React Native

In this chapter, we're going to walk through how to build your first React Native app. We won't get to cover React Native in-depth in this chapter. React Native is a *huge* topic that warrants its own book.

We're going to explain React Native for the React Developer. By the end of this chapter you'll be able to take your React web app and have the foundation to turn it into a React Native app.

As we've seen so far, React is both fun and powerful. In this chapter, the goal is to get an idea of how everything we love about React can be used to build a native iOS or Android application using React Native.

Without getting too deep into the technicalities of how React Native works under the hood, the high-level summary looks like this:

When we build React components, we're actually building representations of our React components, not actual DOM elements using the *Virtual DOM*. In the browser, React takes this virtual DOM and pre-computes what the elements should look like in a web browser and then hands it over to the browser to handle the layout.

The idea behind React Native is that we can take the tree of object representations from our React components and rather than mapping it to the web browser's DOM, we map it to iOS's UIView[177] or Android's android.view[178]. Theoretically, we can use the same idea behind React in the web browser to build native iOS and Android applications. This is the fundamental idea behind React-Native.

In this section, we're going to work through building React components for the native renderers and highlight the differences between React for Native vs. React for the web.

Before we dive into the details, let's start off high level. It's crucial to understand that we're no longer building for the web environment, which means we have different UX, UI, and no URL locations. Despite the fact that the thought processes we've built around building UIs for the web help, they aren't enough to translate to building a great user experience in the mobile native environment.

Let's take a second and look at some of the most popular native applications and play around with them. Rather than use the application from a user perspective, try imagining building the application, taking note of the details. For instance, look for the micro-animations, how the views are setup, the page transitions, the caching, how data is passed around from screen to screen, what happens when we're waiting for information to load.

In general, we can get away with less deliberately designed UI on the web. On mobile however, where our screen-size and data details are more constrained and focused, we are forced to focus on

[177] https://developer.apple.com/reference/uikit/uiview
[178] https://developer.android.com/reference/android/view/View.html

the experience of our application. Building a great UX for a native app requires deliberate execution and attention to detail.

In this chapter one of our goals is to highlight both the syntactic differences as well as the aesthetic differences that exist between both platforms.

Init

As we're focused on getting directly to the code as we work through this section, we'll want to have an application bootstrapped for us so we can experiment and test out building native applications. We'll need to *bootstrap* our application so we have a basic application we can work with as we are learning the different components of React-Native.

In order to bootstrap a React-Native application, we can use the `react-native-cli` tool. We'll need to install the React-Native cli tool by using the Node Package Manager (`npm`). Let's install the `react-native-cli` tool globally so we can access it anywhere on our system:

```
npm i -g react-native-cli@2.0.1
```

Installation documentation

For formal instructions on getting set up with React Native for your specific development environment, check out the official Getting started[179] docs.

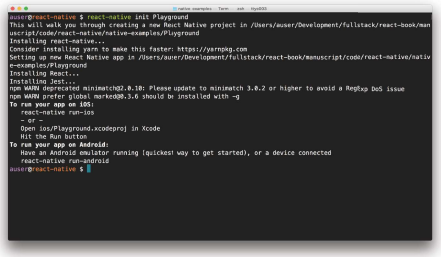
With the `react-native-cli` tool installed, we will have the `react-native` command available in our terminal. If this returns a "not recognized" error, check the getting started docs from above for your development platform.

```
$ react-native
```

To create a new React-Native project, we'll run the `react-native init` command with the name of the application we want to generate. For example, let's generate a React-Native application called `Playground`:

```
$ react-native init Playground
```

[179] http://facebook.github.io/react-native/releases/0.31/docs/getting-started.html



> Your output of the command might look a bit different from above, but as long as you see the instructions on how to launch it. If you see an error, check the Troubleshooting[180] documentation for more instructions on launching the application.

Now we can use our `Playground` app we just created to test and run the code we'll create throughout this section.

Routing

Let's start off our application with routing as it's foundational to nearly every application.

When we implement routing in the web, we're typically mapping a URL to a particular UI. For example, let's take a game that might be written for the web with multiple screens written using the React Router V2 API:

When we implement routing on the web, we're typically mapping some URL to a specific UI. Here's how that looks with React Router V2 with the URLs mapped to the active components:

src/routes.js

```
export const Routes = (
  <Router history={hashHistory}>
    <Route path='/' component={Main}>
      <IndexRoute component={Home} />
      <Route
        path='players/:playerOne?'
        component={PromptContainer}
      />
```

[180] http://facebook.github.io/react-native/releases/0.31/docs/troubleshooting.html#content


```
      <Route
        path='battle'
        component={ConfirmBattleContainer}
      />
      <Route
        path='results'
        component={ResultsContainer}
      />
    </Route>
  </Router>
);
```

URL	Active Component
foo.com	Main -> Home
foo.com/players	Main -> PromptContainer
foo.com/players/ari	Main -> PromptContainer
foo.com/battle	Main -> ConfirmBattleContainer
foo.com/results	Main -> ResultsContainer

React Router lays out this mapping between URL and UI beautifully (i.e. declaratively).

As our user navigates to different URLs in our applications, different components become *active*. The URL is so foundational to React Router that it's mentioned twice in the opening description:

> React Router keeps your UI in sync with the **URL. **...Make the **URL your first thought, not an after-thought.

This is a clean, well-understood paradigm and one that has made React Router as popular as it is. However, what if we're developing for a native app? There is no URL we can refer. There's not really even a similar conceptual mapping of a URL in a native application.

Paradigm shift #1—when building a React Native app, we need to think of our routes in terms of *layers* rather than URL to UI mappings.

The routing layers in a native application structure themselves in a stack format, which roughly speaking is basically an array. When we transition between routes in React Native, we're simply pushing (navigating *into* a view) or popping (navigating *out of* a view) a route onto or from our *route stack*.

> **What is a route stack?**
>
> A *route stack* refers to an array of views a user visits throughout the application. When the user first opens the application, the route stack is going to have a length of 1 (the initial screen). Let's say that the user navigates to a *my account* screen next. The route stack will have two entries, the initial screen and the account screen.
>
> When the user navigates back to the home screen in our fictitious application, the route stack will *pop* (or remove the latest) view leaving the route stack to contain one entry again: the initial screen.

Coming from the web browser, this is a different paradigm than mapping a discrete URL. Instead of mapping to a unique URL, we will be managing an array mapping to a route stack.

In addition, we're not just mapping view components to a view, we'll also be working through how to make transitions between routes. Unlike the web, where we *typically* have independent route transitions, native apps generally need to know about each other as we navigate from one screen to the next.

For example, we'll usually have animations once our view renders, but not have a transition between routes themselves. However, in a native application, we'll still have these animations once our view renders as well as needing to devise a method that one route transitions into the next in the navigation stack.

Most route transitions on native devices are naturally hierarchical and should *generally* appropriately reflect our user's journey through each phase of the app.

In React Native, we can configure these route transitions through *scene configurations*. We'll take a look at how we can customize our *scene transitions* a little bit later, but for now we'll note that different platforms require different route transitions.

On the iOS platform, we're usually handling one of three different transitions between routes:

- Right to left
- Left to right
- Bottom to top

Typically when we push a route on iOS from the right when we're drilling deeper into the same hierarchy, we'll present a view from the bottom when it's a modal screen that is in a different hierarchy. We'll usually `push` a new view in the same hierarchy with the left to right transition and `pop` from the right to left transition after leaving a previous route.

The Android paradigm is slightly different, however. We'll still have the idea of drilling into more content, but rather than a hard left to right transition, we'll have a concept of creating an *elevation* change.

> ## Play around with it
>
> Regardless of the device we're building for, it's a good idea to open popular apps for the platform we're working with and taking note of what the route transitions look like between screens.

Enough theory. Let's get back to implementation. React Native routing has several options. At the time of writing, Facebook has indicated it's working on a new router component for React Native. For now, we'll stick with the tried and tested `Navigator` component to handle routing in our application.

`<Navigator />`

Just like the web version of the router, where we have React components that render to handle routing, the `<Navigator />` component is *just a React component*. This means we'll use it just like any other component in the view.

```
<Navigator />
```

Knowing the `<Navigator />` component is just a react component, we'll want to know:

1. What `props` does it accept?
2. Is it necessary to wrap it in a Higher Order Component?
3. How do we navigate with a single component?

Let's start off with #1, what `props` does it accept?

The `<Navigator />` only requires two `props`, both of them are functions:

- `configureScene()`
- `renderScene()`

Every time we change routes in our application, both the `configureScene()` and `renderScene()` functions will be called.

The `renderScene()` function is responsible for determining which UI (or component) to render next, while the `configureScene()` function is responsible for detailing out which transition type to make (as we looked at earlier).

In order to do their job, we'll need to receive some information about the specific route change that is *about to occur*. Naturally, since these are functions, they will be called with this information as an *argument* to the function.

Rather than talk about it, let's look at what this looks like in code. Starting with the boilerplate, we know now we'll be rendering a component:

src/app/index.js

```
export default class App extends Component {
  configureScene(route) {
    // Handle configuring scene
  }

  renderScene(route, navigator) {
    // handle rendering scene
  }

  render() {
    return (
      <Navigator
        renderScene={this.renderScene}
        configureScene={this.configureScene}
      />
    );
  }
}
```


Beautiful. We'll notice that this answers our second question from above. Typically when we're using the `<Navigator />` component, we'll wrap it in a *Higher Order Function* in order to handle our `renderScene()` and `configureScene()` methods.

`renderScene()`

Now let's take a look at how the `renderScene()` method needs to be implemented. We currently know two things about the `renderScene()` function:

1. It's purpose is to figure out which UI (or component) to render when transitioning into a new scene.
2. It receives an object which represent the route change that is going to occur.

With these two things in mind, let's look at an example of how a `renderScene()` might be implemented:

src/app/index.js

```
renderScene(route) {
  if (route.home === true) {
    return <HomeContainer />
  } else if (route.notifications === true) {
    return <NotificationsContainer />
  } else {
    return <FooterTabsContainer />
  }
}
```

Since the entire purpose of the `renderScene()` function is to determine which UI (or component) to render next, it makes sense that the `renderScene()` function is just one big *if* statement which renders different components based on a property of the `route` object.

The next natural question is "where is the `route` object coming from and how did it get the `home` and `notifications` properties on it?

This is where the react native configuration things get a little bit weird. Remember from above, we mentioned *every time we change routes in our applications, both `configureScene()` and `renderScene()` will be called*? Well, we still haven't answered just *how* we *change routes in our applications*. To answer that question, we'll need to look at the second argument the `renderScene()` function receives: the `navigator`:

```
renderScene(route, navigator) {
  // render scene
}
```

The `navigator` object is an instance object we can use to manipulate our current routing from within our application. The object itself contains some pretty handy methods, including the `push()` and `pop()` methods. The `push()` and `pop()` methods are how we're actually handling manipulating our route changes from within our app by pushing to or popping from the route stack.

Notice that we're receiving the `navigator` object inside our `renderScene()` method here, but the `renderScene()` method is for picking the next scene, not *invoking* route changes. In order to actually *use* the `navigator` object in our rendered scene, we'll need to pass it along to our new route as a `prop`:

src/app/index.js

```
renderScene(route, navigator) {
  if (route.home === true) {
    return <HomeContainer navigator={navigator} />
  } else if (route.notifications === true) {
    return <NotificationsContainer navigator={navigator} />
  } else {
    return <FooterTabsContainer navigator={navigator} />
  }
}
```

With this update, the `<HomeContainer />`, `<NotificationsContainer />`, and the `<FooterTabsContainer />` all have access to the `navigator` instance and can each call out to `push()` or `pop()` from the route stack using `this.props.navigator`.

For instance, let's say that we're working on our home route and we want to navigate to the notifications route. We can *push* to the navigation's route in a method like so:

src/app/containers/home.js

```
handleToNotifications() {
  this.props.navigator.push({
    notifications: true,
  });
}
```

Once we invoke the `handleToNotifications()` method inside the `<HomeContainer />` component, our `renderScene()` method will be called with the `route` argument, which is passed down through the `object` *pushed* onto by the `this.props.navigator.push()` method. When the `renderScene()`

sees the `route` object contains the `notifications` property as true, it returns the `<NotificationsContainer />` as the next component to render.

Don't worry quite yet if this didn't *click* right away. We'll be working with the `<Navigator />` component often. With some more experience under our belts, it will make more sense as we move along.

`configureScene()`

Now that we know how we'll change our routes in our app, we'll need to be able to tell our app *which* transition type to make based on the specific route change that will occur (where the transition type is *left to right*, *right to left*, *modal*, etc). From our discussion above, recall this takes place inside the `configureScene()` method we defined earlier:

```
configureScene(route) {
  // Configure the scene
}
```

The same route object passed to the `renderScene()` method is passed to the `configureScene()` method as well. We can use the same logic from inside the `configureScene()` method as we used in the `renderScene()` method:

src/app/index.js

```
configureScene(route) {
  if (route.home === true) {
    // Transitioning to HomeContainer
  } else if (route.notifications === true) {
    // Transitioning to NotificationsContainer
  } else {
    // Showing FooterTabsContainer
  }
}
```

Instead of rendering components here, we'll tell the `<Navigator />` component which transition type we want to use when transitioning to the new UI. The `<Navigator />` object comes with 10 different types of scene configurations[181], all of which are located on the `Navigator.SceneConfigs` object.

- `Navigator.SceneConfigs.PushFromRight` (default)
- `Navigator.SceneConfigs.FloatFromRight`

[181] http://facebook.github.io/react-native/releases/0.31/docs/navigator.html#configurescene

- `Navigator.SceneConfigs.FloatFromLeft`
- `Navigator.SceneConfigs.FloatFromBottom`
- `Navigator.SceneConfigs.FloatFromBottomAndroid`
- `Navigator.SceneConfigs.FadeAndroid`
- `Navigator.SceneConfigs.HorizontalSwipeJump`
- `Navigator.SceneConfigs.HorizontalSwipeJumpFromRight`
- `Navigator.SceneConfigs.VerticalUpSwipeJump`
- `Navigator.SceneConfigs.VerticalDownSwipeJump`

The configuration we return from the `configureScene()` method will be the transition type that will be used for the specific route change.

For instance, say our user is on iOS and we want every route besides our notifications route to have the standard *left to right* transition and we want the notifications route to come up from the bottom as a *modal* transition. We can implement this behavior with an update the `configureScene()` method like so:

src/app/index.js

```
  configureScene(route) {
    if (route.notifications === true) {
      return Navigator.SceneConfigs.FloatFromBottom;
    }

    return Navigator.SceneConfigs.FloatFromRight;
  }

  renderScene(route, navigator) {
    if (route.home === true) {
      return <HomeContainer navigator={navigator} />
    } else if (route.notifications === true) {
      return <NotificationsContainer navigator={navigator} />
    } else {
      return <FooterTabsContainer navigator={navigator} />
    }
  }

  render() {
    return (
      <Navigator
        renderScene={this.renderScene}
        configureScene={this.configureScene}
      />
```



```
    );
  }
```

Now, when our user navigates to our notifications view, we'll see a nice `FloatFromBottom` transition. For any other route, we'll get the default `FloatFromRight` transition as it's what we specified inside the `configureScene()` method.

Earlier, we talked about how Android has a fundamentally different route transition than iOS. Let's see how we can implement this as well.

In order to detect what platform our app is currently running on, we can use the `Platform` object exported by the `react-native` package. First, we need to get a hold of the `Platform` export from react-native:

src/app/index.js

```
import { Platform } from 'react-native';
```

The `Platform` object has a property called `OS` which has a string that shows which platform the app is currently using. For iOS, the string is `'ios'` and for android it is `'android'`.

src/app/index.js

```
  configureScene(route) {
    if (route.notifications === true) {
      if (Platform.OS === 'android') {
        return Navigator.SceneConfigs.FloatFromBottomAndroid;
      } else {
        return Navigator.SceneConfigs.FloatFromBottom;
      }
    }

    return Navigator.SceneConfigs.FloatFromRight;
  }

  renderScene(route, navigator) {
    if (route.home === true) {
      return <HomeContainer navigator={navigator} />
    } else if (route.notifications === true) {
      return <NotificationsContainer navigator={navigator} />
    } else {
      return <FooterTabsContainer navigator={navigator} />
    }
  }
```

```
render() {
  return (
    <Navigator
      renderScene={this.renderScene}
      configureScene={this.configureScene}
    />
  );
}
```

Now if we navigate to the notifications route while on an Android device (or the simulator), we'll get the `FloatFromBottomAndroid` transition and still get the `FloatFromBottom` while on an iOS device.

With the `<Navigator />` component in place, we now have routing working in our React Native app that works for both Android and iOS.

Web components vs. Native components

The first difference we'll encounter between React and React-Native is the difference of built-in components.

On the web, we can use the native browser components, such as `<div />`, `<a />`, `<span />`, etc. However, in a native application these elements don't exist. As we're relying on the underlying native UI layer to render our layouts, we can't use native web elements, instead we have to use elements that the UI builder knows how to render. Luckily for us, React-Native exports a bunch of these view elements for us out of the box that the underlying view managers understand (and they've made it pretty easy to create our own as well).

These elements aren't a huge conceptual difference for the most part, so we won't dive deep into the differences, but we'll look at a few of the built-in components which are most commonly used in creating a React Native application.

> The following list of built-in components are the most commonly used components. The React-Native ecosystem is active and engaged, so if your application needs a specific component, it may have already been built for you to use, if it's not already implemented into the React Native core.
>
> Before implementing your own component, check to see if it has been built either through https://www.npmjs.com/search?q=react-native[182] or js.coach/react-native[183].

[182] www.npmjs.com/search?q=react-native
[183] https://js.coach/react-native

`<View />`

The `<View />` component is the most fundamental component for building UIs with React Native. The `<View />` component maps directly to the native equivalent for the current platform where our app is running. It maps to the `UIView` on iOS, `android.view` on Android, and even (yes) `<div />` in the web. It's fair to use the `<View />` component the same way we would with a `<div />` on the web.

`<Text />`

The `<Text />` component is used for displaying text in a React Native app. It's important to note that we can't render any plain text that is *not* wrapped in a `<Text>text component</Text>`. The view layers don't know what to do with it, so we have to be explicit when adding plain text to the view.

`<Image />`

The `<Image />` component is used when we want to display any type of image, including network images, static resources, temporary local images, and images from the local device (such as the camera roll/library).

`<TextInput />`

Just like on the web, we can get user input from our users using an input field called `<TextInput />`. The `<TextInput />` component is the main way we can get input from a keyboard on the device. One particular `prop` it accepts is the `onChangeText()` property which is a function that it will call anytime the text in the input field changes.

`<TouchableHighlight />`, `<TouchableOpacity />`, and `<TouchableWithoutFeedback />`

When we want to listen for any press events in React Native, we'll need to use one of the *touchable* components. These *touchable* components are often a point of confusion when we first start out with React Native because in the web, we can just add an `onClick()` prop to any element and it becomes "touchable"."

However, in React Native not only is the component *not* `onClick()` (it's `onPress()`), but most components in React Native don't know what to do with the `onPress()` property. Because of this, we'll need to wrap any module we're interested in having a *touchable* property in one of these touchable components.

Each of the different touchable components has a slightly different effect when it's touched.

- `<TouchableHighlight />` component will decrease the opacity of the component, which will allow the underlay color to shine through.
- `<TouchableOpacity />` decreases the opacity of a component, but does not display an underlay color.
- `<TouchableWithoutFeedback />` invokes the component when it's pressed, but does not give any user feedback on the component itself.

We'll want to use the `<TouchableWithoutFeedback />` component sparingly as we'll most often want to give the user feedback when a *touchable* component is pressed.

Use `<TouchableWithoutFeedback />` sparingly

More often than not, we'll want to give the user feedback that a *touchable* component has been pressed. This is one of the main reasons mobile web browsers don't feel *native*.

`<ActivityIndicator />`

The default loading indicator for React Native is the `<ActivityIndicator />` component. This component works on multiple platforms as it's implemented entirely in JavaScript. One side benefit we get with a component entirely written in JavaScript is that the default styling is updated based upon the platform the app is operating on without any customization required.

`<WebView />`

The `<WebView />` component allows us to display some web content in our React Native application.

`<ScrollView />`

The `<ScrollView />` component allows us to *scroll* along a view. In the web, this is generally handled automatically (and can be controlled/defined in CSS). In a native app, however if our content goes off screen, we won't be able to see it. In a native app, our users expect to be able to scroll content when this happens. This is where we use the `<ScrollView />` component.

If our content is larger than the screen (meaning we'll have to scroll to see it), we'll wrap our `<View />` in a `<ScrollView />` component.

Keep in mind that the `<ScrollView />` component works best for a relatively small list of items since performant lists are critical for a good UX on mobile. The `<ScrollView />` component renders *all* the elements and views of a `<ScrollView />` are rendered regardless if they are shown on screen or not. Because of this, React Native providers another, more performant way to render larger lists with the `<ListView />`.

`<ListView />`

The `<ListView />` component is a performance-focused option for rendering long lists of data. Unlike the `<ScrollView />` component, the `<ListView />` only renders elements that are currently showing on the screen. One interesting note is that the `<ListView />` uses the `<ScrollView />` under the hood, but it adds a lot of nice performance abstractions for us to leverage.

As performant lists are fundamental in building native apps, and the ListView is a bit different than what we're used to in the web, let's dive into how to use the `<ListView />` in more detail.

Let's say that we're building an app similar to Twitter and we want to create a Feed component which will receive an array of tweets and render those tweets to the view. Let's look at how we'd do this on the Web, with ScrollView, then with ListView. (We'll purposefully neglect styling, as that's not the focus of this exercise).

We might implement this on the web like so:

Web version

src/feed.js

```
import PropTypes from 'prop-types';
import React from 'react';

Feed.propTypes = {
  tweets: PropTypes.arrayOf(PropTypes.shape({
    name: PropTypes.string.isRequired,
    user_id: PropTypes.string.isRequired,
    avatar: PropTypes.string.isRequired,
    text: PropTypes.string.isRequired,
    numberOfFavorites: PropTypes.number.isRequired,
    numberOfRetweets: PropTypes.number.isRequired,
  })).isRequired,
};

function Feed ({ tweets }) {
  return (
    <div>
      {tweets.map((tweet) => (
        <div>
          <div>
            <img alt="tweet" src={tweet.avatar} />
            <span>{tweet.name}</span>
          </div>
          <p>{tweet.text}</p>
```

```
          <div>
            <div>Favs: {tweet.numberOfFavorites}</div>
            <div>RTs: {tweet.numberOfRetweets}</div>
          </div>
        </div>
      )
      )}
    </div>
  );
}
```

We have a stateless functional component named Feed which takes in an array of tweets, maps over each of those, and displays some UI for each tweet (in this case, a `<div />` element with some children).

`<ScrollView />` version

We can implement the *same* functionality using the `<ScrollView />` component like so:

src/app/twitter-scrollview.js

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

Feed.propTypes = {
  tweets: PropTypes.arrayOf(PropTypes.shape({
    name: PropTypes.string.isRequired,
    user_id: PropTypes.string.isRequired,
    avatar: PropTypes.string.isRequired,
    text: PropTypes.string.isRequired,
    numberOfFavorites: PropTypes.number.isRequired,
    numberOfRetweets: PropTypes.number.isRequired,
  })).isRequired,
};

function Feed ({ tweets }) {
  return (
    <ScrollView>
      {tweets.map((tweet) => (
        <View>
          <View>
            <Image src={tweet.avatar} />
            <Text>{tweet.name}</Text>
```



```
          </View>
          <Text>{tweet.text}</Text>
          <View>
            <Text>Favs: {tweet.numberOfFavorites}</Text>
            <Text>RTs: {tweet.numberOfRetweets}</Text>
          </View>
        </View>
      )
      )}
    </ScrollView>
  );
}
```

Notice this implementation is pretty similar to what we're used to on the web. We've just swapped out the specific web components for their React Native counterpart, but the actual logic of mapping over each tweet is the same.

Now, let's look at how we can implement the same functionality using the more performant `<ListView />` element:

> Don't sweat the details here as we'll go over everything we need to know about *how* to use the `<ListView />` component shortly. We're using this as an example to highlight the differences between how we can think about implementing the same functionality.

src/app/twitter-listview.js

```
import React, { PropTypes, Component } from 'react';
import { View, Text, ScrollView, Image, ListView } from 'react-native';

class Feed extends Component {
  static props = {
    tweets: PropTypes.arrayOf(PropTypes.shape({
      name: PropTypes.string.isRequired,
      user_id: PropTypes.string.isRequired,
      avatar: PropTypes.string.isRequired,
      text: PropTypes.string.isRequired,
      numberOfFavorites: PropTypes.number.isRequired,
      numberOfRetweets: PropTypes.number.isRequired,
    })).isRequired,
  }

  constructor(props) {
```

```
    super(props);

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

    this.state = {
      dataSource: this.ds.cloneWithRows(this.props.tweets),
    };
  }

  componentWillReceiveProps(nextProps) {
    if (nextProps.tweets !== this.props.tweets) {
      this.setState({
        dataSource: this.ds.cloneWithRows(nextProps.tweets),
      });
    }
  }
  renderRow = ({ tweet }) => {
    return (
      <View>
        <View>
          <Image src={tweet.avatar} />
          <Text>{tweet.name}</Text>
        </View>
        <Text>{tweet.text}</Text>
        <View>
          <Text>Favs: {tweet.numberOfFavorites}</Text>
          <Text>RTs: {tweet.numberOfRetweets}</Text>
        </View>
      </View>
    );
  }
  render() {
    return (
      <ListView
        renderRow={this.renderRow}
        dataSource={this.state.dataSource}
      />
    );
  }
}
```


The `<ListView />` implementation a *lot* different. Let's walk through the differences.

The first thing we'll see is that we've switched from a stateless functional component to a Class component which allows us to keep `state`. Since our component needs to keep track of the data that's currently rendered on screen as well as listen for anytime we get a request to update the data through the `shouldComponentUpdate()` lifecycle hook.

When using a `<ListView />` component, we'll need two things to properly implement a `<ListView />` component:

- `ListView.DataSource` instance
- `renderRow()` function defined in our component

The `ListView.DataSource` instance is a bit out of left-field, so let's look at that first. We'll see that we start off by creating a new instance of the type `ListView.DataSource`. We'll pass it and object that allows us to customize how it works under the hood. Here, we've passed it a `rowHasChanged()` method, which is a function that the `<ListView />` uses to determine if the list needs to be re-rendered.

src/app/twitter-listview.js

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

Taking a step back for a moment, if we think about the fundamental aspect of creating a high-performance list view, making it easy to detect when a row in the list has changed in order to avoid re-rendering rows that haven't changed is pretty important. This is the first optimization that the `<ListView />` handles for us automatically.

The next step is to actually give our `ListView.DataSource` instance data to keep track of and show to the user. This is where the `ds.cloneWithRows()` method works. The `ds.cloneWithRows()` method will set (or update) the data kept internally by the `ListView.DataSource` instance.

src/app/twitter-listview.js

```
    this.state = {
      dataSource: this.ds.cloneWithRows(this.props.tweets),
    };
```

Notice that we're using the `ds.cloneWithRows()` method in both the `constructor()` function as well as the `componentWillReceiveProps()` methods.

src/app/twitter-listview.js

```
  componentWillReceiveProps(nextProps) {
    if (nextProps.tweets !== this.props.tweets) {
      this.setState({
        dataSource: this.ds.cloneWithRows(nextProps.tweets),
      });
    }
  }
```

In both cases, we are receiving tweets we want to show on the screen, so we'll need to make sure we add them to our `dataSource` instance variable, which means in both cases we'll need to call the `cloneWithRows()` method. Without calling the `cloneWithRows()` (or other relatives of the method – more on that later), the data won't be updates on the `ListView.DataSource` instance.

Immutable data

The data keeps by the `dataSource` instance object is *immutable*. *Immutable* objects cannot be updated or changed. Once it's created, it cannot be modified or updated.

When we do want to change the data, we'll have to create a separate copy of the data elsewhere so we can update a local copy and call `cloneWithRows()` with that data. Although this might seem like a limitation, it is a huge performance optimization and it's how we have to interact with the `ListView.DataSource` object.

In the previous example, we're not keeping a copy of the previous tweets because we're receiving a brand new array of tweets on every change. If these changes are incremental, however (adding/removing tweets) we would need to keep a copy of the *unDataSourced* data elsewhere.

Now that we have data kept inside the `dataSource` instance object, we'll need to implement a `renderRow()` function. This is a bit more straight-forward. The `renderRow()` method is called for every single row inside the `dataSource` instance's copy of the data and is responsible for providing a UI component to be rendered for every row.

src/app/twitter-listview.js

```
  renderRow = (tweet) => {
    return (
      <View>
        <View>
          <Image source={{ uri: tweet.avatar }} />
          <Text>{tweet.name}</Text>
        </View>
        <View>
          <Text>{tweet.text}</Text>
          <Text>Favs: {tweet.numberOfFavorites}</Text>
          <Text>RTs: {tweet.numberOfRetweets}</Text>
        </View>
      </View>
    );
  }
```

Notice that in converting from the `<ScrollView />` to the `<ListView />` implementation, we've take the `.map()` call from the `data` *out* of the `render()` function and into the `renderRow()` function. We can treat the JSX inside the `renderRow()` function just like it's the JSX from the `.map()` function as it is called for each item in the `dataSource` instance.

The `<ListView />` component accepts a few other props that we won't go "too" much into detail about here, but it's useful to know they exist. We'll look at the `renderSeparator()`, `renderHeader()`, and `renderFooter()`.

These extra `props` are pretty self-explanatory, but let's look at each one:

`renderSeparator()`

When a `renderSeparator()` prop is passed as a `prop` to the `<ListView />` component, it is expected to be a function that will be responsible for returning a component that will be rendered as a separator between each item in the `<ListView />`.

Rather than adding a border to the component's styles, using the `renderSeparator()` prop is intelligent and won't add a border to the last rendered row.

src/app/twitter-listview.js

```
  renderSeparator = (sectionId, rowId) => {
    return (
      <View key={rowId} style={styles.separator} />
    );
  }

  render() {
    return (
      <ListView
        renderRow={this.renderRow}
        dataSource={this.state.dataSource}
        renderSeparator={this.renderSeparator}
      />
    );
  }
```

renderHeader()

The `renderHeader()` prop accepts a function that is expected to return a component which will be used as the header for the `<ListView />` component. For instance, let's say we wanted to add a search bar at the top of our feed for filtering content. We can use the `renderHeader()` method as a way to achieve this.

src/app/twitter-listview.js

```
  renderHeader = () => <SearchBar />

  render() {
    return (
      <ListView
        renderRow={this.renderRow}
        dataSource={this.state.dataSource}
        renderHeader={this.renderHeader}
      />
    );
  }
```

renderFooter()

The `renderFooter()` prop allows us to add a footer to our ListView. For instance, we might want to show a button to allow the user to fetch more tweets from our tweet list, for example.

src/app/twitter-listview.js

```
  renderFooter = () => <ShowMoreTweets />

  render() {
    return (
      <ListView
        renderRow={this.renderRow}
        dataSource={this.state.dataSource}
        renderFooter={this.renderFooter}
      />
    );
  }
```

Styles

Styling in React Native is not as straight-forward as it is in the web. React Native takes the idea of styling with Cascading StyleSheets (or CSS, for short) to the native app world where we apply styling declarations to a component. As we've seen through our work with React throughout this course, we can apply CSS rules within a React component using the `style` prop on most elements. React Native follows this same approach with React Native elements.

Before we jump too much into *how* we style React Native components, it's important to note that React Native takes the `100%` of styling is done in JavaScript approach. That is, every core component accepts a `style` prop that accepts an object (or an array of objects) that contain a component's styles. As we're in JavaScript, properties that contain a `-`, however must be handled slightly differently. Rather than `background-color` or `margin-left`, we need to camel-case[184] the style name property.

For instance, if we want to add a style of a background color to a component we can add it

src/app/styledViews.js

```
  <View style={{ backgroundColor: 'green', padding: 10 }}>
    <Text style={{ color: 'blue', fontSize: 25 }}>
      Hello world
    </Text>
  </View>
```

One nice feature that React Native introduces that we don't have with the web version of React is the ability to pass an array to the style prop (without a separate library), which React Native will merge into a single style.

[184] https://en.wikipedia.org/wiki/Camel_case

src/app/styledViews.js

```
const ContainerComponent = () => {
  const getBackgroundColor = () => {
    return { backgroundColor: 'red' };
  };

  return (
    <View style={[ getBackgroundColor(), { padding: 10 } ]}>
      <Text style={{ color: 'blue', fontSize: 25 }}>
        Hello world
      </Text>
    </View>
  );
};
```

As it is right now, this is pretty simple. What if we have more than 2-3 style properties for a component? React Native exports a helper for just this case called `StyleSheet`. The `StyleSheet` export allows us to create an abstraction just like we can do with web-based CSS stylesheets.

StyleSheet

The `StyleSheet` object has a `create()` method on it which accepts an object that contains a list of our styles. We can then use the `styles` object the `create()` method returns rather than a raw styles object.

For example, let's decorate a few components:

src/app/styledViews.js

```
const styles = StyleSheet.create({
  container: {
    padding: 10,
  },
  containerText: {
    color: 'blue',
    fontSize: 20,
  },
});

class ExampleComponent extends Component {
  getBackgroundColor() {
    return {
      backgroundColor: 'yellow'
    };
  }

  render() {
    return (
      <View style={[
        this.getBackgroundColor(),
        styles.container
      ]}>
        <Text style={styles.containerText}>
          Hello world
        </Text>
      </View>
    );
  }
}
```


We get a few benefits to using the `StyleSheet` approach. One is that our components are more readable. Even more important is that `StyleSheet` gives us some performance gains that we don't get when applying an object directly to the `styles` prop.

Now that we've seen how to apply styles, let's jump into the architecture of our styles and using Flexbox.

Flexbox

Flexbox as a technology exists gives us the ability to define a layout in an efficient way. We can *lay out*, *align*, and *distribute space* among items in a container, even when the size of the components are unknown or are dynamic. Where creating a dynamic, all-purpose layout with CSS can be cumbersome, flexbox makes this much easier.

In other words, flexbox is all about creating dynamic layouts.

The main concept behind flexbox is that we give a parent element the ability to control the layout of all their child elements rather than having each child element control their own layout. When we give the parent this control, the parent element becomes a *flex container* where the child elements are referred to as *flex items*.

An example of this is instead of having to float all of the element's children left and add a margin to each, we can instead just have the parent element specify having all of its children to be laid out in a row with even space between them. In this way, the layout responsibilities move from the child to the parent component, which overall gives us more fine-tuned control over the layout of our apps.

The *most important* concept to understand about flexbox is that it's based on different axes. We have a *main* axis and a *cross* axis.

By default, in React Native the *main axis* is vertical while the *cross axis* is horizontal. Everything from here on out is built upon the concept of the axes. When we say "which will align all the child elements along the *main* axis," we mean that, by default the children of the parent element will be laid out vertically from top to bottom. When we say that this will align all the child elements on the *cross* axis, we mean, by default the children elements will be laid out horizontally from left to right.

The rest of the flexbox concepts is deciding how we want to align, position, stretch, spread, shrink, center, and wrap child elements along the main and cross axis.

Let's look at the flexbox properties.

`flex-direction`

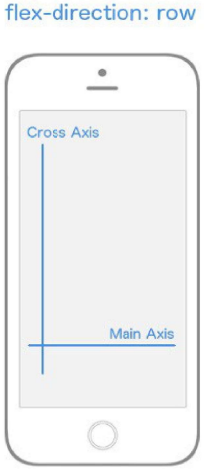
We have been very deliberate in saying the *default behavior* when talking about the main and cross axis. This is because we can actually change which axis is the main axis and which is the cross. We can use the `flex-direction` (in React Native, this is `flexDirection`) property to specify this property.

It can accept one of two values:

- `column`
- `row`

By default, React Native sets all elements to have the `flexDirection: column` property. This is to say that the element's main axis is vertical and it's cross axis is horizontal; just as we saw in the

image above. However, if we define the `flexDirection: row` to be *row*, the axes switch. The main axis becomes horizontal, while the cross axis is vertical.



It's crucial to understand the concepts of the *main* and *cross* axes as our *entire layout* depends upon these different settings.

Let's dive into a few different properties we can use to align child elements along these axes.

Focusing on the *main* axis first, in order to specify how children align themselves along the main axis, we can use the `justifyContent` property.

The `justifyContent` property can accept one of five different values we can use in order to change how the children align themselves along the main axis:

- `flex-start`
- `center`
- `flex-end`
- `space-around`
- `space-between`

Let's walk through what each of these mean.

Follow along

In this section, we *highly* recommend following along by building an application *with* us as we walk through this flexbox introduction.

We've included the sample applications with this book, or you can create your own simply and replace the `index.ios.js` and `index.android.js` code from the sample project with the following code sample.

To create a sample app for yourself, use the `react-native-cli` package:

```
react-native init FlexboxExamples
```

For the following examples, we'll use the following code to demonstrate how flexbox works with laying out child elements.

src/app/styledViews.js

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

export class FlexboxExamples extends Component {
  render() {
    return (
      <View style={ styles.container }>
        <View style={ styles.box } />
        <View style={ styles.box } />
        <View style={ styles.box } />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  box: {
    height: 50,
    width: 50,
    backgroundColor: '#e76e63',
    margin: 10,
  },
});
```



```
export default FlexboxExamples;
```

If you're creating your own example and replacing the `index.[platform].js` files, add the following code to make sure the app is *registered* with React native as the app:

```
import { AppRegistry } from 'react-native'; // at the top of the file
// ...
// at the end of the file
AppRegistry.registerComponent('FlexboxExamples', () => FlexboxExamples);
```

In this code, the only thing we'll be changing for every example is the `container` key of the `styles` object. We can ignore the `flex: 1;` for now (we'll come back to it later).

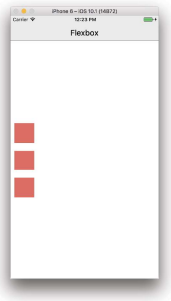
With the code above, we have an app that aligns the content along the main, vertical axis, adding the key `justifyContent: 'flex-start'` results in an app with every child element towards the start of the main axis:

justifyContent: 'flex-start'

```
container: {
  flex: 1;
  justifyContent: 'flex-start';
}
```

The default value for `flexDirection` is `column`, so when we don't have a value for our flex direction, it is set to `column`. Since `justifyContent` targets the *main* axis, our child elements align themselves towards the start of the *main* axis, which is the top left and work their way down.

When we set the value to `justifyContent: 'center'`, our child elements will align themselves towards the center of the main axis:



justifyContent: 'flex-center'

```
container: {
  flex: 1;
  justifyContent: 'center';
}
```

When we set `justifyContent` value to `flex-end`, our child elements align themselves toward the end of the main axis:

justifyContent: 'flex-end'


```
container: {
  flex: 1;
  justifyContent: 'flex-end';
}
```

We can also set our `justifyContent` to be `space-between`, which will align every child so that the space between each child item is even along the main axis:

justifyContent: 'space-between'

```
container: {
  flex: 1;
  justifyContent: 'space-between';
}
```

Finally, we can set our `justifyContent` value to `space-around`, which aligns every child elements so there is even space *around* each element on the main axis:

justifyContent: 'space-around'

```
container: {
  flex: 1;
  justifyContent: 'space-around';
}
```

What happens if we change the value of `flexDirection` of our container to `row` instead of to `column`?

The *main* axis switches to the *horizontal* axis and all of our child elements will still align themselves to the *main axis*, which is now vertical, rather than the *vertical* axis.

flexDirection: 'row'


```
container: {
  flex: 1;
  flexDirection: 'row',
  justifyContent: 'space-around';
}
```

Notice that all we did was change the `flexDirection` value and our layout is a completely new, updated layout. The ability to dynamically update our layout makes flexbox incredibly powerful.

Let's take a look at the cross axis. In order to specify how children align themselves along the cross axis, we'll use the `align-items` (or `alignItems` in React Native) property.

Unlike the `justifyContent` values, we only have four available values to set our `alignItems` property to:

- `flex-start`
- `center`
- `flex-end`
- `stretch`

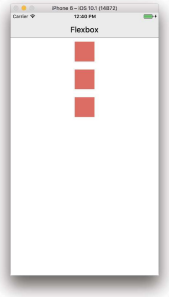
Let's walk through these as well.

When our container's `alignItems` value is set to `flex-start`, all of our child elements will align themselves towards the start of the cross axis:

alignItems: 'flex-start'

```
container: {
  flex: 1;
  alignItems: 'flex-start';
}
```

When our container's `alignItems` value is set to `center`, all of our child elements will align themselves towards the middle of the cross axis:



alignItems: 'center'

```
container: {
  flex: 1;
  alignItems: 'center';
}
```

When our container's `alignItems` value is set to `flex-end`, all of our child elements will align themselves towards the end of the cross axis:

alignItems: 'flex-end'

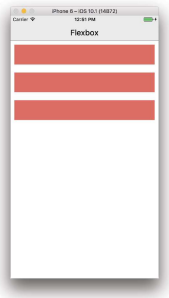
```
container: {
  flex: 1;
  alignItems: 'flex-end';
}
```

When we set our container's `alignItems` value to `stretch`, every child element along the cross axis will be stretched to the entire width of the cross-axis (provided there is not a specified `width` when our `flexDirection: row` or a `height` when our `flexDirection: column`).

Whenever we set `alignItems` to stretch, each child will *stretch* across the full width or height of the parent container, but only if the child element does *not* set a width. This makes sense as flexbox will try to set the width for our child elements automatically if the child doesn't try to override it.

To see this in action, check out the following screen-shots. Notice that we're including the `box` styles here as well to see that we've updated the `box` styles as well.

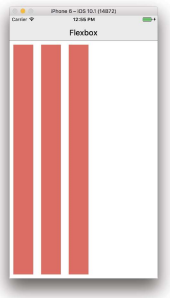
With the container set to: `flexDirection: column`, our layout will look like the following diagram. Notice we've removed the static `width` when we're using `flexDirection: column`:



alignItems: 'stretch'

```
container: {
  flex: 1;
  flexDirection: 'column',
  alignItems: 'stretch';
},
box: {
  height: 50,
  backgroundColor: '#e76e63',
  margin: 10
}
```

With the container set to: `flexDirection: row`, our layout will look like the following diagram. Notice we've removed the static `height` when we're using `flexDirection: row`:



alignItems: 'stretch'

```
container: {
  flex: 1;
  flexDirection: 'row',
  alignItems: 'stretch';
},
box: {
  width: 50,
  backgroundColor: '#e76e63',
  margin: 10
}
```

Breaking this down a bit, the main axis now runs horizontally since our `flexDirection: row` is set to `row`. This means the `alignItems` aligns the child elements along the vertical axis (the new *cross* axis). Since we've removed the height of the child elements and set the `alignItems` to `stretch`, the elements are now going to stretch along the vertical axis for the entire length of the parent component's entire cross axis view. For this example here, this is the entire view.

Through this point, we've been working with a single *flex container* or parent element. If we create nested *flex containers*, the *same logic* applies for the nested container child elements (flex items). Instead of being relative to the entire view (like in our previous examples), they'll position themselves according to the nested parent component. Our *entire UI will be built upon nesting flex containers.*

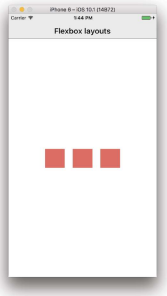
Other dynamic layouts

In flexbox, there is no such thing as a percentage-based styling. Although this *can* make things a bit more difficult, using flexbox to layout our UI is powerful as our layouts will look the way we want regardless of screen-size.

Recall in our earlier example, we set the `flex: 1` value to 1? The `flex` property allows us to specify *relative* widths for our flex containers.

As we've seen over and over, Flexbox is concerned with giving control over to the parent element to handle the layout of its children elements. The `flex` property is a bit different as it allows child elements to specify their height or width in comparison to their sibling elements. The best way to explain flex is to look at some examples,

Let's start with a view that looks like this:

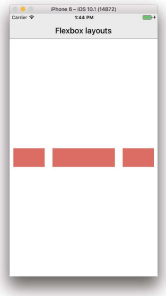


This is implemented using the following styles:

src/app/styledViews.js

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
  },
  box: {
    backgroundColor: '#e76e63',
    margin: 10,
    width: 50,
    height: 50,
  },
});
```

What if we want the UI to have two smaller boxes surrounding one larger box, like so:



We can use the *exact same* layout, but now the middle section is twice as wide as the two surrounding it. The `flex` property allows us to set this dynamically.

src/app/styledViews.js

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

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
  },
  box: {
    backgroundColor: '#e76e63',
    margin: 10,
    width: 50,
    height: 50,
  },
});

export class FlexboxLayouts extends Component {
  render() {
    return (
      <View style={[ styles.container ]}>
        <View style={[ styles.box, { flex: 1 } ]} />
        <View style={[ styles.box, { flex: 2 } ]} />
```

```
        <View style={[ styles.box, { flex: 1 } ]} />
      </View>
    );
  }
}

export default FlexboxLayouts;
```

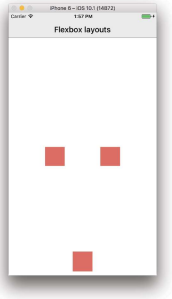
Notice that we didn't add to our styles, we just set the middle sibling's `flex` property to have a `flex: 2`, while the other siblings have a `flex: 1`.

We can read this like the flexbox layout is saying: "make sure the middle sibling is twice as large along the main axis as the first and third children." This is the reason why `flex` can replace percentage-based layouts. In a percentage-based layout, generally is one where the specific elements are relative in size to other elements, which is exactly what we're doing above.

It's important to note that if we place `flex: 1` on an element, the element is going to try to take up as much space as its parent takes up. In most our examples above, we want the "layout area" to be the size of the parent, which in our initial examples was the entire viewport.

Let's go deeeeeeper.

What if we wanted a layout like this:



It's as if the first and third elements are centered both vertically and horizontally, but the second element is aligned all the way at the bottom.

Breaking the layout down in flexbox terms, we basically have the first and third elements are aligned on the main axis, both `center`, while the second element uses `flex-end` along the cross axis, which is vertical. To implement this design, we'll need a way to have the child element override a specific positioning received from it's parent.

Good news! We have the `align-self` (`alignSelf` in React Native) property that allows us to have the child specify it's alignment directly. The `alignSelf` property positions itself among the *cross* axis and has the *same* options as `alignItems` (which makes sense, as it's the child telling the parent what to set it's `alignItems` property is for a single element).

src/app/styledViews.js

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

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
  },
  box: {
    backgroundColor: '#e76e63',
    margin: 10,
    width: 50,
    height: 50,
  },
});

export class FlexboxLayouts extends Component {
  render() {
    return (
      <View style={[ styles.container ]}>
        <View style={styles.box} />
        <View style={[ styles.box, { alignSelf: 'flex-end' } ]} />
        <View style={styles.box} />
      </View>
    );
  }
}

export default FlexboxLayouts;
```

Note that all we did to implement this layout was add a single `alignSelf: flex-end` property to the second child element which overrides the instruction set by the parent element (which is set to `center` in `styles.container`).

Let's look at the differences between flexbox for the web and for React Native.

When we hear the phrase "React Native uses Flexbox for styling," what this really means is that "React Native has it's own Flexbox (and CSS) implementation[185] which is *similar* to Flexbox, but it isn't an exact clone."

These differences can be summed up in two parts, *defaults* and *excluded* properties, where *defaults* which are automatically applied to every element and *excluded* properties are properties that exist in CSS/Flexbox that don't exist in React Native's implementation.

Let's look at the default values that are applied to every element for React Native;

```
box-sizing: border-box;
position: relative;

display: flex;
flex-direction: column;
align-items: stretch;
flex-shrink: 0;
align-content: flex-start;

border: 0 solid black;
margin: 0;
padding: 0;
min-width: 0;
```

Let's walk through some of the non-obvious ones.

`box-sizing: border-box`

The `box-sizing` property makes it so an element's specified width and height are *not* affected by padding or borders – which is what we expect to happen naturally, but browser implementations are weird and this isn't the case by default.

`flex-direction: column`

The `flex-direction` on the web is set to `row`, but the default for `flexDirection` in React Native is `column`.

`align-items: stretch`

The default `alignItems` property is set to `stretch`, whereas in the web it's set to `flex-start`.

[185] https://github.com/facebook/yoga

`position: relative`

By having every element set to relative positioning by default, it makes absolute positioned child elements target the direct parent rather than the typical "closest parent with position relative or absolute". This makes using absolute positioning more consistent while also allowing for left, right, top, and bottom values by default.

`display: flex`

Unlike the web implementation, there is no need to set `display: flex` ever as every element in a React Native app is already set to `display: flex` by default.

Now let's look at the properties that exist on the web, but don't exist for React Native's implementation.

`flex-grow`, `flex-shrink`, `flex-basis`

The three properties available in the web flexbox, which allow an element to grow or shrink a flex item. React Native has a similar property called `flex`, but don't have the same three elements that *do* exist in CSS. It's also important to note that the `flex` property in React Native works differently than in the web.

`flex` on React Native is a number rather than a string and is used to make a specific component larger or smaller than its sibling components. A component with flex set to `2` will take twice the space (either vertically or horizontally depending on its primary axis) as a component with flex set to `1`. If flex is set to `0`, that component will be sized according to its width and height. This brings us to our next "exclude" which is any size unit besides `px`.

In order to accomplish percentage-based layouts, we'll have to use `flex` as we just talked about in the previous section.

Although we did not talk about them directly, React Native allows us to use CSS properties that we're already used to in CSS, like position, zIndex, minWidth, etc. A full list of all the available properties that React Native knows about are available at https://facebook.github.io/react-native/docs/layout-props.html[186].

HTTP requests

We're only going to get so far in building a React Native app without needing to make an HTTP request and fetch some data from an external API. React Native includes an abstraction to make HTTP requests using the `fetch` API. The fetch[187] API provides an interface for fetching resources, which will seem pretty familiar for anyone who has used _XMLHttpRequest‘ or other clients such as axios[188] and superagent[189].

[186] https://facebook.github.io/react-native/docs/layout-props.html
[187] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
[188] https://www.npmjs.com/package/axios
[189] https://github.com/visionmedia/superagent

The `fetch` API uses promises, so before we get to jumping into using the `fetch` API with React Native, let's take a detour and talk about Promises.

What is a promise

As defined by the Mozilla, a `Promise` object is used for handling asynchronous computations which has some important guarantees that are difficult to handle with the callback method (the more old-school method of handling asynchronous code).

A `Promise` object is simply a wrapper around a value that may or may not be known when the object is instantiated and provides a method for handling the value *after* it is known (also known as `resolved`) or is unavailable for a failure reason (we'll refer to this as `rejected`).

Using a `Promise` object gives us the opportunity to associate functionality for an asynchronous operation's eventual success or failure (for whatever reason). It also allows us to treat these complex scenarios by using synchronous-like code.

For instance, consider the following synchronous code where we print out the current time in the JavaScript console:

```
var currentTime = new Date();
console.log('The current time is: ' + currentTime);
```

This is pretty straight-forward and works as the `new Date()` object represents the time the browser knows about. Now consider that we're using a different clock on some other remote machine. For instance, if we're making a Happy New Years clock, it would be great to be able to synchronize the user's browser with everyone Else's using a single time value for everyone so no-one misses the ball dropping ceremony.

Suppose we have a method that handles getting the current time for the clock called `getCurrentTime()` that fetches the current time from a remote server. We'll represent this now with a `setTimeout()` that returns the time (like it's making a request to a slow API):

```
function getCurrentTime() {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    return new Date();
  }, 2000);
}
var currentTime = getCurrentTime()
console.log('The current time is: ' + currentTime);
```

Our `console.log()` log value will return the timeout handler id, which is definitely *not* the current time. Traditionally, we can update the code using a callback to get called when the time is available:


```
function getCurrentTime(callback) {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    var currentTime = new Date();
    callback(currentTime);
  }, 2000);
}
getCurrentTime(function(currentTime) {
  console.log('The current time is: ' + currentTime);
});
```

What if there is an error with the rest? How do we catch the error and define a retry or error state?

```
function getCurrentTime(onSuccess, onFail) {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    // randomly decide if the date is retrieved or not
    var didSucceed = Math.random() >= 0.5;
    if (didSucceed) {
      var currentTime = new Date();
      onSuccess(currentTime);
    } else {
      onFail('Unknown error');
    }
  }, 2000);
}
getCurrentTime(function(currentTime) {
  console.log('The current time is: ' + currentTime);
}, function(error) {
  console.log('There was an error fetching the time');
});
```

Now, what if we want to make a request based upon the first requests value? As a short example, let's reuse the `getCurrentTime()` function inside again (as though it were a second method, but allows us to avoid adding another complex-looking function):

```
function getCurrentTime(onSuccess, onFail) {
  // Get the current 'global' time from an API
  return setTimeout(function() {
    // randomly decide if the date is retrieved or not
    var didSucceed = Math.random() >= 0.5;
    console.log(didSucceed);
    if (didSucceed) {
      var currentTime = new Date();
      onSuccess(currentTime);
    } else {
      onFail('Unknown error');
    }
  }, 2000);
}
getCurrentTime(function(currentTime) {
  getCurrentTime(function(newCurrentTime) {
    console.log('The real current time is: ' + currentTime);
  }, function(nestedError) {
    console.log('There was an error fetching the second time');
  })
}, function(error) {
  console.log('There was an error fetching the time');
});
```

Dealing with asynchronousity in this way can get complex quickly. In addition, we could be fetching values from a previous function call, what if we only want to get one... there are a lot of tricky cases to deal with when dealing with values that are not yet available when our app starts.

Enter Promises

Using promises, on the other hand helps us avoid a lot of this complexity (although is not a silver bullet solution). The previous code, which could be called spaghetti code can be turned into a neater, more synchronous-looking version:


```
function getCurrentTime(onSuccess, onFail) {
  // Get the current 'global' time from an API using Promise
  return new Promise((resolve, reject) => {
    setTimeout(function() {
      var didSucceed = Math.random() >= 0.5;
      didSucceed ? resolve(new Date()) : reject('Error');
    }, 2000);
  })
}
getCurrentTime()
  .then(currentTime => getCurrentTime())
  .then(currentTime => {
    console.log('The current time is: ' + currentTime);
    return true;
  })
  .catch(err => console.log('There was an error:' + err))
```

This previous source example is a bit cleaner and clear as to what's going on and avoids a lot of tricky error handling/catching.

To catch the value on success, we'll use the `then()` function available on the `Promise` instance object. The `then()` function is called with whatever the return value is of the promise itself. For instance, in the example above, the `getCurrentTime()` function resolves with the `currentTime()` value (on successful completion) and calls the `then()` function on the return value (which is another promise) and so on and so forth.

To catch an error that occurs anywhere in the promise chain, we can use the `catch()` method.

> We're using a promise chain in the above example to create a *chain* of actions to be called one after another. A promise chain sounds complex, but it's fundamentally simple. Essentially, we can "synchronize" a call to multiple asynchronous operations in succession. Each call to `then()` is called with the previous `then()` function's return value.
>
> For instance, if we wanted to manipulate the value of the `getCurrentTime()` call, we can add a link in the chain, like so:
>
> ```
> getCurrentTime()
>   .then(currentTime => getCurrentTime())
>   .then(currentTime => {
>     return 'It is now: ' + currentTime;
>   })
>   // this logs: "It is now: [current time]"
>   .then(currentTimeMessage => console.log(currentTimeMessage))
>   .catch(err => console.log('There was an error:' + err))
> ```

Single-use guarantee

A promise only ever has one of three states at any given time:

- pending
- fulfilled (resolved)
- rejected (error)

A *pending* promise can only every lead to either a fulfilled state or a rejected state *once and only once*, which can avoid some pretty complex error scenarios. This means that we can only ever return a promise once. If we want to rerun a function that uses promises, we need to create a *new* one.

Creating a promise

We can create new promises (as the example shows above) using the `Promise` constructor. It accepts a function that will get run with two parameters:

- The `onSuccess` (or `resolve`) function to be called on success resolution
- The `onFail` (or `reject`) function to be called on failure rejection

Recalling our function from above, we can see that we call the `resolve()` function if the request succeeded and call the `reject()` function if the method returns an error condition.

```
var promise = new Promise(function(resolve, reject) {
  // call resolve if the method succeeds
  resolve(true);
})
promise.then(bool => console.log('Bool is true'))
```

Now that we have a familiarity with promises, let's use this in a React Native app. The simplest possible GET implementation to make an HTTP request, the `fetch` method accepts a URL as it's first argument and returns a promise which, when resolved, contains the response object.

For instance, let's say we have a `getGithubUsers()` function where we want to make an API call to fetch some users from github. This `fetch` call could be implemented like so:

src/app/styledViews.js

```
const baseUrl = 'https://api.github.com';

export const getGithubUsers = ({ offset }) => {
  return fetch(`${baseUrl}/users?since=${offset}`)
          .then(response => response.json())
          .catch(console.warn);
};
```

It's possible to use `fetch` for more than just GET requests, of course. We can specify more details for our requests by passing a second argument which contains more details along with the request.

For example, let's say we wanted to create a `gist` on github using the github API. We can easily handle this by using the `fetch` API with a second parameter. We might implement this like so:

src/app/styledViews.js

```
export const makeGist = (
  activity,
  { description = '', isPublic = true } = {}
  ) => fetch(`${baseUrl}/gists`, {
    method: 'POST',
    body: JSON.stringify({
      files: {
        'activity.json': {
          content: JSON.stringify(activity),
        },
      },
      description,
      public: isPublic,
    }),
  })
  .then(response => response.json());
```

The fetch API has a lot of possibilities and can be used across platforms with React Native's implementation. For more details about the API, check out the docs on MDN at https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch[190].

[190] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

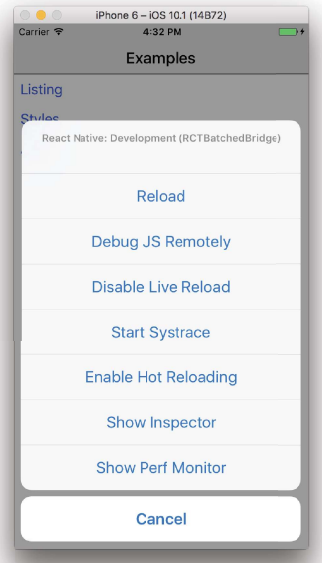
Debugging with React Native

One of the very nicest features that comes along with writing a React Native app is the debugging experience. One of the main goals of the React Native team is to take the development work-flow we're used to working with on the web and bring it to native development.

When we're running our app on the simulator, we can open the debugging menu by platform by pressing:

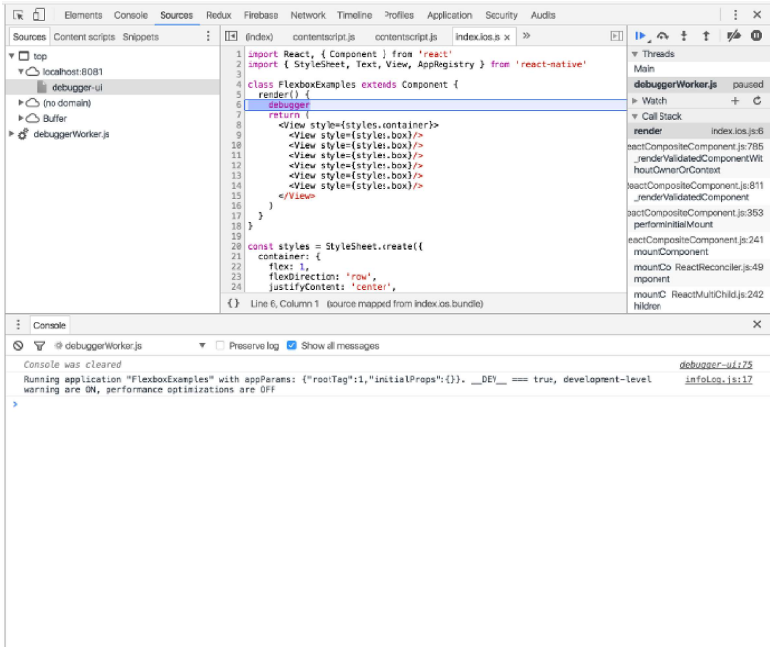
- iOS: `Cmd + D` (or `CTRL + D` on a PC)
- Android: Cmd + M (or `CTRL + M` on a PC)

If this works, this will bring up the in-app developer menu:



From this menu, we have a bunch of different options for debugging. The one we'll work with directly is the menu item with the label: `Debug JS Remotely`. Clicking on this menu item will bring up a window in Chrome[191].

If we open the developer console on the page it opens, we'll see that we can actually debug out native application using the Chrome dev tools!



Where to go from here

The React Native community is active and growing all the time.

The number of React Native meetups is growing all the time and we highly recommend checking one out. The list of world-wide React Native meetups can be found on meetup.com at:

[191] https://google.com/chrome

https://www.meetup.com/topics/react-native/[192].

We are available in the chat room associated with the book, so feel free to stop in and say hello.

The React Native team has an open React Native chat channel available through Discord, which can be found at: https://discordapp.com/invite/0ZcbPKXt5bZjGY5n[193].

Community resources can be found on npmjs.org at https://www.npmjs.com/search?q=react-native[194].

[192] https://www.meetup.com/topics/react-native/
[193] https://discordapp.com/invite/0ZcbPKXt5bZjGY5n
[194] https://www.npmjs.com/search?q=react-native