Getting Started with React Native

Web developers who want to develop native mobile applications face a high barrier to entry because they are forced to learn platform-specific languages and frameworks. Numerous hybrid technologies have tried to simplify this process, but have failed to achieve the performance and appearance that users expect.

This book will show you all the advantages of true native development that React Native has without the steep learning curve, leveraging the knowledge you already have. We do this by getting you up and running quickly with a sample application. We'll introduce you to the fundamentals of creating components and explain how React Native works behind the scenes. Once you have established a solid foundation, you will dive headfirst into developing a real-world application from start to finish. Along the way, we will demonstrate how to create multiple screens and navigate between them, use layout and style native UI components, and access native APIs, such as local storage and geolocation. Finally, we tackle the advanced topic of Native modules, which demonstrates that there are truly no limits to what you can do with React Native.

Who this book is written for

This book is for web developers who want to learn to build fast, good-looking, native mobile applications using the skills they already have. If you already have some JavaScript knowledge or are using React on the web, then you will be able to quickly get up and running with React Native for iOS and Android.

What you will learn from this book

- Set up the React Native environment on both devices and emulators
- Gain an in-depth understanding of how React Native works behind the scenes
- Write your own custom native UI components
- Learn the ins and outs of screen navigation
- Master the art of layout and styles
- Work with device-exclusive data such as geolocation
- Integrate native modules in Objective-C and Java that interact with JavaScript
- Test and deploy your application for a production-ready environment


Ethan Holmes
Tom Bray

Getting Started with React Native

Learn to build modern native iOS and Android applications using JavaScript and the incredible power of React
In this package, you will find:

- The author’s biography
- A preview chapter from the book, Chapter 3 *Beginning with the Example Application*
- A synopsis of the book’s content
- More information on *Getting Started with React Native*
About the Authors

**Ethan Holmes** is a Software Engineer from Vancouver, BC, Canada. He obtained a B.Sc. in computer science from Simon Fraser University. He has primarily been a full-stack web developer working and creating applications for start-ups in the Silicon Beach area. Currently, he is stationed at Cargomatic, disrupting the freight industry. After learning React for the web, learning React Native complemented the skills he obtained as a web developer and allowed him to quickly make the transition to mobile development.

You can follow him on Twitter at @sherclockholmes.
Tom Bray has been developing for the web since the browser wars of the late 90s when DHTML was the buzzword of the day. Creating great user experiences using the cutting edge technologies of the day has always been his passion, from Flash to Flex to Adobe AIR to React, and React Native.

He has created sophisticated software that has been used by large companies, such as Adobe, MySpace, Cisco, Informatica, and Dell; it has been a key contributor to numerous start-ups where he has worn many hats and gained a broad skill set. He currently serves as the Principal Software Architect with Cargomatic where he has designed a system to orchestrate the movement of ocean freight to and from America's ports—a solution that leveraged React Native to assign work to truck drivers and track their progress.

You can follow him on Twitter at @tombray.
Preface

Why are there so many alternatives to using native languages to write mobile apps? And, more importantly, why does the world need yet another approach? Obviously, there must be a problem that hasn't been solved.

Developers want to use just one language to develop for both iOS and Android. Web developers want to reuse their existing JavaScript knowledge and leverage the web frameworks they already know and love. This is why Apache Cordova (PhoneGap) exists. By wrapping a web browser in a native app, developers can package their HTML, CSS, and JavaScript applications in a native shell, but why aren't all mobile applications based on Cordova?

Users expect native performance, with a native user experience. Hybrid apps don't solve the user's problems, they solve the developer's problems. We need a technology that can do both!

React Native changes the game with applications that are truly native. It doesn't use a WebView or transpile JavaScript to native languages. Think of it as native UI components being controlled by a JavaScript brain. The result is a user experience that is indistinguishable from any other native app, and a developer experience that leverages the amazing productivity benefits of JavaScript and the React framework.

Armed with React Native, you'll finally be able to leverage your web development skills in the mobile world without sacrificing quality or performance. It's the Holy Grail, and we're excited to show you what React Native can do and to see what amazing apps you create with it!
What this book covers

Chapter 1, Exploring the Sample Application, is a step-by-step guide to running the sample iOS Application.

Chapter 2, Understanding React Native Fundamentals, covers the basics of React Native and gives brief insight into how the Virtual DOM improves performance. Then there is an introduction to props and state by creating your first component.

Chapter 3, Beginning with the Example Application, begins with generating the project files for iOS and Android. Then it continues with creating the first screens and adding navigation to the application.

Chapter 4, Working with Styles and Layout, covers the ins and outs of laying out and styling content in React Native. Learn how to apply React CSS and Flexbox to your components.

Chapter 5, Displaying and Saving Data, uses ListViews to display data and save notes using the AsyncStorage API.

Chapter 6, Working with Geolocation and Maps, discusses the geolocation API and Map Component.

Chapter 7, Integrating Native Modules, focuses on integrating third party native modules from the React Native community into your applications.

Chapter 8, Releasing the Application, goes through the release process for iOS and Android so you are ready to submit an application to the AppStore or the Google Play Store.
Now that you have an idea about how React Native works and how to create components, let's create your first React Native application. Throughout this book, we will be developing a note-taking application which we'll call ReactNotes. By the end of the book, you'll have a fully featured application that allows you to create notes, save them to a device, view the list of the notes you've saved, take pictures with the device and attach them to your notes, and much more.

In this chapter, we'll build the skeleton of the application, create a HomeScreen and NoteScreen. We'll also add navigation that allows you to switch between the screens, and along the way you'll learn about creating your own components and handling events.

The topics that we will cover in this chapter are:

- How to generate iOS and Android project files
- Examining the React Native starter template
- Creating the first component, SimpleButton
- Debugging with Chrome Developer Tools
- Exploring navigation and transitioning between screens
- Developing the UI to create notes
Generating the projects
To start building our note taking application for iOS, we are going to need a couple of command-line tools.

- React Native 0.14.2 requires Node.js v4+, we are going to use v5.0.0; visit https://nodejs.org for more information (we recommend managing different node versions with NVM https://github.com/creationix/nvm)
- Install the latest version of NPM from https://www.npmjs.com/

Great, now that we have these tools we can install the react-native-cli. The react-native-cli exposes an interface that does all the work of setting up a new React Native project for us:

1. To install react-native-cli, use the npm command:
   ```bash
   npm install -g react-native-cli
   ```

2. Next, we are going to generate a new React Native project called ReactNotes using the cli and the react-native init command. The output of the command looks similar to the following:
   ```bash
   $ react-native init ReactNotes
   ```
   This will walk you through the creation of a new React Native project in/Users/ethanholmes/ReactNotes.

3. Set up a new React Native app in /Users/ethanholmes/ReactNotes:
   ```bash
   create .flowconfig
create .gitignore
create .watchmanconfig
create index.ios.js
create index.android.js
create ios/main.jsbundle
create ios/ReactNotes/AppDelegate.h
create ios/ReactNotes/AppDelegate.m
create ios/ReactNotes/Base.lproj/LaunchScreen.xib
create ios/ReactNotes/Images.xcassets/AppIcon.
   ```
   appiconset/Contents json
create ios/ReactNotes/Info.plist
create ios/ReactNotes/main.m
create ios/ReactNotesTests/ReactNotesTests.m
create ios/ReactNotesTests/Info.plist
create ios/ReactNotes.xcodeproj/project.pbxproj
create ios/ReactNotes.xcodeproj/xcshareddata/xcschemes/
   ReactNotes.xcscheme
To run your app on iOS:

Open /Users/ethanholmes/ReactNotes/ios/ReactNotes.xcodeproj in Xcode
Hit Run button

To run your app on Android:

Have an Android emulator running, or a device connected
cd /Users/ethanholmes/ReactNotes
react-native run-android

The root directory of the Xcode project is generated in the ReactNotes folder, with the same name as we gave react-native-cli when we ran the command. Follow the steps at the end of the React Native set up to see what it produces.
Xcode and the iOS simulator

We are going to start by running the starter template in the iOS simulator through Xcode:

1. In Xcode, select File | Open and navigate to the ReactNotes folder.
2. Open the ReactNotes.xcworkspace file, as shown in the following figure:

3. Click on Run (or Cmd + R) to run the application in the iOS simulator, the following screenshot will be shown:
Chapter 3

Just like that, we already have the React Native template up and running on the iOS simulator!

Welcome to React Native!
To get started, edit index.ios.js
Press Cmd+R to reload,
Cmd+D or shake for dev menu

Just like that, we already have the React Native template up and running on the iOS simulator!

The Android SDK and emulator
Facebook has a detailed step by step guide set up on Android SDK and emulator. You can access the React Native Docs at https://facebook.github.io/react-native/docs/android-setup.html. In this section, we will only cover the basics of running the application on the Android emulator.
When running the project in the iOS simulator, we can run it from the Xcode IDE. Android, on the other hand, doesn't require any particular IDE and can be launched directly from the command line.

To install the Android apk to the emulator, use the following command:

```bash
$ react-native run-android
```

The following screenshot will be generated:

Let's start by modifying the contents of the starter template and display a different message.

![Welcome to React Native!
To get started, edit index.android.js
Shake or press menu button for dev menu](image)
Modifying the React Native starter template

Open `index.ios.js`, located in the root directory, in the text editor of your choice. Here is the code that `react-native-cli` generated:

```javascript
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 */
'use strict';

var React = require('react-native');
var {
    AppRegistry,
    StyleSheet,
    Text,
    View,
} = React;

var ReactNotes = React.createClass({
    render: function() {
        return (
            <View style={styles.container}>
                <Text style={styles.welcome}>
                    Welcome to React Native!
                </Text>
                <Text style={styles.instructions}>
                    To get started, edit `index.ios.js`
                </Text>
                <Text style={styles.instructions}>
                    Press Cmd+R to reload,
                    Cmd+D or shake for dev menu
                </Text>
            </View>
        );
    }
});

var styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#F5FCFF',
    },
});
```
### Beginning with the Example Application

```javascript
welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
},
instructions: {
    textAlign: 'center',
    color: '#333333',
    marginBottom: 5,
},
});

AppRegistry.registerComponent('ReactNotes', () => ReactNotes);
```

Although `react-native-cli` generates the starter template using the ES5 `createClass`, we will be creating our components using ES6 classes.

A lot of things are included in here, but bear with us as we break it down for you. If we take a closer look at the render method, we can see the familiar `View` and `Text` components that we encountered in the previous chapter. Note how the `index.js` file is a component itself (`ReactNotes`). Change the value in line 30 to `Welcome to React Notes!`. Save it and then press `Cmd + R` from the simulator or, in the top menu, navigate to **Hardware | Shake Gesture** and select **Reload** from the pop-up action sheet. The text on screen re-renders to show the text value we just modified! We are no longer constrained to wait for the Xcode to recompile in order to see our changes as we can reload straight from the simulator. Continue making changes and reload it in the simulator to get a feel for the work flow.

### Structuring the application

It's time to add a little interactivity to our application. You can begin by adding a simple button component to the screen that is touchable. In the root directory, create a folder called `App` and another folder inside the `App` folder called `Components`. In the `Components` directory, add a file named `SimpleButton.js`. This will be the directory in which we store and reference the components we create.

Note that the React Native code created in this chapter will work for both iOS and Android. Simply replace `index.ios.js` with `index.android.js` if you are interested in Android only. The screenshots and instructions will be mainly for the iOS simulator.
Creating the SimpleButton component

Let's start by rendering some text to the screen and importing it into our index.ios.js file. In SimpleButton.js, add:

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

export default class SimpleButton extends React.Component {
    render () {
        return (<View>
            <Text>Simple Button</Text>
        </View>);
    }
}
```

ES6 de-structuring assignment var [a, b] = [1, 2]; is used to extract Text and View from the React Native module.

We are going to include our newly created component in index.ios.js and simplify it to ES6 syntax:

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

import SimpleButton from './App/Components/SimpleButton';

class ReactNotes extends React.Component {
    render () {
        return (<View style={styles.container}>
            <SimpleButton />
        </View>);
    }
}
```
Beginning with the Example Application

```javascript
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  }
});
AppRegistry.registerComponent('ReactNotes', () => ReactNotes);
```

The output for the preceding code is:
We're off to a good start; it's time to add some interactivity to our button. In SimpleButton.js, add the TouchableOpacity component to the destructuring assignment. TouchableHighlight, TouchableOpacity, and TouchableWithoutFeedback are similar components that respond to touches, and it takes an onPress prop for a function to react to the touch. Wrap the existing code in the render function with the TouchableOpacity component:

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

export default class SimpleButton extends React.Component {
  render () {
    return (
      <TouchableOpacity onPress={() => console.log('Pressed!')}>  
        <View>
          <Text>Simple Button</Text>
        </View>
      </TouchableOpacity>
    );
  }
}
```
Beginning with the Example Application

Go ahead and try tapping (or clicking) on the text now, you should be able to see that the opacity of the text decreases as you press it. But where has our `console.log(...)` output gone? Open the Developer menu (Hardware | Shake Gesture) and select Debug in Chrome. This opens a Chrome Window at localhost:8081/debugger-ui for debugging, as shown in the following screenshot:

Lo and behold, here is the console log that we specified in our `SimpleButton` component. Behind the scenes, the JavaScript code is being run from inside the Chrome tab and loaded onto the mobile device on startup or reload. From here, you have access to all the Chrome Developer Tools you will normally use, including the addition of break points.

Navigation

Now, it's time to make our application more actionable. Let's begin by transforming our `SimpleButton` into a Create Note button. When the user clicks on the Create Note button, it transitions them to another screen to create notes. To do this, we need our button to be able to accept a function via props from `index.ios.js` to activate the transition. We will add some custom text as well for extra flair:

```javascript
import React, {
  Text,
  TouchableOpacity,
  View
} from 'react-native';
```
export default class SimpleButton extends React.Component {
  render () {
    return (
      <TouchableOpacity onPress={this.props.onPress}>
        <View>
          <Text>{this.props.customText || 'Simple Button'}</Text>
        </View>
      </TouchableOpacity>
    );
  }
}

SimpleButton.propTypes = {
  onPress: React.PropTypes.func.isRequired,
  customText: React.PropTypes.string
};

Now, we have extended our SimpleButton component to be reusable with minimal changes. We can always pass different functions through the onPress prop and add custom text if we choose. This is all that we need to modify our SimpleButton; now to include the transition functionality to our index.io.js file.

The following image shows the validating props revisited page:

Remember propTypes from the previous chapter? If we forget to pass the onPress prop, the console will log a warning reminding us to pass it. Note that there is no warning for customText since it was not set to isRequired.
The Navigator component

The Navigator component is a reimplemention of the UINavigationController provided by React Native to manage various screens. Similar to a stack, you can push, pop, and replace routes onto the Navigator. It is fully customizable on both iOS and Android, which we will cover in the next chapter. Import the Navigator into index.ios.js and replace the contents of the render method with:

```javascript
import React, { 
    AppRegistry,
    Navigator, 
    StyleSheet,
    View
} from 'react-native';

render () {
    return ( 
        <Navigator  
            initialRoute={{name: 'home'}}
            renderScene={this.renderScene} 
        />
    );
}
```

Navigator receives a prop called `initialRoute` that accepts an object to be the first route to be put on the stack. The route object can contain any attribute that you need to pass to the screen components. All we need for now is the name of the screen we want to transition to. Next, we need to create the function to pass to the `renderScene` prop. In the ReactNotes component, we are going to create a function that takes `route` and `navigator` as parameters, as shown:

```javascript
class ReactNotes extends React.Component {
    renderScene (route, navigator) {
        ...
    }
    render () {
        ...
    }
}
```
When we first load our application, the parameter route will be the object we pass into initialRoute. Using a switch statement and looking at the values of route.name allows us to choose the component we want to render:

```javascript
renderScene (route, navigator) {
    switch (route.name) {
        case 'home':
            return (
                <View style={styles.container}>
                    <SimpleButton
                        onPress={() => console.log('Pressed!')}
                        customText='Create Note'
                    />
                </View>
            );
        case 'createNote':
    }
}
```

Here, under the home case, you can see our slightly modified code from the original render method in ReactNotes; we have included the onPress and customText props we created earlier. You can add another component to App/Components/named NoteScreen.js; this screen will contain the functionality to create a new note:

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

export default class NoteScreen extends React.Component {
    render () {
        return (
            <View style={styles.container}>
                <Text>Create Note Screen!</Text>
            </View>
        );
    }
}

var styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
    }
});
```
For now, we are only going to use this screen when we press the Create Note button. In the onPress prop arrow function, we are going to push a new route onto the stack using navigator.push:

```javascript
import NoteScreen from './App/Components/NoteScreen';

class ReactNotes extends React.Component {
    renderScene (route, navigator) {
        switch (route.name) {
            case 'home':
                return (
                    <View style={{styles.container}}>
                        <SimpleButton
                            onPress={() => {
                                navigator.push({
                                    name: 'createNote'
                                });
                            }}
                            customText='Create Note'
                        />
                    </View>
                );
            case 'createNote':
                return {
                    <NoteScreen />
                };
        }
    }
}
```

Note that push also takes a regular JavaScript object, so we need to include the name attribute for our NoteScreen; reload the application in the simulator and press on the Create Note button. A smooth animated transition between the two screens will occur without adding any extra code.

**Navigator.NavigationBar**

At this point you must be thinking A button is OK, but is there a better, more native way to do navigation? Of course, as a part of the Navigator component, you can pass a navigationBar prop to add a persistent top navigation bar across every screen. The Navigator.NavigationBar is a subcomponent that accepts an object that defines the left and right buttons, a title, and styles (although we are going to leave it unstyled until the next chapter). Modify the ReactNotes render function to include the navigationBar, as shown:

```javascript
render () {
    return (
        <Navigator
```
initialRoute={{name: 'home'}}
renderScene={this.renderScene}
navigationBar={
  <Navigator.NavigationBar
    routeMapper={NavigationBarRouteMapper}
    />
  />
};

The **routeMapper** prop accepts an object containing functions for the **LeftButton**, **RightButton**, and **Title** attributes. Let’s insert this object after the imports at the top of index.ios.js:

```javascript
var NavigationBarRouteMapper = {
  LeftButton: function(route, navigator, index, navState) {
    ...
  },
  RightButton: function(route, navigator, index, navState) {
    ...
  },
  Title: function(route, navigator, index, navState) {
    ...
  }
};
```

Advancing the flow of our application to the **CreateNote** screen will require displaying a right-hand button in the navigator bar. Luckily, we already have our simple button set up to push the state onto the navigator. In the **RightButton** function, add:

```javascript
var NavigationBarRouteMapper = {
  ...
  RightButton: function(route, navigator, index, navState) {
    switch (route.name) {
      case 'home':
        return (        ...
      <SimpleButton
        onPress={() => {
          navigator.push({
            name: 'createNote'
          })
        }}
      );
    }
  }
};
```
Beginning with the Example Application

```javascript

 Similar to our previous renderScene method, we can switch on the value of route.name. The default expression in the switch statement is there to ensure that different screens do not return a button unless we include them. Let's also go ahead and add a LeftButton to the NavigationBar when it's on the NoteScreen to return to the home screen.

 var NavigationBarRouteMapper = {
     LeftButton: function(route, navigator, index, navState) {
         switch (route.name) {
             case 'createNote':
                 return (<SimpleButton
                         onPress={() => navigator.pop()}
                         customText='Back'/>
                 );
             default:
                 return null;
         }
     },
     ...,
 };

 The navigator.pop() will remove the route on the top of the stack; thus, returning us to our original view. Finally, to add a title, we do the exact same thing in the Title attributes function:

 var NavigationBarRouteMapper = {
     ...,

     Title: function(route, navigator, index, navState) {
         switch (route.name) {
```
case 'home':
    return (
        <Text>React Notes</Text>
    );

case 'createNote':
    return (
        <Text>Create Note</Text>
    );
}
}
}

Now, let's update the original renderScene function to get rid of the button and include the home screen as a component. Create a new component called HomeScreen; the contents of this screen won't matter much, as we will come back to it later:

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

export default class HomeScreen extends React.Component {
    render () {
        return (
            <View style={styles.container}>
                <Text>Home</Text>
            </View>
        );
    }
}

var styles = StyleSheet.create(
    {
        container: {
            flex: 1,
            justifyContent: 'center',
            alignItems: 'center',
        }
    });

Then import it into index.ios.js or index.android.js:

    import HomeScreen from './App/Components/HomeScreen';

...
Beginning with the Example Application

class ReactNotes extends React.Component {
    renderScene (route, navigator) {
        switch (route.name) {
            case 'home':
                return (<HomeScreen />);
            case 'createNote':
                return (<NoteScreen />);
            }
            ...
        }
    }

Now, let's see how the navigation bar persists across each route:
That's it! Reload and take a look at how the static navigation bar persists across each route:

![iOS Simulator - iPhone 5s - iPhone 5...]

Carrier ☼
4:44 PM
React Notes
Create Note

Home

For a more detailed guide on Navigator, check out the React Native documentation at [https://facebook.github.io/react-native/docs/navigator.html](https://facebook.github.io/react-native/docs/navigator.html). We now have the proper infrastructure to go ahead and start adding the create note functionality to our application.
The NoteScreen – first pass

Now that we have a NoteScreen and can navigate to it, let's start making it useful. We'll need to add some TextInput components, one for the title of the note and one to capture the body. We'll want to automatically set focus on the TextInput for the title, so the user can start typing right away. We'll need to listen to events on the TextInput components, so we can keep a track of what the user has typed by updating the state. We'd also like to know when the user has finished editing the title of the note, so that we can automatically set focus on the TextInput for the body.

First, let's add the TextInput component to our list of dependencies and remove the Text component since we no longer need it:

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

Before we add the TextInput components to the View, let's get a few style updates out of the way:

```javascript
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    marginTop: 64
  },
  title: {
    height: 40
  },
  body: {
    flex: 1
  }
});
```

Note that we've added a marginTop: 64 to the container. This is important because we want to make sure that the NavigationBar doesn't accidentally intercept the onPress events we want our TextInput to receive. We've also added styles for each of the TextInput we're about to add. We'll talk more about styles in detail in Chapter 4, Working with Styles and Layout.
Now, in our render function, let's replace the `Text` component with two `TextInput` components, such as:

```javascript
render () {
    return (
        <View style={styles.container}>
            <TextInput placeholder="Untitled"
                style={styles.title}/>
            <TextInput placeholder="Start typing" style={styles.body}/>
        </View>
    )
}
```

Before we try this out, notice that the `TextInput` component has a placeholder property that allows us to tell the user what the `TextInput` is for without having to take up additional screen real estate by labeling our form fields. I've also specified `multiline={true}` on the second `TextInput` so the user can add as much text as they want.

Now let's refresh the application in the simulator and you should see something like this:
**Beginning with the Example Application**

You should be able to click into TextInput and start typing. If you'd like to use the on-screen keyboard available in the simulator, you can press CMD+K / CTRL+K.

Let's improve the user experience a little bit by making the title TextInput focus automatically and show the keyboard when the user navigates to the NoteScreen:

```jsx
<TextInput
    ref="title"
    autoFocus={true}
    placeholder="Untitled"
    style={styles.title}
/>
```

To be even more user friendly, let's listen for the event that tells us the user has finished editing the title and automatically set focus on the body TextInput. To do that we'll need to make a slight change to the body TextInput so that we can refer to it in our event handler:

```jsx
<TextInput
    ref="body"
    multiline={true}
    placeholder="Start typing"
    style={styles.body}
/>
```

Notice the `ref="body"`. Any React component can be given a `ref` so that it can be referenced in your javascript code. Now, in the title TextInput, we can add an onEndEditing event handler that sets focus on the TextInput body:

```jsx
<TextInput
    autoFocus={true}
    placeholder="Untitled"
    style={styles.title}
    onEndEditing={(text) => {this.refs.body.focus()}}
/>
```

Avoid using refs to set and get values on your components! That's what `state` is for and we'll learn all about state in Chapter 5, Displaying and Saving Data.

Now when you refresh the application in the simulator and navigate to the NoteScreen, you will see that the title TextInput has focus and you should be able to type something. Press Enter and see the focus automatically switch to the body and start typing there as well. If you're not seeing the on-screen keyboard when you try this, press CMD + K / CTRL + K and try again.
Summary

In this chapter, we have created the skeleton of our ReactNotes application, walked you through how to create a new project, created Views and custom components, navigated between the HomeScreen and NoteScreen, and debugged your application.

You now have a solid foundation for all of the topics we'll introduce throughout the rest of the book. However, there are two big problems with this application, it's not pretty and it doesn't do anything! In the next two chapters, we'll solve both of those problems and you'll be well on your way to master React Native!
Where to buy this book
You can buy Getting Started with React Native from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.