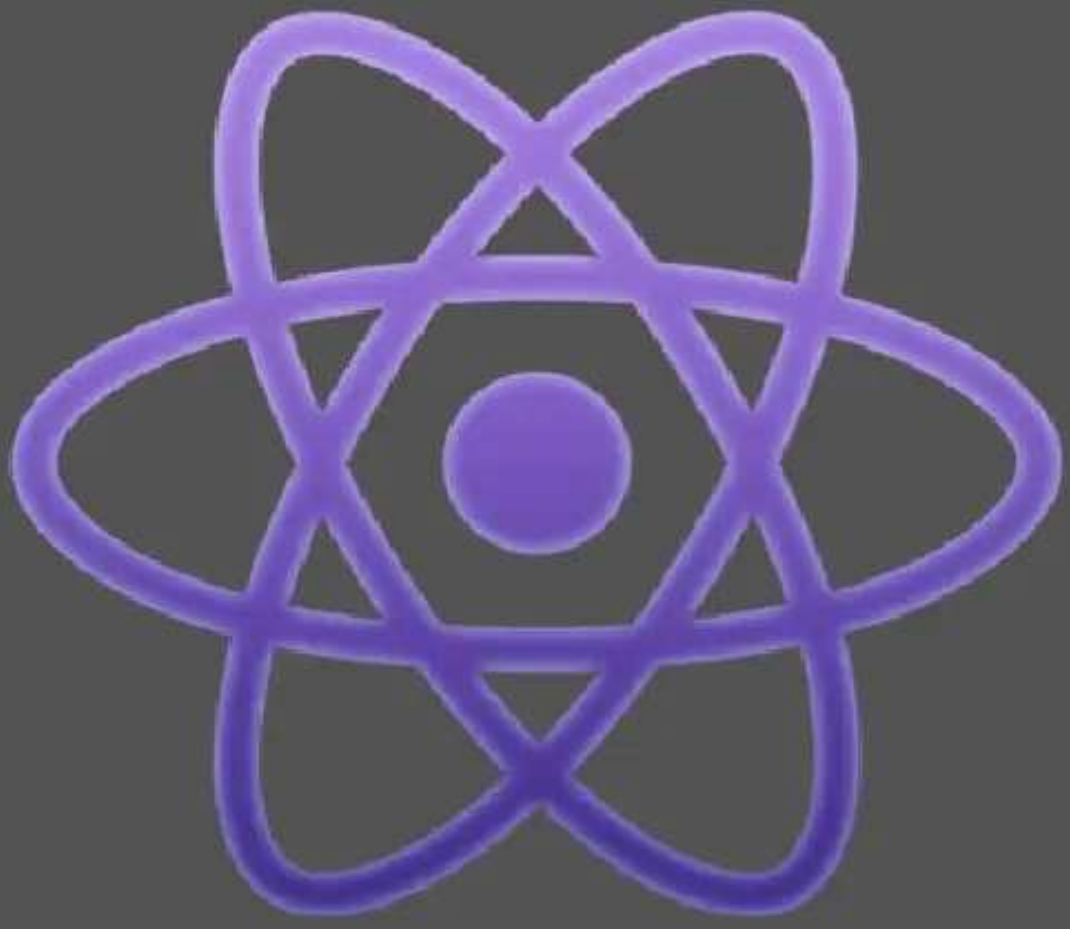
# Beginning React Native with Hooks



Greg Lim

# Beginning React Native with Hooks

Greg Lim

FIRST EDITION: JUNE 2020

# Table of Contents

# Preface

## About this book

Developed by Facebook, React Native is one of the leading frameworks to build native mobile apps for Android and iOS. You use small manageable components to build mobile applications that look and feel truly ‘native’.

In this book, we take you on a fun, hands-on and pragmatic journey to master React Native. You'll start building React Native apps using functional components within minutes. Every section is written in a bite-sized manner and straight to the point as I don’t want to waste your time (and most certainly mine) on the content you don't need. In the end, you will have what it takes to develop a real-life app.

## Requirements

Basic familiarity with HTML, Javascript and object-oriented programming. No prior knowledge of React Native is required as we start from React basics. But if you have previous experience with React, you will progress through the material faster.

## Contact and Code Examples

The source codes used in this book can be found in my GitHub repository at https://github.com/greglim81.

If you have any comments or questions concerning this book to support@i-ducate.com.

# Chapter 1: Introduction

## What is React Native?

React Native is a Javascript framework for building native mobile apps for Android and iOS using React. React in turn is a framework released by Facebook to build rich and interactive user interfaces. But instead of targeting the browser, React Native targets mobile platforms. React native compile React components into native components/widgets to create native iOS and Android applications. It enables web developers to write mobile applications that look and feel truly ‘native’ as React Native renders your applications using real mobile UI components, not webviews. React Native also exposes Javascript interfaces for platform APIs, so your React Native apps can access platform features like the phone camera or the user’s location.

Plus, most of the code you write in React Native can be shared between platforms. Usually, an iOS app and Android app are completely separate apps. iOS apps are coded in Swift, and Android apps in Java or Kotlin. With React Native, we can create one single codebase and build for both platforms. This saves lots of time and money in both the development and continual support for the app.

The popularity of React Native is evident given its usage at Facebook, Instagram, Microsoft, Amazon and thousands of other companies.

### *Requirements*

You can use this book whether you are on a Mac, Linux or Windows machine. All can use React Native to develop Android applications. And while it is possible to have workarounds to build for iOS on non-Mac machines, we recommend that you still develop for iOS on a Mac to avoid unexpected issues appearing in development and get the best experience for testing.

### *Developer Experience*

Although this book covers techniques for developing mobile applications with React Native, many principles that you learn in React Native works the

same as React for e.g. props, state, hooks. If you know how to write React Native code, you can easily transit to web development using React. And not only can you transition from mobile developer to web developer, but your code can also transit. We hope that this book will provide you with a strong base in which you can build applications in React beyond mobile apps.

## *Step by Step*

In this book, I will teach you about React Native from scratch in step by step fashion. You will build applications where you can list products (fig. 1.1a),

Figure 1.1a

input search terms and receive search results via GitHub RESTful API (fig. 1.1b).

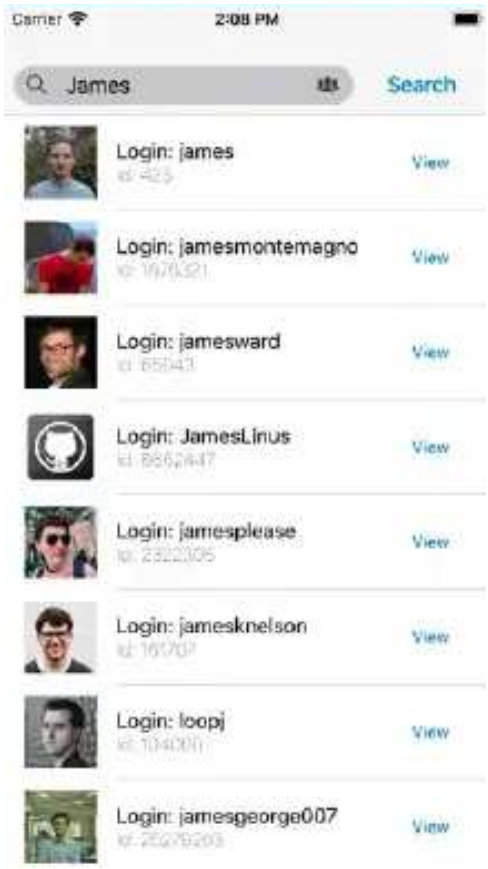
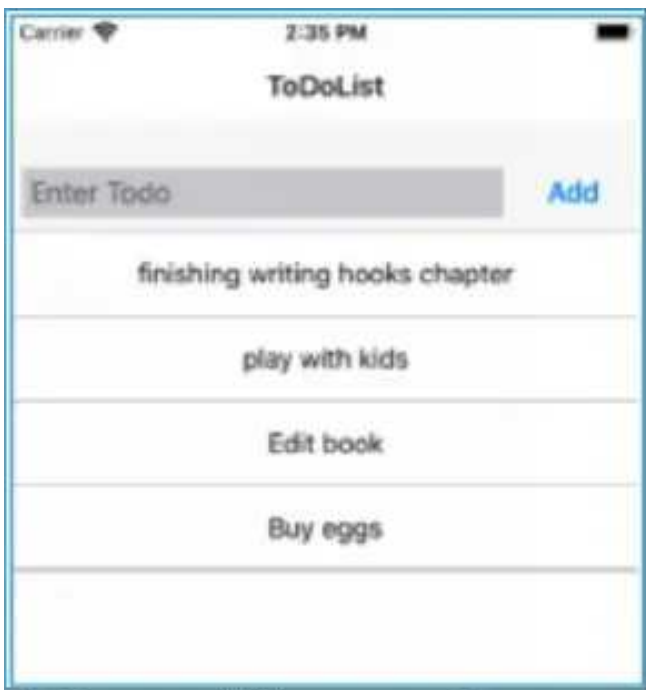
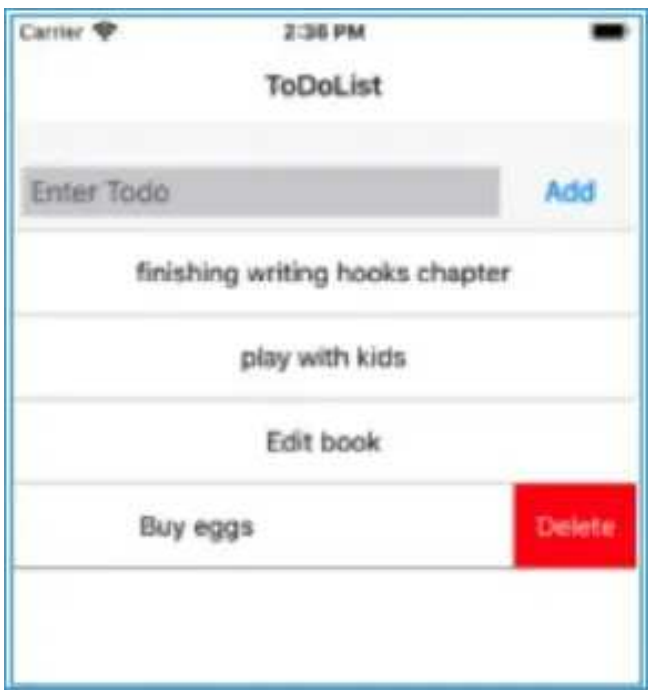



figure 1.1b

You will also build a todos application with full C.R.U.D. operations via a REST API (fig. 1.1c).



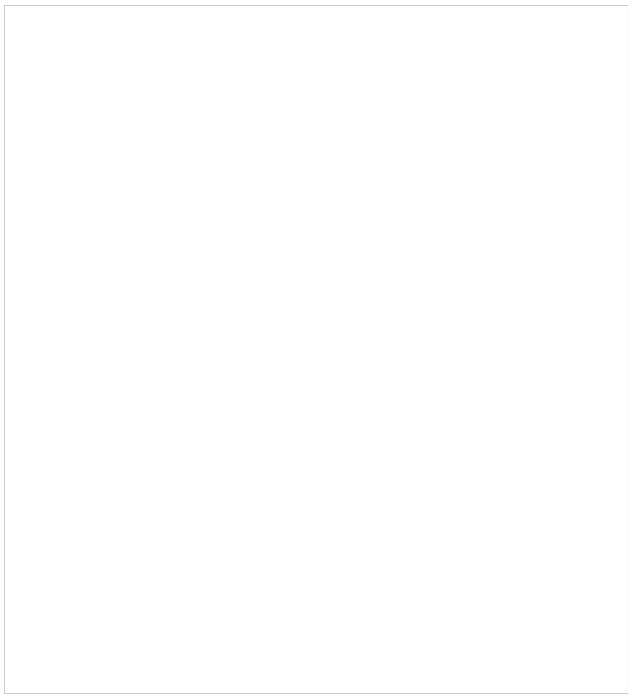


Figure 1.1c

These are the patterns you see on a lot of real-world applications. In this book, you will learn how to implement these patterns with React Native.

## Thinking in Components

A React Native is made up of components. For example, if we want to build a storefront module like what we see on Amazon, we can divide it into components. The search bar component, ProductList component, Product component and Rating component (fig. 1.2).


Figure 1.2

A React Native component has its own data and logic to control and output a portion of the screen via a JSX template.

Components can also contain other components. For example, in *ProductList* component where we display a list of products, we do so using multiple *Product* components. Also, in each *Product* component, we can have a *Rating* component.

The benefit of such an architecture helps us to break up a large application into smaller manageable components. Plus, we can reuse components within the application or even in a different application. For example, we can re-use the rating component in a different application.

Below is an example of a ProductList component that displays a simple string ‘ Products ’ .

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

export default function ProductList() {
    return (
      <View>
        <Text>
            Products
```

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

As mentioned earlier, we define our React Native components using a HTML like syntax known as JSX. JSX is a syntax extension to Javascript. Facebook released JSX to provide a concise syntax for rendering views. They hoped to make React more readable like HTML and XML.

This is the big picture of thinking in terms of components. As you progress through this book, you will see more of this in action.

For those who have experience with React components, they are largely the same with some differences around rendering and styling. We will talk more about this later.

## 1.3 Setting Up and Creating a New Project with Expo

### *Installing Node*

First, we need to install NodeJS. NodeJS is a server-side language and we don't need it because we are not writing any server-side code. We mostly need it because of its *npm* or Node Package Manager. *npm* is very popular for managing dependencies of your applications. We will use *npm* to install other later tools that we need.

Get the latest version of NodeJS from *nodejs.org* and install it on your machine. Installing NodeJS should be pretty easy and straightforward.

To check if Node has been properly installed, type the below on your command line (Command Prompt on Windows or Terminal on Mac):

```
node -v
```

and you should see the node version displayed.

To see if npm is installed, type the below on your command line:

```
npm -v
```

and you should see the npm version displayed.

## *Setting up the Development Environment*

The fastest way to set up the React Native development environment is with Expo CLI. Expo is a set of tools and services (*https://expo.io/*) built around React Native to let us enjoy many features. Among them, build, deploy and update apps without the need for Xcode/Android Studio. Expo provides access to device capabilities like camera, notifications and much more. But the most relevant feature for us right now is that it can get us writing a React Native app within minutes.

Use *npm* to install the Expo CLI command line utility:

```
npm install -g expo-cli
```

Then run the following command to create a new React Native project:

```
expo init <Project Name>
```

You will be asked to choose a template:

```
Choose a template: (Use arrow keys)
  ----- Managed workflow -----
❯ blank           a minimal app as clean as an empty canvas
  blank (TypeScript)    same as blank but with TypeScript configuration
  tabs            several example screens and tabs using react-navigation
 ----- Bare workflow -----
  minimal         bare and minimal, just the essentials to get you started
 minimal (TypeScript)  same as minimal but with TypeScript configuration
```

Choose the default ‘blank’ for now.

Expo will then download and install the dependencies needed to get our project started.

When the folder is created, navigate to it by typing.

```
cd <PROJECT NAME>
```

Next, type

```
npm start
```

This will start the React Native development server and there will be a QR code shown (fig. 1.3).


Figure 1.3
*Running your React Native application*

As mentioned by the Expo CLI, you can choose different ways to run the app with live reloading (when you make a code change, the app automatically reflects it).

You can run the app on your device by installing the Expo client app and connecting to the same wireless network as your computer (fig. 1.4).


Figure 1.4

On Android, use the Expo app to scan the QR code from your terminal to open your project. On iOS, use the built-in QR code scanner of the Camera app.

Alternatively, you can choose to run your app on either the Android emulator or the iOS simulator, but these have to be installed separately first. The

process of installation might take an hour or more. So, a fast option would be to deploy this to your device. Once you are familiar with deploying to your device, you can go ahead to install the simulator as it can help in faster development/testing.

## Project File Review

After you have successfully run your app, let's look at the project files that have been created for us. Open the project folder in your text editor of choice. In this book, we will be using VScode (https://code.visualstudio.com/) which is a good, lightweight and cross-platform editor from Microsoft.

You will find a couple of files when you open the project folder in your code editor (fig. 1.5).

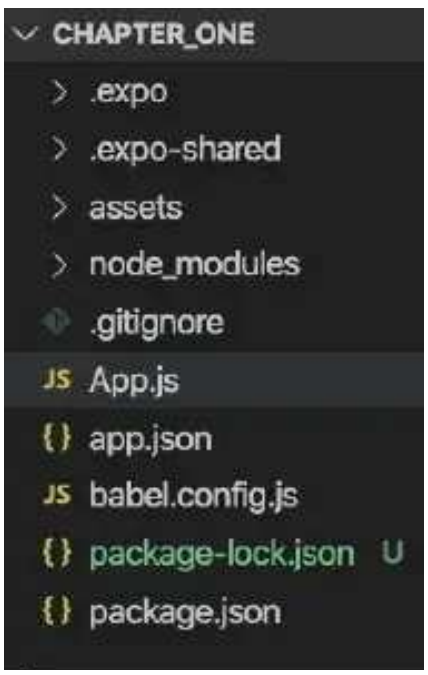



fig. 1.5

We will not go through all the files as our focus is to get started with our first React Native app quickly, but we will briefly go through some of the more important files.

We have *App.js* which renders the *App* root component for our app.

### *App.js*

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

export default function App() {
```

```
  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',
  },
});
```

In *App.js*, we have a function-based component called *App* as evident from the *function* declaration in the method header. Every React Native application has at least one component: the root component, named *App* in *App.js*. The App component controls the view through the JSX template it returns:

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

A component has to return a **single** element, i.e. a single <*View* /> or another custom component that you've defined yourself. We will dwell more on this in the next chapter.

As mentioned earlier, the funny *tag* syntax returned by the component is not HTML but JSX. JSX is a syntax extension to Javascript. We use it to describe what the UI should be like. Like HTML, in JSX, an element ' s type is specified with a tag. The tag ' s attributes represent the properties. Also, the element ' s children can be added between the opening and closing tags. In our JSX, we have used components like *View* and *Text*. We can use them because we have imported them at the top of *App.js*:

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

*For those who have experience with React, <View> in React Native is the equivalent of  in React. <Text> is the equivalent of .

We also have a *styles* object which uses the *StyleSheet* library and is defined at the bottom of the file:

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

});
```

You can define how Views look like with styles. We will discuss more on this later.

The function returns a JSX template in a single React element. For now, our root app component has no state properties or other functions. But later on, we will define application logic to interact with the view through state properties and functions with Hooks.

Back in our project folder, we also have the *package.json* file and *node_modules* folder:

*package.json* is the node package configuration which lists the third-party dependencies our project uses.

*node_modules* folder is created by NodeJS and puts all third-party modules listed in *package.json* in it.

# 1.5 Editing our first React Native Component

In *App.js*, change the text between <Text> to the following:

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

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Learn React Native!</Text>
    </View>
  );
}

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

```
  },
});
```

When you run your app, you should see something like (fig 1.6):


Figure 1.6

Try changing “ Learn React Native ” to some other text. Notice that the app reloads automatically with the revised title. Because React Native compiler is running in the ‘watch’ mode, it detects that there is a file change, re-compiles the app and the app gets refreshed automatically. In the case that your app does not reflect the change (often it is because of an error in the code), you might have to: either close and restart the app on either the device or on the simulator. Or, restart the Expo CLI and run *npm start* again in the Terminal. In the next section we cover the ‘ Red Screen of Death ’ which displays some common errors that developers make.

## The Red Screen of Death

A common error is when we attempt to use a variable without importing or defining it. For e.g. failing to import <Text> (fig. 1.7)

Figure 1.7

The red screen though alarming actually helps you to rectify errors in your code.

# Summary

In this chapter, we have been introduced to the core building blocks of React Native apps which are components. We have also been introduced to the React Native development experience which is creating a new React Native project with Expo. Expo provides tools and services to let us build, deploy and update apps without the need for Xcode/Android Studio. With Expo, we got our first React Native app running. In the next chapter, we will begin implementing more of our app.

# Chapter 2: Creating and Using Components

In the previous chapter, you learned about the core building blocks of React Native apps, components. In this chapter, we will implement a custom function-based component from scratch to have an idea of what it is like to build a React Native app.

## Creating our First Component

In your code editor, open the project folder that you have created in chapter 1. We first add a new file in and call it *ProductList.js* (fig. 2.1.1).

figure 2.1.1

Note the *PascalCase* naming convention of the file; i.e. we capitalize the first letter of each word, *ProductList.* We then append '*.js*' to it.

Next, type out the below code into *ProductList.js*:

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

export default function ProductList() {
  return (
    <View>
      <Text>Product List</Text>
    </View>
  );
```

```
}
```

## *Code Explanation*

*import React from 'react'* imports the ‘react’ library.

```
export default function ProductList()
```

*export default* in the function header makes this component available for other files (e.g. *App.js*) in our application to import it.

In *return*, we specify the JSX that determines the rendering of the component’s view. Our current JSX markup is:

```
<View>
  <Text>Product List</Text>
</View>
```

Note that components must return a single root element. If we have:

```
return (
  <View>
    <Text>Product List</Text>
  </View>
  <View>
    <Text>Course List</Text>
  </View>
);
```

The above will throw an error. So, we typically contain all internal elements under a single element for e.g.:

```
return (
  <View>
    <Text>Product List</Text>
      <Text>Course List</Text>
  </View>
);
```

With these simple lines of code, we have just built our first React Native component!

# 2.2 Using our Created Component

Now, go back to *App.js*. Notice that the contents of *App.js* is very similar to *ProductList.js*.

Remember that App component is the root of our application. It is the view component that controls our entire app.

Now, import and add *<ProductList />* to the template as shown below:

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

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Learn React Native!</Text>
      <ProductList />
    </View>
  );
}

const styles = StyleSheet.create({
        …
});
```

## *Code Explanation*

We have just referred to another component from a component. We can also render *ProductList* many times:

```
return (
  <div>
    Learn React Hooks
    <Products />
    <Products />
    <Products />
  </div>

);
```

Now save the file and go to your browser. You should see the Products component markup displayed with the message (fig. 2.2):


Figure 2.2

Notice that we have to first import our *ProductList* Component using
import ProductList from './ProductList';

For custom components that we have defined, we need to specify their path in the file system. Since *App* component and *ProductList* Component are in the same folder, we use ‘./’ which means start searching from the current folder followed by the name of the component, *ProductList* (without *.js* extension).

*<ProductList />* here acts as a custom tag which allows us to design and render our own custom components that are not part of standard JSX.

## 2.3 Embedding Expressions in JSX

You can embed Javascript expressions in JSX by wrapping it in curly braces. For example, we can define functions, properties and render them in the output. The below has a function *formatName* which takes in a *user* object which holds *firstName* and *lastName* properties. We then call *formatName* in *return* within the curly braces.

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

function formatName(user){
   return user.firstName + ' ' + user.lastName;
```

```
}
export default function App() {

  const user = {
    firstName:'Greg',
    lastName:'Lim'

  };
  return (
    <View style={styles.container}>
      <Text>Hello, {formatName(user)}</Text>
    </View>
  );
}
const styles = StyleSheet.create({
          …
});
```

If the value of the property in the *user* object changes, the view will be automatically refreshed.

Now, the font size is currently quite small. We will increase the font size by applying a style to <*Text*>. Add in the following in **bold**:

```
…
  return (
    <View style={styles.container}>
      <Text style={styles.textStyle}>Hello, {formatName(user)}</Text>
    </View>
  );
…
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center'
  },
  textStyle:{
    fontSize: 50
  }
});
```

And when we run our app, we get (fig. 2.3):


Figure 2.3

This is an example of how you can apply different styles to different elements.

## React Native UI – NativeBase

Defining and applying styles to React Native components can be quite complex. Adding to that is positioning and designing layout using Flexbox. I wish to focus more on the programming aspect of React Native rather than on UI styling and positioning in this book. Thus, I will be using *NativeBase.io* to make professional styling and layout easier. *NativeBase.io* (https://nativebase.io/) is a library of reusable UI components that contain JSX based templates to help build user interface components (like forms, buttons, icons) for React Native applications. Later on in this book, I will revisit some aspects of styling and layout using Flexbox.

### *Install and Setup NativeBase with Expo*

To install NativeBase, go to Terminal, and in your React Native project run:

```
npm install native-base --save
```

Next, install some Expo custom fonts with:

```
expo install expo-font
```

Now, back in *App.js*, we can apply some NativeBase components by adding the following in **bold**:

```
import React from 'react';
~~import { StyleSheet, Text, View } from 'react-native';~~ //remove
import { Container, Header, Content, H1, Text } from 'native-base';

function formatName(user){
  return user.firstName + ' ' + user.lastName;
}

export default function App() {

  const user = {
    firstName:'Greg',
    lastName:'Lim'
  };

  return (
    <Container>
      <Header />

      <Content>
        <H1>Hello,</H1>
        <Text>{formatName(user)}</Text>
      </Content>
    </Container>
  );
}
```

When we run our app now, it has a header with the text in large font.

In NativeBase, a common architecture is to have all components within <Container>. And in it, we have <Header> and <Content> much like in <html>, we have <head> and <body>.

In <Content>, we have the heading tag <H1> much like <h1>. We also have <Text> from native-base which replaces <Text> in the react-native library. These help us render professional looking typography without the need to explicitly state styling. You can refer to https://docs.nativebase.io/Components.html#Typography for more information about typography.

# Displaying a List

We will illustrate displaying a list of products in *ProductList*. In *ProductList.js*, add the codes shown in bold below:

```
import React from 'react';
import { Text, View } from 'react-native';
import { Container, Header, Content, List, ListItem, Text } from 'native-base';

export default function ProductList() {
    const products = ["Learning React","Pro React","Beginning React"];
    const listProducts = products.map((product) =>
        <ListItem key={product.toString()}>
            <Text>{product}</Text>
        </ListItem>
    );

  return (
    <Container>
        <Header />
        <Content>
        <List>
            {listProducts}
        </List>
        </Content>
    </Container>
  );
}
```

And in *App.js*, we import and render *ProductList*:

```
import React from 'react';
import ProductList from './ProductList';

export default function App() {

  return (
    <ProductList />
  );
}
```

Run your app now and you should see the result in fig. 2.4

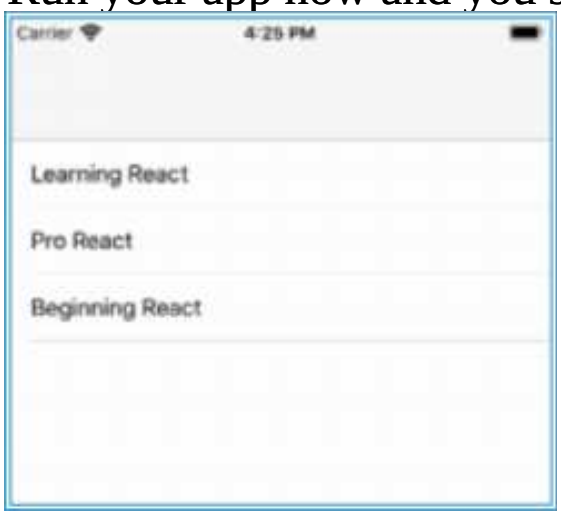



fig. 2.4

## *Code Explanation*

```
const products = ["Learning React","Pro React","Beginning React"];
```

First, in *ProductList* component we declare an array *products* component which contain the names of the products that we are listing.

```
const listProducts = products.map((product) =>
    <ListItem key={product.toString()}>
        <Text>{product}</Text>
    </ListItem>
);
```

We next define an ES6 arrow function

```
(product) =>
        <ListItem key={product.toString()}>
            <Text>{product}</Text>
        </ListItem>
```

that returns a *<ListItem>* element for each product. We then pass in this function into *map* which loops through each element, calls the function that returns a *<ListItem></ ListItem >* element for each product, and we are returned a new array of elements which we assign to *listProducts.*

```
return (
  <Container>
      <Header />
      <Content>
      <List>
          {listProducts}
      </List>
      </Content>

);</Container>
```

We include the entire *listProducts* array inside a <List> element. The *List* component helps us specify scrollable lists of information and provides styling and interaction attributes so that they are intuitive for users to interact with. We will explore more on Lists in the coming chapters. You can find out more on *List* at *https://docs.nativebase.io/Components.html#list-def-headref.*

Note that we have provided a *key* attribute for our list items. A "key" is a special string attribute you need to include when creating lists of elements. If you don’t provide this attribute, you will still have your items listed but a warning message will be displayed. Keys help React Native identify which

items have changed, are added, or are removed. Keys should ideally be strings that uniquely identify a list item among its siblings. Most often, you would use IDs from your data as keys. But in our case, we do not yet have an id. Thus we use *product.toString()*. You should always use keys as much as possible because bugs creep into your code (especially when you do operations like deleting, editing individual list items – you delete/edit the wrong item!) when you do not use it.

## Summary

You have learned a lot in this chapter. If you get stuck while following the code or if you would like to get the sample code we have used in this chapter, visit my GitHub repository at *https://github.com/greglim81/react-native-chp2/* or contact me at support@i-ducate.com.

In this chapter, we created our first custom component. We created a ProductList Component that retrieves product data from an array and later renders it on the page. With NativeBase, we defined and applied professional styling to help build our user interface.

# Chapter 3: Bindings, Props, State and User Interaction

In this chapter, we will explore displaying data by binding controls in a JSX template to properties of a React Native component, how to apply styles dynamically, how to use the component state and how to facilitate user interaction.

## Bindings

In the following code, we show a button in our view using *NativeBase* to make our button look more professional. As mentioned earlier, NativeBase (https://nativebase.io/) is a library of reusable front-end components that contain JSX based templates to help build user interface components (like forms, buttons, icons) for mobile applications.

Let's add a button into our *App* component with the below code:

```
import React from 'react';
import { Container, Header, Content, Button, Text } from 'native-base';

export default function App() {

  return (
    <Container>
      <Header />
      <Content>
        <Button>

          <Text>Click Me!</Text>
        </Button>
      </Content>
    </Container>
  );
}
```

You should get your button displayed like in fig. 3.1.

fig. 3.1

There are times when we want to use different styles on an element. For example, if we add the ‘ danger ’ variant as shown below:

```
return (
  <Container>
    <Header />
    <Content>
      <Button>
        <Text>Click Me!</Text>
      </Button>
      <Button danger>
        <Text>Danger</Text>
      </Button>
    </Content>
  </Container>
);
```

we get the below button style (fig. 3.2).


Figure 3.2

And if I want to disable the button by applying the *disabled* property, I can do the following:

```
<Button danger disabled>
  <Text>Danger</Text>
</Button>
```

More information of styles of *button* and other components are available at the NativeBase site under ‘ Components ’ (fig. 3.3).

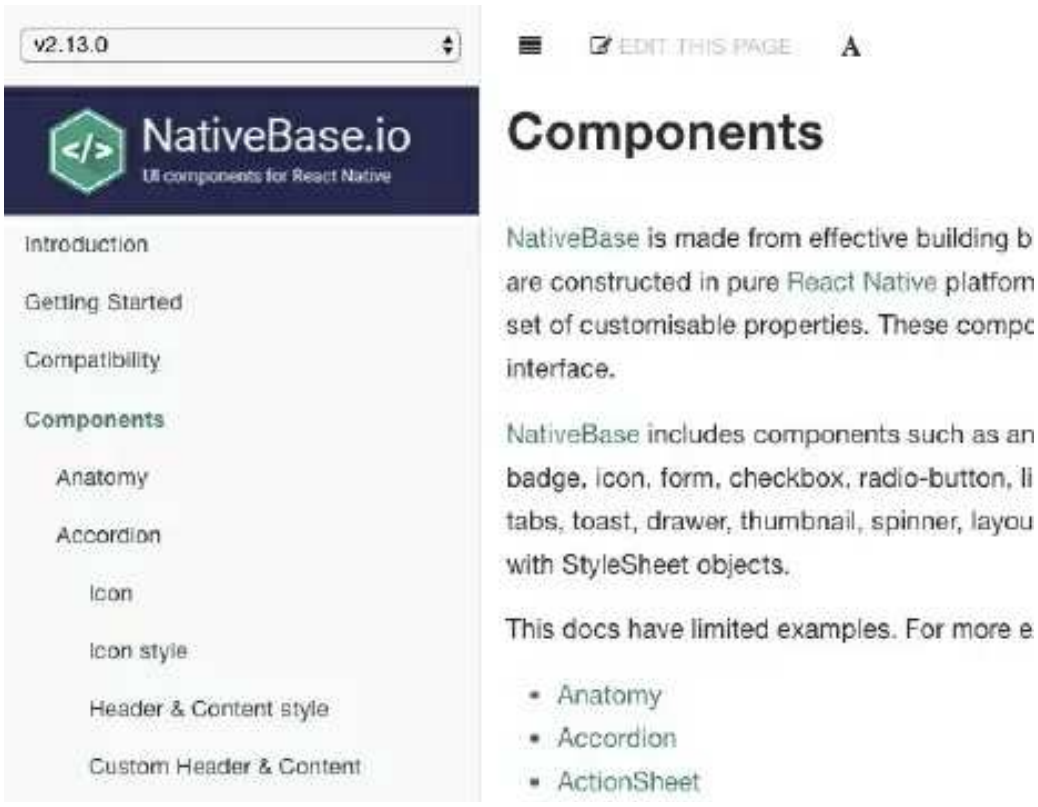


Figure 3.3

## *Disabling Button on Condition*

Now, suppose we want to disable the button based on some condition, we can do the below:

```
import React from 'react';
import { Container, Header, Content, Button, Text } from 'native-base';

export default function App() {

  const isValid = true;

  return (
    <Container>
      <Header />
      <Content>
        <Button disabled={!isValid}>
          <Text>Click Me!</Text>
        </Button>
      </Content>
    </Container>
  );
}
```

That is, when *isValid = false* the *disabled* property will be applied, making the button unclickable. If *isValid = true* the *disabled* property will not be applied, making the button clickable.

# Props

We can pass data into a component by passing in a single object called ‘props’. The ‘props’ object contains JSX attributes. For example, suppose we want to display a list of products with its rating. We will need to assign the rating value to our rating component beforehand. We can do something like: *<Rating rating="4"/>* to display a rating of 4 stars.

‘props’ will contain the value of 4 assigned to the *rating* attribute. To access ‘props’ in our Rating component, we use *props.rating*.

For example, create a new file *Rating.js* (our Rating component), with the below code that renders the rating value on the page.

```
import React from 'react';
import { H1 } from 'native-base';

export default function Rating(props) {

  return (
    <H1>Rating: {props.rating}</H1>
  );
}
```

In *App.js*, add in the codes below into *return*:

```
import React from 'react';
import Rating from './Rating';
import { Container, Header, Content } from 'native-base';

export default function App() {


  return (
    <Container>
      <Header />
      <Content>
        <Rating rating='1' />
        <Rating rating='2' />
        <Rating rating='3' />
        <Rating rating='4' />
        <Rating rating='5' />
      </Content>
    </Container>
  );
}
```

Note that if *App* already has *Container*, *Header* and *Content*. *Rating* should

not need to have them.

If you run your app now, it should display something like (fig. 3.4):


Figure 3.4

To recap, we *return* in *App.js* *<Rating rating="1"/>* . React Native calls the Rating component with *{ rating: '1' }* as the props. Our Rating component returns a *<H1>Rating: 1</H1>* element as a result.

In this example, our *props* object contains only one attribute. But it can contain multiple and even complex objects as attribute(s). We will illustrate this later in the book.

## Props are Read-Only

Note that when we access props in our components, we must never modify them. Our functions must always be 'pure' – which means that **we do not attempt to change our inputs and must always return the same result for the same inputs.** In other words, props are read-only. For example, the below function is impure and not allowed:

```
return (
   <H1>Rating: {props.rating++}</H1>
);
```

We can use React Native flexibly but it has a single strict rule: that all React Native components must act like pure functions concerning their props. So how do we make our application UI dynamic and change over time? Later on, we will introduce the concept of ' state ' , where we use it to change our output over time in response to user actions or network responses without

violating this rule.

But first, we will improve the look of our rating component by showing rating stars like what we see in the Amazon mobile app instead of showing the rating value numerically. A user can click select from a rating of one star to five stars. We will implement this as a component and reuse it in many places. For now, don't worry about calling a server or any other logic. We just want to implement the UI first.

# Improving the Look

To show rating stars instead of just number values, we will use the *Icon* component provided in NativeBase *https://docs.nativebase.io/Components.html#icon-def-headref* which provides popular icons in our React Native project (fig. 3.5).

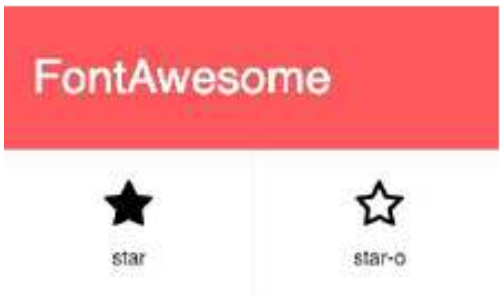



Figure 3.5

To include icons, add the below codes in **bold** into Rating component:

```
import React from 'react';
import { H1, Icon } from 'native-base';
import { Row } from 'react-native-easy-grid';

export default function Rating(props) {

  return (
    <Row>
        <H1>Rating: {props.rating}</H1>


        {props.rating >= 1 ? (
              <Icon type="FontAwesome" name="star" />
        ) : (
              <Icon type="FontAwesome" name="star-o" />
        )}
        {props.rating >= 2 ? (
              <Icon type="FontAwesome" name="star" />

        ) : (  <Icon type="FontAwesome" name="star-o" />
```

```
        )}
        {props.rating >= 3 ? (
            <Icon type="FontAwesome" name="star" />
        ) : (
            <Icon type="FontAwesome" name="star-o" />
        )}
        {props.rating >= 4 ? (
            <Icon type="FontAwesome" name="star" />
        ) : (
            <Icon type="FontAwesome" name="star-o" />
        )}
        {props.rating >= 5 ? (
            <Icon type="FontAwesome" name="star" />
        ) : (
            <Icon type="FontAwesome" name="star-o" />
        )}
    </Row>
  );
}
```

## *Code Explanation*

We first import *Icon* from NativeBase with

```
import { H1, Icon } from 'native-base';
```

In the *return* method, we render the *star* and *star-o* icons with:

```
{props.rating >= 1 ? (
    <Icon type="FontAwesome" name="star" />
) : (
    <Icon type="FontAwesome" name="star-o" />
)}
```

## *Conditional Rendering*

We conditionally render an *star* (filled star) if *props.rating is >= 1*. Else, render *star-o* (empty star). We will dwell more on the *If-Else* conditional code in chapter 5.

The above code is for the first star. The remaining similar repetitions are for the four remaining stars. However, note the change in value of each condition depending on which star it is. For example, the second star's condition should be

```
{props.rating >= 2 ? (
    <Icon type="FontAwesome" name="star" />
```

```
) : (
        <Icon type="FontAwesome" name="star-o" />
)}
```

The second star should be empty if the rating is less than two. It should be filled if the rating is more than or equal to two. The same goes for the third,

fourth and fifth star.
Lastly, we wrap all five stars in the <Row> component so that they are displayed in a row (horizontal). Note that React Native by default displays them in a column (vertical).

### *Running your App*

When we run our app, we get the icons displayed (fig. 3.6):


Figure 3.6

## Adding Local State to a Component

Now, suppose we want our user to be able to change the rating by clicking on the specified star. How do we make our rating component render in response to a user click? And considering that we cannot modify *props.rating*?

This is where we have to add ‘ state ’ to our Rating component. State is similar to props, but it is private and fully controlled by the component. State manages data that will change within a component. Whenever state changes, the UI is re-rendered to reflect those changes. We often refer to this as the component or local state. To add local state to our function component, we first use React Hooks ’ *useState* to assign an initial state:

```
import React, { useState } from 'react';
```

…

```
export default function Rating(props) {
    const [rating, setRating] = useState(props.rating)
  …
```

*useState* is a hook which we call to add some local state to a component. React Native preserves this state between re-renders of the component. *useState* returns an array with two values: the current state value and a function that lets you update it. In the above, we assign the current state rating value to *rating* and the function to update it to *setRating*.

(If you have experience with state in class-components, it ’ s similar to *this.setState*, except that state in class components is a single object that contains one or more attributes. In contrast, state in function components is declared and managed per variable, i.e. *setRating*. This actually makes state related code in a function component more readable and clearer about what we are updating, rather than calling *setState* in a class component. We will explore more on this later)

We initialize our initial state in *useState(props.rating)*. Our current state is a single attribute *rating* which is assigned the value from *props.rating*. You can also initialize rating in state to 0 by default with const [rating, setRating] = useState(0)

Next, replace *props.rating* with just *rating* in *return*:

```
return (
  <Row>
      <H1>Rating: {rating}</H1>

      {rating >= 1 ? (
          <Icon type="FontAwesome" name="star" />
      ) : (
          <Icon type="FontAwesome" name="star-o" />
      )}
      {rating >= 2 ? (
          <Icon type="FontAwesome" name="star" />
      ) : (
          <Icon type="FontAwesome" name="star-o" />
      )}
      {rating >= 3 ? (
          <Icon type="FontAwesome" name="star" />
      ) : (
          <Icon type="FontAwesome" name="star-o" />
      )}
```

```
        {rating >= 4 ? (
            <Icon type="FontAwesome" name="star" />
        ) : (
            <Icon type="FontAwesome" name="star-o" />
        )}
        {rating >= 5 ? (
            <Icon type="FontAwesome" name="star" />
        ) : (
            <Icon type="FontAwesome" name="star-o" />
        )}
    </Row>
  );
```

If you run your app now, it should display the Rating component just like before. The purpose of why we use the local state ' s rating instead of *props.rating* will become more apparent in the following sections.

# Handling Events with States

Next, we want to assign a rating depending on which star the user has clicked. To do so, our component needs to handle the click event. Handling events with React Native components is very similar to handling events on DOM elements. However, with JSX we pass a function as the event handler, rather than a string. For example, to make our rating component handle user clicks, we add the following in **bold** in *import* and in the *return* method:

```
import React, { useState } from 'react';
import { TouchableWithoutFeedback } from "react-native";
import { H1, Icon } from 'native-base';
import { Row } from 'react-native-easy-grid';

…
…
…

  return (
    <Row>
        <H1>Rating: {rating}</H1>

        {rating >= 1 ? (
            <TouchableWithoutFeedback onPress={() => setRating(1)}>
                <Icon type="FontAwesome" name="star" />
            </TouchableWithoutFeedback>
        ) : (
            <TouchableWithoutFeedback onPress={() => setRating(1)}>
                <Icon type="FontAwesome" name="star-o" />
            </TouchableWithoutFeedback>
```

```
)}
{rating >= 2 ? (
        <TouchableWithoutFeedback onPress={() => setRating(2)}>
            <Icon type="FontAwesome" name="star" />
        </TouchableWithoutFeedback>
) : (
        <TouchableWithoutFeedback onPress={() => setRating(2)}>
            <Icon type="FontAwesome" name="star-o" />
        </TouchableWithoutFeedback>
    )}
{rating >= 3 ? (
        <TouchableWithoutFeedback onPress={() => setRating(3)}>
            <Icon type="FontAwesome" name="star" />
        </TouchableWithoutFeedback>
    ) : (
        <TouchableWithoutFeedback onPress={() => setRating(3)}>
            <Icon type="FontAwesome" name="star-o" />
        </TouchableWithoutFeedback>
   )}
{rating >= 4 ? (
        <TouchableWithoutFeedback onPress={() => setRating(4)}>
            <Icon type="FontAwesome" name="star" />
        </TouchableWithoutFeedback>
    ) : (
        <TouchableWithoutFeedback onPress={() => setRating(4)}>
            <Icon type="FontAwesome" name="star-o" />
        </TouchableWithoutFeedback>
      )}
    {rating >= 5 ? (
        <TouchableWithoutFeedback onPress={() => setRating(5)}>
            <Icon type="FontAwesome" name="star" />
        </TouchableWithoutFeedback>
            ) : (
        <TouchableWithoutFeedback onPress={() => setRating(5)}>
            <Icon type="FontAwesome" name="star-o" />
        </TouchableWithoutFeedback>
      )}
  </Row>
);
```

*Touchable* is a wrapper which provides the *onPress* event handler to make views respond properly to touches. In each star, we pass in an arrow function as the event handler with rating value to the *onPress* event of *TouchableWithoutFeedback*. For example, we have `onPress={() => setRating(1)}` to assign a rating of one if a user clicks on this star. We then change the value of the argument to the arrow function depending on which star it is. The second star's *onClick* should be `onClick={() => setRating(2)}`. So, when a user clicks on the second star, the *setRating* method is called with property *rating*

of value two.  When a user clicks on the third star, the *setRating* method is called with property *rating* of value three and so on.

If you are not familiar with arrow functions, i.e. what we have in *onClick*:

```
() => setRating(5)
```

This is the same as:

```
function(){setRating(5)}
```

Many React developers declare functions in this manner thinking they result in shorter and simpler code in certain cases. In React Native development, we have to get used to reading and even writing our own arrow functions. In the function components we defined earlier, they can be implemented with arrow functions as well. We will cover more arrow functions in the course of this book.

Note that we CANNOT modify our state directly like *rating = 1*. Whenever we want to modify our state, we must use the state setter method we declared earlier, i.e. *setRating*

```
const [rating, setRating] = useState(props.rating)
```

Note that whenever any setter method is called, our component automatically re-renders thus showing the updated value on to the view.

## *Running your App*

When you run your app now, you should be able to see your ratings and also adjust their values by clicking on the specified star (figure 3.7).


fig. 3.7

Note that we have five different rating components each having their own local state. Each updates independently. Each rating component does not affect another rating component's state.

## *Summary*

In this chapter, we learned about binding controls in our JSX template to properties of a React Native component. We learned about passing data into a component through props and adding local state with the useState hook. We also learned about handling events through the state. In the next chapter, we will see how to put multiple components together in an application.

Visit my GitHub repository at *https://github.com/greglim81/react-native-chp3/* if you have not already have the full source code for this chapter or contact me at support@i-ducate.com if you encounter any errors with your code.

# Chapter 4: Working with Components

In this chapter, we will learn more about using components, how to reuse them and put them together in an application. Execute the codes in the following sections in your existing project from chapter three.

## Styles

We can use our own styles together with the components provided by NativeBase. These *styles* are scoped only to your component. They won't effect to outer or other components.

To illustrate, suppose we want our filled stars to be orange, in *Rating.js* we add the following in **bold** after *export default Rating*:

```
export default function Rating(props) {
            …
}

const styles={
  starStyle:{
    color: 'orange'
  }
}
```

We created the *styles* object with styling specifications under the Rating component. If required, you can further specify other styling properties like *height*, *backgroundColor*, *fontSize* etc.

To apply this style, add the below *style* attribute in the <*Icon*> component.

```
        …
    <TouchableWithoutFeedback onPress={() => setRating(1)}>
       <Icon type="FontAwesome" name="star" style={styles.starStyle} />
    </TouchableWithoutFeedback>
            …
```

When we run our application, we will see our filled stars in orange (fig. 4.1).

figure. 4.1

# Example Application

We will reuse the rating component that we have made and implement a product listing like in figure 4.2.

fig. 4.2

This is like the list of products on Amazon. For each product, we have an image, the product name, the product release date, the rating component and the number of ratings it has.

Create a new component file *ProductCard.js*. This component will be used to

render one product.

Now, how do we get our template to render each product listing like in figure 4.2? We use the *Card* component in NativeBase (https://docs.nativebase.io/Components.html#card-showcase-headref). Go to nativebase.io, in 'Docs', 'Components', click on 'Card Showcase' (fig. 4.3a). Copy the JSX markup there into our *ProductCard* Component.

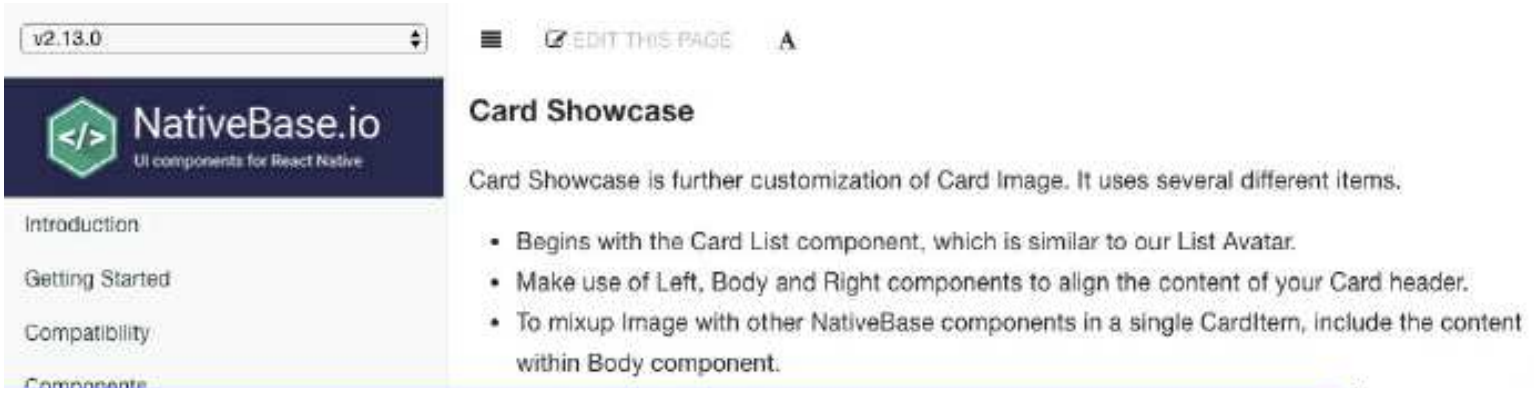



Figure 4.3a

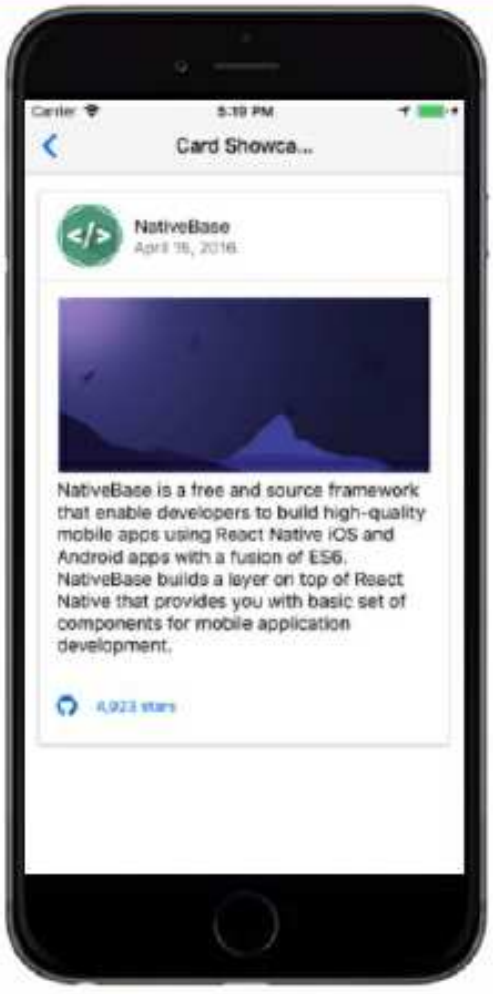



Figure 4.3b

Copy the markup into the *return* method of *ProductCard.js*. We do not need the *Container*, *Header* and *Content* tags since they will be in *App.js*, so

remove them. Your *ProductCard.js* should look something like:

```
import React from 'react';
import { Image } from 'react-native';
import { Card, CardItem, Text, Left, Body, Right } from 'native-base';
import Rating from './Rating';


export default function ProductCard(props) {
  return (
    <Card style={{flex: 0}}>
        <CardItem>
        <Left>
              <Body>
              <Text>{props.data.productName}</Text>
              <Text note>{props.data.releasedDate}</Text>
              </Body>
        </Left>
        <Right>
              <Rating
                    rating={props.data.rating}

              />
        </Right>
        </CardItem>
        <CardItem>
        <Body>
              <Image source={{uri: props.data.imageUrl}} style={{height: 100, width: 100, flex:
1}}/>
              <Text>
              {props.data.description}
              </Text>
        </Body>
        </CardItem>
    </Card>
  );
}
```

## *Code Explanation*

Note that the codes in **bold** use *props* to assign values of our product into our JSX. Essentially, our *ProductCard* component is expecting a props *data* object with the fields:
*imageUrl*, *productName*, *releasedDate* and *description*. We will pass the props in from *ProductList*.

We have also added our rating component that expects an input rating.

```
<Rating
    rating={props.data.rating}
```

/>

## *ProductList.js*

Next in *ProductList.js*, add a method *getProducts* that is responsible for returning a list of products. Type in the below code (or copy it from my GitHub repository ~~*https://github.com/greglim81/react-native-chp4*~~) into *ProductList.js*.

```
 const getProducts = () => {
return [
{
    imageUrl: "http://loremflickr.com/150/150?random=1",
    productName: "Product 1",
    releasedDate: "May 31, 2016",
    description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean porttitor, tellus laoreet venenatis facilisis, enim ex faucibus nulla, id rutrum ligula purus sit amet mauris. ",
    rating: 4,
    numOfReviews: 2

},
    imageUrl: "http://loremflickr.com/150/150?random=2",
    productName: "Product 2",
    releasedDate: "October 31, 2016",
    description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean porttitor, tellus laoreet venenatis facilisis, enim ex faucibus nulla, id rutrum ligula purus sit amet mauris. ",
    rating: 2,
    numOfReviews: 12
},
{
    imageUrl: "http://loremflickr.com/150/150?random=3",
    productName: "Product 3",
    releasedDate: "July 30, 2016",

    description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean porttitor, tellus laoreet venenatis facilisis, enim ex faucibus nulla, id rutrum ligula purus sit amet mauris. ",
    rating: 5,
    numOfReviews: 2
}];
}
```

Notice that in our class, we currently hardcode an array of product objects. Later on, we will explore how to receive data from a server.

For *imageUrl*, we use http://loremflickr.com/150/150?random=1 to render a random image 150 pixels by 150 pixels. For multiple product images, we change the query string parameter *random=2, 3,4* and so on to get a different random image.

The *getProducts* method will be called in our *ProductList* component. We return the results from *getProducts* to a *products* variable. Add the codes below into *ProductList.js*.

## *ProductList.js*

```
import React from 'react';
import { Content } from 'native-base';
import ProductCard from './ProductCard';

export default function ProductList() {

    const getProducts = () => {
        return [
        {
                imageUrl: "http://loremflickr.com/150/150?random=1",
                productName: "Product 1",
                releasedDate: "May 31, 2016",
                description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean porttitor, tellus laoreet venenatis facilisis, enim ex faucibus nulla, id rutrum ligula purus sit amet mauris. ",
                rating: 4,
                numOfReviews: 2
        },
        {
                imageUrl: "http://loremflickr.com/150/150?random=2",
                productName: "Product 2",
                releasedDate: "October 31, 2016",
                description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean porttitor, tellus laoreet venenatis facilisis, enim ex faucibus nulla, id rutrum ligula purus sit amet mauris. ",
                rating: 2,
                numOfReviews: 12
        },
        {
                imageUrl: "http://loremflickr.com/150/150?random=3",
                productName: "Product 3",
                releasedDate: "July 30, 2016",
                description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean porttitor, tellus laoreet venenatis facilisis, enim ex faucibus nulla, id rutrum ligula purus sit amet mauris. ",
                rating: 5,
                numOfReviews: 2
        }];
    }

    const products = getProducts();


    const listProducts = products.map((product) =>
    <ProductCard key={product.productName} data={product} />
      );
```

```
  return (
        <Content>
              {listProducts}
        </Content>
  );
}
```

The code in *return*() is similar to the one in chapter three where we loop through the names in *products* array to list them. This time however, our element is not just simple strings but an object which itself contains several *ProductCard* properties.

The function we define now returns a *<ProductCard>* component with the product *data* object as input for each product. Each *data* object input provides *ProductCard* component with values from properties *imageUrl*, *productName*, *releasedDate*, *description* and *rating*.

We pass in this function into *map* which loops through each element, calls the function that returns a *<ProductCard />* component for each product, and we are returned a new array of *ProductCard* components which we assign to *listProducts*.

Note that we have provided *productName* as *key* attribute for our list items. Remember that "key" is a special string attribute which help React Native identify which items have changed, are added or are removed. Because *productName* might not be unique, I will leave it to you as an exercise on how you can use *Product id* which uniquely identifies a product to be the key instead.

Lastly in *App.js*, we render our *ProductList* component and also populate *<Header>* with a title with the codes in **bold**:

```
import React from 'react';
import { Container, Header, Content, Body, Title } from 'native-base';
import ProductList from './ProductList';

export default function App() {


  return (
    <Container>
      <Header>
        <Body>

          <Title>List of Products</Title>
        </Body>
```

```
        </Header>
        <Content>
          <ProductList />
        </Content>
      </Container>
    );
}
```

Save all your files and you should have your application running like in figure 4.4.


figure 4.4

## Summary

In this chapter, we illustrate how to modify styles taken from NativeBase and reusing components to put them together in our example Product Listing application.

Contact me at support@i-ducate.com if you encounter any issues or visit my GitHub repository at *https://github.com/greglim81/react-native-chp4* for the full source code of this chapter.

# Chapter 5: Conditional Rendering

In this chapter, we will explore functionality to give us more control in rendering JSX.

## Inline If with && Operator

Suppose you want to show or hide part of a view depending on some condition. For example, we have earlier displayed our list of products. But if there are no products to display, we want to display a message like “No products to display”. To do so, in *ProductList.js* of the existing project from chapter four, add the codes in **bold**:

```
  import { Content, H1 } from 'native-base';
…
…

…
  return (
        <Content>
            {listProducts.length > 0 &&
                <Content>{listProducts}</Content>
            }
            {listProducts.length == 0 &&
                <H1>No Products to display</H1>
            }
        </Content>
    );
```

Now when we rerun our app, we should see the products displayed as same as before. But if we comment out our hard-coded data in *ProductList.js* and return an empty array instead, we should get the following message (fig 5.1):

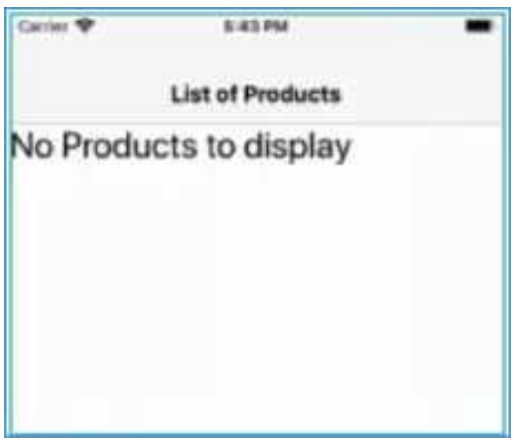



Figure 5.1

*Code Explanation*

```
{listProducts.length > 0 &&
    <Content>{listProducts}</Content>
}
```

Remember that we can embed any expression in JSX by wrapping them in curly braces. Thus, we can use the Javascript logical && operator to conditionally show *listProducts* if *listProducts.length > 0*. If the condition is true, i.e. *listProducts.length > 0* is true, the element right after && which is *<ul>{listProducts}</ul>* will appear in the output. If it is false, React Native will ignore and skip it.

The following expression however evaluates to false and therefore, we don't display the message.

```
{listProducts.length == 0 &&
    <H1>No Products to display</H1>
}
```

When we return an empty array however, "*listProducts.length > 0*" evaluates to false and we do not render the list of products. Instead we display the "No products to display message".

## Inline If-Else with Conditional Operator

The above code can also be implemented with if/else by using the Javascript conditional operator *condition ? true : false*. We have actually previously used this to conditionally render either a filled star or empty one.

```
return (
  <Content>
      {listProducts.length > 0 ? (
          <Content>{listProducts}</Content>
    ) : (
      <H1>No Products to display</H1>
    )}
  </Content>
    );
```

*Code Explanation*

{listProducts.length > 0 ? (

```
        <Content>{listProducts}</Content>
    ) : (
      <H1>No Products to display</H1>
    )}
```

The above code is saying, "If *listProducts* length is > 0, then show <Content> {listProducts}</ Content> . Otherwise (else) show what follows ':' which is <H1>No Products to display</H1> .

## *props.children*

Sometimes, we need to insert content into our component from the outside. For example, we want to implement a component that wraps a NativeBase Card component (fig. 5.2).

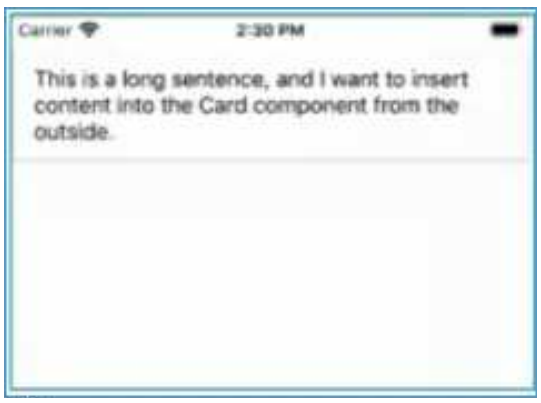



fig. 5.2

Here is an implementation of the Card class-based component.

```
import React, { Component } from 'react';
import { Container, Header, Content, Card, CardItem, Body, Text } from 'native-base';
export default class CardExample extends Component {
  render() {
    return (
      <Container>
        <Header />
        <Content>
          <Card>
            <CardItem>
              <Body>
                <Text>
                  //Your text here
                </Text>
              </Body>
            </CardItem>
          </Card>
        </Content>
      </Container>
    );
```

```
  }
}
```

The markup above can be obtained from:
*https://docs.nativebase.io/Components.html#card-def-headref*

To supply content to the jumbotron component, we can use *attributes* like:
< CardExample **props='…'/>**

This is not ideal however. For we probably want to write a lengthier markup here like,

```
<CardExample>
        This is a long sentence, and I want to insert content into the
        jumbotron component from the outside.
</CardExample>
```

That is to say, we want to insert content into the Card component from the **outside**. To do so, we use *props.children* as shown in the following. Create a new file *MyCard.js* with the below code:

```
import React from 'react';
import { Content, Card, CardItem, Body, Text } from 'native-base';

export default function MyCard(props) {


  return (
    <Content>
        <Card>
            <CardItem>
            <Body>
                <Text>

                {props.children}
                </Text>
            </Body>
            </CardItem>
        </Card>
    </Content>
  );
}
```

If there is a string in between an opening and closing tag, the string is passed as a special prop: *props.children*. So, in the code above, *props.children* will be the string between *<MyCard>* and *</MyCard>* as shown in **bold** below:

```
import React from 'react';
import { Container } from 'native-base';
```

```
import MyCard from './MyCard';

export default function App() {


  return (
    <Container>
      <MyCard>
        This is a long sentence, and I want to insert content into the
        Card component from the outside.
      </MyCard>
    </Container>
  );
}
```

When you run your app, the string will be displayed in the MyCard component nicely (fig. 5.3):

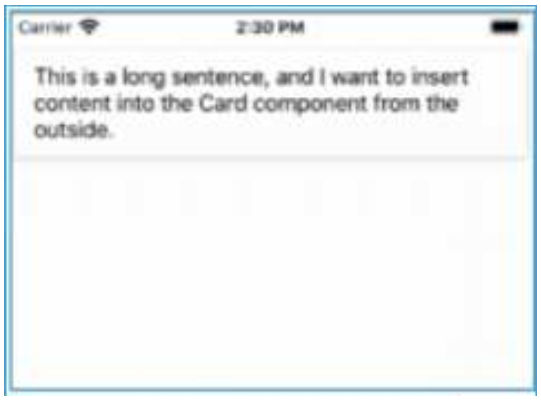



Figure 5.3

*Summary*

In this chapter, we introduced the inline if '&&' operator that gives us more conditional control in rendering our JSX. We have also learned about inserting content into components from the outside using *props.children*.

Contact me at support@i-ducate.com if you encounter any issues or visit my GitHub repository at *https://github.com/greglim81/react-native-chp6* for the source code of *Product.js* and *JumbotronComponent.js*.

# Chapter 6: Building React Native Forms with Hooks

In this chapter, we look at how to implement forms with validation logic with Hooks. As an example, we will implement a login form that takes in fields *email* and *password*.

## Create an Initial JSX Form Template

First, either in a new React Native project or in your existing project from chapter 5, create a new file *MyForm.js* and copy-paste the form template from NativeBase (*https://docs.nativebase.io/Components.html#Form*) into it. It will look like below:

```
import React from 'react';
import { Content, Form, Item, Input } from 'native-base';

export default function MyForm() {
  return (
    <Content>
        <Form>
            <Item>
                <Input placeholder="Enter Email" />
            </Item>
            <Item last>
                <Input placeholder="Enter Password" />
            </Item>
        </Form>
    </Content>
  );
}
```

## And in *App.js*, import and render *MyForm*:

```
import React from 'react';
import { Container } from 'native-base';
import MyForm from './MyForm';

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

You should get a form displayed like in fig. 6.1

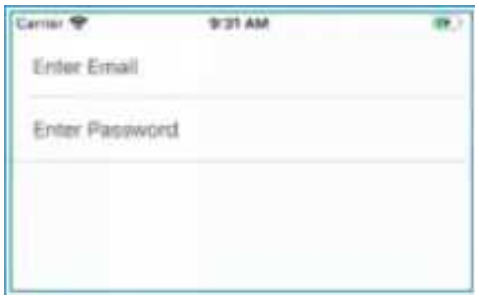



Figure 6.1

# Handling Inputs with *useState*

In most React Native apps, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. And this is done by storing these user-entered data in the component state with *useState*. First, we import *useState* with:

```
import React, {useState} from 'react';
```

Next, add in the below two lines in **bold**:

```
export default function MyForm() {

    const [email, setEmail] = useState("")
    const [password, setPassword] = useState("")

     return (
          …
```

That is, we declare two state variables *email* and *password* along with their state setter methods *setEmail* and *setPassword*. We set their initial value to an empty string “”.

We then handle the *onChangeText* event that is triggered whenever a field value is changed, i.e. the user types into the form field as shown below:

```
<Form>
    <Item>
        <Input
                placeholder="Enter Email"
                onChangeText={text => setEmail(text)}
                />
    </Item>
            …
```

So each time the user types into the form field, *onChangeText* of that field is called and we then call the associated handler function onChangeText={**text => setEmail(text)**} to set what's typed into the field to *email* in state.

We do the same to the password field:

```
<Item last>
    <Input
            placeholder="Enter Password"
            onChangeText={text => setPassword(text)}
            />
</Item>
```

When we do this, we immediately capture what's entered into the form by the user. We can illustrate this by displaying the values below the form by adding:

```
import { Content, Form, Item, Input, Text } from 'native-base';
…
…
…
        </Form>
        <Text>
            Email entered: {email}
        </Text>
        <Text>
            Password entered: {password}
        </Text>
    </Content>
        …
```

When we run the app, we should see the values displayed in the bottom of the form like in fig. 6.2:


Figure 6.2

# Showing Specific Validation Errors

Next, we want to add specific validation errors depending on the input given, for example “Email is required”, or “Email should be a minimum of six characters” and show corresponding validation error alerts when a user submits the form.

To show specific validation errors, we declare two more state variables to store our email and password error messages:

```
export default function MyForm() {

  const [email, setEmail] = useState("")
  const [password, setPassword] = useState("")
  const [emailError, setEmailError] = useState("")
  const [passwordError, setPasswordError] = useState("")

   return (
          …
```

To handle form submission, we define a *handleSubmit* event. We then bind *handleSubmit* to the *onPress* event handler in a *Button*:

```
…
import { Content, Form, Item, Input, Text, Button } from 'native-base';

export default function MyForm() {

  const [email, setEmail] = useState("")
  const [password, setPassword] = useState("")
  const [emailError, setEmailError] = useState("")
  const [passwordError, setPasswordError] = useState("")

  const handleSubmit = () => {
  }

  return (
    <Content>
      <Form>
                  …
                  …
                  …
          <Button onPress={handleSubmit}>
              <Text>Submit</Text>
          </Button>
      </Form>
              …
```

When the form is submitted through the clicking of the button, *handleSubmit*

will be called.

In a normal app, we will want to send the form to some external API e.g. login. But before we send the network request, in *handleSubmit*, we want to perform some client-side validation. For example, if *username* length is zero, if its less then a minimum length, if there are spaces in between etc. We first illustrate this for the *email* field by adding the below:

```
const handleSubmit = () => {
  var emailValid = false;
  if(email.length == 0){
      setEmailError("Email is required");
  }
  else if(email.length < 6){
      setEmailError("Email should be minimum 6 characters");
  }
  else if(email.indexOf(' ') >= 0){
      setEmailError('Email cannot contain spaces');
  }

  else{
      setEmailError("")
      emailValid = true
    }

  if(emailValid){
      alert('Email: ' + email + '\nPassword: ' + password);
  }
}
```

That is, for each *if*-clause, we check for a specific validation, and if so, assign the specific error message to *emailError* with *setEmailError*. Only when it manages to reach the last *else* clause that we know we have no email validation errors and we set the boolean *emailValid* to true.

And if *emailValid* is true, we then show an alert with what has been entered into the form.

## *Running your App*

Now when you run your app, fill in a valid email and password and click on submit, an alert box appears with the inputted values (fig. 6.3).


Figure 6.3

In a normal app, instead of showing an alert, we will usually send the data in a request to some API. We will illustrate this in a later chapter.

# Showing Validation Error Messages

If we enter an invalid email address, our form doesn't submit because of the validation checks we have added. But we should be showing validation errors to the user for her to correct her input as well. We will do that in this section.

First, import the *Badge* component from NativeBase *(https://docs.nativebase.io/Components.html#Badge)*:

```
import { Content, Form, Item, Input, Button, Text, Badge } from 'native-base';
```

*Badges* provide notification messages for user actions. We use *Badge* with the *danger* property to add a red background color to highlight that it is an error notification.

We then add the *Badge* below our input field:

```
return(
  <Content>
    <Form>
      <Item>
        <Input
          placeholder="Enter Email"
          onChangeText={text => setEmail(text)}
          />
      </Item>
      {emailError.length > 0 &&
```

```
<Badge danger>
        <Text>{emailError}</Text>
</Badge>}
        …
        …
        …
```

We wrap a condition around <*Badge*>. We show the error message only if the error message's length is more than 0, indicating that there is an error.

Remember that the syntax '&&' is saying, if *emailError.length > 0*, then show the *Badge* component.

If we run our app now, and submit without filling up the email field, we get (fig. 6.4):

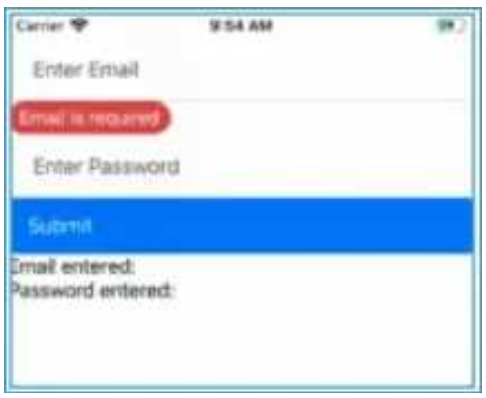



Figure 6.4

Or if we enter an email that is less than 6 characters, we get (fig. 6.5):

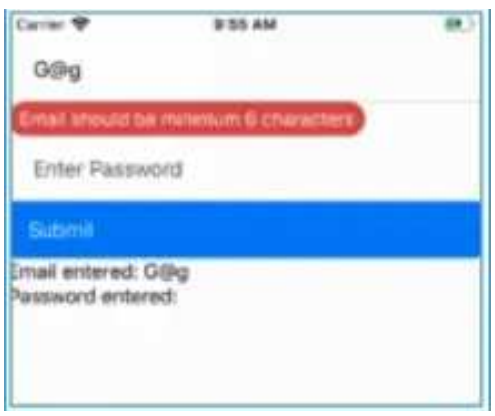



Figure 6.5

So, you see how you can extend your application with more validation checks?

## *Implementing Custom Validation Exercise*

In the previous example, we illustrated how to add custom validation

capabilities to our application's form to fulfill our needs i.e. a valid email cannot be less than 6 characters.

As an exercise, can you try checking for specific validation cases for *password* and showing the custom error message on your own? If you meet with any errors, just refer to the source code at the end of the chapter.

## Clearing the Fields Upon Successful Submit

Currently, after submitting our forms, the user-entered values in the fields remain. They should be cleared out after submission. To do so, in *handleSubmit*, after successful validation, we clear the *email* and *password* state:

```
const handleSubmit = () => {
        …
    …

    if(emailValid && passwordValid){
        alert('Email: ' + email + '\nPassword: ' + password);
        setEmail("");
        setPassword("");
    }
}
```

This still however doesn't clear our fields. We have to actually bind these state values to our fields' values:

```
<Item>
    <Input
        placeholder="Enter Email"
        onChangeText={text => setEmail(text)}
        value={email}
        />
</Item>
 …
    <Item last>
  <Input placeholder="Enter Password"
        onChangeText={text => setPassword(text)}
        value={password}
        />
</Item>
```

Now, when you run your app again and submit the form, because the field values are binded to the state's, the fields will be cleared out.

# Exercise

Note that we have completed our login form, try to come up with your own form and have additional inputs like text-areas, check boxes, radio buttons and more. The markup for them is available at *https://docs.nativebase.io/Components.html#Components*. They all work similar to the input fields that we have gone through.

## *Complete Code*

Below lists the complete code for *MyForm.js* which is also available in my React GitHub repository (https://github.com/greglim81/react-native-chp6).

```
import React, {useState} from 'react';
import { Content, Form, Item, Input, Button, Text, Badge } from 'native-base';

export default function MyForm() {

  const [email, setEmail] = useState("")
  const [password, setPassword] = useState("")
  const [emailError, setEmailError] = useState("")
  const [passwordError, setPasswordError] = useState("")

  const handleSubmit = () => {
    var emailValid = false;
    if(email.length == 0){
        setEmailError("Email is required");
    }
    else if(email.length < 6){
        setEmailError("Email should be minimum 6 characters");
    }
    else if(email.indexOf(' ') >= 0){
        setEmailError('Email cannot contain spaces');
    }
    else{
        setEmailError("")
        emailValid = true
    }

    var passwordValid = false;
    if(password.length == 0){
        setPasswordError("Password is required");
    }
    else if(password.length < 6){
        setPasswordError("Password should be minimum 6 characters");
    }
    else if(password.indexOf(' ') >= 0){
        setPasswordError('Password cannot contain spaces');
```

```
    }
    else{
        setPasswordError("")
        passwordValid = true
    }

    if(emailValid && passwordValid){

        alert('Email: ' + email + '\nPassword: ' + password);
        setEmail("");
        setPassword("");
    }

  }

  return (
    <Content>
        <Form>
            <Item>
                <Input placeholder="Enter Email" onChangeText={text => setEmail(text)} value=
{email}/>
            </Item>
            {emailError.length > 0 &&
            <Badge danger>
                <Text>{emailError}</Text>
            </Badge>}
            <Item last>
                <Input placeholder="Enter Password" onChangeText={text => setPassword(text)}
value={password}/>
            </Item>
            {passwordError.length > 0 &&
            <Badge danger>
                <Text>{passwordError}</Text>
            </Badge>}
            <Item>
                <Text></Text>
            </Item>
            <Button onPress={handleSubmit}>
                <Text>Submit</Text>
            </Button>
        </Form>

        <Text>
            Email entered: {email}
        </Text>
        <Text>
            Password entered: {password}
        </Text>

    </Content>
  );
}
```

## *App.js*

```
import React from 'react';
import { Container } from 'native-base';
import MyForm from './MyForm';

export default function App() {


  return (
    <Container>
      <MyForm />
    </Container>
  );
}
```

## *Summary*

In this chapter, we learned how to create a React Native form with Hooks. We created an initial JSX form with template from NativeBase. We then learned how to use useState with onChangeText and onPress methods to handle form inputs, show form specific form field validation errors and how to validate the form upon submit.

After submitting a form, we need to persist the data by calling the server's API endpoint. We will explore that in chapter ten. But first, in the next chapter, we will explore how to communicate with the server.

Visit my GitHub repository at *https://github.com/greglim81/react-hooks-chapter6* if you have not already have the full source code for this chapter.

# Chapter 7: Getting Data From RESTful APIs with Hooks

In this chapter, we will see how to call backend services to get data through RESTful APIs with the Axios library.

## GitHub RESTful API

Building RESTful APIs is beyond the scope of React Native it requires server-side technology like NodeJS, ASP.NET, Ruby on Rails and so on.

We will illustrate by connecting to the GitHub RESTful API to retrieve and manage GitHub content. You can know more about the GitHub API at

~~https://developer.github.com/v3/~~

But as a quick introduction, we can get GitHub users data with the following URL,

https://api.github.com/search/users?q=<search term>

We simply specify our search term in the URL to get GitHub data for user with name matching our search term. An example is shown below with search term *greg* .

https://api.github.com/search/users?q=greg

When we make a call to this URL, we will get the following json objects as a result (fig. 7.1.1).

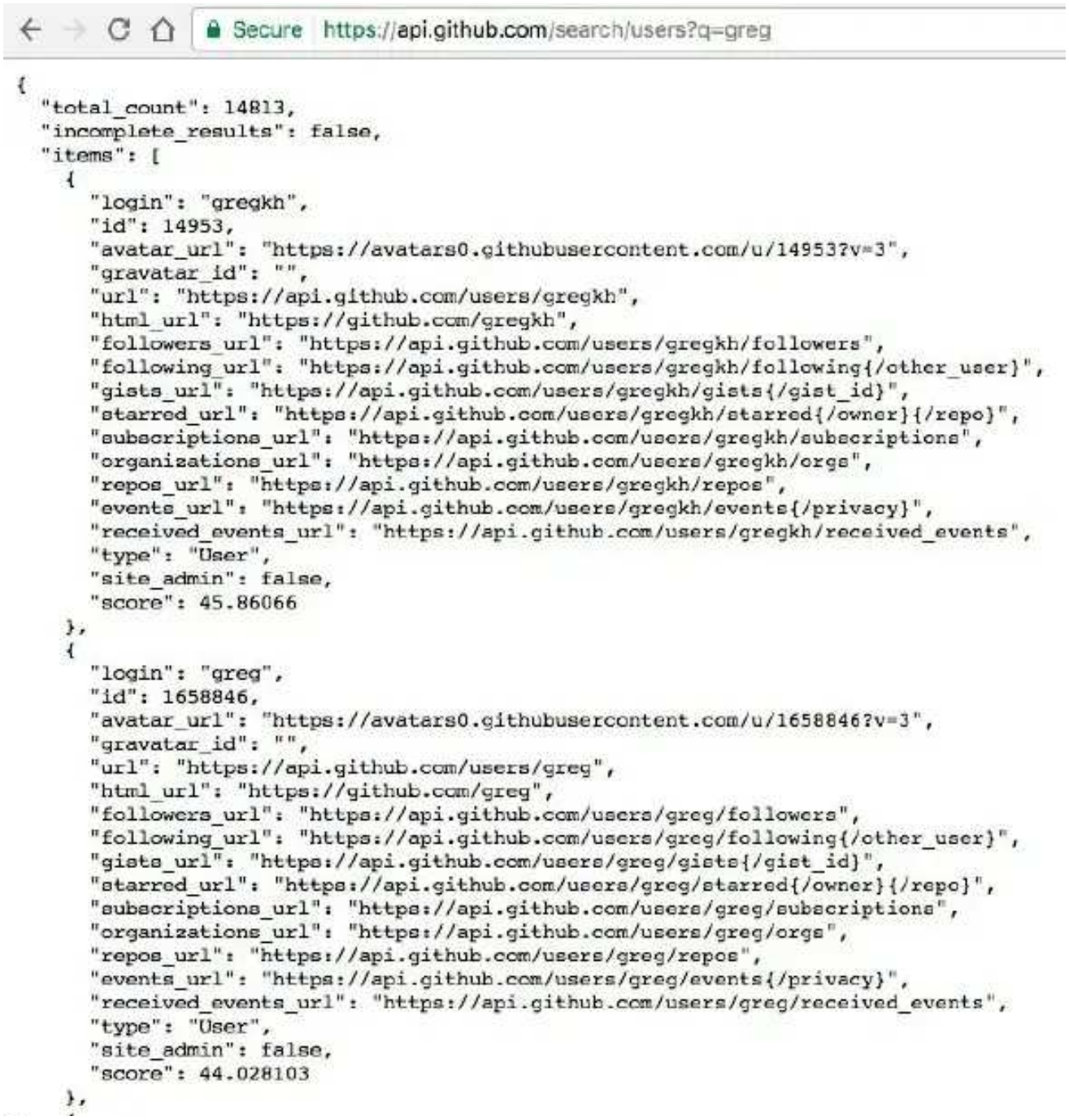


fig. 7.1

# Getting Data

To get data using a RESTful API, we are going to use the Axios library. Axios is a promise-based http client for the browser and NodeJS. We use it to make ajax calls to the server.

Axios provides the *get()* method for getting a resource, *post()* for creating it, *patch()* for updating it, *delete()* for delete and *head()* for getting metadata regarding a resource. We will illustrate using Axios to get data from a RESTful API in the following code example. In chapter nine, we will illustrate using axios for post, patch and delete as well.

To begin, either create a new React Native project or in your existing project from chapter 6, create a new file *GitHub.js* with the below code:

```
import React, { useEffect } from 'react';
import axios from 'axios'; // npm install axios
import { Content } from 'native-base';

function GitHub() {
    useEffect(() =>{
        axios.get("https://api.github.com/search/users?q=greg")
            .then(res => {
                console.log(res.data.items);
            });
    },[])

    return (
        <Content>
        </Content>
    );
}
export default GitHub;
```

*Code Explanation*

The code in *useEffect* will return GitHub data from its API endpoint. We will later explain what's *useEffect* and the usage of it. But we first dwell into the code inside *useEffect*.

To call our API endpoint, we need to use the *axios* library. First, install *axios* by executing the following in Terminal:

```
npm install axios
```

Then in *GitHub.js*, import it using

```
import axios from 'axios';
```

In *axios.get*, we call the GitHub API with argument 'greg'. *axios.get* returns a Promise which we need to subscribe to.

```
axios.get("https://api.github.com/search/users?q=greg")
    .then(res => {
        console.log(res.data.items);
    });
```

Note: If you are unfamiliar with promises, a promise allows us to make sense

out of asynchronous behavior. Promises provide handlers with an asynchronous action's eventual success value. Initially, the promise is pending, and then it can either be fulfilled with a value or be rejected with an error reason. When either of these options happens, the associated handlers queued up by a promise's then method are called. This lets asynchronous methods return values like synchronous methods instead of immediately returning the final value. The asynchronous method returns a promise to supply the value at some point in the future.

In *useEffect*, we use the *get()* method of *axios* and give the url of our API endpoint. We have a search term provided by the user from an input which we will implement later. The return type of *get()* is a promise. We subscribe to this promise with *then* so that when an ajax call is completed, the response is fed to the Promise and then pushed to the component.

We then pass in our callback function *res => console.log(res.data.items)*. Note that we have to access *data.items* property to get the *items* array direct as that is the json structure of the GitHub response. So when our ajax call is completed, we print the list of items returned which is the GitHub users search results.

## *useEffect*

Now, we come to an important question. What's *useEffect*? And why do we place our data request and retrieval code in it? As stated in reactjs.org, "If you're familiar with React class lifecycle methods, you can think of *useEffect* Hook as *componentDidMount*, *componentDidUpdate*, and *componentWillUnmount* combined."

Or if you are not familiar with React class component lifecycle methods, *useEffect* is called after our component renders. For e.g. when our GitHub component first renders, we want to make the API data request. Thus, we place our data retrieval code in it. We will dwell more into *useEffect*, but for now, let's see the results we get when we run our app.

## *Running our App*

Before we run our app, remember that we have to import and call our GitHub component in *App.js*.

```
import React from 'react';
```

```
import { Container } from 'native-base';
import GitHub from './GitHub';

export default function App() {


  return (
    <Container>
      <GitHub />
    </Container>
  );
}
```

Now run your app. In your Terminal, you can see the following result logged:

```
Array [
  Object {
    "avatar_url": "https://avatars3.githubusercontent.com/u/1658846?v=4",
    "events_url": "https://api.github.com/users/greg/events{/privacy}",
    "followers_url": "https://api.github.com/users/greg/followers",
    "following_url": "https://api.github.com/users/greg/following{/other_user}",
    "gists_url": "https://api.github.com/users/greg/gists{/gist_id}",
    "gravatar_id": "",
    "html_url": "https://github.com/greg",
          …
  },
  Object {
    "avatar_url": "https://avatars3.githubusercontent.com/u/14953?v=4",
    "events_url": "https://api.github.com/users/gregkh/events{/privacy}",
    "followers_url": "https://api.github.com/users/gregkh/followers",
    "following_url": "https://api.github.com/users/gregkh/following{/other_user}",
    "gists_url": "https://api.github.com/users/gregkh/gists{/gist_id}",
    "gravatar_id": "",
    "html_url": "https://github.com/gregkh",

      …
  },
…
…
…
```

Our requested json object is a single object containing an items array of size 30 with each item representing the data of a GitHub user. Each *user* object has properties *avatar_url*, *html_url*, *login*, *score*, and so on.

# Storing Results in State

Now that we have made a successful connection to our API, let ' s have a state variable to store our results instead of just logging them to the console.

This will let us be able to display the results to the user. In GitHub.js, add in the following code in **bold**:

```
import React, { useEffect, useState } from 'react';
import axios from 'axios'; // npm install axios

function GitHub() {

    const [data, setData] = useState([]);

    useEffect(() =>{
        axios.get("https://api.github.com/search/users?q=greg")
            .then(res => {
                //console.log(res.data.items);
                setData(res.data.items)
            });
    },[])


    return (
        <div>
        </div>
    );
}
export default GitHub;
```

## *Code Explanation*

We first import the *useState* function:

```
import React, { useEffect, useState } from 'react';
```

We then declare the state variable *data*: `const [data, setData] = useState([]);`
Here, *data* is set to an initial value of an empty array, `[]`. So, you see, a variable in state can be set to any type, String, Array, Boolean, Integer, Object etc.

```
axios.get("https://api.github.com/search/users?q=greg")
    .then(res => {
        //console.log(res.data.items);
        setData(res.data.items)
    });
```

And in the callback function, instead of logging to the console, we use the state setter *setData* to assign the results to *data*.

# Back to *useEffect*

What's the empty array [ ] in the 2nd argument of *useEffect*?

```
useEffect(() =>{
    axios.get("https://api.github.com/search/users?q=greg")
        .then(res => {
            setData(res.data.items)
        });
},[])
```

If we remove the second argument and then re-run our app, what happens?

```
useEffect(() =>{
    axios.get("https://api.github.com/search/users?q=greg")
        .then(res => {
            setData(res.data.items)
        });
});
```

When *useEffect* does not have a second argument, it will be called each time *setData* is called, i.e. each time a state is changed in its body. And this results in a serious consequence because we get an infinite loop! In *useEffect*, we have *setData*, and this causes *useEffect* to be called again which calls *setData* again and this repeats in a never-ending network request. So, if you try to run this, (you don't have too…), you will get multiple requests sent to GitHub until they detect a problem and block us! And we certainly want to avoid this situation. So, let's put back the empty array in *useEffect*.

```
useEffect(() =>{
    axios.get("https://api.github.com/search/users?q=greg")
        .then(res => {
            setData(res.data.items)
        });
},[])
```

With the empty array in the second argument, *useEffect* is called only once when the component first renders.

In essence, without the second argument, *useEffect* is run on every render (initial render and update renders) of the component. If the 2nd argument is an empty array, *useEffect* is only called once, after the component renders for the first time.

But what if we want *useEffect* to be correctly called when the state changes? For e.g. currently out URL is hardcoded to 'greg'. What if we want *useEffect*

to be called when we have a new search term?

Let's add a state property '*searchTerm*' to store a search term entered by the user. Add in the following codes in **bold**:

```
function GitHub() {
  const [data, setData] = useState([]);
  const [searchTerm, setSearchTerm] = useState("greg");
```

We would then append *searchTerm* to our url:

```
    useEffect(() =>{
        axios.get(`https://api.github.com/search/users?q=${searchTerm}`)
            .then(res => {
                console.log(res.data.items);
                setData(res.data.items)
            });
        },[searchTerm]);
```

Note that the URL above has to be enclosed in backticks ` ` in order to append *searchTerm*.

And lastly, we specify *searchTerm* in the array of the second argument. With this, we are saying that *useEffect* should be called when the component first renders, and also each time *searchTerm* changes. In essence, if the 2nd argument contains an array of variables, if any of these variables change, *useEffect* will be called. So, in this case, the hook is not only triggered when the component is first mounted, but when one of its dependencies in the 2nd argument array is updated.

To summarize:
*useEffect* without a second argument, is called each time a state change occurs in its body.
*useEffect* with an empty array in its second argument gets called only the first time the component renders.
*useEffect* with a state variable in the array gets called each time the state variable changes.

We will re-visit *useEffect* again later and as we re-visit it, concepts about it using will become more apparent. For now, we will set the 2nd argument back to an empty array:

```
    useEffect(() =>{
```

```
axios.get(`https://api.github.com/search/users?q=${searchTerm}`)
        .then(res => {
            console.log(res.data.items);
            setData(res.data.items)
        });
},[searchTerm])
```

# Showing a Loader Icon

While getting content from a server, it is often useful to show a loading icon to the user (fig. 7.4.1).

figure 7.5

To do so, in GitHub component, create a state variable called *isLoading* and set it to *true* like in the below code.

```
function GitHub() {

    const [data, setData] = useState([]);
    const [searchTerm, setSearchTerm] = useState("greg");
    const [isLoading, setIsLoading] = useState(true);
```

*isLoading* will be true when loading of results from the server is still going on. We set it to true in the beginning.

Next, in the *then()* method, set *isLoading* to false because at this point, we get the results from the server and loading is finished.

```
useEffect(() =>{
    axios.get(`https://api.github.com/search/users?q=${searchTerm}`)
            .then(res => {
                setData(res.data.items)
                setIsLoading(false);
            });
},[])
```

Lastly, in *return()*, add a *Content* that shows the loading icon. We use the *if* && conditional to make the *Content* visible only when the component is loading.

```
import { Content, H3 } from 'native-base';

…
```

```
…
…
    return (
        <Content>
            { isLoading &&

            }           <H3>Getting data...</H3>
        </Content>
    );
```

If you load your app, you should see the “Getting data” message being displayed for a short moment before data from the server is loaded.

We will now replace the “Getting data” message with a spinner icon. To get the spinner icon, go to *https://docs.nativebase.io/Components.html#Spinner* (fig. 7.6).


figure 7.6

To add the loading icon, replace the “*Getting data…*” message with the following code in bold:

```
import { Content, Spinner } from 'native-base';

…
…
…

        <Content>
            { isLoading && <Spinner /> }
        </Content>
```

We import and specify *<Spinner />* to render the spinner’s animation. You can try out other kinds of colors as in the documentation. When you run your app, the spinner should appear (fig. 7.7).

Figure 7.7

# Implementing a GitHub Results Display Page

We will now implement a page which displays our GitHub user data nicely like in figure 7.8.


figure 7.8

In *return()*, we use the *List Thumbnail* component from *https://docs.nativebase.io/Components.html#list-thumbnail-headref*.

We will slightly modify the markup and include it in our component as shown below **in bold**:

```
import React, { useEffect, useState  } from 'react';
import axios from 'axios'; // npm install axios
import { Content, Spinner, List, ListItem, Left, Thumbnail, Body, Text, Right, Button } from 'native-base';
import { Linking } from 'react-native';
```

```
function GitHub() {

    const [data, setData] = useState([]);
    const [searchTerm, setSearchTerm] = useState("greg");
    const [isLoading, setIsLoading] = useState(true);

    useEffect(() =>{
        axios.get(`https://api.github.com/search/users?q=${searchTerm}`)
            .then(res => {
                //console.log(res.data.items);
                setData(res.data.items);
                setIsLoading(false);
            });
    },[])

    const listUsers = data.map((user) =>
        <ListItem key={user.id} thumbnail>
            <Left>
                <Thumbnail square source={{ uri: user.avatar_url }} />
            </Left>
            <Body>
                <Text>Login: {user.login}</Text>
                <Text note numberOfLines={1}>Id: {user.id}</Text>
            </Body>
            <Right>
                <Button onPress={() => { Linking.openURL(user.html_url)}}
                        transparent>
                    <Text>View</Text>
                </Button>
            </Right>
        </ListItem>
    );

    return (
        <Content>
            { isLoading && <Spinner /> }
            <List>
                {listUsers}
            </List>
        </Content>
    );
}
export default GitHub;
```

## And then, make the below changes to *App.js*:

```
import React from 'react';
import { Container, Header, Text } from 'native-base';
import GitHub from './GitHub';

export default function App() {
```

```
  return (
    <Container>
      <Header>
        <Text>GitHub Users Results</Text>
      </Header>
      <GitHub />
    </Container>
  );
}
```

## *Code Explanation*

**const listUsers = data.map((user) => …**

We use *map* to repeat the *ListItem* object for each user data we get from GitHub.

We then add Javascript JSX expressions wrapped in {} inside the template, i.e. user's id, html_url, avatar_url and login.

```
<ListItem key={user.id} thumbnail>
    <Left>
        <Thumbnail square source={{ uri: user.avatar_url }} />
    </Left>
    <Body>
        <Text>Login: {user.login}</Text>
        <Text note numberOfLines={1}>Id: {user.id}</Text>
    </Body>
    <Right>
        <Button onPress={() => { Linking.openURL(user.html_url)}}
transparent>
            <Text>View</Text>
        </Button>

    </Right>
</ListItem>
```

Finally, we display our data by including *listUsers* in our return template as shown below:

```
return (
    <Content>
        { isLoading && <Spinner /> }
        <List>
            {listUsers}
        </List>
    </Content>
);
```

If you run your app now, you should get a similar page as shown below (fig. 7.9).


Figure 7.9

# Adding an Input to GitHub Results Display Page

We are currently hard coding *searchTerm* to '*greg*' in our request to GitHub. We will now make use of *searchTerm* state so that a user can type in her search terms and retrieve the relevant search results.

Previously, we discussed about appending *searchTerm* to our url:

```
useEffect(() =>{
    axios.get(`https://api.github.com/search/users?q=${searchTerm}`)
        .then(res => {
            setData(res.data.items);
            setIsLoading(false);
        });
    },[searchTerm])
```

And specify *searchTerm* in the array of the second argument so that *useEffect* is called each time *searchTerm* changes.

Now, this however introduces another issue. Each time a user types into the keyboard and adds a character, a request is sent to the GitHub API. For e.g. if I enter 'greg', at least four requests will be sent! And if there is a typo or

deletion, more requests will be sent. This floods the server with unnecessary requests, and we risk ourselves of getting blocked from the API provider. Another issue is that even before previous requests complete successfully, we send a new request, and this piles up! Get the picture?

To resolve this, two popular approaches are to use things like *debounce*, where we delay sending a request until the user has stopped typing for a pre-determined amount of time. Another approach is to have a form with an input field and submit button. In doing so, a request will only be made when the user clicks on submit. In this section, we will illustrate the form submit approach.

What we do first is to encapsulate our *axios* related code into a separate function which will be called both by *useEffect* and submit:

```
useEffect(() =>{
    getData();
},[])

const getData = () => {
    axios.get(`https://api.github.com/search/users?q=${searchTerm}`)
        .then(res => {
            setData(res.data.items)
            setIsLoading(false);
        });
}
```

And because no call to GitHub is made at the beginning now, we initialize *isLoading* to *false* at first as shown below:

```
const [searchTerm, setSearchTerm] = useState("");
 const [isLoading, setIsLoading] = useState(false);
```

Once the user submits the form, we set *isLoading* to *true* just before the call to *axios.get* to show the loading icon. We thus implement *handleSubmit* as:

```
const handleSearch = () => {
        setIsLoading(true);
        getData();
 }
```

Once we are notified of results from the GitHub request, we set *isLoading* to *false* in *getData* to hide the loading icon.

```
const getData = () => {
    axios.get(`https://api.github.com/search/users?q=${searchTerm}`)
```

```
            .then(res => {
                setData(res.data.items)
                setIsLoading(false);
            });
    }
```

Next, we want to add a Searchbar component (*https://docs.nativebase.io/Components.html#search-bar-headref*) for a user to type in her search term (fig. 7.10).

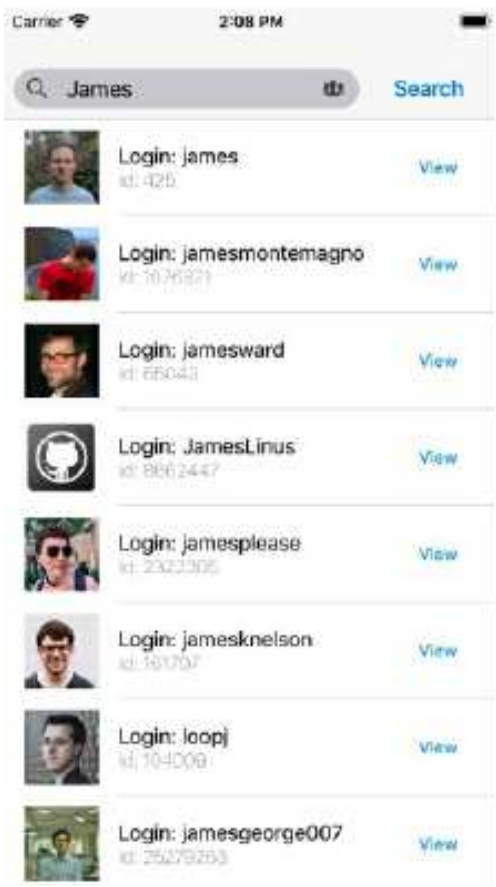



figure 7.10

Add in the codes as shown in **bold**:

```
    return (
        <Container>
            <Header searchBar rounded>
              <Item>
                <Icon name="ios-search" />
                <Input
                        placeholder="Search"
                        onChangeText={text => setSearchTerm(text)}
                        />
                <Icon name="ios-people" />
              </Item>
              <Button onPress={handleSearch} transparent>
                <Text>Search</Text>
              </Button>
```

```
            </Header>
            <Content>
                { isLoading && <Spinner /> }
                <List>
                    {listUsers}
                </List>
            </Content>
        </Container>
    );
```

Import the below from native-base:

import { Content, Spinner, List, ListItem, Left, Thumbnail, Body, Text, Right, Button, **Container, Header, Item, Icon, Input** } from 'native-base';

The input has its *onChangeText* event binded to the state's *searchTerm* property.

*App.js* is then reduced to:

```
import React from 'react';
import { Container } from 'native-base';
import GitHub from './GitHub';

export default function App() {


  return (
    <Container>
      <GitHub />
    </Container>
  );
}
```

*Running your App*

You can now see GitHub user results when the user types in a search term and click 'Search'.

# Refactoring *getData* to use *async*/*await*

*getData* is currently using a callback function. Let us refactor it to use *async*/*await* as shown below:

```
    const getData = async() => {
        const res = await axios.get(`https://api.github.com/search/users?q=${searchTerm}`);

        setData(res.data.items);
        setIsLoading(false);
    }
```

Once we use the *await* keyword, everything reads like synchronous code. Actions after the *await* keyword are not executed until the promise resolves, meaning the code will wait.

And if we use *await*, we have to add *async* to the function declaring that it is

making the request as an asynchronous function.

# Summary

In the chapter, we learned how to implement a GitHub User Search application by connecting our React Native app to the GitHub RESTful API using axios, Promises, useEffect and displaying a loader icon.

Visit my GitHub repository at *https://github.com/greglim81/react-native-chp7* if you have not already have the full source code for this chapter.

Full source code:

## *App.js*

```
import React from 'react';
import { Container } from 'native-base';
import GitHub from './GitHub';

export default function App() {


  return (
    <Container>
      <GitHub />
    </Container>
  );
}
```

## *GitHub.js*

```
import React, { useEffect, useState  } from 'react';
import axios from 'axios';
import { Content, Spinner, List, ListItem, Left, Thumbnail, Body, Text, Right, Button, Container,
Header, Item, Icon, Input } from 'native-base';
import { Linking } from 'react-native';

function GitHub() {

    const [data, setData] = useState([]);

    const [searchTerm, setSearchTerm] = useState("");
    const [isLoading, setIsLoading] = useState(false);
```

```
useEffect(() =>{
    getData();
},[])

const getData = () => {
    axios.get(`https://api.github.com/search/users?q=${searchTerm}`)
        .then(res => {
            setData(res.data.items)
            setIsLoading(false);
        });
}

const handleSearch = () => {
    setIsLoading(true);
    getData();
}

const listUsers = data.map((user) =>
    <ListItem key={user.id} thumbnail>
        <Left>
            <Thumbnail square source={{ uri: user.avatar_url }} />
        </Left>
        <Body>
            <Text>Login: {user.login}</Text>
            <Text note numberOfLines={1}>Id: {user.id}</Text>
        </Body>
        <Right>
            <Button onPress={() => { Linking.openURL(user.html_url)}} transparent>
                <Text>View</Text>
            </Button>
        </Right>
    </ListItem>
);


    return (
        <Container>
            <Header searchBar rounded>
            <Item>
                <Icon name="ios-search" />
                <Input placeholder="Search" onChangeText={text => setSearchTerm(text)}
/>
                <Icon name="ios-people" />
            </Item>
            <Button onPress={handleSearch} transparent>
                <Text>Search</Text>
            </Button>
            </Header>
            <Content>
                { isLoading && <Spinner /> }
                <List>
                    {listUsers}
```

```
                    </List>
                </Content>
            </Container>
        );
}
export default GitHub;
```

# Chapter 8: C.R.U.D. with Hooks

## Project Setup for our ToDo C.R.U.D. App

In this chapter, we will create a ToDo app using hooks to provide functionality to create, read, update and delete todos (fig. 8.1).

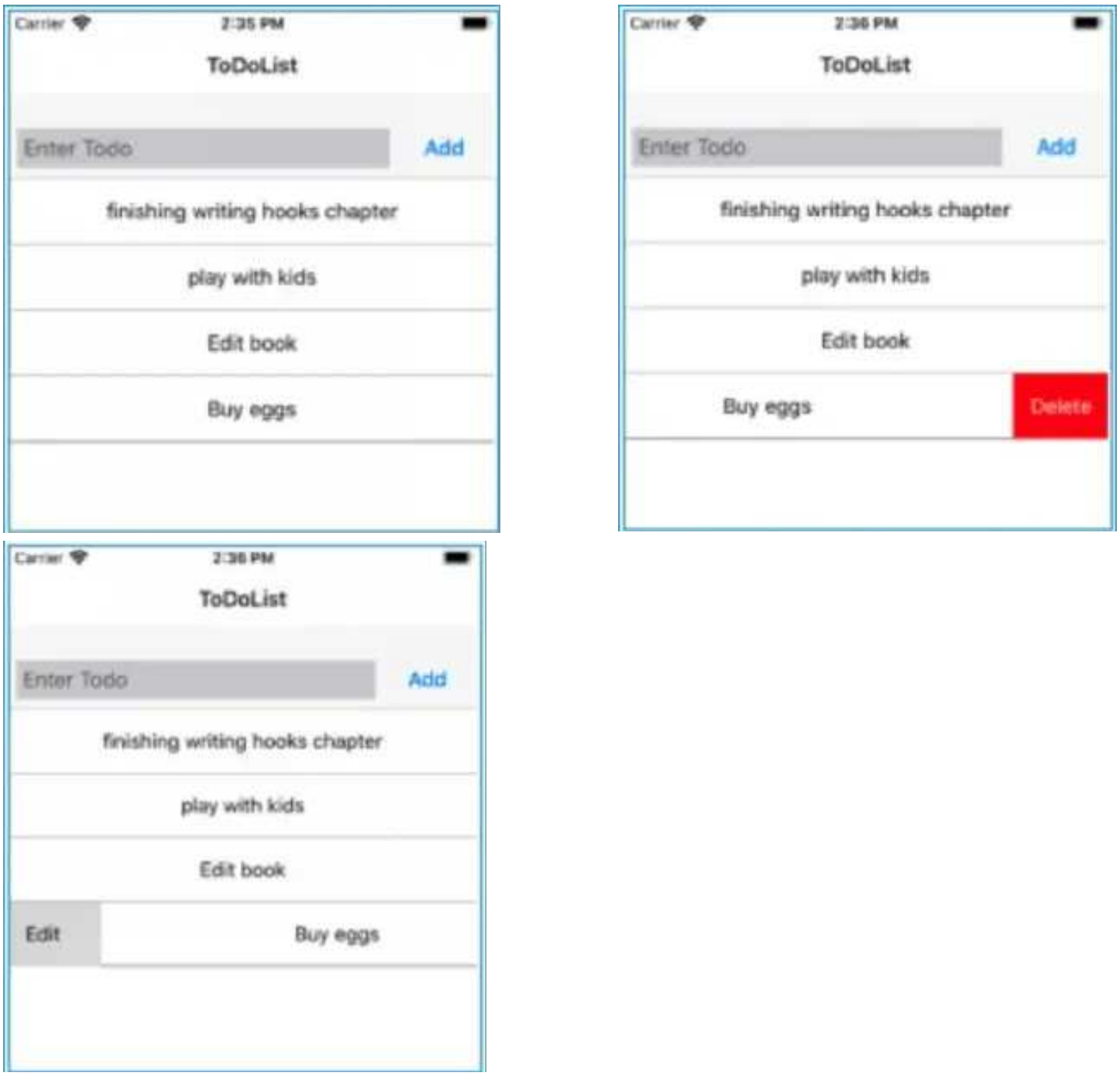



Figure 8.1

In the next chapter, we will then see how to connect our app to an external REST API to persist our data. This app will teach you fundamentals on how to build larger and more complicated apps, in particular managing global state with React Native apps.

To begin, you can either create a new project or work on the existing project from chapter seven.

Before we begin, let's install the *uuid-random* library to help us create unique id values for each of our todos. Run the command:

```
npm install uuid-random
```

# Avoiding Props Drilling with React Context and the *useContext* Hook

Now, how are we going to pass values between components in our app? In React Native apps, we usually pass values to components through props, as what we have been doing in this book.

For example, the logged-in username is often passed around since many components refer to it. Suppose we have the following *App.js*:

```
import React from 'react';
import { Container, Header, Text } from 'native-base';

export default function App() {

  const username = 'Greg'

  return (
    <Container>
        <ProductList username={username}/>
    </Container>
  );
}

function ProductList(props){
  return (
    <Header>
        <Product username={props.username}/>
    </Header>
  )
}

function Product(props){
  return (
        <Text>{props.username}</Text>
  )
}
```

From the *App* component, we pass *username* to the *ProductList* component with *props*, i.e.

<ProductList **username={username}**/>

And then in *ProductList*, we receive it in the *props* object:

```
function ProductList(props){
  return (
    <Header>
        <Product username={props.username}/>
    </Header>
  )
}
```

We repeat the process of passing props to child components (i.e. *Product*) further down:

```
function Product(props){
  return (
        <Text>{props.username}</Text>
  )
}
```

This process of passing data down props to the components that need them is a familiar one. It is frequently called *props drilling*. Props drilling can be quite a hassle when we need to get through a number of components to their destination, and especially when the components in the middle do not have any use of the props.

So, what's a good alternative to avoid props drilling, yet still be able to pass data around in our app?

One solution is to use React Context. React Context allows us to share values

with any components in the component tree that asks for those values.
For e.g. suppose we want to share *username* around in our app. In *App.js*, we create an instance of a context and name it *UserContext*:

```
import React from 'react';
import { Container, Header, Text } from 'native-base';

const UserContext = React.createContext()

export default function App() …
```

Next, with *UserContext*, we set up a *Provider* and put our *ProductList* component between the *Provider* tag:

...

```
const UserContext = React.createContext()

export default function App() {

  const username = 'Greg';

  return (
    <UserContext.Provider value={username}>
      <Container>
        <ProductList />
      </Container>
    </UserContext.Provider>
  );
}
```

We then use the *value* attribute to provide values to child components. In the above, we pass *username* into *value*.

How do we receive *username* in a child component? Now, *React.createContext()* returns an object that contains two values. The first one is *Provider* which we have just used to provide values to child components. The second one is *Consumer* which allows a child component to consume the value from Context, or in our instance, *UserContext*.

To consume *UserContext* from *Product*, add the below code in **bold**:

```
function Product(){
  return (
    <UserContext.Consumer>
      {value => <Text>Received, {value}</Text>}
    </UserContext.Consumer>
  )
}
```

We consume *value* passed down from *App.js* by providing a function between the *UserContext.Consumer* tags.

If you run your app now, you should see the message rendered on your screen, ‘Received, Greg’. And note that *username* doesn’t have to be passed through *ProductList*!

This is how we pass values down from parent components to child (and grandchild) components without having to use *props* thus avoiding *props drilling*. Components can consume the data that’s passed down through React

Context’s *Consumer*.

### *useContext Hook*

However, with the introduction of React hooks, there's a new way to consume context with a hook called *useContext*.

The code to provide a context in *App.js* remains the same. However, in *Product.js*, instead of using *UserContext.Consumer*, we have:

```
import React, {useContext} from 'react';
…
…
…
// no change needed in App or ProductList
function Product(){
  const value = useContext(UserContext)

  return (
    <Text>Received, {value}</Text>
  )
}
```

When we run the above, we get the exact same result. But note that the code looks cleaner. We don't have to put encompassing *UserContext* elements and functions between.

That's all to using React Context with the *useContext* hook to pass data around in our app. Using *Context* instead of passing down *props*, avoid us having to navigate confusing hierarchies. We just pass the context to the *useContext* hook and get back our values from it.

## Replacing Redux with the *useReducer* Hook

We have used Context to pass values around in our app. But how are we going to manage state in our app? We have shown earlier that we manage local component state using the *useState* hook. But in an app, how do we manage a global state across different components?

You might have heard of libraries like Redux to manage state across multiple components in an app. Now with the *useReducer* hook, we have available to us a lot of the functions of Redux. We can create *reducer* functions in order to manage state just like we did in Redux. Let's demonstrate with a basic counter example where we can increment, decrement or reset a count value.

First, in *App.js*, we have to import the *useReducer* hook:

```
import React, {useReducer} from 'react';
```

We then declare our initial state:

```
const initialState = {
  count: 0
}

function App (){

…
```

In *App.js*, we next implement a *reducer* function to implement increment, decrement and reset *count*:

```
function reducer(state, action){
  switch(action.type){
    case "increment":
      return {count : state.count + 1}
    case "decrement":
      return {count : state.count - 1}
    case "reset":
      return initialState
    default:
      return initialState
  }
}
```

*reducer* will take some state, and based on the action type, figure out what to do with our state. For example, if the type is ‘increment’, it will take the existing count in state, add one, and then return a new state with the incremented value. **We should emphasize that the existing state passed into the reducer is never modified or mutated. Rather, the reducer always returns a new state which replaces the old state.** In the following create, read, update, delete operations in our todo app, you will see that we always return a new state to replace the old state.

To use the reducer, add the following in *App.js*:

```
import React, { useReducer } from 'react';
import { Container, Button, Header, Text } from 'native-base';
…
const initialState = {
  count: 0
}
```

```
export default function App() {
  const [state, dispatch] = useReducer(reducer,initialState)
  return (
    <Container>
      <Header><Text>Count: {state.count}</Text></Header>
      <Button onPress={() => dispatch({type: 'increment'})}>
        <Text>Increment</Text>
      </Button>
      <Button success onPress={() => dispatch({type: 'decrement'})}>
        <Text>Decrement</Text>
      </Button>
      <Button warning onPress={() => dispatch({type: 'reset'})}>
        <Text>Reset</Text>
      </Button>
    </Container>
  );
}

function reducer(state, action){
        …
}
```

## *Code Explanation*

We first feed *reducer* and *initialState* into *useReducer*. *useReducer* then returns the current state, which we assign to *state*. It also returns *dispatch* that we use to dispatch increment, decrement and reset actions.

When we run our app, we can increment, decrement or reset our counter which will be rendered on to the view (fig. 8.2).


Figure 8.2

So all of this state management is available to us with the *useReducer* hook. If you are lost with the code at any point, here’s the counter app's full code:

## *App.js*

```
import React, { useReducer } from 'react';
import { Container, Button, Header, Text } from 'native-base';

const initialState = {
  count: 0
}

export default function App() {

  const [state, dispatch] = useReducer(reducer,initialState)

  return (
    <Container>
      <Header><Text>Count: {state.count}</Text></Header>
      <Button onPress={() => dispatch({type: 'increment'})}>
        <Text>Increment</Text>
      </Button>
      <Button success onPress={() => dispatch({type: 'decrement'})}>
        <Text>Decrement</Text>
      </Button>
      <Button warning onPress={() => dispatch({type: 'reset'})}>
        <Text>Reset</Text>
      </Button>
    </Container>

  );
}

function reducer(state, action){
  switch(action.type){
    case "increment":
      return {count : state.count + 1}
    case "decrement":
      return {count : state.count - 1}
    case "reset":
      return initialState
    default:
      return initialState
  }
}
```

# Combining *useContext* and *useReducer* to make Initial App State

With this understanding, we now combine *useContext* and *useReducer* to set up the initial state of our ToDo app.

In *App.js*, let's also remove everything we have previously done and define our initial *todos* state:

*App.js*

```
import React from 'react';
const todosInitialState = {
  todos:[{ id:1, text: "finishing writing hooks chapter"},
    { id:2, text: "play with kids"},
    { id:3, text: "read bible"}
  ]
};
```

*todosInitialState* contains our initial state which is an array *todos*, containing three *todo* objects.

We then define our *todosReducer* function in *App.js* which currently just has a default case:

```
function todosReducer(state, action){
  switch(action.type){
    default:
      return todosInitialState
  }
}
```

And in *App.js*, we then have

```
import React, { useReducer } from 'react';
…

export const TodosContext = React.createContext()

export default function App (){
  const [state, dispatch] = useReducer(todosReducer,todosInitialState)

  return (
      <TodosContext.Provider value={{state,dispatch}}>
        // add child components here
    </TodosContext.Provider>
  )
}
```

We make use of *TodosContext* to make *state* and *dispatch* available to child components.

# ToDo Swipeable List

We next create a *ToDoList* component to list out our todos. Besides listing todos, we also want to be able to swipe-delete and swipe-edit todos, a popular pattern we see in many apps (fig. 8.3).

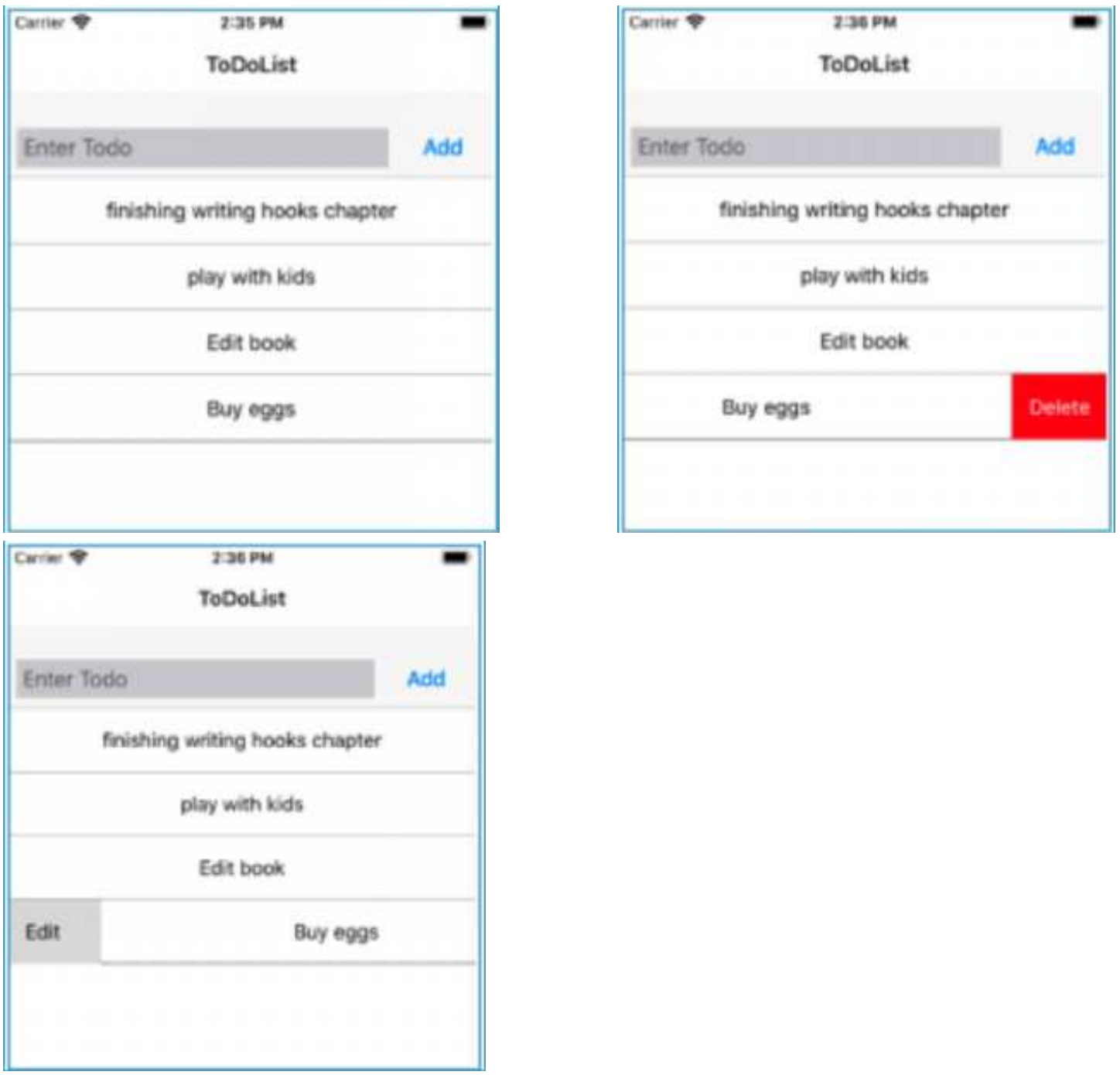



Figure 8.3

To implement a swipeable list, we use the *react-native-swipe-list-view* library (https://www.npmjs.com/package/react-native-swipe-list-view), which provides us a list view with rows that swipe open and closed.

In the Terminal, run the following to install *react-native-swipe-list-view*:

```
npm install --save react-native-swipe-list-view
```

Next, we create a new file *ToDoList.js* with a simplified version of the

example code from the *react-native-swipe-list-view* library. In *ToDoList.js*, fill in the following code:

```
import React, { useContext } from 'react'
import { TodosContext } from './App'
import { SwipeListView } from 'react-native-swipe-list-view';

import { Text, Container, Header } from 'native-base';
import { StyleSheet, View } from 'react-native';

export default function ToDoList() {
    // receive state and dispatch from App.js
    const {state, dispatch} = useContext(TodosContext);

    const renderItem = data => (
        <View style={styles.rowFront}>
            <Text>{data.item.text}</Text>
        </View>
    );

    return (
        <Container>
            <Header><Text>ToDoList</Text></Header>
            <SwipeListView
                data={state.todos}
                renderItem={renderItem}
            />
        </Container>
    );
}

const styles = StyleSheet.create({
    rowFront: {
        alignItems: 'center',
        backgroundColor: '#FFF',
        borderBottomWidth: 0.25,
        justifyContent: 'center',
        height: 50,
      }
});
```

## *Code Explanation*

```
    const {state, dispatch} = useContext(TodosContext);
```

Using *useContext(TodosContext)*, we receive *state* and *dispatch* from *App.js*.

```
    return (
        <Container>
            <Header><Text>ToDoList</Text></Header>
            <SwipeListView
                data={state.todos}
```

```
                renderItem={renderItem}
            />
        </Container>
    );
```

In *return*, we render the *SwipeListView* component from *react-native-swipe-list-view* . We pass in *state.todos* (which contain our array of todos) to the *data* property of *SwipeListView*. We also pass in a function to *renderItem* that determines how to render items in our list. The *renderItem* function is defined in:

```
const renderItem = data => (
    <View style={styles.rowFront}>
        <Text>{data.item.text}</Text>
    </View>
);
```

The *todo* object for a row is returned via *data.item* in the function. We render each row in a *View* with style *styles.rowFront*. You can imagine the Swipeable List as having two layers, the front and back layer. The front layer lists out the items. And when there's a swipe to the left/right, the back layer is exposed to show the Delete or Edit button. *rowFront* styles the front layer:

```
const styles = StyleSheet.create({
    rowFront: {
        alignItems: 'center',
        backgroundColor: '#FFF', // white color
        borderBottomWidth: 0.25,
        justifyContent: 'center',
        height: 50,
       }
});
```

The properties should be self-explanatory. We will later show the styling of the back layer in *rowBack*.

Next in *<Text>*, we render the todo's text with *data.item.text* since *data.item* refers to *todo*. Remember that *todo* has the *id* and *text* property.

## *ToDoList in App.js*

Having defined *ToDoList*, we import and add it to *App.js*:

```
import React, { useReducer } from 'react';
import ToDoList from './ToDoList';

const todosInitialState = {
```

```
  todos:[{ id:1, text: "finishing writing hooks chapter"},
    { id:2, text: "play with kids"},
    { id:3, text: "read bible"}
  ]
};

export const TodosContext = React.createContext()

export default function App() {

  const [state, dispatch] = useReducer(todosReducer,todosInitialState)

  return (
    <TodosContext.Provider value={{state,dispatch}}>
      <ToDoList />
    </TodosContext.Provider>
  )
}

function todosReducer(state, action){
  switch(action.type){
    default:
      return todosInitialState
  }
}
```

If you run your app now, you should see an initial list of *todos* (fig. 8.4).


Figure 8.4

# Adding the Swipe Delete and Edit Buttons to *ToDoList*

Next, we want to add the swipe delete and edit buttons to *ToDoList.js*. To do so, we specify the *renderHiddenItem* property in *SwipeListView* by adding the following:

```
import { StyleSheet, TouchableOpacity, View } from 'react-native';
…
```

```
export default function ToDoList() {

    const {state, dispatch} = useContext(TodosContext);

    const renderItem = data => (
        <View style={styles.rowFront}>
            <Text>{data.item.text}</Text>

        </View>
    );

    const renderHiddenItem = (data, rowMap) => (
        <View style={styles.rowBack}>
            <TouchableOpacity>
                <Text>Edit</Text>
            </TouchableOpacity>
            <TouchableOpacity
                style={[styles.backRightBtn]}
            >
                <Text style={{color: '#FFF'}}>Delete</Text>
            </TouchableOpacity>
        </View>
    );
    return (
        <Container>
            <SwipeListView
                data={state.todos}
                renderItem={renderItem}
                renderHiddenItem={renderHiddenItem}
                leftOpenValue={75}
                rightOpenValue={-75}
            />
        </Container>
    );
}

const styles = StyleSheet.create({
    rowFront: {
        alignItems: 'center',
        backgroundColor: '#FFF',
        borderBottomWidth: 0.25,
        justifyContent: 'center',
        height: 50,
      },
    rowBack: {
        alignItems: 'center',
        backgroundColor: '#DDD',
        flex: 1,
        flexDirection: 'row',

        justifyContent: 'space-between',
        paddingLeft: 15,
      },
```

```
    backRightBtn: {
        alignItems: 'center',
        bottom: 0,
        justifyContent: 'center',
        position: 'absolute',
        top: 0,
        width: 75,
        backgroundColor: 'red',
        right: 0
    }
});
```

## *Code Explanation*

```
<SwipeListView
    data={state.todos}
    renderItem={renderItem}
    renderHiddenItem={renderHiddenItem}
    leftOpenValue={75}
    rightOpenValue={-75}
/>
```

We specify a function to *renderHiddenItem* property which defines how we want to render hidden items. We determine how long we want to open the front row from the left with *leftOpenValue={75}*. This moves the front row to the right by 75 to expose ‘Edit’, currently in the left of the back row (fig. 8.5).

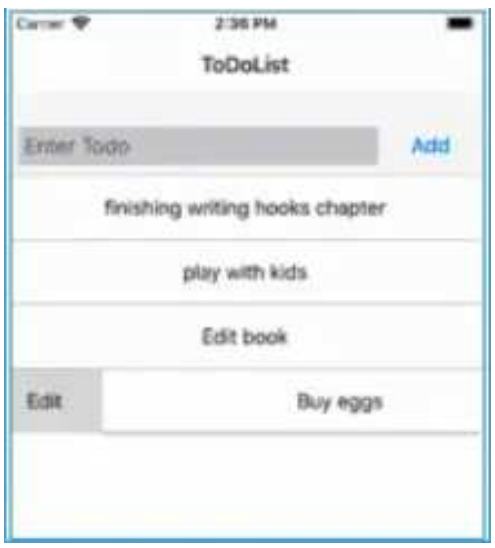



Figure 8.5

*rightOpenValue={-75}* moves the front row to left by 75. This will expose ‘Delete’ in the right of the back row (fig. 8.6).

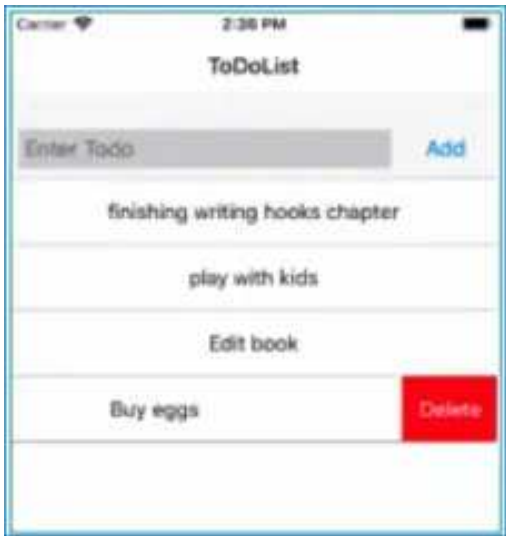



Figure 8.6

How do we position Delete on the right of the back row? We will come to that later. For now, we explain the *renderHiddenItem* function.

```
const renderHiddenItem = (data, rowMap) => (
    <View style={styles.rowBack}>
        <TouchableOpacity>
            <Text>Edit</Text>
        </TouchableOpacity>
        <TouchableOpacity
            style={[styles.backRightBtn]}
        >
            <Text style={{color: '#FFF'}}>Delete</Text>
        </TouchableOpacity>
    </View>
);
```

In *renderHiddenItem*, we define a style for the View which styles the back row (*styles.rowBack*).

```
rowBack: {
    alignItems: 'center',
    backgroundColor: '#DDD',
    flexDirection: 'row',
    justifyContent: 'space-between',
    paddingLeft: 15,
   }
```

To differentiate the back row and front row, we give the back row a color of #DDD (grey) while we have given the front row a color of #FFF (white). The back row View contains both the Edit and Delete button. By default, children of a component (in our case, Edit and Delete are children of the back row view) are aligned from top to bottom (*flexDirection: 'column'*). Because we want Edit and Delete to be aligned from left to right, we change the

*flexDirection* to *row*.

we provide a *paddingLeft* of 15 to provide the Edit button some padding from the left.

```
<TouchableOpacity
    style={[styles.backRightBtn]}
>
    <Text style={{color: '#FFF'}}>Delete</Text>
</TouchableOpacity>
```

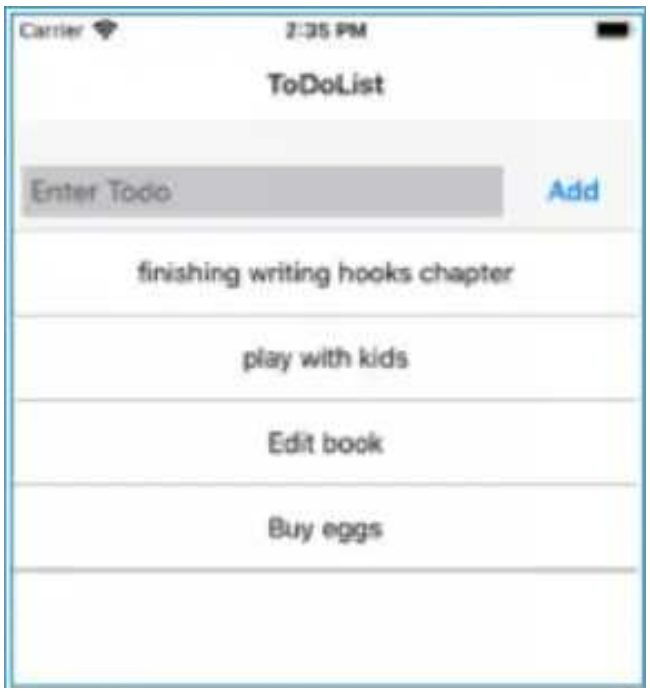
Next, we have to provide a styling for the Delete to make it positioned right of the back row. We also want its background color to be in red. We define its style in *styles.backRightBtn*:

```
backRightBtn: {
    alignItems: 'center',
    bottom: 0,
    justifyContent: 'center',
    position: 'absolute',
    top: 0,
    width: 75,
    backgroundColor: 'red',
    right: 0
}
```

To position it to the right, we specify *right: 0* to specify that we offset 0 pixels from the right edge. We also specify *bottom: 0* and *top:0* for the Delete button to fill the entire height of the row. The rest of the style properties should be self-explanatory.

Note that we wrap Edit and Delete text with *TouchableOpacity* so that it can respond to touches. We will later implement their touch handlers.

When we run our app now, our *todos* should be nicely displayed. And when we swipe left on a todo, ‘Delete’ should be exposed and when we swipe to the right, ‘Edit’ should be exposed (fig. 8.3):



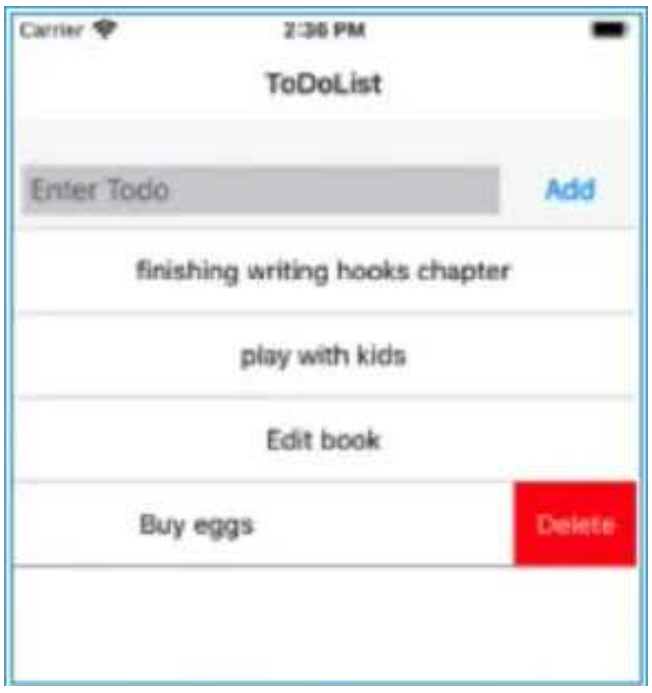





Figure 8.3

# Removing a Todo

Now, let’s see how we remove (delete) a todo. We want to remove a todo when a user swipes right and clicks on *Delete* (fig.). In the delete *TouchableOpacity*, add the *onPress* handler:

```
const renderHiddenItem = (data, rowMap) => (
    <View style={styles.rowBack}>
        <TouchableOpacity>
            <Text>Edit</Text>
        </TouchableOpacity>
        <TouchableOpacity
            style={[styles.backRightBtn]}
```

```
                onPress={() => deleteRow(data.item)}
            >
                <Text style={{color: '#FFF'}}>Delete</Text>
            </TouchableOpacity>
        </View>
    );
```

Also add in the *deleteRow* function:

```
const deleteRow = (todo) => {
    dispatch({type:'delete',payload:todo});
};
```

*deleteRow* uses *dispatch* to fire the *delete* action. We also provide the payload argument, *todo* for the reducer to perform the delete action.

We next need to add to *todosReducer* the *delete* case to handle the action. In *App.js*, add:

```
function todosReducer(state, action){
  switch(action.type){
    case 'delete':
      const filteredTodoState = state.todos.filter( todo => todo.id !== action.payload.id)
      return {...state, todos: filteredTodoState}
    default:
      return todosInitialState
  }
}
```

*state.todos.filter* checks for each element and filters for only *todos* whose id is not equal to the todo's id in the payload (the todo to be deleted). Note that *filter* returns a whole new array for us. It does not change the existing array

and then return it.
With the new filtered *todos*, we return a new state by spreading in all the current state properties with the spread operator '...' and having *todos* array now containing the new filtered *todos*. Or to put it in another manner, '… state' creates a copy of the existing state and we then assign its *todos* with *filteredTodoState*. This ensures that we return a new state, but yet having existing state properties unchanged and only *todos* changed. Remember that this is because the reducer should always return a new state to replace the old state rather than modifying the existing state.

If we run our app now, and click on 'Delete' for a todo, that todo will be removed.

# Adding Todos

To let users create a todo, we will add an input on the top of our *ToDoList* component.

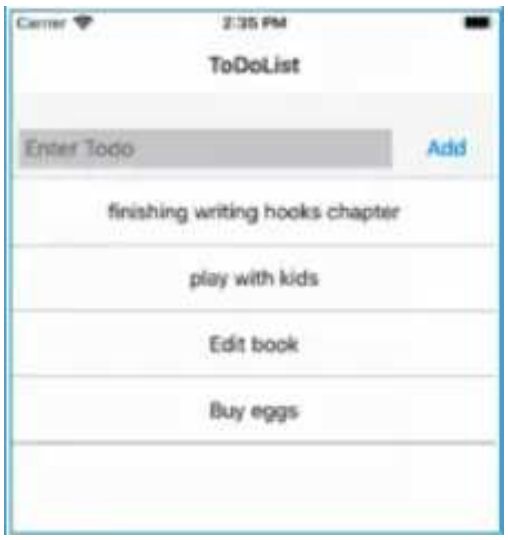



Figure 8.4

We went through forms/inputs in chapter six. So we will skip some explanations regarding implementation of a form/input. Add in the below codes in **bold** into *ToDoList.js*:

```
import React, { useContext, useState } from 'react'
…
import { Text, Container, Header, Item, Input, Button } from 'native-base';
import uuid from 'uuid-random';

export default function ToDoList() {

    const {state, dispatch} = useContext(TodosContext);
  const [todoText, setTodoText] = useState("")

      …
      …
      …
  const handleSubmit = () =>{
    const newToDo = {id: uuid(), text: todoText};
    dispatch({type: 'add', payload: newToDo})
    setTodoText('') // to clear field after adding
  }

  return (
    <Container>
      <Header searchBar>
        <Item>
          <Input
                        placeholder="Enter Todo"
```

```
                        onChangeText={text => setTodoText(text)}
                        value={todoTodoText}
                        />
                    </Item>
                    <Button transparent onPress={handleSubmit}>
                        <Text>Add</Text>
                    </Button>
                </Header>
                <SwipeListView
                    data={state.todos}
                    renderItem={renderItem}
                    renderHiddenItem={renderHiddenItem}
                    leftOpenValue={75}
                    rightOpenValue={-75}
                />
            </Container>
        );
}
```

## *Code Explanation*

```
import { Text, Container, Header, Item, Input, Button } from 'native-base';
import uuid from 'uuid-random';
```

We import the *Header*, *Item*, *Input* and *Button* component from NativeBase to use in our form. We also import *uuid-random* to generate unique ids for our new todos.

```
const [todoText, setTodoText] = useState("")
```

We store the user entered todo text in our component state with *useState*.

```
const handleSubmit = () =>{

    const newToDo = {id: uuid(), text: todoText};
    dispatch({type: 'add', payload: newToDo})
    setTodoText('') // to clear field after adding
}
```

We declare *handleSubmit* and in it, create a new todo with the user-entered text and *uuid()* generating a unique id for us. We then dispatch the ‘add’ action. Next, we bind *handleSubmit* to the Button’s *onPress* event:

```
<Button transparent onPress={handleSubmit}>
    <Text>Add</Text>
</Button>
```

In our Header, we have a searchbar input control for users to enter the todo text. In the control’s *onChangeText*, we bind it to the function that sets the

*todoText* state. These steps should be familiar to you as they are what we have gone through in chapter six.

```
<Header searchBar>
    <Item>
        <Input
                                placeholder="Enter Todo"
                                onChangeText={text => setTodoText(text)}
                                value={todoText}
                                />
    </Item>
    <Button transparent onPress={handleSubmit}>
        <Text>Add</Text>
    </Button>
</Header>
```

We also bind the input control's *value* to *todoText* state so that the input control can be cleared after submission.

## *Add in Reducer*

Now, let's add the 'add' action type case in our *todosReducer* in *App.js*. Add in the 'add' case action as seen in the codes in **bold**:

```
...

function todosReducer(state, action){
  switch(action.type){
    case 'add':
      // add new todo onto array
      const addedToDos = [...state.todos,action.payload]
          // spread our state and assign todos
      return {...state,todos:addedToDos}
    case 'delete':
      const filteredTodoState = state.todos.filter( todo => todo.id !== action.payload.id)
      return {...state, todos: filteredTodoState}
    default:
      return todosInitialState
  }
}
```

## *Code Explanation*

```
const addedToDos = [...state.todos,action.payload]
```

We add in the 'add' case and in it, create a new array with the existing *todos* ('...' spread operator) and adding *newToDo* to it.

```
return {...state,todos:addedToDos}
```

We then return a new state by spreading our existing state and then assigning *addedToDos* to *todos*. By now, you should understand that we shouldn't be modifying state but instead be returning a new one. Else, the application will break.

When you run your app, you will be able to add new todos (fig. 8.5).

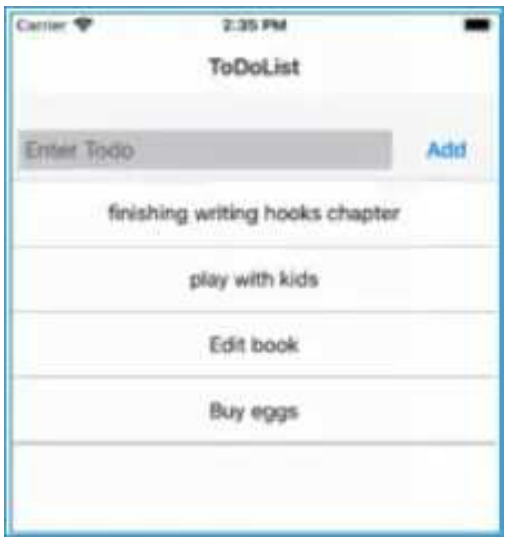



Figure 8.5

# Editing Todos

Next, we want to implement editing our todos. A user will swipe right on the todo and then click on *Edit*. That todo text will then appear in the input field for her to edit.

To do so, we add an *editMode* and *editTodo* to our *ToDoList* local component state.

```
export default function ToDoList() {

    const {state, dispatch} = useContext(TodosContext);
  const [todoText, setTodoText] = useState("")
  const [editMode, setEditMode] = useState(false)
  const [editTodo, setEditTodo] = useState(null)
  const buttonTitle = editMode ? "Edit" : "Add";
```

*editMode* will be set to true when a user clicks on a 'Edit'. *editTodo* will contain the specific todo object to be edited.

We also have a *buttonTitle* that when *editMode* is true, will be set to 'Edit', else, it will be 'Add'. This *buttonTitle* will be displayed as the button's text. So, we render *buttonTitle* between the *Button* component.

```
…
    return (
        <Container>
            <Header searchBar>
                <Item>
                    <Input placeholder="Enter Todo"
                                onChangeText={text => setTodoText(text)}
                                value={todoText} />
                </Item>
                <Button transparent onPress={handleSubmit}>
                    <Text>{buttonTitle}</Text>
                </Button>
            </Header>
```

Next, in the *onPress* of *TouchableOpacity* element for Edit, we specify a function *editRow*:

```
    const renderHiddenItem = (data, rowMap) => (
        <View style={styles.rowBack}>
            <TouchableOpacity
                onPress={() => editRow(data.item, rowMap)}
                >
                <Text>Edit</Text>
            </TouchableOpacity>
            <TouchableOpacity
                style={[styles.backRightBtn]}
                onPress={() => deleteRow(data.item)}
            >
                <Text style={{color: '#FFF'}}>Delete</Text>
            </TouchableOpacity>
        </View>
    );
```

In *editRow*, we provide the *todoText* field with the selected todo's text, set *editMode* to true, and set *editTodo* in state to the selected todo.

```
    const editRow = (todo,rowMap) => {
        setTodoText(todo.text)
        setEditMode(true)
        setEditTodo(todo)
        if (rowMap[todo.id]) {
            rowMap[todo.id].closeRow();
        }
       };
```

Because the particular row that we have swiped on is still left open, we access it with *rowMap[todo.id]* and then close it with *closeRow* .

## *Running your App*

When you select a todo, that todo text should appear in the input (fig. 8.6).


Figure 8.6

Next we implement the actual editing of it.

# Editing

And in *handleSubmit*, we make the following changes.

```
const handleSubmit = () =>{
    if(editMode){
        dispatch({type: 'edit', payload:{...editTodo,text:todoText}})
        setEditMode(false)
        setEditTodo(null)
    }
    else{
        const newToDo = {id: uuid(), text: todoText};
        dispatch({type: 'add', payload: newToDo})
    }

    setTodoText('') // to clear field after adding
}
```

We do a conditional check if *editMode* is true, we dispatch the ‘edit’ action with a changed todo payload *payload:{...editTodo,text:todoText}* to reflect the edited todo text. After the dispatch, we set *editMode* back to false and *editTodo* to null.

Next, we go to *App.js* and add our ‘edit’ action type in *todosReducer*. Add in the following code for ‘edit’:

```
function todosReducer(state, action){
  switch(action.type){

    case 'add':
      const addedToDos = [...state.todos,action.payload]
      return {...state,todos:addedToDos}
```

```
    case 'edit':
      const updatedToDo = {...action.payload}
      const updatedToDoIndex = state.todos.findIndex(t => t.id === action.payload.id)
      const updatedToDos = [
        ...state.todos.slice(0,updatedToDoIndex),
        updatedToDo,

      ];...state.todos.slice(updatedToDoIndex + 1)
      return {...state, todos: updatedToDos}
    case 'delete':
      const filteredTodoState = state.todos.filter( todo => todo.id !== action.payload.id)
      return {...state, todos: filteredTodoState}
    default:
      return todosInitialState
  }
}
```

We first assign the payload to *updatedToDo*. Now, because there is no straightforward function to find an element and then change its content, we have to actually 'slice' up our array to get the items before and after the selected element, and in between, insert *updatedToDo*.

We first get the index of the selected element with:
*const updatedToDoIndex = state.todos.findIndex(t => t.id === action.payload.id)*

And to get the items before the selected element, we use
*state.todos.slice(0,updatedToDoIndex)*
To get the items after the selected element, we use
*state.todos.slice(updatedToDoIndex + 1)*

And to create a new array with the changed todo in between, we have:

```
const updatedToDos = [
  ...state.todos.slice(0,updatedToDoIndex),
  updatedToDo,
  ...state.todos.slice(updatedToDoIndex + 1)
];
```

## *Running your App*

When we run our app now, we can select a todo and then update its text. Congratulations if you made it thus far! In this chapter, we have implemented Create, Read, Update and Delete for our to-dos app. Our current editing of todos is not ideal, however. This is because we have not catered to the case

when a user attempts to edit a todo but decides to cancel the process midway. In the next chapter, we will improve upon this.

And in case you are lost in this chapter, below's the full source code. Alternatively, visit my GitHub repository at

*https://github.com/greglim81/react-native-chp8.*

## *App.js*

```
import React, { useReducer } from 'react';
import ToDoList from './ToDoList';

const todosInitialState = {
  todos:[{ id:'1', text: "finishing writing hooks chapter"},
    { id:'2', text: "play with kids"},
    { id:'3', text: "read bible"}
  ]
};

export const TodosContext = React.createContext()
export default function App() {

  const [state, dispatch] = useReducer(todosReducer,todosInitialState)

  return (
    <TodosContext.Provider value={{state,dispatch}}>
      <ToDoList />
    </TodosContext.Provider>
  )
}

function todosReducer(state, action){
  switch(action.type){
    case 'add':
      // add new todo onto array
      const addedToDos = [...state.todos,action.payload]
          // spread our state and assign todos
      return {...state,todos:addedToDos}
    case 'edit':
      const updatedToDo = {...action.payload}
      const updatedToDoIndex = state.todos.findIndex(t => t.id === action.payload.id)
      const updatedToDos = [
        ...state.todos.slice(0,updatedToDoIndex),
        updatedToDo,
        ...state.todos.slice(updatedToDoIndex + 1)
      ];
      return {...state, todos: updatedToDos}
    case 'delete':
      const filteredTodoState = state.todos.filter( todo => todo.id !== action.payload.id)
```

```
        return {...state, todos: filteredTodoState}
      default:
        return todosInitialState
    }
}
```

## *ToDoList.js*

```
import React, { useContext, useState } from 'react'
import { TodosContext } from './App'
import { SwipeListView } from 'react-native-swipe-list-view';
import { Text, Container, Header, Item, Input, Button } from 'native-base';
import { StyleSheet, TouchableOpacity, View } from 'react-native';
import uuid from 'uuid-random';

export default function ToDoList() {
    // receive state and dispatch from App.js
    const {state, dispatch} = useContext(TodosContext);
    const [todoText, setTodoText] = useState("")
    const [editMode, setEditMode] = useState(false)
    const [editTodo, setEditTodo] = useState(null)
    const buttonTitle = editMode ? "Edit" : "Add";

    const handleSubmit = () =>{
        if(editMode){
            dispatch({type: 'edit', payload:{...editTodo,text:todoText}})
            setEditMode(false)
            setEditTodo(null)
        }
        else{
            const newToDo = {id: uuid(), text: todoText};
            dispatch({type: 'add', payload: newToDo})
        }
        setTodoText('') // to clear field after adding

    }
    const renderItem = data => (
        <View style={styles.rowFront}>
            <Text>{data.item.text}</Text>
        </View>
    );

    const renderHiddenItem = (data, rowMap) => (
        <View style={styles.rowBack}>
            <TouchableOpacity onPress={() => editRow(data.item, rowMap)}>
                <Text>Edit</Text>
            </TouchableOpacity>
            <TouchableOpacity
                style={[styles.backRightBtn]}
                onPress={() => deleteRow(data.item)}
            >
```

```
                <Text style={{color: '#FFF'}}>Delete</Text>
            </TouchableOpacity>
        </View>
    );

    const deleteRow = (todo) => {
        dispatch({type:'delete',payload:todo});
    };

    const editRow = (todo,rowMap) => {
        setTodoText(todo.text)
        setEditMode(true)
        setEditTodo(todo)
        if (rowMap[todo.id]) {
            rowMap[todo.id].closeRow();
        }
    };

    return (
        <Container>
            <Header searchBar>
                <Item>
                    <Input
                        placeholder="Enter Todo"
                        onChangeText={text => setTodoText(text)}
                        value={todoText}
                    />
                </Item>
                <Button transparent onPress={handleSubmit}>
                    <Text>{buttonTitle}</Text>
                </Button>
            </Header>

            <SwipeListView
                data={state.todos}
                renderItem={renderItem}
                renderHiddenItem={renderHiddenItem}
                leftOpenValue={75}
                rightOpenValue={-75}
            />
        </Container>
    );
}

const styles = StyleSheet.create({
    rowFront: {
        alignItems: 'center',
        backgroundColor: '#FFF',
        borderBottomWidth: 0.25,
        justifyContent: 'center',
        height: 50,
    },
```

```
    rowBack: {
        alignItems: 'center',
        backgroundColor: '#DDD',
        flex: 1,
        flexDirection: 'row',
        justifyContent: 'space-between',
        paddingLeft: 15,
    },
    backRightBtn: {
        alignItems: 'center',
        bottom: 0,
        justifyContent: 'center',
        position: 'absolute',
        top: 0,
        width: 75,
        backgroundColor: 'red',
        right: 0
    }
});
```

# Chapter 9: Navigation Between Screens

We have implemented Create, Read, Update and Delete for our to-dos app. As mentioned, our update, or editing of todos is not ideal because we have not catered to the case when a user attempts to edit a todo but decides to cancel the process midway. We will improve it such that when we try to edit a todo, we navigate to a separate Edit screen to do the Edit. And if the user decides to cancel the edit, she can go back to the *ToDoList* screen. In doing so, we also learn how to do navigation between screens in React Native using the React Navigation library (https://reactnavigation.org/).
To get started, we first have to install:

npm install @react-navigation/native

We will also install additional dependencies. Because we are using *Expo*, run:

expo install react-native-gesture-handler react-native-reanimated react-native-screens react-native-safe-area-context @react-native-community/masked-view

We also have to install the stack navigator library with:

npm install @react-navigation/stack

When installation is done, we need to wrap the whole app in *NavigationContainer* which manages our navigation. In *App.js*, import and add the *NavigationContainer* with the codes in **bold**:

## *App.js*

```
import React, { useReducer } from 'react';
import ToDoList from './ToDoList';

import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';
import ToDoDetail from './ToDoDetail';

const todosInitialState = {
  todos:[{ id:'1', text: "finishing writing hooks chapter"},
    { id:'2', text: "play with kids"},
    { id:'3', text: "read bible"}

];
```

```
export const TodosContext = React.createContext()

const Stack = createStackNavigator();

export default function App() {

  const [state, dispatch] = useReducer(todosReducer,todosInitialState)

  return (
    <TodosContext.Provider value={{state,dispatch}}>
      <NavigationContainer>
        <Stack.Navigator initialRouteName="ToDoList">
          <Stack.Screen name="ToDoList" component={ToDoList} />
          <Stack.Screen name="ToDoDetail" component={ToDoDetail} />
        </Stack.Navigator>
        </NavigationContainer>
    </TodosContext.Provider>

  )
}
```

We also have to wrap *createStackNavigator* around it. *createStackNavigator* returns us *Navigator* and *Screen* for our app to transit between screens and manage navigation history. It is similar to how a web browser handles navigation, pushing and popping items from the navigation stack as users interact with it.

Our stack currently has two routes, a *ToDoList* route and a *ToDoDetail* route. A route is specified by the *Screen* component. The *Screen* component receives a *name* prop which is the name of the route and a *component* prop which is the component it will render.

In our case, we have two screens, *ToDoList* and *ToDoDetail*. *ToDoList* screen is represented by the *ToDoList* component and *ToDoDetail* by the *ToDoDetail* component (yet to be implemented).

The initial route for the stack is the *ToDoList* route as defined by *initialRouteName="ToDoList"*. We will define *ToDoDetail* later. For now, let's look at the changes for *ToDoList*.

In *ToDoList*, because we are no longer doing the edit in *ToDoList* but in *ToDoDetail*, we can remove the following from *ToDoList.js*:

```
export default function ToDoList({navigation}) {

    const {state, dispatch} = useContext(TodosContext);
    const [todoText, setTodoText] = useState("")
```

```
  const [editMode, setEditMode] = useState(false)
const [editTodo, setEditTodo] = useState(null)
    const buttonTitle = "Add";
```

We also set *buttonTitle* to just ‘Add’. In *handleSubmit*, we revert it to just for the ‘Add’ action.

```
const handleSubmit = () =>{
    const newToDo = {id: uuid(), text: todoText};
    dispatch({type: 'add', payload: newToDo});
    setTodoText('');
   }
```

The important change however is in *editRow* where we add:

```
const editRow = (todo,rowMap) => {
    if (rowMap[todo.id]) {
        rowMap[todo.id].closeRow();
    }
    navigation.navigate('ToDoDetail', todo)
};
```

Where do we get *navigation*? Notice the *ToDoList* function header now has *navigation* as prop:

```
export default function ToDoList({navigation}) {
```

*navigation* is a prop passed into every screen component in the stack navigator. We can navigate to a route by specifying *navigation.navigate(‘<route_name>’)*. But because we need to also pass data to the route, we pass the data as an object (*todo*) in the second parameter.

Let’s now define our *ToDoDetail*. Create a new file *ToDoDetail.js* with the following code:

## *ToDoDetail.js*

```
import React, {useState, useContext} from 'react';
import { Form, Item, Text, Button, Input } from 'native-base';
import { TodosContext } from './App'

export default function ToDoDetail({route, navigation}) {
    const { text } = route.params;

    const [todoText, setTodoText] = useState(text)
    const {state, dispatch} = useContext(TodosContext);

    return (
      <Form>
```

```
        <Item regular>
            <Input placeholder="Edit Todo"
                                onChangeText={text => setTodoText(text)}
                                value={todoText}
                        />
        </Item>

        <Button onPress={() =>{
                dispatch({
                                type: 'edit', payload:{...route.params,text:todoText}
                        });
            navigation.navigate('ToDoList');
        }}>
            <Text>Edit</Text>
        </Button>
      </Form>
    );
  }
```

## *Code Explanation*

```
export default function ToDoDetail({route, navigation}) {
    const { text } = route.params;
```

We can retrieve the passed data with *route.params* that is available. With `const { text } = route.params`, we assign *route.params.text* to *text* so that we don't have to always explicitly state `route.params.text` .

```
    const [todoText, setTodoText] = useState(text)
```

Using *text*, we then set it to *todoText* state.

```
    const {state, dispatch} = useContext(TodosContext);
```

We then get access to the global state and dispatch with *useContext(TodosContext)*. See how we avoid passing props all around the place with *useContext*?

```
        <Item regular>
            <Input placeholder="Edit Todo"
                                onChangeText={text => setTodoText(text)}
                                value={todoText}
                        />
        </Item>
```

Because we bind our Input's value to *todoText*, it will populate the input with the passed-in todo.

```
        <Button onPress={() =>{
            dispatch({
```

```
                        type: 'edit', payload:{...route.params,text:todoText}
                    });
                navigation.navigate('ToDoList');
            }}>
```

We then have a button which when pressed, dispatches the ‘edit’ action with the payload having the changed *todoText*. We then navigate back to *ToDoList*.

## *Running our App*

When we run our app now, and we click on the ‘Edit’, we are brought to the *ToDoDetail* screen with the todo text displayed in the input. Notice that the header automatically includes a back button to let the user go back to the main screen if they decide not to edit the todo. But if we click on the ‘Edit’ button, we are brought back to *ToDoList* with the todo text edited. In the next chapter, we will see how to persist our data by connecting to an external API.

Visit my GitHub repository at *https://github.com/greglim81/react-native-chp9* if you have not already have the full source code for this chapter.

# Chapter 10: Connecting to an API to Persist Data

We have made progress in our todo app. But our data is not yet persistent. That is, when we reload our application, all the changes we have done to our data is gone. In this chapter, we will connect to an API to enable persistency in create, read, update and delete todos.
The API we are connecting to can be supported by any backend, e.g. Nodejs, Firebase, ASP.Net etc. Setting up a backend is obviously beyond the scope of this book. But to quickly set up a mock API, we will use *json-server* (*https://github.com/typicode/json-server*) which makes it easy for us to set up JSON APIs for use in demos and proof of concepts.

In your Terminal, stop your React Native app first. Then run:

```
npm install -g json-server
```

(Note: you might need *sudo*)

Next, prepare a *todos.json* file which contains the following:

```
{
  "todos":[
          { "id":1, "text": "finishing writing hooks chapter"},
              { "id":2, "text": "play with kids"},
              { "id":3, "text": "read bible"}

} ]
```

These are the same *todos* we have in *App.js*:

Back in Terminal, in the folder that contains *todos.json*, run the command:

```
json-server todos.json
```

This will run a mock REST API server in your local machine and you can see the end point at:

```
http://localhost:3000/todos
```

The endpoint will return an array of *todos* just like in our initial state back in *App.js*.

Now that we have a mock REST API running, let's connect to it from our React Native app. But because we can only connect to the local machine we are developing the app from, we have to run our React Native app on the local machine via either the iOS or Android simulator (fig. 10.1).


Figure 10.1

You can imagine that the REST API is deployed on a server in a real-world scenario. Simply change the URL of the endpoint to point to the server. The rest of the code remains the same.

# Creating a Custom Hook to Fetch Initial App Data

Because we are retrieving our todos from the API, our initial *todos* in *App.js* will just be an empty array.

```
const todosInitialState = {
  todos:[]
};
```

We will next create a custom hook to call our API.  Create a new file *useAPI.js* with the following code:

## *useAPI.js*

```
import {useState, useEffect} from 'react'
import axios from 'axios'

const useAPI = endpoint => {
  const [data, setData] = useState([]) // initial state empty array

  //To call data when component is mounted,
  useEffect(()=> {
    getData()
```

```
  },[])

  const getData = async () => {
    const response = await axios.get(endpoint)
    setData (response.data)
  }

  return data;
}

export default useAPI;
```

*useAPI* has a state, *data* to store the data retrieved from the API. In the *getData* function, we use *axios.get* with the endpoint and then set *data* in state. Because we use *await* on *axios.get*, we need to specify *async* at the declaration of *getData*.

We apply *useEffect* with an empty array in the second argument to call *getData* when the component is mounted.

Notice that *useAPI* does not have any specific relation to *todos*. We name the results retrieved generically as *data* rather than *todos*. This is so that the custom hook can be used not just for our *todos* app, but for other API calls as well. Being in a separate file, other components can easily call *useAPI* by importing it and provide their own endpoint arguments. This illustrates how React hooks provide code reusability.

Now, in *ToDoList.js*, we will call *useAPI*. Add in the codes in **bold**:

```
import React, { useContext, useState, useEffect } from 'react';
…

…
…

import useAPI from './useAPI';
import axios from 'axios'; //npm install axios

export default function ToDoList({navigation}) {

    const {state, dispatch} = useContext(TodosContext);
    const [todoText, setTodoText] = useState("")
    const buttonTitle = "Add";

    const endpoint = "http://localhost:3000/todos/"
    const savedTodos = useAPI(endpoint)


    useEffect(()=>{
        dispatch({type: "get", payload: savedTodos})
      },[savedTodos]) // dispatch whenever savedTodos changes
```

We import *useEffect*. We also import *useAPI* and call *useAPI* with the endpoint and assign the results to *savedTodos*.

In *useEffect*, we then dispatch the *get* action with *savedTodos* as the payload:

In *App.js*, in *todosReducer*, we add the case for “get”:

```
function todosReducer(state, action){
  switch(action.type){
    case 'get':
      return {...state,todos:action.payload}
    case 'add':
      const newToDo = {id: uuidv4(), text: action.payload}
      const addedToDos = [...state.todos,newToDo]
      return {...state,todos:addedToDos}
…
```

When we run our app, we will be able to retrieve our todos from the API and display them.

# Delete Request to Remove Todos

To delete a todo, we need to get the specific url for a todo item. We get that by appending the *todo.id* to the endpoint: *endpoint + todo.id*, e.g. *http://localhost:3000/todos/1*

Thus in *ToDoList*, *deleteRow*, we specify the todo url, *endpoint + todo.id* to *axios.delete* to remove the todo:

```
const deleteRow = async (todo) => {
    await axios.delete(endpoint + todo.id);
    dispatch({type: 'delete',payload:todo});
};
```

Because we use *await* on *axios.delete*, we need to specify *async* at the declaration of the function.

Run your app now and when you delete a todo, it will be removed.

# Perform Post Request to Add Todos

Next, we will perform a post request to add a todo. In *ToDoList.js*, add the below code in *handleSubmit*:

```
const handleSubmit = async () =>{
```

```
    const newToDo = {id: uuid(), text: todoText};
    const response = await axios.post(endpoint,newToDo)
    dispatch({type: 'add', payload: newToDo});
    setTodoText('');
}
```

*axios.post* requires two parameters. The first parameter is the URL of the service endpoint. The second parameter is the object which contains the properties we want to send to our server. We thus call *axios.post* with our endpoint and the new todo object *newToDo*. And because we use *await*, we have to label *handleSubmit* as an asynchronous function with the async keyword. We then continue to dispatch the *add* action with *newToDo* as the payload.

Run your app and you should be able to add todos persistently.

## Performing Patch Request to Update Todos

Finally, let's implement editing a todo. It is just a few lines of code in *ToDoDetail.js*:

```
import axios from 'axios';
…
…
…
    const endpoint = "http://localhost:3000/todos/"

  return (
    <Form>
      <Item regular>
          <Input placeholder="Edit Todo" onChangeText={text => setTodoText(text)} value={todoText} />
      </Item>
        <Button onPress={async () =>{
            await axios.patch(endpoint+ route.params.id,{text:todoText})
            dispatch({type: 'edit', payload:{...route.params,text:todoText}});
            navigation.navigate('ToDoList');
      }}>
          <Text>Edit</Text>
      </Button>
    </Form>
  );
```

We call *patch* with the specific todo's end point and the attribute to be updated (in our case *text*). The rest of the code remains the same. And when we run our app now, we can edit a todo!

we run our app now, we can call it done.

In case you got lost in any of the steps, see the full code at my GitHub repository: *https://github.com/greglim81/react-native-chp10/*

## Deploying your App

Step by step instructions on how to create a production build and submit to the app store is available in the Expo documentation and is relatively straightforward. This book does not show specific steps on deployment because the process changes quite often. Not only do developers have to deal with changes to the submission process on the Apple App Store and the Google Play store, but also changes in the Expo framework itself. Because of this, we recommend you check out Expo's official guide to app deployment (*https://docs.expo.io/workflow/publishing/*). It is a multi page guide that is kept up to date by the Expo community. The guide walks through every step you need to know about, including some extra steps that are specific to Expo. Although we could put together some sections on deployment, they would mostly be rehashing the same information listed in this guide.

## Summary

With this knowledge, you can move on and build more complicated fully functional React Native applications of your own!

Hopefully, you have enjoyed this book and would like to learn more from me. I would love to get your feedback, learning what you liked and didn't for us to improve.

Please feel free to email me at support@i-ducate.com if you encounter any errors with your code or to get updated versions of this book. Visit my GitHub repository at https://github.com/greglim81 if you have not already have the full source code for this book.

If you didn't like the book, or if you feel that I should have covered certain additional topics, please email us to let us know. This book can only get better thanks to readers like you.

Thank you and all the best for your learning journey in React Native!

# About the Author

Greg Lim is a technologist and author of several programming books. Greg has many years in teaching programming in tertiary institutions and he places special emphasis on learning by doing.

Contact Greg at support@i-ducate.com.