Getting Started with React Native

Weather App

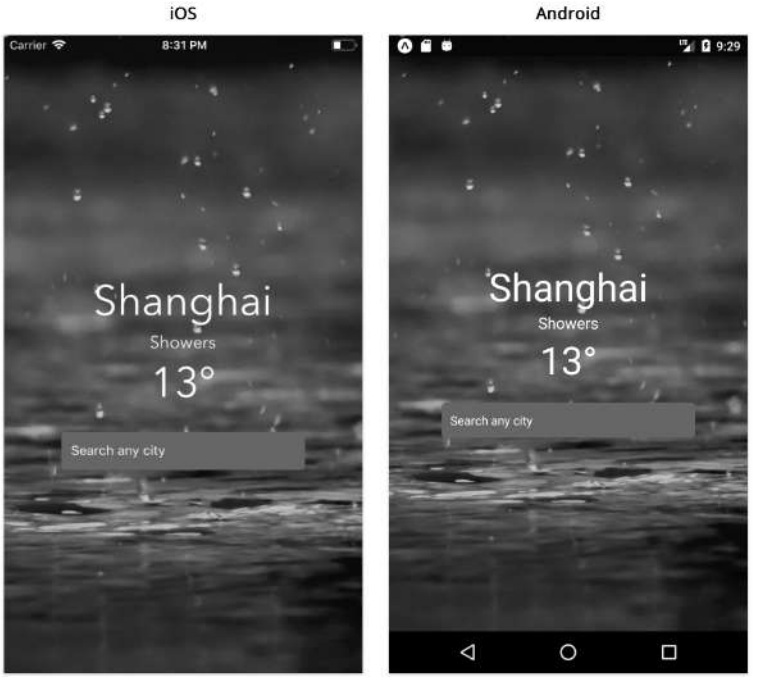
In this chapter we're going to build a weather application that allows the user to search for any city and view its current forecast.

With this simple app we'll cover some essentials of React Native including:

- Using core and custom components
- Passing data between components
- Handling component state
- Handling user input
- Applying styles to components
- Fetching data from a remote API

By the time we're finished with this chapter, you'll know how to get started by building a basic application with local state management. You'll have the foundation you need to build a wide variety of your own React Native apps.

Here's a screenshot of what our app will look like when it's done:



The completed app

In this chapter, we'll build an entire React Native application from scratch. We'll talk about how to set up our development environment and how to initialize a new React Native application. We'll also learn how Expo allows us to rapidly prototype and preview our application on our mobile device. After covering some of the basics of React Native, we'll explore how we compose apps using *components*. Components are a powerful paradigm for organizing views and managing dynamic data.

We're about to touch on a wide variety of topics, like styling and data management. This chapter will exhibit how all these topics fit together at a high-level. In subsequent chapters, we'll dive deep into the concepts that we touch on here.

Code examples

This book is example-driven. Each chapter is set up as a hands-on tutorial.

We'll be building apps from the ground up. Included with this book is a download that contains completed versions of each app as well as each of the versions we develop

along the way (the "sample code.") If you're following along, we recommend you use the sample code for copying and pasting longer examples or debugging unexpected errors. If you're not following along, you can refer to the sample code for more context around a given code example.

The structure of the sample code for all the chapters in this book follows this pattern:

```
├── components/
├── App.js
├── 1/
│   ├── components/
│   └── App.js
├── 2/
│   ├── components/
│   └── App.js
├── 3/
│   ├── components/
│   └── App.js
// ...
```

At the top-level of the directory is `App.js` and `components/`. This is the code for the completed version of the application. Inside the numbered folders (`1/`, `2/`, `3/`) are the different versions of the app as we build it up throughout the chapter.

Here's what a code example in this book looks like:

weather/1/App.js

```
  render() {
    return (
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Text>
      </View>
    );
  }
```

Note that the title of the code block contains the path within the sample code where you can find this example (`weather/1/App.js`).

JavaScript

This book assumes some JavaScript knowledge.

React Native uses Babel[12] as a JavaScript compiler to allow us to develop in the latest version of JavaScript, regardless of what version the underlying platform supports. To understand what we mean by JavaScript versions, you can refer to the Appendix.

We highlight some of JavaScript's newer features in the Appendix. We reference the appendix when relevant.

Starting the project

yarn

We can install all the required tools to begin our project by using `yarn`. `yarn`[13] is a JavaScript package manager – it *automates* the process of managing all the required dependencies from npm, an online repository of published JavaScript libraries and projects, in an application. This is done by defining all our dependencies in a single `package.json` file.

You can refer to the documentation[14] for instructions to install `yarn` for your operating system. The documentation also explains how to install node[15] as well. In order to use the Expo CLI however, `Node.js` v6 or later is required.

Here's a list of some commonly used `yarn` commands:

- `yarn init` creates a `package.json` file and adds it directly to our project.
- `yarn` installs all the dependencies listed in `package.json` into a local `node_modules` folder.
- `yarn add new-package` will install a specific package to our project as well as include it as a dependency in `package.json`. Dependencies are packages needed when we run our code.

[12] https://babeljs.io/
[13] https://yarnpkg.com
[14] https://yarnpkg.com/lang/en/docs/install
[15] https://nodejs.org/en/

- `yarn add new-package --dev` will install a specific package to our project as well as include it as a *development* dependency in `package.json`. Development dependencies are packages needed only during the development workflow. They are not needed for running our application in production.
- `yarn global add new-package` will install the package globally, rather than locally to a specific project. This is useful when we need to use a command line tool anywhere on our machine.

If you're already familiar with `npm` and have it installed, you may use it instead of `yarn` and run its equivalent commands[16]. These tools both manage dependencies specified in a `package.json` file. However, we find that `yarn` results in signficantly more consistent builds when working with React Native, so we recommend using `yarn`.

Watchman

Watchman[17] is a file watching service that watches files and triggers actions when they are modified. If you use macOS as your operating system, the Expo and React Native documentation recommend installing Watchman for better performance. The instructions to install the service can be found here[18].

Expo

Expo[19] is a platform that provides a number of different tools to build fully functional React Native applications without having to write native code. If you've used Create React App[20] before, you'll notice similarities in that no build configuration is required to get up and running.

Building an application also does not require using Xcode for iOS, or Android Studio for Android. This means that developers can build native iOS applications without

[16] https://yarnpkg.com/lang/en/docs/migrating-from-npm/#toc-cli-commands-comparison
[17] https://facebook.github.io/watchman/
[18] https://facebook.github.io/watchman/docs/install.html#installing-on-os-x-via-homebrew
[19] https://expo.io/
[20] https://github.com/facebookincubator/create-react-app

even owning a Mac computer. Using Expo is the easiest way to get started with React Native and is recommended in the React Native documentation[21].

Expo provides two local development tools that allow us to build, preview, share and publish React Native projects:

- **Expo client app**: A client iOS/Android mobile application for previewing projects.
- **Expo CLI**: A local development tool for building React Native projects with no build configuration. This is done through Create React Native App, which is merged directly with the CLI.

Including Native Code

The Expo CLI is not the only way to start a React Native application. If we need to start a project with the ability to include native code, we'll need to use the React Native CLI[a] instead. With this however, our application will require Xcode and Android Studio for iOS and Android respectively.

Expo also provides a number of different APIs for device specific properties such as contacts, camera and video. However, if we need to include a native iOS or Android dependency that is not provided by Expo, we'll need to *eject* from the platform entirely. Ejecting an Expo application means we have full control of managing our native dependencies, but we would need to use the React Native CLI from that point on.

We'll explore how to use React Native CLI and add native modules to a project later on in this book.

[a] https://facebook.github.io/react-native/docs/getting-started.html#installing-dependencies

Previewing with the Expo client

To develop and preview apps with Expo, we need to install its client iOS or Android app[22] to develop and run React Native apps on our device.

[21] https://facebook.github.io/react-native/docs/getting-started.html
[22] https://github.com/expo/expo

Android

On your Android mobile device, install the Expo client on Google Play[23]. You can then select `Scan QR code` and scan this QR code once you've installed the app:

QR Code

If this QR code doesn't work, we recommend making sure you have the latest version of that Expo app installed, and that you're reading the latest edition of this book.

Instead of scanning the QR code, you can also type the project URL, `exp://exp.host/@fullstackio/weather`, inside of Expo to load the application.

iOS

You can install the Expo client via the App Store[24]. With an iOS device however, there is no capability to scan a QR code. This means we'll first need to build the final app in order to preview it. We can do this by navigating to the `weather/` directory in the sample code folder and running the following commands:

```
cd weather
yarn
yarn start
```

This will start the React Native packager. Pressing `e` will allow you to send a link to your device by SMS or e-mail (you'll need to provide your mobile phone number

[23] https://play.google.com/store/apps/details?id=host.exp.exponent
[24] https://itunes.apple.com/us/app/expo-client/id982107779?mt=8

or email address). Once done, clicking the link will open the application in the Expo client.

For the app to load on your physical device, you'll need to make sure that your phone is connected to the same local network as your computer.

Using a supported version

The Expo client app only supports the most recent 5 or 6 releases of Expo and React Native. This book uses Expo SDK 36 (React Native version 0.61). If you're reading this book more than 6 months after the release date, you may receive an error indicating the Expo version of your project is too old if you try to run your app in the Expo client. You can upgrade to the latest version of Expo and React Native by referring to the upgrade guide for your current version in the documentation[25].

Most of the time, updates don't cause significant API changes. However, if you purchased a digital copy of the book, we recommend you download an updated version of the book just in case.

Preparing the app with the CLI

At this point, you should see the final application load successfully on your device. Play around with the app for a few minutes to get a feel for it. Try searching for different cities as well a location that doesn't even exist.

If you plan on building the application as you read through the chapter, you'll need to create a brand new project. Once `yarn` is installed, let's run the following command to install the Expo CLI globally:

```
yarn global add expo-cli@3.11.7
```

The `@3.11.7` specifies the *version* of `expo-cli` to install. It's important to lock in version `3.11.7` so that the version on your machine matches the version used in this book.

We'll call our application `weather` and can use the following command to get started:

[25] https://docs.expo.io/versions/latest/workflow/upgrading-expo

```
expo init weather --template blank@sdk-36 --yarn
```

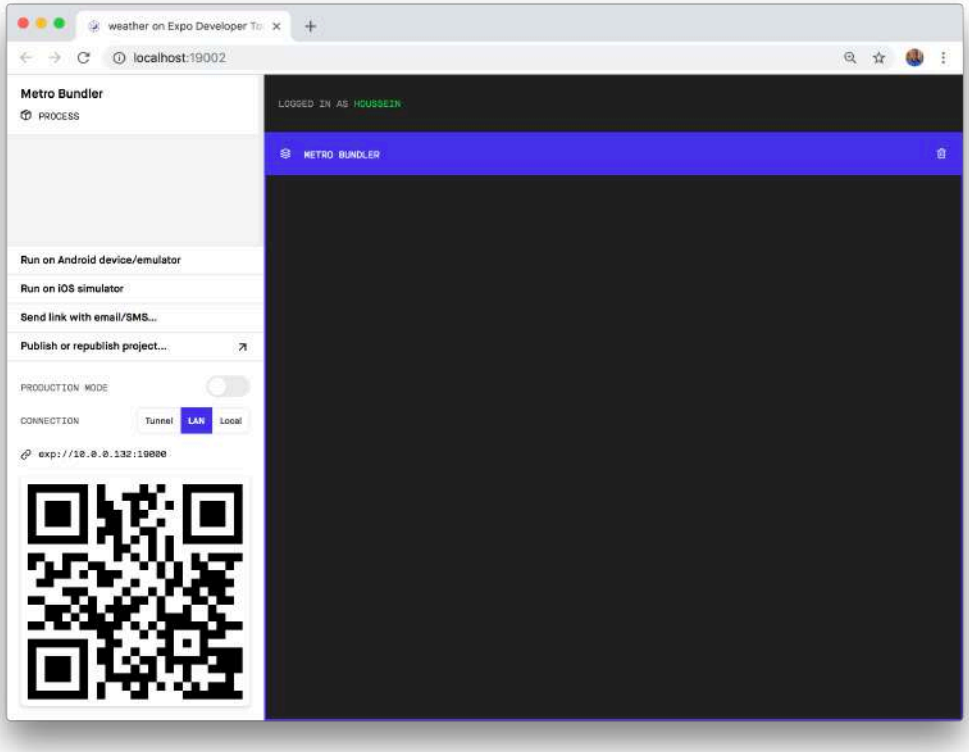
The CLI will take a few minutes to download and extract all the project files. We used the `--template` option to specify that we want the *blank* template built with Expo SDK 36. We used the `--yarn` option to use `yarn` instead of `npm` for installing dependencies. Navigate to the `weather` directory once that command is finished to boot the app:

```
cd weather
yarn start
```

Once the project has finished starting, you should see some information outputted to the console.

With the packager running, scanning the QR code with an Android device or sending a link directly to an iOS device using the `e` hotkey will allow you to preview the application. It is important to remember that a device needs to be connected to the same local network as the computer in order for this to work.

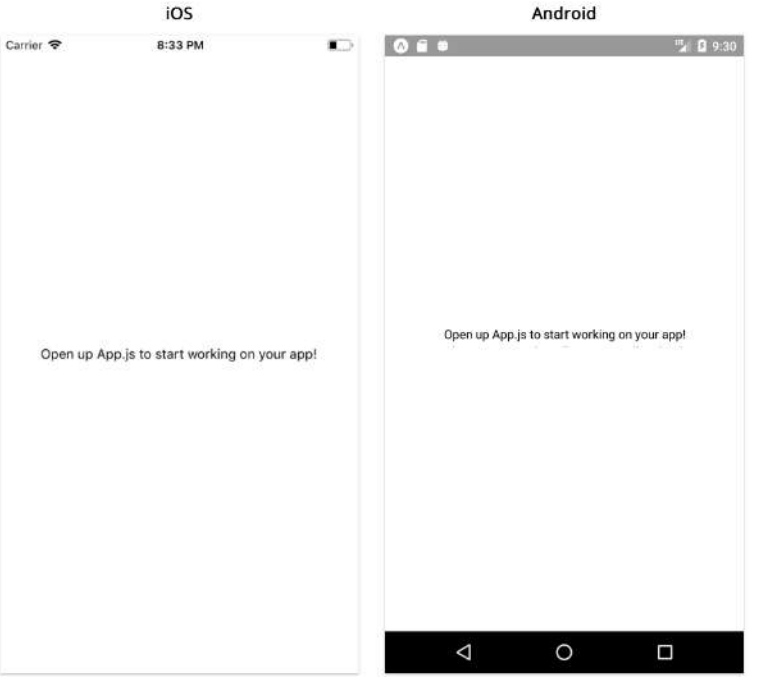
Secondly, a browser tab that renders Expo's Developer Tools should have also opened for you automatically.



Expo Developer Tools

With Expo DevTools, you can see outputted logs easier as well as perform a number of actions, such as sending an email link, directly through its interface instead of typing into the console if you prefer.

Open the application with your Android device by using the QR code or by sending a link to your iOS device. Once it finishes loading, we should see the starting point:



Application

Running on a simulator

As we mentioned, using the Expo client app allows us to run our application without using native tooling (Xcode for iOS, or Android Studio for Android).

However if we happen to have the required build tools we can still run our application in a virtual device or simulator:

- With a Mac, `yarn run ios` will start the development server and run the application in an iOS simulator. We can also start the packager separately with `yarn start` and press `i` to open the simulator.
- With the required Android tools[26], `yarn run android` will start the application in an Android emulator. Similarly, pressing `a` when the React Native packager is running will also boot up the emulator.

Running an application using an emulator/simulator can be useful to test on different devices and screen sizes. It can also be quicker to update and test code changes on a virtual device. However, it's important to run your application on an actual device at some point in order to get a better idea of how exactly it looks and feels.

By default, the Expo CLI comes with *live reload* enabled. This means if you edit and save any file, the application on your mobile device will **automatically reload**. Moreover, any build errors and logs will be displayed directly in the terminal.

Let's see what the directory structure of our app looks like. Open up a new terminal window.

Navigate to this app:

```
cd weather
```

And then run `ls -a` to see all the contents of the directory:

```
ls -a
```

[26] https://facebook.github.io/react-native/docs/getting-started.html

If you're using PowerShell or another non-Unix shell, you can just run `ls`.

Although your output will look slightly different based on your operating system, you should see all the files in your directory listed:

```
├── .expo/
├── assets/
├── node_modules/
├── .gitignore
├── .watchmanconfig
├── App.js
├── app.json
├── babel.config.js
├── package.json
```

Let's go through each of these files:

- `.expo/` contains files that define configurations for the Expo packager and for device connection settings.
- `.assets/` contains a few image assets by default. These are the app icon and splash screen images.
- `node_modules/` contains all third party packages in our application. Any new dependencies and development dependencies go here.
- `.gitignore` is where we specify which files should be ignored by Git. We can see that both the `node_modules/` and `.expo/` directories are already included.
- `.watchmanconfig` defines configurations for Watchman.
- `App.js` is where our application code lives.
- `app.json` is a configuration file that allows us to add information about our Expo app. The list of properties that can be included in this file is listed in the documentation[27]. Examples of properties that can be changed here include the app icon and splash screen.

[27] https://docs.expo.io/versions/latest/workflow/configuration

- `.babel.config.js` allows us to define presets and plugins for configuring Babel[28]. As we mentioned previously, Babel is a transpiler that compiles newer experimental JavaScript into older versions so that it stays compatible with different platforms.
- `package.json` is where we provide information of the application to our package manager as well as specify all our project dependencies.

package.json

Let's take a closer look at the generated `package.json` file:

```
{
  "main": "node_modules/expo/AppEntry.js",
  "scripts": {
    "start": "expo start",
    "android": "expo start --android",
    "ios": "expo start --ios",
    "web": "expo start --web",
    "eject": "expo eject"
  },
  "dependencies": {
    "expo": "~36.0.0",
    "react": "~16.9.0",
    "react": "~16.9.0",
    "react-native": "https://github.com/expo/react-native/archive/sdk-3\
6.0.0.tar.gz"
  },
  "devDependencies": {
    "babel-preset-expo": "~8.0.0"
  },
  "private": true,
}
```

The `scripts` property contains all the commands needed to run the application. `yarn start`, `yarn run android`, and `yarn run ios` allow us to start the application

[28] https://babeljs.io/

development server and/or run on a virtual device or simulator. `yarn run eject` starts the process of ejecting the application from the Expo toolchain. As we mentioned earlier, this can be necessary if we need to include a React Native library that contains native code or if we need to write native code ourselves.

The `dependencies` and `devDependencies` properties define the application and development dependencies respectively. Expo, the React core library and the version of React Native needed for this specific version of Expo are all included as dependencies. `babel-preset-expo`, a Babel preset that contains a number of predefined Babel plugins, is the only development dependency included by default.

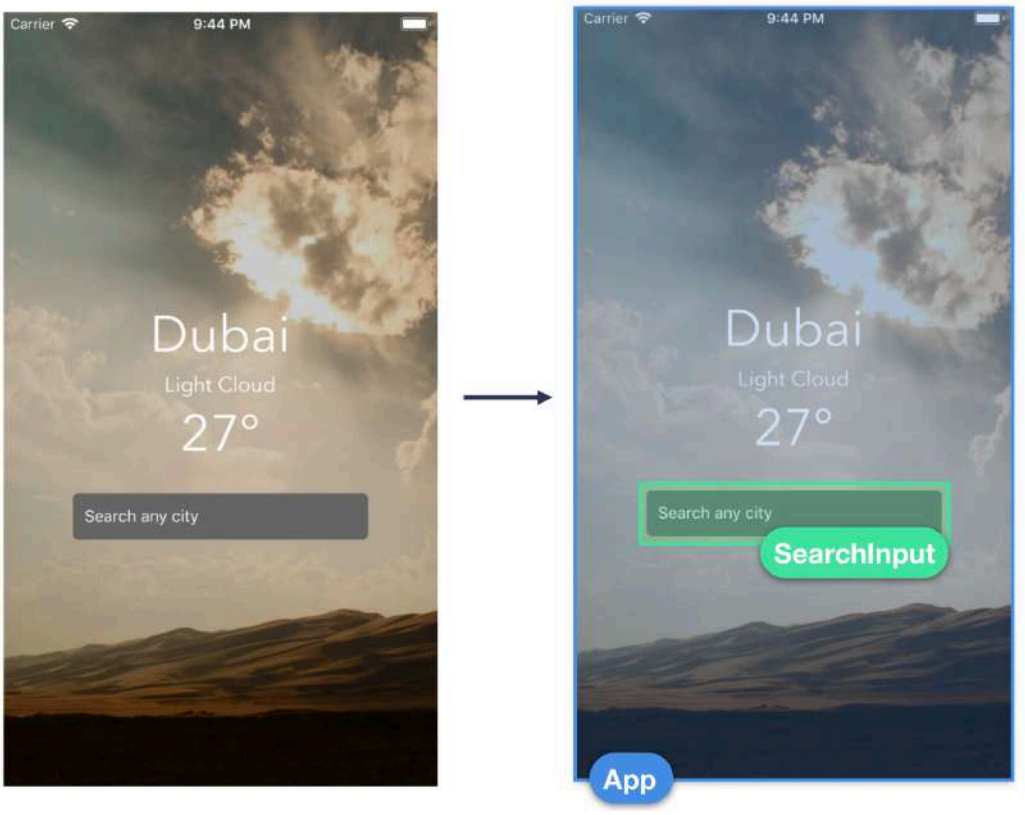
The `utils/` directory

If you look at the book's sample code, you'll note that every application has a `utils/` directory. This directory contains helper functions that the application will use. You don't need to concern yourself with the details of these functions as they're not relevant to the chapter's core concepts.

When we reach the point in the application's development where we need to use a utility provided by `utils/`, we'll remind you to copy over that folder from the sample code. You can also do this immediately after initializing each project.

Components

With newer versions of JavaScript, we can define objects with properties using *classes*. React Native lets us use this syntax to create *components*. Let's take a look at a visual breakdown of the components in our application:



Component Structure

We have an `App` component that represents the entire *screen* and contains the weather information displayed to the use. Inside of this component, we have a `SearchInput` component that allows us to search for different cities.

App

`App` is the only component created with a default application using a blank template. Let's take a look at its file:

weather/1/App.js

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

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Text>
      </View>
    );
  }
}

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

Notice how we have a class defined in our file named `App` that extends `React.Component`. Using `extends` allows us to declare a class as a subclass of another class. In here, we've defined `App` as a subclass of `React.Component`. This is how we specify a specific class to be a *component* in our application.

If you'd like to learn more about how classes work in JavaScript, refer to our Appendix.

We can also attach methods as properties to classes, and the same applies to component classes in React Native. We can see we already have one for this component, the `render` method:

weather/1/App.js

```
5   render() {
6     return (
7       <View style={styles.container}>
8         <Text>Open up App.js to start working on your app!</Text>
9       </View>
10    );
11  }
```

What we see on our device when launching our device matches what we see described in this method. **The `render()` method is the only required method for a React Native component**. React Native uses the return value from this method to determine what to render for the component.

When we use React Native, we represent different parts of our application as **components**. This means we can build our app using different reusable pieces of logic with each piece displaying a specific part of our UI. Let's break down what we already have in terms of components:

- Our entire application is rendered with `App` as our top-level component. Although created automatically as part of setting up a new Expo CLI project, this component is a custom component responsible for rendering what we need in our application.
- The `View` component is used as a layout container.
- Within `View`, we use the `Text` component to display lines of text in our application. Unlike `App`, both `View` and `Text` are **built-in** React Native components that are imported and used in our custom component.

We can see that our `App` component uses and returns an HTML-like structure. This is **JSX**, which is an extension of JavaScript that allows us to use an XML-like syntax to define our UI structure.

JSX

When we build an application with React Native, components ultimately render native views which are displayed on our device. As such, the `render()` method of a

component needs to describe how the view should be represented. In other words, React Native allows us to describe a component's iOS and Android representation in JavaScript.

JSX was created to make the JavaScript representation of components easier to understand. It allows us to structure components and show their hierarchy visually in markup. Consider this JSX snippet:

```
<View>
  <Text style={{ color: 'red' }}>
    Hello, friend! I am a basic React Native component.
  </Text>
</View>
```

In here, we've nested a `Text` component within a `View` component. Notice how we use braces ({}) around an object (`{ color: 'red' }`) to set the style property value for `Text`. In JSX, braces are a delimiter, signaling to JSX that what resides in-between the braces is a JavaScript expression. The other delimiter is using quotes for strings, like this:

```
<TextInput placeholder="This is a string" />
```

Even though the JSX above might look similar to HTML, it is actually just compiled into JavaScript function calls (ex: `React.createElement(View)`). For this reason, we need to import `React` at the top of any file that contains JSX. You can refer to the Appendix for more detail.

During runtime React Native takes care of rendering the actual native UI for each component.

Props

We use the imported `Text` component to wrap each line of text output for our `App` component:

```
<Text>Open up App.js to start working on your app!</Text>
```

And we use the imported `View` component to wrap all the `Text` components:

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

Props allow us to pass parameters to components to customize their features. Here, `View` is used to layout the entire content of the screen. We only have a single prop attached, `style`, that allows us to pass in style parameters to adjust how our `View` component is rendered on our devices. Each built-in component provided by React Native has its own set of valid props that we can use for customization.

> If you're familiar with HTML, it's very similar. For example, in HTML, say you wanted to insert an image named `image.png`. You'd specify an `img` tag with a `src` attribute like this:
>
> ```
> <img src="path/to/image.png">
> ```
>
> To give you an idea of the similarity, in React Native we can include images using the `Image` component. We specify the location using the `source` prop:
>
> ```
> <Image source={require('./image.png')}>
> ```
>
> We'll cover images in greater detail later.

Like our `View` component, many components in React Native accept a `style` prop. Styling is a large topic that we explore throughout this book. However, we can take a look at our `styles` object at the bottom of `App.js` and get an idea of how it works:

weather/1/App.js

```
14 const styles = StyleSheet.create({
15   container: {
16     flex: 1,
17     backgroundColor: '#fff',
18     alignItems: 'center',
19     justifyContent: 'center',
20   },
21 });
```

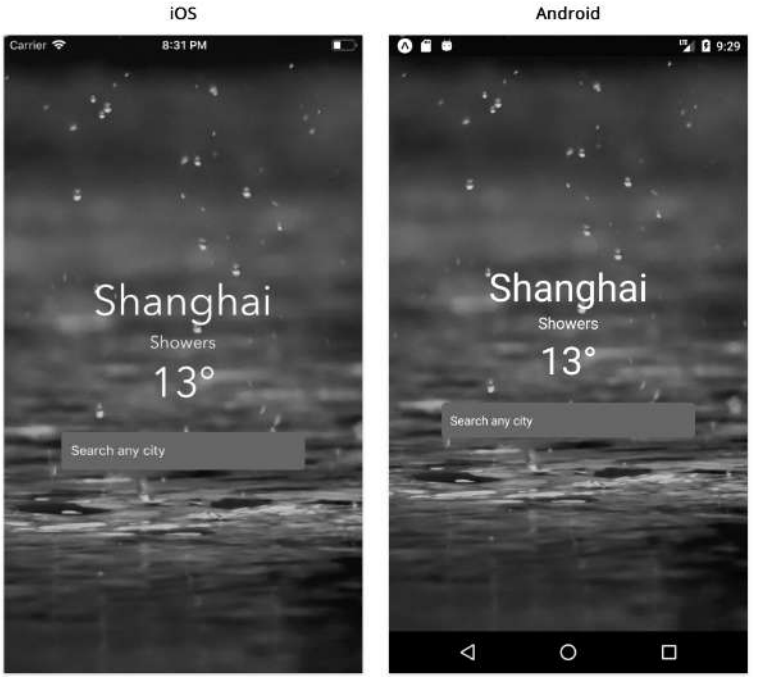
Web developers may recognize that this looks like CSS (Cascading Style Sheets) which is used to style web pages. It's important to note that styling in React Native *does not* use CSS. However, React Native borrows a lot of styling nomenclature from web development. Here, we specify that the text should be centered and that the background color should be white (`#fff`).

If you've used CSS before, you'll find styling in React Native very familiar. If not, don't worry! It's easy to get the hang of it.

Specifically, `styles.container` has the attributes `flex`, `alignItems` and `justifyContent`. These are used to position the `View` in the center of the screen. React Native uses **flexbox** to layout and align items consistently on different device sizes. We'll go into more detail about how exactly flexbox works in later chapters.

To build our weather app, we'll start with layout and styling. Once we have some of the essence of our weather app in place we can begin to explore strategies for managing data.

As we saw in the completed version of the app, we want our app to display the **city**, **temperature**, and **weather conditions** as separate text fields. Although we'll eventually interface with a weather API in order to retrieve actual data, we'll begin with hard-coding these values.



The completed app

Adding styles

To get a better handle on styling, let's try adding an object with a color attribute to one of the text fields:

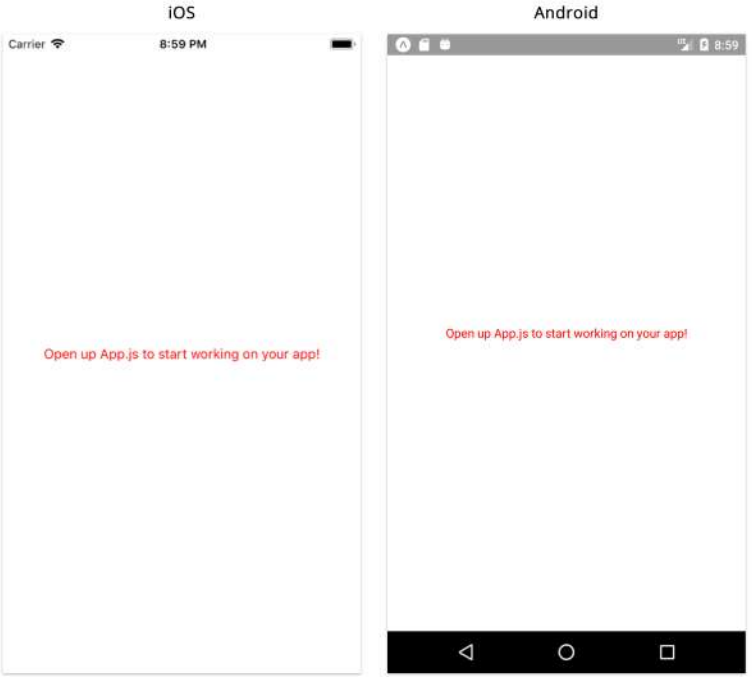
```
<View style={styles.container}>
  <Text style={{ color: 'red' }}>
    Open up App.js to start working on your app!
  </Text>
</View>
```

Note that the outer-most set of brackets above are *delimiters* enclosing our JavaScript statement. Inside of the delimiters is a JavaScript object. In React Native, if the object is small enough it's common to just write it all on one line.

However, the double brackets (`{{}}`) might be confusing. Here's another way of writing the same component:

```
const style = { color: 'red' };

return (
  <View style={styles.container}>
    <Text style={style}>
      Open up App.js to start working on your app!
    </Text>
```

Save `App.js`. We can see our style applied once the application reloads:



As we mentioned previously, **live reload** is enabled by default in Expo. This means that with any change to the code, the application will reload immediately. If you happen to not see any changes reflected as soon as you save the file, you may have to check to see if this is enabled. The documentation[29] explains how to open up the developer menu and enable/disable the feature.

Although we can style our entire component this way, a lot of *inline* styles (or style attributes defined directly *within* the delimeter of the `style` prop) used in a component can make things harder to read and digest.

We can solve this by leveraging React Native's `Stylesheet` API to separate our styles from our component. With `Stylesheet`, we can create styles with attributes similar to CSS stylesheets. We can see that `Stylesheet` is already imported at the top of the file. It's used to declare our first style, `styles.container`, which we use for `View`. We can add a new style called `red` to our `styles`:

[29] https://docs.expo.io/versions/latest/guides/up-and-running.html#cant-see-your-changes

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

We'll then have `Text` use this style:

```
<View style={styles.container}>
  <Text style={styles.red}>
    Open up App.js to start working on your app!
  </Text>
</View>
```

If we save our file and take a look at our app, we can see that the end result is the same.

Now let's add some appropriate styles and text fields in order to display some weather data for a location. To add multiple styles to a single component, we can pass in **an array of styles**:

weather/2/App.js

```
17          <Text style={[styles.largeText, styles.textStyle]}>
18            San Francisco
19          </Text>
20          <Text style={[styles.smallText, styles.textStyle]}>
21            Light Cloud
22          </Text>
23          <Text style={[styles.largeText, styles.textStyle]}>24°</Text>
```

It is important to mention that when passing an array, the styles at the end of the array take precedence over earlier styles, in case of any repeated attributes. We can see that we're referencing three new styles; `textStyle`, `smallText`, and `largeText`. Let's define these within our `styles` object:

weather/2/App.js

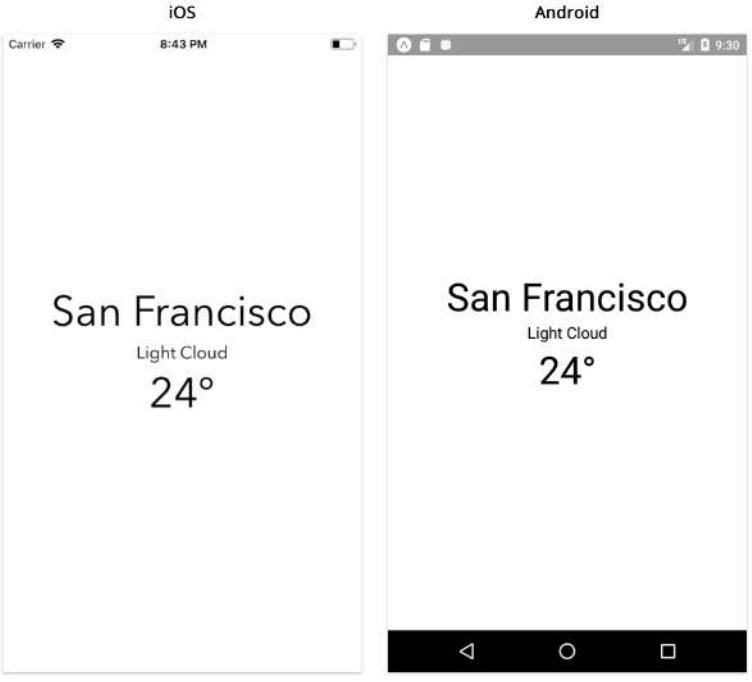
```
36  const styles = StyleSheet.create({
37    container: {
38      flex: 1,
39      backgroundColor: '#fff',
40      alignItems: 'center',
41      justifyContent: 'center',
42    },
43    textStyle: {
44      textAlign: 'center',
45      fontFamily:
46        Platform.OS === 'ios' ? 'AvenirNext-Regular' : 'Roboto',
47    },
48    largeText: {
49      fontSize: 44,
50    },
51    smallText: {
52      fontSize: 18,
53    },
```

- `textStyle` specifies an alignment (`center`) as well as the `fontFamily`. Notice how we use `Platform` to define platform specific fonts for both iOS and Android. We do this because both operating systems provide a different set of native fonts.
- `smallText` and `largeText` both specify different font sizes.

`Platform` is a built-in React Native API. We'll need to make sure to import it:

```
import { StyleSheet, Text, View, Platform } from 'react-native';
```

Let's take a look at our application now:



Styled Text

Platform specific properties

The `Platform` API allows us to conditionally apply different styles or properties in our component based on the device's operating system. The `OS` attribute of the object returns either `ios` or `android` depending on the user's device.

Although this is a relatively simple way to apply different properties in our application based on the user's device, there may be scenarios where we may want our component to be substantially different between operating systems.

We can also use the `Platform.select` method that takes the operating system as keys within an object and returns the correct result based on the device:

```
textStyle: {
  textAlign: 'center',
  ...Platform.select({
    ios: {
      fontFamily: 'AvenirNext-Regular',
    },
    android: {
      fontFamily: 'Roboto',
    },
  }),
},
```

Separate files

Instead of applying conditional checks using `Platform.OS` a number times throughout the entire component file, we can also leverage the use of **platform specific files** instead. We can create two separate files to represent the same component each with a different extension: `.ios.js` and `.android.js`. If both files export the same component class name, the React Native packager knows to choose the right file based on the path extension. We'll dive deeper into platform specific differences later in this book.

Text input

We now have text fields that display the location, weather condition, and temperature. The next thing we need to do is provide some sort of input to allow the user to search for a specific city. Again, we'll continue using hardcoded data for now. We'll only begin using an API for real data once we have all of our components in place.

React Native provides a built-in `TextInput` component that we can import into our component that allows us to accept user input. Let's include it within our `View` container underneath the `Text` components (make sure to import it as well!):

weather/2/App.js

```
        <Text style={[styles.largeText, styles.textStyle]}>
          San Francisco
        </Text>
        <Text style={[styles.smallText, styles.textStyle]}>
          Light Cloud
        </Text>
        <Text style={[styles.largeText, styles.textStyle]}>24°</Text>

        <TextInput
          autoCorrect={false}
          placeholder="Search any city"
          placeholderTextColor="white"
          style={styles.textInput}
          clearButtonMode="always"
        />
```

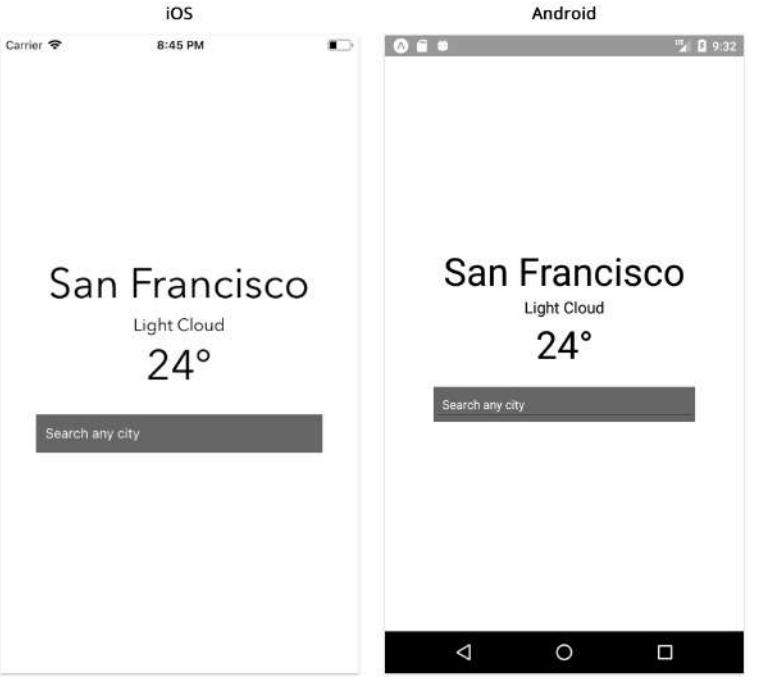
`TextInput` here is being referenced using a *self-closing* tag (`<TextInput />`). JSX allows us to use this shorthand version instead of an opening and closing tag (`<TextInput></TextInput>`) when a component has no children elements nested within.

There are a number of props associated with `TextInput` that we can use. We'll cover the basics here but go into more detail about them in the "Core Components" chapter. Here we're specifying a placeholder, its color, as well as a style for the component itself. Let's create its style object, `textInput`, underneath our other styles:

weather/2/App.js

```
  smallText: {
    fontSize: 18,
  },
  textInput: {
    backgroundColor: '#666',
    color: 'white',
    height: 40,
    width: 300,
    marginTop: 20,
    marginHorizontal: 20,
    paddingHorizontal: 10,
    alignSelf: 'center',
  },
```

As we mentioned previously, all the attributes that we provide styles with in React Native are extremely similar to how we would apply them using CSS. Now let's take a look at our application:



Text Input

We can see that the text input has a default underline on Android. We'll go over how to remove this in a bit.

We've also specified the `clearButtonMode` prop to be `always`. This shows a button on the right side of the input field when characters are inserted that allows us to clear the text. This is only available on iOS.

Text Input Clear Button

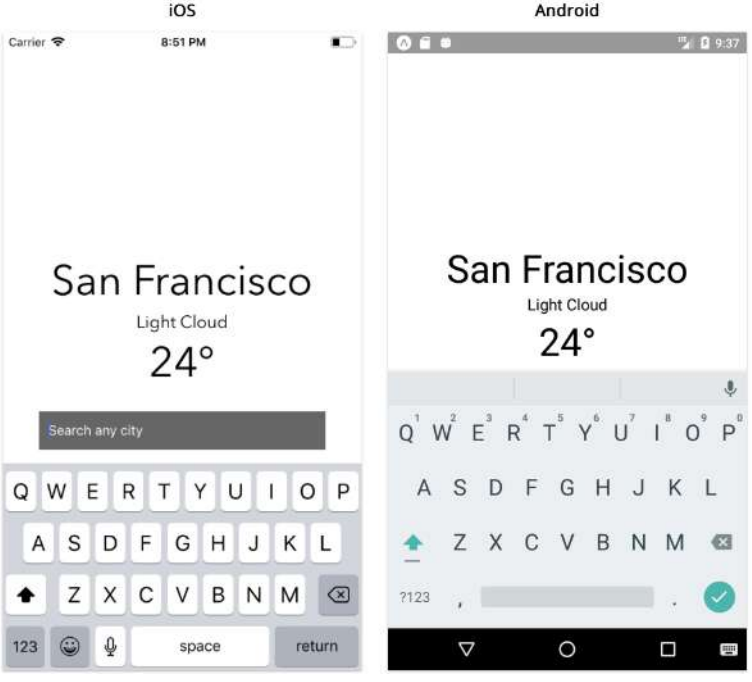
We can now type into the input field!

If you're using the iOS simulator, you can connect your hardware keyboard and use that with any input field. This can be done with `Shift + ⌘ + K` or going to `Hardware -> Keyboard -> Connect Hardware Keyboard`

With this enabled, the software keyboard may not show by default. You can toggle this by pressing `⌘ + K` or going to `Hardware -> Keyboard -> Toggle Software Keyboard`

Now every time you click an input field, the software keyboard will display exactly how it would if you were using a real device and you can type using your hardware keyboard.

However one thing you may have noticed is that when you focus on the input field with a tap, the keyboard pops up and covers it on Android and comes quite close on iOS:



Keyboard

Since the virtual keyboard can cover roughly half the device screen, this is a common problem that occurs when using text inputs in an application. Fortunately, React

Native includes `KeyboardAvoidingView`, a component that solves this problem by allowing us to adjust where other components render in relation to the virtual keyboard. Let's import and use this component instead of `View`:

weather/2/App.js

```
  render() {
    return (
      <KeyboardAvoidingView
        style={styles.container}
        behavior="height"
      >
        <Text style={[styles.largeText, styles.textStyle]}>
          San Francisco
        </Text>
        <Text style={[styles.smallText, styles.textStyle]}>
          Light Cloud
        </Text>
        <Text style={[styles.largeText, styles.textStyle]}>24°</Text>

        <TextInput
          autoCorrect={false}
          placeholder="Search any city"
          placeholderTextColor="white"
          style={styles.textInput}
          clearButtonMode="always"
        />
      </KeyboardAvoidingView>
    );
  }
```

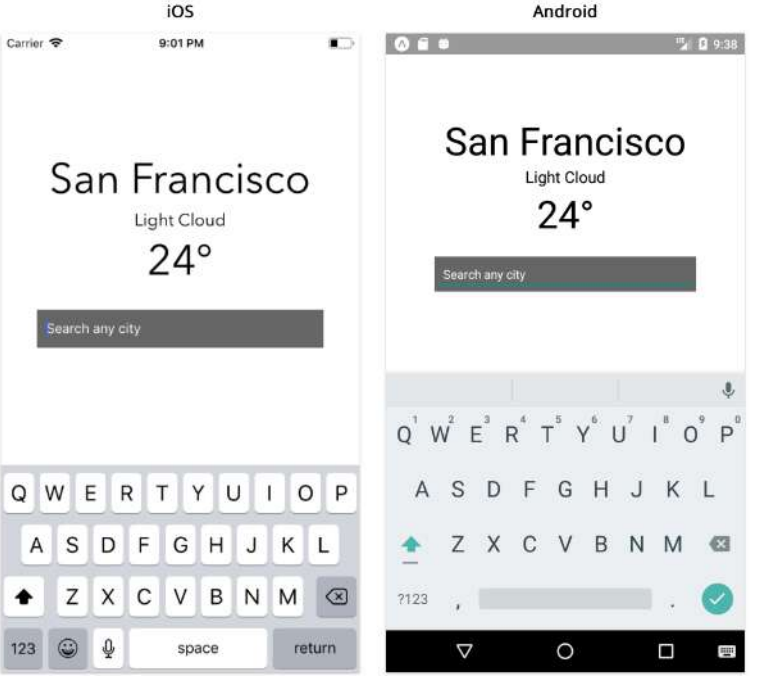
Notice that `KeyboardAvoidingView` accepts a `behavior` prop with which we can customize how the keyboard adjusts. It can change its height, position or bottom padding in relation to the position of the virtual keyboard. Here, we've specified `height`.

And finally, it's important to double check our imports to make sure we have everything that we're using:

weather/2/App.js

```
import React from 'react';
import {
  StyleSheet,
  Text,
  KeyboardAvoidingView,
  Platform,
  TextInput,
} from 'react-native';
```

Now tapping the text input will shift our component text and input fields out of the way of the software keyboard.



Keyboard Avoiding View

Custom components

So far, we've explored how to add styling into our application, and we've included some built-in components into our main `App` component. We use `View` as our component container and import `Text` and `TextInput` components in order to display hardcoded weather data as well as an input field for the user to change locations.

It's important to re-iterate that React Native is **component-driven**. We're already representing our application in terms of components that describe different parts of our UI without too much effort, and this is because React Native provides a number of different built-in components that you can use immediately to shape and structure your application.

However, as our application begins to grow, it's important to begin thinking of how it can further be broken down into smaller and simpler chunks. We can do this by creating **custom components** that contain a small subset of our UI that we feel fits better into a separate, distinct component file. This is useful in order to allow us to further split parts of our application into something more manageable, reusable and testable.

Although our application in its current state isn't extremely large or unmanageable, there's still some room for improvement. The first way we can refactor our component is to move our `TextInput` into a separate component to hide its implementation details from the main `App` component. Let's create a `components` directory in the root of the application with the following file:

```
├── components/
      - SearchInput.js
```

All the custom components we create that we use in our main `App` component will live inside this directory. For more advanced apps, we might create directories within `components` to categorize them more specifically. Since this app is pretty simple, let's use a flat `components` directory.

The `SearchInput` will be our first custom component so let's move all of our code for `TextInput` from `App.js` to `SearchInput.js`:

weather/3/components/SearchInput.js

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

export default class SearchInput extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <TextInput
          autoCorrect={false}
          placeholder={this.props.placeholder}
          placeholderTextColor="white"
          underlineColorAndroid="transparent"
          style={styles.textInput}
          clearButtonMode="always"
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    height: 40,
    width: 300,
    marginTop: 20,
    backgroundColor: '#666',
    marginHorizontal: 40,
    paddingHorizontal: 10,
    borderRadius: 5,
  },
  textInput: {
    flex: 1,
    color: 'white',
  },
});
```

Let's break down what this file contains:

- We export a component named `SearchInput`.
- This component accepts a `placeholder` prop.
- This component returns a React Native `TextInput` with a few of its properties specified wrapped within a `View`.
- We've applied the appropriate styles to our view container including a `borderRadius`.
- We also added `underlineColorAndroid="transparent"` to remove the dark underline that shows by default on Android.

`this` is a special keyword in JavaScript. The details about this are a bit nuanced, but for the purposes of the majority of this book, **`this` will be bound to the React Native component class**. So, when we write `this.props` inside the component, we're accessing the props property on the component. When we diverge from this rule in later sections, we'll point it out.

For more details on this, check out this page on MDN[30].

Custom props

As you may recall, in `App.js` we set the `placeholder` prop for `TextInput` to "Search any city." That renders the text input with a placeholder:

For `SearchInput`, we could hardcode a string again for `placeholder`. But what if we wanted to add a search input elsewhere in our application? It would be nice if `placeholder` was customizable.

[30] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

Earlier in this chapter, we explored how we can use `props` with a number of built-in components in order to customize their features. We can also *create* `props` for custom components that we build as well.

That's what we do here in `SearchInput`. The component accepts the prop `placeholder`. In turn, `SearchInput` uses this value to set the `placeholder` prop on `TextInput`.

The way data flows from parent to child in React Native is through props. When a parent renders a child, it can send along props the child depends on. A component can access all its props through the object `this.props`. If we decide to pass down the string `"Type Here"` as the `placeholder` prop, the `this.props` object will look like this:

```
{ "placeholder": "Type Here" }
```

In here, we'll set up `App` to render `SearchInput` which means that `App` is the *parent* of `SearchInput`. Our parent component will be responsible for passing down the actual value of `placeholder`.

We're getting somewhere interesting now. We've set up a custom `SearchInput` component and by building it to accept a `placeholder` prop, we're already setting it up to be configurable. Based on what it receives, it can render any placeholder message that we'd like.

Importing components

In order to use `SearchInput` in `App`, we need to import the component first. We can remove the `TextInput` logic from `App.js` and have `App` use `SearchInput` instead:

weather/3/App.js

```
import React from 'react';
import {
  StyleSheet,
  Text,
  KeyboardAvoidingView,
  Platform,
} from 'react-native';

import SearchInput from './components/SearchInput';

export default class App extends React.Component {
  render() {
    return (
      <KeyboardAvoidingView
        style={styles.container}
        behavior="height"
      >
        <Text style={[styles.largeText, styles.textStyle]}>
          San Francisco
        </Text>
        <Text style={[styles.smallText, styles.textStyle]}>
          Light Cloud
        </Text>
        <Text style={[styles.largeText, styles.textStyle]}>24°</Text>

        <SearchInput placeholder="Search any city" />
      </KeyboardAvoidingView>
    );
  }
}
```

By moving the entire `TextInput` details into a separate component called `SearchInput`, we've made sure to not have any of its specific implementation details showing in

the parent component anymore. We can also remove the text input's styling defined within the `styles` object.

There's no specific answer to how often we should isolate different UI logic into separate custom components. React Native was built in order to allow us to lay out our entire application in terms of self-contained components, and that means we should separate parts of our application into distinct units with custom functionality attached to them. This allows us to build a more manageable application that's easier to control and understand. We've isolated knowledge of our search input to the component `SearchInput` and we'll continue to isolate specific pieces of our app throughout this chapter.

It's common to separate your imports into two groups: imports from dependencies, and imports from other files in your project. That's why we put a blank line above `SearchInput`. This comes down to personal style preference.

Background image

As we saw in the photo of the completed version of the app at the beginning of this chapter, we can make our application more visually appealing by displaying a background image that represents the current weather condition.

In this book's sample code, we've included a number of images for various weather conditions. If you inspect the `weather/assets` directory, you'll find images like `clear.png`, `hail.png`, and `showers.png`.

If you're following along, copy these two folders over from the sample code into your project:

1. `weather/assets`
2. `weather/utils`

We mentioned earlier that we've included a `utils/` folder for each project in the book's sample code. This folder contains helper functions that we'll use below.

If you're on macOS or Linux, you can use `cp -r` to copy directories:

```
cp -r weather/{assets,utils} ~/react-native-projects/weather/
```

With the `assets` and `utils` folders copied over, let's update our `App` component:

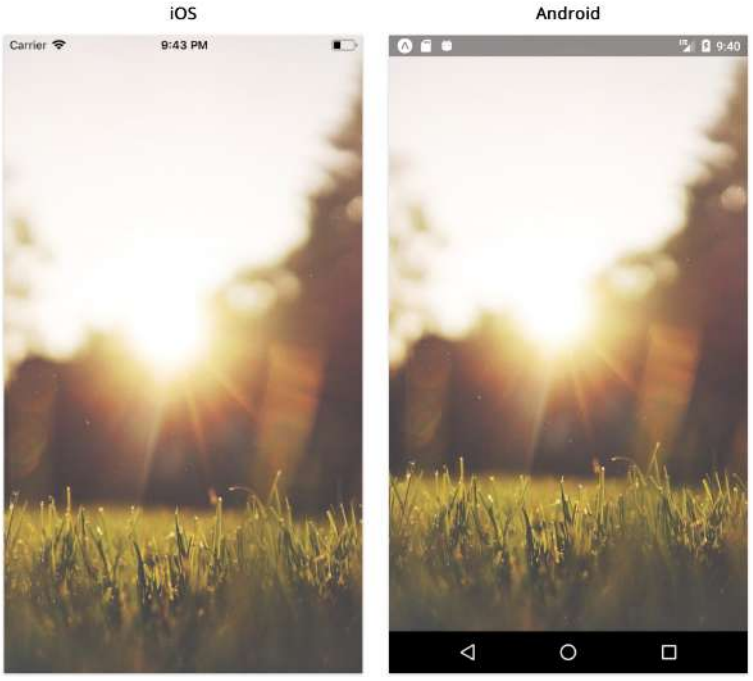
weather/4/App.js

```
import React from 'react';
import {
  StyleSheet,
  View,
  ImageBackground,
  Text,
  KeyboardAvoidingView,
  Platform,
} from 'react-native';

import getImageForWeather from './utils/getImageForWeather';

import SearchInput from './components/SearchInput';

export default class App extends React.Component {
  render() {
    return (
      <KeyboardAvoidingView
        style={styles.container}
        behavior="height"
      >
        <ImageBackground
          source={getImageForWeather('Clear')}
          style={styles.imageContainer}
          imageStyle={styles.image}
        >
          <View style={styles.detailsContainer}>
```

```
            <Text style={[styles.largeText, styles.textStyle]}>
              San Francisco
            </Text>
            <Text style={[styles.smallText, styles.textStyle]}>
              Light Cloud
            </Text>
            <Text style={[styles.largeText, styles.textStyle]}>
              24°
            </Text>

            <SearchInput placeholder="Search any city" />
          </View>
        </ImageBackground>
      </KeyboardAvoidingView>
    );
  }
}
```

In this component, we're importing a `getImageForWeather` method from our `utils` directory which returns a specific image from the `assets` directory depending on a weather type.

For example, `getImageForWeather('Clear')` returns the following image:



Feel free to peek into the implementation details of any function we use from the `utils` directory to get a better idea of how it works.

We also import React Native's built-in `ImageBackground` component. Let's take a closer look at how we're making use of it in our render method:

weather/4/App.js

```
render() {
  return (
    <KeyboardAvoidingView
      style={styles.container}
      behavior="height"
    >
      <ImageBackground
        source={getImageForWeather('Clear')}
        style={styles.imageContainer}
```

```
          imageStyle={styles.image}
        >
```

Conceptually, the `ImageBackground` component is a `View` with an `Image` nested within.

The `source` prop accepts an image location, which we've set to:

`getImageForWeather('Clear')`.

We know this will always return the image displayed above. `ImageBackground` also uses the prop `style` for styling the `View` container and the prop `imageStyle` for styling the image itself. Let's add two new styles and modify the `container` style:

weather/4/App.js

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#34495E',
  },
  imageContainer: {
    flex: 1,
  },
  image: {
    flex: 1,
    width: null,
    height: null,
    resizeMode: 'cover',
  },
```

Defining component styles with a `flex` attribute mean that they will expand to take up any room remaining in their parent container in relation to any sibling components. They share this space in proportion to their defined `flex` values. Since `ImageBackground` is the only nested element within `KeyboardAvoidingView`, setting `imageContainer` to `flex: 1` means that this element will fill up the entire space of its parent component. We've removed `justifyContent` and `alignItems` from `container` so that the `ImageBackground` can take up the entire device screen.

We also used `flex: 1` to style the actual image itself, `image`, to make sure it takes up the entire space of its parent container. With images in particular, the component will fetch and use the actual width and height of the source image by default. For this reason, we've also set its `height` and `width` attribues to `null` so that the dimensions of the image fit the container instead. The `resizeMode` attribute allows us to define how the image is resized when the `Image` element does not match its actual dimensions. Setting this attribute to *cover* means that the image will scale uniformly until it is equal to the size of the component.

The "Core Components" chapter will dive deeper into how flexbox, layout, and the `Image` component work in React Native

We also wrapped all of our `Text` elements and `SearchInput` within a view container styled with `detailsContainer`:

weather/4/App.js

```
<View style={styles.detailsContainer}>
  <Text style={[styles.largeText, styles.textStyle]}>
    San Francisco
  </Text>
  <Text style={[styles.smallText, styles.textStyle]}>
    Light Cloud
  </Text>
  <Text style={[styles.largeText, styles.textStyle]}>
    24°
  </Text>

  <SearchInput placeholder="Search any city" />
</View>
```

Now let's set up its style:

weather/4/App.js

```
  detailsContainer: {
    flex: 1,
    justifyContent: 'center',
    backgroundColor: 'rgba(0,0,0,0.2)',
    paddingHorizontal: 20,
  },
```

Here, we're ensuring the container within `ImageBackground` also fills up the entire space of its parent component as well as have its items aligned at the center of the screen. We also add a semi-transparent overlay to our image by setting the `backgroundColor` of this component.

The last thing we'll need to do here is change our `Text` elements to white instead of black to show more clearly with a background image:

weather/4/App.js

```
  textStyle: {
    textAlign: 'center',
    fontFamily:
      Platform.OS === 'ios' ? 'AvenirNext-Regular' : 'Roboto',
    color: 'white',
  },
```

Try it out

Save the file and take a look at our app. We should now see the background image displayed!

Modifying location

The steps we've taken so far are quite common when starting React Native applications. We hardcode all our data, organize our app into components, and get an idea of the visual layout as well as how it breaks down into components.

However, our app really isn't very useful at this moment. If we take a look at our `SearchInput` component for instance, we can type anything into the input field but nothing actually happens as a result. We need to find a way to track changes made to the component and store that information somewhere. In other words, we need some piece of **mutable data** that updates whenever the user changes or submits the input field.

Instead of having `SearchInput` not actually manage any data that represents the text inputted by the user, let's pass in a `prop` for it called `location` to reflect what the user has inputted into the text input field:

```
render() {
  const location = 'San Francisco';

  return (
    <KeyboardAvoidingView
      style={styles.container}
      behavior="height"
    >
      <ImageBackground
        source={getImageForWeather('Clear')}
        style={styles.imageContainer}
        imageStyle={styles.image}
      >
        <View style={styles.detailsContainer}>
          <Text style={[styles.largeText, styles.textStyle]}>
            {location}
          </Text>
          <Text style={[styles.smallText, styles.textStyle]}>
            Light Cloud
          </Text>
          <Text style={[styles.largeText, styles.textStyle]}>
            24°
          </Text>

          <SearchInput placeholder="Search any city" />
        </View>
```

```
      </ImageBackground>
    </KeyboardAvoidingView>
```

The reason we want to pass in the property that contains our `location` data is we need a way for our child component to modify that field and communicate back up to our container `App` component. Notice how we've moved the static string for `location` into a separate constant which we pass down to `SearchInput`. We've instantiated it as `San Francisco` so that it can show as the first location when the user loads the application. The next thing we just need to do is make sure that this `location` constant is updated when the user actually changes the field in `SearchInput`:

```
export default class SearchInput extends React.Component {
  handleChangeText(newLocation) {
    // We need to do something with newLocation
  }

  render() {
    return (
      <TextInput
        autoCorrect={false}
        placeholder={this.props.placeholder}
        placeholderTextColor="white"
        underlineColorAndroid="transparent"
        style={styles.textInput}
        clearButtonMode="always"
        onChangeText={this.handleChangeText}
      />
    );
  }
}
```

So what did we just do? We've just added `onChangeText` as a new prop to our `TextInput` component. Notice that we don't pass in a specific object or property, but a *function* instead:

```
  onChangeText={this.handleChangeText}
```

This method is invoked everytime the text within the input field is changed. A number of built-in components provided by React Native include *event-driven* props which we can attach specific methods to. We'll explore more throughout this book.

With `onChangeText`, our `TextInput` returns the changed text as an argument which we're attempting to pass into a separate method called `handleChangeText`. Currently our method is blank and we'll explore how we can complete it in a bit.

In React Native, we need to pass in functions when we want to handle certain events related to the component being referenced. For the `TextInput` component, `onChangeText` is set to fire every single time the text within the input field has changed. We need to "listen" to this specific event in our child component (`TextInput`) so that it can notify our parent component (`SearchInput`) to respond to this event. To do this, we pass in a function that calls another function, or in other words, a callback.

This is a common pattern when building components which need to notify a parent component of some event. Unfortunately with the way we've just set it up, it wouldn't work in this example. This is because the function `handleChangeText` has a different local scope than the component instance. We can work around this by binding our function to the correct context of its **`this`** object.

```
<TextInput
  placeholder={placeholder}
  placeholderTextColor="white"
  underlineColorAndroid="transparent"
  style={styles.textInput}
  clearButtonMode="always"
  onChangeText={this.handleChangeText.bind(this)}
/>
```

Now this might seem okay for the current context, but it can quickly become unwieldy if we build our components with `bind` statements in each event handler. One reason why is if we wanted to use `handleChangeText` in multiple different sub-components for example, we would have to make sure to `bind` it to the correct context

every single time. To help solve this, we can take care of handling our event using **property initializers**:

```
export default class SearchInput extends React.Component {
  handleChangeText = (newLocation) => {
    // We need to do something with newLocation
  }

  render() {
    return (
      <TextInput
        autoCorrect={false}
        placeholder={this.props.placeholder}
        placeholderTextColor="white"
        underlineColorAndroid="transparent"
        style={styles.textInput}
        clearButtonMode="always"
        onChangeText={this.handleChangeText}
      />
    );
  }
}
```

This allows us to declare the member methods as arrow functions:

```
  handleChangeText = (newLocation) => {
    // We need to do something with newLocation
  }
```

And we pass the method name to the prop and nothing more:

```
  onChangeText={this.handleChangeText}
```

Property Initializers

Supported by Babel[31], property initializers are still in the proposal phase and have not yet been slated for adoption in future JavaScript versions. Although this pattern is used quite often in many React and React Native applications, it is important to keep in mind that it is still *experimental* syntax.

For more information on the different ways to handle events in React Native, refer to the Appendix.

Now that we've set up our callback correctly, let's modify `handleChangeText` to change our `text` prop in order to change the data to match what the user is typing:

```
handleChangeText = (newLocation) => {
  this.props.location = newLocation;
};
```

Let's run our application and try typing into the `TextInput` field. You'll immediately notice that the first location that shows is `San Francisco`, so we know that the `text` prop is being passed down successfully!

However, if we type anything into our `TextInput`, you'll notice *nothing happens*. Changing the text within the input field does not actually update the parent `location` property and from the way we've designed our component logic, it looks like it should. This is because `this.props`, which is referenced in `SearchInput`, **is actually owned by `App` and not the child component, `SearchInput`**. A component's `props` are **immutable** and create a one-way data pipeline from parent to children.

We have a bit of a problem. We need to find a way to:

- store local data in our child component, `SearchInput`, that represents the value in the input field
- track changes to the search input field as it's updated by the user
- notify our parent component, `App`, whenever our location changes

This is where we can use a component's **`state`**.

[31] https://babeljs.io/docs/plugins/transform-class-properties/

Storing local data

Let's modify our `SearchInput` component once more. Currently the text input within the component does nothing, so let's add some local component state to control actual data. We can do this by adding a `constructor` method to the component. We can then initialize the component's state within this method:

weather/5/components/SearchInput.js

```
export default class SearchInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      text: '',
    };
  }
```

We can use the `constructor` method to to initialize our **component-specific data**, or `state`. We do this here because this method fires before our component is mounted and rendered. Here, we defined our `state` object to only contain a `text` property:

Remember, components in React Native are extended from `React.Component` to create derived classes. `super()` is required in derived classes in order to reference `this` within the constructor.

Much like how we can access the component's props with `this.props`, we can access the component's state via `this.state`. For example if we wanted to output our state property in a single `Text` component, we could do this:

```
export default class HiThere extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      text: 'Hi there!',
    };
  }

  render() {
    return <Text>{this.state.text}</Text>;
  }
}
```

This component would now render `'Hi there!'` since that's how we defined our `state.text` property in our constructor. For our current component however, our `text` property in state will be used to define the text typed by the user into the input field. Let's now modify our component's `render` method to allow for this:

weather/5/components/SearchInput.js

```
  render() {
    const { placeholder } = this.props;
    const { text } = this.state;

    return (
      <View style={styles.container}>
        <TextInput
          autoCorrect={false}
          value={text}
          placeholder={placeholder}
          placeholderTextColor="white"
          underlineColorAndroid="transparent"
          style={styles.textInput}
          clearButtonMode="always"
          onChangeText={this.handleChangeText}
          onSubmitEditing={this.handleSubmitEditing}
        />
```

```
      </View>
    );
  }
```

The first thing we did was destructure the component `props` and `state` objects:

weather/5/components/SearchInput.js

```
  render() {
    const { placeholder } = this.props;
    const { text } = this.state;
```

Destructuring

Instead of using `this.props.placeholder` and `this.state.text` directy, we destructured both objects at the beginning of our `render` method into individual variables (`text` and `placeholder`). Please refer to the Appendix for more details on destructuring assignments.

We then make sure that the `TextInput placholder` prop is still accepting our `props.placeholder` attribute. We also pass `state.text` to a `value` prop:

weather/5/components/SearchInput.js

```
        <TextInput
          autoCorrect={false}
          value={text}
          placeholder={placeholder}
          placeholderTextColor="white"
          underlineColorAndroid="transparent"
          style={styles.textInput}
          clearButtonMode="always"
          onChangeText={this.handleChangeText}
          onSubmitEditing={this.handleSubmitEditing}
        />
```

The `value` prop is responsible for the content showed in the input field. With this, we now know whatever is displayed in input field will **always represent our local state**.

We've also attached two additional props to our component, `onChangeText` and `onSubmitEditing` with methods we haven't set up yet.

Tracking changes to input

Let's take a look at how `onChangeText` can allow us to update our `state` every time the input field is changed. As we just did previously, we're attaching a method to the `onChangeText` prop of `TextInput`:

weather/5/components/SearchInput.js

```
          onChangeText={this.handleChangeText}
```

Previously, we set up a `handleChangeText` method that modifies our location prop value when the user changes the text within the input. We quickly realized that this didn't work. This is because `props` are immutable and are always "**owned**" **by a component's parent** while `state` can be mutated and is "**owned**" **by the component itself**. This is an extremely important pattern to remember while building components with React Native.

This brings us to `setState()`, a method we can use to **change our state** correctly. Let's make use of this in our `handleChangeText` method which we can declare right underneath our constructor:

weather/5/components/SearchInput.js

```
export default class SearchInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      text: '',
    };
  }
```

```
handleChangeText = text => {
  this.setState({ text });
};
```

Shorthand property names

With later versions of JavaScript, we can define objects using shorthand form where possible. Our `handleChangeText` method can also be written in a more explicit syntax:

```
handleChangeText = (text) => {
  this.setState({ text: text });
};
```

Please refer to the Appendix for a little more detail on this concept.

Now we might be tempted to update our state by using `this.state.text = text`, but this will **not work**. For all state modifications after the initial state we've defined in our `constructor`, React provides components with the method `setState()` to do this. In addition to mutating the component's `state` object, this method triggers the React component to re-render, which is essential after the `state` changes.

It's good practice to initialize components with "empty" state as we've done in this component. However, after our `SearchInput` component is initialized, we want to update the state with data the user types into the text input. This is why we use the `text` argument provided into our callback method as part of the `onChangeText` prop and pass that into `this.setState()`.

Never modify state outside of `this.setState()`. This function has important hooks around state modification that we would be bypassing.

We discuss state management in detail throughout the book.

Notifying the parent component

So we've found a way to correctly store local state in our component that represents the text within the search input *and* make sure that it updates as the user changes the

value. We still need to do one more thing which is to notify our parent `App` component when the user submits a new searched value. This is why we've attached a method to the `onSubmitEditing` prop of `TextInput`:

weather/5/components/SearchInput.js

```
          onSubmitEditing={this.handleSubmitEditing}
```

The idea here is we don't necessarily want to communicate with our parent component everytime the user changes the input field. That's why `onChangeText` is purely responsible for storing the latest typed input value into the local state of the component. Fortunately, the `TextInput` component has an `onSubmitEditing` prop which fires when the user **submits** the field and not just changes it. This happens specifically when the user presses the action button of the virtual keyboard in order to *submit* their input. This is where we would want to notify our container component of the typed user data. Let's take a look at how we can set up the `handleSubmitEditing` function that we're passing in:

weather/5/components/SearchInput.js

```
  handleSubmitEditing = () => {
    const { onSubmit } = this.props;
    const { text } = this.state;

    if (!text) return;

    onSubmit(text);
    this.setState({ text: '' });
  };
```

In here, we check if `this.state.text` is not blank (which means the user has typed something into the field), and if that's the case:

1. **Run an `onSubmit` function obtained from the component's props.** We pass `text` as an argument here.
2. Clear the `text` property in state using `this.setState()`

We've seen how `this.props` can be used to pass information down from a parent component to child and we've also seen how built-in components such as `TextInput` can notify their parent component through callbacks in some of their props. Similarly, we can create `props` in our custom components to do the *exact same thing*. In here, we need `SearchInput` to communicate with the `App` component whenever the user submits the input field. We do this because we want our parent component to handle the event of the user typing and submitting a new city. This is why we have an `onSubmit` prop here that gets fired.

The next thing we need to do is pass a method to the `onSubmit` prop of `SearchInput` in `App` and handle the event:

weather/5/App.js

```
<SearchInput
  placeholder="Search any city"
  onSubmit={this.handleUpdateLocation}
/>
```

Let's define local state for this component as well as the `handleUpdateLocation` method:

weather/5/App.js

```
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      location: 'San Francisco',
    };
  }

  handleUpdateLocation = city => {
    this.setState({
      location: city,
    });
  };
```

We defined local state for this component with just a `location` property and have it set to `San Francisco`. We do this to ensure that an initial location is shown when we reload our application. We also included a `handleUpdateLocation` method that takes in a parameter to change our location state. This method will fire everytime the user submits the search input field because we pass this method as the `onSubmit` prop for `SearchInput`.

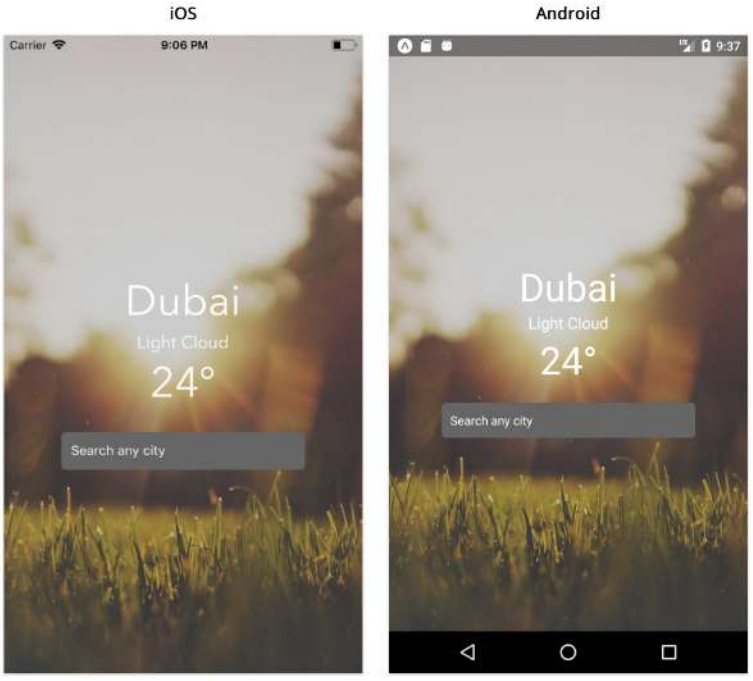
Since we actually have "living" location data represented by what the user submits in the input field, we can now display it in our first `Text` element instead of a hardcoded string:

weather/5/App.js

```
  render() {
    const { location } = this.state;

    return (
      <KeyboardAvoidingView
        style={styles.container}
        behavior="height"
      >
        <ImageBackground
          source={getImageForWeather('Clear')}
          style={styles.imageContainer}
          imageStyle={styles.image}
        >
          <View style={styles.detailsContainer}>
            <Text style={[styles.largeText, styles.textStyle]}>
```

Try it out

If we type any city in the search input and press return, we'll see the name of the city being displayed immediately.



Component State

This shows that we've wired everything correctly!

We've sequenced each of our problems step by step and showed how `props` and `state` differ when trying to pass and store component data. However, `handleUpdateLocation` doesn't really get *real* weather information and just updates the city name that's being displayed. We'll wire it up to get actual weather data soon.

Architecting state

We may have already considered controlling all the location state within `SearchInput` and not having to deal with passing information upwards to a container component. There's no specific answer to *where each piece of state should live* and it depends on the type of application we're building. This is a core concept of building React Native applications and tools like Redux[32] and MobX[33] aim to simplify this even further by allowing you to manage the entire state of the application in a single

[32] https://github.com/reactjs/redux
[33] https://github.com/mobxjs/mobx

location. However, even when we decide to use state management libraries such as these examples, we still need to spend time deciding on how we want to structure our state logic.

In our current app, we need to have `App` know the `location` data in order to display correct weather conditions. `SearchInput` doesn't really need to store this information without actually passing it up to the component that handles the logic. The motivation behind keeping `SearchInput` simple is that we can leverage React's component-driven paradigm. We can re-use it in various places across our application whenever we need a search input.

We can think of `SearchInput` as a component that provides presentational markup and **does not** manage any real application data. Such components accept props from parent components which specify the data a presentational component should render. This parent container component also specifies behavior. If the lower level presentational component has any interactivity — like our search input — it calls a prop-function given to it by the parent. We'll go into more detail about this important pattern throughout this book.

Lifecycle methods

We've wired up how our components communicate with each other to have a new location displayed immediately when the user submits the text input field. However, you'll notice that the city shows a blank string when the app first loads. We *could* instantiate it with the name of an actual city instead but we know we want to be getting actual weather information eventually. Although we haven't set that up just yet, the asynchronous action to fetch actual weather data for a city will be happening in the `handleUpdateLocation`. Therefore it makes sense to call this method when our component first loads. One thing we might be tempted to try is firing this method in our `constructor`:

```
constructor(props) {
  super(props);
  this.state = {
    location: '',
  };

  this.handleUpdateLocation('San Francisco');
}
```

However, firing off asynchronous requests in the `constructor` is typically an *anti-pattern*. This is because the `constructor` is called before the component is first mounted. As such, this method should usually only be used to initialize state and bind methods.

Instead, we can make use of one of React Native's **lifecycle methods**. Like the name suggests, these methods allow you to access specific points in the lifecycle of a component. The term lifecycle here applies to how React Native **instantiates**, **changes** and **destroys** components. We can use lifecycle hooks to do something when these functions are called during different phases of component rendering.

The most common lifecycle method used is the one that allows us to set component data *after* the component is mounted – **`componentDidMount()`**. This method is commonly used to trigger network requests to fetch data that the component would need. To understand when this method fires, let's add it to our component right after our `constructor` with a `console.log`:

```
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      location: 'San Francisco',
    };
  }

  componentDidMount() {
    console.log('Component has mounted!');
  }
```

When we reload our application, we can see `Component has mounted!` outputted directly to our terminal as soon as the component has mounted.

Debugging in React Native

If you've worked with JavaScript on the web, you may be familiar with using `console.log`, `console.warn` or `console.error` to output messages to the browser's console for debugging purposes. Similarly, Expo allows us to use these methods to output logs to our terminal. For more detail about viewing logs, you can refer to the documentation[34].

Aside from logging, React Native also allows us to debug the JavaScript code in our app using the Chrome Developer Tools. With Expo, we can do this by pressing `Debug Remote JS` in the developer menu. You can refer to the documentation[35] to learn more.

Now let's update it to fire `handleUpdateLocation`:

weather/6/App.js

```
  componentDidMount() {
    this.handleUpdateLocation('San Francisco');
  }
```

With this, we can remove `San Francisco` as our default location in `state` and set it to an empty string.

```
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      location: '',
    };
  }
```

[34] https://docs.expo.io/versions/latest/guides/logging.html
[35] https://docs.expo.io/versions/latest/guides/debugging.html

Since we're using `componentDidMount`, we should still see `San Francisco` populated in place of the text field as soon as we reload the app.

Although `componentDidMount()` allows us to create event listeners and fetch network requests right after the component has rendered for the first time, there are number of other lifecycle methods that React Native provides. We'll go through each of them throughout this book.

Networking

We've built all the components that make up the UI of our app and refined it to show a nice background image for the user. As we mentioned previously, the approach we've taken so far is a common pattern used when building brand new React Native applications. We first organized our views using components and then introduced some state and state management.

However, nobody will find our app *useful* unless it's actually connected to real data. When building a new mobile app, chances are we'll need to communicate with a server. Communicating with a server is a crucial component of most mobile applications.

For the purpose of this application, we'll use the MetaWeather[36] API to fetch real weather information. MetaWeather is a weather data aggregator that calculates the most likely outcome from predictions of different forecasters. They provide an API[37] that provides this information over a set of different endpoints:

1. Location search (`/api/location/search/`) which allows us to search for a particular city
2. Location weather information (`/api/location/{woeid}`) which provides a 5 day forecast for a certain location
3. Location day which provides (`/api/location/{woeid}/{date}/`) forecast history and information for a particular day and location

[36] https://www.metaweather.com/
[37] https://www.metaweather.com/api/

WOEID, or Where On Earth ID, is a location identifier that allows us find details about a specific location. For more detail on how exactly the MetaWeather API works, feel free to take a closer look at the documentation[38].

Now that we have a basic understanding of how `state` and `props` control the flow of data between different components, let's move on to using this API to render real weather data. It's possible to put API calls directly in our component methods, but it's usually a good idea to abstract that logic away in its own file. In the `utils` directory, we've set up two separate API calls in `api.js`:

- `fetchLocationId` returns an array of locations based on a search query
- `fetchWeather` returns weather details about a specific location using a location identifier known as Where On Earth ID[39]

The combination of both calls will allow us to search for a city and retrieve its weather information. Feel free to open the file and take a look at how these methods work if you're interested.

Async Functions

Callbacks and Promises are two ways to define asynchronous code in JavaScript. Built on top of promises, **`async`** functions are a newer syntax that allows us to define asynchronous methods in a synchronous manner. Both methods we've set up in `api.js` use this syntax.

Although supported by Babel, it is still in draft proposal stage and will most likely be ratified into a future JavaScript release. Here's the MDN[40] resource if you happen to be interested in learning more about this syntax further.

When building components that fetch information over the network, it's inevitable that the user will have to wait a certain period of time before the data is retreived. With most applications, it makes sense to show a loading indicator of some sort so the user knows they have to wait a bit before they can see the content. Fortunately, React

[38] https://www.metaweather.com/api/
[39] https://developer.yahoo.com/geo/geoplanet/guide/concepts.html
[40] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Native provides a built-in `ActivityIndicator` component that displays a circular loading spinner. Let's update our root `App` component beginning with some new imports:

weather/6/App.js

```
import React from 'react';
import {
  StyleSheet,
  View,
  ImageBackground,
  Text,
  KeyboardAvoidingView,
  Platform,
  ActivityIndicator,
  StatusBar,
} from 'react-native';

import { fetchLocationId, fetchWeather } from './utils/api';
import getImageForWeather from './utils/getImageForWeather';

import SearchInput from './components/SearchInput';
```

We've added the following imports:

- `ActivityIndicator` is a built-in component that displays a circular loading spinner. We'll use it when data is being fetched from the network
- `fetchLocationId`, `fetchWeather` are the methods for interacting with the weather API
- `StatusBar` is a built-in component that allows us to modify the app status bar at the top of the device

Now let's make some changes to our component. We need to apply our network request logic and store that information so that it can be easily displayed. We also need to make sure that a loading indicator is shown while the request is firing. Let's begin with updating our state:

weather/6/App.js

```
  constructor(props) {
    super(props);

    this.state = {
      loading: false,
      error: false,
      location: '',
      temperature: 0,
      weather: '',
    };
  }
```

We just expanded our state object to include `loading`, `error`, `temperature`, and `weather` in addition to `location`. The three latter properties are data we'll retreive from the API. The `loading` property represents when a call is still being made (in order to show a loading icon) and `error` is used to store the error message if our call fails or returns unusable information.

With `setState`, updates to our state can happen *asynchronously*. For this reason, the method accepts a callback as an optional second parameter that allows us to define an action to fire after the state is updated. Consider the following as an example:

```
export default class Example extends React.Component {
  state = {
    weather: '',
  };

  componentDidMount() {
    this.setState({ weather: 'Clear' }, () =>
      console.log(this.state),
    );
  }
}

// { weather: 'Clear' } is logged right after component mounts
```

We can apply this logic to the method responsible for interfacing with our external API: `handleUpdateLocation`:

weather/6/App.js

```
  handleUpdateLocation = async city => {
    if (!city) return;

    this.setState({ loading: true }, async () => {
      try {
        const locationId = await fetchLocationId(city);
        const { location, weather, temperature } = await fetchWeather(
          locationId,
        );

        this.setState({
          loading: false,
          error: false,
          location,
          weather,
          temperature,
        });
      } catch (e) {
        this.setState({
          loading: false,
          error: true,
        });
      }
    });
  };
```

We've updated it to be an asynchronous function that uses `setState` to change our loading attribute to `true`. We also pass in an asynchronous function as its second argument. In here, we first call `fetchLocationId` with the user queried city (if present) and pass the location ID to `fetchWeather` to return an object that contains the required information (`location`, `weather`, and `temperature`). Once complete, our

state is updated with the correct parameters. Moreover, if any of the calls happen to error, the catch statement will update the `error` property in our state to true.

Now that we have our API logic in place, we'll need to do a few things in the UI of our component:

- We need to display a loading spinner *only* when our API calls have fired but not completed
- We should show an error message if the user types in an incorrect address *or* our API call fails
- We need to render the correct weather information for a certain location

Let's take a look at how we can update our `render()` method to do this:

weather/6/App.js

```
render() {
  const {
    loading,
    error,
    location,
    weather,
    temperature,
  } = this.state;

  return (
    <KeyboardAvoidingView
      style={styles.container}
      behavior="height"
    >
      <StatusBar barStyle="light-content" />
      <ImageBackground
        source={getImageForWeather(weather)}
        style={styles.imageContainer}
        imageStyle={styles.image}
      >
        <View style={styles.detailsContainer}>
```

```
<ActivityIndicator
  animating={loading}
  color="white"
  size="large"
/>

{!loading && (
  <View>
    {error && (
      <Text style={[styles.smallText, styles.textStyle]}>
        Could not load weather, please try a different
        city.
      </Text>
    )}

    {!error && (
      <View>
        <Text
          style={[styles.largeText, styles.textStyle]}
        >
          {location}
        </Text>
        <Text
          style={[styles.smallText, styles.textStyle]}
        >
          {weather}
        </Text>
        <Text
          style={[styles.largeText, styles.textStyle]}
        >
          {`${Math.round(temperature)}°`}
        </Text>
      </View>
    )}

    <SearchInput
```

```
                placeholder="Search any city"
                onSubmit={this.handleUpdateLocation}
              />
            </View>
          )}
        </View>
      </ImageBackground>
    </KeyboardAvoidingView>
  );
}
```

It might look like a lot is going on in the file, but let's break it down piece by piece. We first included our `StatusBar` component:

weather/6/App.js

```
      <StatusBar barStyle="light-content" />
```

The `StatusBar` component allows us to customize the status bar of our application using a `barStyle` prop that lets us change the color of the text within the bar. A value of `light-content` renders a lighter color (white) and `dark-content` will change it to a darker color (dark-grey).

With Expo, we can also configure the status bar for Android by modifying `app.json`.

Expo defaults `barStyle` for Android to `light-content` and makes the background translucent. Although this looks fine for our current application, you can remove the translucency by providing a background color. Take a look at the documentation[41] for more details.

We then added `ActivityIndicator` along with assigning its color and size prop:

[41] https://docs.expo.io/versions/latest/guides/configuring-statusbar.html

weather/6/App.js

```
<ActivityIndicator
  animating={loading}
  color="white"
  size="large"
/>
```

Notice how we've also included an `animating` prop which we've set to be our `state.loading` attribute. This prop is responsible for showing or hiding the component entirely.

After that, we've included a curly brace container in our JSX:

weather/6/App.js

```
{!loading && (
  <View>
    {error && (
      <Text style={[styles.smallText, styles.textStyle]}>
        Could not load weather, please try a different
        city.
      </Text>
    )}

    {!error && (
      <View>
        <Text
          style={[styles.largeText, styles.textStyle]}
        >
          {location}
        </Text>
        <Text
          style={[styles.smallText, styles.textStyle]}
        >
          {weather}
        </Text>
        <Text
```

```
                style={[styles.largeText, styles.textStyle]}
              >
                {`${Math.round(temperature)}°`}
              </Text>
            </View>
          )}

          <SearchInput
            placeholder="Search any city"
            onSubmit={this.handleUpdateLocation}
          />
        </View>
      )}
    </View>
```

We've previously seen how JSX allows us to embed JavaScript expressions within curly braces. Fortunately, this lets us include operators as well, allowing us to **conditionally render** certain parts of our UI. In here, `!loading && <...>` means that this statement will evaluate and display the element if and only if `loading` is false. We can see we've pretty much wrapped most of the elements that make up our component within here, and this makes sense since we don't want to show any text fields or the search input while the API call is being fetched.

Conditional Rendering

Using logical && operators within the render method is not the only way to conditionally render parts of the component. At times, this approach can make it harder to read a component file if a significant number of lines are being conditionally rendered.

If this happens, it might be a good idea to use *helper methods*. For example, our render method can be rewritten following this pattern:

```
renderContent() {
  const { error } = this.state;
  return (
    <View>
      {error && <Text>Error</Text>}
      {!error && this.renderInfo()}
    </View>
  );
}
renderInfo() {
  const { info } = this.state;
  return <Text>{info}</Text>;
}
render() {
  const { loading } = this.state;
  return (
    <View>
      <ActivityIndicator
        animating={loading} color="white" size="large"
      />
      {!loading && this.renderContent()}
    </View>
  );
}
```

The React documentation[42] goes into more detail as well as explaining more ways to conditionally render parts of components.

[42] https://reactjs.org/docs/conditional-rendering.html

Now within the content that shows when the API call isn't being fired, we still need to be able to display an appropriate error message if there's an issue. We can use the `state.error` attribute to conditionally display text in this scenario:

weather/6/App.js

```
{error && (
  <Text style={[styles.smallText, styles.textStyle]}>
    Could not load weather, please try a different
    city.
  </Text>
)}

{!error && (
  <View>
    <Text
      style={[styles.largeText, styles.textStyle]}
    >
      {location}
    </Text>
    <Text
      style={[styles.smallText, styles.textStyle]}
    >
      {weather}
    </Text>
    <Text
      style={[styles.largeText, styles.textStyle]}
    >
      {`${Math.round(temperature)}°`}
    </Text>
  </View>
)}

<SearchInput
  placeholder="Search any city"
  onSubmit={this.handleUpdateLocation}
/>
```

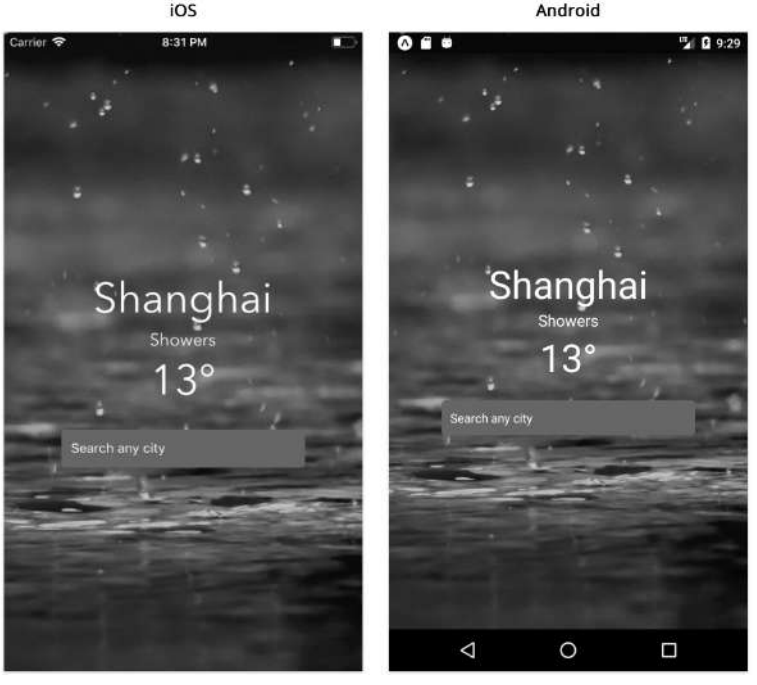
Notice how we now display our state information (`location`, `weather`, and `temperature`) in our `Text` elements instead of hard-coded values. For temperature, we're making use of the JavaScript `Math` object and its `round()` method to round the temperature to the nearest integer.

The last thing we also do is pass the dynamic `weather` attribute to `ImageBackground` instead of a hardcoded `Clear` string:

weather/6/App.js

```
        <ImageBackground
          source={getImageForWeather(weather)}
          style={styles.imageContainer}
          imageStyle={styles.image}
        >
```

Now if we run our application, typing a city into the input field will return its actual weather data!



We've pretty much finished connecting all the major points of our application by wiring in network requests to retreive actual data. After slowly beginning with hardcoded data and building our components that make up the building blocks of our UI, our application now works just as we intended from the beginning of this chapter. The next few sections will explore some additional enhancements to our code but won't add any new functionality to our app.

PropTypes

With React Native, we can include validation functions using the `prop-types` library. This allows us to specify and enforce the type of our component props and ensure that they match what we expect them to be. This can not only help us catch development errors sooner but also provide a layer of documentation to the consumer of our components

We can add `prop-types` as a dependency:

```
expo install prop-types
```

Aside from the "Native Modules" chapter, every application in this book uses Expo's managed workflow, and installing packages with the CLI can be done using `expo install`.

Although it is still possible to use `yarn add` to install any dependencies, using the `expo install` CLI command ensures that a compatible version of the package is installed. It will also automatically use `yarn` if a `yarn.lock` file exists in the project.

Now let's take a look at how we can use `PropTypes` in `SearchInput`:

weather/6/components/SearchInput.js

```
SearchInput.propTypes = {
  onSubmit: PropTypes.func.isRequired,
  placeholder: PropTypes.string,
};

SearchInput.defaultProps = {
  placeholder: '',
};
```

We've defined a `propTypes` object which instructs React to validate the props given to our component. We're specifying that `onSubmit` **must** be a function and `placeholder` **must** be a string. We've also specified `onSubmit` to be required which means it has to be provided to our component and is not optional.

We've left `placeholder` to be optional. For this, we're making use of the `defaultProps` object. This allows us to create our component and not specify `placeholder` if we don't need to, `defaultProps` will take care of providing it's value in that case. It's important to note that the value passed into `defaultProps` also undergoes type-checking as well by the library.

Now what exactly happens when a prop's type is not validated successfully? When a prop is passed in with an invalid type or fails the `propType` validation, a warning is passed into the JavaScript console. These warnings will only be shown in development mode, so if we accidentally deploy our app into production with an improper use of a component, our users won't see the warning.

Class properties

React Native includes class properties transformation[43] from Babel that allows us to simplify how we define our component `state`, `props`, and `propTypes`. For example, we can update the `constructor` in `App.js` to:

[43] https://babeljs.io/docs/plugins/transform-class-properties/

weather/App.js

```
  state = {
    loading: false,
    error: false,
    location: '',
    temperature: 0,
    weather: '',
  };
```

This gets transpiled into the **exact same result** as using a `constructor`. Similarly, we can simplify how we define our `state` in `SearchInput`:

weather/components/SearchInput.js

```
  state = {
    text: '',
  };
```

Moreover, we can also set `propTypes` and `defaultProps` using static properties in our class. In other words, we can remove the object references in `SearchInput` and define a static method *within* the class:

weather/components/SearchInput.js

```
  static propTypes = {
    onSubmit: PropTypes.func.isRequired,
    placeholder: PropTypes.string,
  };

  static defaultProps = {
    placeholder: '',
  };
```

Using this pattern and leveraging class properties transform is purely syntactical sugar over defining methods and objects separately and allows us to write in a cleaner, simpler syntax.

Summary

Congratulations, we've just built our very first React Native application and covered almost all of the essentials needed to build a complete and fully functional mobile app. We began by exploring each of the files generated as a result of starting a new project and how the Expo CLI allows us to run our application smoothly on our device without worrying about Xcode and Android Studio set up. We then built out each of the components that make up our application using the built-in components provided by React Native. While doing so, we dove into the fundamentals of React Native understanding JSX, how to apply custom styling as well as understanding how to use `props` and `state` to manage and control data. We moved on to more complex topics including lifecycle methods and how to use external network calls to provide real content to our application. Finally, we finished off with a brief look into how `propTypes` can add an additional layer of safety by adding type validation to our application. The rest of this book will dive deeper into core concepts of React Native and the concepts learned in this chapter will serve as the foundation for everything else in the text.

So far, we've only scratched the surface of what React Native allows us to do. By knowing the setup/development details like Expo and the core concepts of props, state, and components, you already have the essentials of React Native development under your belt. As of now, you can already build a wide variety of applications using the framework – so go forth and build something amazing!