

No part of the candidate's evidence in this exemplar material may be presented in an external assessment for the purpose of gaining an NZQA qualification or award.



Mana Tohu Mātauranga o Aotearoa
New Zealand Qualifications Authority

Scholarship Technology 2025

93601

EXEMPLAR

Top Scholar

PWM Moorings

Download: Available on The App Store and Google Play



Abstract:

This project is the result of the development of the PWM Mooring Application, a mobile application designed to help members of the Waikawa Boating Club in New Zealand manage mooring accessibility and availability. The app utilises real-time GPS tracking with an interactive map, enabling users to view the locations of moorings, flag those they are staying on, and stay safe and informed while on the water. Incorporating features such as location tracking, navigation, and an Automatic mooring check-in/out system, the app provides a reliable and easy-to-use tool for safe and efficient boating. The outcome is a functional, user-friendly platform that strengthens the club's ability to manage moorings and enhances the overall boating experience for its members.

Introduction:

The Waikawa Boating Club (Inc) is a prominent maritime incorporated society situated in the heart of Waikawa Marina, Picton. It serves as a central hub for boating enthusiasts, offering a variety of activities and facilities for both members and visitors. Recently, my family acquired a marine vessel equipped with sleeping quarters, offering the opportunity for extended voyages and comfortable stays on the water. Whilst exploring on the water in the Marlborough Sounds, we decided to stay the night away from the marina in a nearby bay. Talking to other boat owners, they recommended joining the Waikawa Boating Club as the club provides an extensive array of over 100 moorings dotted around the best anchorages throughout Queen Charlotte and Pelorus Sounds and D'Urville Island. This was a selling point to us to join the club, as it allows us to stay overnight on a safe and tested mooring, giving us a safe and peaceful overnight stay.

Identified Problem:

Currently, the club requires users to purchase an A3 paper copy of all the moorings and their details. This can be a painful task to navigate, as with over 100 moorings, it can be confusing to read. While exploring during the day in the far heads of the Marlborough Sounds with the thought of finding a club mooring nearby to take for the night, we came across some issues. With accounting for the wind, certain moorings became exposed, which causes chop and unpleasant sounds against the boat. This requires us to find a mooring that is sheltered from the wind and is suitable for our boat's size. Unfortunately, all the moorings were occupied. This meant we couldn't get a mooring, which cost us hundreds of dollars in fuel to have to return to the marina and travel in the dark late at night. This created stress and hassle as manoeuvring large vessels in the dark can be a painful task.



Existing club maps



Solution to the identified problem:

This sparked an idea, which is an app that firstly provides a digital version of the locations of each mooring with the information about each one. As well as finding a way to utilise users' GPS data to recognise free and occupied moorings. The creation of this app would allow other boat owners to be able to easily recognise free and occupied moorings, which can ultimately save fuel and lead to safe travel planning, and also have the ability to automatically detect which moorings are nearby, flagging them as occupied for others to know you are on it.

First contact with the WBC Club Commodore (Anne Marett)

This app showed true potential, solving issues many users may face in the Marlborough Sounds. I wanted to get as many stakeholders on board to test my idea. However, I realised that members would need a fully operational version via the App Store or TestFlight to test. This meant that I couldn't get real feedback until it was at a high enough level that the app store approves it as having location data, which can be tricky. Also, if users try out a clunky app, then they may get the impression that a high schooler has made an app that doesn't work and no longer continue to try to use it and future versions. Therefore, I decided to keep the stakeholders small until the version was capable of supporting a successful trial experience.



Hi [redacted] a free mooring at peak time is frustrating. As a club we are not in a position to sponsor you however if your work results in an app that provides this info, then we would happily consider making time for you. A potential issue may be cell phone coverage in some bays. A potential club benefit would be the ability to image non members on our moorings and those who damage them. All the best and please keep us in the loop on progress and any obstacles that we may be able to help with. Regards [redacted]

My first contact regarding this app was with the [REDACTED] Waikawa Boating Club (WBC). The WBC govern the other 2 clubs, the Mana Cruising Club and the Pelorus Boat Club. These 3 clubs share and operate the moorings, so I decided to email the [REDACTED] Waikawa, as they are the main body that oversees everyone. I sent an email to [REDACTED] with the idea of this application. She expressed that this would be a good idea to ensure non-club members do not use the club moorings, and it can provide a way to track the history of the moorings.

Preproduction:

With the problem identified, a solution in mind and the concept approved by the club, I began pre-production in the development of this app. This is the planning phase, which is crucial to lay out the requirements, timelines for the development to run seamlessly.

I broke the preproduction into a few phases.

1. Design mock-ups, identifying the needs for the project and finding ways I can address them
2. Learn and develop my skill set to ensure I don't waste any time during the development.

Predicted Project timeline

This timeline was used to help guide me and ensure I hit deadlines on time. With discussion with a teacher we came up with a rough timeline guide as I had never worked on a project this big, it also helped me try stay on target during the development process.

Date	Goal
Term 1, week 1	Begin preproduction phase of development
Term 1 week 2	Begin rough mocks and explore options to create this app
Term 1 week 3	Begin developing skills into app creation and development
Term 1 week 9	Finish preproduction and begin development of a minimal viable product
Term 2 week 10	Finish minimal viable product, carry out basic tests
Term 3 week 4	Refinement and patches. TestFlight for private testing
Term 3 week 5	Reach stable release with no breaking issues
Term 3 week 10	First real trial in Sounds

Project Management Setup

Common issues developers face when developing applications are that they don't get their app finished in time due to time constraints, deadlines and feature creep, causing the project to get out of control, making the project unmanageable. Therefore, implementing a way to manage current tasks to reach deadlines on time was necessary.

Trello

I used Trello on my project, a project management tool that allows you to break up a project into simple parts. The three default boards are To Do, Doing and Done. I created tasks and added them to the board. As I progressed on my project, I regularly looked at



this board, moving the completed tasks to the appropriate box. This kept me focused during the development process to ensure I avoided getting distracted by feature creep.

GitHub

GitHub is a crucial tool used by almost every software developer. It provides a range of functionality in project management, such as branching, commits, issues and project boards, that enable teams to collaborate and manage their code effectively. I used GitHub from the very beginning, saving all my work in a repository and committing regularly with helpful comments regarding issues and features. By the end of the project, I had over 150 commits and comments. It proved helpful during the development when I took a month break, which resulted in dependency issues and the project falling outdated. I tried upgrading the project; however, it seemed to have corrupted the project and ejected it from an Expo Go version. Thankfully, I had GitHub commits to revert back to. These allow me to revert to previous backups of my project in the event my computer breaks or unintended file deletion.

Word Document Journal

The other tool I used was a Word document. I essentially used it to journal my progress. In this document, I kept a rough timeline of when I added important things, dumped important links and docs regarding complex features such as YouTube videos and libraries. The Word doc became a useful tool to record useful information along the way.

Finalisation of main features in Minimum Viable Product (MVP)

When developing projects, I have found that I keep adding as much as I can into it all at once, resulting in it being too complex and hard to focus on one thing. Therefore, to combat this issue, I decided that before I started, I would work out what main features I needed to create and work on those one by one until I finished them. I ensured not to start other tasks whilst working on others to keep my logic and goals clear. This way, I avoid the development pitfall of feature creep and not getting the MVP (the essential functionality the app requires) completed. I decided to start the project with a clear MVP, and only when this was complete, start to add other extra features. I also wanted to ensure I allowed the project to be easily scalable, which will have an influence on my database schema and app design.

The main features in my app are based on the initial consultation with the WBC

1. A map displaying all moorings the club owns, with information about each one.
2. A way for the user to see if the moorings are occupied or vacant.
3. A way to use GPS data to see if a vessel is near the mooring.
4. A check-in/check-out system
5. A database that stores all the data and is scalable

Extra features if I have time at the end:

1. Filter system of moorings on the map
2. Background location
3. Push notifications
4. Solution to low cell service areas.
5. Redesign of UI

Market research on similar apps.

With the MVP laid out, I conducted some basic research to see if anything like this exists out there already. There was only one similar application called "The Marlborough Cruise Guide".

The Marlborough Cruise Guide

This app is an "interactive app and website showing marinas, anchorages, boat ramps, moorings, facilities, and other local information and images. Information is presented in map format, and includes satellite images and nautical charts."¹ This app has the option on the map view to filter club moorings. However, its function is limited to only being able to show the location; it does not show any information regarding the description and wind conditions. It also doesn't tell you about its occupancy status. It is also outdated, with new moorings not being on it.



*marlborough
Cruise Guide app*

Conclusion from market research

At the end of conducting my market research, I figured that there was no app that does what mine could do. This made me feel inspired and motivated to get started. The Marlborough Cruise Guide was an app packed with unnecessary features, made for ALL boaters in the Marlborough Sounds. No app on the App Store could inform users of mooring occupancy specifically, making mine a potential source of revenue in the future if other boating clubs with similar business models with club moorings get on board.

Conceptual Design Process

I had a clear vision of how I would like the app to look and what features it would have. It was now time to transform these features into a clear, creative reality. To do this, I created a conceptual design. This gave me the ability to create a basic plan that I can work towards in the early stages of development. This would save me time later on, so I could focus on the coding aspect rather than the design process.

Marvel mock-ups

Before I even thought about programming, I told myself I had to make some prototypes of the app, as it would help me initially to come up with designs early on. To begin, I used Marvel, the design platform, to create prototypes for websites and applications. It is just a drag-and-drop site, so you can get a rough idea of div tags, containers and layouts. I made a few versions of how it could look and landed on the ones to the right. It was nothing fancy, very boxy, but it gave me an idea of how it can look and feel. It consisted of a home screen, which displayed a small map in the middle for users to press to navigate to the map page, or by pressing the map in the nav bar. The 2 map page versions consisted of a Google Map layout with moorings represented by pin drops, with information about each mooring above. Also colour coded and with the number of vessels on each mooring. The image furthest on the right shows the expanded info about the mooring. The point of Marvel mock-ups was to quickly show how the app will look and how users will move through it without

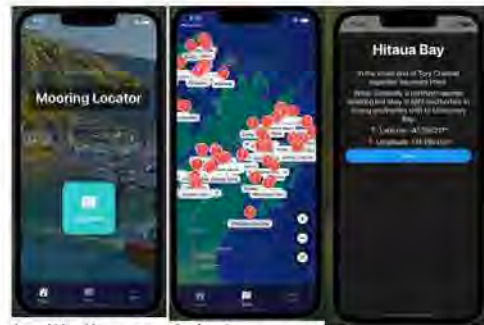


¹ <https://cruiseguide.co.nz/>

needing to write code. It let me gather my ideas and feedback early in the design process.

Xcode Prototype

Once the mock-ups were complete, I had a rough idea of how the app could look. I also wanted a basic prototype to help visualise the app. I watched a few YouTube videos on how to use Xcode and used AI prompts to make the prototype shown on the right. I sourced the mooring location data from the club's website, which contained all the moorings coordinates in the format of a GPX file, which I then managed to load all directly. I also played around with the modals to see what looked the best. The modal that takes up the full screen looks more aesthetically pleasing and more aligned with Apple's ecosystem and design interface docs. Overall, this prototype gave me a rough idea of how to lay out my application, possible areas to improve and the base to begin my backend coding for the location aspect.



Mobile App vs Website vs Pc Application

The whole idea, right from the start, was a mobile device application. This is for ease of use, as every boat owner has a mobile phone for safety on their vessel. It is simply not practical to have a PC on a boat. While some boat owners may carry laptops with them on their boat, it would be a hassle to use them while skippering. With the PC application removed from the picture, it only left the website and mobile apps. I began researching cross-platform development frameworks as well as website development options.

Possible Development Frameworks

When deciding on a possible app development framework, I had a few key considerations while narrowing down my choices. They were as follows:

- The framework must support cross-platform development. Only a single code base is needed for iOS and Android builds.
 - Saves time and makes it more accessible to everyone. Faster to develop
 - Rules out Apple's Xcode with Swift and Android Studio with Java
- The framework must be widely known and accessible. Must have a large number of libraries and documentation on how to use
 - This would make it easier to learn as well as use, having the resources at easy access. Pre-existing libraries are easier to implement, saving time instead of creating my own.
- The framework should be open source and free to use
 - This will ensure the cost of the development remains low
- The framework should be in a language that I am somewhat familiar with (HTML, CSS, Python, JavaScript)
- The framework needs to have direct access to location, notifications and maps
 - This eliminates websites, as handling location can be very tricky, and notifications are difficult to use on a website.

Based on these considerations, I eliminated many development platforms, ending up with the following: Xamarin, React Native, Flutter and PWAs. Database exploration also emerged during this process, but I did some research after I was positive of which framework I would be working with.

Xamarin vs React Native vs Flutter vs PWA

Comparing and contrasting these frameworks is straightforward. Many people have already compared these main frameworks, making this process easy. The main difference that sets these frameworks apart is the programming language. Working in a language that you are familiar with maximises potential and boosts productivity.

- Xamarin uses common .NET languages such as C# and F#, which can be also used to write platform-specific code. This allows developers to access native classes directly in .NET, making it more powerful for teams with strong .NET backgrounds.
- Flutter uses Dart, which was developed by Google. It is an object-oriented language like Java, but with a C-style syntax. It is known for building high-performance applications across multiple platforms. It is a less common language.
- React Native uses JavaScript, which is one of the most common programming languages. It enables developers to build cross-platform apps using native UI components. It has a large developer base and endless libraries, making it accessible and easy to learn. A few students at my school have used React Native for their mobile application framework and have had great outcomes. Testing can be straightforward using a development client like Expo, and just scanning a QR code runs the app without the need to create development builds every time.
- Progressive Web Apps (PWA) are basically websites with a few mobile features. It runs in the browser but can't be downloaded on the device. It lacks the native feel as it only has limited functions. It performs more slowly for complex tasks and cannot be submitted to an app store, therefore creating a lack of user trust and credibility. They are hard to get push notifications working on as well.

Based on these findings, I came to the final decision to use React Native. I downloaded Visual Studio Code and began playing around with React Native to begin learning.

Visual Studio Code

The development tool I used was Visual Studio Code because of my previous experience and the familiarity it provided. I installed multiple extensions to make the development easier. The first was React Native tools, which helped me write and test my React Native code, providing shortcuts and templates whilst programming. Expo tools allowed me to test my app on the fly without building a full native app. The bracket pair Colouriser helped me with lining up my brackets to ensure I remain in the relevant loops. These extensions made coding more efficient and reduced smaller mistakes, making the development faster over time. With integrated source control, I was able to commit and push my changes directly from the app as well as view all my files.

Learning React Native

I put a few weeks aside to learn how React Native works and to get properly set up, watching YouTube videos such as "the complete React Native course: from Zero to Hero²", a 7-hour tutorial that walked me through the basics and some more advanced features. Navigation, styles, maps, buttons, images and most components I have in the app have come from learning them from this video. During production, I would often turn to YouTube to master specific tasks, such as custom maps, markers and location permissions. Before I began working on the app, I played with React Native and followed some tutorials about states, as this came to be an essential part of the final product. I learnt how to set things in a state from a button or text input and then display it to the user. Once I was somewhat comfortable with it all, I began development.

² https://www.youtube.com/watch?v=ANdSdlgsEw&ab_channel=MashSchool

Development of Minimum Viable Product

Setting up Project File

To initialise my React Native project, I had the option of two development environments. The first one is Expo (via the Expo-Go runtime) and the other is React Native CLI.

- Expo Go provides a managed build runtime on the React Native SDK. It gives the ability to offer live reloads, which automatically reload the app on any connected device or simulator. It allows Expo to do all the native build processes, such as configuring Xcode and Android Studio, so I can develop and test instantly without having to compile separate binary files for iOS and Android. The only downside is that it reduces access to custom native modules and low-level APIs.
- The React Native CLI generates a fully "bare" project with separate iOS and Android folders. It allows more flexibility and control over the project but comes at a complex cost. This is called a development build and is intended for long-term production projects such as Instagram or Facebook. This would be a significant learning curve for me.

I decided to opt out of the full range of features and control by using Expo Go instead. This will be way more efficient to learn, which will speed the process up. While this process is not intended for long-term projects, the project can be later ejected into a bare workflow, transitioning it into the standard React Native CLI environment if necessary.

To begin and to install Expo Go, I followed their documentation found on their website.³ The first prerequisite was to install Node.js. This is a JavaScript run-time environment that allows JavaScript to be executed outside the web environment, so in my case, on a simulator or physical device. By installing Node JS, it also gives me access to NPM (Node Package Manager), which I can use to install and manage my JavaScript dependencies, which are libraries or packages that add functionality to the app, such as navigation UI components. With my Node.js and NPM configured, my development was now ready to create the project.

But before generating the project, I created an Expo Go account so my project could be linked to the cloud. I then generated my full project structure by running the command "npx create-expo-app PWMMoorings" This generated all my project files and necessary dependencies into the root of the folder. It creates a base app containing starter components and placeholder text, giving me a functional development environment to begin programming.

With setup complete, I downloaded the Expo Go client from the App Store on my iOS phone to enable live testing. This allowed me to run the app in real time on my phone whilst programming. Expo Go creates a development server on the local network and utilises a package called Metro that bundles the app and sends it to the device instantly. Metro watches for file changes and instantly bundles the app and sends it to the device.

Colour Palette and theme explained

For the app's colour palette and theme, I decided to choose a colour that reflects the ocean environment, seeing as the app is made to be used on the water. The primary colours are deep, dark blues. This dark navy blue is complemented by white and grey highlights, which provide contrast to improve



³ <https://docs.expo.dev/get-started/set-up-your-environment/?platform=ios&device=physical&mode=expo-go>

readability. Buttons and backgrounds should be a darker blue, and important statuses, such as occupancies, should be brighter colours like red or green to stand out against the bluish ocean theme.

The layout of the app should reflect the app's ecosystem to help blend in and be a seamless switch for users on Apple. I wanted the app to have rounded corners, small shadows and nice spacing to give it that modern feel. I aimed to make my app look as modern as possible, but also tried to reflect Apple's UI, which blends flawlessly with other family-friendly, high-quality Apple interfaces.

Home Page

The very first thing I began was the homepage. This was the page that showed up when users first opened the application. The landing page was first built in the `app.js` file. This is the default one-page file that is created when generating the project files. When it is first generated, it contains the basic template code.



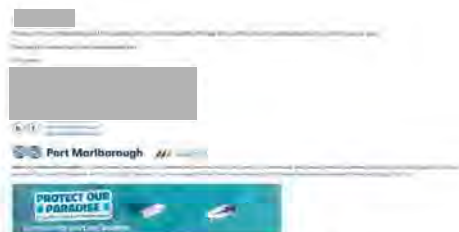
Default template

Intro to basic React Native concepts

Before getting into the specifics of my code, I should explain the core ideas behind React Native. React Native relies on the syntax called JSX, which blends JavaScript with a markup (like HTML). JSX controls how something appears on the screen, but also has the logic that drives it. In React Native, we use JSX to create components. These are functions that take in props (inputs) and return elements that end up being rendered on the screen. For example, on the home page, everything in the return parentheses is JSX because it defines the entire visual layout. For example, `<View>`, `<ImageBackground>` and `<Text>` are all written in this combined markup and logic syntax.

Home Page Development

To start building my own homepage, I used the `ImageBackground` component to set the background of my landing page⁴. The `ImageBackground` component works like the `<Image>` component but also allows you to add children inside it, which makes it possible to create layered layouts. I found an image of Picton Marina from the marina's website and placed it on the page. As I have taken this image from someone else's website, I contacted them and asked for permission, which they granted. With the background set as the image, I placed text over top using the `Text` component. The `<Text>` component allows text to be displayed and styled independently. It also supports nested styling and touch handling for interactive elements.



Email requesting image use rights



⁴ <https://reactnative.dev/docs/imagebackground>

Style Sheets

In React Native, stylesheets work a little differently from CSS⁵. To style a component, you normally wrap it in a `<view>` component, which acts like a `<div>` in HTML by acting as a container for other components. You can then attach a style prop to the view using `style={styles.container}`. All the styles are defined separately at the bottom of the file using `const styles = StyleSheet.create({ ... })`. This keeps everything organised and allows multiple components to reuse the same styles. Using `<View>` with stylesheets allows the structure of the code to be easily maintained, as well as to visually lay out the app's UI.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  image: {
    width: 100,
    height: 100,
  },
  overlay: {
    position: 'absolute',
    top: 0,
    left: 0,
    right: 0,
    bottom: 0,
    backgroundColor: 'transparent',
  },
  text: {
    color: 'white',
    fontSize: 16,
    lineHeight: 24,
    fontWeight: 'bold',
    textAlign: 'center',
    backgroundColor: 'transparent',
    opacity: 0.5,
    padding: 10,
  },
});
```

Bottom Nav Bar

With the landing page mostly completed, I decided to get started with the navigation functionality of the page. I began by researching options and found that the React Navigation library provides clear documentation and examples for implementing a bottom-tab navigator.⁶

I decided to choose this as it was easily accessible and didn't seem too complex. This nav bar lets you switch between screens by using routes that are lazily initialised and only mounted until they are first focused. This was useful for features I planned to add later, such as running distance calculations when a page is opened. This can increase performance overall on the site, leading to a better user experience.

After testing a basic setup of the containers and entry points I reorganised the navigation into a more structured `appNavigator` component. My `app.js` file imports and renders `<appNavigator>`. `App.js` is the very first file that runs when the app is opened, and by returning `<appNavigator>` it essentially gives all control over to the navigator, which then manages the screens and bottom bar navigation. The navbar provides the user with a consistent way to switch between screens as it remains the same on every page.

```
export default function App() {
  return <AppNavigator />; // Updated component name
}
```

Inside the `appNavigator.js`, I used `NavigationContainer` and `createBottomTabNavigator` to build the bottom navigation bar. The `appNav.js` file wraps everything and keeps track of which screen is currently active.

This single `tabs` array stores each tab's name, label, icon and component so it can be easily rendered. It also makes it easy to add or remove a tab by just adding it to the array. Name is the internal route name, eg `home`, label is the text under the icon, and icon is the icon name that is fetched from the installed library of icons. The component is the screen that shows when the tab is pressed.

Inside the `tab.navigator` the code loops over the `tabs` array and automatically creates a `tab.screen` for each tab. This means each object in the array defines a screen in the navbar with its label, icon and component.

```
// appNav.js
import { NavigationContainer } from '@react-navigation/native';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import { HomeScreen, SearchScreen, AddScreen, ProfileScreen } from './screens';

const tabs = [
  { name: 'home', label: 'Home', icon: 'home', component: HomeScreen },
  { name: 'search', label: 'Search', icon: 'search', component: SearchScreen },
  { name: 'add', label: 'Add', icon: 'add', component: AddScreen },
  { name: 'profile', label: 'Profile', icon: 'profile', component: ProfileScreen },
];

const Tab = createBottomTabNavigator();

export default function AppNavigator() {
  return (
    <NavigationContainer>
      <Tab.Navigator>
        {tabs.map(tab => (
          <Tab.Screen
            key={tab.name}
            label={tab.label}
            icon={tab.icon}
            component={tab.component}
          />
        ))}
      </Tab.Navigator>
    </NavigationContainer>
  );
}
```

The `screen options` property is used to style the navbar and control its behaviour. This was useful as it allows me to change what happens when a tab is pressed, for example it would change the colour of the icon and text to red and if not pressed to grey. Also, I can

⁵ <https://reactnative.dev/docs/stylesheet>

⁶ <https://reactnavigation.org/docs/bottom-tab-navigator/>

control whether I want a header shown at the top or not. The header can be customised, for example, by having a title at the top.

Each tab also displays the icons from the Ionicons library, which visually indicate which screen is active.⁷ The setup I have makes it easy for new tabs to be added, which allows me to easily expand my app in the future.



Home Page Update

With my navigation bar and home page mostly complete, I added a large button in the middle of the home page that takes the user to the map screen. This ensured that if people weren't familiar with the application, they had 2 ways of getting to the main screen, one being the nav bar and the other being the large anchor button. To achieve this, I first imported the anchor button image and saved it to my assets folder. This is where I store all my images. I import the anchor image using the `<Image>` component⁸. This component allows me to access network images, static images, temp local images and images from the user's local disk. In my case, I get the image from the local folder assets.

Next, I wrap the image in a `TouchableOpacity`⁹. This is a wrapper view that performs an action when touched. On pressing whatever is inside, it will decrease its opacity, showing it's been touched, and then on release, it restores to full opacity. This follows Nielsen's heuristic of Visibility of System Status, as when it is pressed, the opacity change shows the app has registered the touch. It also follows Consistency and Standards, as this is common on most apps. The `onPress` prop defines what happens when the button is pressed. I use `navigation.navigate('map')` to tell the app to switch to the map screen. I use this in an arrow function. An arrow function is a way to define a function in JS. If I didn't have an arrow function, the `onPress` prop would run immediately when the component renders. Using the arrow function, it only happens in response to the user's touch.

```
<View style={styles.imageContainer}>
  <TouchableOpacity onPress={() => navigation.navigate('Map')}>
    <Image source={require('../assets/anchor.png')} style={styles.imageStyle}
    />
  </TouchableOpacity>
</View>
```

Map Screen

With the nav bar and landing page are working, I began work on the map screen. At first, the screen showed up grey with no map. This was because I was forcing the map to use Google's provider, which required ejecting the project and adding native configuration depending on what device the user is on. Since I wanted to stay in the managed configuration, I reverted to my previous GitHub commit and found a new solution.



Native maps

To integrate a map feature that works across iOS and Android, I used the Expo Maps library. This library automatically utilises the native map of the user's device. Meaning

⁷ <https://reactnativeelements.com/docs/1.2.0/icon>

⁸ <https://reactnative.dev/docs/image>

⁹ <https://reactnative.dev/docs/touchableopacity>

Apple users see Apple Maps, and Android users see Google Maps. This ensures everyone is familiar with the platform they are accustomed to. Expo maps are up-to-date and designed specifically for the Expo ecosystem. This ensured testing runs smoothly and users have better performance. It is also simple to set up and is compatible with the current SDK. In my map screen, I started off by importing the components I need from the library.

import MapView, { Marker } from 'react-native-maps';

The MapView displays the interactive map, and the marker allows me to put pins on specific locations.



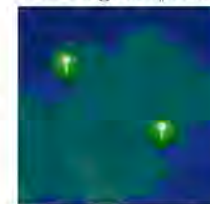
The map view component acts as the main container for displaying the map. I defined the initial region with latitude, longitude and delta values. I set the initial view to this initial region, so when the map loads in this is the location of the whole Marlborough Sounds. The latitudeDelta and longitudeDelta set the initial zoom of the map. Smaller values mean more zoom, whereas larger values mean a larger zoom. In my case, having 1 degree of latitude and longitude is equivalent to 111 kilometres from the centre point of the map. Styling is also applied to the map so it fills the whole screen or is in its defined layout.



Initial region defined

Markers

Within the map view, different elements can be rendered such as polygons, circles, overlays and markers. In my case, I decided to use markers to show moorings on the map. Each <marker> component requires a coordinates property to render the marker. This is an object and contains the latitude and longitude values that are the exact placement of the marker. By default, these markers look like pins on the map. The markers sit inside the map component.

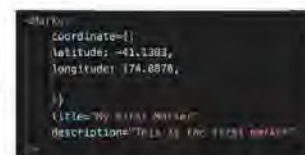


Example of green marker

Markers also have the ability to display metadata, which can help improve usability and display custom text. The title property defines a bold heading that appears when a user taps on the marker itself, inside the callout. Every React Native marker has a built-in callout by default. It has a bold title and a description underneath, all wrapped in a box. These can be customised with images and buttons. Having a callout makes it easy for users to see the information about the selected location, in my case, the moorings. My first map effort involved manually entered coordinates and details.



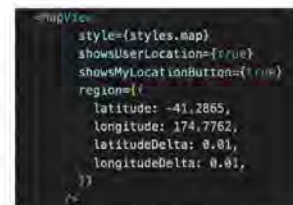
Default callout view



Code to display marker & callout

User's location

Now with my map and markers displayed, I needed a way to get the user's location and display it on the map. To do this, I tried adding 2 location props. Showuserlocation and setting it to true, as well as showsMyLocationButton. Nothing happened, and that was because the app could not automatically access location data without asking the user for permission. This was a critical step that I had missed. The props only tell the map to show the current position and will



only do this if access has been granted by the operating system. Since no permission was granted, the map renders but does not show the blue dot for the user's location. The setup is correct to show the users location; however it needs to have permission granted for it to do so.

Users' location permissions.

To get permissions to use the device's location, I had to use the expo-location permission package to request the permissions on startup. Without this step, the operating system blocks all requests to use location GPS data. I installed the library that provides access to reading the devices' geolocation info, current location or subscribing to location update events.¹⁰ By installing the library, it allowed me to import it at the top of my file using `import * as Location from 'expo-location'`; which then gave me a range of different methods to use the device's GPS data.

useState Intro

During this stage, I introduced my first use of states in the project. I did so by calling React's `useState`¹¹ hook. `useState` hooks are a way functional components can have their own local state, similar to variables in normal JavaScript. Every time a component re-renders, the data is lost, whereas a state variable preserves its data between renders. This solves a big issue of losing crucial data whenever the screen updates. The state is stored internally by React, so no matter how many times the component re-renders, the most recent value is always preserved.

useEffect

I also used my first `useEffect` hook. These allow functional components to run code every render. These are operations that interact with things beyond the internal system itself, like data fetching from databases. The way `useEffects` work is by taking two main arguments. The first is an effect function `{...}`, which is the code that you want to run, and the second is the dependency array `[]`. This array goes at the end of the effect and is what the effect depends on to run. If the array is empty, then it only runs once on mount; however, if the array contains variables, it will run any time those variables change. It tells React when the effect should run.

Asynchronous functions

Another key feature in the location permission code is the first use of asynchronous functions and the `await` syntax. Some operations take a significant amount of time to complete. Therefore, declaring a function as `async` allows me to use the `await` syntax when calling it to ensure the code pauses and waits for a result before continuing. Without declaring some functions as `async`, it would try to continue to run straight away, leading to errors.

Requesting Location Permission

With the main parts of my permission logic explained, I can now get into the code itself.

To start, I first imported the required libraries. React, React Native components for layout and styling, the React Native maps package for displaying maps, and the Expo Location package for accessing device GPS data. I then created a



```

import React, { useState } from 'react';
import { View, Text, Button } from 'react-native';
import MapView from 'react-native-maps';
import * as Location from 'expo-location';

const App = () => {
  const [location, setLocation] = useState(null);

  const requestLocationPermission = async () => {
    try {
      const { status } = await Location.requestPermissionsAsync();
      if (status !== 'denied') {
        const currentLocation = await Location.getCurrentPositionAsync({});
        setLocation(currentLocation);
      }
    } catch (error) {
      console.log(error);
    }
  };

  return (
    <View style={{ flex: 1, alignContent: 'center', justify-content: center; }}>
      <Text>Request Location Permission</Text>
      <Button title="Request Location" onPress={requestLocationPermission}>
        Request Location
      </Button>
      {location ? (
        <MapView style={{ width: 300, height: 150, }}>
          <MapView.Marker latitude={location.coords.latitude} longitude={location.coords.longitude}>
            My Location
          </MapView.Marker>
        </MapView>
      ) : null}
    </View>
  );
};

export default App;

```

¹⁰ <https://docs.expo.dev/versions/latest/sdk/location/>

¹¹ <https://react.dev/reference/react/useState>

state variable using the `useState` hook. For example, in the state I use for my location permissions:

```
const [location, setLocation] = useState(null);
```

Here, `location` holds the current value of the state, and `setLocation` is the setter function that updates it whenever the data changes. States can also take initial values when I first declare them. For instance, in the parentheses after `useState`, that value is called the initial value. This is what is set before `setLocation` has any data passed into it. This can be useful if you are using `true` and `false`. You can pass `true` as the initial value, and this will be preserved until the setter function updates it. In my state, I passed `null` as my initial value, as there is no data before location permissions are approved. Leaving it blank could cause some errors, so it's best practice to initialise the state with `null` or another default value.

To handle the permissions, I use the `useEffect` hook so that the logic runs as soon as the component mounts, as it has an empty dependency array `[]`. This ensures the location is asked straight away to avoid unwanted errors. If the user tries to use the map, then they won't be able to, as their location has not been requested. If I wanted to

make the location permission trigger on a button press, I could add that variable into the dependency array, which could cause it to then request permission. Inside my `useEffect` hook, I defined an asynchronous function called `getPermission`. This function first calls

`Location.requestForegroundPermissionsAsync()` to request the user's permission to access the device's GPS data. As I have imported the Expo location library

by calling that function, it takes care of the actual device requesting itself. If permission was not granted, then the function logs a message to the console and returns early.

There are two important properties that I use when requesting permission. That status property is a string and it indicates either "granted", "denied" or "undetermined". When the user presses allow, the status becomes granted, and then the granted Boolean is set to `True`. When the user presses deny, the status is denied, and then the granted Boolean will be `False`. In the `useEffect`, when the user denies the location, it logs it in the console and returns out of the function early. However, if they allow location permissions, then the function calls `Location.getCurrentPositionAsync()` to retrieve the current GPS coordinates of the user's device. Those coordinates are then stored in the location state using `setLocation(currentLocation)`;

The main reason `async` and `await` have been used is that requesting permissions and retrieving location data takes time, as the user doesn't initially allow them, so by awaiting the permissions, the functions can wait for the status, and it ensures the code doesn't continue before the data is ready. With the location permission functions complete, the app can now receive the phone's GPS data. This now lets the map display the user's location as the blue dot using `showsUserLocation={true}`.



Permission status that can be returned

When it comes to location tracking, there are typically 2 options you can choose from: `Foreground` and `Background`. To compare:

1. Foreground tracking:

This method gets location updates while the app is open and visible to the user. The app cannot retrieve location data while it is minimised or closed.

Pros:

- It is more battery-efficient. Retrieving GPS data can drain battery so while only retrieving that data while the app is open will reduce the battery consumption.
- It is simpler to implement as it requires less setup and maintenance.
- Fewer privacy concerns as it only tracks location while the app is open.

Cons:

- It cannot access location if the app is closed, therefore it makes it hard for apps that require constant location updates.

2. Background tracking:

This method tracks and receives the location even when the app is minimised or the device is locked.

Pros:

- It is great for navigation as it requires the location over time even while the app is not physically open.
- It can be used to trigger updates or notifications based on the user's location and without user interaction.

Cons:

- Background tracking can use a significantly larger amount of battery compared to foreground. This can be a bad thing, as if a user is out at sea and his phone dies, then that could be a safety risk.
- It is very hard to implement and configure, especially on iOS.
- There are numerous privacy concerns and restrictions when it comes to background tracking. Apple and Google are very strict about apps requesting background location, often leading to declined apps during the review process. It must be done 100% correctly or else the app will be denied.

I chose to go with foreground as it would be faster to implement and would also be a safer option as it reduces the need to keep charging the device, which can be especially tricky while out at sea. It will get published quicker from Apple and Android as it poses less of a security risk, and it will also keep the users happier, as they know they aren't being tracked all the time by a high school student.

Tracking live location.

With the permissions granted, I now had access to the device's GPS data. Since the device doesn't provide real-time GPS updates, I needed to create an async listener that continuously tracked the device's position. I created a state called location with its setter function being called setLocation. The initial value is null, as it will not contain any data until permissions are allowed. This state will store the location data once we obtain it.

```
const [location, setLocation] = useState(null);
```

With this state setup, I needed a way to pass the device's location into the state. Using a state makes sure the latest location data is saved during component re-renders, which will become essential for the map's functionality. To track continuously, I used the Location.watchPositionAsync() method from the Expo Location library¹². This function creates a subscription that watches out for changes in the device's location and triggers a callback whenever new data is available. In this listener, I configured these options:

- accuracy: Location.Accuracy.Highest. This ensures the best level of accuracy available on the device. It uses the device's wifi, GPS as well as cellular to obtain the best position possible.
- distanceInterval: 1. This threshold value will trigger a location update if the device moves more than 1 meter. This ensures the location updates while on the move.

```
const subscription = await Location.watchPositionAsync ({
  accuracy: Location.Accuracy.Highest,
  distanceInterval: 1,
  TimeInterval: 1000,
},
{loc} => {
  setLocation(loc); // Update the app's location state
});
```

¹² <https://docs.expo.dev/versions/latest/sdk/location>

- `timeInterval: 1000`. This value will trigger a location update every 1000 milliseconds (1 second), which also provides extremely accurate location data.

When the function is triggered, the callback receives a location object each time the device's location moves 1m or when one second has passed. This location object contains multiple pieces of GPS info.

- `coords.latitude` – the current latitude of the device
- `coords.longitude` – the current longitude of the device
- `coords.altitude` – the current altitude of the device
- `coords.accuracy` – the estimated accuracy of the gps (In meters)
- `coords.speed` – the current speed of the device in m/s^{-1}
- `timestamp` – the exact timestamp when the location was recorded in Unix timestamp format. Unix format counts the number of seconds since January 1st 1970, 00:00:00 UTC. Expo location, however, returns this number in milliseconds, making the number x1000 larger than the standard Unix time. Once converted, it showed the time, date, and time zone.

Example of a location object

```
LOG {"coords": {"accuracy": 5, "altitude": 0, "altitudeAccuracy": 1, "heading": 201.45, "latitude": 37.3351212, "longitude": -122.03256229, "speed": 3.64}, "timestamp": 1759399253687.949}
```

Inside the callback, I set the location data to the location state with the setter function `setLocation`, which ensured I had the most recent location. This allowed any component that uses the location state to automatically re-render whenever the position changes, which is a smooth approach to the user's experience.

Adding a New Page

Now I have the home page done, the map displaying with the user's location, and it makes sense to have all the pages I need in the nav bar. This will allow testers to experience how navigating

around the app feels. I added the remaining extra pages. Check in and menu. These were simple placeholder screens by creating new .js files for each page. I updated the `appNavigator` file to include them in the navigation tabs by importing them as new components and adding entries for each page in the tabs array.

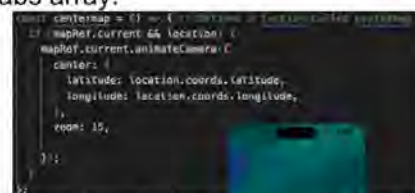


```
export default function TabNavigator() {
  return (
    <Tab.Navigator>
      <Tab.Screen name="Home" component={HomeScreen} />
      <Tab.Screen name="Map" component={MapScreen} />
      <Tab.Screen name="CheckIn" component={CheckInScreen} />
      <Tab.Screen name="Menu" component={MenuScreen} />
    </Tab.Navigator>
  );
}
```

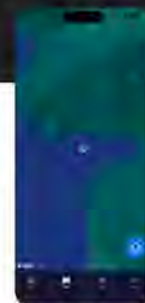
CentreMap Function.

Once I had access to the device's GPS data, I wanted to give my users a way to reposition on their vessel. On Apple Maps, there is no built-in centre location button; however, on Google Maps, this feature is built in. Users might be scrolling or panning away from their current position to check out available mooring, which may cause them to get disoriented. To solve this issue, I created a function called `centerMap`, and it animates the camera to focus on the device's current location.

I use `animateCamera` to smoothly move the camera to the vessel. I could just set the region directly; however, this will be an instant snap and will not look visually pleasing. Therefore, the animated camera looks more like a natural experience and is smoother. The



```
const centerMap = () => {
  mapbox.current.animateCamera({
    center: {
      latitude: location.coords.latitude,
      longitude: location.coords.longitude,
    },
    zoom: 15,
  });
}
```



Map with added center button

whole screen. The text title at the top is made bold to display the name of the mooring, and the description of the mooring is below. It is also slightly transparent, which makes it not stand out too much and be overly striking to the user's eye.

Another alternative, instead of using the markers' built-in callouts, I can use a modal component. The callouts are more restrictive, so a better option could be to go with a full-screen pop-up. I experimented with the modal popups and overlaid the modal on the map screen. It is essentially a page that pops up on the screen that I can customise fully to how I want it, without it having any restrictions. I can add buttons, images and other content.

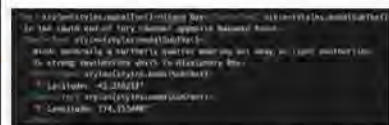


Comparison of modals and callouts

- Callouts are easy, very basic and good for small bits of information
- Models are more flexible, great for large amounts of information and great for user interactions and additional content.



Custom modal



Code for custom modal

First in-person meeting with [REDACTED] – 19th April

While in the Sounds for the Easter break, I presented [REDACTED] with the Xcode prototypes and my progress with the React Native build to get some insight from her and the club's perspective.

In this meeting, we discussed the progress of the application and potential changes and improvements. [REDACTED] said that it was looking really good so far and was happy with the progress. As the moorings are shared with all 3 clubs in the Marlborough Sounds. Pelorus, Mana and Waikawa, the names of the application should represent them. Therefore, the new name is Pelorus, Mana and Waikawa Club Moorings, to be shortened to PWM Club Moorings. We discussed the progress to date:
Currently built:

- Map interface with mooring pins (all to be added soon)

To build:

- Basic backend to store mooring and user data
- User check-in functionality
- Report page for issues, weather, FAQ, contact, and safety guidelines

Challenges:

- Spotty cell coverage: looking into caching/offline map support
- Confirming accurate mooring locations
- Some core features she expressed to me she wanted was the ability to go back in time to see who has checked in on what moorings. I said I could investigate this and get back to her. A way of doing this could be to store data in a DB and display it in another app or website.
- View mooring usage logs
- Manage reports/data

A challenge she faces is non-club members using the moorings. We discussed this and talked about potential ways to solve it. One was to ensure all members had the app, so non-members could be easily identified as they wouldn't appear in any logs or real-time tracking. However, this may be too hard to police, so we may scrap that idea.

I also presented her with the different options for displaying the information for each mooring. She chose the Modal version, which takes up the whole screen.



Marker Callout



Xcode based Modal

Another point was to have everything easily accessible and not too complex, as the club has members ranging in age from 20 to 90. This means the design must be accessible to all users, with big buttons, easy-to-read words, contrasting colours, etc. Pages and colours should be kept simple and basic.

On the menu page, she wanted to display the protocols for the club, such as how to approach moorings when people are already on them, so that everyone is on the same page and there are no misunderstandings.

There was a club dinner coming up where she would mention the app and let some club members know that it was in development and would be finished soon. This would be a good way to spread the word and get everyone excited about the upcoming launch. We can do a beta test so some select members can use it before pushing it out to everyone.

Club GPX Files

While exploring the club's website, I had come across a shared drive with every single club's mooring conveniently there; however, these were in a format .gpx. GPX files are designed to store GPS data, waypoints and routes. The GPX files can be imported into the boat's navigation system, which is very clunky and annoying to use. For our boat, it took me 4 hours to reflash the firmware and make it compatible. For other boat users without fancy navigation systems, this would not be possible. The GPX file contains the name of the mooring, the description, and the coordinates. I needed a way to scrape this data and import it into my map as individual markers.



GPX file format



Gpx files imported into the boats nav system



Database Selection

First, I needed a way to store this data. It wouldn't be logical to have all the markers manually written out as static code in the map file. With 95+ moorings, that would be extremely time-consuming, would create issues for long-term maintenance. If a new mooring was added, removed or updated, to keep records accurate, I would have to edit the code directly every time and then submit the file for review. This is not scalable or futureproofed. Therefore, I needed a database to store my data, from which I could then dynamically fetch the mooring info to display on the map. I began researching databases

that I could integrate into my app. I needed a system that allowed me to store data, including my lat, lng, name, description, etc., be easily able to update the data without having to recompile the app every time, and be able to fetch the data for dynamic markers.

There are two types of databases. Relational databases (SQL) and Non-relational (NoSQL) databases. Relational databases, such as MySQL, are where data is stored in rows and columns with predefined tables. It is often more fixed and structured and is best suited for applications with complex queries or relationships between different tables. As my mooring data didn't require cross-table relationships like many-to-many or one-to-many at this stage, a SQL system wouldn't add much value to the app itself.

The next thing I researched was local databases such as SQLite, which runs directly on the user's device. This could allow the app to read data as fast as it doesn't have much distance to travel from servers across the globe, and could work offline as well. However, the database would need to be built into the app itself, so anytime new moorings were added or updated, they would have to install a new version of the app. This is not scalable or convenient for club members, so I kept exploring my options.

Non-relational cloud databases are more flexible as they store unstructured data and are easily accessible for mobile applications. Some popular examples of these are Atlas MongoDB and Google's Firebase Firestore. These both allow data to be stored in documents, which fits naturally with my marker data, having one document per mooring in a collection of moorings. Each document could have its own fields for lat, lng, name and description. Because these are hosted on the cloud, they can make updates instantly on users' devices by just editing the database online, without the need to rebundle the app every time.

Between Firebase and MongoDB, Firebase stood out by a mile as it has extensive documentation linking it to a React application, as well as having prebuilt SDKs and offline caching. It also provides other services such as authentication.

As a result, I chose Google's Firebase Firestore, as it was most efficient for my app, scalable and a way to future-proof my project. It also provided offline support, which is extremely useful for the Sounds as mobile service is patchy, as well as other services like authentication, which I can use for user accounts.



Simple diagram of basic Firebase Firestore structure

Firebase has multiple paid plans that users can choose from: Spark, Blaze and Flame. I chose the Spark plan as this gives me 20k reads a day, which is how many times a document is returned in a query, plenty for the initial app. If I exceed 20k reads, the app would stop responding until the next day. These 20k reads are shared across all users, so if the app gains a larger user base, I would eventually need to upgrade the plan. Spark is suited for early development and deployment within reasonable test trial user numbers. However, if the app functions as intended and is released to the whole club and all of its members, then upgrading to a higher Firebase plan will be necessary to handle the increased usage.



Club GPX Files continued.

With my database chosen, I now needed to import my GPX files into a collection. As I have never used this before, I read most of the documentation specific to using Firebase in React¹³. I found out that you can also use it with Python, which I am extremely familiar with. I came up with the solution to create a Python script that parses GPX files and imports them into Firebase Firestore.

To begin, I needed a way to parse the GPX files themselves and save them into their own variables so I can use them to upload to Firebase. I scoured the Internet and found a GPX file Parser for Python¹⁴. This contained all the documentation to begin.

I needed a way to iterate through all the files in the folder, as each mooring is a separate GPX file. To do this, I defined a variable as my path to the folder of GPX files. By storing it as a variable, I can easily change the path later on without touching the main logic of the converter.

```
gpx_folder_path = '/Users/user-Stac/ClubMooring-Repo/GPXPoints'
```

Next I used `os.listdir()` function, imported by `os`¹⁵. This returns a list of all the contents of the specified path as a list of strings, in my case all the GPX files.

```
for filename in os.listdir(gpx_folder_path):
    print(f'processing file: {filename}')
```

I combined this with a for loop to iterate through all the files one by one¹⁶.

The next step was to parse the GPX into a variable to use for my Firebase collection.

I updated the script as I realised I need the full path of each file to parse the GPX data correctly. To obtain the full file path, I added an `os.path.join()` function¹⁷. This handles platform-specific cases, such as `\` or `/`, when handling the files. The function now gets the name of the file and adds it to the end of the full folder path.

```
/Users/user-Stac/ClubMooring-Repo/GPXPoints/mooring1.gpx
```



Format update

Next I check if they are `.gpx` files. I do this by having an if statement to check if the file contains `.gpx`¹⁸. If it does, it then moves to the next block of reading the file using the `gpxpy` library. The `'r'` specifies what action I want to do to the file. In my case, I'm only reading it.

```
(file_path, 'r')
```

I now extract the data from the GPX file and save it to separate variables. I use a for loop to iterate through all the waypoints in the GPX file. In my GPX files, I only have one waypoint being the location of the mooring, but for other cases, you can have multiple, which draw a route. The loop pulls out the latitude, longitude, name and description. With the files now successfully parsed and covered to individual variables, I can now handle them accordingly to send to Firebase.

To begin the process of handling them in Firebase, I need a way for the Python script to communicate with the Firebase Cloud Server. This is achieved by using Firebase credentials that point to my account and database. This is a separate confidential file I import at the top of my Python script. If this fell into the wrong hands, they could change anything in my database. I then initialise my Firebase client so it can make read and write changes. `Db` is my database handle so I can perform operations like `.set` or `.add`

```
data = {
    'name': name,
    'latitude': latitudes,
    'longitude': longitudes,
    'description': description,
    'occupied': False
}
```

¹³ <https://mlfirebase.io/>

¹⁴ <https://pypi.org/project/gpxpy/>

¹⁵ <https://www.geeksforgeeks.org/python-os-listdir-method/>

¹⁶ <https://www.geeksforgeeks.org/python-loop-through-folders-and-files-in-directory/>

¹⁷ <https://www.geeksforgeeks.org/python-os-path-join-method/>

¹⁸ https://www.w3schools.com/python/ref_string_endswith.asp

documentation, I would need to select the web modular API, as it was recommended for modern Firebase projects.

To do this, I created an asynchronous function called `querycollection` and referenced this to the "mooring" collection in the database

```
const MooringRef = collection(db, "moorings");
const q = query(MooringRef, where("name", "==", "Alligator North"));
```

I then constructed a simple query that utilises `query` and `where`, which specifies I only want a document that has the `name` field equal to "Alligator North".

```
const moorings = await getDocs(q)
```

To execute the query I use `await getDocs(q)`, which returns all the matching documents.

```
moorings.forEach(mooring => {
  console.log(mooring.data());
});
```

I then loop through all the results using `forEach` and log the results to the console. I knew that I would only receive one result from the query, so it wasn't necessary to use `for each` but later on, when I start receiving 20+ results, I need to use `foreach` to iterate through each document dynamically. This is a crucial step for displaying all the mooring data on the map later on.

The output to the query is: *{ "description": "One of two moorings in the first bay south of Alligator Head, on northeast side of Hikoekoea Bay, Guards Bay. Wind conditions: Sheltered from east through to north to southwest winds. Go to Punt Rails if southerlies get strong.", "latitude": -41.99723333, "longitude": 174.1535333, "name": "Alligator North", "occupied": false }*

GitHub issues.

After I had got this working, I took a month-long break where I didn't touch the project at all. During this time, many of my dependencies and SDKs had become outdated. When I came back and attempted to update so the app could run again, something went wrong, and the project was ejected from the workflow, creating a bare project with individual iOS and Android files. Fortunately, I was able to revert to my previous GitHub commit and recover the project.

Map Markers continued

After I had recovered from that setback, I worked on handling the map markers to try to dynamically iterate through all the documents in the database. I first needed a live listener, similar to the query I did before, but this time it watches out for changes anywhere in the collection and triggers the query again. This is important because mooring information, such as occupancy status, can change at any moment, so each mooring needs to reflect this live.

To achieve this, Firestore provides a `onSnapshot` function. This is different to `getDocs()`, which only receives data once when called. `onSnapshot` sets up a listener that, whenever a document in the collection is added, updated or removed, the listener is triggered and the app receives the latest data automatically. In my app, the listener would watch for any changes in the mooring collection.

```
const mooringsRef = collection(db, "moorings");
const unsubscribe = onSnapshot(mooringsRef, (snapshot) => {
  const mooringList = [];
  snapshot.forEach((doc) => {
    mooringList.push(doc.data().name); // or doc.data().name
  });
  console.log("Current moorings: ", mooringList);
});
```

This establishes a real-time link to the Firestore. Anytime a mooring is updated, such as occupancy status, the listener fetches the new data instantly. It does an initial grab of all data, so every time this runs on the mount, it fetches all the moorings, then when a single

document changes, instead of fetching all the documents again, it only fetches the document that contains the change. This was useful as it keeps document reads low, as they have costs associated. This listener will make all the mooring data up to date when displayed on the map without manually refreshing.

Once the real-time listener was established, the next step was to put this data into a `useState` hook. This will store all the mooring objects that the listener fetches.

```
const [moorings, setMoorings] = useState([])
```

Each object in this array would contain all the mooring names as I am only fetching those from each document. Because they are stored in a state, React can automatically re-render the map whenever the data changes.

Inside the listener, the data is pushed into the

temporary array. When all the data has been retrieved from the snapshot, the temporary array is pushed into the main state array of the app using the `setMoorings` Function.

```
const tempMoorings = [];
snapshot.forEach((doc) => {
  mooringData.push(doc.data().name); // or doc.data().name
});
setMoorings(mooringsData);
```

With this complete the console now logs all the mooring names dynamically. Now that I was confident of this working, I moved to the next stage, which is getting all the data so I can use it on the moorings. To now pull all the data from the documents, instead of calling

```
mooringData.push(doc.data().name);
```

I replaced that with:

```
mooringData.push({
  id: doc.id,
  name: doc.data().name,
  description: doc.data().description,
  latitude: doc.data().latitude,
  longitude: doc.data().longitude,
  occupied: doc.data().occupied,
});
```

This then allowed me to pull the lat and lng as well as the other information directly from the state of each mooring. I could now dynamically create `<marker>` components for every mooring using the state data. To dynamically do this, I map over the moorings array

state. I use this `.map()` array method to go through every element in the moorings array and transform it into items. Inside the markers JSX, I can now use the item data to create JSX components. Inside the JSX, I can access the items' data by `item.latitude`, `item.longitude`, `item.name` and `item.occupied`. This will generate each marker dynamically on the map. This approach makes sure that whenever a mooring is updated or changed in the database, the map renders with the latest data automatically.

```
{moorings.map((item) => {
  return (
    <marker
      key={item.id}
      coordinates={[
        latitude: item.latitude,
        longitude: item.longitude,
      ]}
      pinColor={pinColor}
      title={item.name}
    />
  );
})}
```

First problematic issue

Theoretically, this should have worked. However, it did not. None of the markers were displayed. This really stumped me, and this issue took ages to problem-solve. When I updated a single mooring then that one mooring would then appear only. This was extremely confusing, and I thought it must be an issue with how I have set up the listener or the query. After following the code and using debugging methods of console logging every step, I found that when the component mounts, the use effect runs and sets up the snapshot listener. But because the listener is asynchronous and takes a few seconds to fetch all the mooring data, it doesn't populate the mooring state instantly. The moorings are an empty array, so when the function tries to `.map()` over the moorings state, nothing happens, so nothing appears. When a mooring is then updated in the Firebase, it retriggers the function and only updates the single mooring. To

overcome this issue, I added a loading screen that would stay up until the mooring state is populated.

To add the loading page, I created another state called loading/setloading. This state was in charge of controlling the loading pages' visibility.

```
const [loading, setLoading] = useState(true);
```

The initial value is set as true, as it ensures the map doesn't render until I have set it to false. The loading screen pauses the map render until all the data is ready. The useEffect runs, and the listener populates the data.

```
setMoorings(mooringsData);
```

```
setLoading(false);
```

After this, the loading is set to false and triggers a re-render. By this stage, the moorings state is populated, so. map ()

can iterate over the items. After implementing the loading screen solution, the issue was resolved. The markers now display correctly and are changed in real time to mooring updates. This taught me a valuable lesson to handle async data carefully.

```
if (loading) {
  return (
    <View style={{ flex: 1, justifyContent: "center", alignItems: "center" }}>
      <ActivityIndicator size="large" color="#0000ff"/>
    </View>
  );
}
```

Marker Update

With all the markers now displayed, I wanted to make the colour of the pin itself represent the occupancy status. This will help users identify occupied moorings more efficiently. To achieve this, all I needed to do was add an if-else statement to my mooring item logic. For every mooring in the mooring state, it checks the occupied property to decide what colour the map marker should be. If a mooring is occupied, then it sets the colour to red, and if it's unoccupied, it sets the colour to green. I used these colours as they are the most common universal indicators for stop/go, making them easy to interpret.

It stores this information in the pinColour, which is then assigned to the marker's properties so it displays every render or automatically when the status changes in real time.



```
let pinColor;
if (item.occupied === true) {
  pinColor = "red";
} else {
  pinColor = "green";
}
```

```
return (
  <Marker
    key={item.id}
    coordinate={{
      latitude: item.latitude,
      longitude: item.longitude,
    }}
    pinColor={pinColor}
    title={item.name}
  />
)
```

Modal Update

With the markers now set up, it was time to make them interactive, as requested by the club commodore. To begin, I added the onPress handler to the map marker, which triggers a function whenever the marker is pressed. This allows me to open the modal and show the specific information about the selected mooring.

```
onPress={() => {
  setSelectedMooring(item); // Save the selected mooring to state
  setModalVisible(true); // Show the modal
}}
```

To achieve this, I created two new states. These were used to

```
const [modalVisible, setModalVisible] = useState(false);
const [selectedMooring, setSelectedMooring] = useState(null);
```

control what was displayed inside the modal and the visibility of the modal. The modalVisible state controls whether the modal is open or closed, and selected Mooring stores the data of the mooring being pressed.

When a user taps on a marker, the onPress handler updates the states and saves the mooring information to selectedMooring, as well as sets the modalVisible to true. As I will pass the stored selected mooring data into the modal from the state, I can dynamically display details about the chosen mooring. This is a way to reduce the number of modals. I could have 95+ modals, one for each individual mooring; however, I will implement a single modal that updates with mooring information each time the marker is pressed.

This setup provides users with a more interactive experience by allowing them to access mooring information directly from the map, without lag or interference. In the modal component, I use the same layout as the previous static one, but change the internals JSX

For the content, I displayed the mooring name and description using `selectedMooring.name` and `selectedMooring.description`. I can call these properties because the full mooring object is stored inside the `selectedMooring` state. It saves it as an object in the `selectedMooring` use state, meaning I can directly access any of its properties whenever I need them. I dynamically pull the data from the state and insert those values into JSX, which gets rendered as part of the modal's UI.

Modal Issue #1

This, however, did not work and I received multiple `TypeError`s. "Cannot read property 'name' of null", "Cannot read property 'description' of null", and the same errors for latitude and longitude. I figured out that these occurred because when the modal tried to render the selected state, it was still null. Since null has no properties, it would throw off an error as it was trying to access for example `selectedMooring.name`; however this was set to null originally.

To solve this issue I created a conditional rendering block for the mooring content. I stored the content in a separate variable called `modalContent`. This would only render if there was an object in the `selectedMooring` State. When a marker is pressed, the modal renders the details; however, if `selectedMooring` is null due to not having a selected marker at that time, then there is a fallback message informing that no mooring is selected. This message should never be displayed as the modal is only displayed when a marker is pressed, but for development, it is there to prevent unwanted error messages. The `else` statement prevents the app from trying to access relevant mooring properties of null, which caused the errors from earlier. I also added a timeout before clearing the `selectedMooring` state as it prevents the modal to clear too early while it closes.

Modal Issue #2

Initially the modal was rendered as a child of the `mapView`. This means the `<modal>` component was placed inside the `<mapView>` Tags like the following: →

```
<MapView>
  {moorings.map(item) => (
    <Modal visible={modalVisible}>
      {/* Modal Content */}
    </Modal>
  )}
</MapView>
```

These subtle things ended up causing critical errors, as when the modal was closed, its invisible container would still remain present and mounted, blocking interactions with the map. This resulted in users not being able to pan or scroll on the map properly. After isolating this issue, I moved the modal content outside the `mapView` component so it would not be rendered as a child anymore. Lifting it to become a sibling instead ensures the modal only mounts when it is needed and doesn't interfere with any of the map's

```
const [modalVisible, setModalVisible] = useState(false);
const [selectedMooring, setSelectedMooring] = useState(null);

const handleMarkerPress = (mooring) => {
  setModalVisible(true);
  setSelectedMooring(mooring);
};

const handleCloseModal = () => {
  setModalVisible(false);
};

const handleClearMooring = () => {
  setSelectedMooring(null);
};

const modalContent = (
  <div>
    <h3>{selectedMooring.name}</h3>
    <p>{selectedMooring.description}</p>
    <p>Latitude: {selectedMooring.latitude}</p>
    <p>Longitude: {selectedMooring.longitude}</p>
  </div>
);

return (
  <div>
    <MapView>
      {moorings.map(item) => (
        <Marker key={item.id} position={item.location} title={item.name}>
          {item.description}
        </Marker>
      )}
    </MapView>
    <Modal
      visible={modalVisible}
      onClose={handleCloseModal}
      onClear={handleClearMooring}
    >
      {modalContent}
    </Modal>
  </div>
);
```

```
setModalVisible(false);
setTimeout(() => setSelectedMooring(null), 300);
```

```
const modalContent = (
  <div>
    <h3>{selectedMooring.name}</h3>
    <p>{selectedMooring.description}</p>
    <p>Latitude: {selectedMooring.latitude}</p>
    <p>Longitude: {selectedMooring.longitude}</p>
  </div>
);
```

touches. After this was implemented, the module now mounts and unmounts cleanly without leaving an invisible barrier.

Modal UI Update

I decided that showing the occupancy status in the modal would improve usability. This was a simple add-on. I created an if-else statement that checks the occupied property of the selectedMooring state. If the occupied property is true, then I set occupiedText to a red "occupied" and if it is false, then I set the occupiedText to a green "unoccupied". This occupied Text JSX element is then inserted into the modal's main content, next to the mooring name. When the modal renders, the user can see the name with the occupancy status in the universal colour of occupied for red and unoccupied for green.

```
let occupiedText
if (selectedMooring.occupied === true) {
  occupiedText = <Text style={{ color: "red", fontWeight: "bold" }}>Occupied</Text>
} else {
  occupiedText = <Text style={{ color: "green", fontWeight: "bold" }}>Available</Text>
}
```

More Pages

Next I created pages for safety, about, contact, FAQ and terms and conditions. They won't be displayed in the nav menu at the bottom, they will be displayed on the More page, where users can click on one of interest to view. However, to allow users to navigate to each page from the More page, I created a separate stack navigator called MenuStack using @react-navigation/native-stack. This stack contained the menu screen (more page) and other screens such as About and Contact.

```
import { NavigatorContainer } from '@react-navigation/native-stack';
import { createStackNavigator } from '@react-navigation/stack';
import MenuScreen from './MenuScreen';
import Safety from './Safety';
import About from './About';
import Contact from './Contact';
import FAQ from './FAQ';
import Terms from './Terms';

const Stack = createStackNavigator();
export default function MenuStack() {
  return (
    <Stack.Navigator>
      <Stack.Screen name="Safety" component={Safety}/>
      <Stack.Screen name="FAQ" component={FAQ}/>
      <Stack.Screen name="Contact" component={Contact}/>
      <Stack.Screen name="About" component={About}/>
      <Stack.Screen name="Terms" component={Terms}/>
    </Stack.Navigator>
  );
}
```

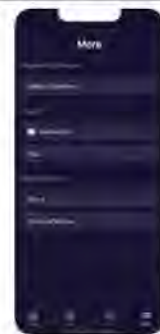
In the menu screen, I displayed each section with an appropriate title, such as important information. Each section contained relevant pages, made from touchable opacity to turn them into buttons. Icons and descriptive text were also added in conjunction to create a visually striking and aesthetic menu page. When the safety button is pressed, it uses the on-press handle to call

```
navigation.navigate('Safety')
```

which moves the current screen to the safety screen. This setup allowed users to click on any page listed on the menu screen and navigate accordingly. The reason I used a separate MenuStack from my main page navigation is that I want to keep the menu pages grouped all together. This allowed me to manage and keep the structure organised, which ultimately makes it easier to maintain.

The UI of the menu page is made to look visually pleasing to the eye and not be too overcrowded. I settled on a dark navy background, which matches my existing theme for the app. Each page contained a placeholder with no back functionality for now.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 10,
    backgroundColor: '#000080',
    color: 'white',
  },
  title: {
    font-size: 24,
    font-weight: 'bold',
    marginBottom: 20,
  },
  list: {
    display: flex,
    flex-direction: row,
    flex-wrap: wrap,
  },
  item: {
    width: 50%,
    padding: 10,
    margin: 5,
    border: 1px solid white,
    border-radius: 10,
    background-color: 'white',
    color: 'black',
  },
});
```



Login Page

With all of the pages intended for the app added with content or place holders, I decided I needed to work on the security of the app. I needed a way to stop members of the

public from accessing my app unless they were a part of the boating club. To do this, I created a simple logic page with Firebase authentication²².

This login screen will be my first point of interaction for my users and will be designed to allow signing in to club accounts. I created a new login screen file and set up my new states, email and password. These states will store the user's input from the corresponding fields.

```
const [email, setEmail] = useState('')
const [password, setPassword] = useState('')
```

I use a text input component to control user inputs, which allows me to directly tie the values to the use state as I use the onChangeText handler. When a user starts typing their email or password, every time text changes, it changes the data that's in the states in real time. This ensures the app always has the latest input values, so I can handle it appropriately in the next step.

```
<TextInput
  placeholder='Email'
  value={email}
  onChangeText={text => setEmail(text)}
  style={styles.input}
/>
```

I don't need to handle signup logic, only login logic, as user accounts will be transferred from their club memberships. This prevents unwanted users from logging into the app and using the club features when they are not a part of the club. A similar format is used to handle logging in. Taking the user input from the state, however, this time using the signInWithEmailAndPassword method instead. When the <TouchableOpacity> signup and login buttons are pressed, it triggers the function to run. If the credentials are correct, Firebase returns a user credential Object. This user object contains important information such as their email address and a UID, or unique Identifier number. These are logged to the console for debugging so I can ensure the signing is successful without errors. This UID comes to be very useful as it can track individual users in the app, linking their actions from the database to their email. Firebase uses encrypted and hashed data to ensure security over all transfers of user credentials. Passwords are never stored directly in the app or in the database; they are managed by Firebase's auth backend. It ensures everyone's data remains protected against unauthorised access.

After a successful login, the user is automatically sent to the home screen of the app. This is done by using React navigation with the login feature. Once Firebase confirms the username and password are correct, the signInWithEmailAndPassword returns a user object that has the UID and email. Now I can call the navigation.navigate('Home') method to redirect the user from the login page to the home page.

If the login fails, from either not having an account or getting the credentials wrong, the .catch block will capture the error and log it to the console for debugging purposes.



```
const unsubscribe = auth.onAuthStateChanged(async () => {
  if (user) {
    navigation.replace('Home')
  }
  return unsubscribe
})
```

²² https://www.youtube.com/watch?v=ql4J6SpLXZA&ab_channel=MadeWithMau

I wrapped everything, the entire login screen, in a `KeyboardAvoidingView`, as when I tested this login page out on a physical device, I noticed that the keyboard would cover the login and register buttons. To overcome this, I added a keyboard-avoiding view, which automatically adjusts the interface when the keyboard appears, making sure all the input fields are visible. I also added a `TouchableWithoutFeedback` component which works by when a user taps outside the input fields, it dismisses the keyboard, allowing them to interact with the buttons without obstruction. This approach improves the usability of the login page to ensure a smooth experience for the user.

```
const handleLogin = () => {
  signInWithEmailAndPassword(auth, email, password)
    .then((userCredential) => {
      // Signed in
      const user = userCredential.user;
      console.log('User logged in successfully', user);
      // Redirect to the main screen
      navigation.replace('Main');
    })
    .catch((error) => {
      const errorCode = error.code;
      const errorMessage = error.message;
      // Show an error message to the user
      console.log('Error logging in', errorCode, errorMessage);
    });
}
```

Persistence is a big factor that needs to be considered, as it improves the user experience drastically. When a user closes the app and reopens it, the app needs to remember who logged in. By using Firebase's `onAuthStateChanged` inside a `useEffect`, I can create a listener to detect when a user is already logged in when the app launches. If the user is already authenticated, then the `navigation.replace('Main')` is called and redirects the user to the main screen, without having to re-enter their credentials. This prevents unnecessary logins and makes the app seem more professional and seamless. The way the app automatically persists the user's login state is through tokens. When a user logs into the app for the first time, it stores a secret token and a refresh token in the device's local storage. These are used to maintain the user's session, so if the app is closed or the device restarts, Firebase can check for the stored token and identify that they have already logged in.

Club update

On the 22nd of August, I reached out to my main contact at the club [REDACTED] to explain my progress and organise a meeting to showcase the app.

I received a reply informing me she had stepped down from her role, and a new club member had taken her place. This was a little concerning as I thought this could be the project over if they weren't keen for me to continue. Fortunately, they were extremely keen and wanted to help as much as they could. The new [REDACTED] and he became my new primary contact at the club.

On 24 August, I had a Zoom meeting with [REDACTED] from Waikawa Boating Club.

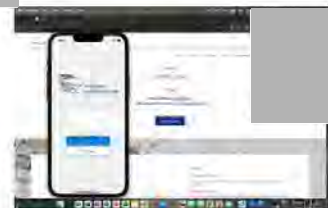
The purpose of the meeting was to introduce the app, explain its functionality, and discuss how it will be implemented and used by club members. [REDACTED] was very keen on the app and its potential benefits. We discussed its purpose, features, and the overall workflow for users.

A significant part of the discussion focused on login and user authentication. We explored how members will log in securely and how to prevent unauthorised access. [REDACTED] agreed to provide a sheet containing all member details, including names, emails, and membership numbers. This information will be uploaded to Firebase, which will allow accounts to be set up automatically for members.

I have just been asked to build a new app for the Waikawa Boating Club. The app will be used by club members to manage their accounts, book boats, and view club news. I have already built a prototype app and shown it to the club members. They are very keen on the app and its potential benefits. We discussed its purpose, features, and the overall workflow for users. I have also been asked to provide a sheet containing all member details, including names, emails, and membership numbers. This information will be uploaded to Firebase, which will allow accounts to be set up automatically for members.

Sorry I am away on boat in Gulf Harbour until November and exchange the [REDACTED] email is the new committee and [REDACTED] is in charge of meetings.

[REDACTED] getting you ticked up of this and happy to discuss to give you the background about this great idea and his project



Branding and design were also covered during the meeting. [REDACTED] confirmed that the app name PWM Moorings is suitable. We discussed adding the remaining club logos to the login page; [REDACTED] is sourcing one logo that I was unable to locate previously. [REDACTED] the [REDACTED] would provide the remaining login information and would also be CC'd into relevant emails to ensure smooth communication.

Promotion of the app to members would primarily be through the club magazine and newsletter, which [REDACTED] manages. This would include instructions for downloading the app and a feedback form for users. We agreed that the app testing and trial period should run throughout September and the first few weeks of October, allowing members sufficient time to use the app and provide feedback.

The new [REDACTED] also reached out and expressed his excitement for the project. He wanted to help make sure all necessary action was taken, so he wanted to be cc'd into all future correspondence.

A few days later, I received a response from [REDACTED]. He had talked with his team and had concluded that the user data should be obtained via volunteers instead of the club providing it. They would help by sending my form to the club's weekly newsletter. After a few weeks, I had 18 volunteer subjects willing to test the app.



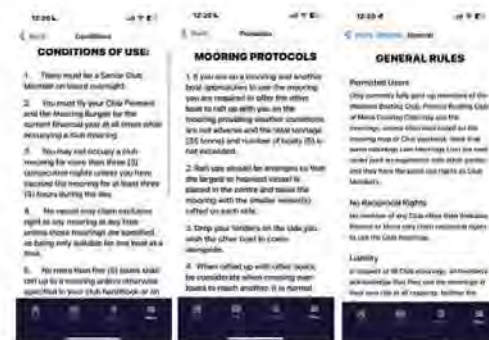
The signup form gathered relevant info regarding their name, email, and vessel device type. With it being close to a 50/50 split of Android to iOS users.

Now that I had a user base, the pressure was on to finalise the app to get this out to them.



Menu Pages continued.

I began working on the general rules page in the menu. This allowed users to view the club rules for mooring use without having to navigate the website, which could become a tedious process. Having all the club rules in one place boosts efficiency and time. The screen would present all the important rules in a structured and readable layout. I decided instead of having all the rules on the page fixed, I would utilise a scrollable view. This is a way to fit more info into the page without having it affect the size and layout. Users will be able to scroll through longer sections of text, making sure all content remains accessible even on smaller



screens. In the scroll View component, I used multiple text components to display these rules, separating them into different sections, with headings. The subheadings and headings were styled to be larger and bolder to help users identify each section. I repeated this format for my other pages with some minor style edits to reflect what is best for the designated page. Conditions and Protocols.

Check In page functionality

For the manual check-in page, I wanted to have a dropdown button with all the moorings, which would make the app more user-friendly. To achieve this, I conducted some research and came across a library called `react-native-dropdown-select-list`²³. I discovered that I would be able to dynamically display all the mooring names from my database, because all the dropdown needed was a simple array of data. To build this, I set up two `useState` hook variables. One to store the currently selected mooring and the other to store the full list of moorings fetched from the database.

```
const [selected, setSelected] = useState("");
const [moorings, setMoorings] = useState([]);
```

I copied over the mooring listener from the map screen as that returns a variable of all the moorings, which I need for the dropdown. I altered the listener to only fetch the mooring name, ID and occupancy status, which adds this to an array passed to the moorings state.

```
useEffect(() => {
  const q = queryCollection('moorings')
  unsubscribe = unsubscribe ? (unsubscribe) : () => {}
  queryCollection('moorings').subscribe((data) => {
    setMoorings(data)
  })
}, [])
```

The `selectList` component from the dropdown library takes this array as the input using the `data` prop. This can either be an array or an array[object]. In my case, each mooring is an object, so this will work. The data will be iterated as options in select lists.

```
const [selected, setSelected] = useState("")
const [moorings, setMoorings] = useState([])

const [value, setValue] = useState("")
const [boxStyle, setBoxStyle] = useState({})
const [dropdownStyle, setDropdownStyle] = useState({})
```

The `save` prop tells the component which field to store the selected value in when the users make the decision (this would be the mooring name). When the user taps the dropdown, it expands to show a list of all the mooring names retrieved from the database.

`setSelected=(val) => setSelected(val)`, connects the dropdown selection to the app's `Selected` state. When a user chooses an option, it automatically passes that option into the `setSelected` function, which updates the state variable with the new value in real time. This ensures the app always knows what mooring has been selected. I can use this stored value later on to handle database writes for a manual check-in.



I styled this check dropdown to be a simple box that takes up the screen width. To my surprise, there was a search feature built into the dropdown so I could search for any mooring I pleased.

```
const [search, setSearch] = useState("")
const [selected, setSelected] = useState("")
const [moorings, setMoorings] = useState([])
const [value, setValue] = useState("")
const [boxStyle, setBoxStyle] = useState({})
const [dropdownStyle, setDropdownStyle] = useState({})
const [searchText, setSearchText] = useState("")
const [searchTextColour, setSearchTextColour] = useState("")
```

In the library, I discovered a search placeholder, `Prop`. I changed this to `Search Moorings`.

However, there was no colour `Prop`.

Researching, I found an issue had been created within their GitHub library. As I wanted the background to be dark blue, the black text would not contrast well at all. After reading the issue created in 2023, an admin and a developer found that you can add a `searchTextColour` prop yourself in the

²³ <https://www.npmjs.com/package/react-native-dropdown-select-list>

nodeModulesFile. Following their steps, I successfully created my own searchPlaceholderTextColour Prop, which I set to white. I also ended up adding the loading indicator, as I would get the issue of the empty array just like on the map screen, so this eliminated that issue.



Check In page Style Update

With the dropdown now working, I went ahead and styled the rest of the check-in page. Adding the theme background colour or a dark navy blue, which contrasts with the new search text placeholder colour. A simple message was added at the top to inform users of what the manual check-in page was for in case they had any confusion.

Check In page continued.

Now that the user had a way to select their mooring, they needed a way to actually submit their selection. To handle this, I created an on-submit function that triggers a submission to the database. However, for now, I just used it to trigger an alert displaying the selected mooring. This can be easily expanded to write check-in data, saving the user, mooring ID and time.

I wanted to ensure the submit could only be pressed once and only when there is a selected mooring. To achieve this, I used a useState variable called formReady.

```
const [formReady, setFormReady] = useState(false);
```

I then set up a useEffect hook that updates every time the selected mooring changes

```
useEffect(() => {
  setFormReady(selected);
  return () => {
    setFormReady(false);
  };
}, [selected]);
```

When a mooring is chosen from the dropdown, the form-ready function triggers as it has "selected" in the dependency array. If there is no currently selected mooring in the dropdown, then FormReady will remain false. I can use this to control the buttons' disabled state and also change their background. Solid when true or grey when false. This prevents empty submissions and shows when the form is ready.

Date/Time Pickers

Once users were able to select the mooring from the dropdown, I wanted them to be able to specify a planned checkout date and time. To achieve this, I decided to integrate a date and time picker using the react-native-community/datetimepicker library. This allows users to select a date and time from a device native picker instead of manually typing it in. It improves accuracy and reduces input errors. Using this library activates the native device pickers, such as a scrollable wheel or a calendar. On iOS, there is no okay or cancel button, unlike the Android version. I needed to manually add these specifically for iOS.

In React Native, when a user selects a date the function onChangeDate grabs the input and updates the state setDate Hook. A similar function handles the time with it updating the setTime Hook.

To allow users to pick their date and times, I used the date picker library. This component is conditionally rendered only when the user taps on the date input field, using the showDatePicker state. I set the picker's mode to date so it appears with days, months and years and applied a spinner style for a scrollable interface. The value prop is linked to the date State, which guarantees any changes made on the picker are reflected immediately. I also



```
(showDatePicker && /
<DateTimePicker
  mode='date'
  display='spinner'
  value={date}
  onChange={onChangeDate}
  minimumDate={new Date()}
  themeVariant='dark'
/>
)
```

restrict dates, limiting the user to not picking any past dates. I repeat the same process for the time.

Unlike the Android pickers, iOS does not provide a cancel or confirm button for the `DateTimePicker` component. Therefore, I manually implemented these buttons for iOS users only. By using `Platform.OS === ios` I can isolate the code for a specific platform.

When the picker is active, buttons are displayed below the input field, which allow the user to dismiss the picker without making a selection, or the confirm button finalises their choice, updating the corresponding states. Adding these buttons allows the user to have the same control no matter their operating system.

To improve usability, I included a text field that displayed the selected time and date, setting `editable=false` so that users cannot type directly into the field.

I used two text inputs on the page, with one for the date and the other for the time. When either is pressed, then using the `onPress` handler, it calls the respective picker, dependent on the operating system. This `TextInput` component provides a visual confirmation for the user of the date or time they have picked.

```

import { Platform } from 'react-native';
import { DateTimePicker } from '@react-native-community/datetimepicker';
import { useState } from 'react';
import { StyleSheet, Text, View } from 'react-native';
import { colors } from '../constants';

const styles = StyleSheet.create({
  container: {
    padding: 10,
  },
  input: {
    width: 100,
    height: 40,
    border: 1px solid #ccc,
    margin: 5,
  },
  picker: {
    width: 100,
    height: 40,
    border: 1px solid #ccc,
    margin: 5,
  },
  buttons: {
    display: flex,
    justify-content: space-around,
    margin-top: 10,
  },
  button: {
    width: 40px,
    height: 20px,
    border: 1px solid #ccc,
    text-align: center,
    line-height: 20,
  },
  text: {
    width: 100,
    height: 20,
    border: 1px solid #ccc,
    margin: 5,
  },
});

export default function DateTimePickerForm() {
  const [selectedDate, setSelectedDate] = useState(new Date());
  const [selectedTime, setSelectedTime] = useState(new Date());
  const [showDatePicker, setShowDatePicker] = useState(false);
  const [showTimePicker, setShowTimePicker] = useState(false);

  const onPressDate = () => {
    setShowDatePicker(true);
  };

  const onPressTime = () => {
    setShowTimePicker(true);
  };

  return (
    <View style={styles.container}>
      <Text style={styles.input}>eg. 15 August 2025</Text>
      <DateTimePicker
        mode="date"
        value={selectedDate}
        onChange={setSelectedDate}
        onOpen={onPressDate}
        onClose={() => setShowDatePicker(false)}
        display="default"
        isIOS={Platform.OS === 'ios'}
      />
      <Text style={styles.picker}>eg. 15:00</Text>
      <DateTimePicker
        mode="time"
        value={selectedTime}
        onChange={setSelectedTime}
        onOpen={onPressTime}
        onClose={() => setShowTimePicker(false)}
        display="default"
        isIOS={Platform.OS === 'ios'}
      />
      <View style={styles.buttons}>
        <Text style={styles.button}>Cancel</Text>
        <Text style={styles.button}>Confirm</Text>
      </View>
      <Text style={styles.text}>
        Date: {selectedDate.toISOString().split('T')[0]}
        Time: {selectedTime.toISOString().split('T')[1].split(':')[0]}
      </Text>
    </View>
  );
}

```

```

<TextInput
  style={styles.input}
  placeholder="eg. 15 August 2025"
  placeholderTextColor="gray"
  editable={false}
  value={checkOutDate}
  onPress={() => {
    setShowTimePicker(true);
    toggleDatePicker();
  }}
/>

```

To provide the user with a clear, informative reference to the current check-in session, I integrated a live check-in time stamp using the `DayJS` library. This timestamp updates every second, displaying both the time and the date in real time. This helps users confirm the exact time they check in. By using a `useEffect` hook, I can update the state `currentTime` with the exact time of the day.

This hook has its interval set to 1000 milliseconds, so it updates every second. It works by continually reloading the current time from `DayJs`, ensuring it's always synchronised with the real world. When the component unmounts, the timer is cleared to prevent memory leaks.

```

useEffect(() => {
  const interval = setInterval(() => {
    setCurrentTime(dayjs());
  }, 1000); // update every second
  return () => clearInterval(interval);
}, []);

const style = styles.liveDate;
const date = (currentTime.format('DDMMYY'))('DDMMYY');
const time = (currentTime.format('HH:mm:ss'))('HH:mm:ss');

```

I also displayed user's email is at the top of the page so they know who's logged in. This was done by retrieving it from `Firebase authentication`.

```

<Text style={styles.useremail}>User: {auth.currentUser?.email}</Text>

```

Submit button update

I updated the submit button logic so instead of enabling the submit button when only just the mooring had been selected, the `checkOutTime` and `checkOutDate` must be filled as well. This prevents users submitting incomplete submissions. The `use effects` dependency array contains the `selected`, `checkOutTime`, and `checkoutDate` `use state` hooks so it runs every time one of these changes. When all 3 have been filled in it will run and the conditions will be true, changing `formReady` to true.

```

useEffect(() => {
  setFormReady(selected && checkOutTime && checkOutDate);
}, [selected, checkOutTime, checkOutDate]);

return () => {
  setFormReady(false);
};

```

Writing to Firestore.

With the check-in page complete, it was now time to write this data to `Firestore`. As I wanted to be able to track check-in history, I created a separate collection called `checkins`. This is where all the check-ins will be stored. Each time a user checks in, the selected mooring, check in time, planned checkout date and time will be written to a new document.

To grab the current time of the check-in I use `serverTimestamp`, which records the exact time in the Firebase server, not from the actual device. This makes it consistent across all devices, so no matter their time zone or clock, they will always be accurate.

I set the check in time and date fields to the corresponding states, the method to manual, mooringID to the selected State and the userID being the current user logged in. This then gets added to the checkins collection. Using a separate collection to the one the moorings are stored in allows me to track the history over time without affecting the moorings data.

When the check in submits it leaves all the old fields still filled so to overcome this I reset all the states to blank after the document has been sent to Firebase, clearing the input boxes. I also trigger a refresh of the mooring list so it's up to date.

Each new checkin document records the user id, the selected mooring and the checkin time/date. However, it does not update the mooring occupancy, so other users won't know the mooring is now occupied. To inform other users, I make an update to the moorings collection by setting the occupancy field to true on the selected mooring. This makes sure that the map reflects the red marker and informs users as it listens in real time for changes, which will trigger the update.

```
const updateMooringOccupancy = doc(db, 'moorings', selected);
await updateDoc(updateMooringOccupancy, {
  occupied: true
});
```

Date and Time format issues

Using the raw check-in times directly in Firebase made it hard to compare against the current date and time. To address this, I combined both planned checkout date and time into one single date object, storing it as `expectedDateAndTime` in Firebase. This makes it easier to compare as I would need to compare the date against the current date and then for the time. This can be problematic with inconsistent time zones or devices. To begin I used `toISOString()`. This stores the date format as `YYYY-MM-DD`, which makes it easier to parse. It saves this new date to the `checkoutDate` state hook.

For the time I do a similar concept but in a different way. I utilise the `dayjs` library to reformat the time into `hh:mm A`. It then stores it into the state hook `checkOutTime`.

With the date and time now in a suitable format, I can begin to combine them. This process involved taking the date from the `checkoutDate` State saved as a string and combining the time state, which is a date object.

Using the `day.js` library, I created a `day.js` object for the date. This converts it from a simple string to a date object, allowing me to easily manipulate it.

```
const dateOnly = days(checkOutDate, 'YYYY-MM-DD');
```

Then I do the same for the time:

```
const timeObj = daysjs(time)
```

As time is already in a date object, I still pass it through `dayjs` as it ensures everything is consistent, so `dayjs` can process this even further. Finally, I combine the two by setting the hours, minutes and seconds from the time object onto the end of the date object. This took many attempts, but eventually I had a single datetime object.

This object represents the planned checkout date and time. Updating this in Firestore was simple by just removing the old separate date and time fields and adding the combined one.

```
method: 'manual',
mooringId: selected,
userId: auth.currentUser.uid,
checkoutDateAndTime: dateOnly.toDate(),
```

```
const dateTime = dateOnly
.set('hour', timeObj.hour())
.set('minute', timeObj.minute())
.set('second', timeObj.second());
```

AutoFreeMoorings logic

Now the check in page can submit manual check-ins and change the occupancy of the moorings, which reflect on the map screen. I had no way to tell the app to mark the mooring as available again when the `checkoutDateAndTime` has passed. Without this, the mooring could remain incorrectly marked as occupied, confusing other vessels. I considered a few options to handle this.

My first was using a Firestore cloud-side function. Cloud functions run server-side in the background, independently from the app. This means it can execute and change data without the user's device having to be online. I could create a scheduled function that periodically checks any active check-ins and compares the `checkoutDateAndTime` with the current server time, updating the mooring status if the time has passed. This approach would make sure moorings are always reflecting their true status without relying on client-side logic.

My other option was to handle this in the app itself. Creating a `useEffect` Hook that, on mount, compares all active moorings with the current date and time. This approach is simpler and avoids writing server-side code. However, it depends on the app being open or not for it to execute. This could result in state data if left too long unopened.

After some research, I found that I was unable to implement the server-side cloud function on my Firebase. This was because my plan, Spark, didn't allow server-side functions as it's a free plan. Therefore, I had to go with the client-side process. To begin automating the process of freeing moorings, I implemented logic directly in the app using `useEffect` hooks. When the component mounts the `useEffectRuns` triggering an `async` function called `autoFreeMoorings` to run.

```
useEffect(() => {
  const autoFreeMoorings = async () => {
    console.log('Checking moorings...');
    const checkInSnapshot = await getDocs(collection(db, 'checkins'));
    const now = dayjs();
    checkInSnapshot.docs.forEach(doc => {
      const data = docSnap.data();
      const checkout = dayjs(data.checkoutDateAndTime.toDate());
      const mooringId = data.mooringId;
      if (checkout.isBefore(now)) {
        console.log('Freeing mooring: ' + mooringId);
        await updateDoc(doc(db, 'moorings', mooringId), {
          occupied: false
        });
      }
      console.log('Moorings: ' + mooringId + ' is now free');
    });
  };
  autoFreeMoorings();
}, []);
```

This firstly retrieves all the check-in documents from the Firestore collection `check-ins`. I set `now` to the current day and time using the `dayjs` library. For each check in the document, the combined checkout date and time is converted back into a JavaScript date object and then wrapped into a `dayjs` object. This process allows me to accurately compare the checkout date and time with the current date and time. If the checkout date and time are before the current time, then the mooring scheduled checkout has passed, so the mooring should be freed up. The app updates the moorings Firestore document, setting `occupied` to `false`. This will mean the map will update and will show the mooring as now free.

If the checkout time has not yet passed, then that means the mooring is still in use. By logging it, it allows me to debug and make sure everything is running smoothly and confirms the function is correct. This removes the need for manual checks and updates and ensures mooring availability is always up to date. The app will run this every time it is opened, as the dependency array is empty.

Automatic Mooring Detector.

It was now time to begin on the automatic mooring function. This was my main feature in the MVP. I had this idea from the start, where boats could drive up to a mooring, and it would automatically detect if a mooring is there and ask the user if they want to check in. It then would remember they are there and detect when they leave, making it unoccupied.

To start, I began thinking of how I could detect nearby moorings based on the user's current GPS location. I could query all the moorings in the database and calculate the distance to the closest one using the GPS coordinates and the mooring coordinates; however, this will be ineffective with database read restrictions. With multiple users using the app on the same day, I would reach my threshold too fast.

To overcome this, I had the idea of a box around the boat that, whenever a mooring enters that zone, it detects it and queries those moorings only. Realistically, there would only be 2 or 3 moorings in the zone at once, so this will keep my reads low. To implement this, I created a function called `FetchNearbyMoorings`. This queries the database for any moorings in a small bounding box around the user's location. The function takes in 2 parameters, being the latitude and longitude, which are retrieved from the device's location. I also created two constants, `DeltaLat` and `DeltaLng`. These represent how far north/south and east/west the app should query from the current position. These values roughly work out to be around 100m in each direction, allowing the app to focus on nearby moorings and ignore distant ones.

```
const [latNearbyMoorings, setLatNearbyMoorings] = useState(0);
const [lonNearbyMoorings, setLonNearbyMoorings] = useState(0);
const [deltaLat, setDeltaLat] = useState(0.001);
const [deltaLng, setDeltaLng] = useState(0.001);
const [nearbyMoorings, setNearbyMoorings] = useState([]);

const fetchNearbyMoorings = async (latitude, longitude) => {
  const querySnapshot = await getDocs(query(
    collection(db, 'moorings'),
    where('latitude', '>=', latitude - deltaLat),
    where('latitude', '<=', latitude + deltaLat),
    where('longitude', '>=', longitude - deltaLng),
    where('longitude', '<=', longitude + deltaLng)
  ));

  const nearbyMoorings = [];
  querySnapshot.forEach(doc => {
    const mooring = doc.data();
    nearbyMoorings.push({
      id: doc.id,
      name: mooring.name,
      description: mooring.description,
      latitude: mooring.latitude,
      longitude: mooring.longitude,
      occupied: mooring.occupied,
    });
  });

  console.log('Nearby Moorings:', nearbyMoorings);
  return nearbyMoorings;
}
```

Inside this function, I use Firestore's `where` query to create a box query that filters out moorings based on the latitude and longitude. This creates an invisible search area around the vessel's current position. The `getDocs(BoxQuery)` retrieves and returns all mooring info that falls into the range. When I ran this, I received an error. Firebase told me that the query needed an index. This is because Firebase requires composite indexes whenever multiple fields are filtered. In my case, this would be latitude and longitude. In the console, it gave me a link to follow to create this in Firebase. It generated the index, and then the query ran. Indexing is important as it makes querying faster and actually possible. Without this, Firebase cannot scan 2 dimensions at once. For each document that is returned, it extracts the ID, name, description, latitude, longitude and occupancy status, saving these into the `nearbyMoorings` state. This function is the foundation of my automatic mooring feature.

```
const [latNearbyMoorings, setLatNearbyMoorings] = useState(0);
const [lonNearbyMoorings, setLonNearbyMoorings] = useState(0);
const [deltaLat, setDeltaLat] = useState(0.001);
const [deltaLng, setDeltaLng] = useState(0.001);
const [nearbyMoorings, setNearbyMoorings] = useState([]);

const fetchNearbyMoorings = async (latitude, longitude) => {
  const querySnapshot = await getDocs(query(
    collection(db, 'moorings'),
    where('latitude', '>=', latitude - deltaLat),
    where('latitude', '<=', latitude + deltaLat),
    where('longitude', '>=', longitude - deltaLng),
    where('longitude', '<=', longitude + deltaLng)
  ));

  const nearbyMoorings = [];
  querySnapshot.forEach(doc => {
    const mooring = doc.data();
    nearbyMoorings.push({
      id: doc.id,
      name: mooring.name,
      description: mooring.description,
      latitude: mooring.latitude,
      longitude: mooring.longitude,
      occupied: mooring.occupied,
    });
  });

  console.log('Nearby Moorings:', nearbyMoorings);
  return nearbyMoorings;
}
```

With the query now returning all nearby moorings within the set distance, I wanted this to visually appear on the screen. I wanted to see where this box was and if I needed to make adjustments. To do this, I added a polygon component to the map. This is in the map view library, which I already use. The polygon draws a visual green rectangle around the user's current location, representing the area the app checks for the nearby mooring query. It uses the same latitude and longitude delta values, so it is the exact same size.



Whenever the user moved, the polygon would update over the location to show the detection zone. Any moorings in that box would be detected by the Firestore query.

Update to the auto free function

Currently, the autofree mooring function is located on the check-in page. Whenever this page is opened, it will run the function to free any stale mooring. However, some users did not go on to this page during their sessions, so I decided to move the whole function to the home page. Every user is faced with this page on app launch, causing the function to run. This will ensure any user who uses the app contributes to ensuring there are no stale manually checked-in moorings present in the database and on the map.

More pages.

While reading about app requirements for the App Store, I discovered that every app needs to contain a terms and conditions page and a privacy page. This was a simple add-on by just updating the menuStack with the new pages and adding them to the more page. I used an online app generator to create the privacy and conditions policies. This was then pasted into the newly created pages and was ready to go.

Submitting to the App Store

With the app's MVP almost complete, I wanted to get a development build onto a real device, not using Expo. This would ensure that the app functions correctly and independently without relying on Expo. This would not be published to the app store, only utilising Apple's app testing program, TestFlight.

First of all, I needed an Apple Developer account. As I am [REDACTED] this is not allowed as it breaches Apple's terms and conditions. After days of phone calls back and forth with the Apple Developer Support Team, they decided that one of my parents would need to set this account up under their name for me to access the Apple Developer Portal. With much discussion, they allowed me to do this on their behalf. I used my father's iCloud email and created the account. To set one of these accounts up, Apple charges \$99 USD per membership year, costing me almost \$150 NZD after tax. I submitted my first version to Apple 1.0.0. I intended to use this version to ensure the permissions work as expected, the box and location work, as well as to monitor the Firebase read data.

Sentry

To aid with testing, I used Sentry, an application that monitors performance and error tracking.²⁴ It is used by millions around the world and on popular apps like Disney and Microsoft to help developers catch and fix errors fast. When an issue or bug is detected on a user's device, this logs it to the dashboard with relevant information about the device and location of the error. It also provides a screen replay of the issue on the

²⁴ <https://sentry.io>

user's device and console logs. I installed this into my app's root file, app.js. When any issues occur anywhere in the app it's captured and sent to me.



AutoFreeMooring Fix

I noticed that I was receiving a high volume of document reads every 60 seconds. This was because the auto Free Mooring logic was querying the whole checkins collection to find documents that contain expired check-outs. It would fetch the whole collection and then check for expired checkouts. This approach was ineffective and would exceed Spark restrictions if I continued to leave it untouched. To fix this, I added a WHERE statement to the query that compares the checkout date and the current date and time. If the checkoutdateandtime value is less than or equal to the current, then it will only retrieve that document.

```
where("checkoutDateAndTime", "<=", new Date())
```

By using this condition, it filters moorings only with state check-ins. This significantly reduces Firestore reads and improves the scalability of the app overall.

I also discovered that this would keep returning the same check-in doc, even after the mooring was marked as unoccupied. This is because all the past check-ins would still be queried. To fix this, I added a processed field to each check-in document to prevent the app from freeing the same mooring multiple times. Without this process field, the free mooring function would detect the same expired mooring every time it ran and try to update the mooring occupancy and the check-in doc every time. This was unnecessary and caused more high reads and writes on the database. With the new processed field, I can identify what check-ins are still active. Once a check-in goes through the auto free function, I can set the process field to true, signifying it has been handled and no further action is required on that document.

Map mooring modal update:

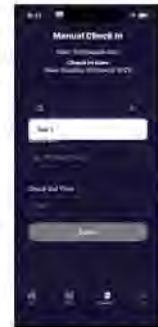
If a user checks into a mooring, other users can see the red occupied marker; however, they are in the unknown about the details of the check-in. I wanted to be semi-transparent with users of vessel check-ins. With talking to other club members, they expressed how being able to see the current occupants check out info would be easier instead of being in the dark. To add this small feature, I edited the mooring modal at the bottom with a small check in the section. This contains the checkout date and time of the user, but still hides who is on it and their name; this is for privacy.

To implement this, I create a new useEffect hook with two where statements. One to match the mooringID to the selected mooring and the other to filter the unprocessed checkins. This query is a real time listener that uses onSnapshot, so any changes are immediately shown without the need to refresh. When a valid check in is detected, the function receives its checkoutdateandtime and reformats it nicely at the bottom of the modal. If there are no active checkins for that modal it shows no active check ins. This was all the info stays up to date about mooring availability, while personal details remain hidden.

Manual check in dropdown update:

Now that I had a function to fetch all nearby moorings based on the user's location, I decided to update the dropdown to use this data instead of the whole moorings collection. Previously, users could theoretically reserve or check into moorings from far away from their position. This was inaccurate and could lead to user confusion.

To solve this, I changed the dropdown's data source to the nearbyMoorings State, which is updated every few seconds. I copied the nearby mooring logic over to the check-in page to get access to this state. When a mooring is returned, it is added to an array containing the name, ID and occupancy status. Occupied moorings are marked as disabled and greyed out in the dropdown, meaning users cannot physically select already occupied moorings. The dropdown now automatically refreshes to show the user's nearby moorings and live occupancy data, improving the check-in process for users.



Update to Map Page

As the map page and check-in were beginning to become clustered and messy, with repeated nearby mooring logic, I decided to move them to a dedicated file, which would contain all the logic and location permissions/location listener. I aimed to reduce the number of queries to the database, so I copied and pasted all of it over to the new file. To access the nearbyMoorings function, I can easily call the useNearbyMoorings() custom hook from any page that needs mooring data, such as the manual check-in or auto system, without duplicating code.

I then discovered that because I import the hooks separately on different pages, they each trigger separate Firestore queries when opened. This resulted in double the amount of reads from the database. To optimise performance and reduce this, I needed a way to let each page share the same state without requiring the database every time.

To fix this issue, I implemented React's useContext hook²⁵. This allows data to be shared across multiple components. This made the structure of the app cleaner and easier to manage as I used useContext to share the nearby moorings state with the other pages. I wrapped the main navigator inside a context provider, making the nearby moorings data globally accessible. Pages that need access to this shared state can import the context and retrieve data like so.

```
const { nearbyMoorings, location, loading } = useContext(NearbyMooringsContext);
```

On the nearby moorings page, I returned the states through the provider, allowing other pages to now use the shared state.

```
return (
  <NearbyMooringsContext.Provider value={{ nearbyMoorings, location, loading }}>
    {children}
  </NearbyMooringsContext.Provider>
);
```

The method allowed other pages to access shared data without calling query functions twice, improving performance and reducing reads. It's made the structure cleaner and easier to maintain.

Automatic mooring occupier continued.

With the app now returning the closest few moorings with a bounding box query, it was now time to begin the next step of finding the mooring actually closest to the boat. The bounding box can return multiple moorings within the latitude and longitude range, but cannot determine which one is nearest to the vessel. This is the reason why I created a

²⁵ <https://react.dev/reference/react/useContext>

bounding box query; otherwise, without it, I would be calculating the closest distance to every single mooring, significantly reducing the performance of the app. This will be extremely useful when I need to make decisions about what mooring to automatically mark as occupied. If there are 2 moorings in the box, this will find the closest one and save it to a state called `nearestNearByMooring`.

To find the closest mooring out of the array of `nearbyMooring`, I would use a library called `geolib`. I use the function `findNearest`, which takes a reference point of the boat's location and an array of coordinates from all the moorings and returns the single mooring that's geographically closest.

```
const nearestNearByMooring = findNearest(location, nearbyMoorings);
```

This nearest mooring object gets saved to the state variable `nearestNearByMooring`. By having this the app can now make decisions based on the mooring currently closest.

Next, with the `nearestNearByMooring` containing the nearest mooring I can calculate the distance of the mooring to the vessel in meters. I use the

`getDistance` function from the `geolib` library to calculate this. This returns a number in meters that is saved to the state `distanceToClosestNearByMooring`



I had to put the `getDistance` logic in an if-else statement, as when the boat is driving and no moorings are nearby, the array of nearby moorings will be undefined, as it will be empty, causing the `getDistance` to break. Therefore, setting them to null allows them to be blank, and then when a mooring comes into vacancy, it runs the `getDistance` and `findNearest` logic without breaking. If I had two moorings close to each other, I would only use the closest one.

Under the hood of the `getDistance` Function, it is quite complex. As the earth is a sphere, you cannot directly draw a straight line from the mooring to the vessel. I must account for the Earth's curvature. Using the Haversine formula, it calculates the shortest distance between 2 points, being the mooring and the vessel. It ignores all hills and is the shortest distance over the earth's surface, giving an "as-the-crow-flies"²⁶ distance.



With the distance to the closest mooring now calculated, I integrated a function to detect when a vessel is within a predefined proximity to a mooring. I used a 20m threshold. If the vessel enters this range, I mark the mooring as occupied. However, to avoid unwanted false check-ins if a vessel just passes by a mooring, I implemented a timer function. This timer starts when a vessel enters the 20m range and counts down from 20 minutes. Only if the vessel remains in this proximity while the timer finishes the mooring, then it automatically marks as occupied, with a check-in document created, and the mooring status updated. If the vessel moves away from the mooring before the timer has finished, then it is cleared and stays unoccupied. This stops the mooring auto function from accidentally triggering.



²⁶ https://en.wikipedia.org/wiki/As_the_crow_flies

I ended up changing the values for testing purposes. I increased the distance threshold to 400m and the timer duration to 5 seconds.

I improved the logic to handle consecutive proximity events. If a new timer begins while an old timer is still running, the old timer clears and is replaced with a new one. This makes sure only the most recent proximity event controls the occupancy check. Stopping overlapping timers from causing inconsistent mooring status. When the vessel leaves, the timer is cleared to prevent memory leaks, as the constant use of timers could crash or slow the app down.

I updated the automatic mooring logic to handle repeated occupied triggers. Previously, a new timer would start even if a mooring was already occupied, and if the user left the mooring, then the state would update incorrectly. In the new version, a timer only begins if there isn't one already and the mooring is currently unoccupied. This will prevent duplicate timers. When the boat leaves the proximity, the timer is cleared, and if the mooring was occupied, it is now marked as occupied. This makes sure the `isOccupied` state shows accurately if a vessel is at a mooring or not.

The logic now updates the relevant documents regarding the nearby mooring status. If a mooring is within 400m of a vessel, it updates the occupancy to true, and if they leave, it marks it unoccupied.

```

// Check if the vessel is within the proximity of the mooring
const isWithinProximity = (vessel, mooring) => {
  const distance = calculateDistance(vessel, mooring);
  return distance <= 400;
};

// Check if the mooring is currently occupied
const isOccupied = (mooring) => {
  return mooring.isOccupied;
};

// Update the mooring status
const updateMooringStatus = (mooring, vessel) => {
  if (isWithinProximity(vessel, mooring) && !isOccupied(mooring)) {
    // Mark the mooring as occupied
    mooring.isOccupied = true;
    // Start a timer to mark the mooring as unoccupied after 5 seconds
    setTimeout(() => {
      mooring.isOccupied = false;
    }, 5000);
  }
};

// Handle the proximity event
const handleProximityEvent = (vessel) => {
  // Find the nearest mooring
  const nearestMooring = findNearestMooring(vessel);
  // Update the mooring status
  updateMooringStatus(nearestMooring, vessel);
};

```

```

// Handle the proximity event
const handleProximityEvent = (vessel) => {
  // Find the nearest mooring
  const nearestMooring = findNearestMooring(vessel);
  // Update the mooring status
  updateMooringStatus(nearestMooring, vessel);
};

// Find the nearest mooring
const findNearestMooring = (vessel) => {
  const distances = nearbyMoorings.map(mooring => {
    return {
      mooring,
      distance: calculateDistance(vessel, mooring)
    };
  });
  return distances.sort((a, b) => a.distance - b.distance)[0].mooring;
};

```



Flow chart of automatic mooring function

Automatic mooring safeguard

I wanted multiple ways to prevent unwanted check-ins and inaccurate data. One way I implemented this was if a user is checked into a mooring and they suddenly have a new mooring in the array that's different to the occupied one, then this clearly indicates they have left or closed the app and opened it at a new location. Or if they are still within 400m, but there is another mooring closer. The logic automatically sets the previous mooring to unoccupied and clears the `isOccupied` status, as well as clearing the `currentMooring` state. This prevents moorings left incorrectly marked as occupied.

```

// Clear the current mooring
const clearCurrentMooring = () => {
  currentMooring = null;
};

// Update the mooring status
const updateMooringStatus = (mooring, vessel) => {
  if (isWithinProximity(vessel, mooring) && !isOccupied(mooring)) {
    // Mark the mooring as occupied
    mooring.isOccupied = true;
    // Start a timer to mark the mooring as unoccupied after 5 seconds
    setTimeout(() => {
      mooring.isOccupied = false;
    }, 5000);
  }
};

```

Visual mooring alerts

When a user enters the zone and checks into a mooring, they don't actually get told that they have checked in, apart from the map marker turning red. To keep users transparent of the inner workings, I added an alert that pops up when the automatic trigger marks a mooring as occupied.

```

alert(`You have marked ${nearestNearByMooring.key} as occupied`);

```

I then decided that what if the alert asks the user if they want to automatically check in. This gives the user control and freedom, according to Jacob Nielsen's usability Heuristics.

This two-option alert consists yes or no and a message from the native alerts function of the device. If the user presses no, the check-in aborts, but if they press yes, then it goes through with the check-in to the moorings and adds a new doc to the checkins collection. Users are unable to cancel the alert, as it would cause the function to break; therefore, I set cancellable to false. This gives them the option of either yes or no. If they tap out of the alert, it will not dismiss.

Menu page update – settings screen.

I added a settings screen in the menu page designed to allow users to manage their accounts and location permissions on a single page. If a user wants to sign out, they can do so by pressing the log out button, which triggers `auth.signOut()` method to sign the user out and navigate them to the login screen. Any issues during this are displayed in alert form.

Another feature on this page was the ability to recall location permissions, which is critical for the app's core functionality, such as detecting a mooring. If a user denied location permission on download, then that would make the app virtually unusable; however, if they decide they want to enable it, they can navigate to this page and press enable location. This will open the settings app, open to the app settings, so they can re-enable it. If the permissions are already granted, then the user will receive an alert stating this. This method accounts for the limitations developers face with handling Apple and Android's strict limits, which prevent apps from programmatically requesting users for location access after the initial denial.

Map Markers bug fix

With the new automatic mooring check-in function working, I came across an error. When they press yes on the alert to check in, it instantly makes the mooring occupied. Now, when a user opens the modal on the marker, the app breaks. This is because it's trying to fetch a checkout date and time that doesn't exist.

To fix the issue and to improve the accuracy of the automatic mooring system, I implemented an expected checkout modal that pops up asking the user to enter a checkout date and time. This would act as a safeguard in case the checkout auto logic failed. By prompting the user to provide the expected checkout info, users can view when they expect to leave, as well as maintain the accuracy of the occupancy data.

The modal uses the `DateTimePicker` components reused from the check-in page. Users can select a date and time formatted correctly to store in Firebase and ensuing platform-specific functionality with dedicated confirm and cancel buttons on iOS. Once the user confirms the expected date and time, the app combines them and updates the check-in document in the database.

This system tries to ensure that a vessel that remains on a mooring for an extended time has an expected checkout date. If a user were to check in and never return to the app after the checkout date will be used to ensure the mooring frees up after the checkout date has passed, thanks to the auto-free mooring logic implemented on the home screen.

Auto check in update

```
if (timer.current === null && (isOccupied === false || currentMooring?.key !== nearestNearByMooring.key) ) {
```

This updated if statement ensures that the timer used for proximity only activates under the correct conditions. The logic checks whether the timer is currently inactive and whether the user is not already occupying a mooring. Or the nearest mooring is different

to the currently occupied one. If these conditions are met it means the vessel has entered the range of a mooring that is not yet occupied. This prevents unnecessary timers from starting when a user is already checked in or when the same mooring remains closest, reducing false check-ins.

First live trial

On 5th September, I flew up to the Sounds for the weekend for the first on-site trial of the MVP. This date marked the completion of the basic MVP, from this point on all changes to the app would be bug fixes and improvements. Once I was happy with the functionality then I pushed

it out to the volunteers for testing on the App Store. This real test was specifically for the app's purposes to ensure it works correctly in its intended environment. On the plane to the Marlborough Sounds I worked on tidying up the manual check-in page and ran tests to ensure my app was error-free.

When I arrived on site I went straight to the boat to begin testing the application in real-world conditions. I stayed the night on one of the club moorings to observe how the automatic check-in system behaved.

While the boat entered the proximity of the mooring, I had the app open console logging to track the events to help with debugging and error catching. When I came within 400m of the mooring, the app successfully prompted me to check in, confirming the auto check-in systems were functional and working as designed. Instead of burning fuel to test the functionality of driving in and out of the moorings proximity, I decided to drop the boat's electric tender and cruise around with my laptop, which monitors the logs.



Active check in persistence.

One key issue I identified that was when a user closed the app during an active check-in the app would lose its memory state. On reopening, the app would think they had no current check-ins and prompt them to check in, even if they were at the same mooring.

To solve this issue, I developed a `restoreactivecheckin()` function that runs on mount. This function queries the database for any active check-ins where `processed == false` associated with the current user id. If one exists, the app restores the previous session by updating the local states (`currentMooring` and `currentCheckinDocId`) with the users active mooring data. This state persistence allows users to leave and reopen the app without losing session data.

However, the app still prompts the user on reopen. To prevent this, I refined the condition that triggers the alert.

Before

```
if (timer.current === null && isOccupied === false)
```

After

```
if (timer.current === null && (isOccupied === false || currentMooring?.key !== nearestNearByMooring.key))
```

This ensures that if the user is still physically on the same mooring the app does not prompt it. To back this up further, I added another safeguard.

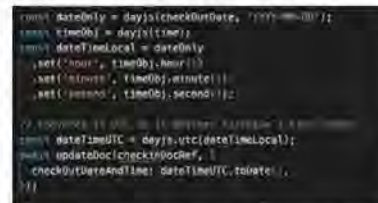
This extra condition checks whether the user remains within the 400m of the current mooring and prevents duplicate prompts for the user. This makes the app more reliable and realistic as users who stay the night on a mooring won't keep the app open the whole duration of their stay. They will open to check in, close it at night and reopen it in the morning to checkout.

This also allows users to sign out on their device and log in to a new device while on a mooring. This would not be possible with async local storage so fetching the activecheck check-ins via the database expands the persistence across multiple devices instead of just one. I tested this and it worked without flaw!



UTC bug

During testing, I noticed that moorings were automatically making themselves unoccupied, a short time after checking in automatically. At first, I thought it was caused by GPS fluctuations as while on the water, as GPS can be less precise. I checked and observed only a small variation in the GPS logs. To test my theory, I adjusted the bounding box size and increased the mooring proximity threshold to see if the false unoccupying moorings would stop. However, this was not the case, they still continued. I confirmed that the GPS inaccuracies were within 5 m, so this was not the root cause. After further investigation, while testing, I discovered that if I aborted an auto check-in by closing the app after pressing yes and ignoring the checkout modal, this error would not occur. That meant it had to be an issue with the date and times. I found that it was an issue with how Firebase stores and compares the timestamps in the auto-free mooring logic. Firebase stores their timestamps in UTC format (Coordinated Universal Time); however, the app used the devices local times. This meant for auto free mooring comparisons, whenever a mooring was set for a specific time, the UTC version of that time would be around 1 day behind, making the mooring instantly unoccupied.



To fix this, I implemented the daysjs UTC plugin. This ensures all timestamps stored convert the local device's time/date to UTC that matches the Firebase format. When a user sets a checkout time and date the app converts to UTC before saving.

Bounding box perspective

During the live testing, I also discovered that the green box used to show the mooring proximity area was significantly larger than anticipated. When testing on simulators, it didn't seem too big, but 400x400 is a large area. When this was put in context, it would almost take up the whole bay, which is not correct. This was due to a lack of perspective, which I wouldn't have been able to identify unless it was for the real testing. To resolve this issue, I adjusted the box sizes by decreasing the deltaLat and deltaLng values to a better realistic distance. I also adjusted the proximity. This made proximity detection more accurate.

Map marker hitbox:

While talking to older members of the boating club, I showed them the progress and let them operate the app for themselves. A few mentioned that the mooring marker modals wouldn't open when pressing on the mooring. At first I assumed it was a bug, but when I tested on their device directly I discovered it wasn't the functionality. It was the marker's

hitbox. The default map markers only register touch inputs at the very tip of the small point. This makes it difficult to press accurately. A lot of my user base is older with larger fingers, so this is especially frustrating for them.

To fix this problem, I implemented my own custom markers to replace the default pins. I used the custom marker prop, which allows me to import my own image; however, it doesn't allow me to resize. After I implemented the necessary logic to switch between red and green and submitted it to Apple, I soon realised that the markers were incorrectly sized. Even though they looked small on my simulator, once Apple approved the app and it was updated on a real device, they blew up drastically.

These looked old and ugly, so I decided to change them again. I switched to using the icon-based markers since the library already imported them for the navbar icons. This gave me control over the layout and interaction with the modal and the map. By using vector-based icons, I was able to customise the onpress handling to ensure the whole icon was clickable, not just the tip. This approach not only improved accessibility for older users and reduced dexterity, users it also made the UI cleaner and more consistent across different devices.



Firestore performance monitoring.

While testing my app over this test duration I constantly monitored my apps live read and write usage. In the past 24hrs of app usage during my tests, I ended up opening the app 9 times which fetches the 100 moorings 9 times. This caused the app to record over 900 reads in just initialisation alone. This would mean if 500 users opened the app once per day the system would exceed Firestore's 50k daily free limit and would cause the app to not respond.



The database is currently hosted in the NAM5 (North America) region, which is unnoticeable for the database size, but for future expansions, this would start to cause issues. On the Spark plan, I receive 50k free reads and writes, then the app stops responding. However, there is another plan called Blaze, and this still contains the free 50k reads and writes, but as soon as this threshold is reached, it begins a pay-as-you-go service. If the club were to follow through with this app after the project and roll the app out to every member (100s), then this plan would be essential. If I breached this threshold on the Blaze plan, then the price per 100k reads is \$0.03; however, if I migrated the server location to a place closer, like Sydney, then this price is halved. The Blaze plan also allows advanced cloud features, such as cloud functions, that are not available on my current Spark plan. These will be real considerations once the app goes live with all members.

Preventing multiple people checking into same mooring

Another key issue to solve was preventing multiple people checking into the same mooring at once. Without anything to manage this, two users could accidentally check into the same mooring which would cause confusion and inaccurate data. This could happen if two moorings are close enough and with one being occupied and another free.

To fix this, I added logic that checks the current occupancy status of a mooring before allowing a user to check in. If a user tries to check into an occupied mooring, the system compares the ID of the currently signed-in user with the ID of the user who is already checked in. If these match, then this is returned as is, prompting an alert. However, if the mooring isn't theirs, then they get a message saying the mooring is already occupied. This logic ensures that each mooring can only have one active check-in at a time, preventing data conflicts. It also improves the reliability of the auto free mooring system as there's now a clear one-to-one relationship between users and their active moorings.

```

const handleCheckIn = async (mooring) => {
  const mooringCheckIn = docRef('moorings', mooring.key);
  const mooringCheckInSnapshot = await mooringCheckIn.get();
  const mooringCheckInData = mooringCheckInSnapshot.data();
  if (mooringCheckInData?.occupiedBy === auth.currentUser.uid) {
    Alert.alert('Error', 'This mooring is already occupied.');
    console.log('Mooring occupied by:', mooringCheckInData.occupiedBy);
    return;
  }
  mooringCheckIn.set({
    occupiedBy: auth.currentUser.uid,
  });
  console.log('This mooring is already yours, no need to prompt.');
};

```

At home testing

While I ran out of time during the weekend, when I was in the Sounds for testing, I still wanted to properly test the auto check-in and check-out logic. Since I couldn't test on the water every day, I came up with the idea of placing temporary test moorings around my property instead. I set up several moorings along the driveway and then used my car to simulate the movement, driving up and down to trigger the GPS-based detection. This allowed me to replicate the same location behaviour as on the water and helped me to identify future issues.



UTC issues part 2...

While performing more tests at home, I discovered an ongoing issue with the checkout date and time again. The issue was the UTC timestamps. The previous fix didn't seem to fix it. The bug was that if a user checked in the morning and set a checkout date for the same day before lunch, then the UTC timestamp would become several hours behind the NZDT. As a result, the mooring would be automatically unoccupied as the auto-free mooring function would compare the past date to the current device's time and date. If I were to check in after lunch, then it would work fine and cause no issues. This is how I narrowed this issue down.

For example, if I checked in at 10:00 AM NZDT, it would store as 9:00 PM UTC on the previous day, resulting in an instant unintended unoccupancy.

To fix this, I updated the logic to first build a local date and time using the DayJs library, then explicitly convert to UTC in a new method than before. This ensures all time stamps are consistent regardless of the user's device time zone.

```

// Convert the date and time to UTC
const utcDate = dayjs(checkoutDate).format('YYYY-MM-DD HH:mm:ss');
const utcTime = dayjs().format('YYYY-MM-DD HH:mm:ss');

// Convert the date and time to UTC
const utcDate = dayjs(checkoutDate).format('YYYY-MM-DD HH:mm:ss');
const utcTime = dayjs().format('YYYY-MM-DD HH:mm:ss');

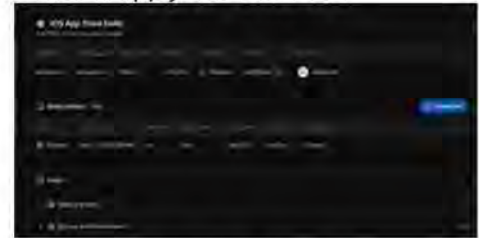
```

Submitting for production

With it being mid-September, I was under pressure to get a version to my test subjects. To submit a production build to the App Store, I ran `eas build --platform ios & eas submit --platform ios`. This uploads the build to Expo, which then generates a .ipa file and connects via my Apple ID and submits to the App Store



automatically. This process normally takes around 20 minutes in total. Every time I submit a new build, I must increase the build number and version in app.json. Once this has been uploaded, I can either choose the build to be a TestFlight test or submit for distribution. While TestFlight is simple for some users, the generation of my test subjects may not find it easy to set up. Therefore, I decided I wanted to push it to the App Store. To begin this process, I created a new version and began filling in relevant details such as privacy and accessibility information. Once this was done, I could upload screenshots to display on the app store's page. It required multiple devices of different sizes, so this process took longer than expected. I used simulators and ran the app on each one.



When apps are submitted to the App Store, they undergo a review process. This is when an Apple developer reads your program to ensure you do not exploit the App Store for its intended use. For me, this process took 2 days, and when I checked, I saw rejected. Apple rejected the submission due to a lack of information and business model, so I replied to them with their new information. While anxiously reading all the scenarios on Reddit about people having to wait weeks for Apple's reply. They responded within 20 minutes and approved the application! This meant the app was now on the App Store, so searching "PWM Moorings" will show my app as the first result.



Top 57 in NZ 🐝

User account creation

Now that my app was on the App Store, anyone could view it or download it. Therefore, I needed to restrict it to users at the club only. The login page only had a login button and not a signup button, as this will stop non-club members from signing up

Once this was done, I set up a system to create accounts manually, as there is no sign-up function within the app itself.

Now that I had the responses of people wanting to sign up for the trial, I needed a way to make accounts for every user. One way to do this was by going into Firebase Auth and manually adding every user with a password. This would have been very tedious and time-consuming, so I decided to keep exploring my options. Also, I needed a way to give them the ability to set their own passwords. This is because I removed the sign-up button on the main login page, as the app is on the App Store anyone can download it and use it, so by removing it, it creates a private, access-only application. The Apple review did not like this, but after some replies to their questions, it got approved.



I found that you can reset passwords for individual users in the Firebase Auth portal, which sends them an email from Firebase with a link to a website to reset their passwords. Doing this manually emails them with a premade template you can edit. To take advantage of this, I reformatted the template so it doesn't say reset your password it says create your password. However, upon further investigation, you cannot mass send these emails out; only send one at a time in Firebase, so I explored another option. The last one was a simple Python script that connected to my Firebase Auth, taking emails from an array, and iterating through per email by creating a user account. Once the account is created, it completes another action and generates the account a password reset link. This just logs it to my console. It does not email it to the user as Firebase doesn't support this, so I utilised a Python emailer and set the link as a variable and passed it into the emailer. I put a small message with the link to the app and then the password reset link. Every time an email is passed into the script, it creates an account, generates a password reset link, passes it into the Python emailer and emails the user their link through my Gmail account.

I wanted a way to quickly email all the users instead of pasting all the emails from the form into the script. By using a library called pandas, my script can iterate through an Excel spreadsheet and create the account directly from the Python script.

While testing this out, I realised that the password reset link only lasts one hour. As I was sending this out at 1 am, no one would see it, so I needed a new way to get around it. That's when I thought, "Why don't I just set everyone's password for them?", so I did. I set everyone's password to "Wbc123" so users could log in. Once this was fixed, I then executed the script.

I then emailed them with relevant information and provided a feedback form, once they had concluded their testing.



User testing - Apple

The day I sent the accounts out, I had to leave for a 10-day trip with no access to devices. When I returned, I found that most of the users had signed into the app, and a few users had tested it! This was a successful start to phase one of the app trial.



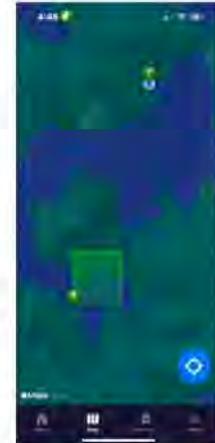
Proof of check in history

First issue from live trial: bounding box doesn't follow on cold start

During the live trial, I had a few people reach out and explain that while launching the app for the first time, the bounding box that determines the range for nearby moorings did not update or follow the user's location, which had an effect on the auto occupier function not running.

At first, I thought this was an issue with offline functionality. When going through patchy zones, the service drops out and causes the box to stop following. However, testing offline and online, I confirmed the issue was with how the app was being initiated. The first time the app is launched, the location state is null as the user has not allowed permission yet.

Because the bounding box replicated on the user's GPS coordinates to calculate distance to nearby moorings, they failed to run when the location state hadn't been set yet. The functions were being called before the location data was available.



To fix this, I introduced a `locationReady` state to act as a green light that the location has been received. I wrapped all my logic that requires this location in my `LocationReady` checks to prevent it from executing prematurely.

Once the first location update is obtained, I set `locationReady` to true, allowing both the bounding box and auto-occupy mooring logic to run now that the location data is valid.

This fix resolved the cold start issues.

```
if (!restoreDone || !nearbyMoorings || !locationReady) return;
if (!location) {
  console.log('Location not ready yet!');
  return;
}
// nearbyMoorings.length === 0
console.log('Waiting for nearby moorings!');
return;
```

Android date time picker issue 1

An issue I discovered on Android while preparing the Android version was that when a user tries to select a date and time on the Android-specific picker, the picker would instantly snap back to 12 pm on the current day, regardless of the user's selection. This behaviour persisted even when I stripped the whole page back to its bare functionality. I initially suspected it was the min-max properties to stop users from selecting dates in the future. I even copied all the logic for the automatic mooring model and it worked fine. After systematically deleting everything systematically, I discovered the issue was caused by the clock state (`currentTime`) updating every second.

```
useEffect(() => {
  const interval = setInterval(() => {
    setCurrentTime(days);
  }, 1000); // update every second
  return () => clearInterval(interval);
}, []);
```

The component was continually setting `currentTime` using `setInterval`, which triggered 1-second re-renders. On Android, this caused the date and time picker to reset its displayed value to match the current system date and 12 pm time whenever this component re-rendered. This overrode all user input, making it useless. To fix this I removed it completely. I deleted the states linked to it as well as the logic. Once this was done the date and time pickers would work as expected and users can freely select any date or time without it snapping back to 12pm.

Android date time picker issue 2

When I first got the Android working to what I expected to be good enough, I sent it to a few close Android users before sending it for my test trial. They reported that the pickers on the manual check-in screen seemed to be disabled because the touchable hitbox was extremely small. This made the UI feel unresponsive or disabled.

The issue was caused by how the `<Pressable>` overlays the text input components. On Android, the picker area didn't cover the visible input so taps outside didn't trigger the picker. To fix this, I wrapped each `<TextInput>` in a `<View>` to handle the layout control more cleanly. I applied a `StyleSheet.absoluteFillObject` to the `<Pressable>` overlay. This made it cover the input field completely, and I also added temporary borders to help visualise the hitboxes.

After adding these changes, the hitboxes now fully cover the input fields, so users can open the time and date pickers freely without any issues.

iOS manual checkout. Submit button falls off the screen

On iOS, I noticed that on small enough devices, when the time picker was open, the confirm button would fall below the navbar not allowing users to select the time they wish. This happens because the pickers expand downwards, pushing all their elements below them, including the confirm and cancel buttons. On iOS, the date and time picker would render inline and take up vertical space. This increased the vertical height as it was already high with the checkout date box and dropdown, pushing the buttons below.

To fix this, I ended up hiding other content on the screen while the time was open. When the time picker is open, the checkout date box is hidden, as well as the text that cancels the margins linked to it. This allows the checkout times confirm and cancel buttons to display all within the screen. Once a user confirms or cancels the picker, the hidden content is shown, returning the page to normal. I also wrapped the submit button in a condition, whenever the checkout date or time pickers are open, the submit button is hidden, as you can slightly see if off the screen, making it unprofessional.

Map centre button. iOS.

Also, I discovered the map centre button on iOS devices would only centre the map on the user's location taken from the first location on startup. Pressing the centre button would take them to the first place they opened the app in the current session. This does not help users find their vessel; it only makes it harder and more confusing.

I initially suspected it was caused by the device's caching or location permissions; however, while reading the code carefully, I found the problem in the `handleCenter` function.

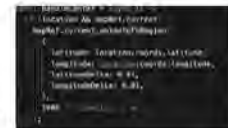
The cause was that the function always relied on the location state at app launch. Because it was not updated asynchronously before the button was pressed, the map would only centre at the first recorded location. I altered the function to be async to fetch the latest location data before animating to the map. This change ensures the centre location follows the user's most recent location, allowing the app to update correctly at any time during the session.



Before



After



Android submission:

Submitting to the Play Store takes a lot longer than publishing to the App Store due to the multiple testing stages required before the app can go public. For testing, I created a closed testing group, which is a requirement for public release. Each time I had a new version, I would have to manually upload the app file to the closed track for a review (usually takes a few hours) then the testers would be able to update their app. Testers were manually added by email, and they must be signed into the same Google account associated with that email. Then they can access the app via a link. Once testers received the link, it directed them to the Play Store for my early release app. After successfully gaining 12 downloads within this closed testing group, I would become eligible for public release. Google then requires a 14-day review period once the user threshold is met. At the time of writing this report, the user threshold was met, and the app was officially published on the Play Store.



Android Account creation

On Sunday, 5th October, I created all the Android users. This completed the other half of my user base. I took longer to get the Android version ready as I had to individually configure each time picker and components specific to Android devices. I followed the same process and emailed the users with their passwords with the python script.

Android API map crash.

When finally submitting the app to my users to test on Android, I ran into a massive error that cost my testers a full day of no functionality. The map screen would show white. However, this immediately told me it had something to do with the API, as the Google logo still appeared in the bottom right. This issue was a little bizarre, as on my simulators, it would show up. I needed a way to identify the issue of the actual app downloaded that's when I can view the logs of the production-built app on a real device. After enabling a few developer settings, I could plug the phone into my computer to view the phone's real-time logs in Android Studio logcat. When I opened the map, I saw this message pop up.



Authorization failure. Please see

<https://developers.google.com/maps/documentation/android-sdk/start> for how to correctly set up the map.

Error requesting API token. StatusCode=INVALID_ARGUMENT

This confirmed this was an API key validation error and that the maps SDK could not authorise my API key for the app. I looked in my Google Cloud Console and found my current key was restricted only to the SHA-1 fingerprint for my local debug key. This means it works fine for local builds and testing on my own device. But when uploading the app to Google Play, Google Play app signing resigned my AAB with a different release of the SHA-1 Fingerprint.

This caused the maps SDK on the user's device to try to validate the API key against the release SHA-1 fingerprint, which wasn't listed in the API key restrictions. This results in the map failing to load, producing the errors in the logcat



To fix this, I found the correct SHA-1 fingerprint in the Play Store files and added it to the API restrictions in the Google Cloud Console. Once this was complete, I had to push a new update, and after that, all maps worked from then on and were displayed. This then allowed all the Android users to test the app, which then led to overwhelmingly positive feedback.

Conclusion

Next steps

With this project mostly complete from the programming side of view, the next step is to push it out to the entire club if they are willing to take it fully on board. As there are 3 clubs that share the club moorings in the Marlborough Sounds, this will create hundreds of potential users if they are willing to use it, or if the club makes it mandatory for all users.

If this is the case, I would need some sort of compensation for my work in the future, including additional features and fees. I am considering a charge of \$5 per user, which the club would pay and add to the users' club fees. This would cover my costs as well as paying for ongoing maintenance, bug fixes and time spent adding or improving additional features. If the clubs take it fully on board, upgrading the Firebase plan will be essential to allow for a higher number of reads and writes, as well as performance for users.

Future features

While I am extremely happy and grateful for this outcome, I have identified some potential next steps to complete in the future. -

- With the service poor in sheltered areas around the Marlborough Sounds region, the app becomes functionless as it requires an internet connection to communicate with the database. I have investigated ways to send occupancy data to the database in these areas. One way is that because every boat has a VHF radio, users could take advantage of this and broadcast over the channel to a base station, reading the signals and updating the database. Or users could manually speak their occupancy status over VHF and have an NLP listening on the other end, converting it into text, then interpret it and update the database. However, with some of the users being from the older generation and Starlink and similar internet providers becoming more accessible, more and more boats are becoming equipped with Wi-Fi, so in the years to come, this will not be an issue as retirement becomes a priority for some.
- A way to cater to multiple vessels on each mooring. As this is allowed by the club, having a way to identify the limit per mooring, as each mooring has different weights it can hold. A potential solution could be when users first login it asks for their vessel length and then it automatically filters out mooring that they are unable to use as well as adding up the total limit on each mooring so if it reaches threshold then the app informs the user the mooring cannot handle any more vessels, sending them away and reducing the risk of mooring drag and breakage.
- Background location with push notifications. While this app has been functional during the testing phase, to improve it would be to add background location functionality with push notifications. This will not require the user to have the app open anytime they want to check in, with background locations, which happens without the user having to manually act, resulting in a smoother check-in process. And in conjunction, push notifications will add usability by asking the user if they wish to check in or informing the user that the app has checked them in automatically.



Data coverage



- If IoT sensors are integrated into every mooring, then it eliminates the need for each user to grant location permissions. These sensors could communicate directly within their own network and send data to the database. Providing a more accurate method by relying on individual devices' GPS data.

Final remarks

The development and success of PWM Moorings have far exceeded my expectations. In the past year, I have gone with zero experience in mobile application development (with minimal knowledge in JavaScript) to developing a cross-platform app published on the App Store and Google Play Store, globally accessible. Currently, the active user base sits at around 50 closed users who have voluntarily signed up to be a part of the trial via the club's weekly newsletter, ClubBites. These test subjects have been actively testing and providing feedback throughout the project. Clocking in with over 70 automatic and manual check-ins recorded in the database. As well as reaching 57th at one point for top free navigation apps in New Zealand, marking a significant milestone in the project, passing popular apps such as Garmin Drive, Garmin captain and a few places behind eRoads. And to top it all off, earning the top search result for searching up "moorings" in the app store.



Club Bites



This project has taught me how to deal with big projects and about both technical development and effective project management. I've learned how important it is to plan, stay organised and manage my time effectively to meet deadlines I set myself to keep the project on track. Working alongside genuine stakeholders, including several members of the Waikawa boating clubs' executive committee, was a completely new experience for me. Collaborating with them as well as other club members gave me a valuable understanding of how to gather, interpret and apply user feedback to improve the final product.

Turning their feedback into real changes helped me see the importance of getting users involved throughout the development process and considering multiple perspectives. I'm extremely grateful to the club for their ongoing support and for allowing me the opportunity to work on a real project that benefits their community. This whole project has strengthened both my technical skills but also my ability to manage large and collaborative projects. Through regular phone calls, early morning text messages and hundreds of emails with stakeholders, I've greatly improved my communication skills and learnt how to collaborate effectively in a real-world project environment.

User Feedback



Statement from the club

After accepting the role of a [REDACTED] The annual general meeting of the Waikawa Boating Club I was duly handed [REDACTED] project and was asked to follow through.

After a couple of discussions with [REDACTED] and seeing the proof of concept I found that his project had merits. [REDACTED] then went onto the implementation phase of his project. I then solicited volunteers from the Waikawa Boating Clubs cruising members and invited them to join in on [REDACTED] project.

[REDACTED] has since released an IOS operating system and an android operating system for members to trial with successful results.

The next phase of [REDACTED] project will see him tackle the idea of royalties and a revenue stream, I personally see great benefits for not only the three clubs and in the sounds but also clubs nationwide and globally to use his booking system for moorings and the attached pop-up advertising etc revenue streams possible. There is no reason why this could not go Global.

From all of us at the Waikawa Boating Club we wish [REDACTED] all the best in his further pursuits.

Warmest regards,

[REDACTED]

Future implications

This project has the potential to extend way beyond its current use within New Zealand. The entire concept behind PWM moorings is that it displays real-time occupancy of shared maritime resources, which can be easily adapted and scaled globally. In an email exchange between the Waikawa Boating Club committee, [REDACTED] mentioned that if executed carefully, this system could be adopted in regions such as the Caribbean. This demonstrates that the idea has genuine international potential in the wide maritime industry.

Also adding to the statement from [REDACTED] while talking to the [REDACTED] she expressed her interest in this project and the potential expansion it could impose within New Zealand and the potential to find capital backing in the future.

The tech behind the app is dynamic and designed for scalability. I ensured minimal hardcoding so new moorings or entirely new locations can be added with simplicity, making it straightforward to deploy in other clubs or regions. Once the app is fully robust, my next step would be to reach out to other boating clubs around New Zealand and then clubs worldwide to explore collaborations and gain insight into how their systems work and how I could adapt my system to see the best fit.

Beyond its current use with moorings, this technology could be adapted into other applications within or beyond the maritime sector

- Marina berth management: Basically, the same system, but instead of moorings, it manages berth occupancy in marinas, allowing vessels to see what berths are free, book spaces or even notify the harbour master automatically when occupied.
- Book a batch for moorings: Another future concept is a booking system that works like book a batch, but for moorings. Development is already underway, and the idea remains confidential to protect intellectual property and to secure a first-mover advantage in the market.
- EV charger and parking systems: Companies like ChargeNet have already developed EV charger occupancy data through charging networks; however, a new potential could be a GPS-based system in car parks to live monitor space tracking and occupancy.
- Commercial shipping: The same principle as the moorings, but applied to manage anchorage zones or loading bays, leading to a decrease in congestion and improved efficiency.