React Native: Into a new world of rapid iOS development

Create iOS native apps in JavaScript with powerful tool support

Sing Li

Get on the mobile fast track with React Native, the fastest-growing platform and tool set for iOS development. Put aside the Objective-C and Swift tutorials, and start coding iOS apps with JavaScript. Get an introduction to React Native, learn about its architecture and internal operations, and work through an example application that highlights React Native programming techniques.

Two years after releasing the phenomenally successful React JavaScript open source UI library, Facebook/Instagram unleashed another sensation during the March 2015 F8 conference. To the delight of a global community of admiring developers, including staunch supporters of its older React sibling, the open source React Native project leapt to prominence on GitHub. Within three short months, React Native gained the support of more than 170 global contributors and garnered more than 15,000 stars. What astounding capabilities warrant such an enthusiastic reception? In a nutshell: super-easy iOS application development in JavaScript. React Native delivers on this goal and more, making it the fastest-growing development platform and tool since iOS came into being.

This tutorial introduces React Native and explains its architecture and internal operations. You'll work through a complete app example that highlights key React Native development techniques. A brief overview of available tools for React Native is also included. You do need to be familiar with React programming to appreciate the essence of React Native; this tutorial assumes that you have that familiarity. Motivated JavaScript developers can readily learn both libraries in an afternoon. In contrast, initiation into conventional iOS development usually takes months, typically requiring a multiweek warmup to the venerable Objective-C (or newer Swift) programming language, an understanding of iOS-specific layout and UI construction techniques and tools, mastery of the nuances of the complex Xcode IDE, and familiarity with a multifaceted build workflow.

Using IBM MobileFirst with React Native apps

Be sure to read my tutorial “Write an iOS 8 game with React Native and Advanced Mobile Access” to find out how you can easily enhance your React Native apps with the rich services offered by the MobileFirst platform on IBM Cloud.
Inside React Native
React Native adapts the best features of React — already a mature library used by tens of thousands of loyal JavaScript developers — for mobile development. With React, you can build applications out of high-performance reusable components and craft these lean components. React optimizes performance by isolating the web developer from the browser's quirky DOM — through the notion of a virtual DOM. Your React code modifies the virtual DOM, and the React runtime takes care of optimizing the refresh and update of the physical DOM.

React Native architecture
In React Native, too, you program by modifying a virtual DOM. But unlike with React, this DOM is not backed by a physical browser. Instead, the virtual DOM's target rendering output consists of bona fide native iOS UI components.

All of this magic is carried out through an optimized asynchronous bridge. Figure 1 illustrates React Native’s runtime architecture.

Figure 1. High-level runtime architecture of React Native
In this flexible architecture, the JavaScript interpreter can run anywhere. Typically, the interpreter runs on a thread inside the native app on the device or emulator. However, during remote debugging, the JavaScript code can run in the developer's browser (see the "Powerful tools, instant gratification" section). Time-consuming tasks such as image decoding, disk I/O, and layout can also be farmed out to other worker threads, fully taking advantage of the parallelism offered by the multicore CPUs available on modern mobile devices.

**Rendering to native iOS platform components**

As much as possible, React Native's programming model shields the fact that you're not running code in a browser. However, many properties and attributes that are available with standard HTML components are unavailable on native iOS platform UI components (and the reverse). So you can use only a subset of the standard attributes when programming with React Native. (See the React Native API documentation for details.)

**ES6 JavaScript syntax**

You might notice in the example application's source code that the JavaScript syntax looks slightly different from what you usually use. The reason is that the React Native tool set includes the Babel JavaScript compiler. Babel supports both JSX syntax and the upcoming ECMAScript 6 (ES6) syntax and compiles down to ES5. ES5 has been the JavaScript standard since December 2009 and is currently supported by all modern browsers and server-side JavaScript engines.

Unlike DOM components, iOS UI components don't respond to CSS styling. For this reason, React Native enforces all styling through JavaScript StyleSheet objects and not CSS. However, React Native supports a familiar CSS-like syntax that offers a subset of familiar CSS attribute selectors.

User-input event handling in an iOS application is substantially different from in a typical web application. React Native offers a comprehensive Gesture Responder system modeled after iOS's own. Applications can tweak the Gesture Responder if necessary to gain fine-grained control over event propagation and custom escalation through tiers of components.

Last but not least, a typical mobile developer using Objective-C/Swift has access to rich support for the underlying platform APIs — from camera control to network access to phone contact management. React Native cannot possibly support all of these features and still remain lean. Instead, the asynchronous bridge at the center of the React Native architecture supports bidirectional export of arbitrary Objective-C APIs to JavaScript. This design makes it possible for the React Native team — along with the greater React Native developer community — to migrate platform features and APIs gradually.

**A tour of the React Native example app**

The example for this tutorial is a document and video viewer app. When you start the app, the initial screen displays a scrollable list of articles, as shown in Figure 2.
Figure 2. App display of a selectable list of articles

If you select an article, that article is displayed in a web view for reading, as shown in Figure 3.
Figure 3. Article that's been selected for reading

You can return to the list of articles by touching the top-left Articles link. Touch Videos to view the list of available open source videos, shown in Figure 4.
Figure 4. App showing the list of available videos

Select any video to start playing it in a new video view, shown in Figure 5.
Figure 5. App playing the open source Elephants Dream movie

You can stop the video playback at any time by touching the top-left Videos link to return to the list of videos.

Figure 6 shows the storyboard flow of scenes in this app. From the main screen displaying the article, you can navigate to the list of videos or select an article to show it in the web view. From the list of videos, you can select a video to start playing it, or return to the list of articles. Behind the scenes, a <NavigationiOS> component is the orchestrator of the storyboard flow (see the "Orchestrating views with the <NavigatoriOS> component" section).
Installing React Native and running the example app

Apple mandates that all iOS development be performed on a Mac. You need a version of the latest Xcode installed on your Mac to work on iOS apps, including those you build with React Native. Make sure you're on a Mac that meets all the requirements in this Getting Started guide. Also follow the guide to get React Native installed via:

```bash
npm install -g react-native-cli
```

In the project directory, run:

```bash
npm install
```

React Native has a generator that creates a working skeletal boilerplate app that you can customize. All the tools you need are integrated into the app, and all you need to do is open its .xcodeproj file and run it in Xcode. Xcode builds the app, starts a PC-to-emulator/device bridging packaging server, and starts the app in the iOS emulator (or on an actual device).

Download the tutorial's sample code, which contains the completed dwviewer app. Run `npm install` at the root directory, and then open the Xcode project file via:

```bash
open dwviewer.xcodeproj
```
Then click run in Xcode to build the project, start the packaging server in a shell, and start the app in the emulator. The app should display the initial screen shown in Figure 2.

Once a React Native app is running in the emulator, you can press a hot key to trigger a reload of JavaScript code via the packaging server. (When the app runs on a device, shaking the device has the same effect.) The packaging server also periodically scans your project directory for JavaScript source changes and alerts you of any compilation errors as you code. Xcode's app profiler starts automatically so you can monitor the memory and CPU requirements of your app in real time while it's running in the emulator.

Diving into the code
You'll find all the example app's code in the index.ios.js file. A supplementary datasource.js file supplies the app with data.

Laying out an article-list cell
As a React user, you'll find the structure of the code to be familiar. Along with JavaScript code (see the ES6 JavaScript syntax sidebar), you can see render() calls that use JSX syntax. The code in Listing 1 defines a single article-list cell.

Listing 1. Single article-list cell definition

```javascript
renderArticle: function(article) {
  return (
    <TouchableOpacity
      activeOpacity={0.5}
      onPress={() => this.showArticle(article.url)}>
      <View style={styles.cellcontainer}>
        <View style={styles.celltitlerow}>
          <Text style={styles.title}>{article.title}</Text>
          <Text style={styles.year}>({article.year})</Text>
        </View>
        <Text style={styles.description}>{article.desc}</Text>
      </View>
    </TouchableOpacity>
  );
},
```

The article-list cell is a composition of <TouchableOpacity>, <View>, and <Text> components.

Responding to touch instantly with <TouchableOpacity>

FlexBox polyfill
If you're like most React Native developers, you started out in web application development and are familiar with CSS Flexible Box Layout. iOS native UI components don't support CSS FlexBox, but most typical iOS application layouts can be fully specified using FlexBox. React Native supports the FlexBox Layout model via a polyfill in its StyleSheet object implementation. Experienced web developers will feel right at home laying out iOS UI screens with FlexBox, thanks to the polyfill. Other polyfills available include requestAnimationFrame, XMLHttpRequest, and fetch.

The <TouchableOpacity> component wraps the laid-out cell and provides an opacity-change feedback animation, handled entirely by the native component, when the user touches the cell. This sort of instant feedback is what distinguishes native iOS apps from web apps that run on...
mobile devices. The \texttt{activeOpacity} property specifies the opacity to use (half of fully opaque) when the cell is touched. The \texttt{<TouchableOpacity>} component also forwards the event (a React \textit{synthetic} event) via the \texttt{onPress} handler. In this case, the \texttt{ArticleView}'s own \texttt{showArticle()} method is called with the article's URL. React Native also has the \texttt{<TouchableHighlight>} component for even more control over the instant-feedback effect when a user touches the wrapped component.

\section*{Styling components through the JavaScript StyleSheet API and FlexBox}

In \texttt{Listing 1}, note the use of \texttt{style={styles.*}} to style the various components. React Native uses a JavaScript \texttt{StyleSheet API} that supports the FlexBox layout (see the \texttt{FlexBox polyfill} sidebar).

The app's styles are defined in the code in \texttt{Listing 2}.

\subsection*{Listing 2. Style definitions}

\begin{verbatim}
var styles = StyleSheet.create({
  ... 
  cellcontainer: {
    flex: 1, 
    flexDirection: 'column', 
    justifyContent: 'center', 
    alignItems: 'stretch', 
    backgroundColor: '#F5FCFF', 
    borderWidth: 0.5, 
    borderColor: '#d6d7da', 
    padding: 5, 
  },
  celltitlerow: {
    flex: 1, 
    flexDirection: 'row', 
    justifyContent: 'flex-start', 
    alignItems: 'stretch', 
    backgroundColor: '#F5FCFF', 
  },
  ... 
});
\end{verbatim}

In \texttt{Listing 2}, you can see the familiar CSS-like selectors (for JavaScript compatibility, camelcase is used in lieu of hyphenated selector names) and FlexBox layout options.

\section*{Orchestrating views with the \texttt{NavigatorIOS} component}

When an article cell is touched, the \texttt{showTouch()} method is called, and the article is displayed with a \texttt{<WebView>} component:

\begin{verbatim}
showArticle: function(articleURL) {
  this.props.navigator.push({
    title: "Article", 
    component: WebView, 
    passProps: {url: articleURL}, 
  });
},
\end{verbatim}

The \texttt{<WebView>} component displays the selected article in a configurable minibrowser. The \texttt{<WebView>} is animated in from the right by a \texttt{<NavigatorIOS>} component, referenced through the \texttt{navigator} prop. \texttt{<NavigatorIOS>} wraps the standard iOS UIKit \texttt{UINavigationController}'s
views stack management APIs familiar to most classic iOS developers. `<NavigatorIOS>` offers various APIs to push, pop, and manipulate a stack of views. It also offers forward and backward navigation via optional touchable buttons on the title bar. The `<NavigationIOS>` component is the root component of the app, as shown in Listing 3.

### Listing 3. Creating the `<NavigationIOS>` component

```jsx
var dwViewerApp = React.createClass({
  render: function() {
    return (
      <NavigatorIOS ref='nav' style={styles.container}
        initialRoute={{
          component: ArticlesView,
          title: 'Articles',
          rightButtonTitle: 'Videos',
          onRightButtonPress: this.onButPress,
        }}
      />
    );
  }
});
```

The `initialRoute` prop specifies the initial view to display, in this case the `ArticlesView`. The route includes the title to display with the view and optional button titles and touch handlers. These buttons, right and left, are displayed on the title bars if specified. The `<NavigationIOS>` passes down a `navigator` prop to all its direct children, in this case `ArticlesView`. The children can use this `navigator` prop to manipulate the managed stack of views.

### Working with the `<ListView>` component

The list of articles is displayed by the `ArticlesView` component, which composes a React Native `<ListView>`, as shown in Listing 4.

### Listing 4. Composing a `<ListView>`

```jsx
var ArticlesView = React.createClass({
  render: function() {
    return (  
      <ListView
        dataSource={this.state.dataSource}
        renderRow={this.renderArticle}
        style={styles.listView}
      />
    ),
  }
});
```

Although `<ListView>` doesn't wrap a native UIKit `UITableView` component (which almost all native iOS apps use to display lists), it supplies its own version that can maintain adequate performance over asynchronous bridged operation.

The `<ListView>` loads its data through the `dataSource` prop, specified here as the `ArticleView`'s `dataSource` state variable. The `renderRow` prop is set to the `renderArticle()` method that you saw earlier, which renders a single article cell.
The `dataSource` state variable loads data for the `<ListView>` through `datasource.js`, which defines the data as an array of statically coded JSON objects. Data for the `<ListView>` is fetched once, when the view first displayed, via `ArticlesView`'s `componentDidMount` handler:

```javascript
var DataSource = require('./datasource');
...
componentDidMount: function() {
    this.setState({
        dataSource:    this.state.dataSource.cloneWithRows(DataSource.getArticles()),
        loaded: true,
    });
},
```

### Navigating between articles and videos

When the user touches the Videos button on the top right, `ArticleView`'s `onButPress()` method is called. This method calls `<NavigatorIOS>`'s `push()` method to push an instance of the `VideosView` onto the stack, causing the list of videos to slide in from the right:

```javascript
onButPress: function() {
    this.refs.nav.push({
        component: VideosView,
        title: "Videos",
    });
},
```

An Articles button is displayed on the title bar when `VideosView` is shown (see Figure 4). The user can touch the button to pop the current view off the stack and go back to the list of articles.

### Laying out the video-list cell

A video-list cell is laid out by composing React Native components in the `VideosView`'s `renderVideo` method, as shown in Listing 5.

### Listing 5. Laying out a video-list cell

```javascript
renderVideo: function(video) {
    return (  
        <TouchableOpacity activeOpacity={0.5} onPress={() => this.showVideo(video.title, video.url)}>
            <View style={styles.videoscontainer}>
                <Image source={{uri: video.icon}} style={styles.vicon} />
                <View style={styles.videocellcontainer}>
                    <View style={styles.celltitlerow}>
                        <Text style={styles.title}>{video.title}</Text>
                        <Text style={styles.year}>({video.year})</Text>
                    </View>
                    <Text style={styles.description}>{video.desc}</Text>
                </View>
            </View>
        </TouchableOpacity>
    );
},
```

This layout looks similar to the layout of the article cell. The only addition is an `<Image>` component, which shows the video's icon. To be accessible offline, these static PNG files must be added to the app's resource bundle via the app's `Images.xcassets` file. See the Adding Static Resources to...
your App using Images.xcassets instructions (scroll down to see the section) in the React Native documentation.

**Integrating a community-contributed React Native component**

The loading of VideosView's ListView and the handling of video selection is identical to ArticlesView's and is left as an exercise for you to trace through. One main difference is the display of the actual video.

Instead of using a `<WebView>` to display the video, the app uses a third-party `<Video>` React Native component: react-native-video, created by community member Brent Vatne. If you want to add this component to your own app, make sure you follow Vatne's detailed instructions (scroll down to see them) for installing it into your Xcode project. When you create your own apps, you might want to explore the growing number of community-contributed React Native components on GitHub.

This third-party `<video>` component is wrapped by the `VideoViewer` class, as shown in Listing 6.

**Listing 6. Wrapping the `<Video>` component**

```javascript
var VideoViewer = React.createClass({
  render: function () {
    return(<View style={styles.vidcontainer}>
      <Video source={this.props.source} rate={1.0} volume={1.0} muted={false} paused={false} resizeMode="stretch" style={styles.video} />
    </View>);
  },
});
```

This `VideoViewer` is pushed onto the view stack when a video is selected via `VideosView`'s `showVideo()` method:

```javascript
showVideo: function(title, url) {
  this.props.navigator.push({
    title: title,
    component: VideoViewer,
    passProps: {source: {uri: url}},
  });
},
```

This concludes your examination of the app's code.

**Powerful tools, instant gratification**

Extraordinary debugging support

Not only can you debug remotely via Chrome over WebSocket. Also, when your program crashes during execution, stack traces are displayed in a series of red cells within a
ListView on the device's screen. Touching any cell triggers the referenced source file being open on the developer's PC (thanks to the connected packaging server), within the editor of the developer's choice (set by the REACT_EDITOR variable), with the cursor landing on the exact offending line of code!

Simply as an open source library, React Native is a significant step. But what makes it shine is the pragmatic tool set that React Native is also released with. These tools take advantage of the existing rich and mature tools that are available for JavaScript developers.

Figure 7 shows the in-app developer menu, which pops up via a hot key when you're running your React Native app — either in the emulator or on an actual device. You can reload the app, start profiling the code, and measure the current rendering frames per second (FPS). And you can remote-debug your JavaScript code via either Chrome's or Safari's built-in JavaScript debugger. With remote debugging, you can single-step, examine or modify point-in-time variable values, set breakpoints, and much more.

Figure 7. Native App Running showing in-app developer menu

With the Inspect Element option, you can inspect the styles and attributes of the React Native components while the native iOS app is running. The display pops up on the device's screen and is similar to the already familiar browser-based inspect-element layout. Figure 8 shows this in-app overlay.
Figure 8. In-app overlay supporting dynamic browser-like element inspection

These kinds of in-app, onscreen developer-assist features are what native iOS developers using Objective-C/Swift have longed for but never quite experienced. JavaScript developers using React Native can enjoy them well before the official 1.0 release.

Conclusion

If you're a JavaScript developer who's dreamed about writing iOS applications for the App Store, your dream can now become a reality. React Native extends your React programming skills to iOS today, and it will soon extend them to Android. The rich community support ensures the availability of an endless flow of new components and feature APIs. React Native already has an unstoppable momentum that I'm certain will make it the mainstream iOS development platform of choice. React Native makes iOS mobile app development more fun than ever before.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code for this article's example app</td>
<td>mo-react-native-ios8.zip</td>
<td>142KB</td>
</tr>
</tbody>
</table>
Related topics

- React: Create maintainable, high-performance UI components
- Create an IBM Watson explorer with React
- Obtain the latest update of this tutorial's sample source code at the author's GitHub repository.
- React Native: Visit the official React Native site and read the reference docs.
- UIKit Framework Reference: Learn more about all the available iOS platform UI components and platform APIs at Apple's official iOS developer website.
- react-native GitHub repository: Get the source code of React Native from the GitHub repository.

© Copyright IBM Corporation 2015
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)