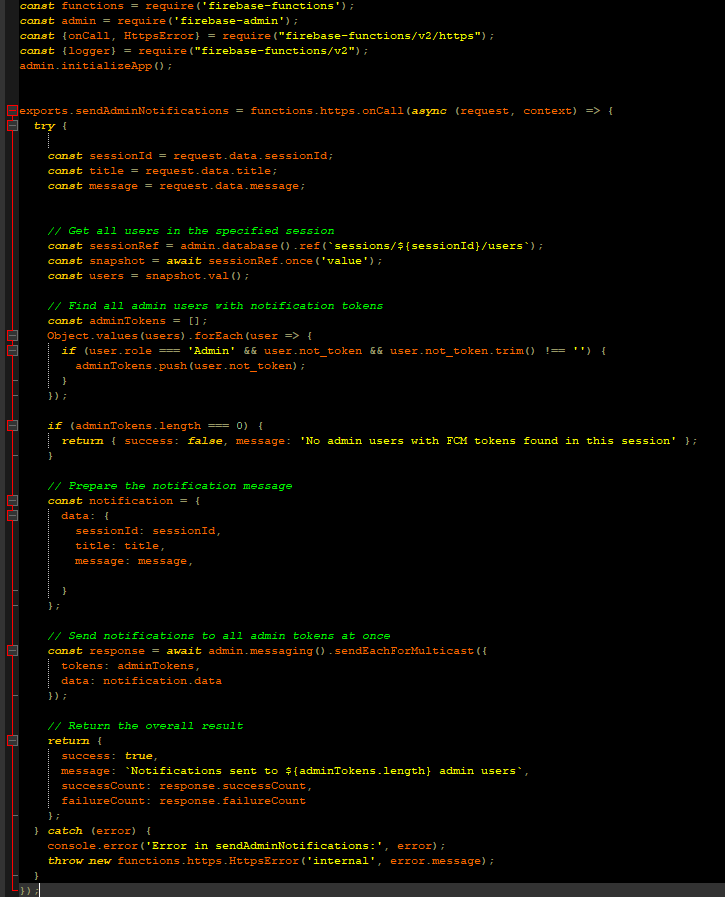
Figure 21: cloud function and map with marker.

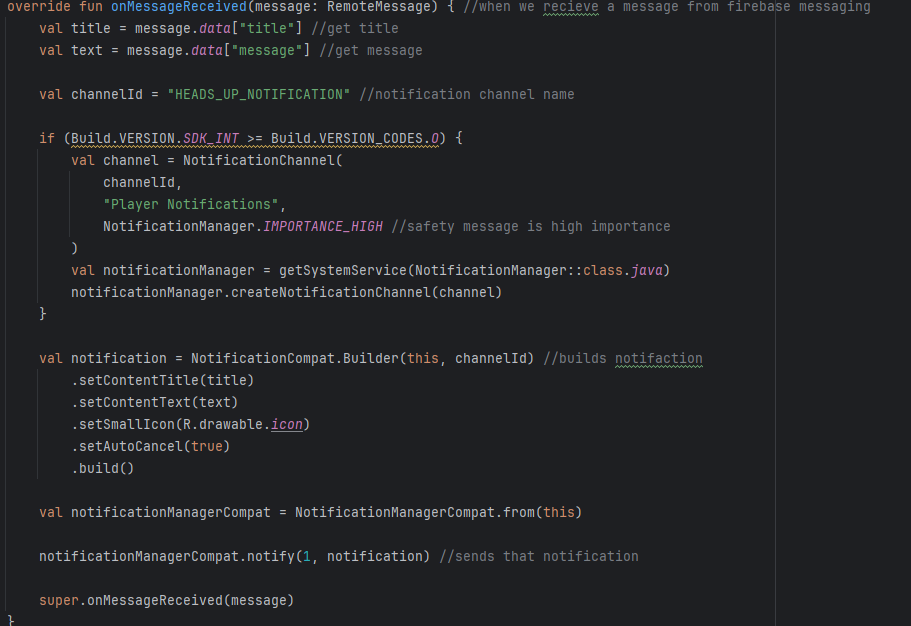
Figure 22: ping function

The ping and urgent ping functions purpose is to send the https request to the cloud function talked about earlier, it formats the request using map to match the expected input of the function and then sends it, if it is successful it logs it.

In order to make sure that users only send notifications when needed a pop-up was introduced to make sure user don't accidentally send notifications, as this was an issue encountered in the live test.

Then finally to display the notification sent from firebase messaging on the phone Notification service was created, Notification service is a background service that's only job is to wait to receive a message from Firebase messaging and to display it on the phone.

It does this by overriding the FirebaseMessagingService, which is a class that is provided by firebase messaging to implement receiving notifications. In order to show the notification we implement the onMessageReceived method which is called when firebase messaging sends a notification to the phone. To send the notification we first setup a notification channel for our message and set the importance, which in this case will be high, then we build the notification using a builder. Finally we send the notification through the NotifactionManagerCompat. Finally in order for our service to be recognised it was also declared in the android manifest as a service.

Figure 23: onMessageRecieved code

**Sites / Games**

The Sites / Games system purpose is to allow users to create sites and organise games underneath them, games will inherit some properties of sites, this is to reduce the amount of repetition when create game lists for a site. In addition another feature is the ability to add users to a site with their User ID, this gives them access to all the games within the site, this is needed as a lot of sites have more than one marshal working at a time. Then finally the last feature of this system is to allow marshals to load in the previously created games into a session, loading the brief, markers and site info onto all session users.

The way this system works is by first getting the data of all the sites the current user is in, each site has a list of userIDs and for every site it finds with the current user in that list it will take it from the snapshot and save it as a site, this wasn't done as usual with the other lists such as player list as Site has a list in it of the UserIDs, so converting it directly into the object wont work, so instead we convert the data into maps which will allow us to extract the fields and put them into a site object. Then we get every site that contains the current users user ID and put it in a final site list.

Figure 24: site\_view\_screen code snippet

We then display this site list using lazy list, using a siteListItem composable to represent each site in the list, the siteListItem also has a drop down menu for editing the site, deleting it and adding users to it.

Figure 25: lazy list snippet

Then in order to create a game in a site a user has to simply press on the site in the list, it will then go to a different lazy list screen that displays all the games in that site, it does this the same way as the site list, however since the game objects do not contain a list we can just directly map the snapshot to the objects.



Then in the game list the user can edit the games by selecting them which navigates them to the editor or delete them by pressing the delete button on the list item.

**Editor**

The purpose of the Editor is to add a way for user to make and edit games and sites through the app.

The editor was developed as a composable function so it could be navigated to easily from the sites or game view pages when these are in edit mode, it was decided that users shouldn't be able to edit games mid game as this would cause issues and increase complexity. When the composable is first loaded it will load on the markers if there are any from the game into a markers mutable list, the markers mutable list is where all the markers that are in the game/site are stored while they are being edited new markers will be added in here.

In order for users to be able to edit their site easily and quickly a map view was created with a cross hair in the middle to designate where markers would be placed to the user, the reason a cross hair was used rather than creation where tapping as the cross hair allows for much more precise creation of markers, initially the only type of map mark that was able to be created was the marker, marker creation is handled by a composable function called editor dialog, which is a dialog that takes in the information such as title description and what marker team and type of marker does the user want to create and on confirm sends a callback with all the info need to create the marker.

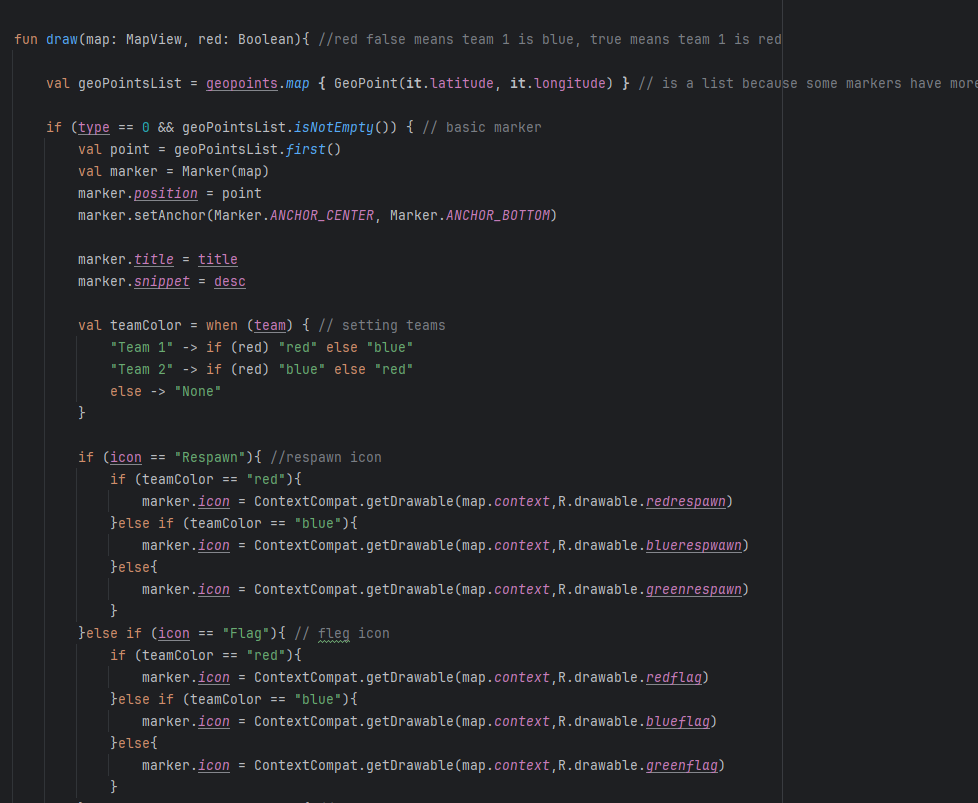
Figure 26: mapview

Then to enable user to add more detail to their site maps, the ability to add polygons and lines was added as well, these are to mark out structures or areas that may not appear on the map normally, their creation is similar to the above marker however it uses a different dialog as it needs different parameters.

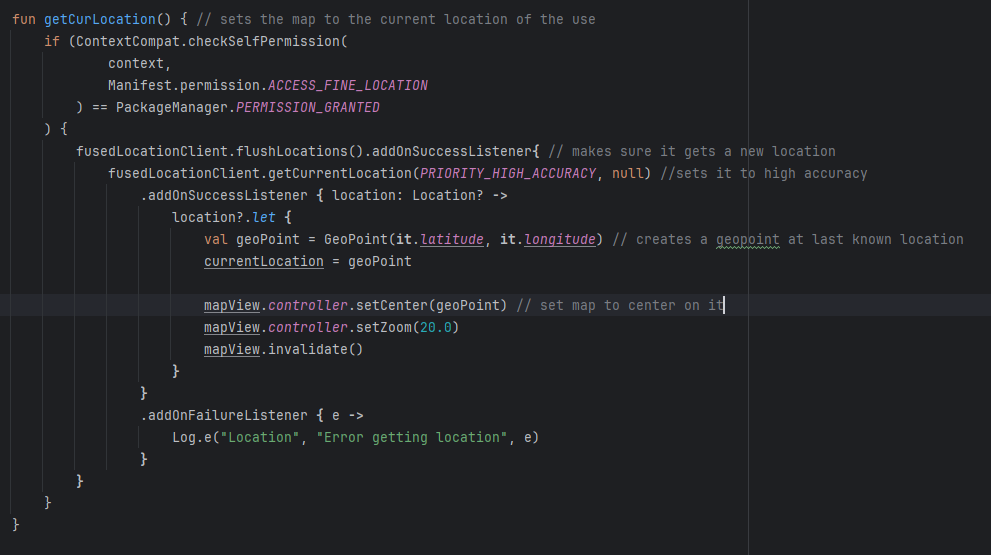
Figure 27: polyline editor code snippet

In order to have a way of swapping between each marker type a drop-down menu with each marker type in it was created and when a marker type is selected from this menu it will change the UI to what this marker needs.

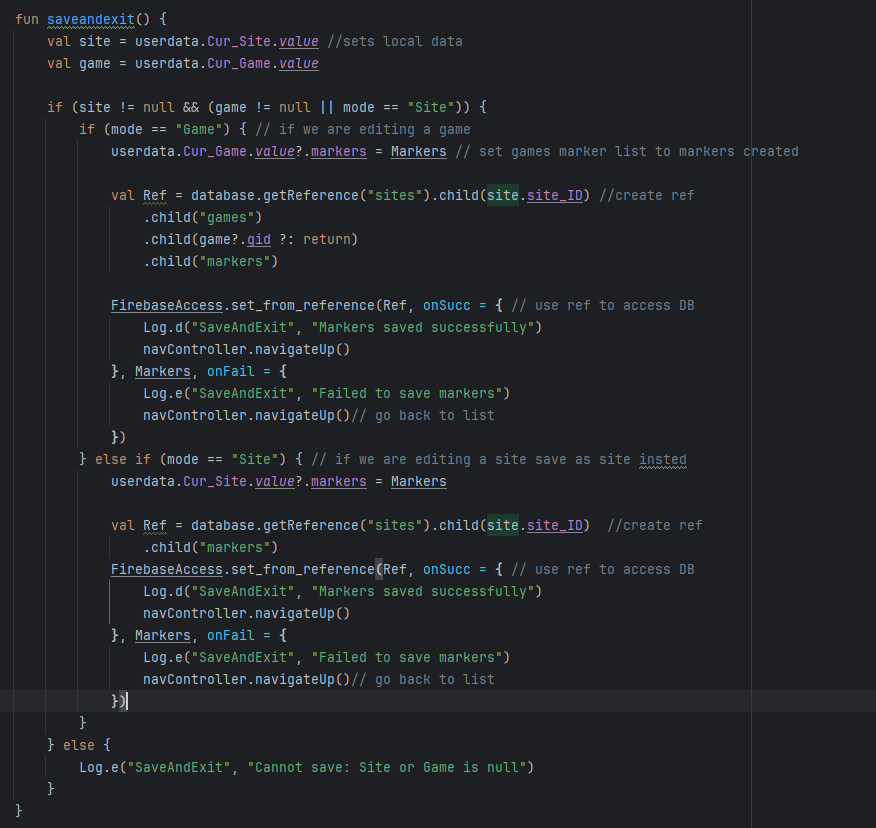
When a marker is created it is created as a MapObject, the map object was created to handle every type of marker in the application and it contains all the logic to draw itself so when a list of the object is iterated over it is simple to implement drawing them, in addition both site and game objects have a draw function that goes over the list of MapObject inside them and calls draw for each of them. When the mapObject is finalised it is added to a markers mutable list which will be used to save them later.

Figure 28: map object code snippet

One issue encountered in some testing was placing markers accurately, in some cases the map provided by Open street maps wasn't helpful in placing markers, so the my location button was added, it calls the getCurLocation when pressed. The way this works is it uses a fused location client to get the user last location and then moves the map and the cross hair to that last location, one issue encountered in testing was that if two points where to close to each other the my location button would not update and it would just go to the previous location, to fix this the latest location is flushed each time the function is called.

Figure 29: get cur location

And finally to save the created map object list to either the game or site that is being edited once the user presses the back arrow, it will automatically save and exit, calling the save and exit function, this function was developed in order to handle saving and exiting from the editor, it works by firstly making sure a game or a site is saved in the app data, this is what's being edited. It then sets the markers list in the database in that selected site/game to whatever is in the markers list currently, it will then navigate back to either the game or site list depending on what was being edited.

Figure 30: saveandexit

The Editor function is quite large and complicated, however it fufils the use cases for editing sites and games quite well.

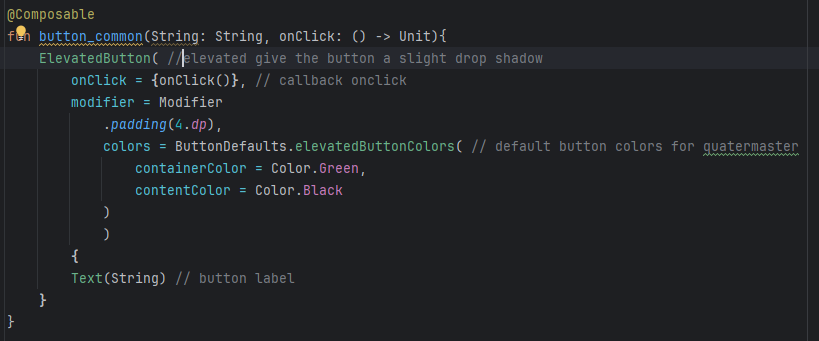
**UI**

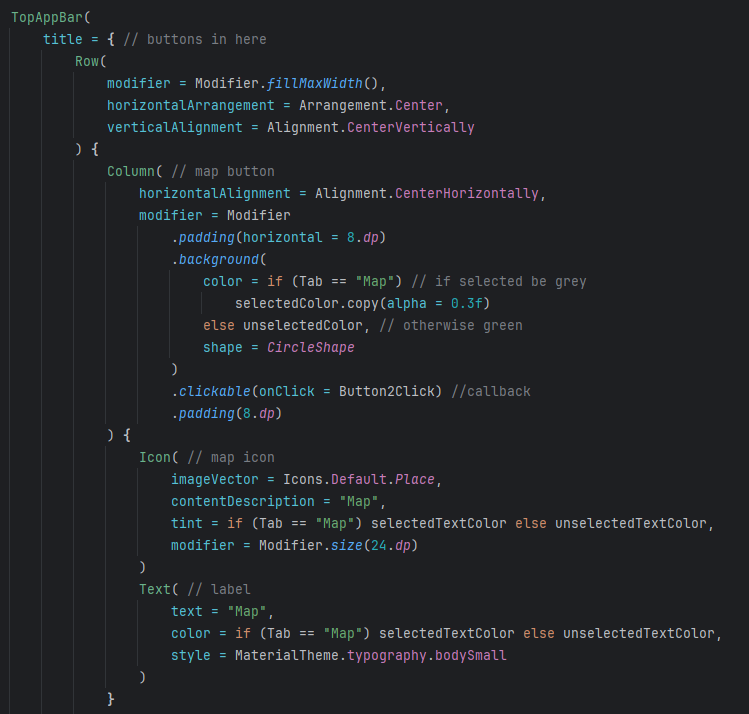
In android native development Jetpack compose is the latest way to develop UI for applications, it works in a programmatic way and each UI element can be turned into a function to allow it to be reused saving massive amounts of time, the UI is very important as without a good UI the app quality and usability suffers.

In order to implement the UI set out in the design section in compose UI elements such as navbars and common dialog and buttons where all split into different compose functions.

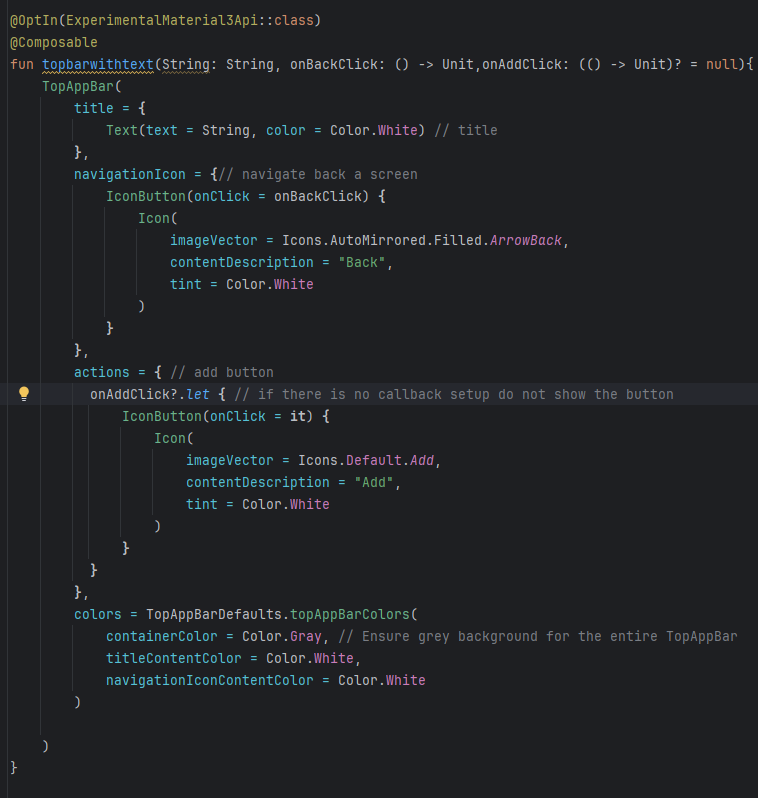
common\_button purpose is to standardise all buttons across the entire interface, its very simple functioning as a wrapper for a button, with callbacks existing for on click and a standard modifier.

Gametopbar is the App bar that is drawn at the top of the game map and game info pages, its purpose is the facilitate the navigation through all of the session side functionality of the app, on the top there are 2 icon buttons one for map and one for info, the unselected button appears green as it is intractable while the selected one appears greyed out to show that it isn't. This was implemented using the top app bar compose function which is an inbuilt compose function designed to make appbars.

Figure 31: common button function

Figure 32: game top bar code snippet

Another composeable made was the topbarwithtext, this function is designed to be used with the game select and site select screens similar to the previous bar it uses the TopAppBar composable to create a top bar with a back button and a title, it also has a callback for an add button which is used in the create and edit site game lists to add a site or a game to the displayed list, otherwise if there is no callback setup it will not show the button.

Figure 33: top bar with text

Another top bar was created for the editor as initially the topbar with text was used however the addition of a dropdown menu was needed to allow users to select what type of marker they wanted, this composable is mainly a modifaction of topbar with text, however it has alot more callbacks to accommodate each drop-down menu item.

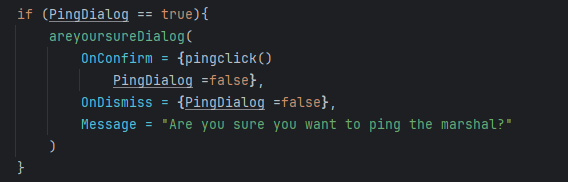
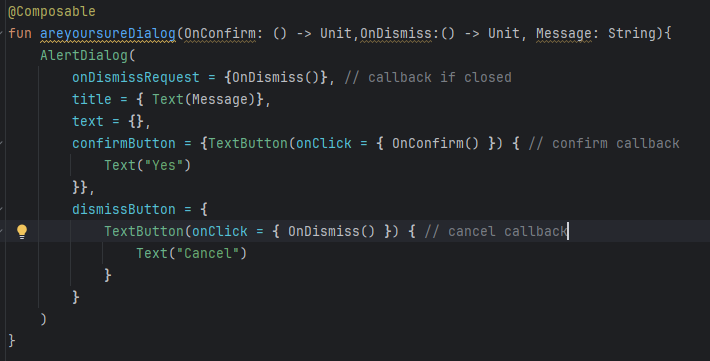


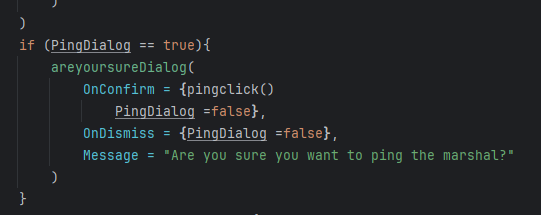
Figure 34: snippet of editor top bar

Finally for the last top bar created for the app is the infotopbar, this composable is used to create the topbar seen on the main screen, the only difference between this and top bar with text, is the inclusion of a dropdown, to give the players options to change username, show user ID and to access the sites list for editing, in addition since this is the root of the navhost, it dose not have a back button.

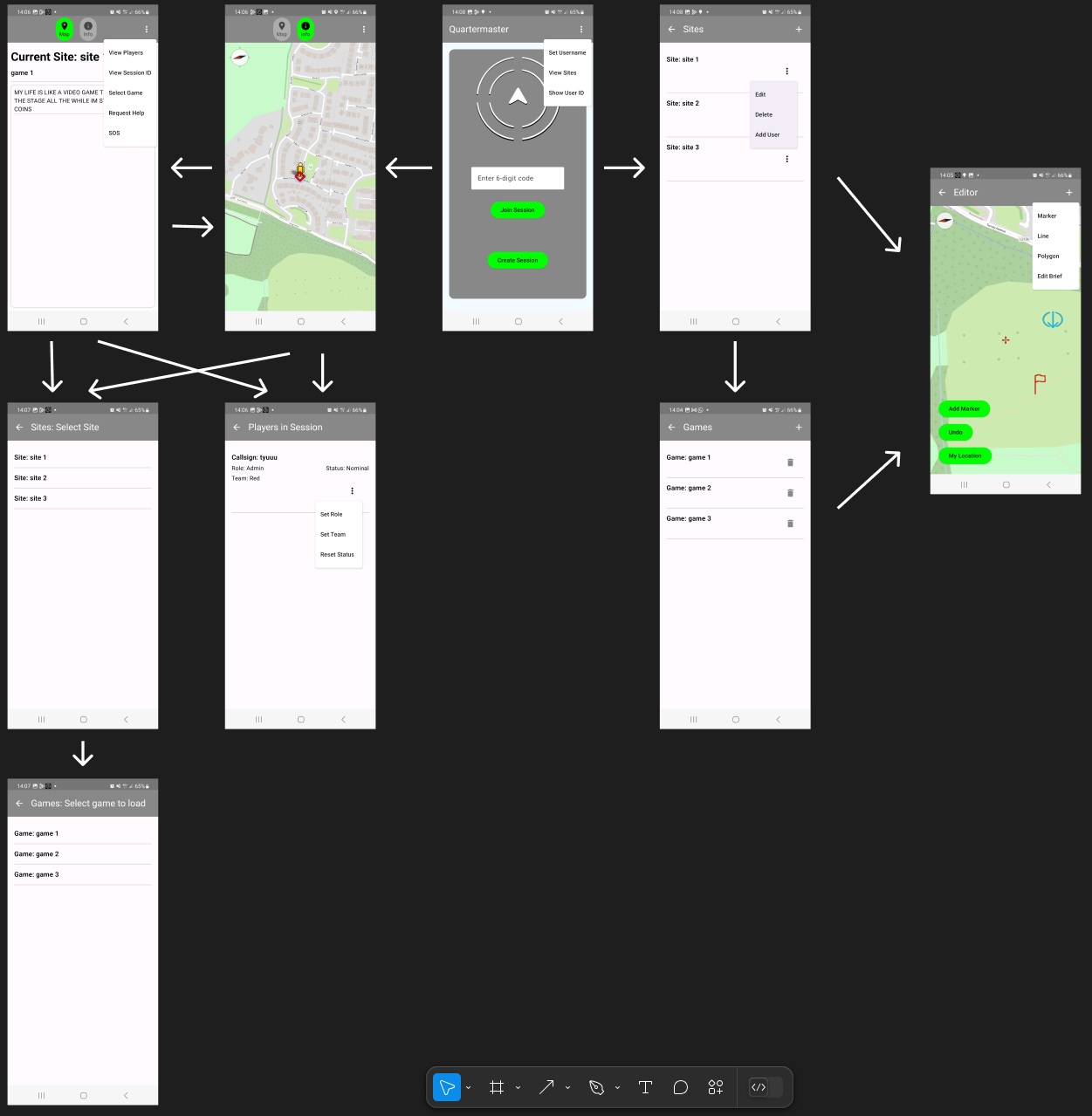
Figure 35: info top bar

In addition many dialogues where used throughout the project these dialogues where created using Jetpack compose, some of these where for text input such as when the user wants to change their username, other times they are used to confirm actions the user takes, all of these work by having the compose function have a remember showdialog boolean which controls if the dialog is to be shown, then if it is true there is a if statement that will call the dialog function to display it.

Figure 36: areyousuredialog function

Figure 37: are you sure getting called

**Final UI**



above is the final UI in the latest build of the application, it was built to keep with the design plan set out in chapter 3. it was kept as minimalistic and simple as possible to aid navigation and identifaction of elements.

## 4.3. Other Sections

**Issues Encountered and solved**

Issue 1: Learning Compose and Kotlin

The First Issue encountered was my lack of experience with creating a large project like this with Kotlin and compose, previously I have worked with developing android with java and XML but clearly android development was moving to Kotlin with compose so it was the only option, this lead to delays and mistakes in the code as I began to learn how compose and Kotlin based android development worked, this was solved by learning through doing and by the end of the development process I had a much better grasp on how to use these tools effectively.

Issue 2: Tracking in background

The second Issue encountered was a that my original method of tracking location didn't work while the app was in the background or when the phone was closed, this was a major flaw that seriously effect the effectiveness of the application, in order to fix this research was done into using some kind of service to handle the tracking which lead into using a foreground service to track the location, foreground services are services that notify the user that they are running and what they are doing. Its recommended that any location tracking service is a foreground service for user privacy. I then modified my original code and used my research to create a foreground service that tracked the location of my user, even when the phone was closed resolving this issue.

Issue 3: Firebase Messaging

The Third issue encountered was how the notifications for the SOS feature where going to be handled originally the plan was to add onto the location service and have it check the session for either updates to each users status or to have an entirely new “messages” part of each session that would contains SOS or other messages for future functionality, however I felt that this approach wasn't optimal as it could lead to 15 second delays for these very important and time sensitive notifications. The solution to this issue was using cloud functions with cloud messaging to handle the sending of the notifications through sending https requests from the app to the function. The issue with this was that cloud functions required the paid tier of firebase known as “blaze”. I didn't want to do this as google has charged me before a large amount of money after a VM was left running to long. In the end the blaze plan was purchased however spending would be watched carefully.

## 4.4. Conclusions

This chapter covered the development of the Quartermaster app going through each of the major features developed and how they work, it also covers how the UI was created using Compose as well as challenges encountered when developing the application.

# 5. Testing and Evaluation

## 5.1 Introduction

In this chapter we will go through the various testing and evaluation done to the application, in it we go through the types of testing done such as white and black box tests, discussing how each test was carried out and created and any issues encountered. Finally we will evaluate the application based on the original requirements.

## 5.2 White Box Testing

**Introduction**

White box testing is a from testing that assumes that the tester has full knowledge of the code and its structure, this method of testing is mainly used to test the correctness of code and to find logical errors. In this section we will cover how testing was created what tests where run and conclusions drawn from these tests

**Implementing White Box testing on an android environment**

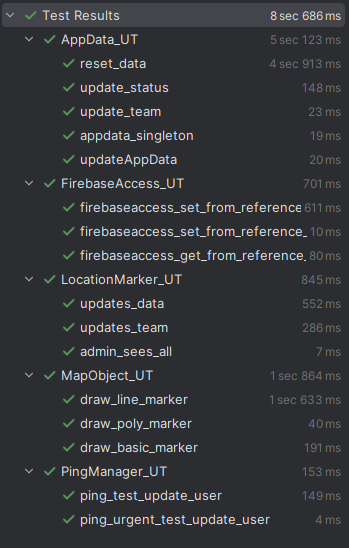
Due to Quartermaster being native android application testing becomes significantly harder as much of the internal code cannot be unit tested due to it needing to run on an android environment and not in a local Java Virtual Machine. This created limitation in the types of test that could be run. As any android specific code or class that didnt exist on the local Java Virtual Machine would not be able to be tested. The way this was fixed was by using Robolectric, Robolectric is a framework that allows unit tests to run in a simulated android environment, solving this issue.

The library used to create these tests in the end was the Mockk library as it is designed for use with Kotlin and can successfully mock final classes when doing testing. Which was an issues encountered when trying to use mockito.

### Unit Tests

Unit testing is a white box testing method where individual components of the application are tested in isolation to ensure they work. Unit testing is useful for catching bugs when code is changed and ensuring that outputs stay consistent.

In the end 16 unit tests where created to test the basic function of the application. These tests where designed to cover the basic functionality of the app and to make sure that important functions return the desired results.

Figure 38: Unit Tests

#### AppData\_UT

**Setup**

In order to test AppData first we have to mock what AppData needs to function such as the application context, the firebase database and the firebase access object. We then can create our AppData object using the mock application context. Then we have to override some functions in the mock object so they return our mock objects instead of trying to get a real one.

private lateinit var appData: AppData  
private val mockApp: Application = *mockk*(relaxed = true)  
var mockAccess = *mockk*<FirebaseAccess>(relaxed = true)  
val mockRef = *mockk*<DatabaseReference>(relaxed = true)  
val mockDatabase: FirebaseDatabase =*mockk*()  
  
@Before  
fun setup(){  
 *mockkStatic*(FirebaseDatabase::class)  
 appData = AppData.getInstance(mockApp)  
 appData.FirebaseAccess = mockAccess  
 *every* **{** FirebaseDatabase.getInstance() **}** returns mockDatabase  
 *every* **{** mockRef.child(any()) **}** returns mockRef  
 *every* **{** mockDatabase.getReference(any()) **}** returns mockRef  
}

**Update Appdata**

The function to update some values inside AppData was tested, this is a simple test as we are just asserting that the data matches the expected values after the update.

@Test  
fun updateAppData(){  
 //act  
 appData.updateAppData("123","321","Admin","Red","Nominal")  
 //assert  
 assertEquals("123", appData.user\_ID.*value*)  
 assertEquals("321", appData.Session\_ID.*value*)  
 assertEquals("Admin", appData.Role.*value*)  
 assertEquals("Red", appData.Team.*value*)  
 assertEquals("Nominal", appData.Status.*value*)  
}

**reset\_Data**

The function to reset the values inside AppData was tested. To test it some junk data was set, then the reset\_data function was called and then the appdata was checked to make sure it was reset.

@Test  
fun reset\_data(){  
 //arrange  
 appData.updateAppData("123","321","Admin","Red","Nominal")  
 //act  
 appData.reset\_data()  
 //assert  
 assertEquals("", appData.user\_ID.*value*)  
 assertEquals("", appData.Session\_ID.*value*)  
 assertEquals("Player", appData.Role.*value*)  
 assertEquals("Not Set", appData.Team.*value*)  
 assertEquals("Nominal", appData.Status.*value*)  
 assertEquals("", appData.Username.*value*)  
 assertNull(appData.Cur\_Site.*value*)  
 assertNull(appData.Cur\_Game.*value*)  
  
}

**Update team**

The function to update a players team was tested. To test it update\_team was called and then is was checked to see if mockAccess had set\_from\_reference called and then checked that the team in appdata was updated.

@Test  
fun update\_team(){  
 //act  
 appData.update\_team("Red")  
 //assert  
 *verify* **{** mockAccess.set\_from\_reference(  
 ref = mockRef,  
 onSucc = any(),  
 set = "Red",  
 onFail = any()  
 )  
 **}** assertEquals("Red", appData.Team.*value*)  
}

**update\_status**

The function to update a players status was also tested. To test it update status was called and then it was checked to see if mockAccess had set\_from\_reference called and then checked that the status was updated in AppData.

@Test  
fun update\_status(){  
 //act  
 appData.update\_status("I NEED A MEDIC BAG")  
 //assert  
 *verify* **{** mockAccess.set\_from\_reference(  
 ref = mockRef,  
 onSucc = any(),  
 set = "I NEED A MEDIC BAG",  
 onFail = any()  
 )  
 **}** assertEquals("I NEED A MEDIC BAG", appData.Status.*value*)  
}

**appdata\_singleton**

The singleton nature of appdata was also tested. This is to make sure that 2 appdata object cannot exist at the same time and that if getInstance is called to get appdata that its the same object instance as before.

@Test  
fun appdata\_singleton(){  
 //act  
 val testinstance = AppData.getInstance(mockApp)  
 testinstance.Team.*value* = "blah"  
 //assert  
 assertEquals(appData,testinstance)  
 assertEquals(appData.Team.*value*, "blah")  
}

#### FirebaseAccess\_UT

**Setup**

In order to setup testing for firebase access mocks where created of all methods that are needed for Firebase Access to work including mocking the database ref, mocking the task and mocking the on success and fail of the callbacks. Finally in setup the firebaseAccess object that will be tested is created.

private lateinit var firebaseAccess: FirebaseAccess  
val mockRef = *mockk*<DatabaseReference>(relaxed = true)  
val mockTask = *mockk*<Task<Void>>(relaxed = true)  
val onSucc = *mockk*<() -> Unit>(relaxed = true)  
val onFail = *mockk*<(Exception) -> Unit>(relaxed = true)  
  
@Before  
fun setup() {  
 firebaseAccess = FirebaseAccess()  
}

**firebaseaccess\_get\_from\_reference\_callback**

The callback from get from reference was tested to make sure it worked and that it returned a snapshot. To test if the callback worked a mock snapshot and it was setup so that when my mockRef had addListenerforsingleeventvalue was called its listener would be captured then the mockSnap could be simulated later, then we assert that the result of the get from reference is the same as the mockSnap we captured earlier confirming that its working.

@Test  
fun firebaseaccess\_get\_from\_reference\_callback(){  
 //arrange  
 val mockSnap = *mockk*<DataSnapshot>()  
 val listenerSlot = *slot*<ValueEventListener>()  
 var result: DataSnapshot? = null  
  
 *every* **{** mockRef.addListenerForSingleValueEvent(capture(listenerSlot)) **}** answers **{}** //Act  
 firebaseAccess.get\_from\_reference(mockRef, callback = **{** snapshot **->** result = snapshot  
 **}**)  
 listenerSlot.captured.onDataChange(mockSnap)  
  
 // assert  
 *assert*(result == mockSnap)  
  
}

**firebaseaccess\_set\_from\_reference\_callback**

The successful callback from set from reference was tested to make sure it worked and that the database is successfully updated. To test if the database was updated by the function the setValue function that would be called was set so that it would return mocktask to simulate it being called, then the addonSuccessListener was set so that when it was called it would return mock task and the listener would be captured, then the set from reference is called and it is verified that the callback success was invoked aswell as that fail was not invoked.

@Test  
fun firebaseaccess\_set\_from\_reference\_callback(){  
 //arrange  
 val succSlot = *slot*<OnSuccessListener<Void>>()  
  
 *every* **{**mockRef.setValue(any()) **}** returns mockTask  
 *every* **{**mockTask.addOnSuccessListener(capture(succSlot)) **}** returns mockTask  
 //Act  
 firebaseAccess.set\_from\_reference(mockRef,onSucc,"BORGER",onFail)  
 succSlot.captured.onSuccess(null)  
 // assert  
 *verify***{** onSucc.invoke()**}** *verify*(exactly = 0)**{**onFail.invoke(any())**}**}

**firebaseaccess\_set\_from\_reference\_callback\_fail**

The fail callback from the set from reference function was tested to make sure that it would be called if the function failed. To test this much of the same approach of the previous test was used however since we are intending on triggering failure a failslot was added to capture the failure listener and an exception was created so that onFailure could be called successfully. Then the listeners where setup to be captured into the slots and the set from reference was called and forced to fail. Then is was asserted that fail was invoked and success wasnt.

@Test  
fun firebaseaccess\_set\_from\_reference\_callback\_fail(){  
 //arrange  
 val excep = Exception("Failed")  
 val succSlot = *slot*<OnSuccessListener<Void>>()  
 val failSlot = *slot*<OnFailureListener>()  
  
 *every* **{**mockRef.setValue(any()) **}** returns mockTask  
 *every* **{**mockTask.addOnSuccessListener(capture(succSlot)) **}** returns mockTask  
 *every* **{**mockTask.addOnFailureListener(capture(failSlot)) **}** returns mockTask  
 //Act  
 firebaseAccess.set\_from\_reference(mockRef,onSucc,"BORGER",onFail)  
 failSlot.captured.onFailure(excep)  
 // assert  
 *verify***{** onFail.invoke(excep)**}** *verify*(exactly = 0)**{**onSucc.invoke()**}**}

#### LocationMarker\_UT

**Setup**

In Order to test location marker first mocks where created of what location marker needed to work such as the map view and Appdata which where then used to create the object we will test, map view isnt needed now but every function in location marker uses it.

private lateinit var locationMarker: LocationMarker  
private val mockUserData: AppData = *mockk*()  
private val mockMap: MapView = *mockk*()  
  
@Before  
fun setUp() {  
 locationMarker = LocationMarker(mockUserData, *mockk*())  
}

**Updates Data**

The updates data function of location marker was tested to make sure that when handleuserupdates is called it will update the appData with the database in order to maintain sync. To test this a testuser was created and a list of users with that user in it was also created to test on. Then we call handle userupdates with out mock data and assert that the user updated is our current user and that their team was updated.

fun updates\_data(){  
 //arrange  
 val currentUser = user(userId = "UID\_123", team = "Red", role = "Player")  
 val users = *listOf*(  
 user(userId = "UID\_123", team = "Blue", role = "Player"),  
 user(userId = "UID\_125", team = "Red", role = "Player")  
 )  
 var updatedUser: user? = null  
  
 //act  
 locationMarker.handleUserupdates(currentUser,users,mockMap, updateAppData = **{** user **->** updatedUser = user**}**)  
  
 //assert  
 *assert*(updatedUser != null)  
 *assert*(updatedUser?.userId == "UID\_123")  
 *assert*(updatedUser?.team == "Blue")  
}

**Updates Team**

The updates team test is designed to test the update marker function in location marker, it is specifically testing that it will draw the friendly users on the map and not the enemy user. To test this a fake list of users was created containing the current user a friendly user and a enemy user, then we use spyk to create a locationMarkerobject, spyk functions halfway between creating a mock and a real object, with every method running the same besides the one we specify which in this case is the update marker function, which we just want to run without throwing anything. Then we run the handle user updates on our suslocationMarker and then verify that update marker was called with our friendly user.

fun updates\_team(){  
 //arrange  
 val currentUser = user(userId = "UID\_123", team = "Red", role = "Player")  
 val users = *listOf*(  
 user(userId = "UID\_123", team = "Red", role = "Player"),  
 user(userId = "UID\_125", team = "Red", role = "Player"),  
 user(userId = "UID\_125", team = "Blue", role = "Player")  
 )  
  
 val susLocationMarker = *spyk*(locationMarker)  
 *every* **{** susLocationMarker.updatemarker(any(), any()) **}** *just* Runs  
 //act  
 susLocationMarker.handleUserupdates(  
 currentUser,  
 users,  
 mockMap,  
 updateAppData = **{}** )  
 //assert  
 *verify*(exactly = 1) **{** susLocationMarker.updatemarker(mockMap, users[1])  
 **}**}

**Admin sees all**

The admin sees all test is designed to test updatemarker in the case that the user is an admin, admins no matter the role can see all users for their safety. To test this we create a fake list of users we then spyk on the location marker and set its update marker to run without throwing errors. Then we act and call handle user updates on the location marker. Then verify that update marker was called for all non admin users.

fun admin\_sees\_all(){  
 //arrange  
 val currentUser = user(userId = "UID\_123", team = "Red", role = "Admin")  
 val users = *listOf*(  
 user(userId = "UID\_123", team = "Red", role = "Admin"),  
 user(userId = "UID\_125", team = "Red", role = "Player"),  
 user(userId = "UID\_125", team = "Blue", role = "Player")  
 )  
  
 val susLocationMarker = *spyk*(locationMarker)  
 *every* **{** susLocationMarker.updatemarker(any(), any()) **}** *just* Runs  
 //act  
 susLocationMarker.handleUserupdates(  
 currentUser,  
 users,  
 mockMap,  
 updateAppData = **{}** )  
 //assert  
 *verify* **{** susLocationMarker.updatemarker(mockMap, users[1]) **}** *verify* **{** susLocationMarker.updatemarker(mockMap, users[2]) **}**}

#### MapObject\_UT

**Setup**

In order to test MapObject first we must mock anything it needs to interact with to run, so we then mocked a mapview and context and created an overlay list to act as a place to store the overlays. When setup mock map so it returns our mock context and used overlay list to store the created overlays so we can validate them.

private val mockMap: MapView = *mockk*(relaxed = true)  
private val mockContext: Context = *mockk*(relaxed = true)  
private val overlayList = *mutableListOf*<Overlay>()  
  
@Before  
fun setup() {  
 *every* **{** mockMap.*context* **}** returns mockContext  
 *every* **{** mockMap.*overlays* **}** returns overlayList  
}

**draw basic marker**

The draw basic marker test is designed to test that basic map markers are being drawn on the map with the right parameters. To do this test firstly we create a geopoint we will use to create the marker and we create a mock drawable for the marker. We also create a slot to capture what drawable is passed so we can test what icon is getting drawn. We then create a mapObject with our mocked data and some filler values and then call that objects draw function. Finally we assert that the marker has been drawn by checking our overlay list and asserting that the marker matches the expected outcome.

fun draw\_basic\_marker(){  
 //arrange  
 val testGeopoint = *mutableListOf*(MapObject.GeoPointData(2.0,-1.0))  
 val mockDrawable :Drawable = *mockk*()  
  
 val slot = *slot*<Int>()  
 *every* **{** ContextCompat.getDrawable(mockContext, capture(slot)) **}** returns mockDrawable  
  
 val testmapObject = MapObject(  
 type = 0, // 0 for marker  
 title = "Test Title",  
 desc = "Test Desc",  
 icon = "Flag",  
 geopoints = testGeopoint  
 )  
 //act  
 testmapObject.draw(mockMap, true)  
 //assert  
 val marker = overlayList.*find* **{ it** is Marker **}** as Marker  
 *assert*(marker != null)  
 *assert*(marker.*title* == "Test Title")  
 *assert*(marker.*snippet* == "Test Desc")  
 *assert*(marker.*icon* == mockDrawable)  
}

**draw line**

The draw line marker test is designed to test if line markers are being drawn correctly. To do this test we created a list of two points that the line will be drawn between, then we created the test object and had it draw the line. Then we check the overlay list to see if the line has been added as a polyline and that the parameters of the line are correct.

fun draw\_line\_marker(){  
 //arrange  
 val testGeopoints = *mutableListOf*(MapObject.GeoPointData(2.0,-1.0), MapObject.GeoPointData(3.0,-2.0))  
  
 val testmapObject = MapObject(  
 type = 1, // 1 for line  
 title = "Test Title",  
 desc = "Test Desc",  
 geopoints = testGeopoints  
 )  
 //act  
 testmapObject.draw(mockMap, true)  
 //assert  
 val polyline = overlayList.*find* **{ it** is Polyline **}** as Polyline  
 *assert*(polyline.*actualPoints*.size == 2)  
 *assert*(polyline.*title* == "Test Title")  
 *assert*(polyline.*snippet* == "Test Desc")  
}

**draw polygon**

The draw poly marker test is designed to test if polygon markers are being drawn correctly. To do this a list of 4 geopoints are created to be the points of the test marker, when then create the marker and call the draw method with our mock map. Then we check out overlay list to assert that our polygon has been drawn correctly and with the right parameters.

@Test  
fun draw\_poly\_marker(){  
 //arrange  
 val testGeopoints = *mutableListOf*(MapObject.GeoPointData(2.0,-1.0),   
 MapObject.GeoPointData(3.0,-2.0),  
 MapObject.GeoPointData(3.0,-3.0),  
 MapObject.GeoPointData(4.0,-2.0))  
  
 val testmapObject = MapObject(  
 type = 2, // 2 for polygon  
 title = "Test Title",  
 desc = "Test Desc",  
 geopoints = testGeopoints  
 )  
 //act  
 testmapObject.draw(mockMap, true)  
 //assert  
 val polygon = overlayList.*find* **{ it** is Polygon **}** as Polygon  
 *assert*(polygon.*actualPoints*.size == 4)  
 *assert*(polygon.*title* == "Test Title")  
 *assert*(polygon.*snippet* == "Test Desc")  
}

#### PingManager\_UT

**Setup**

In order to test ping manager a way to test Https calls had to be setup, in order to do this everything that PingManager needed to run was mocked including appdata, firebase functions, a mock callable we use in place of a real one, a mock result and a mock task.

Then in the setup function we set what we want these mocks to-do when called and we setup the pingManager object that we will test.

private val mockAppData: AppData = *mockk*(relaxed = true)  
private val mockFirebaseFunction: FirebaseFunctions = *mockk*()  
private lateinit var pingManager: PingManager  
private val mockCallable: HttpsCallableReference = *mockk*()  
private val mockResult: HttpsCallableResult = *mockk*()  
private val mockTask: Task<HttpsCallableResult> = *mockk*()  
  
@Before  
fun setup(){  
 // setup https mocking  
 pingManager = PingManager(mockAppData,mockFirebaseFunction)  
 *every* **{** mockFirebaseFunction.getHttpsCallable("sendAdminNotifications") **}** returns mockCallable  
 *every* **{** mockCallable.call(any()) **}** returns mockTask  
 *every* **{** mockTask.*isSuccessful* **}** returns true  
 *every* **{** mockTask.continueWith<String>(any()) **}** returns Tasks.forResult("Success: Notification sent")  
 *every* **{** mockTask.*result* **}** returns mockResult  
}

**ping test update user / ping urgent test update user**

The ping test update user is designed to test if the ping function worked correctly and send a https request, To test this a result task was created that called the pingManager.ping on creation with a fake user id and passing the result into resultTask. We can then verify if it was called correctly by checking our appdata to see if the status was updated and if the result task is successful.

@Test  
fun ping\_test\_update\_user(){  
 //act  
 val resultTask = pingManager.ping("SID\_123").*result* //assert  
 *verify***{** mockAppData.update\_status("Help\_Needed")**}** assertEquals("Success: Notification sent",resultTask)  
}

@Test  
fun ping\_urgent\_test\_update\_user(){  
 //act  
 val resultTask = pingManager.urgentping("SID\_123").*result* //assert  
 *verify***{** mockAppData.update\_status("Critical")**}** assertEquals("Success: Notification sent",resultTask)  
}

### Instrumentation Tests

#### Introduction

Instrumentation tests are a type of functional test that are used in android, usually to test elements that cannot be tested in a Unit Test such as UI elements, Instrumentation tests unlike unit tests are ran on the android device you are testing allowing for interaction with UI elements and other androids systems that cant usually be tested.

Instrumentation test where used in Quartermaster to test UI elements and their functionality.

There where 6 instrumentation tests created for this application mainly testing interaction with the landing page.

#### 

#### Common\_Button\_IT

**Button\_common\_on\_click**

Button\_common\_on\_click is a test designed to test if the common\_buttons onclick callback works when it is pressed, it does this by using compose test rule, which is a way to load compose ui elements into the test for testing to create the button on a blank screen it then presses the button and checks to see if click is true and that the button is visible and displayed.

@get:Rule  
val composeTestRule = *createComposeRule*()  
  
@Test  
fun button\_common\_on\_click(){  
 //arrange  
 var click = false  
  
 composeTestRule.setContent **{** button\_common("TEST") **{** click = true  
 **}  
 }** //act  
  
 composeTestRule.*onNodeWithText*("TEST").*performClick*()  
  
 //assert  
 *assert*(click)  
 assertTrue(composeTestRule.*onNodeWithText*("TEST").*isDisplayed*())  
}

#### Create\_Session\_IT

**Create\_Session\_Success**

Create\_Session\_Success is a test designed to test how the create session button performs on a successful call and creates both the session and the user within it with admin privileges. The test does this by firstly creating all the mocks needed to run create session in a test environment and setting up those mocks to respond in a way needed to run the test. Then the compose test rule content was setup to display the button as it would be in the application, with the create session button having mocks filled in instead of normal data. We also set up slots to capture data to check later in the test to assert that all is functioning as it should.

val mockID = MutableLiveData("UID\_123")  
val mockUsername = MutableLiveData("Kozi")  
val mockMessaging = *mockk*<FirebaseMessaging>()  
val mockTask = *mockk*<Task<String>>()  
val mockListener = *slot*<OnCompleteListener<String>>()  
val userCaptor = *slot*<user>()  
val sessionIdCaptor = *slot*<String>()  
var onSucc = false  
val snackbarHostState = SnackbarHostState()  
val sessionCaptor = *slot*<session>()  
val onSuccCaptor = *slot*<() -> Unit>()  
  
*mockkStatic*(Random::class)  
*mockkStatic*(FirebaseMessaging::class)  
  
*every* **{** mockuserData.user\_ID **}** returns mockID //return user ID  
*every* **{** mockuserData.Username **}** returns mockUsername // return username  
*every* **{** Random.nextInt(100000,999999) **}** returns 123456 // return session id consitent  
*every* **{** FirebaseMessaging.getInstance() **}** returns mockMessaging //return mock instead  
*every* **{** mockMessaging.*token* **}** returns mockTask  
*every* **{**mockTask.addOnCompleteListener(capture(mockListener))**}** answers **{** val listen = mockListener.captured  
 val succTask = *mockk*<Task<String>> **{** *every* **{** *isSuccessful* **}** returns true  
 *every* **{** *result* **}** returns "fake-fcm-token"  
 *every* **{** *exception* **}** returns null  
 **}** listen.onComplete(succTask)  
 mockTask  
**}***every* **{**mockFirebaseAccess.set\_from\_reference(any(), capture(onSuccCaptor), capture(sessionCaptor), any())  
**}**answers **{** onSuccCaptor.captured.invoke()  
**}**composeTestRule.setContent **{** Scaffold(snackbarHost = **{** SnackbarHost(hostState = snackbarHostState) **}**) **{** paddingValues **->** Box(// make sure everything aligns  
 modifier = Modifier  
 .*fillMaxSize*()  
 .*padding*(paddingValues),  
 contentAlignment = Alignment.Center  
 ) **{** Create\_session(  
 userdata = mockuserData,  
 FirebaseAccess = mockFirebaseAccess,  
 onSucc = **{** user, SID **->** userCaptor.captured = user  
 sessionIdCaptor.captured =SID  
 onSucc = true  
 **}**,  
 snackbar = snackbarHostState  
 )  
 **}  
 }  
}**

then once the test is arranged we can test the button and then assert if both created the session and added the user, we do this by comparing our expected values to the values captured from the sessionCaptor and the user Captor

//act  
  
composeTestRule.*onNodeWithText*("Create Session").*performClick*()  
  
composeTestRule.waitUntil(5000)**{** onSucc **}**//assert  
  
assertTrue(onSucc)  
  
assertEquals("123456", sessionCaptor.captured.session\_Id)  
assertEquals(1, sessionCaptor.captured.users.size)  
assertTrue(sessionCaptor.captured.users.containsKey("UID\_123"))  
  
val user = sessionCaptor.captured.users["UID\_123"]  
assertNotNull(user)  
assertEquals("UID\_123", user?.userId)  
assertEquals("Kozi", user?.username)  
assertEquals("Admin", user?.role)  
  
assertEquals("UID\_123", userCaptor.captured.userId)  
assertEquals("Kozi", userCaptor.captured.username)  
assertEquals("Admin", userCaptor.captured.role)  
assertEquals("123456", sessionIdCaptor.captured)

**Create\_Session\_Fail**

Create Session Fail is a test deigned to test the behaviour of the application when creating a session fails for whatever reason and that it shows the user an error message to info the user why. In order to test this firstly all the objects that where needed for create session to be tested where mocked and then setup so they will respond as expected, in order to setup so the create session will fail it was setup so that set from reference would return a mock exception

//arrange  
val mockID = MutableLiveData("UID\_123")  
val mockUsername = MutableLiveData("Kozi")  
val mockMessaging = *mockk*<FirebaseMessaging>()  
val mockTask = *mockk*<Task<String>>()  
val mockListener = *slot*<OnCompleteListener<String>>()  
val userCaptor = *slot*<user>()  
val sessionIdCaptor = *slot*<String>()  
var onSucc = false  
val snackbarHostState = SnackbarHostState()  
val onFailCaptor = *slot*<(Exception) -> Unit>()  
  
*mockkStatic*(Random::class)  
*mockkStatic*(FirebaseMessaging::class)  
  
*every* **{** mockuserData.user\_ID **}** returns mockID  
*every* **{** mockuserData.Username **}** returns mockUsername  
*every* **{** Random.nextInt(100000,999999) **}** returns 123456  
*every* **{** FirebaseMessaging.getInstance() **}** returns mockMessaging  
*every* **{** mockMessaging.*token* **}** returns mockTask  
*every* **{**mockTask.addOnCompleteListener(capture(mockListener))**}** answers **{** val listen = mockListener.captured  
 val succTask = *mockk*<Task<String>> **{** *every* **{** *isSuccessful* **}** returns true  
 *every* **{** *result* **}** returns "fake-fcm-token"  
 *every* **{** *exception* **}** returns null  
 **}** listen.onComplete(succTask)  
 mockTask  
**}***every* **{**mockFirebaseAccess.set\_from\_reference(  
 any(),  
 any(),  
 any<session>(),  
 capture(onFailCaptor)  
)  
**}**answers **{** onFailCaptor.captured.invoke(*mockk*<Exception>())  
**}**composeTestRule.setContent **{** Scaffold(snackbarHost = **{** SnackbarHost(hostState = snackbarHostState) **}**) **{** paddingValues **->** Box(// make sure everything aligns  
 modifier = Modifier  
 .*fillMaxSize*()  
 .*padding*(paddingValues),  
 contentAlignment = Alignment.Center  
 ) **{** Create\_session(  
 userdata = mockuserData,  
 FirebaseAccess = mockFirebaseAccess,  
 onSucc = **{** user, SID **->** userCaptor.captured = user  
 sessionIdCaptor.captured =SID  
 onSucc = true  
 **}**,  
 snackbar = snackbarHostState  
 )  
 **}  
 }  
}**

Then once the mocks where ready and the compose was setup the create session button is pressed and then we wait until the failed to create session snackbar appears and we assert that it exists and that the session creation failed.

//act  
  
composeTestRule.*onNodeWithText*("Create Session").*performClick*()  
  
composeTestRule.waitUntil(5000)**{** try {  
 composeTestRule.*onNodeWithText*("Failed to Create Session").*isDisplayed*()  
 true  
} catch (\_: Exception) {  
 false  
} **}**//assert  
  
composeTestRule.*onNodeWithText*("Failed to Create Session").assertExists()  
  
assertFalse(onSucc)

#### Join\_Session\_IT

**Setup**

Before we can get to each test some setup needed to be done, in order to test functions that use compose a compose rule is created which allows us to create screens and test the buttons within them. In addition some common objects where mocked to save us from repeating code in the tests.

@get:Rule  
val composeTestRule = *createComposeRule*()  
  
var mockFirebaseAccess = *mockk*<FirebaseAccess>()  
var mockContext: Context = *mockk*()  
var mockuserData: AppData = *mockk*()

**Join\_Session\_Input\_Visible\_wrongcode**

Join session input visible wrongcode is a test designed to test what happens when a code is input and it is incorrect and matches with nothing in the snapshot. The expected behaviour is an error message informing the user. This was tested firstly by setting up a slot to capture the snapshot from the get from reference from out mock and have it return a fakesnapshot that always returns false when checked if a session exists, simulating a successful read that always fails to find. Then we setup the compose with the compose rule and act by pressing the join session button after having input a code. Then on a successful test the error message should appear explain why no session was joined.

fun Join\_Session\_Input\_Visable\_wrongcode() {  
  
 //arrange  
 val callbackCap = *slot*<(DataSnapshot) -> Unit>()  
 val mockID = MutableLiveData("UID\_123")  
 val mockUsername = MutableLiveData("Kozi")  
 val snackbarHostState = SnackbarHostState()  
  
 *every* **{** mockuserData.user\_ID **}** returns mockID  
 *every* **{** mockuserData.Username **}** returns mockUsername  
  
 *every* **{** mockFirebaseAccess.get\_from\_reference(any(), capture(callbackCap)) **}** answers **{** val fakeSnap = *mockk*<DataSnapshot> **{** *every* **{** exists() **}** returns false  
 **}** callbackCap.captured.invoke(fakeSnap)  
 **}** composeTestRule.setContent **{** Scaffold(snackbarHost = **{** SnackbarHost(hostState = snackbarHostState) **}**) **{** paddingValues **->** Box(// make sure everything aligns  
 modifier = Modifier  
 .*fillMaxSize*()  
 .*padding*(paddingValues),  
 contentAlignment = Alignment.Center  
 ) **{** Join\_session(  
 userdata = mockuserData,  
 firebaseAccess = mockFirebaseAccess,  
 onSucc = **{** \_, \_ **-> }**,  
 snackbar = snackbarHostState  
 )  
 **}  
 }  
 }** //act  
  
 composeTestRule.*onNodeWithText*("Enter 6-digit code").*performTextInput*("123456")  
  
 composeTestRule.*onNodeWithText*("Join Session").*performClick*()  
  
 //assert  
 composeTestRule.waitUntil(5000) **{** composeTestRule.*onAllNodesWithText*("Failed to Join Session - Code Invalid")  
 .fetchSemanticsNodes().*isNotEmpty*()  
 **}** composeTestRule.*onNodeWithText*("Failed to Join Session - Code Invalid").assertExists()  
}

**Join\_Session\_Input\_Visible\_rightcode**

Join session input visible right code is a test designed to test what happens when join session is pressed with a correct code in the input field. The expected outcome is that the user is added to that session. This was tested by firstly setting up the fake user data and setting up some slots to capture data to verify later such as the code and the user added to the session. Get from reference is setup to always return the fake snapshot simulating a successful read. Then the data being written to the firebase such as the user to the session is simulated. Then the compose test rule is setup to display the join session button and input. Finally the button is clicked and we assert that join session was called and that the fake user was added to the fake snap by using the slots we setup earlier.

@Test  
fun Join\_Session\_Input\_Visable\_Rightcode() {  
  
 //arrange  
  
 val callbackCap = *slot*<(DataSnapshot) -> Unit>()  
 val mockID = MutableLiveData("UID\_123")  
 val mockUsername = MutableLiveData("Kozi")  
 val codeCaptor = *slot*<String>()  
 val userCaptor = *slot*<user>()  
 var SuccCalled = false  
  
 *every* **{** mockuserData.user\_ID **}** returns mockID  
 *every* **{** mockuserData.Username **}** returns mockUsername  
  
 *every* **{** mockFirebaseAccess.get\_from\_reference(any(), capture(callbackCap)) **}** answers **{** val fakeSnap = *mockk*<DataSnapshot> **{** *every* **{** exists() **}** returns true  
 **}** callbackCap.captured.invoke(fakeSnap)  
 **}** val onSucc = *slot*<()-> Unit>()  
 *every***{** mockFirebaseAccess.set\_from\_reference(any(), capture(onSucc), any(), any())  
 **}** answers **{** onSucc.captured.invoke()  
 **}** val snackbarHostState = SnackbarHostState()  
 composeTestRule.setContent **{** Scaffold(snackbarHost = **{** SnackbarHost(hostState = snackbarHostState) **}**) **{** paddingValues **->** Box(// make sure everything aligns  
 modifier = Modifier  
 .*fillMaxSize*()  
 .*padding*(paddingValues),  
 contentAlignment = Alignment.Center  
 ) **{** Join\_session(  
 userdata = mockuserData,  
 firebaseAccess = mockFirebaseAccess,  
 onSucc = **{** user, input\_code **->** userCaptor.captured = user  
 codeCaptor.captured = input\_code  
 SuccCalled = true  
 **}**,  
 snackbar = snackbarHostState  
 )  
 **}  
 }  
 }** //act  
  
 composeTestRule.*onNodeWithText*("Enter 6-digit code").*performTextInput*("123456")  
  
 composeTestRule.*onNodeWithText*("Join Session").*performClick*()  
  
 composeTestRule.waitUntil(5000) **{** SuccCalled  
 **}** //assert  
  
 assertTrue(SuccCalled)  
 assertEquals("UID\_123",userCaptor.captured.userId)  
 assertEquals("Kozi",userCaptor.captured.username)  
 assertEquals("123456", codeCaptor.captured)  
}

**Join\_Session\_Input\_Visible\_nocode**

Join Session input visible nocode is a test designed to test what happens when join session is pressed with no code and it is expecting to see an error message explaining why. This was tested firstly by setting up the compose screen using compose rule to display the join session button then acting by pressing the button in it. On a successful test it will only pop up with an error message explaining why no session was joined.

//arrange  
 val snackbarHostState = SnackbarHostState()  
 composeTestRule.setContent **{** Scaffold(snackbarHost = **{** SnackbarHost(hostState = snackbarHostState) **}**) **{** paddingValues **->** Box(// make sure everything aligns  
 modifier = Modifier  
 .*fillMaxSize*()  
 .*padding*(paddingValues),  
 contentAlignment = Alignment.Center  
 ) **{** Join\_session(  
 userdata = mockuserData,  
 firebaseAccess = mockFirebaseAccess,  
 onSucc = **{** \_, \_ **-> }**,  
 snackbar = snackbarHostState  
 )  
 **}  
 }  
 }** //act  
   
 composeTestRule.*onNodeWithText*("Join Session").*performClick*()  
  
 //assert  
  
 composeTestRule.waitUntil(5000) **{** composeTestRule.*onAllNodesWithText*("Code must be 6 digits long")  
 .fetchSemanticsNodes().*isNotEmpty*()  
 **}** val inputField = composeTestRule.*onNodeWithText*("Enter 6-digit code")  
 assertTrue(inputField.*isDisplayed*())  
  
 composeTestRule.*onNodeWithText*("Code must be 6 digits long").assertExists()  
}

## 5.3 Black Box Testing

**Introduction**

Black box testing is a form of testing that assumes that the tester has zero knowledge of the internals of the application, this method of testing is mainly used to test how well users interact with the application.

**Usability Testing**

## 5.4 Evaluation

### Live User Test

## 5.5 Conclusion

# 6. Issues and Future Work

# Bibliography

[1] ‘Web Apps vs. Native Apps vs. Hybrid Apps - Difference Between Types of Web and Mobile Applications - AWS’, Amazon Web Services, Inc. Accessed: Nov. 13, 2024. [Online]. Available: <https://aws.amazon.com/compare/the-difference-between-web-apps-native-apps-and-hybrid-apps/>

[2]‘Kotlin vs Java: the 12 differences you should know’. Accessed: Nov. 13, 2024. [Online]. Available: <https://www.imaginarycloud.com/blog/kotlin-vs-java>

[3] ‘What Is NoSQL? NoSQL Databases Explained’, MongoDB. Accessed: Nov. 16, 2024. [Online]. Available: <https://www.mongodb.com/resources/basics/databases/nosql-explained>

[4]‘NoSQL Vs SQL Databases’, MongoDB. Accessed: Nov. 16, 2024. [Online]. Available: <https://www.mongodb.com/resources/basics/databases/nosql-explained/nosql-vs-sql>

[5] T. Lagos Jenschke, M. Dias de Amorim, and S. Fdida, ‘Nearby connections strategies: Features, usage, and empirical performance evaluation’, *Internet of Things*, vol. 23, p. 100895, Oct. 2023, doi: [10.1016/j.iot.2023.100895](https://doi.org/10.1016/j.iot.2023.100895).

[6] ‘Overview | Nearby Connections’, Google for Developers. Accessed: Nov. 16, 2024. [Online]. Available: <https://developers.google.com/nearby/connections/overview>

[7] ‘Mapbox vs Google Maps — What are the differences?’, SoftKraft. Accessed: Nov. 18, 2024. [Online]. Available: <https://www.softkraft.co/mapbox-vs-google-maps/>

[8]H. Mehta, P. Kanani, and P. Lande, ‘Google Maps’, *IJCA*, vol. 178, no. 8, pp. 41–46, May 2019, doi: [10.5120/ijca2019918791](https://doi.org/10.5120/ijca2019918791).

[9] J. Vargas, S. Srivastava, D. Tuia, and A. Falcao, ‘OpenStreetMap: Challenges and Opportunities in Machine Learning and Remote Sensing’, *IEEE Geosci. Remote Sens. Mag.*, vol. 9, no. 1, pp. 184–199, Mar. 2021, doi: [10.1109/MGRS.2020.2994107](https://doi.org/10.1109/MGRS.2020.2994107).

[10] ‘Android vs iOS Development: Which Should I Learn First?’ Accessed: Nov. 17, 2024. [Online]. Available: <https://www.upwork.com/resources/android-vs-ios-which-should-i-learn-first>

[11] X. Li, X. Zhang, X. Ren, M. Fritsche, J. Wickert, and H. Schuh, ‘Precise positioning with current multi-constellation Global Navigation Satellite Systems: GPS, GLONASS, Galileo and BeiDou’, *Sci Rep*, vol. 5, no. 1, p. 8328, Feb. 2015, doi: [10.1038/srep08328](https://doi.org/10.1038/srep08328).

[12] Z. Ozdemir and B. Tugrul, ‘Geofencing on the Real-Time GPS Tracking System and Improving GPS Accuracy with Moving Average, Kalman Filter and Logistic Regression Analysis’, in *2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, Ankara, Turkey: IEEE, Oct. 2019, pp. 1–6. doi: [10.1109/ISMSIT.2019.8932766](https://doi.org/10.1109/ISMSIT.2019.8932766).

[13] M. B. Garcia, ‘Location-based marketing using mobile geofencing: lessons learned from a user-centred application development research’, *International Journal of Technology Marketing*, vol. 17, no. 1, pp. 1–29, Jan. 2023, doi: [10.1504/IJTMKT.2023.127322](https://doi.org/10.1504/IJTMKT.2023.127322).

[14] ‘What is GDPR, the EU’s new data protection law?’, GDPR.eu. Accessed: Nov. 28, 2024. [Online]. Available: https://gdpr.eu/what-is-gdpr/

[15]

L. A. Tawalbeh, A. Basalamah, R. Mehmood, and H. Tawalbeh, ‘Greener and Smarter Phones for Future Cities: Characterizing the Impact of GPS Signal Strength on Power Consumption’, *IEEE Access*, vol. 4, pp. 858–868, 2016, doi: [10.1109/ACCESS.2016.2532745](https://doi.org/10.1109/ACCESS.2016.2532745).

[16] J. Sharkey, ‘Coding for Life--Battery Life, That Is’. http://files.blogjava.net/jicheng687/W\_0300\_CodingforLife-BatteryLifeThatIs.pdf

[17] ‘What is Agile? | Agile 101 | Agile Alliance’. Accessed: Nov. 26, 2024. [Online]. Available: <https://www.agilealliance.org/agile101/>

[18] ‘https://www.idpublications.org/wp-content/uploads/2015/05/Agile-Software-Development-Methodology.pdf’. Accessed: Nov. 26, 2024. [Online]. Available: <https://www.idpublications.org/wp-content/uploads/2015/05/Agile-Software-Development-Methodology.pdf>

[19] ‘https://www.researchgate.net/profile/Udesh-S-Senarath/publication/353324450\_Waterfall\_Methodology\_Prototyping\_and\_Agile\_Development/links/60f41f71fb568a7098b9d035/Waterfall-Methodology-Prototyping-and-Agile-Development.pdf’. Accessed: Nov. 26, 2024. [Online]. Available: <https://www.researchgate.net/profile/Udesh-S-Senarath/publication/353324450_Waterfall_Methodology_Prototyping_and_Agile_Development/links/60f41f71fb568a7098b9d035/Waterfall-Methodology-Prototyping-and-Agile-Development.pdf>

[20] A. T. de Aquino, J. A. L. Barboza Júnior, N. de Araújo Moreira, and P. Peixoto Praça, ‘Impacts of GPS module on energy consumption and machine-learning based battery lifetime estimation’, 2023, doi: [10.48545/ADVANCE2023-FULLPAPERS-3\_3](https://doi.org/10.48545/ADVANCE2023-FULLPAPERS-3_3).

[21] ‘Access location in the background | Sensors and location | Android Developers’. Accessed: Apr. 10, 2025. [Online]. Available: <https://developer.android.com/develop/sensors-and-location/location/background>

[22] Philipp Lackner, *How to Track Your Users Location in the Background in Android - Android Studio Tutorial*, (Sep. 28, 2022). Accessed: Apr. 14, 2025. [Online Video]. Available: <https://www.youtube.com/watch?v=Jj14sw4Yxk0>