CCT College Dublin

Assessment Cover Page

|  |  |
| --- | --- |
| Module Title: | Cross Platform Development |
| Assessment Title: | group |
| Lecturer Name: | David González |
| Student Full Name: | Youssef sabry, Daniel Bobuh, Louis Esvenn Canilang, Terence Teche |
| Student Number: | 2023318, 2023468, 2023429, 2023303 |
| Assessment Due Date: | 28/05/2025 |
| Date of Submission: | 28/05/2025 |

Declaration:

|  |
| --- |
| By submitting this assessment, I confirm that I have read the CCT policy on Academic Misconduct and understand the implications of submitting work that is not my own or does not appropriately reference material taken from a third party or other source. I declare it to be my own work and that all material from third parties has been appropriately referenced. I further confirm that this work has not previously been submitted for assessment by myself or someone else in CCT College Dublin or any other higher education institution. |

Technical report

[Technical Report 2](#_Toc199275474)

[Design Decisions 2](#_Toc199275475)

[API Integration Challenges and Solutions 3](#_Toc199275476)

[OpenStreetMap API – Geocoding 3](#_Toc199275477)

[Open-Meteo API – Weather Data 4](#_Toc199275478)

[Internet and Location Permissions 4](#_Toc199275479)

[State Management Approach 5](#_Toc199275480)

[Main State Variables 5](#_Toc199275481)

[Use of useEffect 5](#_Toc199275482)

[Local Storage Integration 6](#_Toc199275483)

[Justification for Simplicity 6](#_Toc199275484)

[Reflections on the Development Process 6](#_Toc199275485)

[Potential Future Improvements 7](#_Toc199275486)

[1. Modular Code Structure 7](#_Toc199275487)

[2. User Preference Persistence 7](#_Toc199275488)

[3. Accessibility Enhancements 7](#_Toc199275489)

[4. Visual Data Presentation 8](#_Toc199275490)

[5. Error Handling for Location Services 8](#_Toc199275491)

[6. Splash Screen or Loading Animation 8](#_Toc199275492)

[7. Offline Support 8](#_Toc199275493)

[8. Multi-language Support 8](#_Toc199275494)

# Technical Report

## Design Decisions

Daniel developed the entire application inside the main file named index.tsx. All of the logic and user interface are contained in this single file. This includes fetching weather data, managing user input, handling GPS permissions, displaying current and forecasted weather, and supporting user preferences such as theme and temperature units.

The decision to keep everything in index.tsx was intentional. Daniel wanted to maintain a clear and centralized workflow during development. By having all features in one place, it became easier to control the app’s behavior, debug issues, and manage state without switching between multiple files or components.

For weather data, Daniel used two APIs:

* OpenStreetMap (Nominatim) to convert user-entered city names into geographic coordinates (latitude and longitude).
* Open-Meteo to fetch weather data including the current temperature, wind speed, humidity, hourly forecast, and a 7-day daily forecast.

These APIs were chosen because they are free, publicly accessible, and do not require an API key. Integration was done using basic fetch requests directly within the index.tsx file.

Daniel included a text input field for searching cities and added support for displaying past searches using local storage (AsyncStorage). The app loads previous searches when it starts and allows the user to clear them at any time.

The app also features a button to use the device's current location, which is handled using the expo-location package. If permission is denied or the location cannot be accessed, the app displays an error using the react-native-toast-message package.

To improve user experience and accessibility, Daniel implemented:

* A dark/light mode toggle, which changes the app’s styling using dynamic state
* A Celsius/Fahrenheit toggle that updates the temperature units across the app
* Animations for the weather display section to provide smooth visual feedback
* Responsive design, using useWindowDimensions() and Platform to adapt layout and font sizes based on the device type and screen orientation

All state management in the app is handled using useState and useEffect. The code also includes clear inline comments to explain the purpose of each section and function, which helps others (including non-coders) understand how the app works.

By keeping the structure compact and readable, Daniel was able to focus on implementing all required features without overcomplicating the architecture. This approach made it easier to test functionality and ensure that all core requirements were met in a single, fully functional component.

## API Integration Challenges and Solutions

Youssef used two separate public APIs in the project: OpenStreetMap (Nominatim) for geocoding and Open-Meteo for weather data. Both were integrated using direct fetch requests within the index.tsx file, supported by utility functions from geocode.js and weather.js.

### OpenStreetMap API – Geocoding

To allow users to search weather by city name, Youssef implemented a geocoding function in the geocode.js file. This function takes a city name and sends a request to the OpenStreetMap (Nominatim) API. The goal is to receive valid latitude and longitude coordinates for use in weather data requests.

A challenge Youssef faced here was that the API could sometimes return empty arrays or multiple results, especially for cities with common names. To solve this, Youssef filtered the result to pick only the first valid match and extracted its latitude, longitude, and country fields.

Youssef also added error handling so that if no location is found or the user enters an invalid city name, the app displays a clear message using react-native-toast-message. This prevents the app from trying to fetch weather data without valid coordinates.

### Open-Meteo API – Weather Data

Youssef used the Open-Meteo API to fetch current and forecasted weather data. This was handled in a dedicated file called weather.js, which contains a function that constructs the API URL based on the latitude and longitude provided by the geocode function or the device’s GPS location.

One of the main challenges was working with Open-Meteo’s structured response. The data comes in arrays, and Youssef had to correctly map daily and hourly forecast values to specific UI elements in the app. For example, to show a 7-day forecast, Youssef extracted the appropriate entries from the daily section of the API response and matched weather codes to condition icons using a helper function.

To prevent repeated or unnecessary requests, the app checks whether data already exists in the state before re-fetching. Additionally, the API requests are wrapped in try-catch blocks to handle network failures or bad responses gracefully.

### Internet and Location Permissions

Youssef added another layer of control by checking the device’s internet connection using the @react-native-community/netinfo package. Before making any API call, the app checks if the device is online. If it is not, the user is informed immediately with a toast, and no request is sent.

When requesting the device's current location, Youssef used the expo-location package. A common issue was when users denied location access, which caused failures in getting coordinates. To address this, Youssef implemented a check for permission status and handled both approval and denial with proper messages and fallbacks.

Overall Integration Strategy

* API calls were separated into helper files (geocode.js and weather.js) for better modularity and reuse.
* All asynchronous logic was handled with async/await and wrapped in error handling blocks.
* Toast notifications were used to provide clear and immediate feedback to users if something went wrong.
* Internet connectivity was checked before every request to avoid silent failures.
* API data was normalized and transformed before use, such as matching weather codes to icons, extracting only necessary fields, and formatting date/time values using day.js.

By addressing each of these challenges, Youssef was able to create a stable and responsive app that gives users accurate and real-time weather information based on both manual searches and GPS location.

## State Management Approach

Terence managed all of the application’s state using React's built-in useState and useEffect hooks inside the index.tsx file. Since the entire app is built within this single component, it was important to keep the state logic organized and minimal, while still covering all features.

### Main State Variables

The following are the main state variables Louis used in the application:

* city: Stores the user’s current input for the city name in the search field.
* searchHistory: An array of the user’s most recently searched cities, retrieved from and saved to local storage using AsyncStorage.
* weatherData: The core object containing all fetched weather data from Open-Meteo. This includes current temperature, wind speed, humidity, weather codes, daily high and low temperatures, and hourly forecasts.
* isCelsius: A boolean value that determines whether temperatures are displayed in Celsius or Fahrenheit. This state is updated when the user toggles the unit switch.
* darkMode: A boolean that stores whether the app is currently in dark mode or light mode. Toggling this state changes the entire UI styling accordingly.
* loading: A boolean used to show a loading indicator while weather data is being fetched.
* currentDateTime: Stores the current date and time, formatted using dayjs, and updated when new weather data is loaded.

These state values are updated based on user actions such as typing a city, submitting a search, or pressing the location button.

### Use of useEffect

Terence used multiple useEffect hooks to handle side effects:

* One useEffect loads the search history from AsyncStorage when the app first launches.
* Another useEffect applies the fade-in animation for the weather display every time weatherData is updated.
* Additional useEffect hooks monitor screen size changes and trigger visual updates accordingly.

### Local Storage Integration

To store and retrieve data between sessions, Louis used AsyncStorage. This was used specifically for saving and loading the user’s recent search history. When the app launches, loadHistory() is called to load any previously saved cities. When the user performs a new search, saveToHistory() adds it to the array and updates the local storage.

This approach avoids the need for more complex state management libraries like Redux, since the scale of the application is small and the state is limited to one screen.

### Justification for Simplicity

Terence decided to use only useState and useEffect because they are simple to implement and sufficient for the project’s scope. All interactions happen in a single component, so a centralized store was not needed. This approach also made the code easier to read, test, and debug during development.

## Reflections on the Development Process

During the development of this weather application, we gained practical experience in React Native, especially in managing asynchronous operations, working with APIs, and designing user-friendly mobile interfaces. By building the entire application logic inside a single component (index.tsx), we were able to focus on functionality, maintain control over the app's flow, and reduce unnecessary complexity.

One of the most important lessons we learned was how to handle external APIs effectively. Working with both the OpenStreetMap and Open-Meteo APIs required us to process responses, extract the relevant data, and handle edge cases, such as when a city was not found or when the internet connection was unavailable. Implementing error handling using try/catch and providing real-time feedback to the user using toast notifications improved the app’s reliability and usability.

We also learned how to improve the user experience through thoughtful features such as dark and light themes, animated weather cards, responsive layouts, and persistent storage using AsyncStorage. Saving and reloading search history helped us understand how local data can enhance convenience without needing a back-end database.

Designing for different screen sizes using useWindowDimensions() and Platform allowed us to better understand responsive design in mobile development. This made the app more adaptable for both phones and tablets, and laid the groundwork for potential improvements in UI scaling and layout optimization.

Although we implemented everything in one file, we prioritized clean structure and detailed code comments. Functions were clearly separated by their purpose including geolocation, weather data retrieval, unit toggles, theme handling, and history management which made the project more organized and easier to debug or update.

We also came to appreciate the importance of anticipating failure states. Whether it was a denied location permission, no internet access, or incorrect user input, we aimed to give the user clear, helpful messages instead of allowing the app to fail silently.

Overall, the project gave us a strong foundation in practical React Native development. It demonstrated how to connect multiple technologies like APIs, device features, and local storage into a cohesive and functional mobile application. It also encouraged us to think critically about performance, user interaction, and design consistency.

## Potential Future Improvements

While the application meets its functional requirements and delivers a complete weather experience, there are several areas where we identified room for future enhancement, both technically and from a user experience perspective.

1. Modular Code Structure  
Currently, all logic and UI components are implemented inside the index.tsx file. Although this kept the code centralized and easy to manage during early development, breaking the code into smaller reusable components and utility modules would improve readability, scalability, and maintainability. For example, the weather display, forecast cards, and search bar could be placed into separate component files.

2. User Preference Persistence  
At the moment, temperature unit (Celsius/Fahrenheit) and theme mode (dark/light) settings reset each time the app is restarted. In future updates, we could store these preferences in AsyncStorage, so the app remembers user selections between sessions.

3. Accessibility Enhancements  
The app could be improved by adding accessibility features such as screen reader support, text scaling based on system settings, and better contrast handling in both dark and light modes. These changes would make the app more inclusive for users with different visual or physical needs.

4. Visual Data Presentation  
Currently, the weather information is presented as plain text and emojis. Replacing or supplementing these with charts or weather icons could enhance visual clarity. For example, line graphs for hourly temperature or icons representing rain, clouds, and sun could make the forecast easier to understand at a glance.

5. Error Handling for Location Services  
When GPS access is denied or unavailable, we currently display a toast message. A more user-friendly approach would be to guide the user with a prompt to open device settings and enable location services. This would improve usability, especially for users unfamiliar with system permissions.

6. Splash Screen or Loading Animation  
Adding a custom splash screen or an animated loading state (such as moving clouds or a fading weather icon) would improve the app’s polish and perceived performance. It would also help create a more engaging first impression when launching the app.

7. Offline Support  
The app currently notifies users when the device is offline but does not offer fallback data. A future enhancement could include caching the most recently fetched weather data, allowing it to be displayed even when the user is offline.

8. Multi-language Support  
Expanding the app to support multiple languages would make it more accessible to a broader user base. This could be achieved using internationalization libraries like i18n-js or react-native-localize.

These improvements are realistic and achievable with the current codebase and would significantly enhance the app’s functionality, usability, and professionalism.