# REACT NATIVE WEATHER APP

# APP DEVELOPMENT CLUB

Dean Kajosevic

Welcome to the React Native Weather App Lab. Today, you will be learning the key concepts of the React Native framework, CSS, TypeScript, and some UI/UX design.

React Native is a framework that allows for IOS and Android mobile development using JavaScript (or a superset of JavaScript, such as TypeScript), rather than having to code separate programs in Swift and Kotlin (the respective coding languages of IOS and Android). Furthermore, React Native facilitates the creation of components, which are snippets of code that can be reused across the program.

1. **Installing Dependencies**
   To start off, we will be using Expo as a framework to help deploy our React Native project. I will be doing my coding in Visual Studio Code, but you can use any IDE that you prefer.

   However, to use Expo, you must have Node.js installed on your computer.
   If Node.js is already installed, skip to step 2.

   Homebrew (Mac only): the unofficial MacOS package manager. Tis the simplest way to download software from across the internet with the magic words of "brew install (*whatever package you want here*)"
   Node.js: JavaScript runtime environment. Runs JavaScript programs.
   a. On Mac, install it with `brew install node`

2. **Making the Expo project**
   First, access a directory that you would like to store your project in (you can make a directory in the terminal command line by typing:

   mkdir (directory name)

   To access this directory, you can enter:

cd (directory name)

Once you are in the directory that you would like to hold your project in, you will use a Node.js command to create your project directory. Type:

Npx create-expo-app --template // Yes, the two consecutive dashes are correct
→ Follow the prompt to enter the project name: reactnative-weather
→ Choose Blank (TypeScript)

Choosing a blank TypeScript project will configure the React Native project to use TypeScript, which is the coding language that we will be using for this project.

TypeScript adds type declaration to JavaScript

cd into your project directory, then type:

npm start

Select ios or android (depends on what you want to test on). This will allow you to see the progress that you are making on the application.

If you do not have Expo Go installed, then press Y to install when prompted.

If you are on Mac, I suggest downloading Xcode from the app store and downloading the iPhone simulator within it. It should be prompted when you install Xcode, if not you can find it in settings. ChatGPT and the internet will help if necessary.

If you do not want to opt into Xcode to run the simulator, you can install the ExpoGo app on the phone and scan the given QR code when you execute "npm start" to simulate your React Native project on your mobile phone.

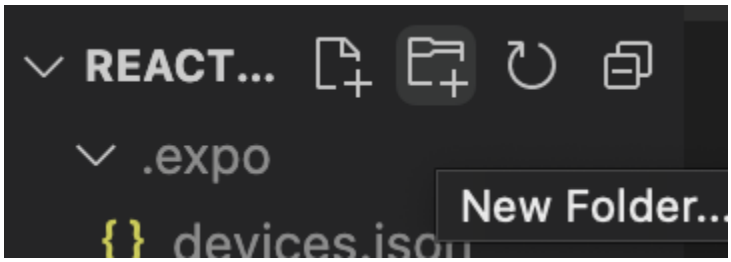
If you are using the ExpoGo application on your phone, you need to be on the same internet as the server running your project.

3. **Getting familiar with React Native**

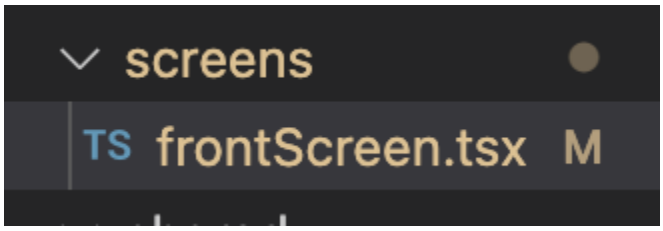
   Once you open your project in an IDE like Visual Studio Code, you will be prompted with multiple already configured files. Most notably, the App.tsx file.

   If you open your App.tsx file, you will see code that displays text on your screen. We are going to leave this for now and come back to this later, as this file will be in charge of displaying what is on the mobile device.

   Next, let's create a "screens" directory in our project using the new folder button. This directory will hold all of our files that we will use to display a different screen on our application.




   The first file that we will add to this folder will be named "frontScreen.tsx". Notice the .tsx extension on the file name? This extension indicates that the file is a TypeScriptx file, which means that TypeScript and JavaScript code may be used.




   In the frontScreen.tsx file, we want to import some dependencies that will allow us to utilize elements of React Native.

```
import React from "react";
import { useState, useEffect } from "react";
import { View, Text, StyleSheet } from "react-native";
```

We will get into useState and useEffect later, but View, Text, and StyleSheet are classified as some of React Native's core components. These components are essential to build the User Interface (UI) of our mobile application.

View is used for styling, grouping components, and laying out a mobile application.

Text is used to display text on the screen. All text that you want to display must be wrapped by <Text> </Text> in React Native, which is not the case in React (the web development framework).

If you have any prior experience with web development, <View> has similar functionality to .

Now that you have been introduced to View and Text, let's put these components into action.

4. **Setting up frontScreen.tsx**
   After you import the dependencies in step 3, we need to create a function named FrontScreen() in our frontScreen.tsx file.

```
export default function FrontScreen() {
return();
}
```

A `function` is a block of code that is meant to execute a certain function and be reusable within other parts of a program.

The keyword `export` is used to export the function to the rest of the project directory, so the function can be used in other files. This will be crucial when we want to display our screen in App.tsx.

The keyword `default` is used to export a single function from a file, which will remove the need for curly braces {} when importing the function in other files.

The `return();` line is what is spit out from the function when it is called. This is where we will be inserting our UI components so it can be displayed on our emulator.

Finally, we are going to throw some UI components into our return statement in the `FrontScreen()` function.

Add the following code inside the return statement:

```
<View style={styles.container}>
    <Text style={styles.title}>Weather</Text>
</View>
```

Simply, we added code that creates a group (View) for our Text component to lie within. The text "Weather" that lies between the `<Text></Text>` brackets is the text that will be displayed.

Take note that there must always be an opening bracket `<Text>` and a closing bracket `</Text>` for each type of component. You can shortcut this if you have nothing to put between the brackets by doing `<View />` or `<(component name)/>`.

5. **Modifying App.tsx to show FrontScreen**
   Even though we made changes to FrontScreen, they are still not visible on the emulator. This is the case because we have not modified App.tsx, which holds the code that the emulator displays.

```
import FrontScreen from './screens/frontScreen';

export default function App() {
 return (
   <FrontScreen />
 );
}
```

All you have to do is import FrontScreen and add the component to the return statement. Now you can see your changes!

6. **Getting familiar with CSS**

# APP DEVELOPMENT CLUB - REACT NATIVE WEATHER APP

Cascading Style Sheets (known as CSS) is a styling language that is used to alter the appearance of HTML, XML, or in this case React Natives UI components.

If you look at the code that we inserted into `FrontScreen()`, there is the phrase `style={}` after the `View` and `Text` words in their opening brackets. This sets the styling to `styles.container` and `styles.title`, which are both defined in a style sheet.

To create a Style Sheet, we use the imported library from step 3. This can be done by copying the following:

```
const styles = StyleSheet.create({
 container: {
   paddingTop: 70,
   justifyContent: "center",
   alignItems: "center",
   backgroundColor: "#f0f0f0",
 },
 title: {
   fontSize: 24,
   fontWeight: "bold",
   justifyContent: "center",
 },
});
```

This code snippet should be placed at the very bottom of the frontScreen.tsx file.

Here, you are now able to see where we refer to `styles.container` and `styles.title`. The `const` keyword declares that the given variable (`styles`) can not be reused in the code. We set `styles = StyleSheet.create` in order to create a new style sheet with the given styles that we create within the parentheses.

`container` and `title` are two different styles that we created in the StyleSheet to add some flavor to our UI components. There are different styling options for each UI component (i.e. there are some styling variables that could be for `Text` components, but not for `View` components ****font).

Here is the link to see CSS documentation for React Native and learn what these different styling variables do to the components.

Correction: StyleSheet is technically React Native's own version of styling, but it is an abstraction similar to CSS styling. So, I will use these terms interchangeably for my sanity.

Now that we understand what a StyleSheet is, let's review the code from earlier:

```
<View style={styles.container}>
    <Text style={styles.title}>Weather</Text>
</View>
```

In this code, we want to assign the container style that we created to the View component. We are able to reference the container style in our StyleSheet by referring to the `styles.container` (remember, we named our StyleSheet styles).

7. **Signing up for weatherAPI**

   Now that we covered some of the basics, it is time to throw you all in the deep end. A large portion of making a weather app is fetching weather data that can be used to find updated, and accurate, information.

   An API is an Application Programming Interface, which allows for different software applications to share data/communicate with each other. In our case, we will be using weatherAPI's weather api (funny).

   Follow this link to sign up for weatherAPI.

   After you log in, you will be taken to a page for your account where your API key will be blatantly in front of your eyes. Make sure that you do not share your API key, so nobody can mess with your API key and potentially 1. lose the money

you spend on an api key, 2. get banned from using the API key for malicious API calls.

An API key essentially gives you access to the API service that is in question. In our case, weatherAPI has a free version that we will be using.

On the topic of not getting our API keys stolen, people will often make GitHub repositories to flex their projects, but forget to add files that use their API keys to the .gitignore file. This means that there are plenty of keys out there (that were paid for) ready to be stolen online.

In the case that you plan on adding this project to your GitHub, I am going to teach you some good practice in hiding your API Keys.

8. **Hiding API keys**
   This section will be extremely easy, so just follow along.

   In your main project directory, create a new directory named `constants`. This directory will be used to hold files that have constant variables that we will reference in other pieces of code. Inside the constants directory, make a file named index.ts.

   Index.ts should consist of one line of code that looks exactly like this, but input your own API Key where it says to:

```
export const apiKey = "insert API key in these quotes";
```

   The `export` and `const` keywords should be familiar to you by now, because I explained it in the previous steps.

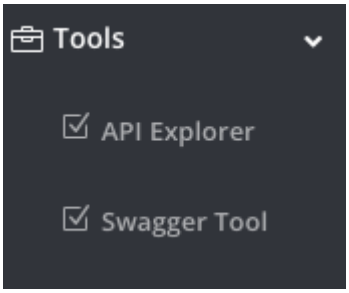
   Essentially, we are exporting our API key as a variable name so our future code can reference our API key through the apiKey variable rather than the actual key itself. We do this so you only have to add this file to .gitignore and not hide your all of your code that involves API calls (so you can show off to employers).

   Whenever you have a .gitignore file (if you make the project a repository), then add this code to your .gitignore file:

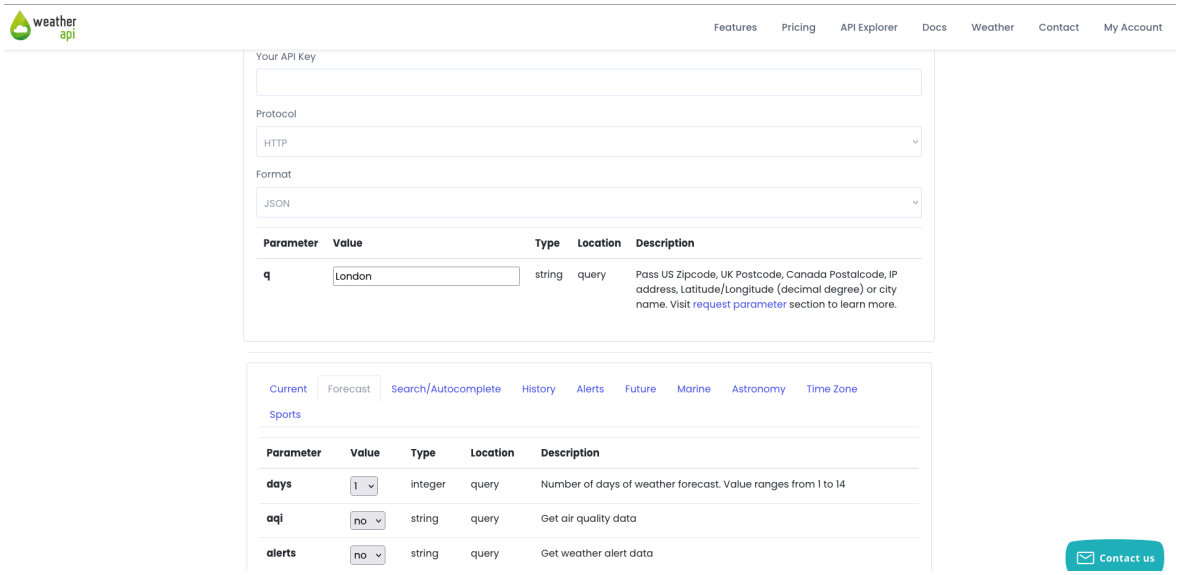
```
# Ignore weatherAPI key
/constants/index.ts
```

9. **Fetching API data**

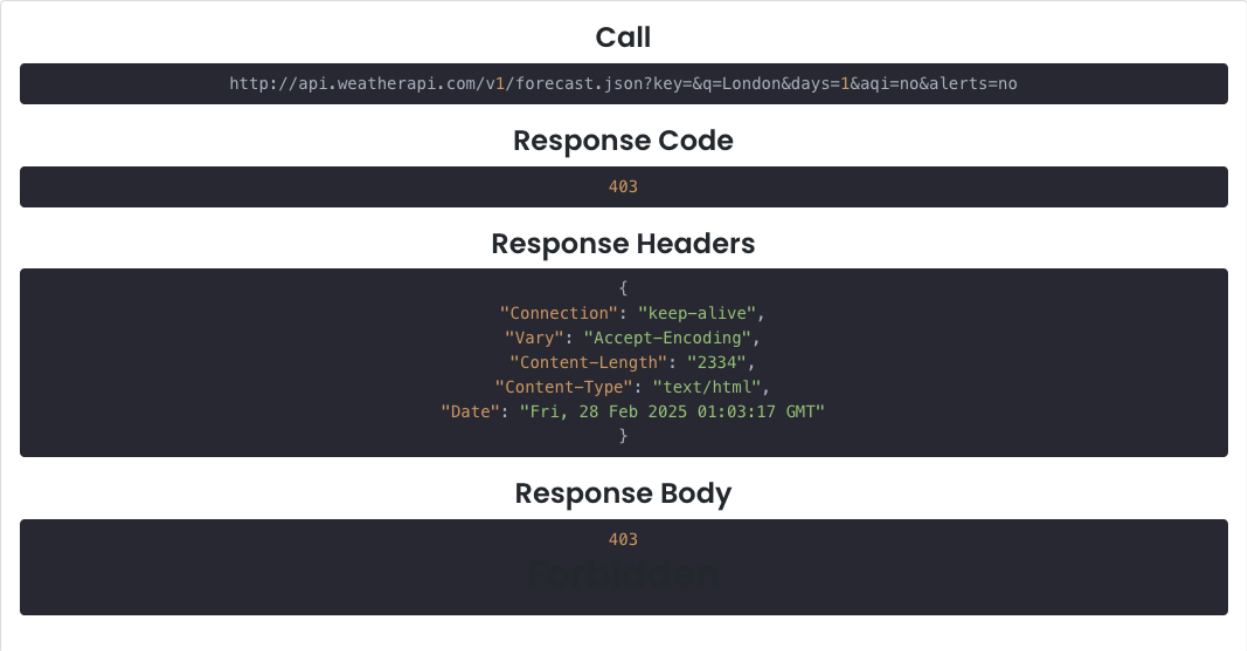
When you are at the main page after logging in, click the API explorer tab on the lefthand side.




Once you click on API Explorer, you will be redirected to this page:




Here, you will want to insert your API key, leave http and JSON as the selected boxes below, click the Forecast box, set days=1, aqi = no, and alerts=no. Then click the Show Response box.




You will get a response that looks similar to what is shown above. Here, we are going to copy the Call link somewhere. We will use this later.

Essentially, with the settings we enabled, the link of the call will allow us to pull information from a forecasted weather report for tomorrow (hence days=1), at London. We will make it so we can change the weather data that we grab through this link in the code.

Beside the point, we need to make interfaces that will allow us to declare type variable for TypeScript that we can use later on when fetching data from our code.

Make a new directory named shared under the project directory, then make a file named interfaces.ts. This is where we will be making interfaces, which set types to variables that we will be using.

In interfaces.ts, add the following code:

```
export interface ForecastParams {
   cityName: string;
   days: number;
```

```
}

export interface LocationsParams {
   cityName: string;
}
```

Here, we are assigning parameters with their respective types.

**10. Creating the weatherAPI.ts file**

Finally, we are going to get into coding API calls, which are references to our API to collect data.

To start, we must make a directory named utils and create a file named weatherAPI.ts. Now add some import statements into weatherAPI.ts to achieve our goals:

```
import axios from "axios";
import { apiKey } from "../constants";
import { ForecastParams, LocationsParams } from "../shared/interfaces";
```

Axios is a JavaScript library that is used in API calls to send http requests, which in our case gets the data from the API.

`apiKey` is the variable that we are importing from the constants file, which holds our API key.

`ForecastParams` and `LocationsParams` are both the interfaces that we created, and will be using to hold variable types.

```
const forecastEndpoint = (params : ForecastParams): string => {
   return
`http://api.weatherapi.com/v1/forecast.json?key=${apiKey}&q=${params.cityName}&
days=${params.days}&aqi=no&alerts=no`
};
const locationsEndpoint = (params: LocationsParams): string => {
   return
`http://api.weatherapi.com/v1/search.json?key=${apiKey}&q=${params.cityName}`
};
```

Add this code below, which defines functions that generate endpoint URLs for WeatherAPI HTTP requests. The `forecastEndpoint` function constructs a URL for retrieving forecast data, while the `locationsEndpoint` function generates a URL for location-based search results.

We set the `params` to type `ForecastParams` and `LocationsParams` respectively (this is necessary in TypeScript).

Furthermore, we use ${} in the return strings so we can modify the http requests with our own variables. This will allow us to search up different locations and different days, rather than being stuck with London and days=1.

```
const apiCall = async (endpoint: string) => {
   const options = {
       method: 'GET',
       url: endpoint
   }
   try{
       const response = await axios.request(options);
       return response.data;
   }
   catch(err) {
       console.log('error: ', err);
       return null;
   }
}

export const fetchWeatherForecast = (params: ForecastParams) => {
   return apiCall(forecastEndpoint(params));
}

export const fetchLocations = (params: LocationsParams) => {
   return apiCall(locationsEndpoint(params));
}
```

The `async` keyword is used to make the function asynchronous, which returns a Promise. The `await` keyword can also be used in an async function in order to delay the execution of a program until a Promise is reached. If an error is thrown during an async function, then a rejected Promise is returned.

This code creates an `apiCall` method that sends GET requests through axios. The try and catch statements are used to catch any errors and prevent the program from crashing in the case of one.

The Fetch methods at the bottom are used to return the data received from the GET request.

At last, we are finished with the weatherAPI.ts file. Congratulations, now you know how to make API calls using axios.

**11. Cookin up weatherDisplay.tsx**

Alright, sorry for all the directories, but we are creating another one in the project directory named components. Inside of components, make a file named weatherDisplay.tsx.

Let's begin by adding imports that we will be using in this file:

```
import React from "react";
import { useState, useEffect } from "react";
import { View, Text, TouchableOpacity, StyleSheet, TextInput, ActivityIndicator
} from "react-native";
import { ForecastParams, LocationsParams } from "../shared/interfaces";
import { fetchLocations, fetchWeatherForecast } from "../utils/weatherAPI";
```

Next, we will develop the WeatherDisplay function right under it, which will end up displaying the API data we fetch as text.

```
export default function WeatherDisplay({ cityName }: { cityName: string }) {
   const [loading, setLoading] = useState<boolean>(true);
   const [error, setError] = useState<string | null>(null);
   const [temperature, setTemperature] = useState<number | null>(null);
    useEffect(() => {
     const getWeather = async () => {
       try {
         setLoading(true);
         setError(null);
```

```
        const params: ForecastParams = {cityName, days: 3};
        const data = await fetchWeatherForecast(params);
         if (data) {
          setTemperature(data.current.temp_f);
        } else {
          setError("Failed to featch weather data");
        }
      }
      catch (err) {
        setError("An error occured while fetching data");
      }
      finally {
        setLoading(false);
      }
    };
    getWeather();
  }, [cityName]); // fetch data when cityName is changed

  return ();
}
```

Starting with the function declaration, `export default function WeatherDisplay({ cityName }: {cityName: string}),` which has the prop cityName and the colon is used to declare cityName's type. We use parameters in functions if we want to be able to feed a variable from a different area to be used inside a function.

```
  const [loading, setLoading] = useState<boolean>(true);
  const [error, setError] = useState<string | null>(null);
  const [temperature, setTemperature] = useState<number | null>(null)
```

The code snippet above is an example of React state declarations. For the sake of explanation, I will only refer to the first line when explaining what state declaration is.

const is used to declare loading as a constant state variable, which is going to track if the data is loading. `setLoading` is a function that is created to update the value/status of the loading state variable.

We set `const [loading, setLoading]` equal to `useState<boolean>(true);` to declare that we are doing state declarations.

`<boolean>` refers to the type of the state variable, and `(true)` sets the default value of loading to true. You may also see `<string | null>`, which means that the type is either of the two. Fun fact, this is called a Union type!

```
useEffect(() => {
  const getWeather = async () => {
    try {
      setLoading(true);
      setError(null);


      const params: ForecastParams = {cityName, days: 3};
      const data = await fetchWeatherForecast(params);
       if (data) {
        setTemperature(data.current.temp_f);
      } else {
        setError("Failed to featch weather data");
      }
    }
    catch (err) {
      setError("An error occured while fetching data");
    }
    finally {
      setLoading(false);
    }
  };
  getWeather();
}, [cityName]); // fetch data when cityName is changed
```

`useEffect` is a hook that will call getWeather for us whenever there is a change to the cityName. Next, we ensure that loading is set to true and error is set to null so we do not run into any issues from prior API calls. We made the getweather function asynchronous because it fetches data.

Next, we set params to cityName and the day that we want to forecast. In this case, we will put 3 days. Then we set `data` = to `fetchWeatherForecast(params)` which will give us the forecast data.

The `if (data)` statement checks if data exists, if not it will set an error rather than set the temperature to the current temperature. Finally will execute after the catch to set loading to false.

The end of the code snippet includes a `[cityName]` dependency array, which makes the getWeather function only call itself if there is a name in the cityName variable.

After completing the getWeather method, we can start populating the return statement with some code:

```
  return (
    <View style={styles.location}>
      {loading ? (
        <ActivityIndicator size="large" color="#0000ff" />
      ) : error ? (
        <Text>{error}</Text>
      ) : cityName && temperature !== null ? (
        <Text>City: {cityName} Temperature: {temperature}°</Text>
      ) : (
        <Text>No data available</Text>
      )}
    </View>
  );
```

This return statement will return text that is dependent on the state of the program. For example, there are different displays depending on if the application is loading, has an error, has the necessary data, or does not have any data available.

The term for this formatting to check the conditions is known as a ternary expression. Below is a simple format for ternary expressions:

```
condition ? value_if_true : value_if_false
```

Here are some styles that are used in our return statement that I made for you:

```
const styles = StyleSheet.create({
```

```
  container: {
    paddingTop: 70,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#f0f0f0",
  },
  title: {
    fontSize: 24,
    fontWeight: "bold",
    justifyContent: "center",
  },
  text_entry: {
    fontSize: 16,
    color: "#333",
    alignContent: "center",
  },
  separator: {
    justifyContent: "center",
    alignItems: "center",
    marginVertical: 20,
    paddingTop: 80,
    fontSize: 80,
  },
  location: {
    justifyContent: "center",
    alignItems: "center",
    marginVertical: 20,
  },
```

Once you are finished with this step, the final code should look like this:

```
import React from "react";
import { useState, useEffect } from "react";
import { View, Text, TouchableOpacity, StyleSheet, TextInput, ActivityIndicator
} from "react-native";
import { ForecastParams, LocationsParams } from "../shared/interfaces";
import { fetchLocations, fetchWeatherForecast } from "../utils/weatherAPI";

export default function WeatherDisplay({ cityName }: { cityName: string }) {
  const [loading, setLoading] = useState<boolean>(true);
  const [error, setError] = useState<string | null>(null);
  const [temperature, setTemperature] = useState<number | null>(null);
   useEffect(() => {
```

```
    const getWeather = async () => {
      try {
        setLoading(true);
        setError(null);


        const params: ForecastParams = {cityName, days: 3};
        const data = await fetchWeatherForecast(params);
         if (data) {
          setTemperature(data.current.temp_f);
        } else {
          setError("Failed to featch weather data");
        }
      }
      catch (err) {
        setError("An error occured while fetching data");
      }
      finally {
        setLoading(false);
      }
    };
    getWeather();
  }, [cityName]); // fetch data when cityName is changed

  return (
    <View style={styles.location}>
      {loading ? (
        <ActivityIndicator size="large" color="#0000ff" />
      ) : error ? (
        <Text>{error}</Text>
      ) : cityName && temperature !== null ? (
        <Text>City: {cityName} Temperature: {temperature}°</Text>
      ) : (
        <Text>No data available</Text>
      )}
    </View>
  );
}
 const styles = StyleSheet.create({
  container: {
    paddingTop: 70,
    justifyContent: "center",
```

```
    alignItems: "center",
    backgroundColor: "#f0f0f0",
  },
  title: {
    fontSize: 24,
    fontWeight: "bold",
    justifyContent: "center",
  },
  text_entry: {
    fontSize: 16,
    color: "#333",
    alignContent: "center",
  },
  separator: {
    justifyContent: "center",
    alignItems: "center",
    marginVertical: 20,
    paddingTop: 80,
    fontSize: 80,
  },
  location: {
    justifyContent: "center",
    alignItems: "center",
    marginVertical: 20,
  },
});
```

## 12. Working on the Search Bar

Go back to your components directory, and create a searchBar.tsx file.

Here are the import statements that we will be using:

```
import React, { useState } from 'react';
import { View, Text, TextInput, StyleSheet } from "react-native";
```

Pretty simple, right! :D ***thank god

Moving forward, we will create the Search Bar function:

```
export default function SearchBar({ onSearch }: {onSearch: (text: string) =>
void}) {
   const [searchText, setSearchText] = useState<string>("");
```

```
    const handleInputChange = (text: string) => {
        setSearchText(text); // update state to input string when input changes
        onSearch(text); // sends input to parent component
        console.log("You typed:", text)
    };

    return (
        <View style={styles.search_bar}>
            <TextInput style={styles.input} placeholder="Enter City Name"
            value={searchText} onChangeText={setSearchText} // local editing
            onSubmitEditing={() => onSearch(searchText)} // search only on enter
            returnKeyType="search" // changes keyboard to show search button
            />
        </View>
    )
}
```

Starting with the first line, we pass a prop onSearch into the SearchBar, which will be a string. The => declares the return type of of the function, void.
Next, we have a React state declaration for searchText, which has a default value of empty string.

Afterward, there is a function named `handleInputChange`. This function takes a string as an argument and sets the searchText to it, it also calls onSearch with text as its argument.

The `console.log` line is just used to show what is typed in the console for debugging purposes.

After creating the function for handling input, we will populate the the return statement with React UI components.

`<TextInput>` has a variety of settings to it; `placeholder` sets the text that will appear when there is no text in the search bar, `value` links the input to the state searchText, `onChangeText` sets the searchText, `onSubmitEditing` calls onSearch when enter key is pressed (this is put in place so the cityName is not changed on every keystroke), and `returnKeyType` changes the keyboard to show a search button.

Add the following styles I provide, and the code will look like this:

```
import React, { useState } from 'react';
import { View, Text, TextInput, StyleSheet } from "react-native";

export default function SearchBar({ onSearch }: {onSearch: (text: string) =>
void}) {
   const [searchText, setSearchText] = useState<string>("");

   const handleInputChange = (text: string) => {
       setSearchText(text); // update state to input string when input changes
       onSearch(text); // sends input to parent component
       console.log("You typed:", text)
   };

   return (
       <View style={styles.search_bar}>
           <TextInput style={styles.input} placeholder="Enter City Name"
           value={searchText} onChangeText={setSearchText} // local editing
           onSubmitEditing={() => onSearch(searchText)} // search only on enter
           returnKeyType="search" // changes keyboard to show search button
           />
       </View>
   )
}

const styles = StyleSheet.create({
   search_bar: {
       flexDirection: "row",
       alignItems: "center",
       backgroundColor: "gray",
       width: "90%",
       marginVertical: 10,
       alignSelf: "center",
       borderRadius: 5,
       height: 50,
       paddingLeft: 3,
     },
   input: {
       height: 40,
       width: "80%",
       borderColor: "gray",
```

```
        borderWidth: 1,
        borderRadius: 5,
    }
});
```

### 13. Modifying the frontScreen.tsx file

At long last, we have finished developing the components for the program. It is time to show you how components are applied in React Native.

Add the remaining imports and modify the return statement of FrontScreen:

```
import React from "react";
import { useState, useEffect } from "react";
import { View, Text, StyleSheet } from "react-native";
import WeatherDisplay from "../components/weatherDisplay";
import SearchBar from "../components/searchBar";


export default function FrontScreen() {
 const [cityName, setCityName] = useState<string>("San Diego");

 return (
   <View style={styles.container}>
     <Text style={styles.title}>Weather</Text>
     <View style={styles.search_bar} >
       <SearchBar onSearch={setCityName} />
     </View>
     <WeatherDisplay cityName={cityName}/>
   </View>
 );
}
```

As you can see, we implemented the SearchBar component and the WeatherDisplay component. We also added a React Native state declaration for the cityName.

Add the following styles and your file should look like this:

```
import React from "react";
import { useState, useEffect } from "react";
import { View, Text, StyleSheet } from "react-native";
import WeatherDisplay from "../components/weatherDisplay";
import SearchBar from "../components/searchBar";
```

# APP DEVELOPMENT CLUB - REACT NATIVE WEATHER APP

```
export default function FrontScreen() {
 const [cityName, setCityName] = useState<string>("San Diego");

 return (
   <View style={styles.container}>
     <Text style={styles.title}>Weather</Text>
     <View style={styles.search_bar} >
       <SearchBar onSearch={setCityName} />
     </View>
     <WeatherDisplay cityName={cityName}/>
   </View>
 );
}

const styles = StyleSheet.create({
 container: {
   paddingTop: 70,
   justifyContent: "center",
   alignItems: "center",
   backgroundColor: "#f0f0f0",
 },
 title: {
   fontSize: 24,
   fontWeight: "bold",
   justifyContent: "center",
 },
 search_bar: {
   flexDirection: "row",
   alignItems: "center",
   backgroundColor: "gray",
   width: "90%",
   marginVertical: 10,
   alignSelf: "center",
   borderRadius: 5,
   height: 50,
 },
});
```

Since all of these components were added to FrontScreen, which is called in App.tsx, your changes should be present on the emulator!

Congrats on finishing part A of the Weather App lab, where we learned to pull API data and display it on an application using a Search Bar!

The next lab will be focused on updating the User Interface to make it look pretty.

# PART 2

**DISCLAIMER: I am not using the UI from Justin's UI Lab. I decided to style on my own, so the following code that I add is a direct addition to part 1 of this lab. I did not make any of the new files that Justin made in his lab.**

In part 2, we will be adding some more functionality to our program by adding appropriate images to represent weather conditions and add some auto completion tools to the search bar.

1. **Deeper dive into API calls and JSON files**
   In this section, we are changing up the API calls to extract current data instead of forecast data. Furthermore, we will display an image depending on the weather conditions outside.

   First, let's look at the documentation for our weather API once again. Here, we can take note of the different requests that we can make:

Request

**Request URL**

Request to WeatherAPI.com API consists of base url and API method. You can make both HTTP or HTTI

Base URL: http://api.weatherapi.com/v1

| API | API Method |
|---|---|
| Current weather | /current.json or /current.xml |
| Forecast | /forecast.json or /forecast.xml |
| Search or Autocomplete | /search.json or /search.xml |
| History | /history.json or /history.xml |
| Alerts | /alerts.json or /alerts.xml |
| Marine | /marine.json or /marine.xml |
| Future | /future.json or /future.xml |
| Time Zone | /timezone.json or /timezone.xml |
| Sports | /sports.json or /sports.xml |
| Astronomy | /astronomy.json or /astronomy.xml |
| IP Lookup | /ip.json or /ip.xml |

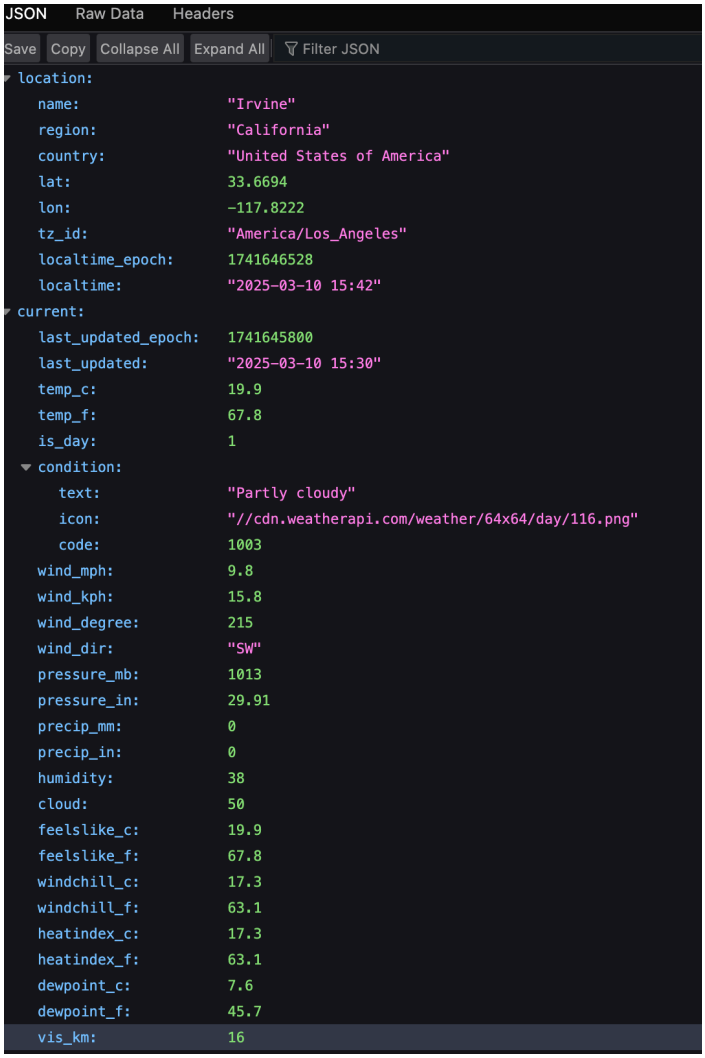
Since we were previously using json requests in the past, we will stick with those.

Next, we will go to our weatherAPI.ts file and edit our code to correctly represent our collection of current weather data.

```
import axios from "axios";
import { apiKey } from "../constants";
import { ForecastParams, CurrentParams } from "../shared/interfaces";


const forecastEndpoint = (params : ForecastParams): string => {
   return
`http://api.weatherapi.com/v1/forecast.json?key=${apiKey}&q=${params.cityName}&
days=${params.days}&aqi=no&alerts=no`
};
const currentEndpoint = (params: CurrentParams): string => {
   return
`http://api.weatherapi.com/v1/current.json?key=${apiKey}&q=${params.cityName}`
};

const apiCall = async (endpoint: string) => {
```

```
    const options = {
        method: 'GET',
        url: endpoint
    }
    try{
        const response = await axios.request(options);
        return response.data;
    }
    catch(err) {
        console.log('error: ', err);
        return null;
    }
}

export const fetchWeatherForecast = (params: ForecastParams) => {
    return apiCall(forecastEndpoint(params));
}

export const fetchWeatherCurrent = (params: CurrentParams) => {
    return apiCall(currentEndpoint(params));
}
```

As you may have noticed, we changed every name that included "Location" and made it say "Current." Furthermore, we changed the http request link to say current.json. If we paste that link, without the quotes and replace the blue text with our actual API key and city name, into our web browser, we will see this:




Now, you can see how we pull data earlier in the lab using `data.temp_f`. This image will be our guide to understand what data we are actually pulling from the API and what keywords we should use to locate certain information.

2. **Implementing new API calls**
   Finally, we understand what a JSON file is made up of, and we can pull any specific data that we need from our JSON. So, with this knowledge, we shall edit weatherDisplay.tsx once again.

   Add the following useState hooks to your WeatherDisplay function, right below your other useState hooks:

```
   const [weatherImage, setWeatherImage] =
useState<string>("../assets/images/CloudsSun.png");
   const [weatherQuality, setWeatherQuality] = useState<string>("Partly
Cloudy");
```

Also make sure that all references to fetchLocation and LocationParams are changed to fetchWeatherCurrent and CurrentParams (since we changed those variables earlier).

Next, we are going to comment out our declaration of params to ForecastParams for the time being to fix the functionality of displaying current weather conditions.

Go to your useEffect hook and add any new lines:

```
useEffect(() => {
     const getWeather = async () => {
       try {
         setLoading(true);
         setError(null);



         //const params: ForecastParams = {cityName, days: 3};
         const params: CurrentParams = {cityName};
         const data = await fetchWeatherCurrent(params);
          if (data) {
           setTemperature(data.current.temp_f);
           setWeatherImage(data.current.condition.icon) // this value will be a
string that has the image of the weather
           setWeatherQuality(data.current.condition.text) // holds weather
quality as a string
         } else {
           setError("Failed to fetch weather data");
         }
       }
       catch (err) {
         setError("An error occured while fetching data");
       }
       finally {
         setLoading(false);
       }
```

```
    };
    getWeather();
  }, [cityName]); // fetch data when cityName is changed
```

As you can see, we made use of our new useState hooks to set the weather image and weather quality to their respective variables. We will now add these to a View component so they will be displayed on our phone.

In the return statement, we are going to add a <View>, <Text>, and <Image> component: Warning, you might get an error when using Image because you may have not imported it yet at the top of the file, do that now

```
  return (
    <View style={styles.location}>
      {loading ? (
        <ActivityIndicator size="large" color="#0000ff" />
      ) : error ? (
        <Text style={styles.location_text}>{error}</Text>
      ) : cityName && temperature !== null ? (
        <View style={{alignItems: "center"}}>
          <Image source={{ uri: `https:${weatherImage}` }} style =
{styles.image} />

          <Text style={styles.location_text}>
            {cityName}:  {temperature}°
          </Text>
          <Text style = {styles.location_text}>
            {weatherQuality}
          </Text>

        </View>
      ) : (
        <Text style={styles.location_text}>No data available</Text>
      )}
    </View>
  );
```

In the Image component, we use uri which allows us to have a dynamic URL, so the weather app can update the image appropriately, depending on the current weather conditions of a given location.

Moreover, we use the back ticks instead of quotes in order to allow us to insert ${weatherImage} as a variable into the url string.

The 'https:' that lies before the call to the weatherImage variable is prepending https: to the incomplete url that is retrieved from the JSON file.

```
"//cdn.weatherapi.com/weather/64x64/day/116.png"
```

Below, I am going to attach the styles that I am using for the Text, Image, and View components. But, you do not have to copy these exactly if you want your display to look different:

```
const styles = StyleSheet.create({
  container: {
    paddingTop: 70,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#f0f0f0",
  },
  title: {
    fontSize: 24,
    fontWeight: "bold",
    justifyContent: "center",
  },
  text_entry: {
    fontSize: 16,
    color: "#333",
    alignContent: "center",
  },
  separator: {
    justifyContent: "center",
    alignItems: "center",
    marginVertical: 20,
    paddingTop: 80,
    fontSize: 80,
  },
  location: {
    justifyContent: "center",
    alignItems: "center",
    marginVertical: 20,
```

```
  },
  location_text: {
    color: "white",
    fontSize: 20,
    fontWeight: "bold",
    alignContent: "center",
  },
  image: {
    width: 200,
    height: 200,
  },
```

Now, go back to weatherDisplay and comment CurrentParams, while uncommenting ForecastParams and changing the fetch to fetchWeatherForecast. As you can see, you are still able to pull current data from the forecast url.

3. **Adding future forecast displays**

   The goal of this section is to have two little icons that represent the weather in the next two days, followed by the average temperature. We will implement this in weatherDisplay.tsx.

   First, we will make an array that will hold the next two forecast days.

```
const [forecasts, setForecasts] = useState<ForecastDay[]>([]);
```

   Next, we will create an array that retrieves that values that will be put in these arrays, then assigns the array to the useState hook. We are going to put the following code in our useEffect hook under the if statement that checks if data exists. Also, I added another condition to the if statement to check if forecasted data exists:

```
if (data && data.forecast.forecastday) {
            setTemperature(data.current.temp_f);
            setWeatherImage(data.current.condition.icon) // this value will be a
string that has the image of the weather
            setWeatherQuality(data.current.condition.text) // holds weather
quality as a string

            const forecastData: ForecastDay[] =
data.forecast.forecastday.slice(1,3).map((day: { day: { avgtemp_f: number;
condition: { icon: string } }}) => ({
              temp: day.day.avgtemp_f, // slice takes each of the next two days,
map turns data into temp and image
              image: `https:${day.day.condition.icon}`
            }));
            setForecasts(forecastData); // sets array of forecasted data into
forecasts array
```

   In TypeScript, this code looks like a headache because of all the type declarations for the days. Fortunately, it prevents type mismatching when getting jumbled in all of these different variables.

   In the arrow function forecastData, we are grabbing two elements from data.forecast.forecastday (through slice(1,3) which takes tomorrow and the next day).

Additionally, .map iterates over each day that we selected, returning temp and image. The temp and image from each day are inputted into the forecastData array, which is then assigned to the forecast state.

Now, you will receive an error because you have not yet declared an interface for the ForecastDay type. Go to interfaces.ts and add the following code:

```
export interface ForecastDay {
   temp: number;
   image: string;
}
```

Remember to import this interface into weatherDisplay.tsx!

Once we finish grabbing the data for the forecasted days, we will make this data visible on our mobile app. Go back to weatherDisplay.tsx and add the following code to the return statement:

```
      ) : cityName && temperature !== null ? (
        <View style={{alignItems: "center"}}>
          <Image source={{ uri: `https:${weatherImage}` }} style =
{styles.image} />

          <Text style={styles.location_text}>
            {cityName}:  {temperature}°
          </Text>
          <Text style = {styles.location_text}>
            {weatherQuality}
          </Text>

          <View style={{flexDirection: "row", paddingTop: 70}}>
            {forecasts.map(({temp, image}, index) => (
              <View key={index} style={{paddingHorizontal: 50, alignItems:
"center"}}>
                <Image source ={{ uri: image }} style={{height:50,width:50}}
/>
                <Text style={styles.location_text}>{temp}°</Text>
                <Text style={styles.location_text}>in {index + 1} {index === 0
? "day" : "days"}</Text>
              </View>
            ))}
          </View>
        </View>
```

```
        ) : (
```

Only add the new parts of course. In this code snippet, I am creating a <View> component to hold the new components that we will be adding, and display it in a row (hence the flexDirection: "row" statement).

Then, we are making a function for each index of forecast to have its own View component that displays the information for each of the next two days.

Once you finish implementing this code and save your changes, your app should look similar to this:

4. **Adding Dropdown Menu**

   Make a new file named dropdownMenu.tsx inside the components section, then add the following code:

```
import React from "react";
import { useState, useEffect } from "react";
import { View, Text, Image, StyleSheet, TouchableOpacity, ActivityIndicator }
from "react-native";
import { SearchParams, SearchResults } from "../shared/interfaces";
import { fetchLocationSearch } from "../utils/weatherAPI";

interface dropDownProps {
   searchText: string;
   setSearchText: (text: string) => void;
   onClick: (text: string) => void;
}

export default function DropdownMenu({onClick, searchText, setSearchText}:
dropDownProps) {
   const [cities, setCities] = useState<SearchResults[]>([]);
   const [loading, setLoading] = useState<boolean>(false);
   const [error, setError] = useState<string | null>(null);

   useEffect (() => {
       const getSearches = async () => {
           if (!searchText.trim()) return; // Avoid unnecessary API calls

           try {
               setLoading(true);
               setError(null);

               const params: SearchParams = {searchText};
               const data = await fetchLocationSearch(params);
               console.log(data);

               if (data) {
                   const availableCities: SearchResults[] =
data.slice(0,3).map((city: SearchResults) => ({
                       name: city.name,
                       region: city.region
                   }));
```

```
                setCities(availableCities);
            }
            else {
                setError("");
            }
        }
        catch (error) {
            setError("Failed to find search data");
        }
        finally {
            setLoading(false);
        }
    };
    getSearches();
}, [searchText]);

const handleCityClick = (name: string) => {
    console.log("City clicked:", name);
    onClick(name); // sends input to parent component
    setSearchText(name);
}

return (
    <View style={{flexDirection: "column"}}>
        {loading ? (
            <ActivityIndicator size="large" color="#0000ff" />
        ) : error ? (
            <Text>{error}</Text>
        ) : cities.length > 0 ? (
            cities.map((city, index) => (
            <TouchableOpacity key={index} style={styles.box} onPress={() =>
handleCityClick(city.name)}>
                <Text style={styles.cityText}>
                    {city.name}, {city.region}
                </Text>
            </TouchableOpacity>
            ))
        ) : (
            <Text>
                No results found
            </Text>
        )}
```

```
        </View>
    );
}

const styles = StyleSheet.create({
    cityText: {
        color: "white",
        fontSize: 14,
        fontWeight: "bold",
        alignContent: "center",
    },
    box: {
        backgroundColor: "#3498db",  // Blue color
        paddingVertical: 12,
        paddingHorizontal: 20,
        borderRadius: 10,           // Rounded corners
        alignItems: "center",
    }
})
```

In this code, we are taking the first three cities from the data that is gathered from weatherAPI's autocompletion JSON and adding it to an array named cities.

After we place each city into the cities array, we are adding a clickable box using the TouchableOpacity component, which will set the input of our search to the corresponding city name of the box that we clicked on.

Furthermore, we style these boxes so they fit the design of our program.

5. **Changing other files to be compatible**
   Compatibility issues may arise in our code due to some undefined types. Go to your interfaces.ts file and make sure that the code looks like this:

```
export interface ForecastParams {
    cityName: string;
    days: number;
}

export interface CurrentParams {
    cityName: string;
```

```
}

export interface SearchParams {
   searchText: string;
}

export interface SearchResults {
   name: string;
   region: string;
}

export interface ForecastDay {
   temp: number;
   image: string;
}
```

This is our file that defines most of our interfaces.

Now, ensure that your weatherDisplay.tsx file is the same:

```
import React from "react";
import { useState, useEffect } from "react";
import { View, Text, Image, StyleSheet, TextInput, ActivityIndicator } from
"react-native";
import { ForecastParams, CurrentParams, ForecastDay } from
"../shared/interfaces";
import { fetchWeatherCurrent, fetchWeatherForecast } from
"../utils/weatherAPI";

export default function WeatherDisplay({ cityName }: { cityName: string }) {
   const [loading, setLoading] = useState<boolean>(true);
   const [error, setError] = useState<string | null>(null);
   const [temperature, setTemperature] = useState<number | null>(null);
   const [weatherImage, setWeatherImage] =
useState<string>("../assets/images/CloudsSun.png");
   const [weatherQuality, setWeatherQuality] = useState<string>("Partly
Cloudy");
   const [forecasts, setForecasts] = useState<ForecastDay[]>([]);
    useEffect(() => {
     const getWeather = async () => {
       try {
         setLoading(true);
```

```
        setError(null);



        const params: ForecastParams = {cityName, days: 6};
        //const params: CurrentParams = {cityName};
        const data = await fetchWeatherForecast(params);
         if (data && data.forecast.forecastday) {
          setTemperature(data.current.temp_f);
          setWeatherImage(data.current.condition.icon) // this value will be a
string that has the image of the weather
          setWeatherQuality(data.current.condition.text) // holds weather
quality as a string

          const forecastData: ForecastDay[] =
data.forecast.forecastday.slice(1,3).map((day: { day: { avgtemp_f: number;
condition: { icon: string } }}) => ({
            temp: day.day.avgtemp_f, // slice takes each of the next two days,
map turns data into temp and image
            image: `https:${day.day.condition.icon}`
          }));
          setForecasts(forecastData); // sets array of forecasted data into
forecasts array

        } else {
          setError("Failed to fetch weather data");
        }
      }
      catch (err) {
        setError("An error occured while fetching data");
      }
      finally {
        setLoading(false);
      }
    };
    getWeather();
  }, [cityName]); // fetch data when cityName is changed

  return (
    <View style={styles.location}>
      {loading ? (
        <ActivityIndicator size="large" color="#0000ff" />
      ) : error ? (
```

# APP DEVELOPMENT CLUB - REACT NATIVE WEATHER APP

```
        <Text style={styles.location_text}>{error}</Text>
      ) : cityName && temperature !== null ? (
        <View style={{alignItems: "center"}}>
          <Image source={{ uri: `https:${weatherImage}` }} style =
{styles.image} />

          <Text style={styles.location_text}>
            {cityName}:  {temperature}°
          </Text>
          <Text style = {styles.location_text}>
            {weatherQuality}
          </Text>

          <View style={{flexDirection: "row", paddingTop: 70}}>
            {forecasts.map(({temp, image}, index) => (
              <View key={index} style={{paddingHorizontal: 50, alignItems:
"center"}}>
                <Image source ={{ uri: image }} style={{height:50,width:50}}
/>
                <Text style={styles.location_text}>{temp}°</Text>
                <Text style={styles.location_text}>in {index + 1} {index === 0
? "day" : "days"}</Text>
              </View>
            ))}
          </View>
        </View>
      ) : (
        <Text style={styles.location_text}>No data available</Text>
      )}
    </View>
  );
}
 const styles = StyleSheet.create({
  container: {
    paddingTop: 70,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#f0f0f0",
  },
  title: {
    fontSize: 24,
    fontWeight: "bold",
```

```
    justifyContent: "center",
  },
  text_entry: {
    fontSize: 16,
    color: "#333",
    alignContent: "center",
  },
  separator: {
    justifyContent: "center",
    alignItems: "center",
    marginVertical: 20,
    paddingTop: 80,
    fontSize: 80,
  },
  location: {
    justifyContent: "center",
    alignItems: "center",
    marginVertical: 160,
  },
  location_text: {
    color: "white",
    fontSize: 20,
    fontWeight: "bold",
    alignContent: "center",
  },
  image: {
    width: 200,
    height: 200,
  },
});
```

Some adjustments were made to change where the weather is displayed to make room for the dropdown menu.

I also changed the searchbar props to make it so the searchbar accurately reflects changes made by the dropdown menu. Go to searchBar.tsx:

```
import React, { useState } from 'react';
import { View, Text, TextInput, StyleSheet } from "react-native";

interface SearchBarProps {
   searchText: string;
```

```
    setSearchText: (text: string) => void;
    onSearch: (text: string) => void;
}

export default function SearchBar({ onSearch, searchText, setSearchText}:
SearchBarProps) {

    const handleInputChange = (text: string) => {
        setSearchText(text); // update state to input string when input changes
        onSearch(text); // sends input to parent component
        console.log("You typed:", text)
    };

    return (
        <View style={styles.search_bar}>
            <TextInput placeholder="Enter City Name"
            value={searchText} onChangeText={setSearchText} // local editing
            onSubmitEditing={() => onSearch(searchText)} // search only on enter
            returnKeyType="search" // changes keyboard to show search button
            />
        </View>
    )
}

const styles = StyleSheet.create({
    search_bar: {
        flexDirection: "row",
        alignItems: "center",
        backgroundColor: "#b5a1f8",
        width: "90%",
        marginVertical: 10,
        alignSelf: "center",
        borderRadius: 10,
        height: 50,
        paddingLeft: 3,
      },
});
```

Finally, change frontScreen.tsx to reflect the changes we made and implement the dropdown menu:

```
import React from "react";
```

# APP DEVELOPMENT CLUB - REACT NATIVE WEATHER APP

```
import { useState, useEffect } from "react";
import { View, Text, StyleSheet, Image } from "react-native";
import WeatherDisplay from "../components/weatherDisplay";
import SearchBar from "../components/searchBar";
import DropdownMenu from "../components/dropdownMenu";


export default function FrontScreen() {
 const [cityName, setCityName] = useState<string>("San Diego");
 const [searchText, setSearchText] = useState<string>("San Diego")

 return (
   <View style={styles.container}>
     <Text style={styles.title}>Weather</Text>
     <View style={styles.search_bar} >
       <SearchBar onSearch={setCityName} searchText={searchText}
setSearchText={setSearchText}/>
       {cityName.length > 0  && <DropdownMenu onClick={setCityName}
searchText={searchText} setSearchText={setSearchText}/> }
     </View>
     <WeatherDisplay cityName={cityName} />
   </View>
 );
}

const styles = StyleSheet.create({
 container: {
   //justifyContent: "center",
   paddingTop: 60,
   alignItems: "center",
   backgroundColor: "#424242",
   flex: 1,
 },
 title: {
   fontSize: 24,
   fontWeight: "bold",
   justifyContent: "center",
   color: "white",
 },
 search_bar: {
   flexDirection: "column",
   alignItems: "center",
```

```
  //backgroundColor: "#b5a1f8",
  width: "90%",
  //paddingTop: 10,
  alignSelf: "center",
  borderRadius: 10,
  height: 50,
},
image: {
  width: 200,
  height: 200,
},
temp: {
  fontSize: 160,
  color: "white",
  paddingTop: 50,
},
location: {
  fontSize: 50,
  color: "white",
},
horizontalLine: {
  height: 7,
  backgroundColor: "white",
  width: "100%",
  marginVertical: 10,
},
});
```

Congratulations, the weather app is complete!

If you need something to cross reference, below is the link to the repository of this project:

https://github.com/Dsneezy/weather-app-reactnative/