Building a Minimalist Weather App with React Native and Expo

Aman Mittal Follow
Mar 27, 2018 · 7 min read

React Native is a great framework to develop cross-platform mobile applications for the platforms iOS and Android. In this, I'm going to take you through the process of building a "minimalist" weather application using React Native by fetching real-time data. If you've never worked with React Native, you can use this walkthrough to kickstart your journey as a mobile application developer, and add a cool project to your portfolio.

Getting Started: Requirements

If you have some experience working your way with Reactjs, you'll have no problem following this tutorial. If you're anewbie to JavaScript or Reactjs ecosystem, I want to halt right here and go through this awesome resource that can help you with understanding the basic concepts in this tutorial. (Don't spend too much time if you're not interested in building web applications using Reactjs; just go through the nitty-gritty.)

Please note that React Native is not a hybrid mobile app framework like others available. It uses a bridge between Javascript and native APIs of a specific platform. Do take a look at React Native Official Docs to read more about this.

I'll be using Expo, which is described as "the fastest way to build an app". It's an open-source set of tools and services that comes in handy, especially if you're getting started in the React Native world. The development tool I am going to use for Expo is Expo XDE.

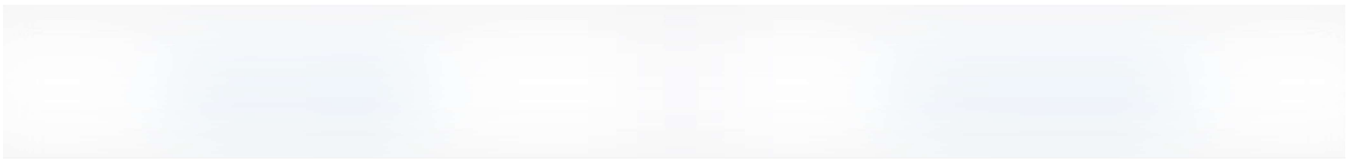
Requirements summary

- You know how to write JavaScript
- Familiar with React
- Nodejs installed on your local machine
- Simple `npm` commands

That's all. Let us get started with the development process.

Getting Started (for real this time)

Open the Expo XDE after its installation and click on "Create New Project":



Enter the name of your application and click "Create". The name of the application will be in lowercase (I don't know why; Expo XDE UI does not support uppercase characters).

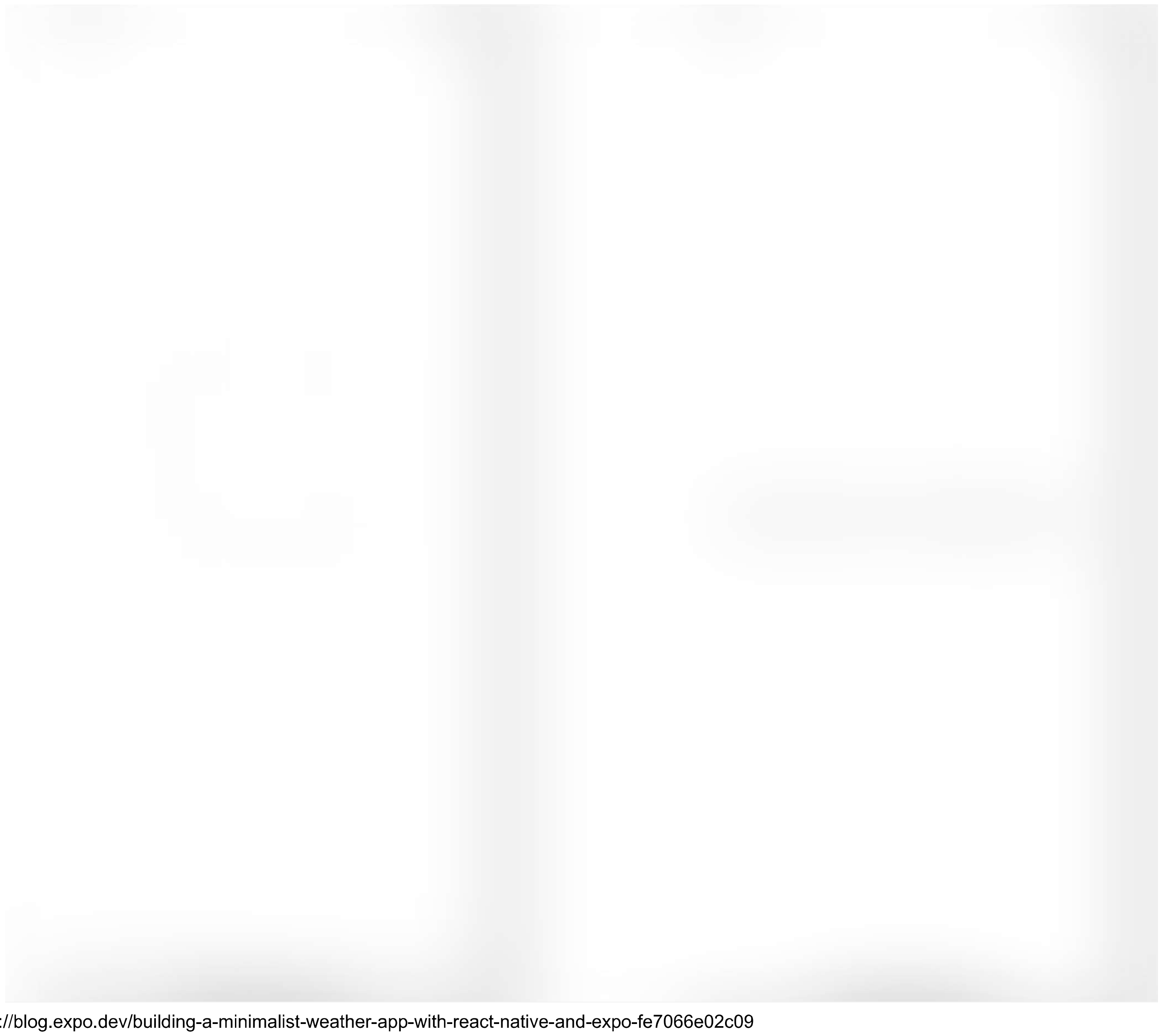
Behind the scenes, Expo's using React Native Package manager to simulate the application and the load dependencies from the app's `package.json` file. The benefit of using Expo XDE is that you don't have to open multiple terminal windows, and you can test the app while still developing on a real device. Once it's done generating a source code of our app, we can start it in a simulator on our local machine to preview the default app it comes with.

If you're on a Mac, make sure you have Xcode installed. If you're using Windows, please follow the instructions to install Android Studio to run the simulator.

If you want to skip simulating the app and run it on an actual device without generating any `.apk` or `.ipa`, install the Expo client and scan the QR code generated by Expo XDE.

Once the source code is done bundling, you'll be prompted with a success message in the Expo XDE terminal:

And you'll be able to see that our default app is running on the device:



The message displayed here is the same code that is rendered by `App.js` in the root of our app.

Change the `<Text>` to

and you'll see the output being rendered and the app is automatically reloaded live. (You don't have to refresh it to see the changes.)

This completes our first Getting Started step. In the next step, we will build a static prototype of what our app is going to look like.

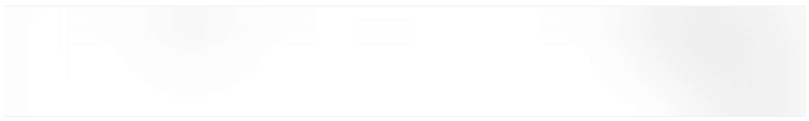
The Prototype

In this step, we develop our first screen, which will be a loading screen.

In your `App.js`, define a local state:

The code above states that when our local state object `isLoading` is false, we'll show the name of the application. This is what we're going to render. Later on, instead of displaying the name of application, we'll show the weather here once our API has successfully fetched the data. For now,let's stick to this message so we can first work on the question: what if our app is in the state of loading? Let's add the message text to indicate that the app is fetching the data.

When our app is done loading the data from the API, we set the state of `isLoading` to false.



First Screen

We’ll define a new Weather component at `./components/Weather.js`. The boilerplate code for every weather condition screen will be the same: it’ll be divided into two views, a header, and a body. The header will show the weather condition icon and temperature, and the body will display the text associated with the weather condition.

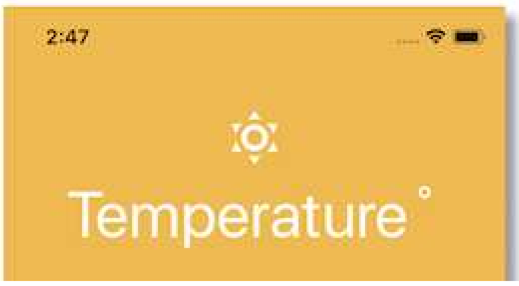
In Weather.js, we start by defining two containers inside the main container: `headerContainer` and `bodyContainer`. Do note that we’re defining `Weather` component not as a class but as a function, in order to receive props and since it won’t be managing a state.

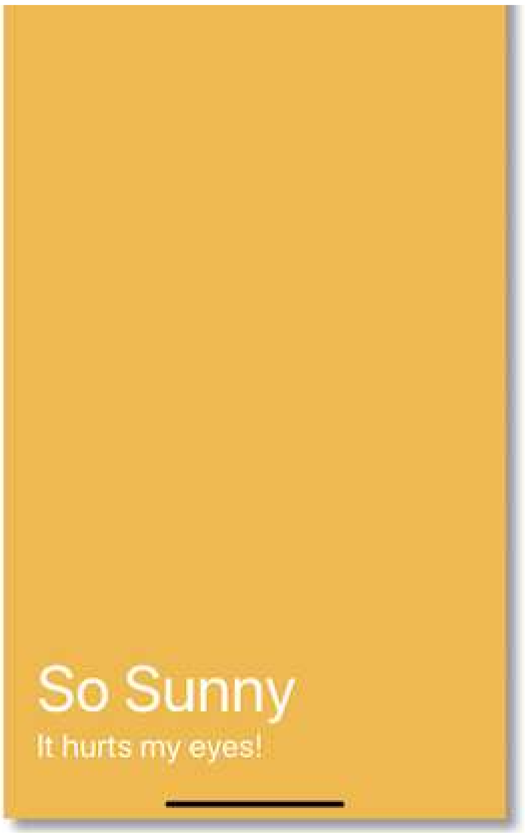
We'll be using `MaterialCommunityIcons` which comes with Expo (one of the perks) as a sub-library of a humongous library called `vector-icons`.

```
import React from 'react';
import { View, Text, StyleSheet } from 'react-native';
import { MaterialCommunityIcons } from '@expo/vector-icons';

const Weather = () => {
  return (
    <View style={styles.weatherContainer}>
      <View style={styles.headerContainer}>
        <MaterialCommunityIcons size={48} name="weather-sunny" color={'#fff'} />
        <Text style={styles.tempText}>Temperature˚</Text>
      </View>
      <View style={styles.bodyContainer}>
        <Text style={styles.title}>So Sunny</Text>
        <Text style={styles.subtitle}>It hurts my eyes!</Text>
      </View>
    </View>
    </View>
  );
};

const styles = StyleSheet.create({
  weatherContainer: {
    flex: 1,
    backgroundColor: '#f7b733'
  },
  headerContainer: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center'
  },
  tempText: {
    fontSize: 48,
    color: '#fff'
  },
  bodyContainer: {
    flex: 2,
    alignItems: 'flex-start',
    justifyContent: 'flex-end',
    paddingLeft: 25,
    marginBottom: 40
  },
  title: {
    fontSize: 48,
    color: '#fff'
  },
  subtitle: {
    fontSize: 24,
    color: '#fff'
  }
});

export default Weather;
```

black6.javascript hosted with ❤ by GitHub view raw

This how our app looks after the prototypal stage is complete:





Fetching The Data

To fetch real-time weather data, I found the Open Weather Map API to be highly useful and consistent. To communicate with the API, you'll need an API key (register yourself as a user on the site to get your API key). Please note that it takes at least 10 minutes for Open Weather API to activate the API key; once it's available, tag along.

Go to the API section, and you'll see that our need is satisfied by the Current Weather data. I'm going to store my API key in the `./utils/WeatherAPIKey.js` file (I know, not the best name for a file).

```javascript
export const API_KEY = 'YOUR_API_KEY HERE';
```

block7.javascript hosted with ❤ by GitHub view raw

The way the Open Weather API works is that we need to feed it longitude and latitude coordinates from our device's location. It'll then fetch the data from its server as a JSON object. From the server, we now need two things: the temperature, and the weather condition. We should have both stored in our local state in `App.js`.

```javascript
import React from 'react';
import { StyleSheet, Text, View, Animated } from 'react-native';

import { API_KEY } from './utils/WeatherAPIKey';

import Weather from './components/Weather';

export default class App extends React.Component {
  state = {
    isLoading: false,
    temperature: 0,
    weatherCondition: null,
    error: null
  };

  componentDidMount() {
    navigator.geolocation.getCurrentPosition(
      position => {
        this.fetchWeather(position.coords.latitude, position.coords.longitude);
      },
      error => {
        this.setState({
          error: 'Error Gettig Weather Condtions'
        });
      }
    );
  }

  fetchWeather(lat = 25, lon = 25) {
    fetch(
      `http://api.openweathermap.org/data/2.5/weather?lat=${lat}&lon=${lon}&APPID=${API_KEY}&ur
    )
      .then(res => res.json())
      .then(json => {
        console.log(json);
      });
  }

  render() {
    const { isLoading } = this.state;
    return (
      <View style={styles.container}>
        {isLoading ? <Text>Fetching The Weather</Text> : <Weather />}
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff'
  }
});
```

We start by importing the API key we just defined, then updating our state with `temperature`, `weatherCondition`, and `error`. We're using `componentDidMount()`, a lifecycle method which helps us re-render once our API is done fetching the data. It'll also help us in updating the state. We are also using JavaScript's `navigator` API to get the current location. (This is where a JavaScript API will communicate with a native one using a bridge.) We pass on the values of latitude and longitude to our custom function `fetchWeather`, where the API of Open Weather Map is called.

The result we get is in JSON format, and if you console log it, you'll be able to see the result as a JSON object in Expo terminal where there are a lot of values. We need only the temperature value and weather condition. We then update local state with the new values obtained. `&units=metric` at the end of our API call converts the temperature from Kelvin to Celsius.

Now, all we have to do is pass the two values of our local state as props to the `Weather` Component and then update it such that it can receive those props.

First, in `App.js`:

Update the `Weather.js`:

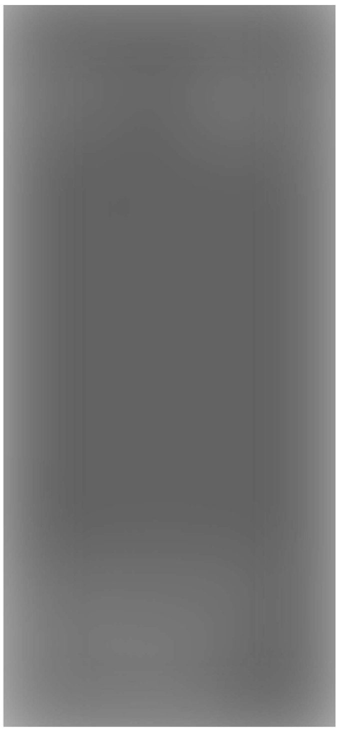
Since we’ve done the hard part of fetching the real-time data, we need to make our `Weather` component behave dynamically based on the values it’s getting. This entire dynamic behavior will be associated with `weatherCondition`.

Dynamic Behaviour

Using `weatherCondition`, we can define changes in our background, title, subtitle and weather icon. Let's start by pre-defining weather conditions in a file, `./utils/WeatherConditions.js`.

These weather conditions are provided from the Open Weather API. Then, let's import this file in our `Weather.js`. We'll also define PropTypes now for the two props we're receiving from `App.js`. Take a look below, it's simple:

Most of the source code is the same. We're now just making some additions by using available props with weather conditions to dynamically change the background, icon, weather name, and the subtitle. You can play around with the styling to make it look more minimalistic or more exquisite—it's up to you.



Note: Before running the application on your actual device, make sure you turn "on" internet access and location on the device for this app to work. (We haven't talked about App Permissions in this article, since it's a bit out of scope.)

Source & Example

The whole code for this application is available at this **Github Repo**.

. . .

Had fun learning through this tutorial? Follow our guest-blogger Aman Mittal on Twitter, on his blog or on his website! (This article is originally published on ZeoLearn.com.)

React Native Expo Mobile App Development JavaScript React