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.

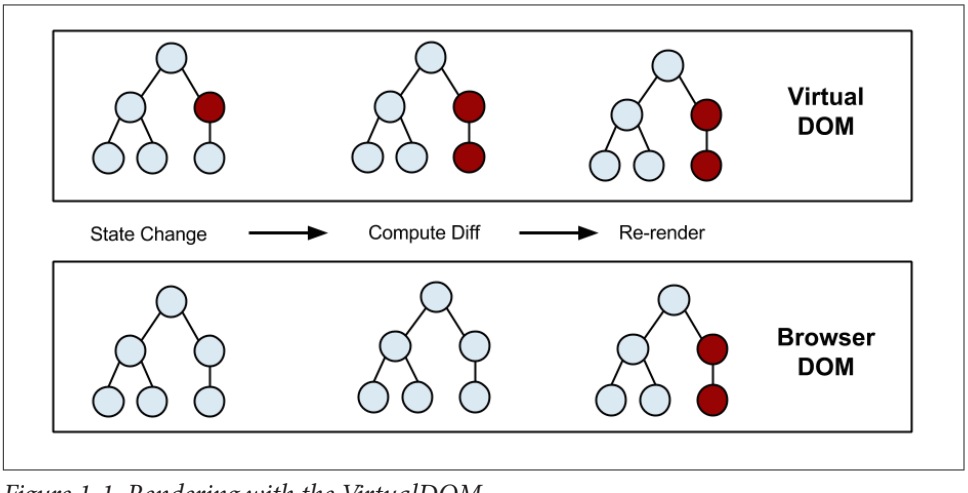

Figure 1-1. Rendering with the VirtualDOM

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.

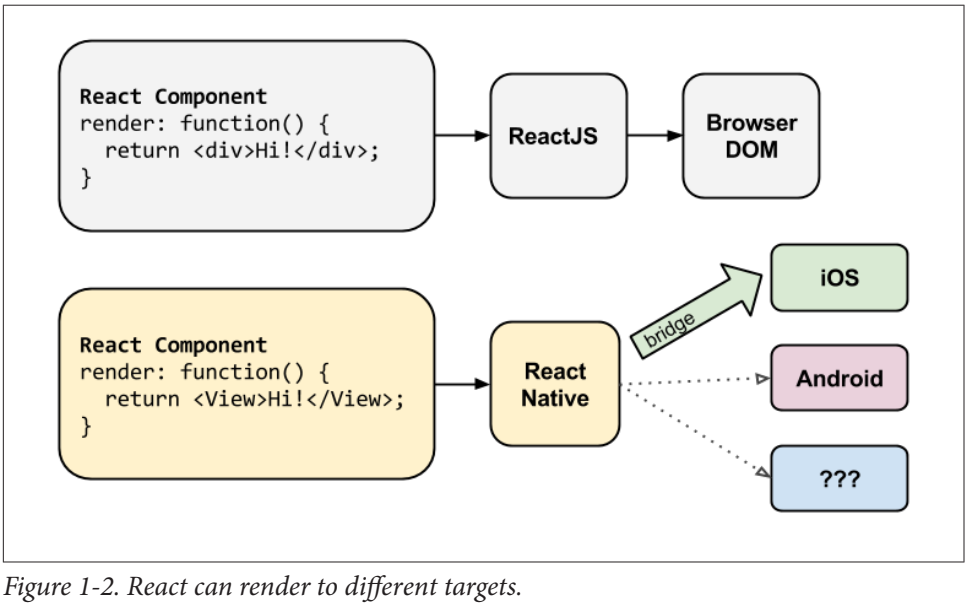

Figure 1-2. React can render to different targets.

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.

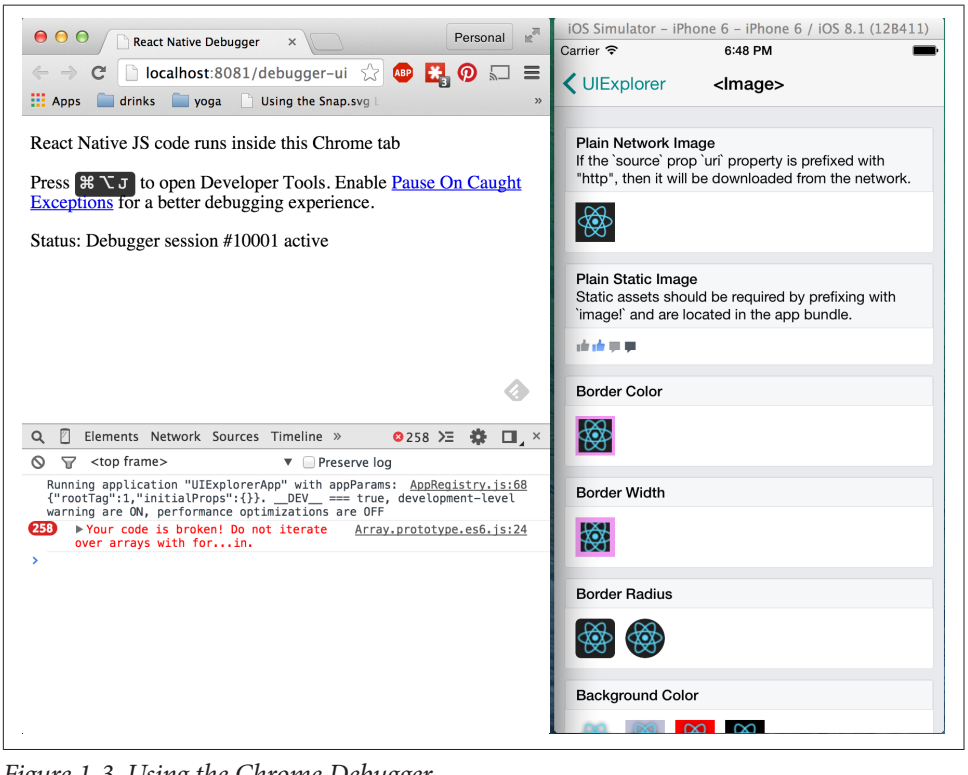

Figure 1-3. Using the Chrome Debugger

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:

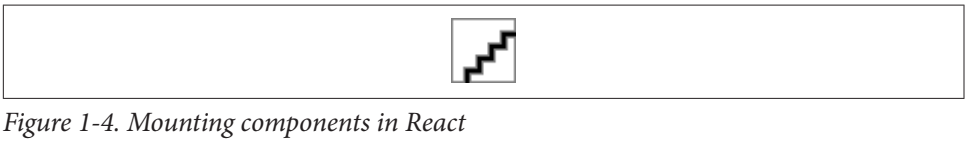

Figure 1-4. Mounting components in React

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.