O'REILLY®

Early Release

RAW & UNEDITED

Learning React Native

BUILDING NATIVE MOBILE APPS WITH JAVASCRIPT

Bonnie Eisenman

Learning React Native

Building Mobile Applications with Javascript

Bonnie Eisenman

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

{{ title }}
by Author Name

If your book has an ISBN or a book ID number, add it here as well.

Table of Contents

Preface Title

This Is an A-Head

Congratulations on starting your new project! We've added some skeleton files for you, to help you get started, but you can delete any or all of them, as you like. In the file called chapter.html, we've added some placeholder content showing you how to markup and use some basic book elements, like notes, sidebars, and figures.

CHAPTER 1
Introduction

React is a Javascript library for building user interfaces. Open-sourced by Facebook in 2013, it has been met with excitement from a wide community of developers. Corporate adopters have included the likes of Netflix, Yahoo!, Github, and Codecademy. Users praise React for its performance and flexibility, as well as its declarative, component-based approach to building user interfaces. As one might expect, React was designed for the needs of Facebook's development team, and is therefore suited particularly well to complex web applications that deal heavily with user interaction and changing data.

In January of 2015, the React team announced a new project: React Native. React Native uses React to target platforms other than the browser, such as iOS and Android, by implementing a so-called "bridge" between Javascript and the host platform. It promises web developers the ability to write real, natively rendering mobile applications, all from the comfort of a Javascript library that they already know and love.

How is this possible? And perhaps more importantly — how can you take advantage of it? In this book, we will start by covering the basics of React Native and how it works. Then, we will explore how to use your existing knowledge of React to build complex mobile applications with React Native, taking advantage of host platform APIs such as geolocation, the camera, and more. By the end, you will be equipped with all the necessary knowledge to deploy your iOS applications to the App Store. Though we will be focusing on iOS-based examples, the principles we will cover apply equally well to React Native for Android and other platforms.

Are you ready to write your own mobile applications using React? Great — let's get started!

How Does React Native Work?

The idea of writing mobile applications in Javascript feels a little odd. How is it possible to use React in a mobile environment? In this section, we will explore the technical underpinnings that enable React Native. We will first need to recall one of React's features, the Virtual DOM, and understand how it relates to React Native for mobile.

The Virtual DOM in React

In React, the Virtual DOM acts as a layer between the developer's description of how things ought to look, and the work done to actually render them onto the page. To render interactive user interfaces in a browser, developers must edit the browser's DOM, or Document Object Model. This is an expensive step, and excessive writes to the DOM have a significant impact on performance. Rather than directly render changes on the page, React computes the necessary changes by using an in-memory version of the DOM. It then re-renders the minimal amount of your application necessary.

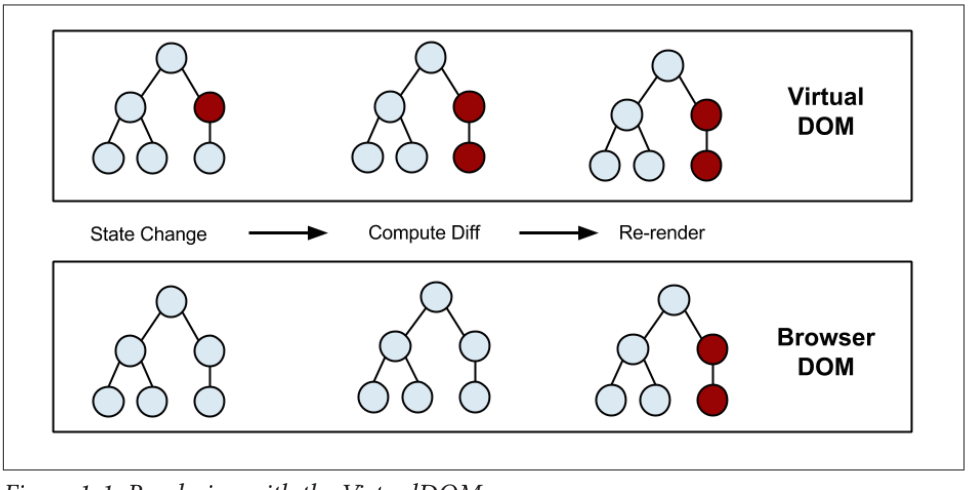


Figure 1-1. Rendering with the VirtualDOM

In the context of React on the web, most developers think of the Virtual DOM primarily as a performance optimization. Indeed, the Virtual DOM was one of React's early claims to fame, and most articles discussing React's benefits mention performance and the Virtual DOM in the same breath. This focus makes a lot of sense, given the context. At the time of React's release, the Virtual DOM approach to rendering gave it a large performance advantage over existing Javascript frameworks, especially when dealing with changing data or otherwise render-intensive applications. (Since then, React's model for DOM updates has started to be adopted by other

frameworks, such as Ember.js's new Glitter engine, though React maintains a strong lead when it comes to performance.)

Extending the Virtual DOM

The Virtual DOM certainly has performance benefits, but its real potential lies in the power of its abstraction. Placing a clean abstraction layer between the developer's code and the actual rendering opens up a lot of interesting possibilities. What if React could render to a target other than the browser DOM? Why should React be limited to the browser? After all, React already "understands" what your application is *supposed* to look like. Surely the conversion of that ideal to actual HTML elements on the page could be replaced by some other step.

During the first two years of React's public existence, some onlookers noticed this intriguing possibility. Netflix, for instance, modified React so that they could render to a huge variety of platforms including televisions and DVD players. Flipboard demonstrated how to render React to the HTML `<canvas>` element. Then, at React Conf in January 2015, Facebook announced a new library, React Native, that does the same for iOS and Android, allowing React to render *natively* to mobile platforms.

There's that word again: *native*. What does it mean to render natively? For React on the web, this means that it renders to the browser DOM. With React Native, native rendering means that React renders using native APIs for creating views.

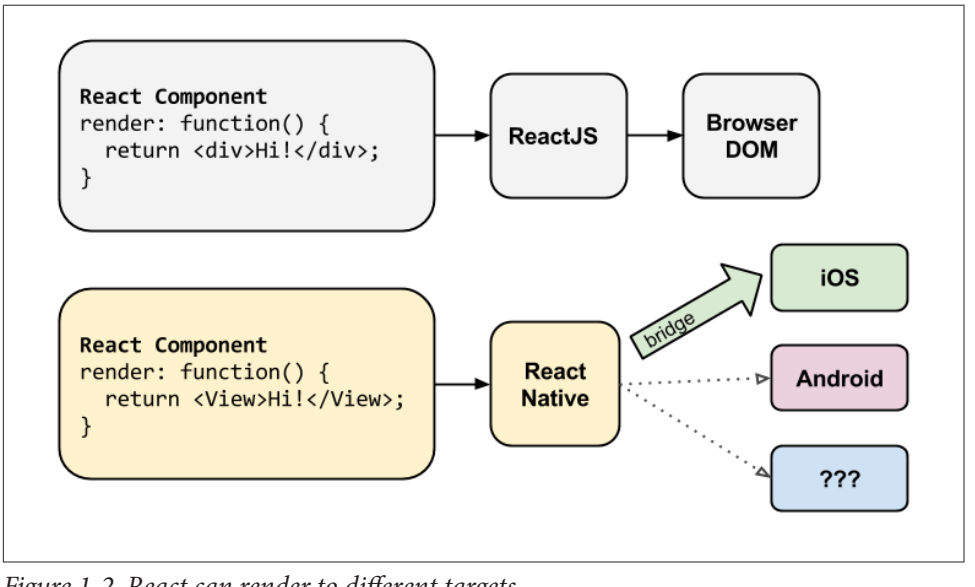


Figure 1-2. React can render to different targets.

This is all possible because of the "bridge," which provides React with an interface into the host platform's native UI elements. React components return markup from their `render` function, which instructs React how to render them. With React for the web, this translates directly to the browser's DOM. For React Native, this markup is translated to suit the host platform, so a `<View/>` might become an iOS-specific `UIView`.

While the projects from Flipboard and Netflix are not affiliated with React Native, the basic idea is the same. Flipboard has essentially built a bridge from React to the HTML5 `<canvas/>` element, while Netflix has built many bridges from React to hundreds of smart TVs, DVD players, and so on. The Virtual DOM gives React the flexibility to swap different rendering logic in and out, so as long as a bridge can be constructed, React can render to virtually any platform. Version 0.14 of React split the library into two separate packages, `react` and `react-dom`, further emphasizing that the core of React is not dependent on any platform-specific rendering logic. (You can read more about this in the relevant blog post.)

The Implications of "Native"

The fact that React Native actually renders using its host platform's standard rendering APIs enables it to stand out from most existing methods of cross-platform application development. Existing frameworks that enable you to write mobile applications using combinations of Javascript, HTML, and CSS typically render using webviews. While this approach can work, it also comes with drawbacks, especially around performance. Additionally, they do not usually have access to the host platform's set of native UI elements. When these frameworks do try to mimic native UI elements, the results usually "feel" just a little off; reverse-engineering all the fine details of things like animations takes an enormous amount of effort.

By contrast, React Native actually translates your markup to real, native UI elements, leveraging existing means of rendering views on whatever platform you are working with. There is no need to reverse-engineer animations or the finer details of a design, because React is consuming the existing platform API. Additionally, React works separately from the main UI thread, so your application can maintain high performance without sacrificing capability. The update cycle in React Native is the same as in React: when `props` or `state` change, React Native re-renders the views. The major difference between React Native and React in the browser is that React Native does this by leveraging the UI libraries of its host platform, rather than using HTML and CSS markup.

The net result of the "native" approach is that you can create applications using React and Javascript, without the usual accompanying trade-offs. React Native for iOS was open-sourced in March 2015, with Android support expected to be released by the end of the year. Any team willing to put in the work to write a "bridge" between React

and any given host platform can add support for other platforms as well, so it will be interesting to see where Native expands to next.

For developers accustomed to working on the web with React, this means that you can write mobile apps with the performance and look-and-feel of a native application, while using the tools that you know and love. This in turn has huge implications for how we think about cross-platform application development, prototyping, and code reuse.

Why Use React Native?

One of the first things people ask after learning about React Native is how it relates to standard, native platform development. How should you make the decision about whether or not to work in React Native, as opposed to traditional, platform-specific development?

Whether or not React Native is a good fit for your needs depends heavily on your individual circumstances and background knowledge. For developers who are comfortable working with Javascript for the web, and React specifically, React Native is obviously exciting — it can leverage your existing skillset to turn you into a mobile developer overnight, without requiring a significant investment in platform-specific languages and development paradigms. On the other hand, if you are already accustomed to traditional mobile development, then React Native's advantages are less immediately apparent. Let's take a look at some of the benefits and considerations that come with using React Native.

Learn once, write anywhere

The React team has been very clear that they are not chasing the "write once, run anywhere" dream of cross-platform application development. This makes sense: an interaction that feels magical on one platform will be completely unsuited to others. Rather than focusing on total code reuse, React Native prioritizes *knowledge* reuse: it allows you to carry your knowledge and skillset across platforms. The React team dubs this "learn once, write anywhere."

"Learn once, write anywhere" means that you can create interesting projects on new platforms, without investing time and energy in learning an entirely new stack. Knowledge of React is enough to be able to work effectively across multiple platforms. This has implications not just for individual programmers, but also for your development teams. For instance, on a small team of engineers, committing to hiring a full-time mobile developer may be difficult. However, if all members of your team can cross-train between web and mobile, the barrier to releasing a mobile application can be drastically lowered. Knowledge transfer empowers your team to be more flexible about your projects.

Of course, while you will not be able to reuse *all* of your code, working in React Native means that you can share plenty of it. Business logic and higher-level abstractions within your application can be recycled across platforms, as we will discuss in detail later. Netflix, for instance, has a React application that runs on literally hundreds of different platforms, and can expand quickly to new devices thanks in part to code reuse. And the Facebook Ads Manager application for Android shares 87% of its codebase with the iOS version, as noted in the React Europe 2015 keynote.

This also means that developers who are already comfortable working on any given platform can still benefit from using React Native. For instance, an iOS developer equipped with knowledge of React Native can write mobile applications that can then be adapted easily for Android.

It's also worth mentioning a perhaps less-obvious bonus: you will be using React for the "write anywhere" part of this equation. React has gained popularity rapidly, and for good reason. It is fast, and flexbile; the component-based approach implicitly encourages you to write clean, modular code that can scale to accommodate complex applications. The facets of React that are most appreciated by the community apply equally well to React Native.

Leveraging the native platform

React Native gives you the benefits of native development, and allows you to take advantage of whatever platform you are developing for. This includes UI elements and animations, as well as platform-specific APIs, such as geolocation or the camera roll.

We can contrast this with existing cross-platform mobile solutions. Many of them allow you to develop inside of thinly-disguised browser components, which then re-implement pieces of the native UI. However, these applications often feel a little "off." It is usually impossible to fully replicate native UI elements, and as a result, aspects of your application such as animations and styling can feel clunky or just less-than-natural. React Native lets you neatly circumvent this problem.

Compared with traditional mobile application development, React Native does not have *quite* the same access to its host platform. There is a slight layer of added complexity to using APIs not yet supported by the React Native core, for instance, and synchronous APIs in particular pose a challenge.

Developer tools

The React Native team has baked strong developer tools and meaningful error messages into the framework, so that working with robust tools is a natural part of your development experience. As a result, React Native is able to remove many of the pain points associated with mobile development.

For instance, because React Native is "just" Javascript, you do not need to rebuild your application in order to see your changes reflected; instead, you can hit CMD+R to refresh your application just as you would any other webpage. Having worked with both Android and iOS development previously, I always cringe to think about how much wasted time was spent waiting to recompile my applications for testing. React Native shrinks that delay from minutes to milliseconds, allowing you to focus on your actual development work.

Additionally, React Native lets you take advantage of intelligent debugging tools and error reporting. If you are comfortable with Chrome or Safari's developer tools, you will be happy to know that you can use them for mobile development, as well. Likewise, you can use whatever text editor you prefer for Javascript editing: React Native does not force you to work in Xcode to develop for iOS, or Android Studio for Android development.

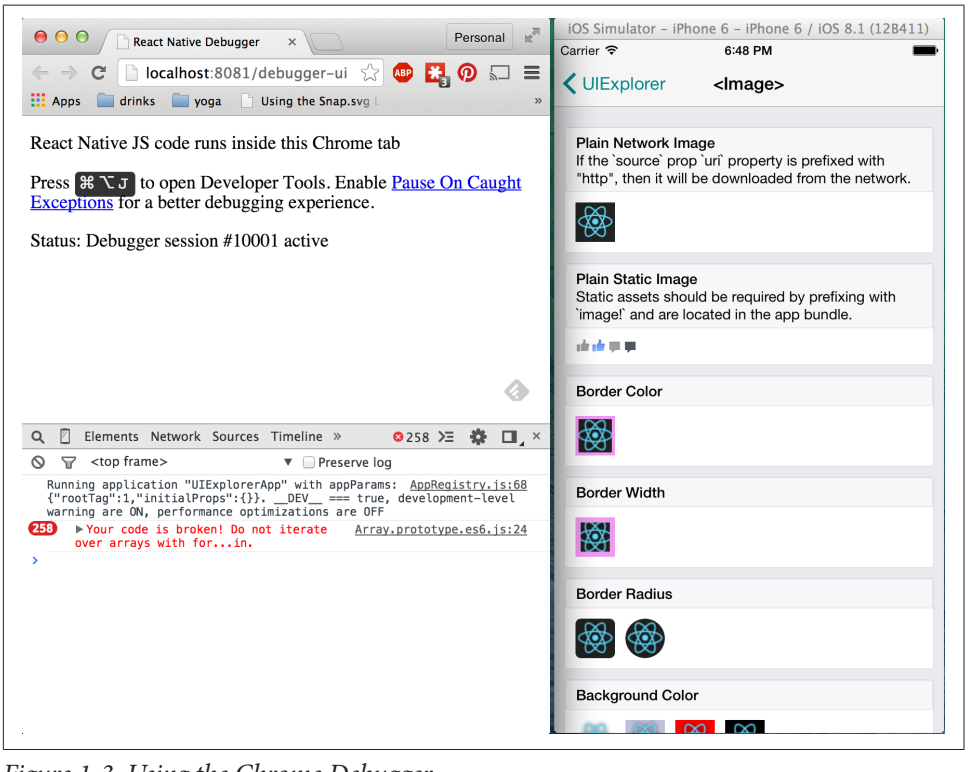


Figure 1-3. Using the Chrome Debugger

Besides the day-to-day improvements to your development experience, React Native also has the potential to positively impact your product release cycle. Apple, for instance, has indicated that they are open to allowing Javascript-based changes to an app's behavior to be loaded over-the-air with no new release cycle necessary.

All of these small perks add up to saving you and your fellow developers time and energy, allowing you to focus on the more interesting parts of your work and be more productive overall.

Using existing platform knowledge

For developers already accustomed to writing traditional native applications on a given platform, moving to React Native does not obsolete your platform-specific knowledge. Far from it; there are a number of circumstances in which this knowledge becomes critical. Occasionally, you will discover that the React Native "bridge" for a given platform does not yet support host platform APIs or features that you will need. For instance, being comfortable working with Objective-C enables you to jump in and add to the React Native "bridge" when necessary, thereby extending React Native's support for iOS.

Community support will likely play a large factor in how React Native evolves on new platforms. Developers who are able to contribute back to the platform by building out the bridge will be better equipped to explore using React Native on new platforms, as well as lay the groundwork for its success.

Risks and Drawbacks

The largest risk that comes with using React Native is related to its maturity level. As a young project, React Native will inevitably contain its share of bugs and unoptimized implementations. This risk is somewhat offset by its community. Facebook, after all, is using React Native in production, and the community contributions have been proceeding at a good tempo. Nevertheless, working with bleeding-edge technology does mean that you can expect a few paper cuts along the way.

Similarly, by using React Native, you are relying heavily on a library that is not necessarily endorsed by the host platform you are working on. Because the Native team may not be able to coordinate with Apple, for instance, we can imagine that new iOS releases could require a scramble to get the React Native library up to speed. For most platforms, however, this should be a relatively minor concern.

The other major drawback is what happens when you encounter a limitation in React Native. Say that you want to to add support for a host platform API not yet built into the React Native core library. If you are a developer who is only comfortable in Javascript, there is a nontrivial learning curve to add this support yourself. We will address exactly this scenario later in the book, and I will demonstrate how to expose Objective-C interfaces to your Javascript code.

Summary

React Native has much to offer for any developer, but a lot will depend on your individual situation. If you are already comfortable working with React, the value proposition of React Native is clear: it gives you a low-investment way to transfer your skills and write for multiple platforms, without the usual drawbacks of cross-platform development frameworks. Even for developers who are comfortable building traditional native mobile applications, there is a lot of appeal here, too. On the other hand, if total platform stability is a must for you, React Native might not be the best choice.

That being said, with React Native applications doing quite well in the iOS App Store already, this is a less risky proposition than you might think. Facebook, at least, is betting heavily on the React Native approach to mobile development. Though the main Facebook mobile applications are still written traditionally, you can see React Native succeeding in the wild by downloading Facebook's Groups app for iOS, which features a blend of React Native and Objective-C code; or the Ads Manager for iOS, which is a 100% React Native application.

Hopefully all of this has left you excited to begin working with React Native! In the next section, we will tackle the information you will need to know in order to work with React Native as opposed to React in the browser.

Native VS React for Web

While React Native is very similar to React for the web, it comes with own considerations, quirks, and behaviors. In this section, we will take a look at some of these differences, in order to set you up for getting your hands dirty with building iOS applications in the next chapter. Our focus will be on using React Native for iOS, but the best practices we will be covering can apply equally well to React Native on other platforms as well.

The React Native Lifecycle

If you are accustomed to working in React, the React lifecycle should be familiar to you. When React runs in the browser, the render lifecycle begins by mounting your React components:

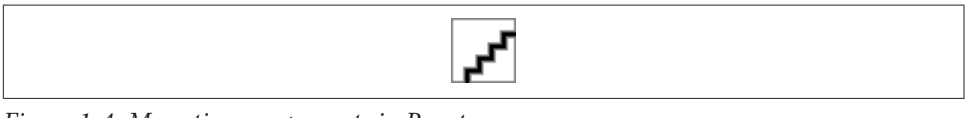


Figure 1-4. Mounting components in React

After that, React handles the rendering and re-rendering of your component as necessary.

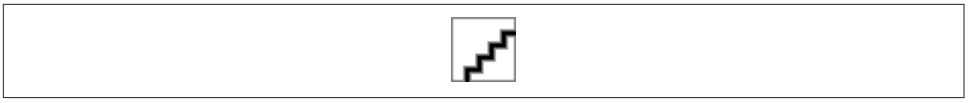


Figure 1-5. Re-rendering components in React

Understanding the rendering stage in React should be fairly straightforward. The developer returns HTML markup from a React component's `render` method, which React then renders directly into the page as necessary.

For React Native, the lifecycle is the same, but the rendering process is slightly different, because React Native depends on the **bridge**. We looked at the bridge briefly earlier in Figure 1-2. The bridge translates APIs and UI elements from their host platform underpinnings (in the case of iOS, Objective-C) into interfaces accessible via Javascript.

The other item worth noting about the React Native lifecycle is that React Native works off of the main UI thread, so that it can perform the necessary computations without interfering with the user's experience. On mobile, there is usually a single UI thread, and time spent performing computations on the UI thread prevents the user from interacting with your application. It is important that React Native stay as unobtrusive as possible, especially in a resource-constrained environment like mobile.

Working with Views in React Native

When writing in React for web, you render normal HTML elements: `<div/>`, `<p/>`, `<span/>`, `<a/>`, and so on. With React Native, all of these elements are replaced by platform-specific React components. The most basic is the cross-platform `<View/>`, a simple and flexible UI element that can be thought of as analogous to the `<div/>`. On iOS, the `<View/>` component renders to a `<UIView/>`.

Table 1-1. Basic Elements

React	React Native
`<div/>`	`<View/>`
`<span/>`	`<TextView/>`
` , ``	`<ListView/>`
`<img/>`	`<ImageView/>`

Other components are platform-specific. For instance, the `<DatePickerIOS />` component (predictably) renders the iOS standard date picker. Here is an excerpt from

the `UIExplorer` sample app, demonstrating an iOS Date Picker. The usage is straightforward, as you would expect from React:

```
<DatePickerIOS
  date={this.state.date}
  mode="date"
  timeZoneOffsetInMinutes={this.state.timeZoneOffsetInHours * 60}
/>
```

This renders to the standard iOS date picker:

Figure 1-6. The iOS Date Picker

Because all of our UI elements are now React components, rather than basic HTML elements like the `<div/>`, you will need to explicitly import each component you wish to use. For instance, we needed to import the `<DatePickerIOS />` component like so:

```
var React = require('react-native');
var {
  DatePickerIOS
} = React;
```

The `UIExplorer` application, which is bundled into the standard React Native examples, allows you to view all of the supported UI elements. I encourage you to examine the various elements included in the `UIExplorer` app. It also demonstrates many styling options and interactions.

Because these components vary from platform to platform, how you structure your React components becomes even more important when working in React Native. In

React for web, we often have a mix of React components: some manage logic and their child components, while other components render raw markup. If you want to reuse code when working in React Native, maintaining a separation between these types of components becomes critical. A React component that renders a `<DatePickerIOS />` element obviously cannot be reused for Android. However, a component that encapsulates the associated *logic* can be reused. Then, the view-component can be swapped out based on your platform.

Styling Native Views

On the web, we style React components using CSS, just as we would any other HTML element. Whether you love it or hate it, CSS is a necessary part of the web. React usually does not affect the way we write CSS. It does make it easier to use (sane, useful) inline styles, and to dynamically build class names based on `props` and `state`, but otherwise React is mostly agnostic about how we handle styles on the web.

Non-web platforms have a wide array of approaches to layout and styling. When we work with React Native, thankfully, we can utilize one standardized approach to styling. Part of the bridge between React and the host platform includes the implementation of a heavily pruned subset of CSS. This narrow implementation of CSS relies primarily on flexbox for layout, and focuses on simplicity rather than implementing the full range of CSS rules. Unlike the web, where CSS support varies across browsers, React Native is able to enforce consistent support of style rules. Much like the various UI elements, you can see many examples of supported styles in the `UIExplorer` application.

React Native also insists on the use of inline styles, which exist as Javascript objects rather than separate stylesheet files. The React team has advocated for this approach before in React for web applications. On mobile, this feels considerably more natural. If you have previously experimented with inline styles in React, the syntax will look familiar to you:

```
// Define a style...
var style = {
        backgroundColor: 'white',
        fontSize: '16px'
};

// ...and then apply it.
var tv = <TextView style={style}> A styled TextView </TextView>;
```

React Native also provides us with some utilities for creating and extending style objects that make dealing with inline styles a more manageable process. We will explore those later.

Does looking at inline styles make you twitch? Coming from a web-based background, this is admittedly a break from standard practices.Working with style objects,

as opposed to stylesheets, takes some mental adjustments, and changes the way you need to approach writing styles. However, in the context of React Native, it is a useful shift. We will be discussing styling best practices and workflow later on. Just try not to be surprised when you see them in use!

JSX and React Native

In React Native, just as in React, we write our views using JSX, combining markup and the Javascript that controls it into a single file. JSX met with strong reactions when React first debuted. For many web developers, the separation of files based on technologies is a given: you keep your CSS, HTML, and Javascript files separate. The idea of combining markup, control logic, and even styling into one language can be confusing.

Of course, the reason for using JSX is that it prioritizes the separation of *concerns* over the separation of technologies. In React Native, this is even more strictly enforced. In a world without the browser, it makes even more sense unify our styles, markup, and behavior in a single file for each component. Accordingly, your `.js` files in React Native are in fact JSX files. If you were using vanilla Javascript when working with React for web, you will want to transition to JSX syntax for your work in React Native.

Thinking about host platform APIs

Perhaps the biggest difference between React for web and React Native is the way we think about host platform APIs. On the web, the issue at hand is often fragmented and inconsistent adoption of standards; still, most browsers support a common core of shared features. With React Native, however, platform-specific APIs play a much larger role in creating an excellent, natural-feeling user experience. There are also many more options to consider. Mobile APIs include everything from data storage, to location services, to accessing hardware such as the camera. As React Native expands to other platforms, we can expect to see other sorts of APIs, too; what would the interface look like between React Native and a virtual reality headset, for instance?

By default, React Native for iOS includes support for many of the commonly used iOS features, and React Native can support any asynchronous native API. We will take a look at many of them throughout this book. The array of options can seem dizzying at first if you are coming from a browser-based background in web development. Happily, React Native makes it straightforward and simple to make use of host platform APIs, so you can experiment freely. Be sure to think about what feels "right" for your target platform, and design with natural interactions in mind.

Inevitably, the React Native bridge will not expose all host platform functionality. If you find yourself in need of an unsupported feature, you have the option of adding it to React Native yourself. Alternatively, chances are good that someone else has done

so already, so be sure to check in with the community to see whether or not support will be forthcoming.

Also worth noting is that utilizing host platform APIs has implications for code reuse. React components that need platform-specific functionality will be platform-specific as well. Isolating and encapsulating those components will bring added flexibility to your application. Of course, this applies for the web, too: if you plan on sharing code between React Native and React, keep in mind that things like the DOM do not actually exist in React Native.

Summary

React Native represents a new and intriguing way of using Javascript to target other platforms. For a web developer familiar with React, it is a tantalizing opportunity to get up-and-running with mobile app development with very little overhead. There's a lot to like about React Native's approach, as well as plenty of unknowns to explore.

In order to fully take advantage of React Native, there is a lot to learn. The good news is that your experience with working with React on the web will give you the foundation you need to be successful with React Native. As we begin writing iOS apps with React Native in the next chapter, I think you will be pleasantly surprised by how at home you feel working on a mobile platform. We will begin by tackling the basics of getting up and running, and learning how to adjust your React best practices to work on mobile. By the last chapter, you will be ready to deploy a robust iOS application to the App Store.

CHAPTER 2

Getting Started

In this chapter, we will cover how to set up your local development environment for working with React Native. Then, we will go through the basics of creating a simple iOS application, which you will then be able to deploy to your own iOS device.

Though this book uses iOS for the examples, most of the content covered applies equally well to React Native for other platforms. If you are working with React Native for Android, or another platform, feel free to skip the environment setup steps. You can install React Native in accordance with the online documentation and then jump directly into the React Native code examples.

Setting Up Your Environment

Setting up your development environment will enable you to follow along with the examples in the book, and will let you write your own applications!

In order to develop for iOS, you will need to purchase or otherwise acquire two things: a Mac, and an iOS developer's license. This applies equally to ordinary iOS development, as well as iOS development using React Native. An iOS developer's license is reasonably priced at $99/year. While you do not need to buy one just yet, without it, you cannot easily deploy your application to a physical device. Nor can you release your work on the App Store.

You will also need to install Xcode, if you have not done so already. Xcode can be installed from the Mac App Store. Happily, while you will need to install Xcode and keep it running, React Native allows you to write code using whichever text editor you normally prefer for your Javascript projects.

Installing React Native

Instructions for installing React Native can be found in the official React Native documentation at facebook.github.io/react-native. The official website will be your most up-to-date reference point for specific installation steps.

You will need to use Homebrew, a common package manager for OS X, in order to install React Native's dependencies. From the command line:

```
brew install node
brew install watchman
brew install flow
```

Then, install the React Native command-line tools:

```
npm install -g react-native-cli
react-native init FirstProject
```

With that, you should be set up to start working on React Native applications.

Creating a New Application

You can use the React Native command line tools to create a new application. This will generate a fresh project with all of the React Native and iOS boilerplate for you.

```
react-native init FirstProject
```

The resulting directory should have the following structure:

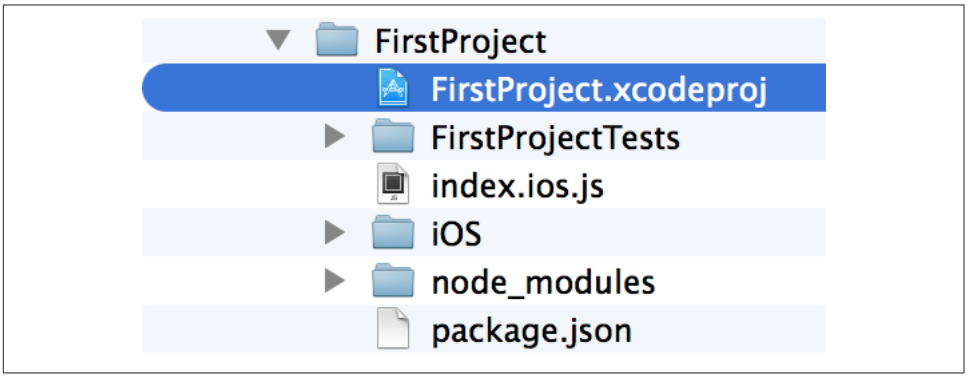


Figure 2-1. File structure in the default project

There are a few things worth noting about the basic file structure here. First, all iOS boilerplate code is located in the `iOS` directory. Your React code is located in the `index.ios.js` file, which is in the project root. Node modules, as usual, can be found in the `node_modules` folder. Finally, the `FirstProject.xcodeproj` file is what you will use to open the project in Github.

If you would prefer, you can download the project from the Github repository for this book.

Running a React Native Application

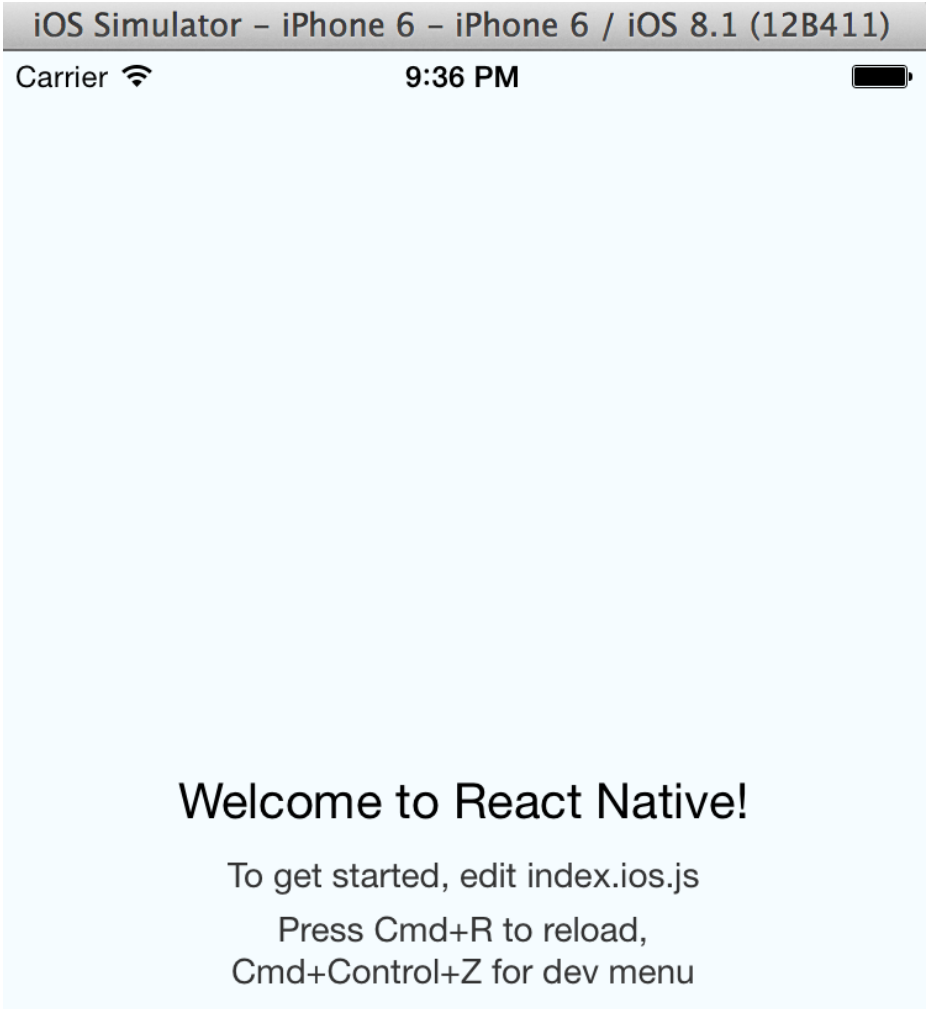
Now, let's try running the application. Open `FirstProject.xcodeproj` in XCode.

When you press "play," the React packager should automatically launch. If it fails to launch, or prints an error, try running `npm install` and `npm start` from the `First Project` directory.

It should look like this:

Figure 2-2. The React packager

Once the packager is ready, it will print `React packager ready`. Then, the iOS simulator will launch with the default application. It should look something like this:



You need the packager running at all times while developing in order to have changes in your code reflected in the app. If the packager crashes, you can restart it by navigating to your project's directory and running `npm start`.

Uploading to Your Device

To upload your React Native application to a physical device, you will need an iOS developer account with Apple. You will then need to generate a certificate and register your device.

If you have not yet purchased a developer's license, navigate to the developer program page and complete your purchase. The enrollment page can be found at *https://developer.apple.com/programs/ios/*.

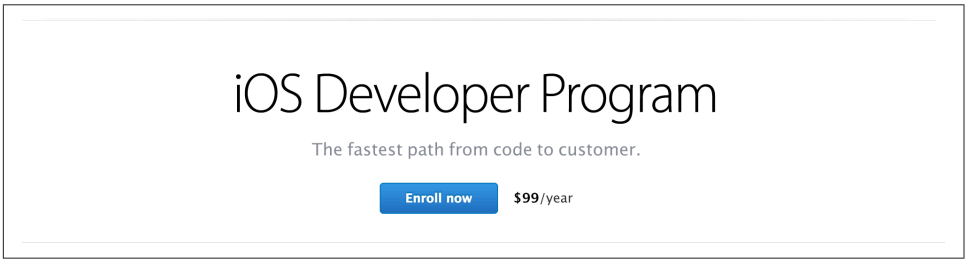


Figure 2-4. The iOS developer program.

After registering with Apple, open Xcode's preferences and add your account.

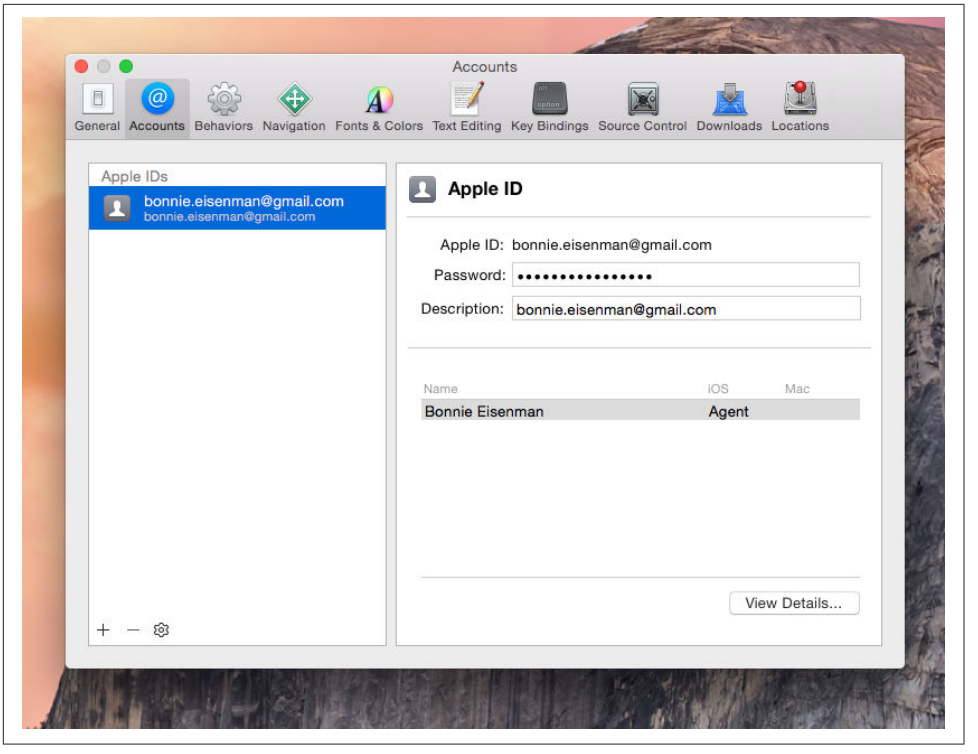


Figure 2-5. Add your account in XCode's preferences pane.

Next, you will need to obtain a certificate for your account. The easiest way to do this is to check the "General" pane in Xcode. You will notice a warning symbol, along with the option to "fix issue". Click that button. XCode should walk you through the next few steps required in order to get a certificate from Apple.

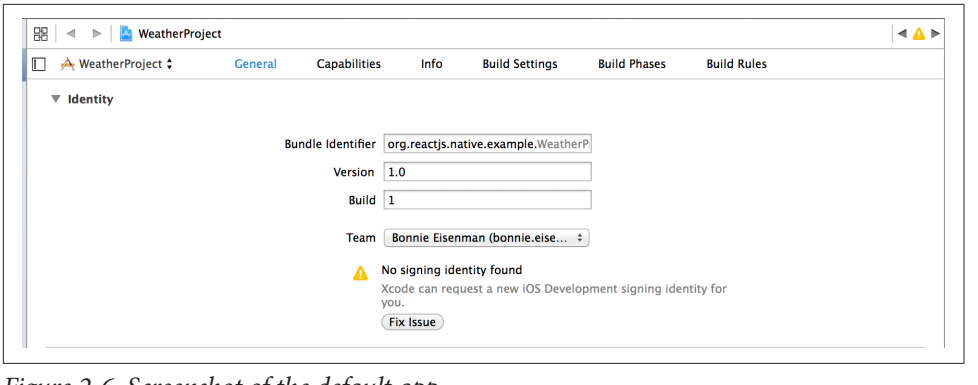


Figure 2-6. Screenshot of the default app.

Having obtained a certificate, you're nearly done. The final step is to log on to developer.apple.com and register your device.

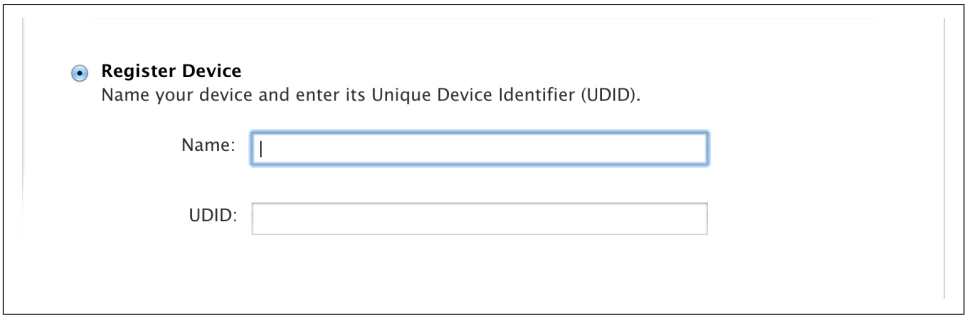


Figure 2-7. Registering your device in the iOS developer member center.

Obtaining your device's UDID is simple. Open iTunes, and select your device. Then, click on the serial number; it should now display the UDID instead, and the UDID will be copied over to your clipboard.

Once you have registered your device with Apple, it should appear in your list of approved devices.

This registration process can also be used later on, if you wish to distribute an early release to other test devices. For individual developers, Apple gives you an allotment of 100 devices per year through the developer program.

Lastly, we need to make a quick change to our code before we can deploy. You will need to alter your `AppDelegate.m` file to include your Mac's IP address instead of localhost. If you do not know how to find your computer's IP address, you can run `ifconfig` and then use the `inet` value under `en0`.

For example, if your IP address was `10.10.12.345`, you should edit the `jsCodeLocation` to look like this:

```
jsCodeLocation = [NSURL URLWithString:@"http://10.10.12.345:8081/index.ios.bundle"];
```

Phew! With all of that out of the way, we can select a physical device as the deploy target in Xcode.

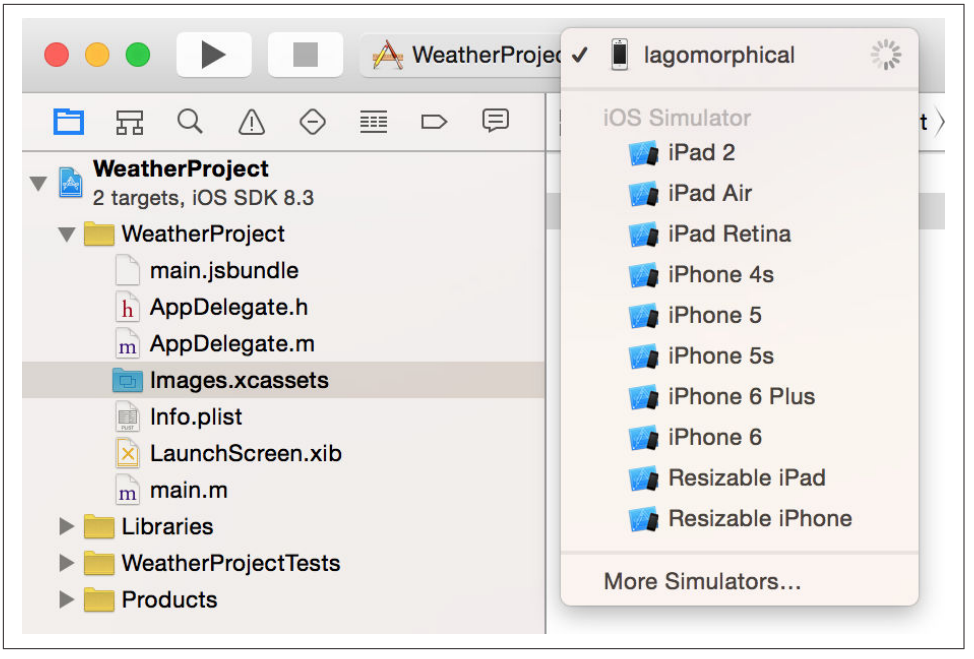


Figure 2-8. Select your iOS device as the deploy target.

Once that is done, click the "play" button. The app should load onto your device, just as it did in the simulator! If you close the app, you will see that it's been loaded onto your home screen, too.

Summary

Creating the React Native equivalent of "hello world" is as easy as running `react-native init HelloWorld`! The React Native command line tools are very helpful for getting started, and React Native does not have many dependencies that you will need to install.

Getting set up with Xcode in order to deploy applications is trickier, but hopefully you made it through along with us. Now that you have done the legwork of registering your device, obtaining a certificate, and so on, in the future, uploading a React Native application to your device will be as simple as pushing "play."

Exploring the Sample Code

Now that you have launched and deployed the default application from the last section, let's figure out how it works. In this section, we will dig into the source code of the default application and explore the structure of a React Native project.

Attaching a component to the view

When a React Native application launches, how does a React component get bound to the view? What determines which component is rendered? We can find the answer inside of `AppDelegate.m`. Notice the following lines:

```
RCTRootView *rootView =
  [[RCTRootView alloc] initWithBundleURL:jsCodeLocation
  moduleName:@"FirstProject"
  launchOptions:launchOptions];
```

The React Native library prefixes all of its classes with `RCT`, meaning that `RCTRootView` is a React Native class. In this case, the `RCTRootView` represents the root React view. The remainder of the boilerplate code in `AppDelegate.m` handles attaching this view to a `UIViewController` and rendering the view to the screen. These steps are analogous to mounting a React component to a DOM node with a call to `React.render`.

For now, the `AppDelegate.m` file contains two things that you ought to know how to modify.

The first is the `jsCodeLocation` line, which we edited earlier in order to deploy to a physical device. As the comments in the generated file explain, the first option is used for development, while the second option is used for deploying with a pre-bundled file on disk. For now, we will leave the first option uncommented. Later, once we prepare to deploy applications to the App Store, we will discuss these two approaches in more detail.

Secondly, the `moduleName` passed to the RCTRootView determines which component will be mounted in the view. This is where you can choose which component should be rendered by your application.

In order to use the `FirstProject` component here, you need to register a React component with the same name. If you open up `index.ios.js`, you'll see that this is accomplished on the last line:

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

This exposes the FirstProject component so that we can use it in `AppDelegate.m`. For the most part, you will not need to modify this boilerplate, but it's good to know that it's there.

Imports in React Native

Let's take a closer look at the `index.ios.js` file. Observe that the `require` statements used are a bit different than normal:

```
var React = require('react-native');
var {
  AppRegistry,
```

```
    StyleSheet,
    Text,
    View,
} = React;
```

There's some interesting syntax going on here. React is required as usual, but what is happening on the next line?

One quirk of working with React Native is that you need to explicitly require every Native-provided module you work with. Things like `<div>` don't simply exist; instead, you need to explicitly import components such as `<View>` and `<Text>`. Library functions such as `Stylesheet` and `AppRegistry` are also explicitly imported using this syntax. Once we start building our own applications, we will explore the other React Native functions that you may need to import.

The FirstProject Component

Let's take a look at the `<FirstProject>` component. This should all look comfortably familiar, since `<FirstProject>` is written just like an ordinary React component. The main difference is its use of `<Text>` and `<View>` components instead of `<div>` and `<span>`, and the use of style objects.

As I mentioned earlier, all styling in React Native is done with style objects rather than stylesheets. The standard method of doing this is by utilizing the `Stylesheet` library. You can see how the style objects are defined towards the bottom of the file. Note that only `<Text>` components can take text-specific styles like `fontSize`, and that all layout logic is handled by flexbox. We will discuss how to build layouts with flexbox at greater length later on.

Summary

The sample application is a good demonstration of the basic functions you will need to create React Native applications. It mounts a React component for rendering, and demonstrates the basics of styling and rendering in React Native. It also gave us a simple way to test our development setup, and try deploying to a real device. However, it's still a very basic application, which is why we'll be building a more full-featured app later in this chapter.

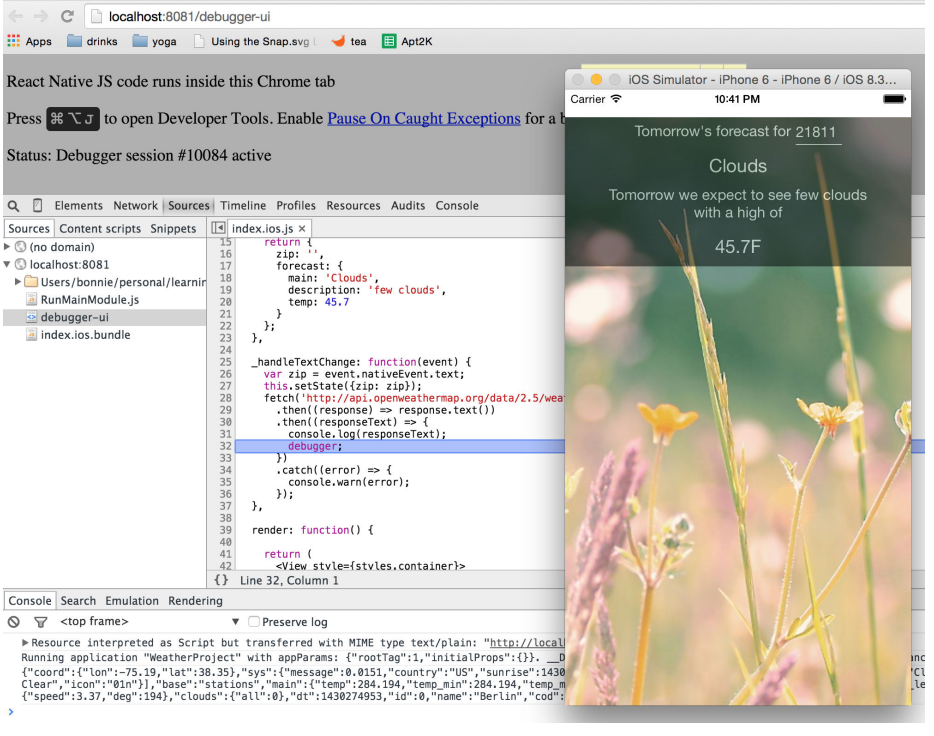
Introduction to the Tools

Before we start building in earnest, let's take a quick look at some useful tools. This section will be by no means exhaustive, but it should give you an idea of what tools will support you while developing with React Native. Some, like the Chrome debugger integration, are Native-specific, while others, such as Flow and Jest, behave indistinguishably from their ReactJS counterparts.

Using the Chrome Debugger

As the default application will tell you, you can access the developer menu by pressing CMD+CTRL+Z. From there, you can turn on the Chrome and Safari debuggers. If you are using a physical device, you can access the developer menu by shaking your device.

You can use the Chrome or Safari debuggers to view the output of `console.log` statements, or to step through your code with a debugger, just like you would when developing for the web. Note that this does cause a minor slowdown to your development process if you like to refresh rapidly — it takes an extra second or two for the debugger to attach to the iOS simulator each time.



Targeting Different Devices

One important thing to remember about mobile development is that different devices have different specifications. Compared to the fragmented web ecosystem of various browsers, operating systems, and screen sizes, mobile is somewhat simpler. Still, it is important to test your application on different devices. For iOS development, it is important to decide if you want to build an iPad or an iPhone application (or a universal application, targeting both families of devices). Even within that separation,

there are important differences; for example, the iPhone 6 has a larger screen than the iPhones that preceded it.

XCode has various device simulators to assist you in testing your application. You can change your deployment target by clicking on the current target in the top-left corner of Xcode:

Type-Checking with Flow

Flow is a Javascript library for static type-checking. It relies on type inference to detect type errors even in unannotated code, and allows you to slowly add type annotations to existing projects.

Running Flow is simple:

```
flow check
```

The default application comes with a `.flowconfig` file, which configures Flow's behavior. If you see many errors related to files in `node_modules`, you may need to add this line to your `.flowconfig` under `[ignore]`:

```
.*/node_modules/.*
```

You should then be able to run `flow check` without seeing any errors:

```
$ flow check
$ Found 0 errors.
```

Feel free to use Flow to assist you as you develop your React Native applications.

Testing with Jest

React Native supports testing of React components using Jest. Jest is a unit testing framework built on top of Jasmine. It provides aggressive automocking of dependencies, and it meshes nicely with React's testing utilities.

To use Jest, you will first need to install it:

```
npm install jest-cli --save-dev
```

Update your `package.json` file to include a `test` script:

```
{
  ...
  "scripts": {
    "test": "jest"
```

```
    }
    ...
}
```

This will run `jest` when you type `npm test`.

Next, create the *`tests`* directory. Jest will recursively search for files in a *`tests`* directory, and run them.

```
mkdir __tests__
```

Now let's create a new file, *`tests`*`/dummy-test.js`, and write our first test:

```
'use strict';

describe('a silly test', function() {
 it('expects true to be true', function() {
   expect(true).toBe(true);
 });
});
```

Now if you run `npm test`, you should see that the test has passed.

Of course, there is much more to testing than this trivial example. Better references can be found in the sample `Movies` app in the React Native repository.

For instance, here is a shortened version of the test file for `getImageSource` in the `Movies` example application:

```
/**
 * Taken from https://github.com/facebook/react-native/blob/master/Examples/Movies/__tests__/getIm
 */

'use strict';

jest.dontMock('../getImageSource');
var getImageSource = require('../getImageSource');

describe('getImageSource', () => {
  it('returns null for invalid input', () => {
    expect(getImageSource().uri).toBe(null);
  });
  ...
});
```

Note that you need to explicitly prevent Jest from mocking files, and then require your dependencies afterwards. If you want to read more about Jest, I recommend starting with their documentation.

Summary

You are free to use or ignore these tools as you like, and chances are you will find some more useful than others. However, they can be great additions to your work-

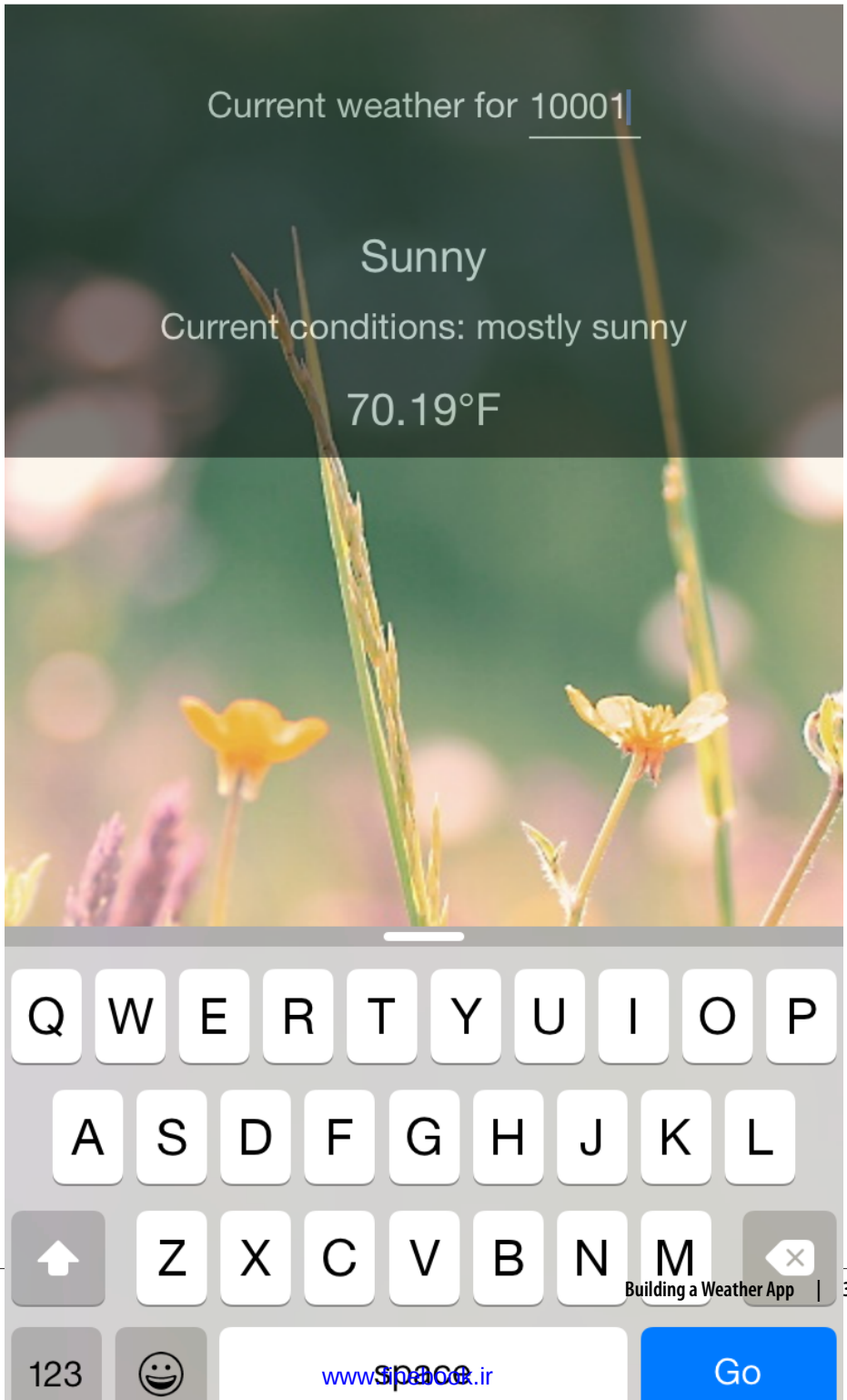
flow, so be sure to try them! In the next section, we will be building a more complex application, and these tools can help you debug and test as you follow along.

Building a Weather App

We will be building off of the sample application to create a weather app. (You can create a new one for this example with `react-native init WeatherProject`.) This will give us a chance to explore how to utilize and combine stylesheets, flexbox, network communication, user input, and images into a useful app that we can then deploy to a device.

This section may feel like a bit of a blur, since we'll be focusing on an overview of these features rather than deep explanations of them. The Weather App will serve as a useful reference in future sections as we discuss these features in more detail. Don't worry if it feels like we're moving quickly!

The final application with have a text field where users can input a zip code. It will then fetch data from the OpenWeatherMap API and display the current weather.



Handling User Input

We want the user to be able to input a zip code and get the forecast for that area, so we need to add a text field for user input. We can start by adding zip code information to our component's initial state:

```
getInitialState: function() {
  return {
    zip: ''
  };
},
```

Then, we should also change one of the TextViews to display `this.state.zip`.

```
<Text style={styles.welcome}>
  You input {this.state.zip}.
</Text>
```

With that out of the way, let's add a `<TextInput>` component. This is a basic component that allows the user to enter text.

```
<TextInput
  returnKeyType='go'
  onSubmitEditing={handleText}/>
```

The `<TextInput>` component's props are documented on the React Native site. Here, we are setting the `returnKeyType` to *go* to indicate that we will perform an action after the user hits the return key. There are plenty of other props that we could set: for instance, changing `keyboardType` to *numeric* would restrict the user to inputting numbers, but then we would not have a return key to use. Other configurable options included turning auto-correction and auto-capitalization on and off, or adding placeholder text.

You can also pass the `<TextInput>` additional callbacks in order to listen to other events, such as `onChange` or `onFocus`, but we do not need them at the moment.

The handleText callback looks like this:

```
var handleText = (event) => {
 console.log(event.nativeEvent.text);
 this.setState({zip: event.nativeEvent.text})
};
```

The console statement is extraneous, but it will allow you to test out the debugger tools if you so desire.

You will also need to update your import statements.

```
var React = require('react-native');
var {
  ...
  TextInput
```

```
  ...
} = React;
```

Now, try running your application using the iOS simulator. It should be ugly and unstyled, but you should be able to successfully submit a zip code and have it be reflected in the `<Text>` component.

If we wanted, we could add some simple input validation here to ensure that the user typed in a five-digit number, but we will skip that for now.

Displaying Data

Now let's work on displaying the forecast for that zip code. We will start by adding some mock data to `getInitialState`:

```
getInitialState() {
  return {
    zip: '',
    forecast: {
      main: 'Clouds',
      description: 'few clouds',
      temp: 45.7
    }
  }
}
```

For sanity's sake, let's also pull the forecast rendering into its own component.

```
var Forecast = React.createClass({
  render: function() {
    return (
      <View>
        <Text style={styles.bigText}>
          {this.props.main}
        </Text>
        <Text style={styles.mainText}>
          Current conditions: {this.props.description}
        </Text>
        <Text style={styles.bigText}>
          {this.props.temp}°F
        </Text>
      </View>
    );
  }
});
```

The `<Forecast>` component just renders some `<Text>` based on its props. The styling is similarly pretty simple. Here's the excerpt from `StyleSheet.create` below:

```
bigText: {
  flex: 2,
  fontSize: 20,
```

```
    textAlign: 'center',
    margin: 10,
    color: '#FFFFFF'
  },
  mainText: {
    flex: 1,
    fontSize: 16,
    textAlign: 'center',
    color: '#FFFFFF'
  }
```

Add the `<Forecast>` component to your app's render method, passing it props based on the `this.state.forecast`. We'll address issues with layout and styling later.

Adding a Background Image

Plain background colors are boring. Let's display a background image to go along with our forecast. Assets such as images need to be added to the Xcode project. Select the `Images.xcassets` folder, and then add a new Image Set. Then, you can drag and drop an image into the set. Make sure the Image Set's name matches the filename, otherwise React Native will have difficulty importing it.

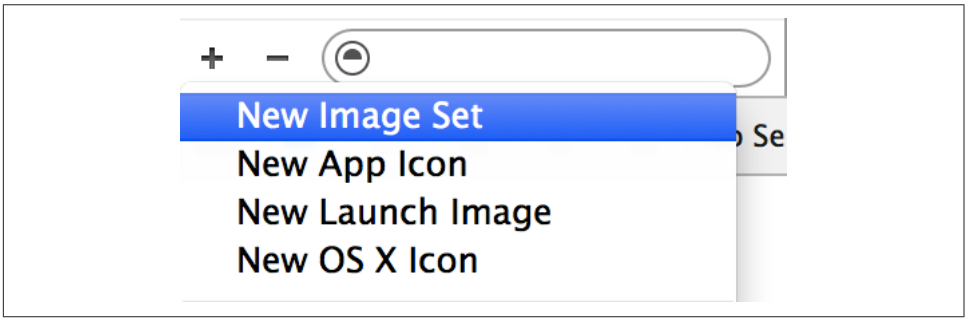


Figure 2-9. Add a new ImageSet.

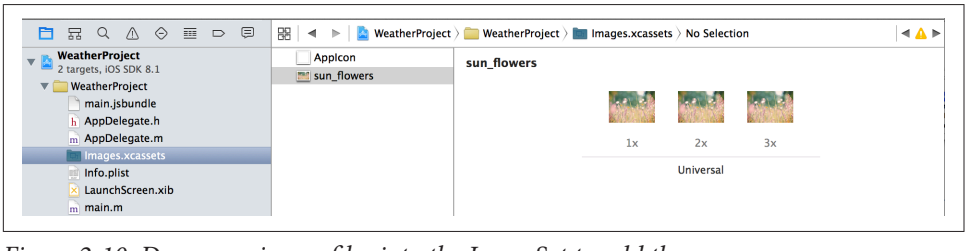


Figure 2-10. Drag your image files into the ImageSet to add them.

The @2x and @3x decorators indicate an image with a resolution of twice and thrice the base resolution, respectively. Because the WeatherApp is designated as a universal

application (meaning one that can run on iPhone or iPad), Xcode gives us the option of uploading images at the various appropriate resolutions.

Now that the image has been imported, let's hop back to our React code. To add a background image, we don't set a `background` property on a `<div>` like we can do on the web. Instead, we use an `<Image>` component as a container.

```
<Image source={require('image!flowers')}
       resizeMode='cover'
       style={styles.backdrop}>
  // Your content here
</Image>
```

The `<Image>` component expects a `source` prop, which we get by using `require`. The call to `require(image!flowers)` will cause React Native to search in Xcode's image assets for a file named `flowers`.

Don't forget to style it with `flexDirection` so that its children render as we'd like them to:

```
backdrop: {
  flex: 1,
  flexDirection: 'column'
}
```

Now let's give the `<Image>` some children:

```
<Image source={require('image!flowers')}
       resizeMode='cover'
       style={styles.backdrop}>
  <View style={styles.overlay}>
   <View style={styles.row}>
     <Text style={styles.mainText}>
       Current weather for
     </Text>
     <View style={styles.zipContainer}>
       <TextInput
         style={[styles.zipCode, styles.mainText]}
         returnKeyType='go'
         onSubmitEditing={this._handleTextChange}/>
     </View>
   </View>
   <Forecast
      main={this.state.forecast.main}
      description={this.state.forecast.description}
      temp={this.state.forecast.temp}/>
 </View>
</Image>
```

You'll notice that I'm using some additional styles that we haven't discussed yet, such as `row`, `overlay`, and the `zipContainer` and `zipCode` styles. You can skip ahead to the end of this section to see the full stylesheet.

Fetching Data from the Web

Next, let's explore using the networking APIs available in React Native. You won't be using jQuery to send AJAX requests from mobile devices! Instead, React Native implements the Fetch API. Usage of the Fetch API is straightforward:

```
fetch('http://www.somesite.com')
  .then((response) => response.text())
  .then((responseText) => {
    console.log(responseText);
  });
```

We will be using the OpenWeatherMap API, which provides us with a simple endpoint that returns the current weather for a given zip code.

To integrate this API, we can change the callback on the `<TextInput>` component to query the OpenWeatherMap API:

```
_handleTextChange: function(event) {
  var zip = event.nativeEvent.text;
  this.setState({zip: zip});
  fetch('http://api.openweathermap.org/data/2.5/weather?q=' + zip + '&units=imperial')
    .then((response) => response.json())
    .then((responseJSON) => {
      console.log(responseJSON); // Take a look at the format, if you want.
      this.setState({
        forecast: {
          main: responseJSON.weather[0].main,
          description: responseJSON.weather[0].description,
          temp: responseJSON.main.temp
        }
      });
    })
    .catch((error) => {
      console.warn(error);
    });
}
```

Note that we want the JSON from the response. The Fetch API is pretty straightforward to work with, so this is all we will need to do.

The other thing that we can do is to remove the placeholder data, and make sure that the forecast does not render if we do not have data yet.

First, clear the mock data from `getInitialState`:

```
getInitialState: function() {
  return {
    zip: '',
    forecast: null
  };
}
```

Then, in the render function, update the rendering logic:

```
var content = null;
if (this.state.forecast !== null) {
  content = <Forecast
              main={this.state.forecast.main}
              description={this.state.forecast.description}
              temp={this.state.forecast.temp}/>;
}
```

Finally, replace your rendered `<Forecast>` component with `{content}` in the `render` function.

Putting it Together

For the final version of the application, I've reorganized the `<WeatherProject>` component's `render` function and tweaked the styles. The main change is to the layout logic, diagrammed here:

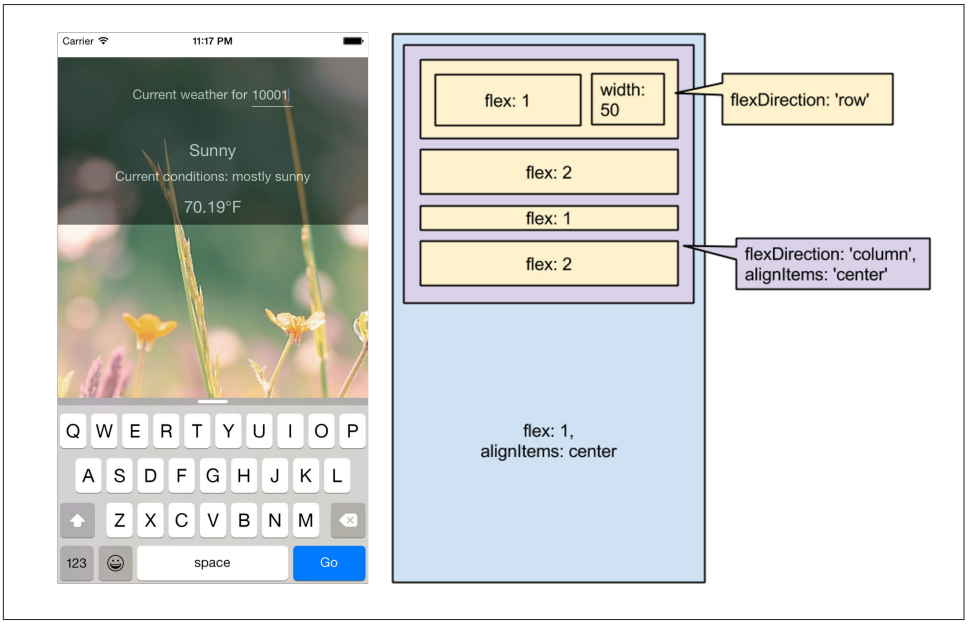


Figure 2-11. Layout of the finished weather application.

OK. Ready to see it all in one place? Here's the finished code in full, including the stylesheets:

```
'use strict';

var React = require('react-native');
var {
```

```
  AppRegistry,
  StyleSheet,
  Text,
  View,
  TextInput,
  Image
} = React;

var Forecast = React.createClass({
  render: function() {
    return (
      <View>
        <Text style={styles.bigText}>
          {this.props.main}
        </Text>
        <Text style={styles.mainText}>
          Current conditions: {this.props.description}
        </Text>
        <Text style={styles.bigText}>
          {this.props.temp}°F
        </Text>
      </View>
    );
  }
});

var WeatherProject = React.createClass({
  getInitialState: function() {
    return {
      zip: '',
      forecast: null
    };
  },

  _handleTextChange: function(event) {
    var zip = event.nativeEvent.text;
    this.setState({zip: zip});
    fetch('http://api.openweathermap.org/data/2.5/weather?q='
      + zip + '&units=imperial')
      .then((response) => response.json())
      .then((responseJSON) => {
        this.setState({
          forecast: {
            main: responseJSON.weather[0].main,
            description: responseJSON.weather[0].description,
            temp: responseJSON.main.temp
          }
        });
      })
      .catch((error) => {
        console.warn(error);
      });
```

```
  },

  render: function() {
    var content = null;
    if (this.state.forecast !== null) {
      content = <Forecast
                  main={this.state.forecast.main}
                  description={this.state.forecast.description}
                  temp={this.state.forecast.temp}/>;
    }
    return (
      <View style={styles.container}>
        <Image source={require('image!flowers')}
               resizeMode='cover'
               style={styles.backdrop}>
          <View style={styles.overlay}>
           <View style={styles.row}>
             <Text style={styles.mainText}>
               Current weather for
             </Text>
             <View style={styles.zipContainer}>
               <TextInput
                 style={[styles.zipCode, styles.mainText]}
                 returnKeyType='go'
                 onSubmitEditing={this._handleTextChange}/>
             </View>
           </View>
           {content}
         </View>
        </Image>
      </View>
    );
  }
});

var baseFontSize = 16;

var styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    paddingTop: 30
  },
  backdrop: {
    flex: 1,
    flexDirection: 'column'
  },
  overlay: {
    paddingTop: 5,
    backgroundColor: '#000000',
    opacity: 0.5,
    flexDirection: 'column',
```

```
    alignItems: 'center'
  },
  row: {
    flex: 1,
    flexDirection: 'row',
    flexWrap: 'nowrap',
    alignItems: 'flex-start',
    padding: 30
  },
  zipContainer: {
    flex: 1,
    borderBottomColor: '#DDDDDD',
    borderBottomWidth: 1,
    marginLeft: 5,
    marginTop: 3
  },
  zipCode: {
    width: 50,
    height: baseFontSize,
  },
  bigText: {
    flex: 2,
    fontSize: baseFontSize + 4,
    textAlign: 'center',
    margin: 10,
    color: '#FFFFFF'
  },
  mainText: {
    flex: 1,
    fontSize: baseFontSize,
    textAlign: 'center',
    color: '#FFFFFF'
  },
  placeHolder: {
    flex: 1
  }
});

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

Now that we're done, try launching the application on either the iOS simulator or on your physical device. How does it look? What would you like to change?

You can view the completed application in the Github repository.

Summary

For our first real application, we've already covered a lot of ground. We introduced a new UI component, `<TextInput>`, and learned how to use it to get information from the user. We demonstrated how to implement basic styling in React Native, as well as how to use images and include assets in our application. Finally, we learned how to

use the React Native networking API to request data from external web sources. Not bad for a first application! And all it took was 150 lines of code.

Hopefully, this has demonstrated how quickly you can build React Native applications with useful features that feel at home on a mobile device.

If you want to extend your application further, here are some things to try:

- Place `<Forecast>` in its own file and `require` it.
- Write Jest tests for the components used here.
- Add more images, and change them based on the forecast.
- Add validation to the zip code field.
- Switch to using a more appropriate keypad for the zip code input.
- Display the 5-day weather forecast.

Once we cover more topics, such as geolocation, we will be able to extend the weather application in even more ways.

Of course, this has been a pretty quick survey. In the next few chapters, we will focus on gaining a deeper understanding of React Native best practices, and look at how to use a lot more features, too!

CHAPTER 3

Components for Mobile

In the last chapter, we built a simple weather app. In doing so, we quickly touched upon the basics of building interfaces with React Native. In this chapter, we will take a closer look at the mobile-based components we use for React Native, and how they compare to basic HTML elements. Mobile interfaces are based on different primitive UI elements than typical webpages, and thus, it's no surprise that we need to use different components.

In this chapter, we will start with a more detailed overview of the most basic components: `<View>`, `<Image>`, and `<Text>`. Then, we will discuss how touch and gestures factor into React Native components, and how to handle touch events. Next, we will cover higher-level components, such as the `<ListView>`, `<TabView>`, and `<NavigatorView>`, which allow you to combine other views into standard mobile interface patterns.

Analogies between HTML Elements and iOS Native Components

When developing for the web, we make use of a variety of basic HTML elements. These include `<div>`, `<span>`, and `<img>`; as well as organizational elements such as `<ol>`, `<ul>`, and `<table>`. (We could include a consideration of element such as `<audio>`, `<svg>`, `<canvas>`, and so on, but we'll ignore them for now.)

When dealing with React Native, we don't use these HTML elements, but we use a variety of components that are nearly analogous to them.

Table 3-1. Analogous HTML and Native Components

HTML	React Native
div	View
img	Image
span, p	Text
ul/ol, li	ListView, child items

While these elements serve roughly the same purposes, they are not interchangeable. Let's take a look at how these components work on mobile with React Native, and how they differ from their browser-based counterparts.

Text for the Web versus React Native

Rendering text is a deceptively basic function; nearly any application will need to render text somewhere. However, text within the context of React Native and mobile development works differently from text rendering for the web.

When working with text in HTML, you can include raw text strings in a variety of elements. Furthermore, you can style them with child tags such as `<strong>` and `<em>`. So, you might end up with an HTML snippet that looks like:

```
<p>The quick <em>brown</em> fox jumped over the lazy <strong>dog</strong>.</p>
```

In React Native, only `<Text>` components may have plain text nodes as child components. (In fact, `<Text>` components are the only components which can accept non-component children.) In other words, this is not valid:

```
<View>
  Text doesn't go here!
</View>
```

Instead, wrap your text in a `<Text>` component.

```
<View>
  <Text>This is okay!</Text>
</View>
```

When dealing with `<Text>` components in React Native, you no longer have access to sub-tags such as `<strong>` and `<em>`, though you can apply styles to achieve similar effects through use of attributes such as `fontWeight` and `fontStyle`. Here's how you might achieve a similar affect by making use of inline styles:

```
<Text>
  The quick <Text style={{fontStyle: "italic"}}>brown</Text> fox
  jumped over the lazy <Text style={{fontWeight: "bold"}}>dog</Text>.
</Text>
```

This approach could quickly become verbose. You'll likely want to create styled components as a sort of shorthand when dealing with text:

```
var styles = StyleSheet.create({
  bold: {
          fontWeight: "bold"
  },
  italic: {
          fontStyle: "italic"
  }
});

var Strong = React.createClass({
  render: function() {
        return (
    <Text style={styles.bold}>
      {this.props.children}
    </Text>);
  }
});

var Em = React.createClass({
  render: function() {
        return (
    <Text style={styles.italic}>
      {this.props.children}
    </Text>);
  }
});
```

Once you have declared these styled components, you can freely make use of styled nesting. Now the React Native version looks quite similar to the HTML version:

```
<Text>
  The quick <Em>brown</Em> fox jumped
  over the lazy <Strong>dog</Strong>.
</Text>
```

Similarly, React Native does not inherently have any concept of header elements (h1, h2, etc), but it's easy to declare your own styled `<Text>` elements and use them as needed.

In general, when dealing with styled text, React Native forces you to change your approach. Style inheritance is limited, so you lose the ability to have default font settings for all text nodes in the tree. One again, Facebook recommends solving this by using styled components:

> You also lose the ability to set up a default font for an entire subtree. The recommended way to use consistent fonts and sizes across your application is to create a component MyAppText that includes them and use this component across your app. You can also use this component to make more specific components like MyAppHeaderText for other kinds of text.
>
> —React Native Documentation

The Text component documentation has more details on this.

You've probably noticed a pattern here: React Native is very opinionated in its preference for the reuse of styled components over the reuse of styles. We'll discuss this further in the next chapter.

The Image Component

If text is *the* most basic element in an application, images are a close contender, for both mobile and for web. When writing HTML and CSS for the web, we include images in a variety of ways: sometimes we use the `<img/>` tag, while at other times we apply images via CSS, such as when we use the `background-image` property. In React Native, we have a similar `<Image>` component, but it behaves a little differently.

The basic usage of the `<Image>` component is straightforward; just set the `source` prop:

```
<Image source={require('image!puppies')} />
```

How does that `require` call work? Where does this resource live? Here's one part of React Native that you'll have to adjust based on which platform you're targeting. On iOS, this means that you'll need to import it into the assets folder within your XCode project. By providing the appropriate @2x and @3x resolution files, you will enable XCode to serve the correct asset file for the correct platform. This is a nice change from web development: the relatively limited possible combinations of screen size and resolution on iOS means that it's easier to create targeted assets.

For React Native on other platforms, we can expect that the `image!` require syntax will point to a similar assets directory.

It's also worth mentioning that it is also possible to include web-based image sources instead of bundling your assets with your application. Facebook does this as one of the examples in the UIExplorer application:

```
<Image source={{uri: 'https://facebook.github.io/react/img/logo_og.png'}}
       style={{width: 400, height: 400}} />
```

When utilizing network resources, you will need to specify dimensions manually.

Downloading images via the network rather than including them as assets has some advantages. During development, for instance, it may be easier to use this approach while prototyping, rather than carefully importing all of your assets ahead of time. It

also reduces the size of your bundled mobile application, so that users needn't download all of your assets. However, it means that instead you'll be relying on the user's data plan whenever they access your application in the future. For most cases, you'll want to avoid using the URI-based method.

If you're wondering about working with the user's own images, we'll discuss the Camera Roll during our exploration of platform-specific APIs.

Because React Native emphasizes a component-based approach, images *must* be included as an `<Image>` component instead of being referenced via styles. For instance, last chapter, we wanted to use an image as a background for our weather application. Whereas in plain HTML and CSS you would likely use the `background-image` property to apply a background image, in React Native you instead use the `<Image>` as a container component, like so:

```
<Image source={require('image!puppies')}>
  {/* Your content here... */}
</Image>
```

Styling the images themselves is fairly straightforward. In addition to applying styles, certain `props` control how the image will be rendered. You'll often make use of the `resizeMode` prop, for instance, which can be set to `resize`, `cover`, or `contain`. The `UIExplorer` app demonstrates this well:

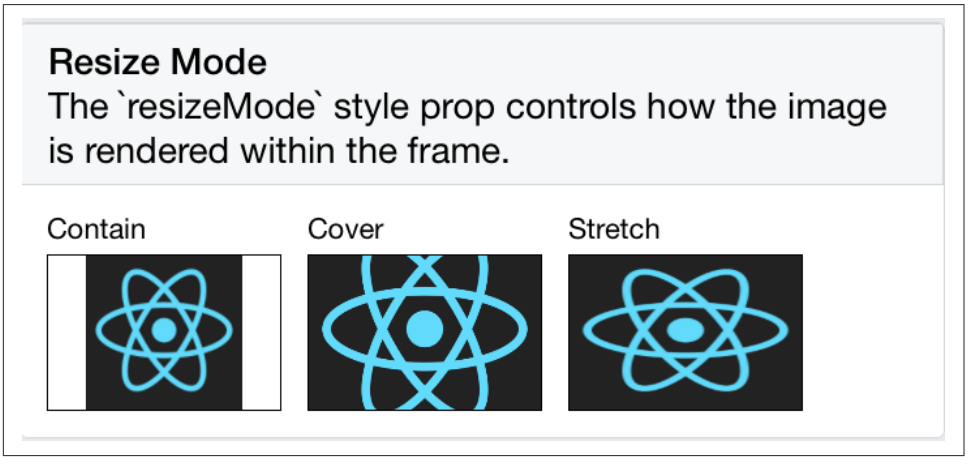


Figure 3-1. The difference between resize, cover, and contain

The `<Image>` component is easy to work with, and very flexible. You will likely make extensive use of it in your own applications.

Working with Touch and Gestures

Web-based interfaces are usually designed with mouse-based controllers in mind. We use things like hover state to indicate interactivity and respond to user interaction. For mobile, unsurprisingly, it's touch that matters. Mobile platforms have their own norms around interactions that you'll want to design for. This varies somewhat from platform to platform: iOS behaves differently from Android, which behaves differently yet again from Windows Phone.

React Native provides a number of APIs for you to leverage as you build touch-ready interfaces. In this section, we'll look at the `<TouchableHighlight>` container component, as well as the lower-level APIs provided by `PanResponder` and the Gesture Responder system.

Using TouchableHighlight

Any interface elements that respond to user touch (think buttons, control elements, and so on) should usually have a `<TouchableHighlight>` wrapper. TouchableHighlight causes an overlay to appear when the view is touched, giving the user visual feedback. This is one of the key interactions that causes a mobile application to feel *native*, as opposed to a mobile-optimized website, where touch feedback is limited. As a general rule of thumb, you should use `<TouchableHighlight/>` anywhere that would be a button or a link on the web.

At its most basic usage, you just need to wrap your component in a `<TouchableHighlight>`, which will add a simple overlay when pressed. The `<TouchableHighlight>` component also gives you hooks for events such as `onPressIn`, `onPressOut`, `onLongPress`, and so on. You could use these, for instance, to build menus that only appear on long presses; and so on.

Here's an example of how we can wrap a component in a `<TouchableHighlight>` in order to give the user feedback:

```
<TouchableHighlight
  onPressIn={this._onPressIn}
  onPressOut={this._onPressOut}
  style={styles.touchable}>
    <View style={styles.button}>
      <Text style={styles.welcome}>
        {this.state.pressing ? 'EEK!' : 'PUSH ME'}
      </Text>
    </View>
</TouchableHighlight>
```

When the user taps the button, an overlay appears, and the text changes.

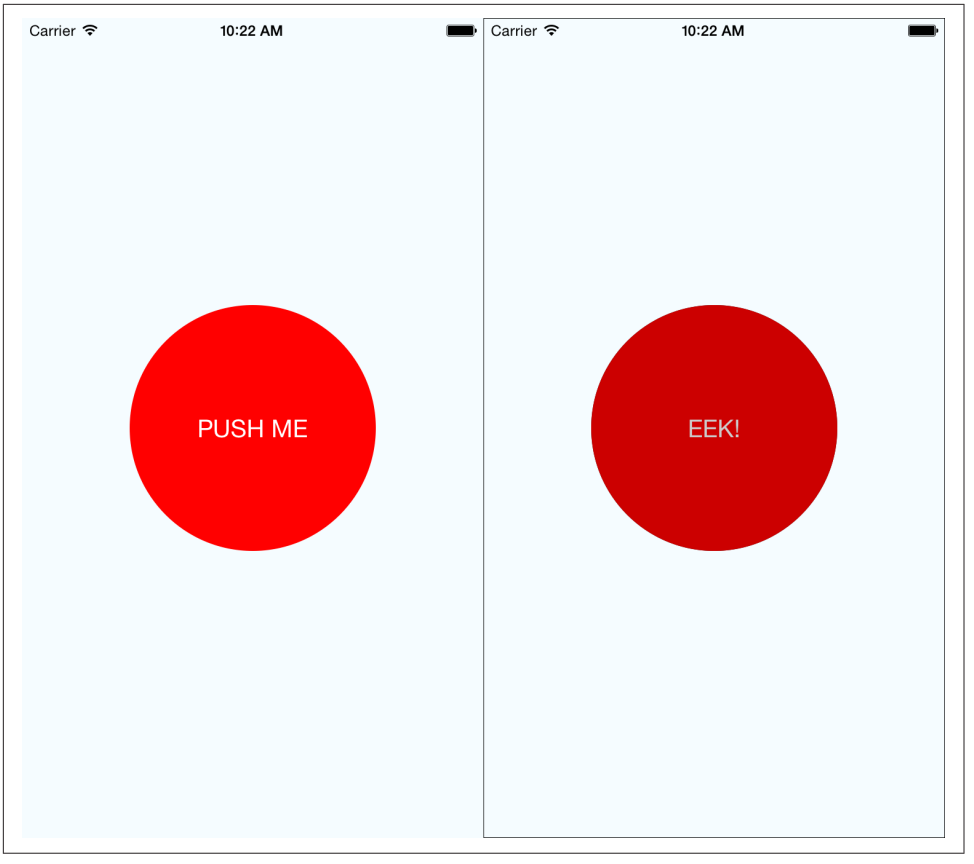


Figure 3-2. Left: unpressed state. Right: pressed state, with highlight.

This is a contrived example, but it illustrates the basic interactions that make a button "feel" touchable on iOS. The overlay is a key piece of feedback that informs the user that an element can be pressed. Note that in order to apply the overlay, we don't need to apply any logic to our styles; the `<TouchableHighlight>` handles the logic of that for us.

Here's the full code for this button component:

```
'use strict';

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

```
var Button = React.createClass({
  getInitialState: function() {
    return {pressing: false};
  },
  _onPressIn: function() {
    this.setState({pressing: true});
  },
  _onPressOut: function() {
    this.setState({pressing: false});
  },
  render: function() {
    return (
      <View style={styles.container}>
        <TouchableHighlight onPressIn={this._onPressIn}
                            onPressOut={this._onPressOut}
                            style={styles.touchable}>
          <View style={styles.button}>
            <Text style={styles.welcome}>
              {this.state.pressing ? 'EEK!' : 'PUSH ME'}
            </Text>
          </View>
        </TouchableHighlight>
      </View>
    );
  }
});

// Styles
var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
    color: '#FFFFFF'
  },
  touchable: {
    borderRadius: 100
  },
  button: {
    backgroundColor: '#FF0000',
    borderRadius: 100,
    height: 200,
    width: 200,
    justifyContent: 'center'
  },
```

```
});

module.exports = Button;
```

Try editing this button to respond to other events, by using hooks like `onPress` and `onLongPress`. The best way to get a sense for how these events map onto user interactions is to experiment using a real device.

The Gesture Responder System

What if you want to do more than just make things "tappable"? React Native also exposes two APIs for custom touch handling: `GestureResponder` and `PanResponder`. GestureResponder is a lower-level API, while PanResponder provides a useful abstraction. We'll start by looking at how the GestureResponder system works, because it's the basis for `PanResponder`'s API.

Touch on mobile is fairly complicated. Most mobile platforms support multitouch, which means that there can be multiple touch points active on the screen at once. (Not all of these are necessarily fingers, either; think about the difficulty of, e.g., detecting the user's palm resting on the corner of the screen.) Additionally, there's the issue of which view should handle a given touch. This problem is similar to how mouse events are processed on the web, and the default behavior is also similar: the topmost child handles the touch event by default. With React Native's gesture responder system, however, we can override this behavior if we so choose.

The *touch responder* is the view which handles a given touch event. In the previous section, we saw that the `<TouchableHighlight>` component acts as a touch responder. We can cause our own components to become the touch responder, too. The lifecycle by which this process is negotiated is a little complicated. A view which wishes to obtain touch responder status should implement four props:

- `View.props.onStartShouldSetResponder`
- `View.props.onMoveShouldSetResponder`
- `View.props.onResponderGrant`
- `View.props.onResponderReject`

These then get invoked according to the following flow, in order to determine if the view will receive responder status:

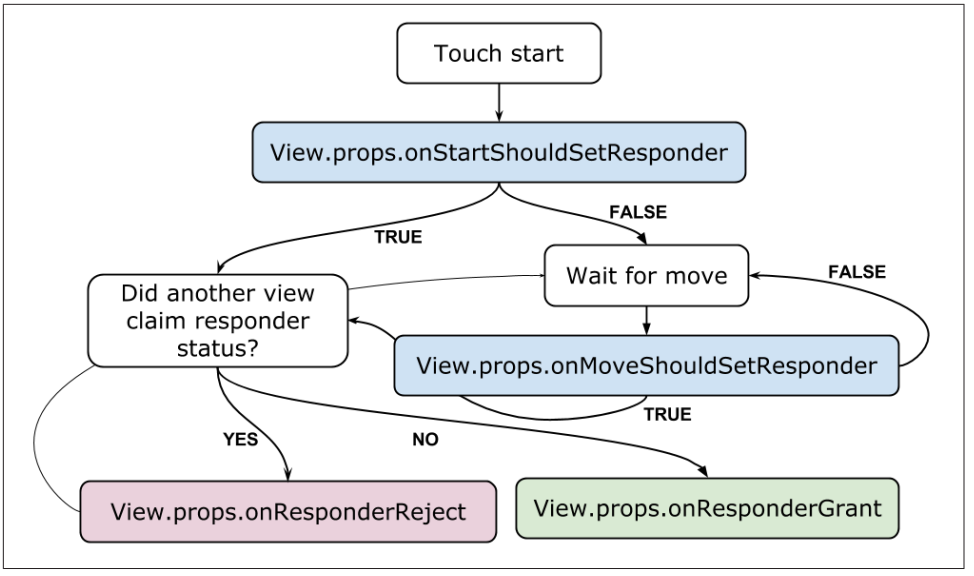


Figure 3-3. Obtaining touch responder status

Yikes, that looks complicated! Let's tease this apart. First, a touch event has three main lifecycle stages: *start*, *move*, and *release*. (These correspond to mouseDown, mouseMove, and mouseUp in the browser.) A view can request to be the touch responder during the *start* or the *move* phase. This behavior is specified by `onStartShouldSetResponder` and `onMoveShouldSetResponder`. When one of those functions returns `true`, the view attempts to claim responder status.

After a view has attempted to claim responder status, its attempt may be *granted* or *rejected*. The appropriate callback — either `onResponderGrant` or `onResponderReject` — will be invoked.

The responder negotiation functions are called in a bubbling pattern. If multiple views attempt to claim responder status, the deepest component will become the responder. This is typically the desired behavior; otherwise, you would have difficulty adding touchable components such as buttons to a larger view. If you want to override this behavior, parent components can make use of `onStartShouldSetResponderCapture` and `onMoveShouldSetResponderCapture`. Returning `true` from either of these will prevent a component's children from becoming the touch responder.

After a view has successfully claimed touch responder status, its relevant event handlers may be called. Here's the excerpt from the Gesture Responder documentation:

- `View.props.onResponderMove`

  — The user is moving their finger

- `View.props.onResponderRelease`
  — Fired at the end of the touch, ie "touchUp"
- `View.props.onResponderTerminationRequest`
  — Something else wants to become responder. Should this view release the responder? Returning true allows release
- `View.props.onResponderTerminate`
  — The responder has been taken from the View. Might be taken by other views after a call to onResponderTerminationRequest, or might be taken by the OS without asking (happens with control center/ notification center on iOS)

Most of the time, you will primarily be concerned with `onResponderMove` and `onResponderRelease`.

All of these methods receive a synthetic touch event object, which adheres to the following format (again, excerpted from the documentation):

- `changedTouches` - Array of all touch events that have changed since the last event
- `identifier` - The ID of the touch
- `locationX` - The X position of the touch, relative to the element
- `locationY` - The Y position of the touch, relative to the element
- `pageX` - The X position of the touch, relative to the screen
- `pageY` - The Y position of the touch, relative to the screen
- `target` - The node id of the element receiving the touch event
- `timestamp` - A time identifier for the touch, useful for velocity calculation
- `touches` - Array of all current touches on the screen

You can make use of this information when deciding whether or not to respond to a touch event. Perhaps your view only cares about two-finger touches, for example.

This is a fairly low-level API; if you want to detect and respond to gestures in this way, you will need to spend a decent amount of time tuning the correct parameters and figuring out which values you should care about. In the next section, we will take a look at `PanResponder`, which supplies a somewhat higher-level interpretation of user gestures.

PanResponder

Unlike `<TouchableHighlight>`, `PanResponder` is not a component, but rather a class provided by React Native. It provides a slightly higher-level API than the basic events

returned by the Gesture Responder system, while still providing access to those raw events. A `PanResponder` `gestureState` object gives you access to the following, in accordance with the PanResponder documentation:

- `stateID` - ID of the gestureState- persisted as long as there at least one touch on screen
- `moveX` - the latest screen coordinates of the recently-moved touch
- `moveY` - the latest screen coordinates of the recently-moved touch
- `x0` - the screen coordinates of the responder grant
- `y0` - the screen coordinates of the responder grant
- `dx` - accumulated distance of the gesture since the touch started
- `dy` - accumulated distance of the gesture since the touch started
- `vx` - current velocity of the gesture
- `vy` - current velocity of the gesture
- `numberActiveTouches` - Number of touches currently on screeen

As you can see, in addition to raw position data, a `gestureState` object also includes information such as the current velocity of the touch and the accumulated distance.

To make use of `PanResponder` in a component, we need to create a `PanResponder` object and then attach it to a component in the `render` method.

Creating a PanResponder requires us to specify the proper handlers for `PanResponder` events:

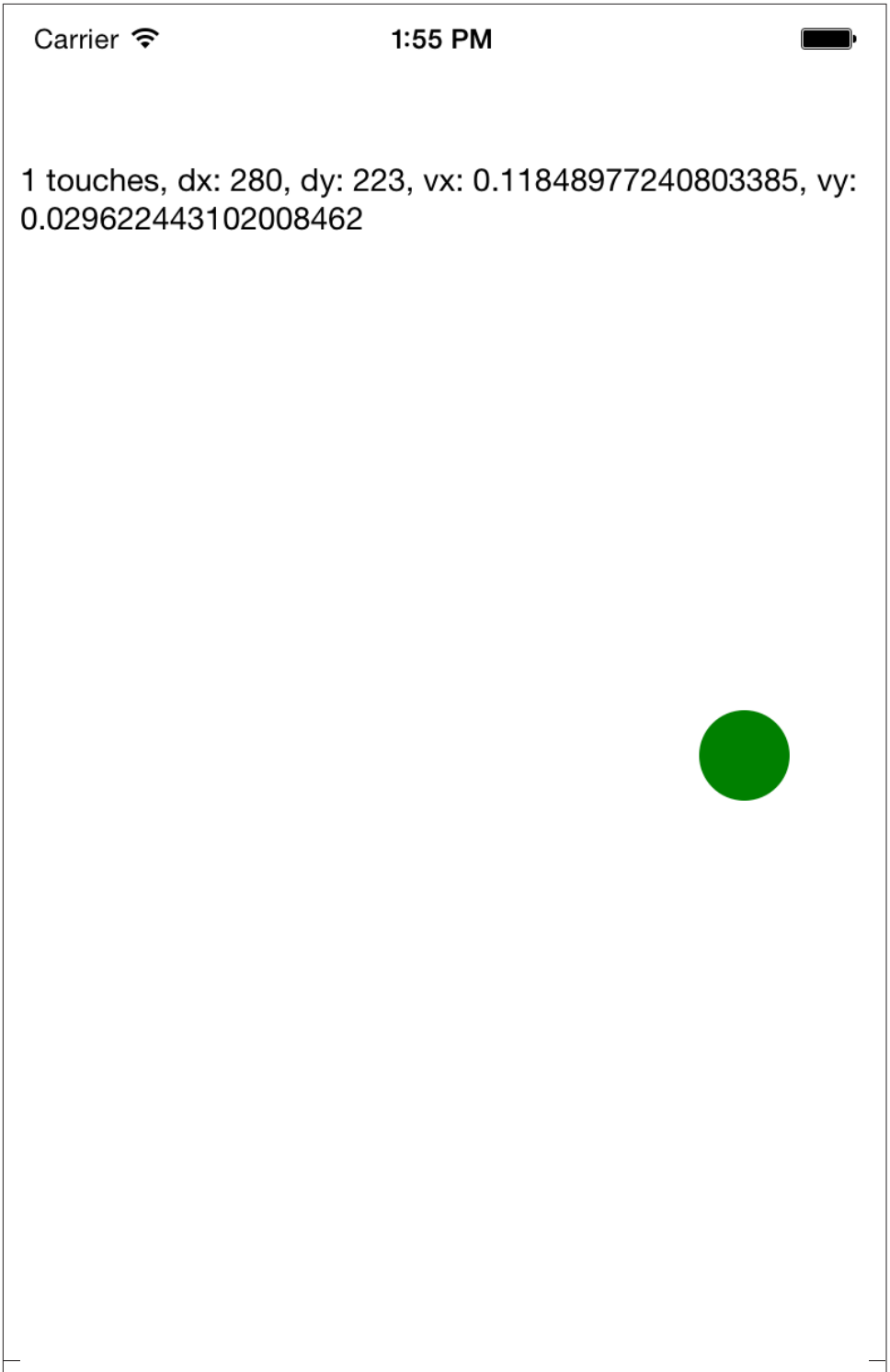
```
this._panResponder = PanResponder.create({
  onStartShouldSetPanResponder: this._handleStartShouldSetPanResponder,
  onMoveShouldSetPanResponder: this._handleMoveShouldSetPanResponder,
  onPanResponderGrant: this._handlePanResponderGrant,
  onPanResponderMove: this._handlePanResponderMove,
  onPanResponderRelease: this._handlePanResponderEnd,
  onPanResponderTerminate: this._handlePanResponderEnd,
});
```

Then, we use spread syntax to attach the `PanResponder` to the view in the component's `render` method.

```
render: function() {
  return (
    <View
      {...this._panResponder.panHandlers}>
      { /* View contents here */ }
    </View>
  );
}
```

After this, the handlers that you passed to the `PanResponder.create` call will be invoked during the appropriate move events, if the touch originates within this view.

Here's a modified version of the PanResponder example code provided by React Native. This version listens to touch events on the container view, as opposed to just the circle, and so that the values are printed to the screen as you interact with the application. If you plan on implementing your own gesture recognizers, I suggest experimenting with this application on a real device, so that you can get a feel for how these values respond.



```
// Adapted from https://github.com/facebook/react-native/blob/master/Examples/UIExplorer/PanRespon

'use strict';

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

var CIRCLE_SIZE = 40;
var CIRCLE_COLOR = 'blue';
var CIRCLE_HIGHLIGHT_COLOR = 'green';

var PanResponderExample = React.createClass({

  statics: {
    title: 'PanResponder Sample',
    description: 'Basic gesture handling example',
  },

  _panResponder: {},
  _previousLeft: 0,
  _previousTop: 0,
  _circleStyles: {},
  circle: (null : ?{ setNativeProps(props: Object): void }),

  getInitialState: function() {
    return {
      numberActiveTouches: 0,
      moveX: 0,
      moveY: 0,
      x0: 0,
      y0: 0,
      dx: 0,
      dy: 0,
      vx: 0,
      vy: 0,
    }
  },

  componentWillMount: function() {
    this._panResponder = PanResponder.create({
      onStartShouldSetPanResponder: this._handleStartShouldSetPanResponder,
      onMoveShouldSetPanResponder: this._handleMoveShouldSetPanResponder,
      onPanResponderGrant: this._handlePanResponderGrant,
      onPanResponderMove: this._handlePanResponderMove,
      onPanResponderRelease: this._handlePanResponderEnd,
      onPanResponderTerminate: this._handlePanResponderEnd,
    });
```

```
    this._previousLeft = 20;
    this._previousTop = 84;
    this._circleStyles = {
      left: this._previousLeft,
      top: this._previousTop,
    };
  },

  componentDidMount: function() {
    this._updatePosition();
  },

  render: function() {
    return (
      <View
        style={styles.container}
        {...this._panResponder.panHandlers}>
        <View
          ref={(circle) => {
            this.circle = circle;
          }}
          style={styles.circle}
        />
        <Text>{this.state.numberActiveTouches} touches, dx: {this.state.dx},
        dy: {this.state.dy}, vx: {this.state.vx}, vy: {this.state.vy}</Text>
      </View>
    );
  },

  _highlight: function() {
    this.circle && this.circle.setNativeProps({
      backgroundColor: CIRCLE_HIGHLIGHT_COLOR
    });
  },

  _unHighlight: function() {
    this.circle && this.circle.setNativeProps({
      backgroundColor: CIRCLE_COLOR
    });
  },

  _updatePosition: function() {
    this.circle && this.circle.setNativeProps(this._circleStyles);
  },

  _handleStartShouldSetPanResponder: function(e: Object, gestureState: Object): boolean {
    // Should we become active when the user presses down on the circle?
    return true;
  },

  _handleMoveShouldSetPanResponder: function(e: Object, gestureState: Object): boolean {
    // Should we become active when the user moves a touch over the circle?
```

```
    return true;
  },

  _handlePanResponderGrant: function(e: Object, gestureState: Object) {
    this._highlight();
  },

  _handlePanResponderMove: function(e: Object, gestureState: Object) {
    this.setState({
      stateID: gestureState.stateID,
      moveX: gestureState.moveX,
      moveY: gestureState.moveY,
      x0: gestureState.x0,
      y0: gestureState.y0,
      dx: gestureState.dx,
      dy: gestureState.dy,
      vx: gestureState.vx,
      vy: gestureState.vy,
      numberActiveTouches: gestureState.numberActiveTouches
    });

    this._circleStyles.left = this._previousLeft + gestureState.dx;
    this._circleStyles.top = this._previousTop + gestureState.dy;
    this._updatePosition();
  },
  _handlePanResponderEnd: function(e: Object, gestureState: Object) {
    this._unHighlight();
    this._previousLeft += gestureState.dx;
    this._previousTop += gestureState.dy;
  },
});

var styles = StyleSheet.create({
  circle: {
    width: CIRCLE_SIZE,
    height: CIRCLE_SIZE,
    borderRadius: CIRCLE_SIZE / 2,
    backgroundColor: CIRCLE_COLOR,
    position: 'absolute',
    left: 0,
    top: 0,
  },
  container: {
    flex: 1,
    paddingTop: 64,
  },
});

module.exports = PanResponderExample;
```

Choosing How to Handle Touch

How should you decide when to use the touch and gesture APIs discussed in this section? It depends on what you want to build.

In order to provide the user with basic feedback, and indicate that something is "tappable," like a button, use the `<TouchableHighlight>` component.

In order to implement your own, custom touch interfaces, use either the raw Gesture Responder system, or a `PanResponder`. Chances are that you will almost always prefer the `PanResponder` approach, because it also gives you access to the simpler touch events provided by the Gesture Responder system. If you are designing a game, or an application with an unusual interface, you'll need to spend some time building out the interactions you want by using these APIs.

For many applications, you won't need to implement any custom touch handling with either the Gesture Responder system or the `PanResponder`. In the next section, we'll look at some of the higher-level components which implement common UI patterns for you.

Working with Organizational Components

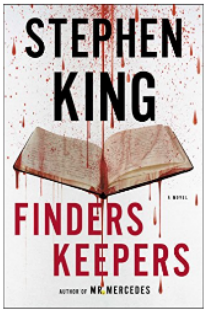
In this section, we're going to look at organizations components which you can use to control general flow within your application. This includes the TabView, NavigatorView, and ListView, which all implement some of the most common mobile interaction and navigational patterns. Once you have planned out your application's navigational flow, you'll find that these components are very helpful in making your application a reality.

Using the ListView

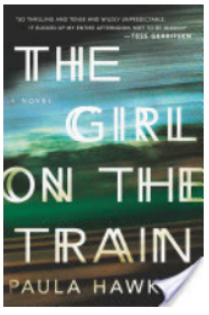
Let's start by using the `<ListView>` component. In this section, we are going to build an app that displays the New York Times Bestseller's List and lets us view data about each book. If you'd like, you can grab your own API token from the NYTimes. Otherwise, use the API token included in the sample code.

Carrier 8:33 PM

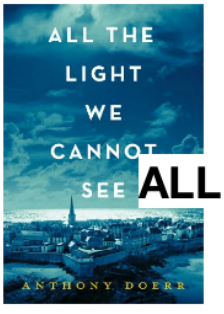
Bestsellers in Hardcover Fiction



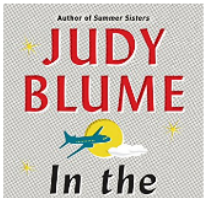
Stephen King
FINDERS KEEPERS



Paula Hawkins
THE GIRL ON THE TRAIN

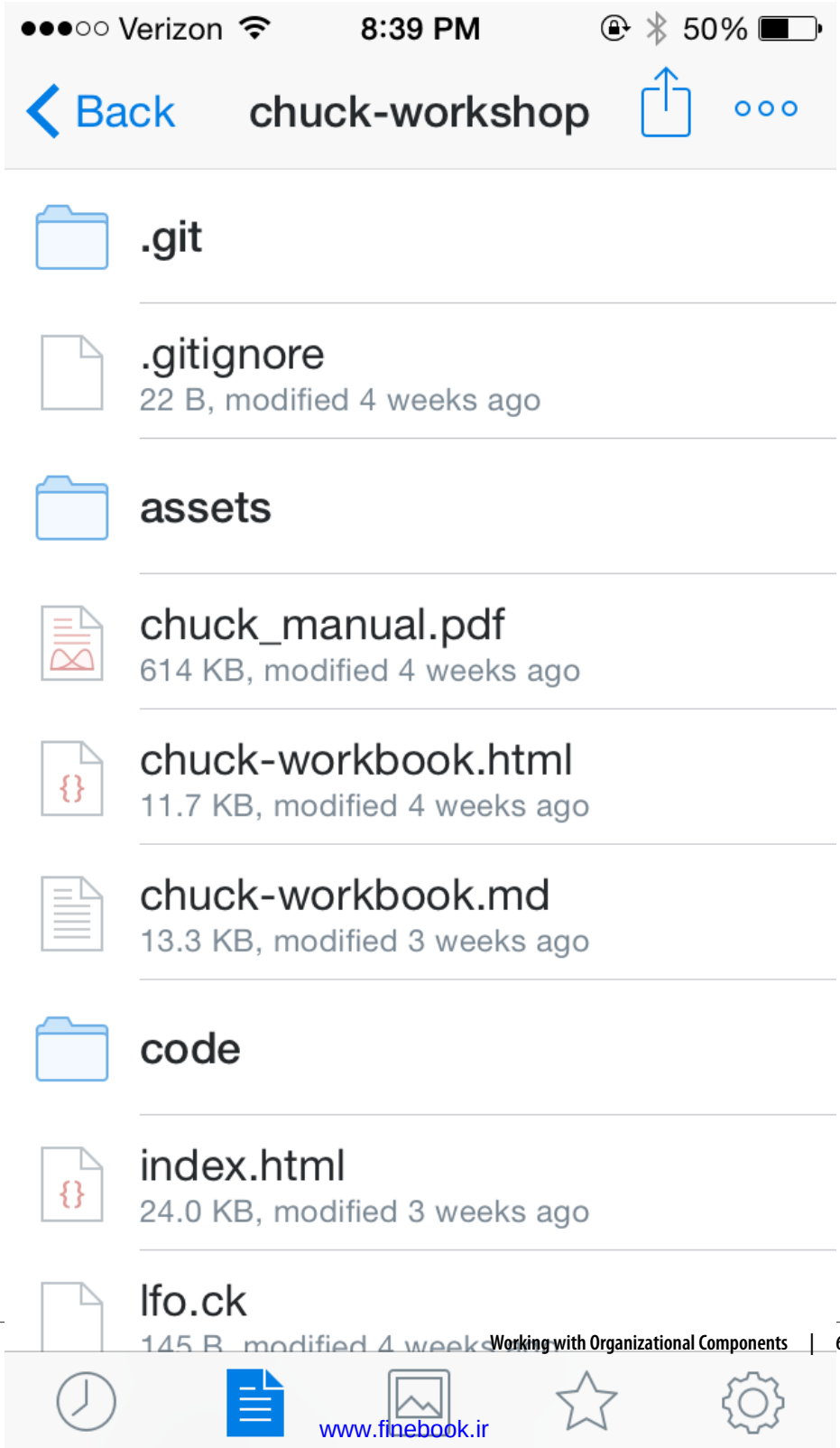


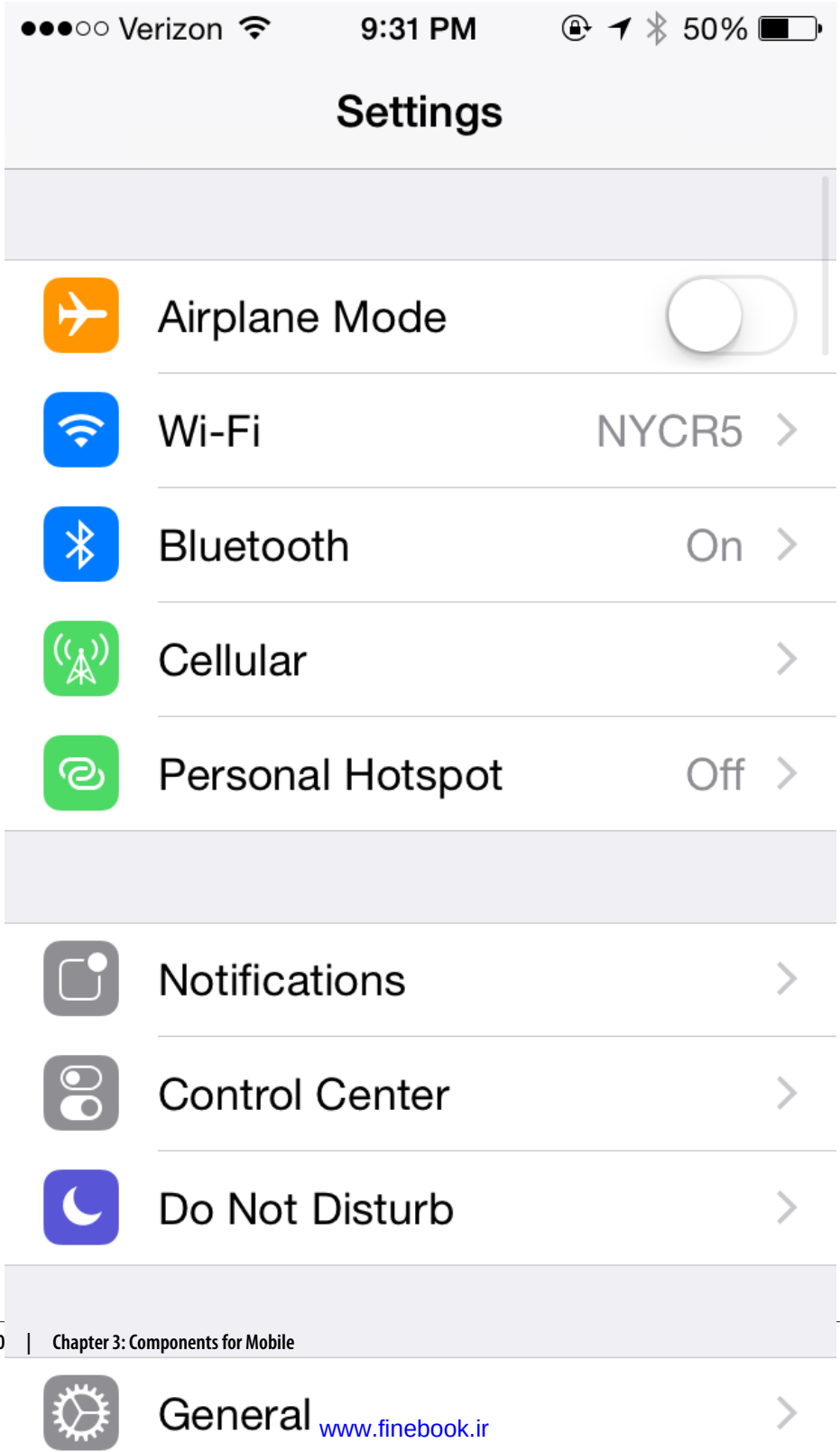
Anthony Doerr
ALL THE LIGHT WE CANNOT SEE



Judy Blume
IN THE UNLIKELY EVENT

Lists are extremely useful for mobile development, and you will notice that many mobile user interfaces feature them as a central element. A `<ListView>` is literally just a list of views, optionally with special views for section dividers, headers, or footers. For example, here are a few ListViews as seen in the Dropbox, Twitter, and iOS Settings apps.





ListViews are a good example of where React Native shines, because it can leverage its host platform. On mobile, the native ListView element is usually highly optimized so that rendering is smooth and stutter-free. If you expect to render a very large number of items in your ListView, you should try to keep the child views relatively simple, to try and reduce stutter.

The basic React Native ListView component requires two props: `dataSource` and `renderRow`. `dataSource` is, as the name implies, a source of information about the data that needs to be rendered. `renderRow` should return a component based on the data from one element of the `dataSource`.

This basic usage is demonstrated in `SimpleList.js`. We'll start by adding a `dataSource` to our `<SimpleList>` component. A `ListView.DataSource` needs to implement the `rowHasChanged` method. Here's a simple example:

```
var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
```

To set the actual contents of a `dataSource`, we use `cloneWithRows`. Let's return the `dataSource` in our `getInitialState` call.

```
getInitialState: function() {
  var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
  return {
    dataSource: ds.cloneWithRows(['a', 'b', 'c', 'a longer example', 'd', 'e'])
  };
}
```

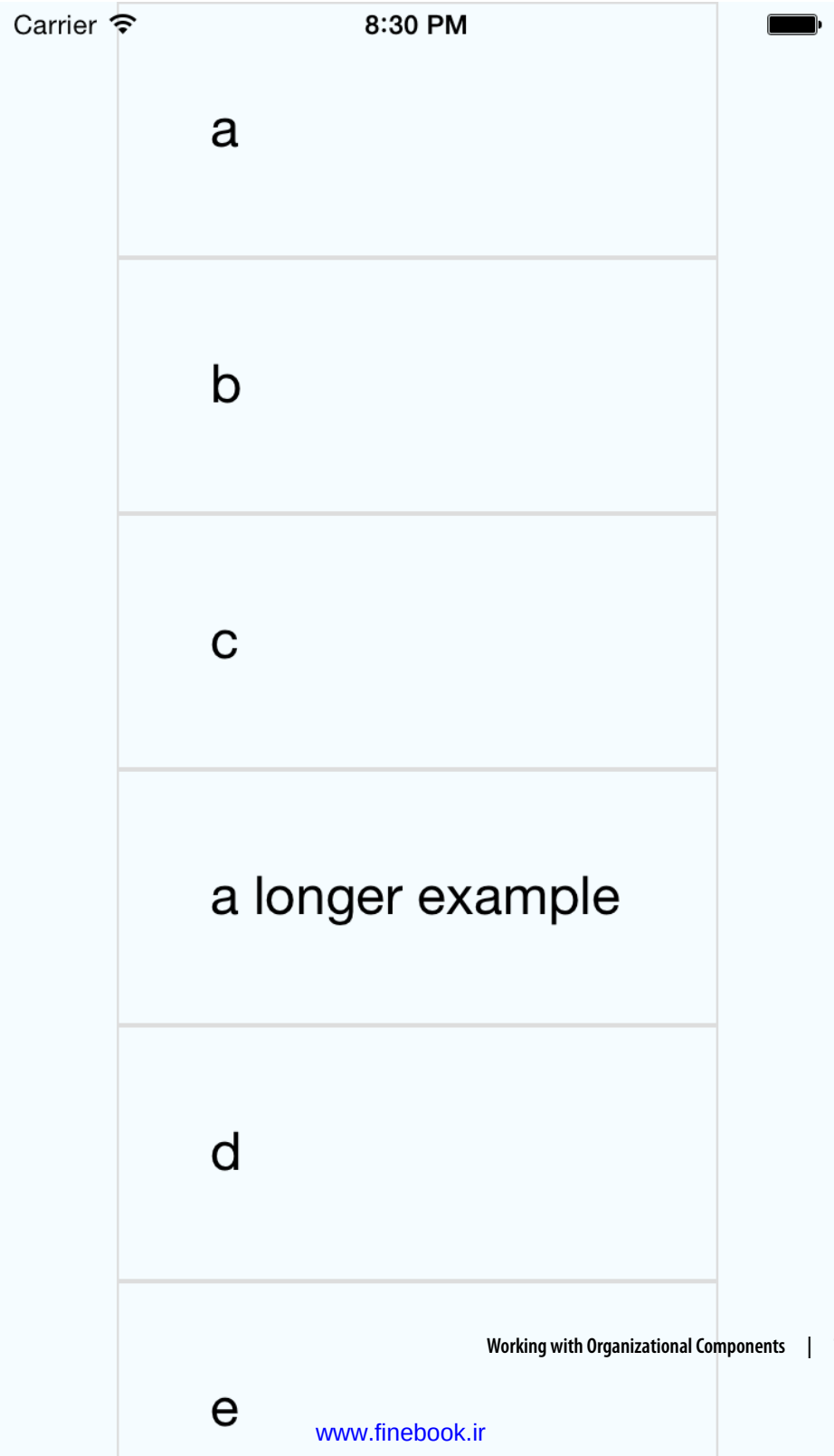
The other prop we need is `renderRow`, which should be a function that returns some JSX based on the data for a given row.

```
_renderRow: function(rowData) {
  return <Text style={styles.row}>{rowData}</Text>;
}
```

Now we can put it all together to see a simple `ListView`, by rendering a `ListView` like so:

```
<ListView
  dataSource={this.state.dataSource}
  renderRow={this._renderRow}
  />
```

It looks like this:



What if we want to do a little more? Let's create a ListView with more complex data. We will be using the NYTimes API to create a simple BestSellers application, which renders the bestsellers list.

First, we initialize our data source to be empty, because we'll need to fetch the data:

```
getInitialState: function() {
  var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
  return {
    dataSource: ds.cloneWithRows([])
  };
}
```

Then, we add a method for fetching data, and update the data source once we have it. This method will get called from `componentDidMount`.

```
_refreshData: function() {
  var endpoint = 'http://api.nytimes.com/svc/books/v3/lists/hardcover-fiction?response-format=jso
  fetch(endpoint)
    .then((response) => response.json())
    .then((rjson) => {
      this.setState({
        dataSource: this.state.dataSource.cloneWithRows(rjson.results.books)
      });
    });
}
```

Each book returned by the NYTimes API has three properties: `coverURL`, `author`, and `title`. We update the ListView's render function to return a component based on those props.

```
_renderRow: function(rowData) {
  return <BookItem coverURL={rowData.book_image}
                   title={rowData.title}
                  author={rowData.author}/>;
},
```

We'll also toss in a header and footer component, to demonstrate how these work. Note that for a ListView, the header and footer are not *sticky*; they scroll with the rest of the list. If you want a sticky header or footer, it's probably easiest to render them separately from the `<ListView>` component.

```
_renderHeader: function() {
  return (<View style={styles.sectionDivider}>
    <Text style={styles.headingText}>
      Bestsellers in Hardcover Fiction
    </Text>
    </View>);
},

_renderFooter: function() {
  return(
```

```
        <View style={styles.sectionDivider}>
          <Text>
            Data from the New York Times bestsellers list.
          </Text>
        </View>
        );
    },
```

All together, the BookList application consists of two files: `BookListV2.js` and `BookItem.js`. (`BookList.js` is a simpler file which omits fetching data from an API, and is included in the Github repository for your reference.)

```
'use strict';

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

var BookItem = require('./BookItem');
var API_KEY = '73b19491b83909c7e07016f4bb4644f9:2:60667290';

var BookList = React.createClass({
  getInitialState: function() {
    var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
    return {
      dataSource: ds.cloneWithRows([])
    };
  },

  componentDidMount: function() {
    this._refreshData();
  },

  _renderRow: function(rowData) {
    return <BookItem coverURL={rowData.book_image}
                     title={rowData.title}
                     author={rowData.author}/>;
  },

  _renderHeader: function() {
    return (<View style={styles.sectionDivider}>
      <Text style={styles.headingText}>
        Bestsellers in Hardcover Fiction
      </Text>
      </View>);
  },
```

```
  _renderFooter: function() {
    return(
      <View style={styles.sectionDivider}>
        <Text>Data from the New York Times bestsellers list.</Text>
      </View>
      );
  },

  _refreshData: function() {
    var endpoint = 'http://api.nytimes.com/svc/books/v3/lists/hardcover-fiction?response-format=js
    fetch(endpoint)
      .then((response) => response.json())
      .then((rjson) => {
        this.setState({
          dataSource: this.state.dataSource.cloneWithRows(rjson.results.books)
        });
      });
  },

  render: function() {
    return (
        <ListView
          style={{marginTop: 24}}
          dataSource={this.state.dataSource}
          renderRow={this._renderRow}
          renderHeader={this._renderHeader}
          renderFooter={this._renderFooter}
          />
    );
  }
});

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#FFFFFF',
    paddingTop: 24
  },
  list: {
    flex: 1,
    flexDirection: 'row'
  },
  listContent: {
    flex: 1,
    flexDirection: 'column'
  },
  row: {
    flex: 1,
    fontSize: 24,
    padding: 42,
```

```
    borderWidth: 1,
    borderColor: '#DDDDDD'
  },
  sectionDivider: {
    padding: 8,
    backgroundColor: '#EEEEEE',
    alignItems: 'center'
  },
  headingText: {
    flex: 1,
    fontSize: 24,
    alignSelf: 'center'
  }
});

module.exports = BookList;
```

The `<BookItem>` is a simple component that handles rendering each child view in the list.

```
// BookItem.js
'use strict';

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


var styles = StyleSheet.create({
  bookItem: {
    flex: 1,
    flexDirection: 'row',
    backgroundColor: '#FFFFFF',
    borderBottomColor: '#AAAAAA',
    borderBottomWidth: 2,
    padding: 5
  },
  cover: {
    flex: 1,
    height: 150,
    resizeMode: 'contain'
  },
  info: {
    flex: 3,
    alignItems: 'flex-end',
    flexDirection: 'column',
    alignSelf: 'center',
    padding: 20
```

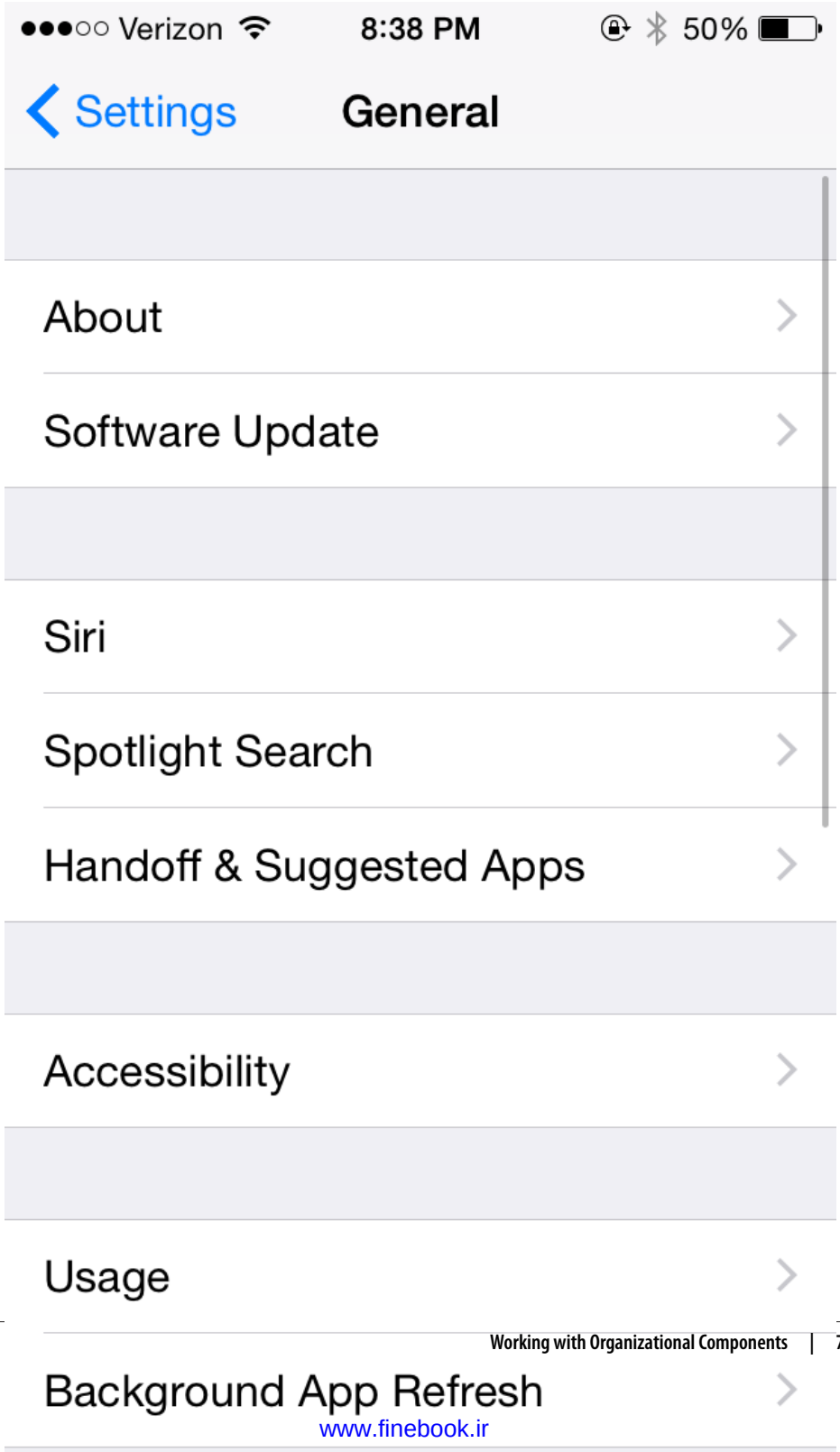
```
  },
  author: {
    fontSize: 18
  },
  title: {
    fontSize: 18,
    fontWeight: 'bold'
  }
});

var BookItem = React.createClass({
  propTypes: {
    coverURL: React.PropTypes.string.isRequired,
    author: React.PropTypes.string.isRequired,
    title: React.PropTypes.string.isRequired
  },

  render: function() {
    return (
      <View style={styles.bookItem}>
        <Image style={styles.cover} source={{uri: this.props.coverURL}}/>
        <View style={styles.info}>
          <Text style={styles.author}>{this.props.author}</Text>
          <Text style={styles.title}>{this.props.title}</Text>
        </View>
      </View>
      );
  }
});

module.exports = BookItem;
```

If you have complex data, or very long lists, you will need to pay attention to the performance optimizations enabled by some of ListView's more complex, optional properties. For most usages, however, this will suffice.

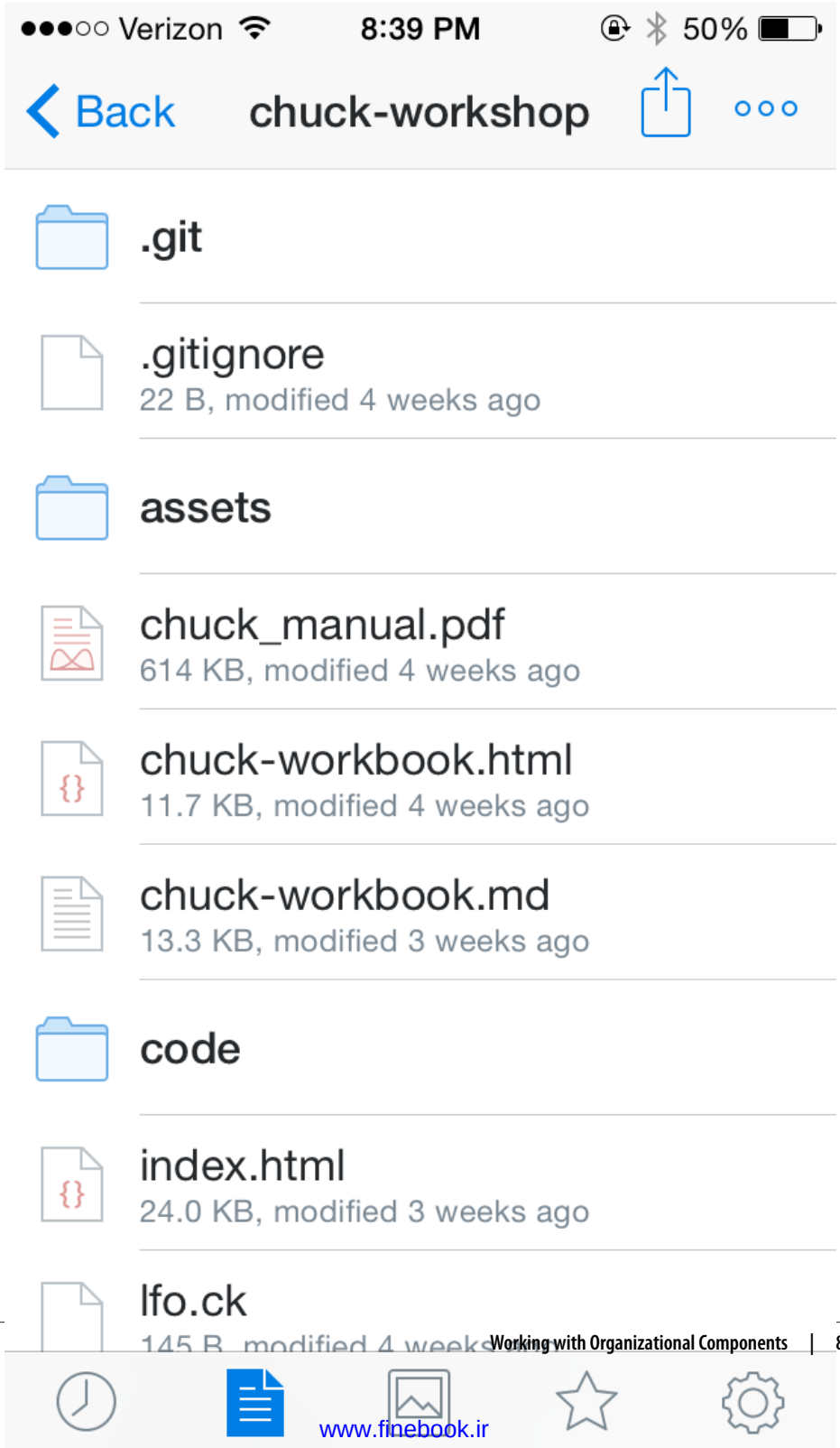
Using Navigators

The `<ListView>` is a good example of combining multiple views together into a more usable interaction. On a higher level, we can use a `<Navigator>` or a `<TabBarIOS>` to present different screens of an app, much as we might have various pages on a website. We'll start by taking a look at the `<Navigator>`.

A `<Navigator>` is a subtle but important component, and is used in many common applications. For instance, the iOS Settings app is essentially just a combination of `<Navigator>` with many `<ListView>` components.



The Dropbox app is also a good example which makes use of the `<Navigator>`:



A `<Navigator>` allows your application to transition between different screens (often referred to as "scenes"), while maintaining a "stack" of routes, so that you can push, pop, or replace states. You can think of this as analogous to the history API on the web. A "route" is the title of a screen, coupled with an index.

On iOS, this interaction often manifests in the "back" functionality in the top-left corner of the screen. For instance, in the Settings app, initially the stack is empty. When you select one of the sub-menus, the initial scene is pushed onto the stack. Tapping "back" will pop it back off.

If you're interested in how this plays out, the `UIExplorer` app has a good demo of the various ways of using the Navigator API.

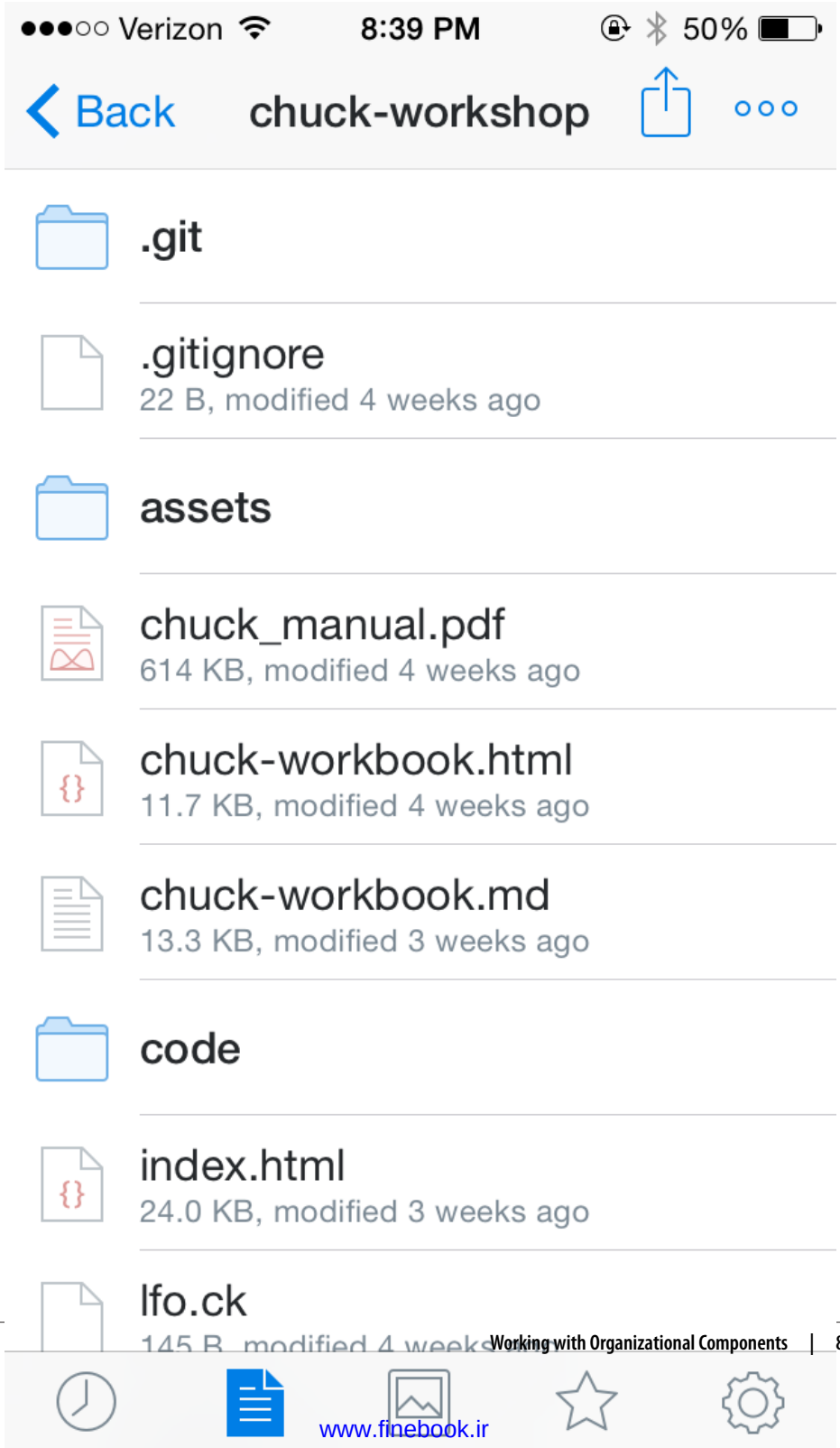
On iOS with React Native, you actually have your choice of two different components in order to implement the Navigator behavior: `<Navigator>` and `<NavigatorIOS>`. `<Navigator>` is a cross-platform component implemented by the React Native team purely in Javascript. Thus, the `<Navigator>` is cross-platform compatible, and will work on both Android and iOS, while the `<NavigatorIOS>` component is iOS-specific. The `<NavigatorIOS>` has a more constrained API, and also takes advantage of Apple's standard animations and behaviors, so it looks more natural; the `<Navigator>` animations are re-implemented to mimic the `<NavigatorIOS>`, and are therefore somewhat less refined.

Which is appropriate for your application? It depends. The React Native documentation offers a comparison of the two, along with pros and cons, and a basic demonstration of usage.

If you want your application to be cross-platform, you should strongly consider using the `<Navigator>`. The animations and performance are sufficiently good, and it's more customizable than the `<NavigatorIOS>`. On the other hand, if fidelity to the iOS platform is a concern, the `<NavigatorIOS>` is the obvious choice.

TabBarIOS

A `<TabBarIOS>` is useful when there are a limited number of main screens in your app. The Dropbox application is a good example of this interaction pattern, combined with a `<NavigatorView>`.



Unlike the <NavigatorIOS>, the <TabBarIOS> does not have a generic version. In part, this is due to the fact that the interaction guidelines around tabs tend to be platform-specific.

On iOS, for instance, Apple is very specific in its recommendations for how tabs should be used for navigation. According to the current interaction guidelines, tab bars should be used to organize application-level information; tab bars should not provide controls for content within the current screen. Visually, the guidelines are also specific: a tab bar should be translucent; must appear on the bottom of the screen; may display no more than five tabs at a time; and may display a badge (a number or exclamation point) to communicate "app-specific information."

By contrast, Google's guidelines on Tab usage allow for tab bars to positioned at the top or bottom of the screen, and are generally less prescriptive.

The TabBarIOSExample provided with React Native is a good demonstration of TabBar usage.

Summary

In this chapter, we dug into the specifics of a variety of the most important components in React Native. We discussed how to utilize basic low-level components, like <Text> and <Image>, as well as higher-order components like <ListView>, <Navigator>, and <TabBarIOS>. We also took a look at how to use various touch-focused APIs and components, in case you want to build your own custom touch handlers. At this point, you should be equipped to build basic, functional applications using React Native! Now that you've acquainted yourself with the components discussed in this chapter, building upon them and combining them to create your own applications should feel remarkably similar to working with React on the web.

Of course, building up basic, functioning applications is only part of the battle. In the next chapter, we'll focus on styling, and how to use React Native's implementation of styles to get the look and feel you want on mobile.

CHAPTER 4

Styles

It's great to be able to build functional applications, but if you can't style them effectively, you won't get very far! In the second chapter, we built a simple weather application with some basic styles. While this gave us an overview of how to style React Native components, we glossed over many of the details. In this chapter, we will take a closer look at how styles work in React Native. We'll cover how to create and manage your stylesheets, as well as the details of React Native's implementation of CSS rules. By the end of this chapter, you should feel comfortable creating and styling your own React Native components and applications.

If you want to share styles between your React Native and web applications, the `react-style` project on Github provides a version of React Native's style system for the web.

Declaring and Manipulating Styles

When working with React for the web, we typically use separate stylesheet files, which may be written in CSS, SASS, or LESS. React Native takes a radically different approach, bringing styles entirely into the world of Javascript and forcing you to link style objects explicitly to components. Needless to say, this approach tends to provoke strong reactions, as it represents a significant departure from CSS-based styling norms.

To understand the design of React Native's styles, first we need to consider some of the headaches associated with traditional CSS stylesheets. Vjeux's "CSS in JS" slide-deck provides a good overview. CSS has a number of problems. All CSS rules and class names are global in scope, meaning that styling one component can easily break another if you're not careful. For instance, if you include the popular Twitter Bootstrap library, you will introduce over 600 new global variables. Because CSS is not

explicitly connected to the HTML elements it styles, dead code elimination is difficult, and it can be nontrivial to determine which styles will apply to a given element.

Languages like SASS and LESS attempt to work around some of CSS's uglier parts, but many of the same fundamental problems remain. With React, we have the opportunity to keep the desirable parts of CSS, but also the freedom for significant divergence. React Native implements a subset of the available CSS styles, focusing on keeping the styling API narrow yet still highly expressive. Positioning is dramatically different, as we'll see later in this chapter. Additionally, React Native does not support psuedo-classes, animations, or selectors. A full list of supported properties can be found in the docs.

Instead of stylesheets, in React Native we work with Javascript-based style *objects*. One of React's greatest strengths is that it forces you to keep your Javascript code — your components — modular. By bringing styles into the realm of Javascript, React Native pushes us to write modular styles, too.

In this section, we'll cover the mechanics of how these style objects are created and manipulated in React Native.

Inline Styles

Inline styles are the simplest way, syntactically, to style a component in React Native, though they are not usually the *best* way. The syntax for inline styles in React Native is the same as for React for the browser:

```
<Text>
  The quick <Text style={{fontStyle: "italic"}}>brown</Text> fox
  jumped over the lazy <Text style={{fontWeight: "bold"}}>dog</Text>.
</Text>
```

Inline styles have some advantages. They're quick and dirty, allowing you to rapidly experiment.

However, you should avoid them in general, because they're less efficient. Inline style objects must be recreated during each render pass. Even when you want to modify styles in response to props or state, you need not use inline styles, as we'll see in a moment.

Styling with Objects

If you take a look at the inline style syntax, you will see that it's simply passing an object to the `style` attribute. There's no need to create the style object in the `render` call, though; instead, you can separate it out:

```
var italic = {
  fontStyle: 'italic'
};
```

```
var bold = {
  fontWeight: 'bold'
};

...

render() {
  return (
    <Text>
      The quick <Text style={italic}>brown</Text> fox
      jumped over the lazy <Text style={bold}>dog</Text>.
    </Text>
    );
}
```

`PanDemo.js` gives us a good example of a use case in which the immutability provided by `Stylesheet.Create` is a hinderance rather than a help. Recall that we wanted to update the location of a circle based on movement — in other words, each time we received an update from the `PanResponder`, we needed to update state as well as change the styles on the circle. In this circumstance, we don't want immutability at all, at least not for the style controlling the circle's location.

Therefore, we can use a plain object to store the style for the circle.

Using Stylesheet.Create

You will notice that almost all of the React Native example code makes us of `Stylesheet.create`. Using `Stylesheet.create` is strictly optional, but in general you'll want to use it. Here's what the docs have to say:

> StyleSheet.create construct is optional but provides some key advantages. It ensures that the values are immutable and opaque by transforming them into plain numbers that reference an internal table. By putting it at the end of the file, you also ensure that they are only created once for the application and not on every render.

In other words, `Stylesheet.create` is really just a bit of syntactic sugar designed to protect you. Use it! The vast majority of the time, the immutability provided by `Stylesheet.create` is helpful. It also gives you the ability to do prop validation via propTypes: styles created with `Stylesheet.create` can be verified using the `View.propTypes.Style` and `Text.propTypes.Style` types.

Style Concatenation

What happens if you want to combine two or more styles?

Recall that earlier we said that we should prefer reusing styled components over styles. That's true, but sometimes style reuse is also useful. For instance, if you have a

`button` style and an `accentText` style, you may want to combine them to create an `AccentButton` component.

If the styles look like this:

```
var styles = Stylesheet.create({
  button: {
    borderRadius: '8px',
    backgroundColor: '#99CCFF'
  },
  accentText: {
    fontSize: '18px',
    fontWeight: 'bold'
  }
});
```

Then you can create a component that has *both* of those styles applied through simple concatenation:

```
var AccentButton = React.createClass({
  render: function() {
    return (
      <Text style={[styles.button, styles.accentText]}>
        {this.props.children}
      </Text>
    );
  }
});
```

As you can see, the `style` attribute can take an array of style objects. You can also add inline styles here, if you want:

```
var AccentButton = React.createClass({
  render: function() {
    return (
      <Text style={[styles.button, styles.accentText, {color: '#FFFFFF'}]}>
        {this.props.children}
      </Text>
    );
  }
});
```

In the case of a conflict, such as when two objects both specify the same property, React Native will resolve the conflict for you. The rightmost elements in the style array take precedence, and falsy values (`false`, `null`, `undefined`) are ignored.

You can leverage this pattern to apply conditional styles. For example, if we had a `<Button/>` component and wanted to apply extra style rules if it's being touched, we could do this:

```
<View style={[styles.button, this.state.touching && styles.highlight]} />
```

This shortcut can help you keep your rendering logic concise.

In general, style concatenation is a useful tool for combining styles. It's interesting to contrast concatenation with web-based stylesheet approaches: @extend in SASS, or nesting and overriding classes in vanilla CSS. Style concatenation is a more limited tool, which is arguably a good thing: it keeps the logic simple and makes it easier to reason about which styles are being applied and how.

Organization and Inheritance

In most of the examples so far, we append our style code to the end of the main Javascript file with a single call to `Stylesheet.create`. For example code, this works well enough, but it's not something you'll likely want to do in an actual application. How should we actually organize styles? In this section, we will take a look at ways of organizing your styles, and how to share and inherit styles.

Exporting Style Objects

As your styles grow more complex, you will want to keep them separate from your components' Javascript files. One common approach is to have a separate folder for each component. If you have a component named `<ComponentName/>`, you would create a folder named `ComponentName` and structure it like so:

```
- ComponentName
  |- index.js
  |- styles.js
```

Within `styles.js`, you create a stylesheet, and export it:

```
'use strict';

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

var styles = Stylesheet.create({
  text: {
    color: '#FF00FF',
    fontSize: 16
  },
  bold: {
    fontWeight: 'bold'
  }
});

module.exports = styles;
```

Within `index.js`, we can import our styles like so:

```
var styles = require('./styles.css');
```

Then we can use them in our component:

```
'use strict';

var React = require('react-native');
var styles = require('./styles.css');
var {
  View,
  Text,
  StyleSheet
} = React;

var ComponentName = React.createClass({
  render: function() {
    return (
      <Text style={[styles.text, styles.bold]}>
        Hello, world
      </Text>
    );
  }
});
```

Passing Styles as Props

You can also pass styles as properties. The propType `View.propTypes.style` ensures that only valid styles are passed as props.

You can use this pattern to create extensible components, which can be more effectively controlled and styled by their parents. For example, a component might take in an optional style prop:

```
'use strict';

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

var CustomizableText = React.createClass({
  propTypes: {
    style: Text.propTypes.Style
  },
  getDefaultProps: function() {
    return {
      style: {}
    };
  },
  render: function() {
    return (
      <Text style={[myStyles.text, this.props.style]}>Hello, world</Text>
    );
```

```
  }
});
```

Reusing and Sharing Styles

We typically prefer to reuse styled components, rather than reusing styles, but there are clearly some instances in which you will want to share styles between components. In this case, a common pattern is to organize your project roughly like so:

```
- js
  |- components
     |- Button
        |- index.js
        |- styles.js
  |- styles
     |- styles.js
     |- colors.js
     |- fonts.js
```

By having separate directories for components and for styles, you can keep the intended use of each file clear based on context. A component's folder should contain its React class, as well as any component-specific files. Shared styles should be kept out of component folders. Shared styles may include things such as your palette, fonts, standardized margin and padding, and so on.

`styles/styles.js` requires the other shared styles files, and exposes them; then your components can require `styles.js` and use shared files as needed. Or, you may prefer to have components require specific stylesheets from the `styles/` directory instead.

Because we've now moved our styles into Javascript, organizing your styles is really a question of general code organization; there's no single correct approach here.

Positioning and Designing Layouts

One of the biggest changes when working with styling in React Native is positioning. CSS supports a proliferation of positioning techniques. Between `float`, absolute positioning, tables, block layout, and more, it's easy to get lost! React Native's approach to positioning is more focused, relying primarily on `flexbox` as well as absolute positioning, along with the familiar properties of `margin` and `padding`. In this section, we'll look at how layouts are constructed in React Native, and finish off by building a layout in the style of a Mondrian painting.

Layouts with Flexbox

Flexbox is a CSS3 layout mode. Unlike existing layout modes such as `block` and `inline`, flexbox gives us a direction-agnostic way of constructing layouts. (That's

right: finally, vertically centering is easy!) React Native leans heavily on flexbox. If you want ot read more about the general specification, the MDN documentation is a good place to start.

With React Native, the following props are related to flexbox:

- flex
- flexDirection
- flexWrap
- alignSelf
- alignItems

Additionally, these related values impact layout:

- height
- width
- margin
- border
- padding

If you have worked with flexbox on the web before, there won't be many surprises here. Because flexbox is so important to constructing layouts in React Native, though, we'll spend some time now exploring how it works.

The basic idea behind flexbox is that you should be able to create predictably structured layouts even given dynamically sized elements. Since we're designing for mobile, and need to accommodate multiple screen sizes and orientations, this is a useful feature.

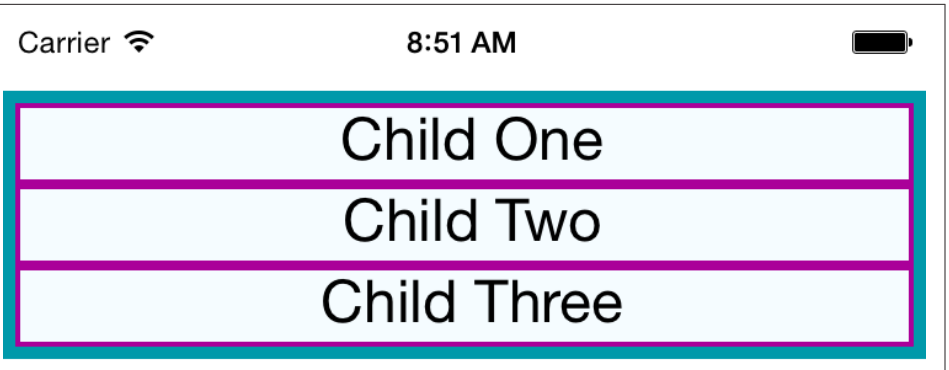
We'll start with a parent `<View/>`, and some children.

```
<View style={styles.parent}>
  <Text style={styles.child}> Child One </Text>
  <Text style={styles.child}> Child Two </Text>
  <Text style={styles.child}> Child Three </Text>
</View>
```

To start, we've applied some basic styles to the views, but haven't touched the positioning yet.

```
var styles = StyleSheet.create({
  parent: {
    backgroundColor: '#F5FCFF',
    borderColor: '#0099AA',
    borderWidth: 5,
    marginTop: 30
```

```
  },
  child: {
    borderColor: '#AA0099',
    borderWidth: 2,
    textAlign: 'center',
    fontSize: 24,
  }
});
```



Next, we will set `flex` on both the parent and the child. By setting the `flex` property, we are explicitly opting-in to flexbox behavior. `flex` takes a number. This number determines the relative weight each child gets; by setting it to `1` for each child, we weight them equally.

We also set `flexDirection: column` so that the children are laid out vertically. If we switch this to `flexDirection: row`, the children will be laid out horizontally instead.

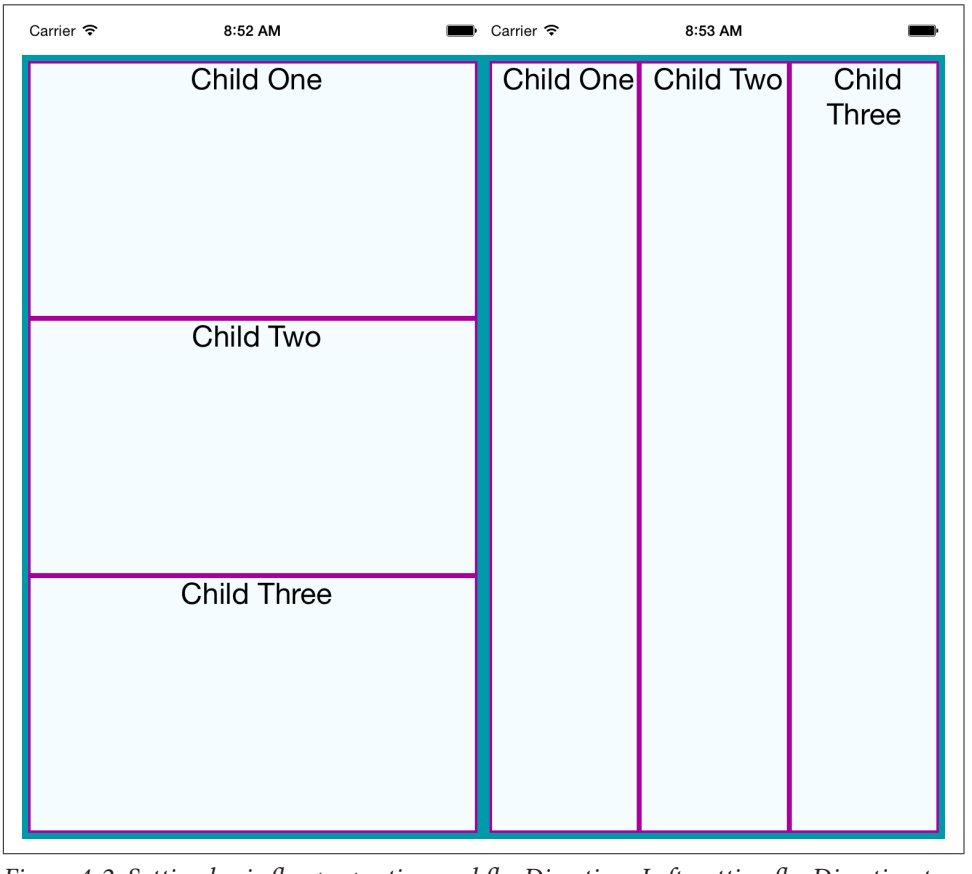


Figure 4-2. Setting basic flex properties, and flexDirection. Left: setting flexDirection to column. Right: setting flexDirection to row.

```
var styles = StyleSheet.create({
  parent: {
    flex: 1,
    flexDirection: 'column',
    backgroundColor: '#F5FCFF',
    borderColor: '#0099AA',
    borderWidth: 5,
    marginTop: 30
```

```
  },
  child: {
    flex: 1,
    borderColor: '#AA0099',
    borderWidth: 2,
    textAlign: 'center',
    fontSize: 24,
  }
});
```

If we set `alignItems`, the children will no longer expand to fill all available space in both directions. Because we have set `flexDirection: row`, they will expand to fill the row. However, now they will only take up as much vertical space as they need.

Then, the `alignItems` value determines *where* they are positioned along the cross-axis. The cross-axis is the axis orthogonal to the flexDirection. In this case, the cross axis is vertical. `flex-start` places the children at the top, `center` centers them, and `flex-end` places them at the bottom.

Let's see what happens when we set `alignItems`:

```
var styles = StyleSheet.create({
  parent: {
    flex: 1,
    flexDirection: 'row',
    alignItems: 'flex-start',
    backgroundColor: '#F5FCFF',
    borderColor: '#0099AA',
    borderWidth: 5,
    marginTop: 30
  },
  child: {
    flex: 1,
    borderColor: '#AA0099',
    borderWidth: 2,
    textAlign: 'center',
    fontSize: 24,
  }
});
```

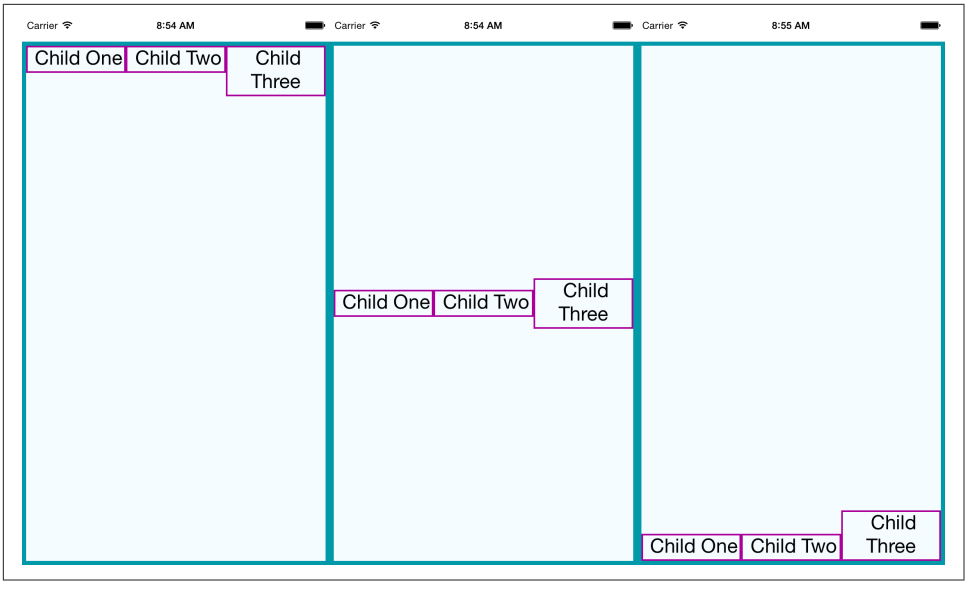


Figure 4-3. Setting alignItems positions children on the cross-axis, which is the axis orthogonal to the flexDirection. Here we see flex-start, center, and flex-end.

Using Absolute Positioning

In addition to flexbox, React Native supports absolute positioning. It works much as it does on the web. You can enable it by setting the position property:

```
position: absolute
```

Then, you can control the component's positioning with the familiar properties of `left`, `right`, `top`, and `bottom`.

An absolutely positioned child will apply these coordinates relative to its parent's position, so you can lay out a parent element using flexbox and then use absolute position for a child within it.

There are some limitations to this. We don't have `z-index`, for instance, so layering views on top of each other is a bit complicated. The last view in a stack typically takes precedence.

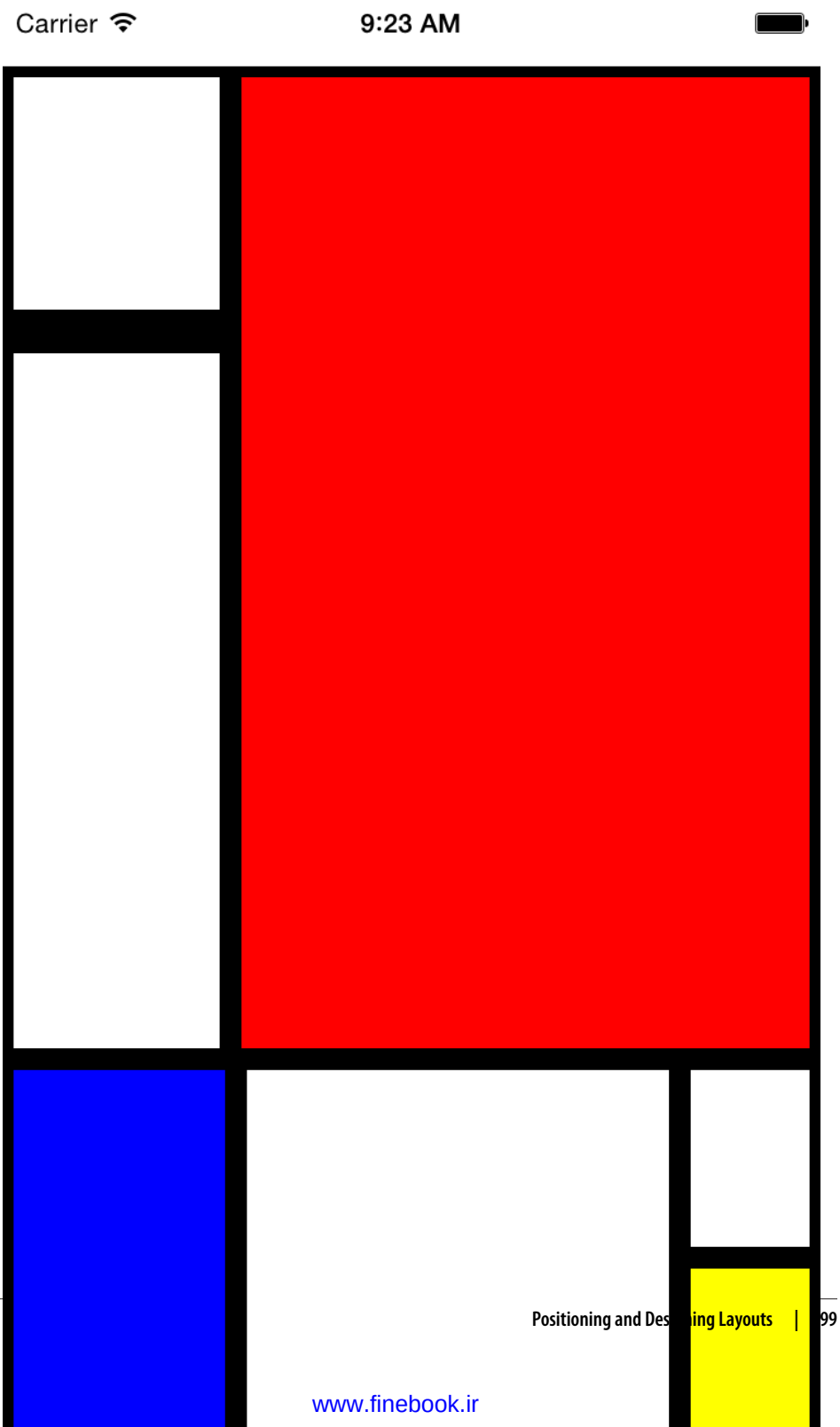
Absolute positioning can be very useful. For instance, if you want to create a container view that sits below the iOS status bar, absolute positioning makes this easy:

```
container: {
  position: 'absolute',
  top: 30,
  left: 0,
  right: 0,
```

```
    bottom: 0
}
```

Putting it Together

Let's try using these positioning techniques to create a more complicated layout. Say we want to mimic a Mondrian painting. Here's the end result:



How should we go about constructing this kind of layout?

To start with, we create a `parent` style to act as the container. We will use absolute positioning on the parent, because it's most appropriate: we want it to fill all available space, except with a 30 pixel offset at the top, due to the iOS status bar. We'll also set its `flexDirection` to `column`.

```
parent: {
  flexDirection: 'column',
  position: 'absolute',
  top: 30,
  left: 0,
  right: 0,
  bottom: 0
}
```

Looking back at the image, we can divide the layout up into larger blocks. These divisions are in many ways arbitrary, so we'll pick an option and roll with it. Here's one way we can segment the layout:

We start by cutting the layout into a top and bottom block:

```
<View style={styles.parent}>
  <View style={styles.topBlock}>
  </View>
  <View style={styles.bottomBlock}>
  </View>
</View>
```

Then we add in the next layer. This includes both a "left column" and "bottom right" sector, as well as the actual `<View/>` components for cells three, four, and five.

```
<View style={styles.parent}>
  <View style={styles.topBlock}>
    <View style={styles.leftCol}>
    </View>
    <View style={[styles.cellThree, styles.base]} />
  </View>
  <View style={styles.bottomBlock}>
    <View style={[styles.cellFour, styles.base]}/>
    <View style={[styles.cellFive, styles.base]}/>
    <View style={styles.bottomRight}>
    </View>
  </View>
</View>
```

The final markup contains all seven cells:

```
<View style={styles.parent}>
  <View style={styles.topBlock}>
    <View style={styles.leftCol}>
      <View style={[styles.cellOne, styles.base]} />
      <View style={[styles.base, styles.cellTwo]} />
```

```
      </View>
      <View style={[styles.cellThree, styles.base]} />
    </View>
    <View style={styles.bottomBlock}>
      <View style={[styles.cellFour, styles.base]}/>
      <View style={[styles.cellFive, styles.base]}/>
      <View style={styles.bottomRight}>
        <View style={[styles.cellSix, styles.base]} />
        <View style={[styles.cellSeven, styles.base]} />
      </View>
    </View>
  </View>
```

Now let's add the styles that make it work. (Show stylesheet.)

```
// Mondrian/style.js

'use strict';

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

var styles = StyleSheet.create({
  parent: {
    flexDirection: 'column',
    position: 'absolute',
    top: 30,
    left: 0,
    right: 0,
    bottom: 0
  },
  base: {
    borderColor: '#000000',
    borderWidth: 5
  },
  topBlock: {
    flexDirection: 'row',
    flex: 5
  },
  leftCol: {
    flex: 2
  },
  bottomBlock: {
    flex: 2,
    flexDirection: 'row'
  },
  bottomRight: {
    flexDirection: 'column',
    flex: 2
  },
  cellOne: {
```

```
    flex: 1,
    borderBottomWidth: 15
  },
  cellTwo: {
    flex: 3
  },
  cellThree: {
    backgroundColor: '#FF0000',
    flex: 5
  },
  cellFour: {
    flex: 3,
    backgroundColor: '#0000FF'
  },
  cellFive: {
    flex: 6
  },
  cellSix: {
    flex: 1
  },
  cellSeven: {
    flex: 1,
    backgroundColor: '#FFFF00'
  }
});

module.exports = styles;
```

Summary

In this chapter, we looked at how styles work in React Native. While in many ways styling is similar to how CSS works on the web, React Native introduces a different structure and approach to styling. There's plenty of new material to digest here! At this point, you should be able to use styles effectively to create the mobile UIs you need with React Native. And best of all, experimenting with styles is easy: being able to hit "reload" (CMD+R) in the iOS Simulator grants us a tight feedback loop. (It's worth noting that with traditional iOS development, editing a style would typically require rebuilding your application. Yikes.)

If you want more practice with styles, try going back to the Bestsellers or Weather projects, and adjusting their styling and layouts. As we build more sample applications in future chapters, you'll have plenty of material to practice with, too!