Activity Tracking Application (Motion)

Gerard McKendry

N00220460

Supervisor: Catherine Noonan

Second Reader: Mohammed Cherbatji

Abstract

The aim of this project is to construct a mobile application using a cross-platform development framework which also uses features that are exclusive to mobile devices, making something unique and unlike previous web development projects completed during the course. To this end, the project is an application built using React Native using the Expo framework, and uses the location features of a mobile device. The application itself, named Motion, is an activity tracker application that allows the user to track their activities, view them later, and share them with their friends. The purpose of the application is to enable users to share exercise data and improve their fitness together with their friends. The steps involved in the development of the system included getting the mapping and location tracking software to work, creating links between views of the application that would make it simple for users to navigate, providing a friends system for users to add and see the data of their friends, and adding a backend that ties it together with a connection to a database that would store all of this data. Testing was carried out throughout and after implementation. Results from the testing show that the app does have a few breakage points that would need to be resolved, but that the core idea of the app works overall. Further work that could be carried out includes stronger incentives, such as the addition of an achievements system to encourage users to get out and record activities, as well as more ways for friends to interact with each other within the application.

Introduction and Project Context

Project Aim & Objectives

The main aim of this project was to gain a better understanding of how mobile applications are developed, as this was a topic that was not addressed in the course to this point but was a point of interest. Thus, the project became focused on building a mobile application using a specific type of framework known as a cross-platform development framework. This refers to a framework that can be used for building an application that works on multiple platforms, in this case, multiple models of phones, on a singular codebase. In particular, it felt pertinent that the application involve some kind of interaction that was unique to mobile devices and couldn't be implemented into a desktop application; examples of this would be location tracking, the camera (likely usage of QR codes), touchscreen gestures, the accelerometer and gyroscope, and more. After some deliberation, it was decided that the application would lean towards the implementation of location features, and thus the idea for an application to track user activities became the focus of the project.

Success Criteria & Scope

- Create a mobile application using a cross-platform development framework
- Make use of at least one feature that is applicable only to mobile devices, in this case location features
- Gain a greater understanding of the development process behind mobile applications
- Make use of a database that users could upload their activities to in order to store them

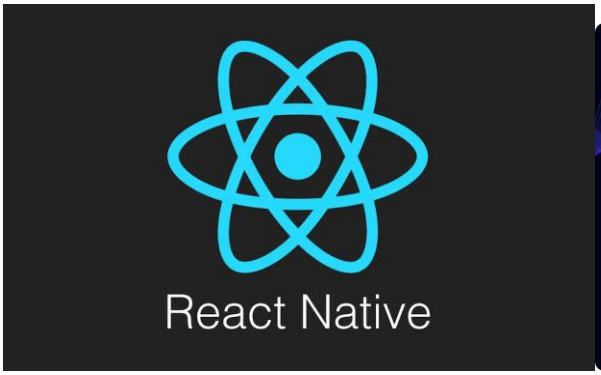
Research and Background

Literature Review

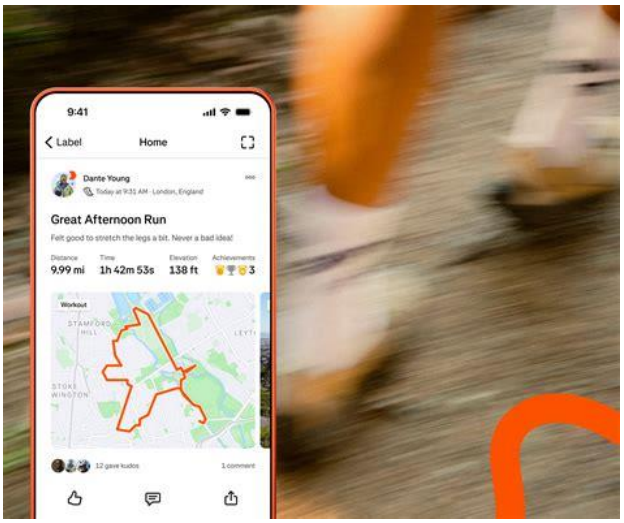
Before starting this project, it was vital to do research into the area the app would be built in. The goal of the project was to make a mobile application, but the type of framework to be used hadn't been finalized. Originally, research was done into building Android-specific applications, using Android Studio and the Kotlin programming language. However, this plan changed after doing research into cross-platform development frameworks.

Cross-platform development is a development style that allows for an application to be built for multiple platforms at the same time. There are many benefits to this approach. For one, it means that developers are working on one singular codebase, rather than having to maintain parity between multiple platforms. Developing for Android and iOS using their own respective frameworks means that developers have to be aware of the individual solutions for each platform; Android applications are programmed using Android Studio and the Kotlin programming language, while Apple recommends that iOS applications be built using XCode and SwiftUI. On top of this, XCode requires the usage of the MacOS operating system, meaning that having a device running that OS is a prerequisite. Cross-platform frameworks avoid this hassle. Another benefit is that changes made apply to the app on both platforms, rather than introducing unnecessary work having to implement the same changes for both platforms.

At this point there was still the question of which cross-platform framework to use. The main frameworks for consideration were React Native and Flutter. React Native is a framework that bridges the gap between Facebook's React framework and the development of native apps for mobile devices. It allows for the development of mobile apps using JavaScript and TypeScript, which makes it appealing for developers that are already familiar with those languages. It has been used for the development of applications such as Facebook, Microsoft Outlook and Teams, Discord, and Shopify. Flutter, on the other hand, is developed by Google and uses the Dart programming language. Because of its ability to use its own engine to render widgets rather than relying on web view or OEM widgets, it is highly performant compared to other frameworks. After researching this, the decision was made to make use of React Native for this project.



At this point, research was also done into similar existing applications. The app with the strongest similarities to this idea was Strava. Strava is an application that allows for the tracking of activities, sharing activities with friends, collecting virtual badges when achieving certain milestones, integrating with external devices, and more. This is where the inspiration came from to be able to share your activity data with your friends in Motion.



Technical Research

At this point, research had to be done into the specific technologies that would be used in the development of this application. React Native had been settled on as the framework of choice, so I did research into that first. React Native has a guide on their website that explains how to test your application on a physical device, by connecting it to your computer via USB cable and running it via the command line. However, this turned out to be unnecessary. The React Native docs also state that the usage of a React Native

framework is recommended for app development, and points to Expo specifically as a strong choice. As such, Expo became the React Native framework of choice.

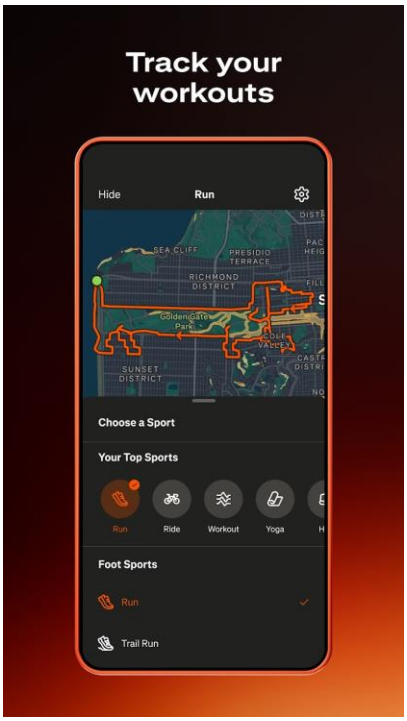
While it was essentially a certainty that React Native would give the tools needed for the development of Motion, it was still good to do research and make sure that I had knowledge of those tools before going straight into developing the application. Specifically, I wanted to figure out what the packages I would need to use would be. The relevant packages found were “expo-location” and “react-native-maps", though react-native-maps would later on be substituted for “expo-maps" after some technical difficulties. expo-location is a package that allows for the fetching of a user’s location, and also allows for background tracking, so the app can continue tracking the user’s location even if they close the app during an activity. expo-maps is a package that allows for integration with Google Maps and Apple Maps, providing a way to display the user’s location as they move around, and display the route that they took during an activity after the fact.

Requirements Analysis

Requirements Gathering

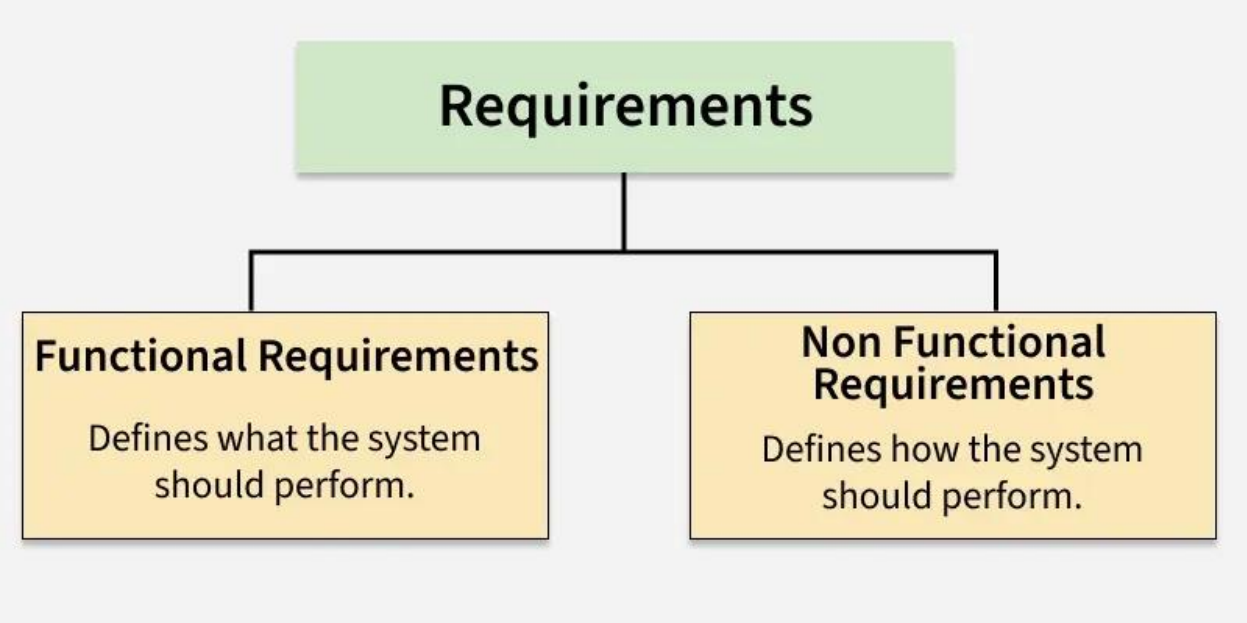
Similar applications

There are a number of applications with similar premises to Motion, but Strava is by far the most popular similar activity tracking application. It provides similar tracking features to what was planned for Motion, allowing users to track their activities as they walk around.



Strava, as a fully featured application, has a wide range of other features too. These include linking with external devices such as smartwatches, detailed reports taking into account multiple activities and showing progress over time, badges to show achievements that the user has obtained, more community integration such as allowing friends to comment on each other's activities to congratulate them, community competitions, and more. Motion, as a simple prototype would not be able to come close to being as featured as Strava, but some inspiration was taken from it, such as displaying users' accumulated stats on their profile and allowing friends to view each other's activities.

Requirements Modelling



Functional requirements

Functional requirements are the functions that the application must be able to perform to meet the needs of the user.

- The user should be able to register an account and log into it.
- The application should be able to track the user's location during physical activity.
- It should store the route that the user took, as well as the start time and end time of the activity, and upload them to the database.
- The user should be able to access a list of the activities they previously uploaded.
- The application should be able to display the route that the user took during the activity on a map.
- The user should be able to send and accept friend requests to and from other users.
- A user should be able to see their friends' activities, but should not be allowed to see the activities of users that aren't their friend.
- A user should have statistics on their profile that are visible to their friends that act as a metric of how active the user has been.

Non-functional requirements

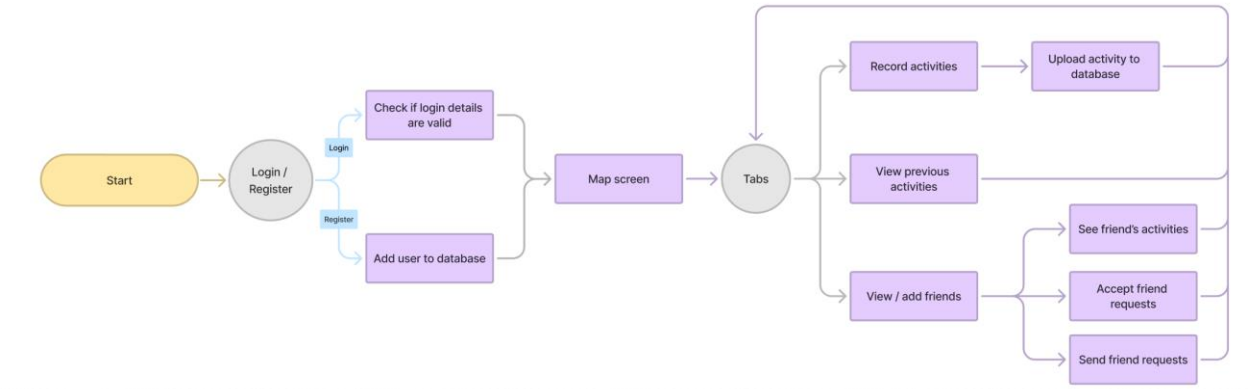
Non-functional requirements refer to requirements that are not technically necessary for the app to function, but still matter for the security and reliability of the app, making for a better and safer user experience. Even though these features do not strictly make the application function, they are still vital.

- The users' details, especially passwords, should be stored securely, in particular making use of a hashing algorithm to obscure passwords from potential hackers.
- The app should be performant, responding quickly to user input with no noticeable hitches or drops in performance.
- It should communicate quickly and effectively with the database without hanging for significant periods.
- It should be scalable, with a design and codebase that gives way to the addition of new features.
- It should give the user detailed error messages when something goes wrong, helping them to diagnose or avoid the problem in future.

Design

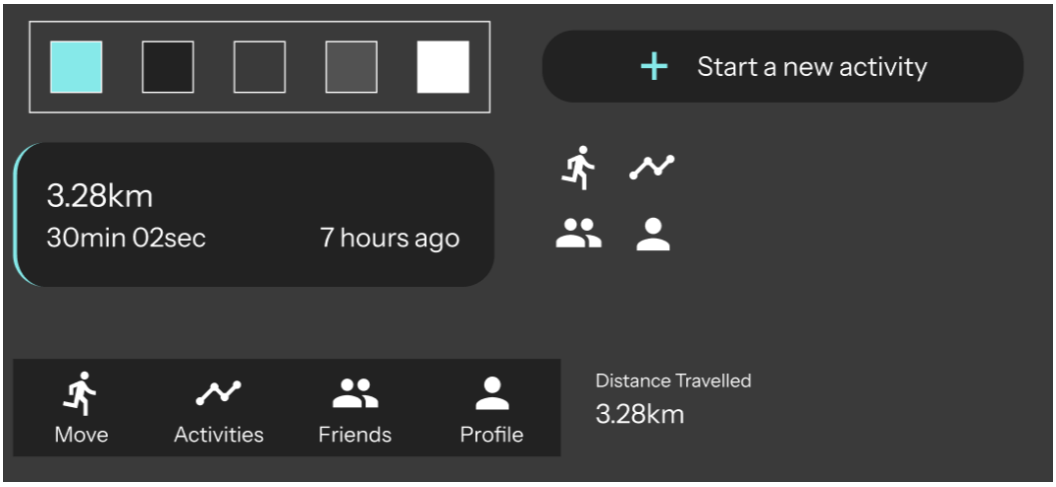
Interface Design

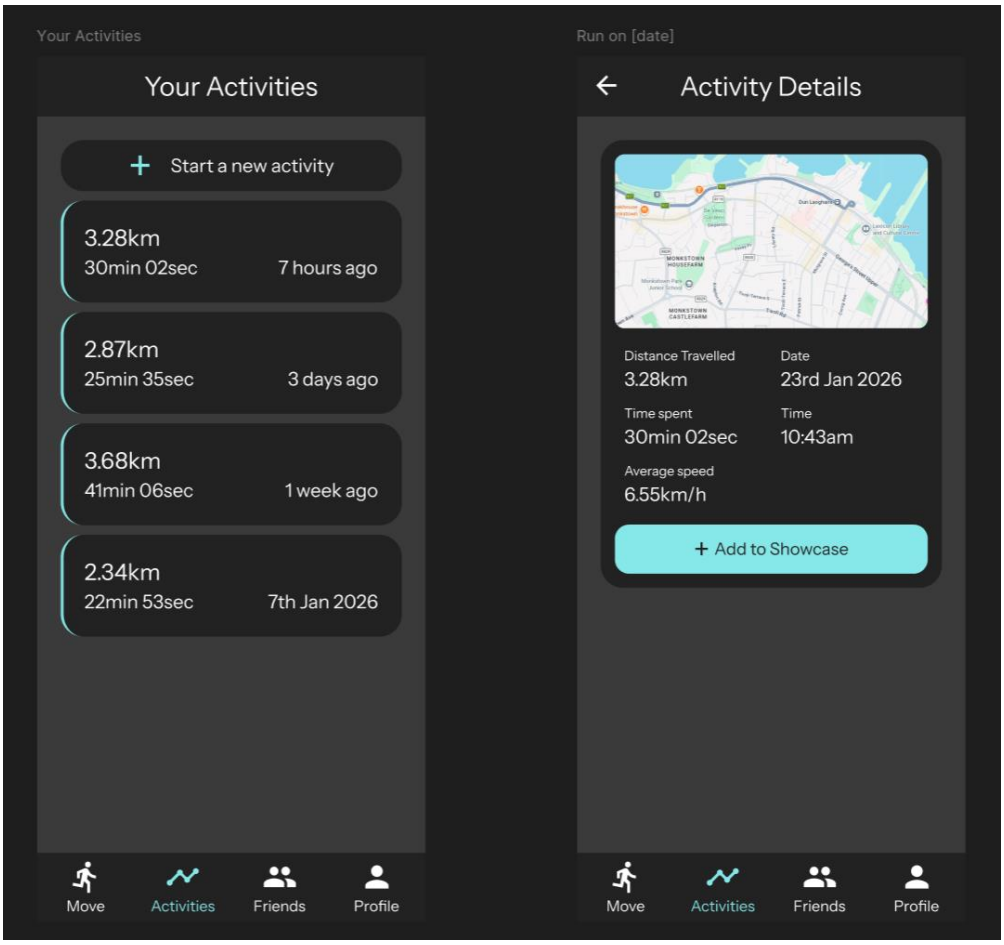
User flow diagram



Above is a user flow diagram that lays out how a user's experience with the application would typically go. The "start" on the left side is when the user opens the application, and then are prompted to login or register. Depending on the user's choice, they'll either create a new account, or have their login details checked against the credentials in the database. Then, the user is taken the map screen by default. From here, the user has open access to the application. They can choose to start doing activities, in which case they'll be added to the database. They can also choose to view previous activities or see their friends list and friend requests. From the friends list, they can view their friends' activities, or add new friends on the friend requests list.

Final Wireframes





Process Design

Libraries used

Motion is built using React Native, using the Expo framework. Expo is a framework that provides file-based routing and universal libraries that are a significant aid while developing an application. The application uses several libraries that are compatible with Expo:

- expo-maps: A library that allows for maps to be displayed using the Google Maps and Apple Maps API. These maps can display the user's location, display "polylines" between sets of coordinates, and be styled using Google's JSON map styling wizard.
- expo-location: A library that provides multiple functions for interfacing with the phone's location tools. This library allows the application to request location permissions on the device, get the current position of the user, set up a task that gets the user's location repeatedly, and has other features not used for this project such as geofencing.
- expo-task-manager: Handles the repeated nature of the location-tracking task set up by expo-location.
- axios: Allows the application to make API calls to the backend server, which in turn modifies the database.
- react-native-async-storage: Allows for the storage of persistent data such as details about the user's authentication state in device storage.

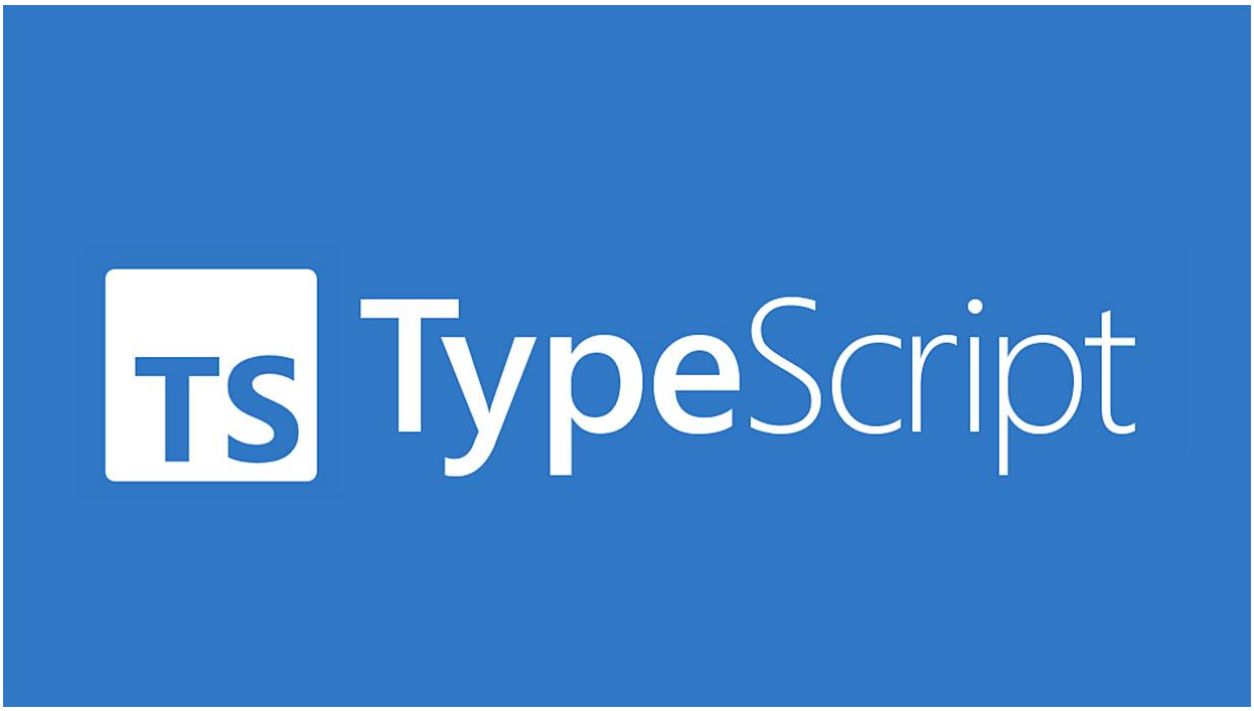
In addition, the backend uses several libraries for its functionality:

- express: This is the main library used, allowing for the running of a server that can listen on a specific port and accept API calls made to that port. It makes use of a router system that reads the URL given and tries to match it to a route and a HTTP method to give the desired behaviour.
- mongoose: Allows the server to interact with the database. Models are defined via schemas that contain each of the columns in that model, then data can be written to and fetched from the database using those models.
- dotenv: Allows the server to read values from the .env file.
- express-validator: Acts as a bridge between express and validator.js, allowing data to be validated through the use of middleware as it is sent to a route.
- bcrypt: Allows for the hashing of users' passwords before being added to the database.

TypeScript

The application primarily uses basic JavaScript on the backend, but on the frontend, it uses TypeScript. JavaScript is an incredibly widely used language, with its loosely typed nature making it an easy language to start learning with and making development a breeze. Despite this, it does have numerous issues, many of which are related to said loose typing. JavaScript does not allow developers to lock variables to a specific variable type, meaning that oftentimes you may encounter bugs because a variable is cast to a different type than is expected. As a simple example, a developer may try to use user inputs to take two numbers to add together. A user can input “2” and “4”, but when the program attempts to add them together, it assumes that the text inputs are supposed to be strings, and JavaScript uses string concatenation to add the strings together, resulting in an answer of “24”.

TypeScript addresses these issues by adding static typing to JavaScript. This allows developers to add type annotations to variables, locking them to a specific chosen type. In the previous example, the developer could mandate that the “answer” variable be a number. When trying to add them together, the TypeScript compiler would notify the developer that adding two strings together cannot return a number, showing them exactly how the error is being caused. Because of this, writing TypeScript code can take slightly longer, occasionally requiring the developer to create new types to satisfy the strongly typed nature of the language, but it saves a significant amount of time through highlighting the causes of unintended behaviour.



Event Handling

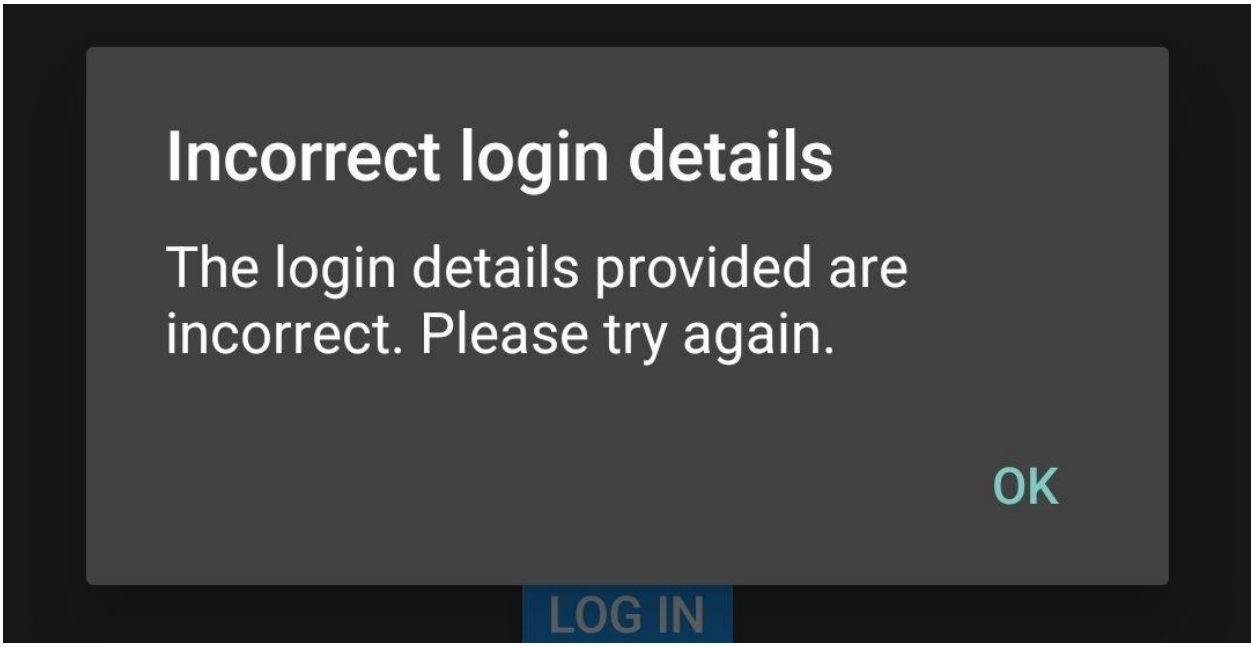
React Native allows for the handling of events through various hooks. One of these is the "useState" hook. A variable can be defined as a state variable using the useState function. This splits the variable into a variable representing its current state, and a "setter" function, which can be run to set the new state of the variable. When setting the variable using this setter function, React Native re-renders the component it's a part of, causing the view to reflect the new state of the variable.

Another example of how React Native handles events is using the "useEffect" hook. This hook allows you to synchronise a component with an outside system. It contains a function and a dependency array. Whenever one of the dependencies is changed, React runs the function again. With an empty dependency array, the function runs just once when the component is loaded. This makes it particularly useful for running Axios requests that should only request data from the database one time, when the component is loaded. This is a programming pattern that is used in many views around the application.

Error Handling

It is ideal for the application to be able to handle errors gracefully. Try-catch blocks are used throughout the codebase to attempt to run code that has a danger of failure. In particular, it's often used around code that runs Axios requests, as the backend can return

HTTP errors that can be caught and handled on the frontend. Use can also be made of React Native's "Alert" dialogs to display error information to the user. The application has a showErrorAlert function that can be imported from a script to quickly show an alert message when an error is caught.



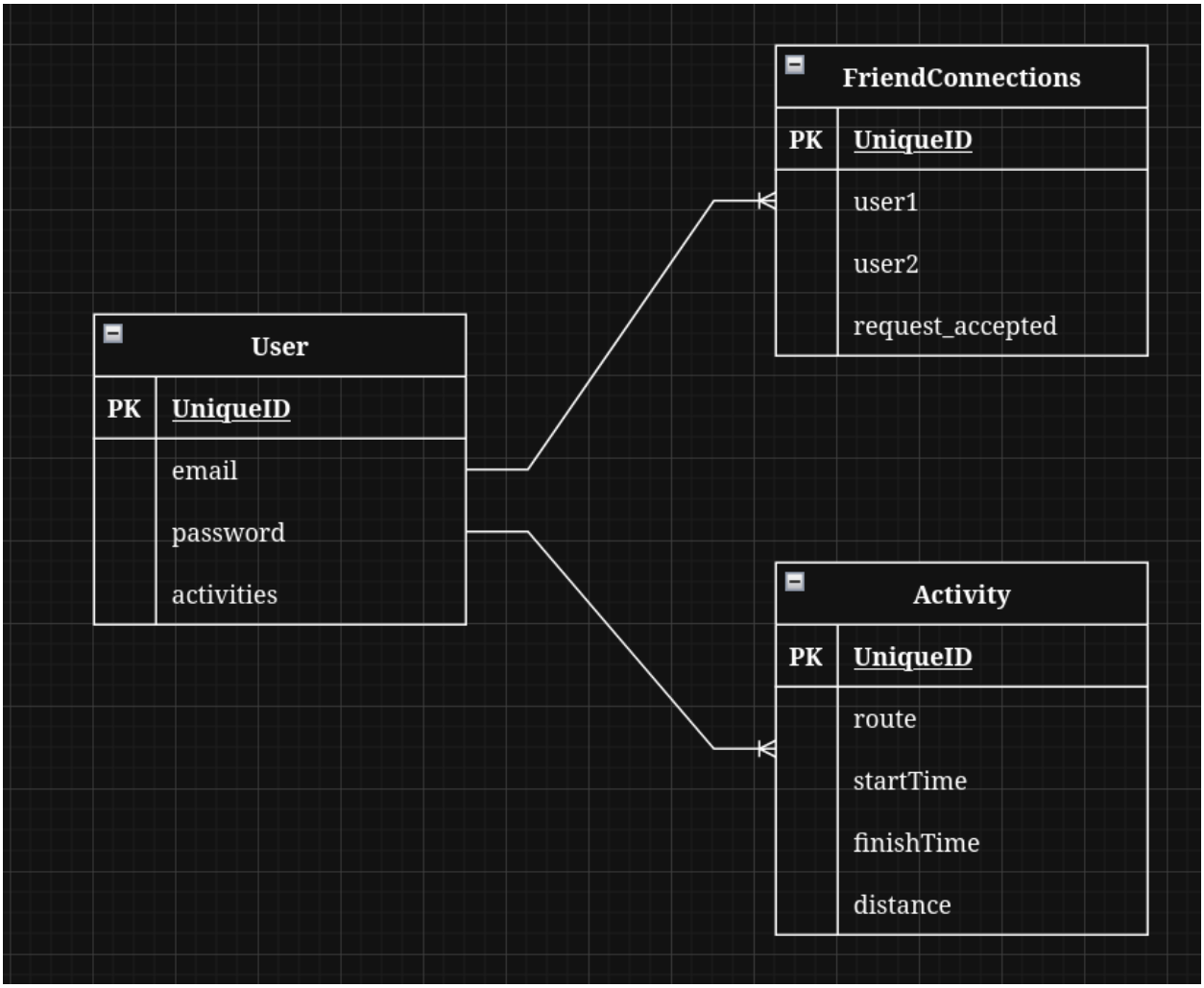
Use of Concurrency

React Native inherently makes strong use of concurrency; as it is a mobile application, it needs to appear and feel responsive, and as such, users don't want to feel like the app is constantly pausing to run code and freezing the app in the meantime. However, time is also needed to perform database queries and other actions that can take some amount of time to finish. For this reason, asynchronous functions are used for these time-sensitive operations.

Async functions in JavaScript always return a Promise, which is an object that represents the eventual return state of an async function. We can use the "await" keyword with these functions to force the program to wait until the Promise has resolved before continuing with the resolution of the outer function. This means that we can wait for the database query to complete and for the backend to return a response before continuing with execution, ensuring that values are not set incorrectly by running too early. Typically in this case, it's best to display a loading indicator until the function is completed before showing the content that was fetched.

Database Design

Entity Relationship Diagram



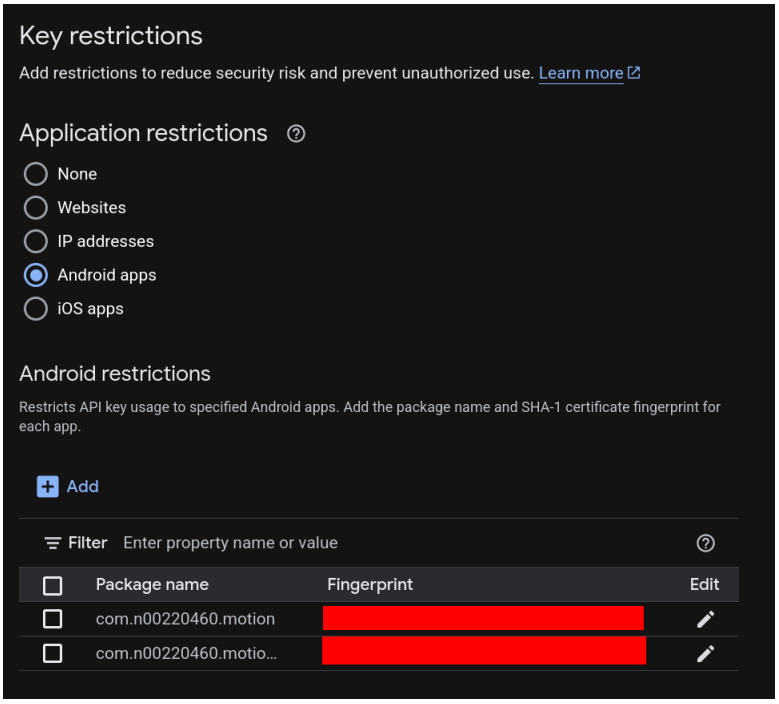
Shown above is an ERD, or entity relationship diagram, showing the initial plan for the database. The database was to be built using MongoDB, a NoSQL database solution. Had an SQL database been used, the one-to-many relationship between the User and Activity models would have required a “UserID” foreign key in the Activity model. Instead, MongoDB allows for an “activities” array to exist within the User model, containing a list of all the user’s activities. Later on, this database design would change with the addition of a “speed” property in the Activity model, the User model would have a username, activityDistance and activityTime properties (which track certain stats about the user over time), and the FriendConnection model would be split into two models, FriendRequest and FriendConenction. This would be to simplify database queries by not always having to specify whether a connection had been accepted or not to get data about requests or connections.

Implementation

Major Challenges & Solutions

Google Maps API + react-native-maps

Trying to use the Google Maps API alongside the react-native-maps library was the source of significant technical difficulties near the beginning of the project. react-native-maps allows for a map to be displayed using the MapView component. This works by default in Expo Go, but doesn't work when a development build of the application is made. Instead, a development build requires a Google Maps API key. The API key needs to be placed into the app.json file and a variable from react-native-maps named PROVIDER_GOOGLE should be passed into the MapView component as a property. However, this did not work even after repeated attempts to fix it. Worse still, react-native-maps did not supply an error message of any kind, instead just showing the map as a grey screen. This meant there was effectively no feedback as to what exactly was going wrong and how to fix it. I switched away from react-native-maps and tried using expo-maps instead, and eventually managed to get the map to display correctly. This did require storing the API key in the app.json file, which is a vulnerability that should be noted, but it was the only solution found after a long time of debugging.



IconSymbol Mapping

The default solution that Expo comes with for displaying icons on the screen is the IconSymbol component. This is a component that takes a symbol name, size, colour, and style, and displays an icon on the screen. The tabs bar that Expo comes with also uses these IconSymbols to display the tab icons. However, using an icon is not as easy as finding a list of icons and typing the name of the one you want to use. Opening icon-symbol.tsx, you can find a MAPPING object.

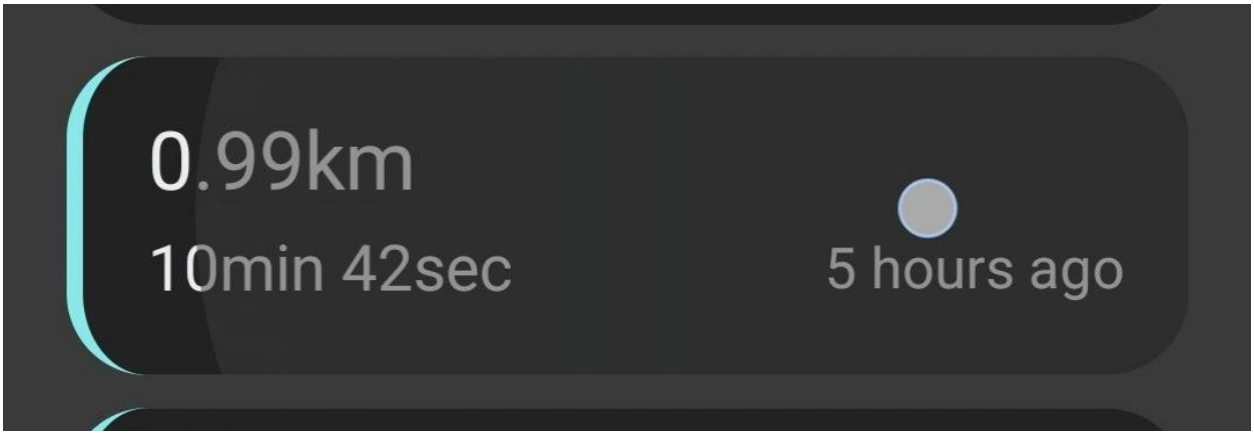
```
/**
 * Add your SF Symbols to Material Icons mappings here.
 * - see Material Icons in the [Icons Directory](https://icons.expo.fyi).
 * - see SF Symbols in the [SF Symbols](https://developer.apple.com/sf-symbols/) app.
 */

// using IconSymbols requires them to be mapped here
// the iOS symbol is on the right, android is on the left
// you cannot use the SF Symbols app listed above unless you have an apple device
// so i used this repo to find them instead https://github.com/andrewtavis/sf-symbols-online
const MAPPING = {
    "house.fill": "home",
    "paperplane.fill": "send",
    "chevron.left.forwardslash.chevron.right": "code",
    "chevron.right": "chevron-right",
    "person.fill": "person",
    "figure.run": "directions-run",
    "chart.bar.fill": "show-chart",
    "person.2.fill": "people-alt",
    "person.badge.plus.fill": "person-add-alt-1",
    checkmark: "check",
    cross: "clear",
    trash: "delete",
} as IconMapping;
```

This object contains a list of key-value pairs; the key is the “SF Symbol” name, which is the system that Apple uses. The value is the “Material Icon” name, which is the system that Android uses. To map the symbols, you need to find an SF Symbol that maps to a symbol from Material Icons, ideally an icon that conveys the same meaning. As is stated in the second comment in the screenshot, the URL provided in the first comment links to a website asking you to install the SF Symbols application that is only available for MacOS to be able to see the SF Symbols. I do not have access to a computer running MacOS making this infeasible. After searching, I eventually found a GitHub repository created by Andrew Tavis that lists the SF Symbols without the need of an Apple device. Using this, the mapping can be created, finding the Material Icons name from the other link provided in the component.

Pressable components and AndroidRipple

React Native comes with both a Button and a Pressable component, which each having a prop allowing for the execution of a function when pressed. While the Button component is simple to set up and use quickly, Pressable allow for a much larger range of customisation options. In particular, Pressable has access to the “android_ripple” prop, which produces a circle of colour that spreads outwards from where the press was registered, giving strong feedback to the user that their press was registered. This prop accepts a RippleConfig option, which itself accepts various values to customise the ripple effect. The first issue with the ripple effect was that it simply wasn’t appearing, though this was fixed quickly by setting the “foreground” property in the ripple config to be true.

The next issue was more tricky to figure out; most of the components in Motion have rounded edges, and this effect is achieved by using the “borderRadius” property in the stylesheet variables. The problem this caused was that the ripple effect ignored the rounded corners completely, flowing outside the component and filling it as if it were fully rectangular. The “borderless” property in the ripple config did not affect this. The solution was applying the “overflow” style of “hidden” to the component’s styles, meaning that anything that went outside the border of the component would be fully hidden.



User appears logged in falsely

The way authorization is programmed using Motion, the details about the user’s authentication state are stored in the AuthState context, and provided via the AuthContext.Provider component. This contains functions related to authorization, but also contains the user’s unique ID, that is used to validate the user’s identity. If the user doesn’t have an ID assigned to them, the base layout file redirects them to the login view.

To store the user's identity persistently, the user's ID is stored in device storage using AsyncStorage, and loaded whenever the app is opened.

The problem that arises from this is that the user would open the app, the AuthProvider function would load the user's ID from memory, and then grant them access to protected routes, even if the user no longer has a valid session. From this point, any database queries that are made would fail, because the user doesn't have a session and as such is not considered to be logged in by the database. To fix this, after getting the user's ID from storage, an Axios request is made to the "api/auth/me" route, which can check the user's authentication state. If the server returns the response that shows that the user is not authenticated, the userId is cleared from storage and the user is prompted to log in again to restart their session.

```
// when the app is loaded, see if the user had a previously active session
const getAuthFromStorage = async () => {
    try {
        const value = await AsyncStorage.getItem(authStorageKey);
        if (value !== null) {
            const auth = JSON.parse(value);
            // check if the user is still authenticated
            // if not, clear auth variables and redirect to login
            const response = await authService.getUser();
            if (response.authenticated === false) {
                setUserId("");
                storeAuthState({ userId: "" });
                router.replace("/login");
                return;
            }
            setUserId(auth.userId);
        }
    } catch (error) {
        console.error("error getting auth state:", error);
    }

    setIsReady(true);
};
```

Validation using express-validator

Multiple issues arose with the usage of express-validator while trying to use the backend routes. When an Axios request fails, you can handle it as an Axios error specifically by using an if statement on the catch block checking "if(isAxiosError(error))". However, while this allows for the extraction of a HTTP response code, after debugging for a while I could not find a way to extract the error message that is set in the validator object. This meant

that whenever a request would fail at validation middleware, the only information that could be extracted about the source of the error was a “400 Bad Request” response.

This was particularly an issue while working on account registration. The express-validator documentation does not provide much actual information about how the validation schemas should be set up. When trying to use the “isStrongPassword” validator with only the minLength and minNumber, the program would run into repeated errors while meeting the requirements set. After checking the validator.js page which is documented better, it was revealed that the issue was that the isStrongPassword validator has default options: if the options aren’t provided, it enforces at least one capital and one lowercase letter, one number, and one symbol. To get around this issue, these values were added to the validation schema and set to 0, to override the default values.

Another validation issue that arose was using the “isLatLng” validator. While attempting to send values to the backend, the latitude and longitude values were being sent as an object, and being rejected repeatedly. Again, express-validator does not have good documentation on how this validator should be used, but the validator.js documentation reveals that it is checking for a string with the format “[lat],[lng]” or “[lat], [lng]”. This caused problems with the static typing being used on the frontend, as the Activity type specified that the route should be an array of Coords objects, each of which is an object with “lat” and “lng” properties. Changing this to accept strings so that the object could be uploaded to the backend as-is caused typing errors elsewhere, where the route property was expected to be an array of objects. For a while, the solution used was to have a separate type called “ActivityUploadable” that had the route as an array of strings, but this was more hassle than it was worth. The eventual solution was to just upload the route as an array of objects and do the validation of the route array in the router function itself, rather than trying to force it to work in the validation schema. Once the array is validated, it’s converted to the string format to be stored in the database, and then converted back to an object when it is sent to the frontend.

Coords and Coordinates Objects

During the development of the application, the Coords interface was created to represent a position on Earth’s surface using latitude and longitude, given the keys “lat” and “lng”. This ended up becoming an issue when trying to display polylines using expo-maps. This package’s library contains a GoogleMaps.View component which accepts an array of polylines, which are lines that are displayed between coordinates on the map. These polylines use the GoogleMapsPolyline type, which accept an array of Coordinates objects.

This Coordinates type from expo-maps was incompatible with the Coords interface that had been used for the project thus far, so Coords objects could not be used to display polylines.

The ideal solution at this point would likely have been to go back and refactor the codebase to remove the Coords interface and instead use the Coordinates type wherever applicable. However, due to time constraints, a quicker solution was used. In the utils folder, there is a typeConverters script, that contains a few functions for converting objects between types, There is a converter named coordsToCoordinates, that takes a Coords object as a parameter and returns an expo-maps Coordinates object, and the inverse coordinatesToCoords, that converts a Coordinates object to a Coords object. This isn't the ideal solution and given more time it would have been better to refactor the code written using the Coords interface, but it works for a project of this scale.

```
// take a Coords object and return a Coordinates object
function coordsToCoordinates(coords: Coords): Coordinates {
    return {
        latitude: coords.lat,
        longitude: coords.lng,
    };
}

// take a Coordinates object and return a Coords object
function coordinatesToCoords(coords: Coordinates): Coords {
    return {
        lat: coords.latitude || 0,
        lng: coords.longitude || 0,
    };
}
```

Date Object Abnormalities

When getting activities from the database, there were some problems using the Date objects that were returned from it. JavaScript Date objects have the capability to be cast to numbers by simply placing a "+" symbol before them; doing this returns an integer value representing the milliseconds from the Unix epoch (January 1st, 1970) to that time. Using this, you can do arithmetic operations to find the number of milliseconds between two date objects. However, the date objects being returned from the database did not cast correctly to numbers, returning NaN instead. The solution to this turned out to be to recreate the Date object, by passing it into a "new Date(x)" constructor. After doing this, casting to a number would work properly. This was also an issue when trying to use the "toLocaleString" method to return a human-readable string representation of the date.

Again, after creating a new Date object with the constructor, this method started to work properly.

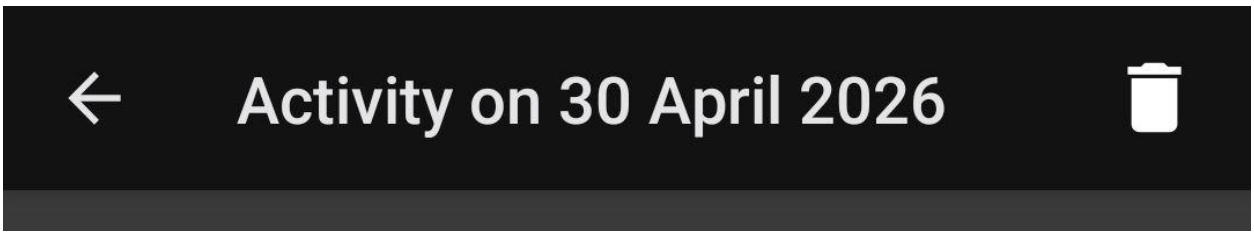
```
function calculateDuration(startDate: Date, endDate: Date): string {
    // javascript requires me to cast the dates as new dates
    // otherwise functions and operations won't work on them
    // for some reason
    const start = new Date(startDate);
    const end = new Date(endDate);

    const timeInMs = +end - +start;
```

Dynamic changing of status bar / header

The status bar at the top of the screen is used to display the name of the screen the user is on, as well as a back button if the screen is not the base route of a tab, and it can optionally contain other buttons or links. These properties are typically defined in the Layout function, which provides a stack navigator with the details needed to display each route. For example, the friends layout function gives an options prop to the Stack.Screen component that provides the name to be shown at the top, as well as a header button to be displayed that links to the /friends/requests view.

The issue with this is that if the desired effect is to display a header using details that are fetched when the screen is loaded, the Layout component doesn't have access to those details. An example of this is in the [activityId] route, where the desired behaviour is to display the date that the activity took place on; this data is unavailable to the Layout component. Instead, the Layout component is given the default title of "Loading activity details...". Then, in the ActivityView component, the "useNavigation" hook is used to allow access to the header's options. Finally, the "setOptions" method on the object created by the useNavigation hook is used to set the options based on the loaded information. In this case, this also includes a Pressable component that calls a function in that View, which would also ordinarily be inaccessible from the Layout component.

```
// sets the title based on the date, and inserts a header button to delete activity
navigation.setOptions({
    title: `Activity on ${activityDate}`,
    headerRight: () => (
        <Pressable onPress={confirmDeleteActivity}>
            <IconSymbol size={28} name="trash" color="#FFFFFF" />
        </Pressable>
    ),
});
setactivityDate(activityDate);
```



Sending Friend Requests

Sending friend requests was one of the trickier things in terms of how it was implemented on the backend. This is because there are many instances in which it should fail to create the request. The call to add a user as a friend should fail if:

- There is a problem with authentication (the request must be sent from an active session)
- The username to add as a friend does not exist
- The recipient and the sender can't be the same user
- The sender shouldn't have an active friend request to that user already
- The recipient shouldn't have an active friend request to the sender (In this case, the ideal behaviour would likely be to automatically accept the incoming friend request, but there was not enough time to implement this properly)
- The users shouldn't already be friends

Only after all of these conditions are checked should the friend request be sent. Thus, each of these conditions needs to be checked for in the database using Mongoose queries to the database.

Running a function in an internal / external component

Throughout the application, ScrollViews are used to render a list of items returned from the database. To make sure that the user always has a way to get the current state of the data,

a RefreshControl component is used. A RefreshControl component allows for the user to scroll up when they're already at the top of the list to initiate a refresh. A function can be specified to run whenever this refresh happens. As an example, the ActivitiesView component has a fetchUserActivities function that fetches the user's activities, sorts them, and puts a list of ActivityCard components onto the screen showing each of the activities in the database. The RefreshControl component in the ScrollView calls the fetchUserActivities function again, meaning that the list will be fetched anew, and displayed again.

A problem arises when this refresh function needs to be called from inside or outside of a component. This is the case in the friend requests view. This view returns a list of friend request cards, each representing a friend request to or from a different user. There are three cases in which we want to refresh the screen:

- When the RefreshControl component is triggered
- When the user deletes a friend request, so it stops showing immediately
- When the user sends a new friend request, so it can be displayed immediately

As the RefreshControl is in the same component as the fetchRequests function, the RefreshControl can simply call that function. Activating this in an internal component is a little bit trickier, but all it requires is passing the function into the internal component, in this case into the RequestCard component, as a prop. This allows that component to access and call the function. The trickiest is calling the function from an external component. To do this, the external component needs to pass a function into the internal component that sets the value of a reference object created by the useRef function. When the internal component is loaded, it sets the .current property of the reference object to the refresh callback function. Once the callback is set, it can be called from the external function by calling the .current property.

```
// sets "callbackRef.current" to the RequestsView refresh function
// this is so we can refresh after creating a new request
// https://stackoverflow.com/questions/66445560/how-does-one-trigger-an-action-in-a-child-functional-component-in-react
function registerCallback(callback: Function) {
    callbackRef.current = callback;
}
```

```
requestList.push(
    <RequestCard
        key={req.id}
        id={req.id}
        type={showType}
        username={
            showType === "incoming"
                ? req.sender.username
                : req.recipient.username
        }
        refresh={fetchRequests}
    />,
);
```

Polyline not displaying on the map view

There was a significant amount of difficulty experienced while trying to set up the map view to display the user's position as they walked around. The idea was to use the same solution as was used for the activity view, but to make it dynamically change by using a state variable. This seemed simple, but after trying to simply pass a polyline into the map component with coordinates from the Route variable, the polyline would not appear at the user walked around. This was despite making sure using a test display that the length of the route array was indeed increasing as new locations were pushed to the array.

The next attempt to fix it was to add a new state variable just containing the polyline, and setting the state of it whenever the route array was changed. A problem was foreseen with this, namely that the value of the state variable isn't changed immediately when the state is set, meaning that attempting to set the polyline based on the value of the route state variable would not necessarily get the true current value of the variable.

To work around this, an alternate method of setting the state was used – by passing a lambda function into the state setter and passing an “prevState” value into that, it could be guaranteed that the prevState variable would be the most recent state of the variable. Then, the lambda function could calculate the new state of the variable, and before returning it, it could do additional operations using it; in this case, setting the state of the polyline variable based on the new state of the route variable. This did not work. Again, when checking via a test display, locations were being added to the coordinated property of the polyline array, but the line was not appearing on the map.

Finally, a different solution was tried. The route and the polyline are stored completely independently of each other. The lambda function is still used for setting the route in case the state variable hasn’t changed before it is written to again. This time, the destructuring syntax is used to append to the end of the route array, rather than pushing a new value to it. The same is done for the polyline, destructuring the previous state and appending the new value to the end of the array. This solution finally caused the polyline to appear on the map.

Keeping a consistent zoom level

This was an issue that an optimal solution unfortunately couldn’t be found for. The expo-maps map display allows by default for the user to zoom in and look around the map. This is a positive feature as it allows the user to swipe around and use touchscreen gestures to see where their route has taken them and where they might want to go next. The issue here is how React Native re-renders components. Whenever it sees a change to the polyline, it has to re-render the map to display the change. When it re-renders, it resets the camera position and zoom level to the values that are specified in the cameraPosition property of the map component. This means that whenever the route is updates, the camera will snap back to the user, which can be annoying from a user’s perspective.

The solution attempted for this was to store the zoom level as a state component. The map component does come with a useful “onCameraMove” property, in which a function can be run using various details about the camera’s current position. The idea was that the zoom level state would be set to the current zoom level whenever the camera is zoomed in or out, and so the current zoom level would always be used as the initial position when the map is re-rendered. However, after trying to implement this, it was found that the camera would display extremely jittery behaviour whenever attempting to zoom in or out. This was because whenever the zoom level was changed, the map would see the state variable change and trigger a re-render, but it would attempt to use the old zoom value instead of the newly set one. This would result in it repeatedly trying to fight the user whenever they try to change the zoom level. After attempting fixes for a while, it was decided to just disable the zoom controls instead.

Development Environment & Tools

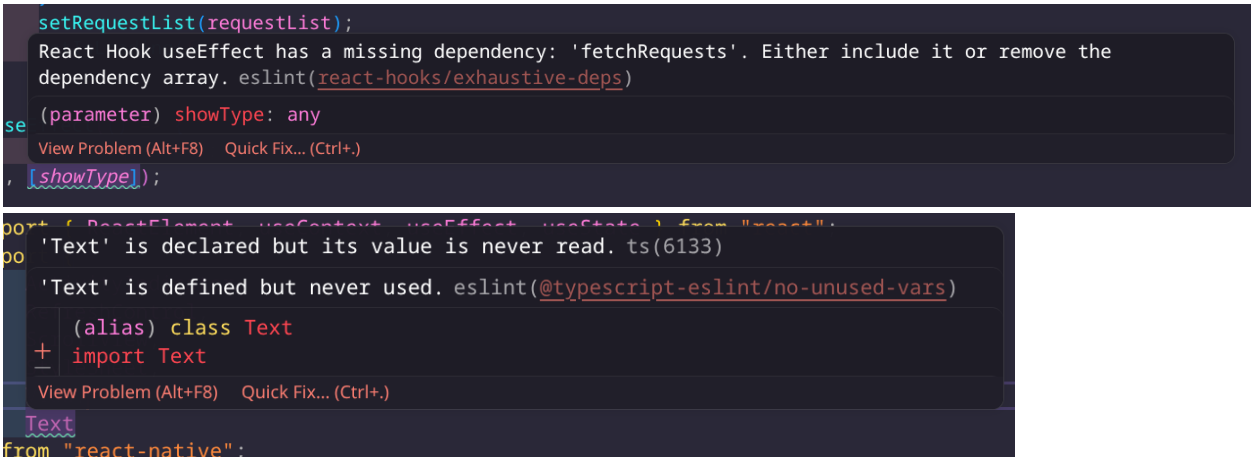
Visual Studio Code / VSCodium

Visual Studio Code (more specifically an open-source distribution named VSCodium) was the code editor of choice for this project. It comes with a myriad of features for speeding up development, including an interactive file tree display, integration with GitHub, switching lines using Alt+Up/Down, highlighting multiple instances of the same text with Alt+D, creating multiple cursors that allow you to make the same changes in multiple places at once, and more. It also has a large variety of user-created extensions, some of which add debugging and IntelliSense for different types of files, and some of which just make development easier in various ways using visual indicators. Some examples of the ones used in the development of this project are:

- Expo Tools: An extension that adds IntelliSense for the Expo configuration files
- ESLint: Enables the usage of ESLint with VSCode, which will be explained below
- Highlight matching tag: When clicking onto a JSX component, it will highlight the corresponding opening or closing tag, making it easy to see which components are enclosed within those tags
- indent-rainbow: An extension that provides a colour to each level of indentation in your code, making indentation much more readable at a glance

ESLint

ESLint is a linter, meaning that it searches developers' code for problems and helps to provide solutions to them. For example, it can find variables that are being read despite being undeclared, parts of functions that will never run due to necessarily having to return before getting to them, keywords in unexpected places, and more. When the linter finds an error like this, it will display a line under it, and hovering over it gives a description of the problem and may also provide a way to quickly fix it. This is another tool that speeds up debugging significantly by diagnosing issues before they would become a real problem.



Prettier

Prettier is a code formatter that allows developers to save time by not having to worry about getting every bit of indentation and formatting right, handling the majority of formatting itself. Prettier formats code automatically based on a set of rules while always preserving the functionality of the code itself. Developers can choose to use it to format a file all at once, or set it up to format the code every time the file is saved.

Git / GitHub

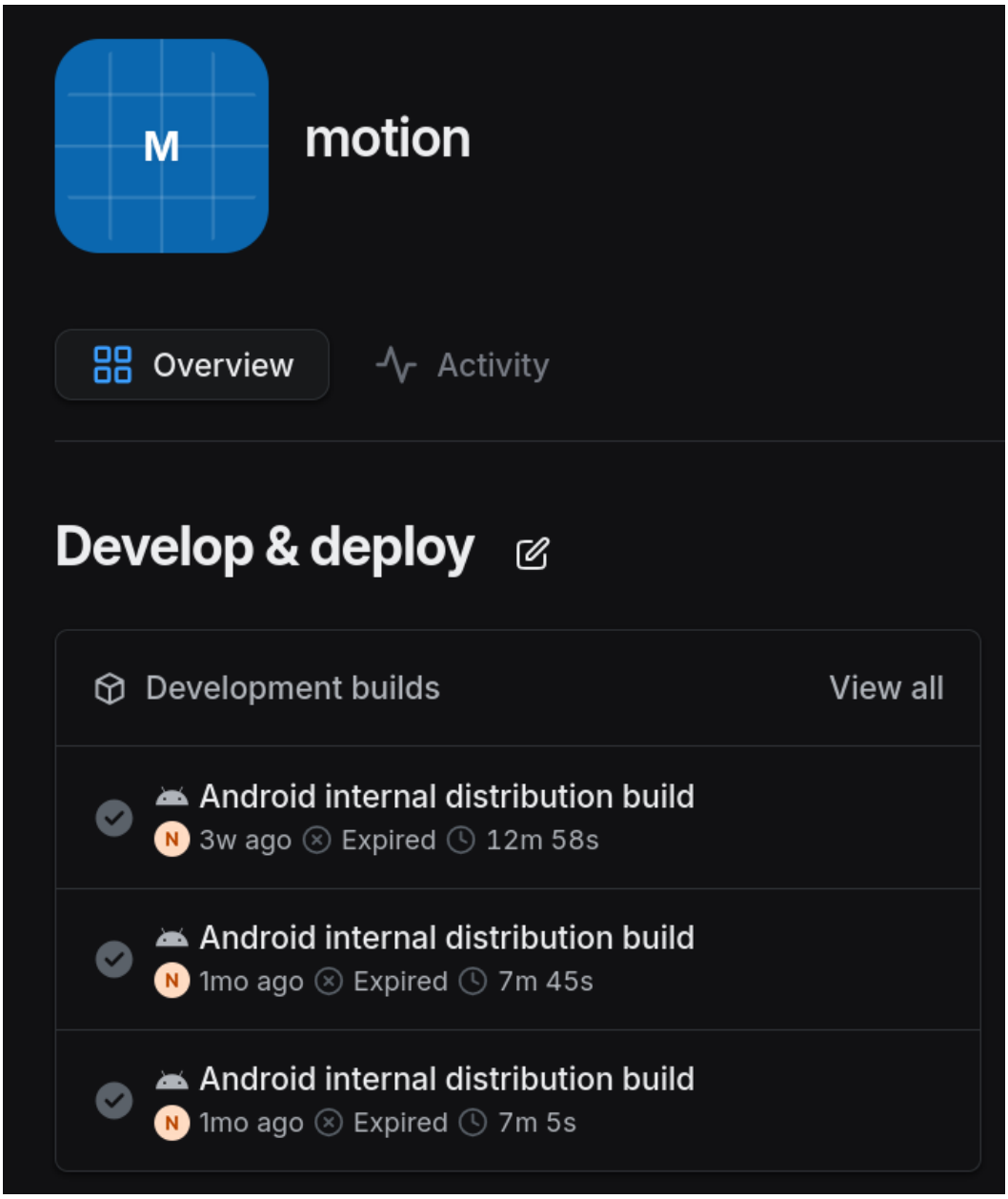
Git is version control software that manages different versions of a codebase. Developers add their code to a "repository", which allows Git to track the changes in it. Once the codebase is stored as a repository, the developer can stage commits using the "git add" command, then commit them by using the "git commit" command. Once the changes are committed, they are stored as a commit that can be accessed at any time using the "git checkout" command. Developers can also make branches, which as the name implies, creates a code "branch" that can be changed and committed to independently of the main

branch. Examples of how this can be used are by having a “dev” branch that you apply consistent updates to and then merge the the main branch, or having each collaborator working on a different branch and then merging their changes together later.

GitHub is a website that developers can use to host their Git repositories online, making collaboration much easier. Git has a “git push” command that pushes all of the commits to the upstream URL, which you can specify as a GitHub repository, allowing for the repo to be accessed online. GitHub repositories can be public or private. Developers who want to share their code can leave their repo private, and other developers can make pull requests to change parts of the code, or create variants of the program starting from the same codebase as the original, called “forks”. Pushing to GitHub also provides a much easier way to work on the same repository on multiple computers, as you can pull the changes from the GitHub repo and start working on it again from a different location.

Expo Application Services

There are a few different types of builds for Expo apps, but the main type used during development is the development build. Expo provides a cloud service called Expo Application Services that allows for these builds to be created on Expo servers, although if you're using a free account, it can take up to a few hours to create a development build at peak times. The configuration for the development build is specified in the app.json file. After using the EAS CLI tool to create a development build, a QR code is displayed. Scanning it allows you to install the application package for Android. Then, running "npx expo start" starts the development server and displays another QR code. Scanning this code inside of the development build allows you to run the application from the development server. This development build comes with a few extra features, the most notable being Fast Refresh, which makes the application refresh each time a change is made to the source code, so changes are reflected immediately.


M
motion
Overview
Activity
Develop & deploy
Development builds
View all
Android internal distribution build
N 3w ago Expired 12m 58s
Android internal distribution build
N 1mo ago Expired 7m 45s
Android internal distribution build
N 1mo ago Expired 7m 5s

Testing and Evaluation

Unit testing

Unit testing refers to a type of testing in which individual components, or “units”, of code are tested individually to ensure that they run correctly in isolation. For example, a test could run a function given specific inputs, and expect a given output that should be produced from those inputs. If the test fails, then there is likely some kind of issue with the code. This is useful because as the codebase expands, changes may be made to those functions, which may break the expected behaviour. Running these tests regularly ensures these changes haven’t unintentionally broken anything.

Testing for the frontend of this application is done using Jest, a library which allows developers to define test suites in “.test.ts(x)” files, which will be run when the testing command is run; in this instance, “npm run test” is set to run “jest --watchAll”, which watches files for changes and reruns the tests whenever it sees one. The unit tests are defined in the “__tests__” folder, and mostly focus on the database service files. “utils.test.tsx” tests the various utility functions used in the application, such as the type converters and the functions to convert times to string representations.

```
it("converts undefined coordinates to 0 by default", () => {
    const coordinatesList: Coordinates[] = [
        { latitude: undefined, longitude: undefined },
        { latitude: undefined, longitude: undefined },
        { latitude: undefined, longitude: undefined },
    ];
    const expected: Coords[] = [
        { lat: 0, lng: 0 },
        { lat: 0, lng: 0 },
        { lat: 0, lng: 0 },
    ];

    const coordsList: Coords[] = coordinatesList.map((c) =>
        coordinatesToCoords(c),
    );
    expect(coordsList).toEqual(expected);
});

it("converts an array of coord strings into a route array", () => {
    const routeStrings: string[] = ["1,2", "3,4", "5,6"];
    const expected: Coords[] = [
        { lat: 1, lng: 2 },
        { lat: 3, lng: 4 },
        { lat: 5, lng: 6 },
    ];

    const coordsList: Coords[] = routeStringsToCoords(routeStrings);
    expect(coordsList).toEqual(expected);
});
```

There is a problem when it comes to testing the Axios requests, which is that the tests can’t wait for an Axios promise to be returned. Instead, it's necessary to use a library that

mocks the response from Axios, called “jest-mock-axios". This library allows us to define a response to pretend that Axios returned, and use that response to test against.

```
describe("get activity by id", () => {
    it("should return activity details when api call succeeds", async () => {
        const activityId = "123";

        const mockResponse = {
            route: [
                { lat: 1, lng: 1 },
                { lat: 2, lng: 2 },
                { lat: 3, lng: 3 },
            ],
            startTime: new Date("2026-04-21T13:28:00"),
            finishTime: new Date("2026-04-21T13:58:00"),
            distance: 200,
            speed: 100,
        };

        const promise = ActivityService.getActivityById(activityId);

        expect(mockAxios.get).toHaveBeenCalledWith(
            `https://motion-backend-6m2q.onrender.com/api/activities/${activityId}`,
        );

        const responseObj = { data: mockResponse };
        mockAxios.mockResponse(responseObj);

        const result = await promise;

        expect(result).toEqual({
            route: [
                { lat: 1, lng: 1 },
                { lat: 2, lng: 2 },
                { lat: 3, lng: 3 },
            ],
            startTime: new Date("2026-04-21T13:28:00"),
            finishTime: new Date("2026-04-21T13:58:00"),
            distance: 200,
            speed: 100,
        });
    });
});
```

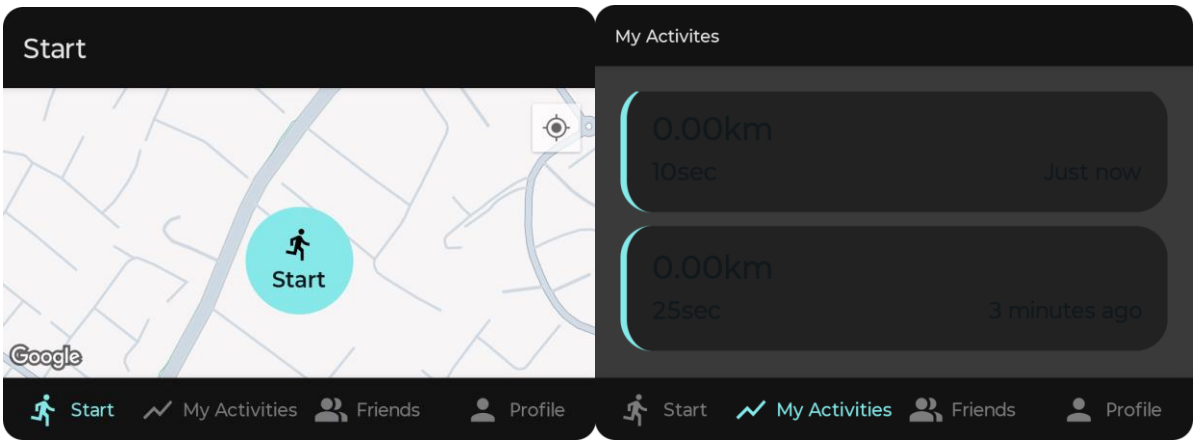
User testing

Near the end of development, a preview build of the application was sent out to a few testers to acquire feedback; This testing was done somewhat informally and didn’t have a set plan behind it, other than getting feedback on the application and requesting that

testers report bugs that they found. Ideally, this would be done with a proper testing plan and consent forms, but time constraints put a limit on what could be done.

Through this testing, there were a small number of bugs found that hadn't been found before. The main one was that the application displayed incorrectly when the user's phone was set to light mode, with the text being dark on top of a dark background. This could be fixed by creating a new build of the app with the "userInterfaceStyle" in the app.json file set to "dark", rather than "automatic".

One tester also had a phone on which the application could be displayed on a smaller screen; the application happened to look fine on that screen for the most part, and would require only a minimal redesign to accommodate for these types of screens.



Unresolved bugs

There is a major unresolved issue in the final build of the app which can cause it to become unopenable. From debugging, it appears to be related to closing the app while the location tracking task is currently active. In the development build, when closing the app during tracking and reopening it, it displays an error relating to a call to "ExpoTaskManager.unregisterTaskAsync" being rejected. The error provides little information to work with, and searching for it online returns no results with that specific error. The best guess that can be made as to the reason for the error is the task creation function in the map view component causing an error if the task is already running, but a solution could not be found for this.

Another bug that went unresolved relates to viewing a friend's activity. Tapping on the activity card for a friend's activity sends the user to the activity view screen for that activity. The issue is that the activity view shown this way does not include a back button to return to the previous screen, and does not provide any way to return to the base activities list. I

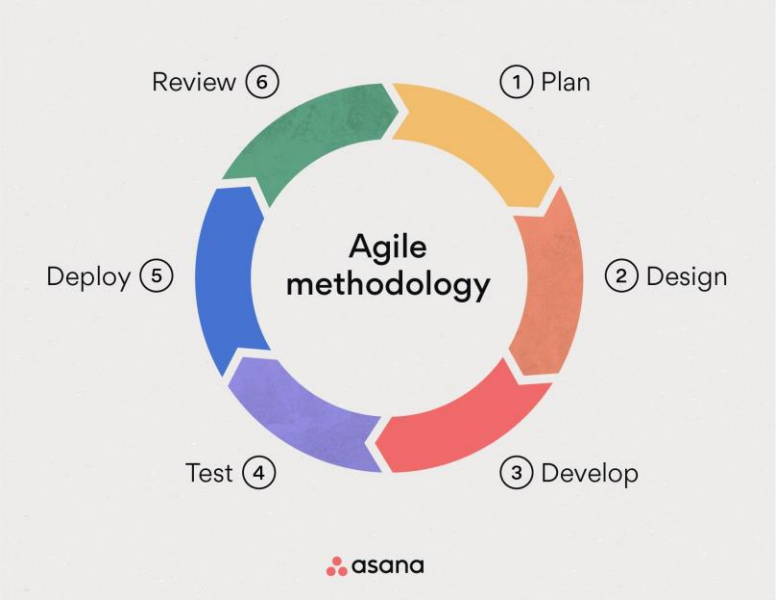
believe that the solution is likely related to Expo Router's "shared route" feature, but there wasn't enough time to find a working solution to this.

Project Management

Methodology

The methodology that was planned to be used for this project is agile methodology. Agile methodology is characterised by flexibility, consistent pace of work and testing throughout the development process. This stands in contrast to waterfall methodology, which is a strategy in which the project is broken down into large, fixed phases, and testing is typically only done after the development period is complete.

Agile methodology is often used alongside the idea of the sprint cycle. Sprints refer to a plan in which a set amount of work or a specific goal is planned to be reached within a short period of time. After the sprint is complete, the work is reviewed and tested, and the next sprint begins, potentially with adjustments if more or less work than expected was completed.



While a sprint plan for this project was written, it wasn't adhered to as closely as would have been ideal. This was largely because of technical problems with the mapping API near the start of the project that took a lot of time to resolve, and also caused a hit to motivation. In a future project, it would be an improvement if the sprint plan underwent a revision to account for this, so that work could resume more quickly at a normal pace.

Conclusion and Future Work

Summary of Findings & Project Aim

Overall, I would consider the project to generally be a success in terms of meeting the aims set out at the beginning. While the app is not fully stable, at outlined in the Unresolved Bugs section, it is a functional mobile application with effective use of mapping tools and mobile location services. It accomplishes the main goals of being an application developed using a cross-platform development framework, making use of elements only available to mobile devices. It is unfortunate that a build could not be made for Apple devices, but as I have no access to a mobile device running iOS or a computer running MacOS, making a build for iOS was infeasible. Still, the Expo documentation provides plenty of information that would help with the development for an Apple device should it become a possibility in the future.

Critical Reflection on Methodology

From a methodology point of view, the project could have been handled better. The plan for the project was to use agile methodology and use a sprint cycle, but as outlined in the Project Management section, this did not work out ideally due to technical difficulties near the beginning of the project. In a future project, this would be handled by discussing the issues being had in a sprint review, and adjusting the goals for future sprint cycles based off that.

Limitations of current prototype and suggestions for future

The application is functional in terms of what it is built to do, but in its current form it is still quite limited and there is significant room for modifications and new features. For example, the app has a friend requests feature, but it doesn't provide much to actually allow the user to engage with the friends you've added, such as allowing for users to comment on their friends' activities. There also isn't much in the way of allowing users to personalise their app experience. These are some of the features that would ideally be implemented or expanded on if development of the app were to continue:

- An "index" page that would allow the user to see recent activity from all of their friends in chronological order, similar to a social media timeline

- More ways to interact with friends, such as commenting on their activities or following a route that they did originally
- Achievements that users could earn for doing a certain number of activities or reaching milestones, encouraging users to keep tracking their activities
- A more robust system for tracking while the app is closed (ExpoTaskManager is supposed to do this, but it doesn't work in the current implementation)
- Ways to track progress over time, such as a screen that shows the distance travelled on activities on a week-to-week basis
- More personalization features, such as privacy settings (allowing users to hide activities from friends) and theme settings

Final conclusions

To conclude, I believe the project was a success overall. Most of the features that were planned to be implemented were implemented by the end, and the app is functional with a minimal number of large bugs. The project management was not optimal and could have been handled better, ultimately, the goals set out at the start were achieved regardless. The development of this application granted me a lot of insight into how mobile apps are developed, which is something I had next to no experience with at the beginning of the project. It was also a good experience in self-taught learning; other modules in the course to this point all gave us a basis in the framework to build off, but this was an instance where I had to put the effort in to learn about the framework and build the project from the ground up. For these reasons, I think this project was a very positive personal experience.