

Part 2

Developing applications in React Native

With the basics covered, you can start adding features to your React Native app. The chapters in this part cover styling, navigation, animations, and elegant ways to handle data using data architectures (with a focus on Redux).

Chapters 4 and 5 teach how to apply styles either inline with components or in stylesheets that components can reference. And because React Native components are the main building blocks of your app's UI, chapter 4 spends some time teaching useful things you can do with the `View` component. Chapter 5 builds on the skills taught in chapter 4. It covers aspects of styling that are platform specific, as well as some advanced techniques, including using flexbox to make it easier to lay out an application.

Chapter 6 shows how to use the two most-recommended and most-used navigation libraries, React Navigation and React Native Navigation. We walk through creating the three main types of navigators—tabs, stack, and drawer—and how to control the navigation state.

Chapter 7 covers the four things you need to do to create animations, the four types of animatable components that ship with the Animated API, how to create custom animatable components, and several other useful skills.

In chapter 8, we explore handling data with data architectures. Because Redux is the most widely adopted method of handling data in the React ecosystem, you use it to build an app, meanwhile learning data-handling skills. We show how to use the Context API and how to implement Redux with a React Native app by using reducers to hold the Redux state and delete items from the example app. We also cover how to use providers to pass global state to the rest of the app, how to use the `connect` function to access the example app from a child component, and how to use actions to add functionality.

4

Introduction to styling

This chapter covers

- Styling using JavaScript
- Applying and organizing styles
- Applying styles to `View` components
- Applying styles to `Text` components

It takes talent to build mobile applications, but it takes *style* to make them great. If you're a graphic designer, you know this intuitively, deep in your bones. If you're a developer, you're probably groaning and rolling your eyes. In either case, understanding the fundamentals of styling Reactive Native components is critical to making an engaging application that others want to use.

In all likelihood, you have some experience with CSS, even if it's nothing more than seeing the syntax. You can easily understand what a CSS rule like `background-color: 'red'` is meant to do. As you begin reading this chapter, it may appear as though styling components in React Native is as simple as using camelCase names for CSS rules. For instance, setting the background color on a React Native component uses almost the same syntax, `backgroundColor: 'red'`—but be forewarned, this is where the similarities end.

Try not to hang on to how you did things in CSS. Embrace the React Native way, and you'll find that learning how to style components is a much more pleasant experience—even for a developer.

The first section of this chapter provides an overview of styling components. We'll make sure you understand the various ways to apply styles to components and discuss how to organize styles in an application. Forming good organizational habits now will make things easier to manage and will facilitate the use of more advanced techniques down the road.

Because React Native is styled using JavaScript, we'll talk about how to start thinking of styles as code and how to take advantage of JavaScript features like variables and functions. The final two sections explore styling `View` components and `Text` components. In some cases, we'll use short examples to explain a topic, but for the most part, we'll walk through styling something real. You'll take what you learn and apply it to the construction of a Profile Card.

For all the example code in this chapter, you can start with the default generated app and replace the contents of `App.js` with the code from the individual listings. Complete source files can be found at www.manning.com/books/react-native-in-action and in the book's Git repository at <https://github.com/dabit3/react-native-in-action> under chapter-4.

4.1 Applying and organizing styles in React Native

React Native comes with many built-in components, and the community has built many more you can include with your projects. Components support a specific set of styles. Those styles may or may not be applicable to other types of components. For example, the `Text` component supports the `fontWeight` property (`fontWeight` refers to the thickness of the font), but the `View` component doesn't. Conversely, the `View` component supports the `flex` property (`flex` refers to the layout of components within a view), but the `Text` component doesn't.

Some styling elements are similar between components but not the same. For example, the `View` component supports the `shadowColor` property, whereas the `Text` component supports the `textShadowColor` property. Some styles, like `ShadowPropTypesIOS`, only apply to a specific platform (in this case, to iOS).

Learning the various styles and how to manipulate them takes time. That's why it's important to start with fundamentals like how to apply and organize styles. This section will focus on teaching those styling fundamentals, so you'll have a good foundation from which to start exploring styles and building the example Profile Card component.

TIP For a solid reference on how to make mobile apps usable, see Matt Lacey's *Usability Matters* (Manning, 2018; www.manning.com/books/usability-matters).

4.1.1 Applying styles in applications

To compete in the marketplace, mobile applications must have a sense of style. You can develop a fully functional app, but if it looks terrible and isn't engaging, people aren't

going to be interested. You don't have to build the hottest-looking app in the world, but you do have to commit to creating a polished product. A polished, sharp-looking app greatly influences people's perception of the app's quality.

You can apply styles to elements in React Native in a number of ways. In chapters 1 and 3, we went over inline styling (shown in the next listing) and styling using a StyleSheet (listing 4.2).

Listing 4.1 Using inline styles

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

export default class App extends Component {
  render () {
    return (
      <View style={{marginLeft: 20, marginTop: 20}}>
        <Text style={{fontSize: 18,color: 'red'}}>Some Text</Text>
      </View>
    )
  }
}
```

Applies an inline style to a React Native component

Applies multiple inline styles at once

As you can see, it's possible to specify multiple styles at once by supplying an object to the styles property.

Listing 4.2 Referencing styles defined in a StyleSheet

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

export default class App extends Component {
  render () {
    return (
      <View style={styles.container}>
        <Text style={[styles.message,styles.warning]}>Some Text</Text>
      </View>
    )
  }
}
```

References the container style defined in the styles stylesheet

Uses an array to reference both the message and warning styles from the stylesheet

```
const styles = StyleSheet.create({
  container: {
    marginLeft: 20,
    marginTop: 20
  },
  message: {
    fontSize: 18
  },
  warning: {
    color: 'red'
  }
});
```

Defines the styles using StyleSheet.create

Functionally, there's no difference between using an inline style versus referencing a style defined in a `StyleSheet`. With `StyleSheet`, you create a style object and refer to each style individually. Separating the styles from the render method makes the code easier to understand and promotes reuse of styles across components.

When using a style name like `warning`, it's easy to recognize the intent of the message. But the inline style `color: 'red'` offers no insight into why the message is red. Having styles specified in one place rather than inline on many components makes it easier to apply changes across the entire application. Imagine you wanted to change warning messages to yellow. All you have to do is change the style definition once in the stylesheet, `color: 'yellow'`.

Listing 4.2 also shows how to specify multiple styles by supplying an array of style properties. Remember when doing this that the last style passed in will override the previous style if there's a duplicate property. For example, if an array of styles like this is supplied, the last value for `color` will override all the previous values:

```
style=[{color: 'black'}, {color: 'yellow'}, {color: 'red'}]
```

In this example, the color will be red.

It's also possible to combine the two methodologies by specifying an array of styling properties using inline styles and references to stylesheets:

```
style=[{color: 'black'}, styles.message]
```

React Native is very flexible in this regard, which can be both good and bad. Specifying inline styles when you're quickly trying to prototype something is extremely easy, but in the long haul, you'll want to be careful how you organize your styles; otherwise your application can quickly become a mess and difficult to manage. By organizing your styles, you'll make it easier to do the following:

- Maintain your application's codebase
- Reuse styles across components
- Experiment with styling changes during development

4.1.2 **Organizing styles**

As you might suspect from the previous section, using inline styles isn't the recommended way to go: stylesheets are a much more effective way to manage styles. But what does that mean in practice?

When styling websites, we use stylesheets all the time. Often we use tools like Sass, Less, and PostCSS to create monolithic stylesheets for the entire application. In the world of the web, styles are in essence global, but that isn't the React Native way.

React Native focuses on the component. The goal is to make components as reusable and standalone as possible. Having a component dependent on an application's stylesheet is the antithesis of modularity. In React Native, styles are scoped to the component—not to the application.

How to accomplish this encapsulation depends entirely on your team's preference. There's no right or wrong way, but in the React Native community, you'll find two common approaches:

- Declaring stylesheets in the same file as the component
- Declaring stylesheets in a separate file, outside of the component

DECLARING STYLESHEETS IN THE SAME FILE AS THE COMPONENT

As you've done so far in this book, a popular way to declare styles is within the component that will be using them. The major benefit of this approach is that the component and its styles are completely encapsulated in a single file. This component can then be moved or used anywhere in the app. This is a common approach to component design, one you'll see often in the React Native community.

When including the stylesheet definitions with the component, the typical convention is to specify the styles after the component. All the listings in this book have, so far, followed this convention.

DECLARING STYLESHEETS IN A SEPARATE FILE

If you're used to writing CSS, putting your styles into a separate file might seem like a better approach and feel more familiar. The stylesheet definitions are created in a separate file. You can name it whatever you want (styles.js is typical), but be sure the extension is .js; it's JavaScript, after all. The stylesheet file and component file are saved in the same folder.

A file structure like that shown in figure 4.1 retains the close relationship between components and styles and affords a bit of clarity by not mixing style definitions with the functional aspects of the components. Listing 4.3 corresponds to a styles.js file that would be used to style a component like ComponentA and ComponentB in the figure. Use meaningful names when defining your stylesheets, so it's clear what part of a component is being styled.

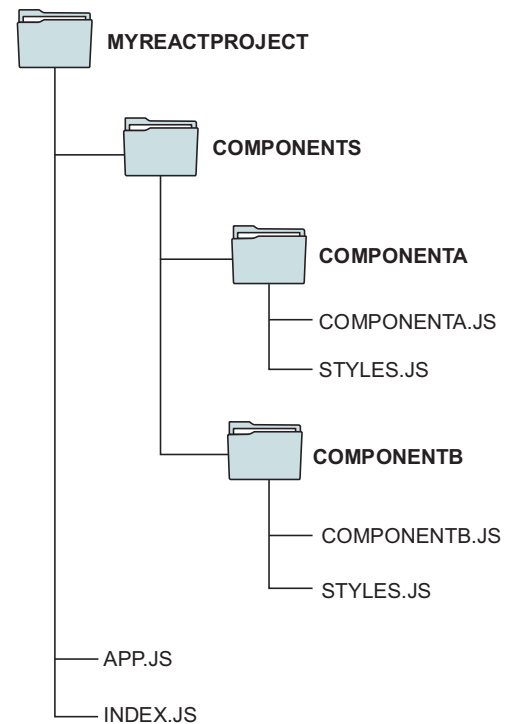


Figure 4.1 An example file structure with styles separated from components in a single folder instead of a single file

Listing 4.3 Externalizing a component's stylesheets

```
import { StyleSheet } from 'react-native'

const styles = StyleSheet.create({
  container: {
    marginTop: 150,
    backgroundColor: '#ededed',
    flexWrap: 'wrap'
  }
})

const buttons = StyleSheet.create({
  primary: {
    flex: 1,
    height: 70,
    backgroundColor: 'red',
    justifyContent: 'center',
    alignItems: 'center',
    marginLeft: 20,
    marginRight: 20
  }
})

export { styles, buttons }
```

Creates a stylesheet, and saves it in the styles constant

Defines a style for the container. It can be referenced by the component as styles.container.

Creates a second stylesheet, and saves it in the buttons constant

Defines a style for the primary button. It can be referenced by the component as buttons.primary.

Exports both the styles and buttons stylesheets so the component will have access to the constants

The component imports the external stylesheets and can reference any styles defined within them.

Listing 4.4 Importing external stylesheets

```
import { styles, buttons } from '../component/styles'

<View style={styles.container}>
  <TouchableHighlight style={buttons.primary} />
  ...
</TouchableHighlight>
</View>
```

Imports multiple stylesheets exported from styles.js

Reference to the buttons.primary style created in styles.js

Reference to the styles.container style created in styles.js

4.1.3 Styles are code

You've already seen how JavaScript is used to define styles in React Native. Despite having a full scripting language with variables and functions, your styles have been rather static, but they certainly don't have to be!

Web developers have fought with CSS for years. New technologies like Sass, Less, and PostCSS were created to work around the many limitations of cascading stylesheets. Even a simple thing like defining a variable to store the primary color of a site was impossible without CSS preprocessors. The CSS Custom Properties for Cascading Variables Module Level 1 candidate recommendation in December 2015 introduced the concept of custom properties, which are akin to variables; but at the time of writing, fewer than 80% of browsers in use support this functionality.

Let's take advantage of the fact that we're using JavaScript and start thinking of styles as code. You'll build a simple application that gives the user a button to change the theme from light to dark. But before you start coding, let's walk through what you're trying to build.

The application has a single button on the screen. That button is enclosed by a small square box. When the button is pressed, the themes will toggle. When the light theme is selected, the button label will say White, the background will be white, and the box around the button will be black. When the dark theme is selected, the button label will say Black, the background will be black, and the box around the button will be white. Figure 4.2 shows what the screen should look like when the themes are selected.

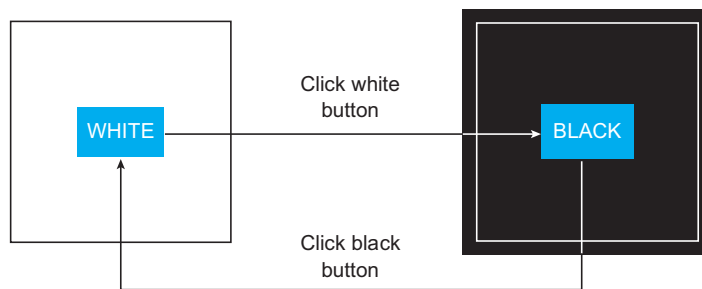


Figure 4.2 A simple application that supports two themes, white and black. Users can press the button to toggle between a white background and a black background.

For this example, organize the styles in a separate file, `styles.js`. Then, create some constants to hold the color values, and create two stylesheets for the light and dark themes.

Listing 4.5 Dynamic stylesheets extracted from the main component file

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

export const Colors = {
  dark: 'black',
  light: 'white'
};

const baseContainerStyles = {
  flex: 1,
  justifyContent: 'center',
  alignItems: 'center'
};

const baseBoxStyles = {
  justifyContent: 'center',
  alignItems: 'center',
  borderWidth: 2,
  height: 150,
  width: 150
};

const lightStyleSheet = StyleSheet.create({
```

Constant defining the colors that will correspond to the light and dark themes

JavaScript object to hold the base container styles

JavaScript object to hold the base box styles

Creates the stylesheet for the light theme

```

    container: {
      ...baseContainerStyles,
      backgroundColor: Colors.light
    },
    box: {
      ...baseBoxStyles,
      borderColor: Colors.dark
    }
  }
});

```

```

const darkStyleSheet = StyleSheet.create({
  container: {
    ...baseContainerStyles,
    backgroundColor: Colors.dark
  },
  box: {
    ...baseBoxStyles,
    borderColor: Colors.light
  }
});

```

Creates the stylesheet
for the dark theme

Returns the dark
theme if useDarkTheme
is true; otherwise
returns the light theme

Function that
will return the
appropriate
theme based
on a Boolean
value

```

export default function getStyleSheet(useDarkTheme){
  return useDarkTheme ? darkStyleSheet : lightStyleSheet;
}

```

Once the styles have been configured, you can start building the component app in App.js. Because you only have light and dark themes, create a utility function, `getStyleSheet`, which takes a Boolean value. If `true` is supplied, the dark theme will be returned; otherwise the light theme will be returned.

Listing 4.6 Application that toggles between light and dark themes

```

import React, { Component } from 'react';
import { Button, StyleSheet, View } from 'react-native';
import getStyleSheet from './styles';

```

```

export default class App extends Component {

```

Imports the `getStyleSheet` function
from the externalized styles

```

  constructor(props) {
    super(props);
    this.state = {
      darkTheme: false
    };
    this.toggleTheme = this.toggleTheme.bind(this);
  }

```

Initializes the component's state
to show the light theme by default

To avoid exceptions, the
`toggleTheme` function must
be bound to the component.

Toggles the
theme value
in state
whenever the
function is
called

```

  toggleTheme() {
    this.setState({darkTheme: !this.state.darkTheme});
  }

```

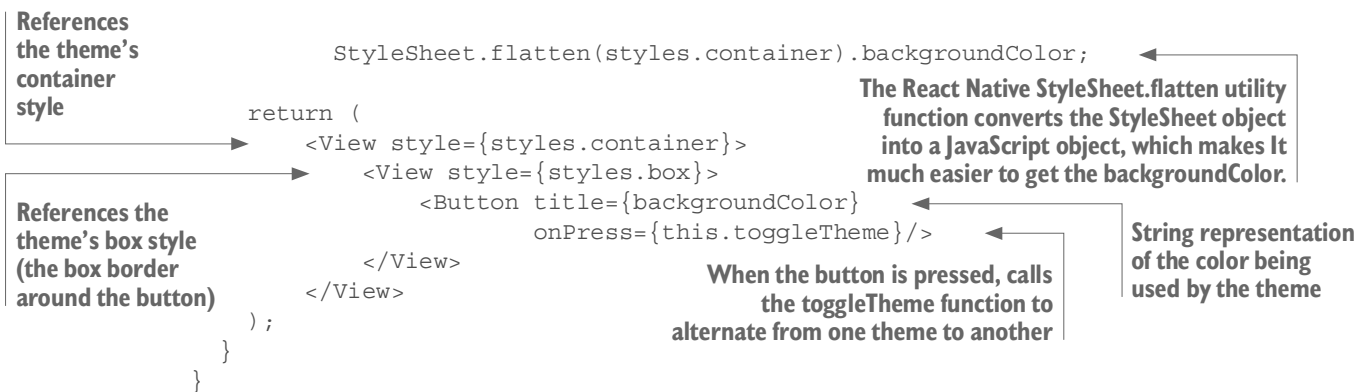
Uses the imported `getStyleSheet` function
to get the appropriate stylesheet for
whichever theme should be displayed

```

  render() {

    const styles = getStyleSheet(this.state.darkTheme);
    const backgroundColor =

```



The application toggles themes: feel free to experiment and take it a bit further. Try changing the light theme to a different color. Notice how easy it is, because the colors are defined as constants in one place. Try changing the button label in the dark theme to be the same color as the background instead of always white. Try creating an entirely new theme, or modify the code to support many different themes instead of just two—have fun!

4.2 Styling view components

Now that you've had a proper overview of styling in React Native, let's talk more about individual styles. This chapter covers many of the basic properties you'll use all the time. In chapter 5, we'll go into more depth and introduce styles you won't see every day and styles that are platform specific. But for now, let's focus on the basics: in this section, that's the View components. The View component is the main building block of a UI and is one of the most important components to understand to get your styling right. Remember, a View element is similar to an HTML `div` tag in the sense that you can use it to wrap other elements and build blocks of UI code in it.

As you progress through the chapter, you'll use what you've learned to build a real component: a Profile Card. Building the Profile Card will show how to put everything together. Figure 4.3 shows what the component will look like at the end of this section. In the process of creating this component, you'll learn how to do the following:

- Create a border around the profile container using `borderWidth`
- Round the corners of that border with `borderRadius`
- Create a border that looks like a circle by using a `borderRadius` half the size of the component's width
- Position everything using `margin` and `padding` properties

The next few sections will teach the styling techniques you'll need to know to create the Profile Card component. We'll start easy by talking about how to set a component's background color. You'll be able to use that same technique to set the background color of the Profile Card.

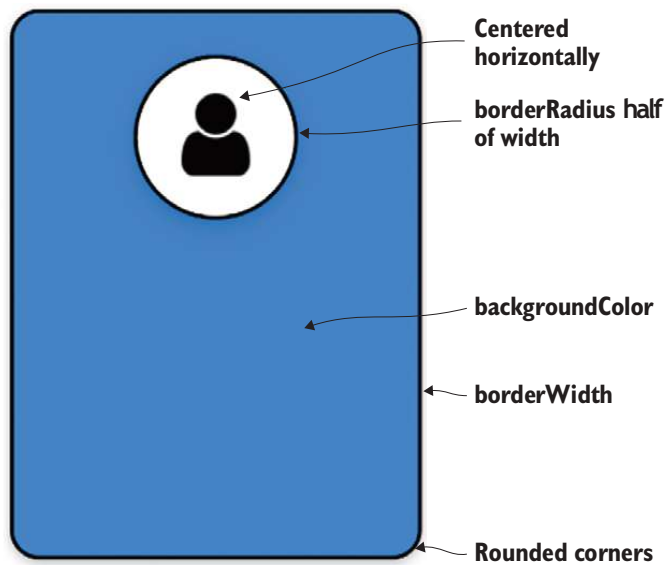


Figure 4.3 The Profile Card component after the structural View components have been styled. The Profile Card is a rectangle with rounded corners and a circular section for a profile image.

4.2.1 *Setting the background color*

Without a splash of color, a user interface (UI) looks boring and dull. You don't need an explosion of color to make things look interesting, but you do need a bit. The `backgroundColor` property sets the background color of an element. This property takes a string of one of the properties shown in table 4.1. The same colors are available when rendering text to the screen as well.

Table 4.1 Supported color formats

Supported color format	Example
<code>#rgb</code>	<code>'#06f'</code>
<code>#rgba</code>	<code>'#06fc'</code>
<code>#rrggbb</code>	<code>'#0066ff'</code>
<code>#rrggbbaa</code>	<code>'#ff00ff00'</code>
<code>rgb(number, number, number)</code>	<code>'rgb(0, 102, 255)'</code>
<code>rgb(number, number, number, alpha)</code>	<code>'rgba(0, 102, 255, .5)'</code>
<code>hsl(hue, saturation, lightness)</code>	<code>'hsl(216, 100%, 50%)'</code>
<code>hsla(hue, saturation, lightness, alpha)</code>	<code>'hsla(216, 100%, 50%, .5)'</code>
Transparent background	<code>'transparent'</code>
Any CSS3-specified named color (black, red, blue, and so on)	<code>'dodgerblue'</code>

Fortunately, the supported color formats are the same ones supported by CSS. We won't go into great detail, but because this may be the first time you've seen some of these formats, here's a quick explanation:

- `rgb` stands for red, green, and blue. You can specify the values for red, green, and blue using a scale from 0–255 (or in hexadecimal 00–ff). Higher numbers mean more of each color.
- `alpha` is similar to opacity (0 is transparent, 1 is solid).
- `hue` represents 1 degree on a 360-degree color wheel, where 0 is red, 120 is green, and 240 is blue.
- `saturation` is the intensity of the color from a 0% shade of gray to 100% full color.
- `lightness` is a percentage between 0% and 100%. 0% is darker (closer to black), and 100% is light (closer to white).

You've seen `backgroundColor` applied in previous examples, so let's take things a step further in the next example. To use your new skills to create something real, let's start building the Profile Card. Right now, it won't look like much, as you can see in figure 4.4—it's just a 300×400 colored rectangle.

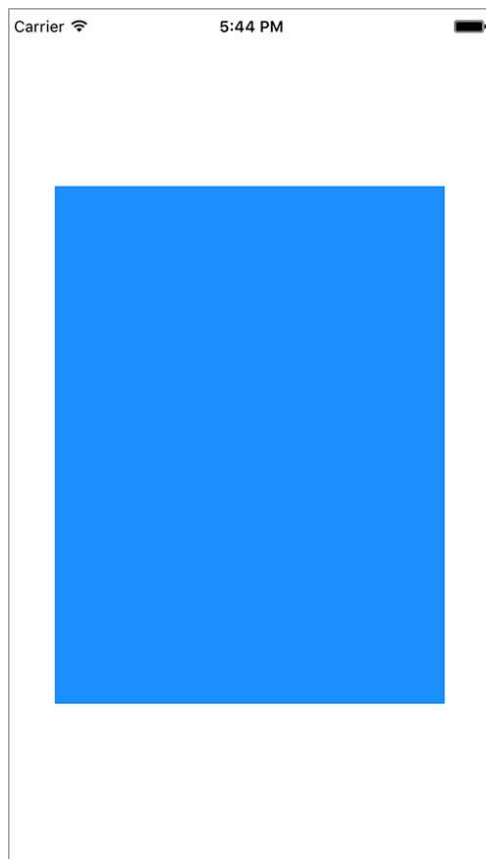


Figure 4.4 A simple 300×400 colored rectangle that forms the base of the Profile Card component

The following listing shows the initial code. Don't worry about the fact that most of it has nothing to do with styling. We'll walk through each piece, but you need a foundation from which to start.

Listing 4.7 Initial framework for the Profile Card component

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

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

const profileCardColor = 'dodgerblue';

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  cardContainer: {
    backgroundColor: profileCardColor,
    width: 300,
    height: 400
  }
});
```

The outermost View element references the container style that centers the child View component.

The inner View element will become the Profile Card component.

Defines the color for the Profile Card in a variable in case you need to use it in more than one place

Style definition for the outermost container

Style definition for the Profile Card container

Sets the Profile Card backgroundColor to the constant set earlier

The first View component is the outermost element. It acts as a container around everything else. Its sole purpose is to center child components on the device's display. The second View component will be the container for the Profile Card. For now, it's a 300 × 400 colored rectangle.

4.2.2 Setting border properties

Applying a background color to a component definitely makes it stand out, but without a crisp border line delineating the edge of the component, it looks like the component is floating in space. A clear delineation between components will help users understand how to interact with your mobile application.

Adding a border around a component is the best way to give screen elements a concrete, real feeling. There are quite a few border properties, but conceptually there are only four: `borderColor`, `borderRadius`, `borderStyle`, and `borderWidth`. These properties apply to the component as a whole.

For the color and width, there are individual properties for each side of the border: `borderTopColor`, `borderRightColor`, `borderBottomColor`, `borderLeftColor`, `borderTopWidth`, `borderRightWidth`, `borderBottomWidth`, and `borderLeftWidth`. For the border radius, there are properties for each corner: `borderTopRightRadius`, `borderBottomRightRadius`, `borderBottomLeftRadius`, and `borderTopLeftRadius`. But there's only one `borderStyle`.

CREATING BORDERS WITH THE COLOR, WIDTH AND STYLE PROPERTIES

To set a border, you must first set `borderWidth`. `borderWidth` is the size of the border, and it's always a number. You can either set a `borderWidth` that applies to the entire component or choose which `borderWidth` you want to set specifically (top, right, bottom, or left). You can combine these properties in many different ways to get the effect you like. See figure 4.5 for some examples.

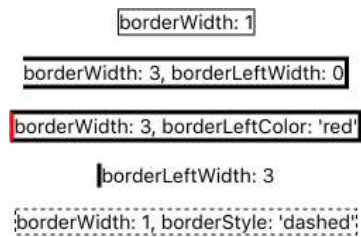


Figure 4.5 Examples of various combinations of border style settings

As you can see, you can combine border styles to create combinations of border effects. The next listing shows how easy this is to do.

Listing 4.8 Setting various border combinations

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

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <Example style={{borderWidth: 1}}>
          <Text>borderWidth: 1</Text>
        </Example>
        <Example style={{borderWidth: 3, borderLeftWidth: 0}}>
          <Text>borderWidth: 3, borderLeftWidth: 0</Text>
        </Example>
        <Example style={{borderWidth: 3, borderLeftColor: 'red'}}>
          <Text>borderWidth: 3, borderLeftColor: 'red'</Text>
        </Example>
        <Example style={{borderLeftWidth: 3}}>
          <Text>borderLeftWidth: 3</Text>
        </Example>
        <Example style={{borderWidth: 1, borderStyle: 'dashed'}}>
          <Text>borderWidth: 1, borderStyle: 'dashed'</Text>
        </Example>
      </View>
    );
  }
}

const Example = (props) => (
  <View style={ [styles.example, props.style] }>
    {props.children}
  </View>
);
```

Sets borderWidth to 1

Increases borderWidth to 3, removes the left border, and sets borderLeftWidth to 0

Sets borderWidth to 3, adds back the left border, and sets the color to red

Sets only a left border, with borderLeftWidth set to 3

Changes borderStyle from the default solid to dashed

Reusable Example component with a default set of styles that can easily be overridden by passing in style properties

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  example: {
    marginBottom: 15
  }
});
```

When only `borderWidth` is specified, `borderColor` defaults to 'black' and `borderStyle` defaults to 'solid'. If `borderWidth` or `borderColor` is set at the component level, those properties can be overridden by using a more specific property like `borderWidthLeft`; specificity takes precedence over generality.

NOTE `borderStyle` is a bit buggy, and I suggest sticking with the default, solid border. If you try to change the border width of any side and have `borderStyle` set to 'dotted' or 'dashed', you'll get an error. This will probably be fixed at some point, but for now don't spend too much time scratching your head if `borderStyle` doesn't work the way you expect. File that away in your brain, and let's move along.

USING BORDER RADIUS TO CREATE SHAPES

Another border property that can be used to great effect is `borderRadius`. A lot of objects in the real world have straight edges, but seldom does a straight line convey any sense of style. You wouldn't buy an automobile that looked like a box. You want your car to have nice curved lines that look sleek. Using the `borderRadius` style gives you the ability to add a bit of style to your applications. You can make many different, interesting shapes by adding curves in the right spots.

With `borderRadius`, you can define how rounded border corners appear on elements. As you may suspect, `borderRadius` applies to the entire component. If you set `borderRadius` and don't set one of the more specific values, like `borderTopLeftRadius`, all four corners will be rounded. Look at figure 4.6 to see how to round different borders to create cool effects.

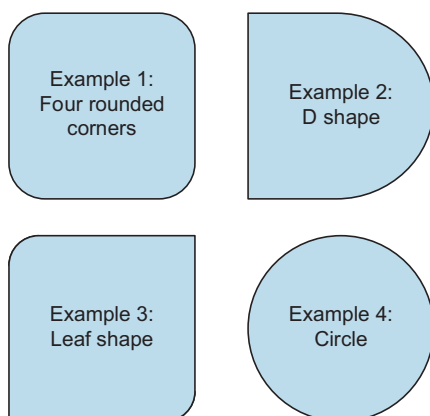


Figure 4.6 Examples of various border radius combinations. Example 1: a square with four rounded corners. Example 2: a square with the right two corners rounded, making a D shape. Example 3: a square with the opposite corners rounded, which looks like a leaf. Example 4: a square with a border radius equal to half the length of a side, which results in a circle.

Creating the shapes in figure 4.6 is relatively simple, as shown in listing 4.9. Honestly, the trickiest part about this code is making sure you don't make the text too big or too long. I'll show you what I mean shortly, in listing 4.10.

Listing 4.9 Setting various border radius combinations

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

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <Example style={{borderRadius: 20}}>
          <CenteredText>
            Example 1:{"\n"}4 Rounded Corners
          </CenteredText>
        </Example>
        <Example style={{borderTopRightRadius: 60,
          borderBottomRightRadius: 60}}>
          <CenteredText>
            Example 2:{"\n"}D Shape
          </CenteredText>
        </Example>
        <Example style={{borderTopLeftRadius: 30,
          borderBottomRightRadius: 30}}>
          <CenteredText>
            Example 3:{"\n"}Leaf Shape
          </CenteredText>
        </Example>
        <Example style={{borderRadius: 60}}>
          <CenteredText>
            Example 4:{"\n"}Circle
          </CenteredText>
        </Example>
      </View>
    );
  }
}

const Example = (props) => (
  <View style={ [styles.example, props.style] }>
    {props.children}
  </View>
);

const CenteredText = (props) => (
  <Text style={ [styles.centeredText, props.style] }>
    {props.children}
  </Text>
);

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    flexWrap: 'wrap',
```

Example 1: a square with four rounded corners

This is JavaScript, so you can specify a hard return inline with the text by using {"\n"}.

Example 2: a square with the right two corners rounded

Example 3: a square with the opposite corners rounded

Example 4: a square with a border radius equal to half the length of a side

Reusable component for rendering the centered text elements

React Native uses flexbox to control layout.

```

        marginTop: 75
    },
    example: {
        width: 120,
        height: 120,
        marginLeft: 20,
        marginBottom: 20,
        backgroundColor: 'grey',
        borderWidth: 2,
        justifyContent: 'center'
    },
    centeredText: {
        textAlign: 'center',
        margin: 10
    }
  }
});

```

Style that centers the text within the text components

Pay particular attention to the style that centers the text. You got lucky by using `margin: 10`. If you used `padding: 10`, the background of the text component would occlude the underlying border stroke of the View component (see figure 4.7).

By default, a Text component inherits the background color of its parent component. Because the bounding box of the Text component is a rectangle, the background overlaps the nice rounded corners. Obviously, using the `margin` property solves the problem, but it's also possible to remedy the situation another way. You could add `backgroundColor: 'transparent'` to the `centeredText` style. Making the text component's background transparent allows the underlying border to show through and look normal again, as in figure 4.6.

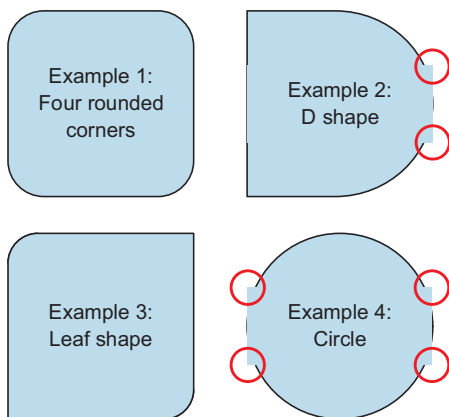


Figure 4.7 This is what figure 4.6 would look like if the `centeredText` style used `padding: 10` instead of `margin: 10` to position the text. The small circles highlight the points at which the bounding box of the Text component overlaps the border of the View component.

ADDING BORDERS TO YOUR PROFILE CARD COMPONENT

With your newfound knowledge of border properties, you can almost complete the initial layout of the Profile Card component. Using only the border properties from the last section, you can transform the 300×400 colored rectangle into something that more closely resembles what you want. Figure 4.8 shows how far you can get with an image and the techniques you've learned so far. It includes an image to use as a placeholder for a person's photo; you'll find it in the source code. But the circle is created by manipulating the border radius as described in the previous examples.

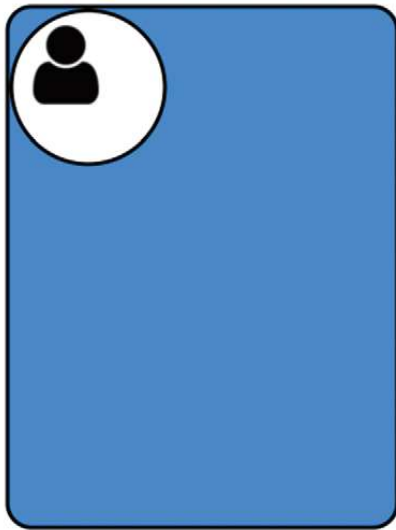


Figure 4.8 Incorporating border properties into the Profile Card component transforms the 300 × 400 colored rectangle into something more akin to what you want for the final Profile Card component.

Clearly there are some layout issues with the Profile Card, but you're almost there. We'll discuss how to use the margin and padding styles in the next section to get everything aligned correctly.

Listing 4.10 Incorporating border properties into the Profile Card

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

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.cardContainer}>
          <View style={styles.cardImageContainer}>
            <Image style={styles.cardImage}
              source={require('./user.png')} />
          </View>
        </View>
      </View>
    );
  }
}

const profileCardColor = 'dodgerblue';

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  cardContainer: {
    borderColor: 'black',
    borderWidth: 3,
    borderStyle: 'solid',
```

Imports the Image component from react-native

user.png is located in the same directory as the application code.

Adds the border properties to the Profile Card

```

        borderRadius: 20,
        backgroundColor: profileCardColor,
        width: 300,
        height: 400
    },
    cardImageContainer: {
        backgroundColor: 'white',
        borderWidth: 3,
        borderColor: 'black',
        width: 120,
        height: 120,
        borderRadius: 60,
    },
    cardImage: {
        width: 80,
        height: 80
    }
});

```

The image container is a 120 × 120 square with a borderRadius of 60 (half of 120), which results in a circle.

Styles for the actual image

The differences between listing 4.10 and the previous Profile Card code (listing 4.7) have been bolded to highlight the incremental changes.

4.2.3 Specifying margins and padding

You could explicitly position every component on the screen and lay it out exactly like you want, but that would be extremely tedious if the layout needed to be responsive to user actions. It makes more sense to position items relative to one another, so if you move one component, the other components can move in response based on their relative positions.

The margin style allows you to define this relationship between components. The padding style lets you define the relative position of a component to its border. Using these properties together provides a great deal of flexibility when laying out components. You'll use these properties every day, so it's important to understand what they mean and do.

Conceptually, margins and padding work exactly the same as they do in CSS. The customary depiction of how margins and padding relate to borders and the content area still applies (see figure 4.9).

Functionally, you're likely to run into bugs when dealing with margins and padding. You might be tempted to call them "quirks," but either way they're a pain. For the most part, margins on View components behave reasonably well and work across iOS and Android. Padding tends to work a little differently between OSs. At the time of writing, padding text components in an Android environment doesn't work at all; I suspect that will change in an upcoming release.

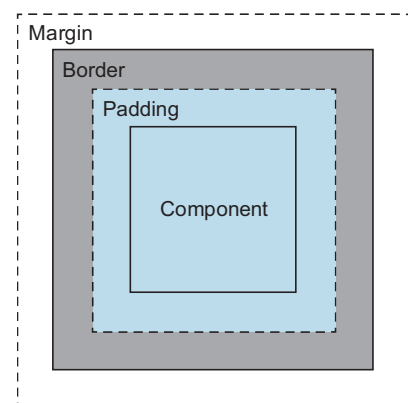


Figure 4.9 A common depiction of how margins, padding, and borders interrelate

USING THE MARGIN PROPERTY

When laying out components, one of the first problems to solve is how far the components are from one another. To avoid specifying a distance for each component, you need a way to specify a relative position. The margin property allows you to define the perimeter of the component, which determines how far an element is from the previous or parent component. Articulating the layout this way allows the container to figure out where the components should be positioned with respect to one another rather than you having to calculate the position of every single component.

The available margin properties are `margin`, `marginTop`, `marginRight`, `marginBottom`, and `marginLeft`. If only the general margin property is set, without another, more-specific value such as `marginLeft` or `marginTop`, then that value applies to all sides of the component (top, right, bottom, and left). If both margin and a more-specific margin property are specified (for example, `marginLeft`), then the more-specific margin property takes precedence. It works exactly the same as the border properties. Let's apply some of these styles: see figure 4.10.

The margins all position the components as expected, but notice how the Android device clips the component when negative margins are applied. If you plan to support both iOS and Android, test on each device from the beginning of your project. Don't develop on iOS and think everything you styled will behave the same on Android. Listing 4.11 shows the code for the examples in figure 4.10.

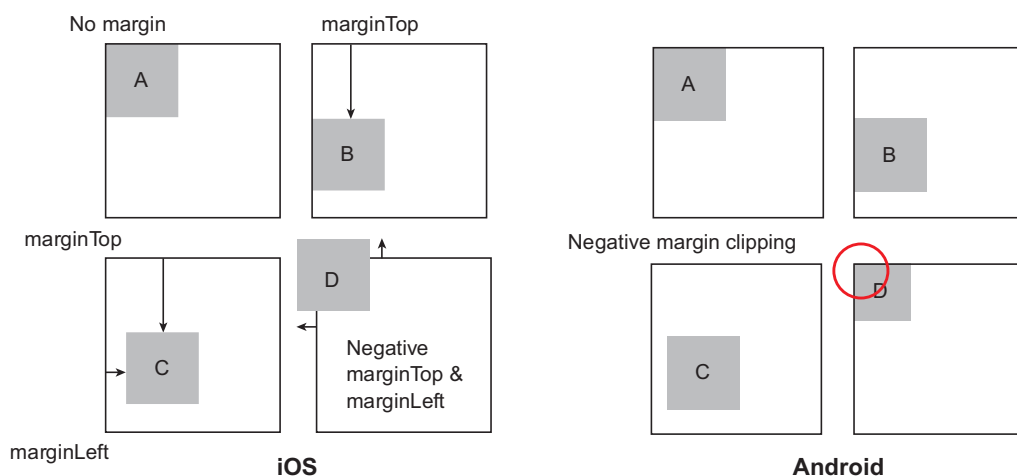


Figure 4.10 Examples of applying margins to components. In iOS, example A has no margins applied. Example B has a top margin applied. Example C has top and left margins. Example D has both negative top and negative left margins. In Android, negative margins behave a bit differently: the component is clipped by the parent container.

Listing 4.11 Applying various margins to components

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

export default class App extends Component<{}> {
```

```

render() {
  return (
    <View style={styles.container}>
      <View style={styles.exampleContainer}>
        <Example>
          <CenteredText>A</CenteredText>
        </Example>
      </View>
      <View style={styles.exampleContainer}>
        <Example style={{marginTop: 50}}>
          <CenteredText>B</CenteredText>
        </Example>
      </View>
      <View style={styles.exampleContainer}>
        <Example style={{marginTop: 50, marginLeft: 10}}>
          <CenteredText>C</CenteredText>
        </Example>
      </View>
      <View style={styles.exampleContainer}>
        <Example style={{marginLeft: -10, marginTop: -10}}>
          <CenteredText>D</CenteredText>
        </Example>
      </View>
    </View>
  );
}

const Example = (props) => (
  <View style={[styles.example, props.style]}>
    {props.children}
  </View>
);

const CenteredText = (props) => (
  <Text style={[styles.centeredText, props.style]}>
    {props.children}
  </Text>
);

const styles = StyleSheet.create({
  container: {
    alignItems: 'center',
    flex: 1,
    flexDirection: 'row',
    flexWrap: 'wrap',
    justifyContent: 'center',
    marginTop: 75
  },
  exampleContainer: {
    borderWidth: 1,
    width: 120,
    height: 120,
    marginLeft: 20,
    marginBottom: 20,
  }
});

```

Base example with no margins applied

marginTop of 50

marginTop of 50 and marginLeft of 10

Applies negative margins to marginTop and marginLeft

```

    },
    example: {
      width: 50,
      height: 50,
      backgroundColor: 'grey',
      borderWidth: 1,
      justifyContent: 'center'
    },
    centeredText: {
      textAlign: 'center',
      margin: 10
    }
  }
});

```

USING THE PADDING PROPERTY

You can think of margins as the distance between elements, but padding represents the space between the content of the element and the border of the same element. When padding is specified, it allows the content of the component to not be flush against the border. In figure 4.9, the `backgroundColor` property bleeds through the component's edges up to the border, which is the space defined by padding. The available properties available for padding are `padding`, `paddingLeft`, `paddingRight`, `paddingTop`, and `paddingBottom`. If only the main padding property is set without another, more-specific value such as `paddingLeft` or `paddingTop`, then that value is passed to all sides of the component (top, right, bottom, and left). If both padding and a more-specific padding property are specified, such as `paddingLeft`, then the more-specific padding property takes precedence. This behavior is exactly like borders and margins.

Rather than create a new example to show how padding is different than margins, let's reuse the code from listing 4.11 and make a few tweaks. Change the margin styles on the example components to padding styles, and add a border around the `Text` components and change their background color. Figure 4.11 shows what you'll end up with.

Listing 4.12 Modifying listing 4.11 to replace margins with padding

```
import React, { Component } from 'react';
```

```
...
```

```

<View style={styles.container}>
  <View style={styles.exampleContainer}>
    <Example style={{}}> ← Example A: unchanged, with
      <CenteredText>A</CenteredText>      no margins or padding
    </Example>
  </View>
  <View style={styles.exampleContainer}>
    <Example style={{paddingTop: 50}}> ← Example B: marginTop
      <CenteredText>B</CenteredText>      changed to paddingTop
    </Example>
  </View>
  <View style={styles.exampleContainer}>
    <Example style={{paddingTop: 50, paddingLeft: 10}}> ← Example C: marginTop and marginLeft
      <CenteredText>C</CenteredText>      changed to paddingTop and paddingLeft, respectively
    </Example>
  </View>
</View>

```

```

    </Example>
  </View>
  <View style={styles.exampleContainer}>
    <Example style={{paddingLeft: -10, paddingTop: -10}}>
      <CenteredText>D</CenteredText>
    </Example>
  </View>
</View>
...
},
centeredText: {
  textAlign: 'center',
  margin: 10,
  borderWidth: 1,
  backgroundColor: 'lightgrey'
}
});

```

Example D: `marginLeft` and `marginTop` changed to `paddingLeft` and `marginTop`, respectively. The negative values remain.

Adds a border and background color to the Text component

Unlike margins, which specify the space between the component and its parent component, padding applies from the border of the component to its children. In example B, padding is calculated from the top border, which *pushes* the Text component B down from the top border. Example C adds a `paddingLeft` value, which also *pushes* the Text component C inward from the left border. Example D applies negative padding values to `paddingTop` and `paddingLeft`.

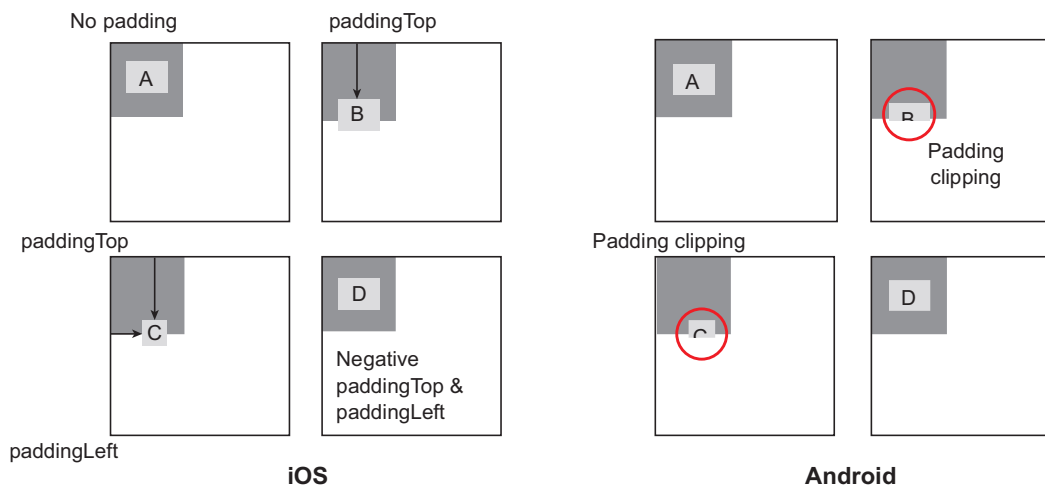


Figure 4.11 Changing the margin styles from the previous example to padding styles. Example A, with no padding, looks the same as when no margins are applied. Example B shows the component with `paddingTop` applied. Example C is the same, but it also applies `paddingLeft`. Example D applies negative padding values to `paddingTop` and `paddingLeft`, which are ignored.

A few interesting observations can be made. Example B and example C are both clipped on the Android device. Example C's Text component's width is compressed, and the negative values for padding are ignored in example D.

4.2.4 Using position to place components

So far, everything we've looked at has been positioned relative to another component, which is the default layout position. Sometimes it's beneficial to take advantage of absolute positioning and place a component exactly where you want it. The implementation of the position style in React Native is similar to CSS, but there aren't as many options. By default, all elements are laid out relative to one another. If position is set to absolute, then the element is laid out relative to its parent. The available properties for position are relative (the default position) and absolute.

CSS has other values, but those are the only two in React Native. When using absolute positioning, the following properties are also available: top, right, bottom, and left.

Let's look at a simple example to demonstrate the difference between relative and absolute positioning. In CSS, positioning can get much more confusing, but in React Native the "everything has relative positioning by default" makes it much easier to position items. In figure 4.12, blocks A, B, and C are laid out relative to one another in a row. Without any margin or padding, they're lined up one after another. Block D is a sibling to the ABC row of blocks, meaning the main container is the parent container for the ABC row and block D.

Block D is set to {position: 'absolute', right: 0, bottom: 0}, so it's positioned in the lower-right corner of its container. Block E is also set to {position: 'absolute', right: 0, bottom: 0}, but its parent container is block B, which means block E is positioned absolutely but with respect to block B. Block E appears in the lower-right corner of block B, instead. Listing 4.13 shows the code for this example.

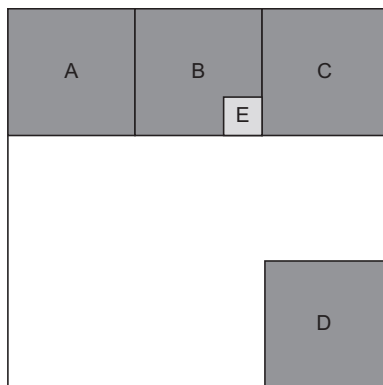


Figure 4.12 An example showing blocks A, B, and C laid out relative to one another. Block D has an absolute position of right: 0 and bottom: 0. Block E also has an absolute position of right: 0 and bottom: 0, but its parent is block B and not the main container, whereas D's parent was the main container.

Listing 4.13 Relative and absolute positioning comparison

```

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

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.row}>
          <Example>
            <CenteredText>A</CenteredText>
          </Example>
          <Example>
            <CenteredText>B</CenteredText>
            <View style={[styles.tinyExample,
              {position: 'absolute',
                right: 0,
                bottom: 0}]}>
              <CenteredText>E</CenteredText>
            </View>
          </Example>
          <Example>
            <CenteredText>C</CenteredText>
          </Example>
        </View>
        <Example style={{position: 'absolute',
          right: 0, bottom: 0}}>
          <CenteredText>D</CenteredText>
        </Example>
      </View>
    );
  }
}

const Example = (props) => (
  <View style={[styles.example, props.style]}>
    {props.children}
  </View>
);

const CenteredText = (props) => (
  <Text style={[styles.centeredText, props.style]}>
    {props.children}
  </Text>
);

const styles = StyleSheet.create({
  container: {
    width: 300,
    height: 300,
    margin: 40,
    marginTop: 100,
    borderWidth: 1
  },
  row: {

```

Row containing blocks A, B, and C

Block E is absolutely positioned in the lower-right corner of its parent container, block B.

Block D is absolutely positioned in the lower-right corner of its parent container.

The flexbox direction is specified as row, so the blocks are in a row across the screen.

```

        flex: 1,
        flexDirection: 'row'
      },
      example: {
        width: 100,
        height: 100,
        backgroundColor: 'grey',
        borderWidth: 1,
        justifyContent: 'center'
      },
      tinyExample: {
        width: 30,
        height: 30,
        borderWidth: 1,
        justifyContent: 'center',
        backgroundColor: 'lightgrey'
      },
      centeredText: {
        textAlign: 'center',
        margin: 10
      }
    }
  });

```

NOTE In listing 4.13, the `flexDirection` property is specified as `'row'`, so the blocks are in a row across the screen. React Native uses an open source, cross-platform layout library called Yoga (<https://yogalayout.com>). Yoga implements the flexbox layout mode, which you often see in CSS and is used frequently in React Native. We'll spend a lot of time in the next chapter talking about flexbox. Margins, padding, and position are all great layout tools, but flexbox is the tool you'll use most often.

We're finished with the fundamentals of styling View components. You've learned about some layout techniques: margins, padding, and position. Let's revisit the Profile Card component and fix the pieces that aren't yet laid out properly.

4.2.5 Profile Card positioning

The following listing has the code changes that need to be made to listing 4.10 to space the circle and user image properly and center everything. Figure 4.13 shows the result.

Listing 4.14 Modifying Profile Card styles to fix the layout

```

...
cardContainer: {
  alignItems: 'center',
  borderColor: 'black',
  borderWidth: 3,
  borderStyle: 'solid',
  borderRadius: 20,
  backgroundColor: profileCardColor,
  width: 300,
  height: 400
}

```

← Aligns the circle in the horizontal center of the Profile Card

```

    },
    cardImageContainer: {
      alignItems: 'center',
      backgroundColor: 'white',
      borderWidth: 3,
      borderColor: 'black',
      width: 120,
      height: 120,
      borderRadius: 60,
      marginTop: 30,
      paddingTop: 15
    },
  },
  ...

```

Provides padding between the inner part of the circle and the contained image

Aligns the user image in the horizontal center of the circle

Provides space between the top of the circle and the top of the Profile Card

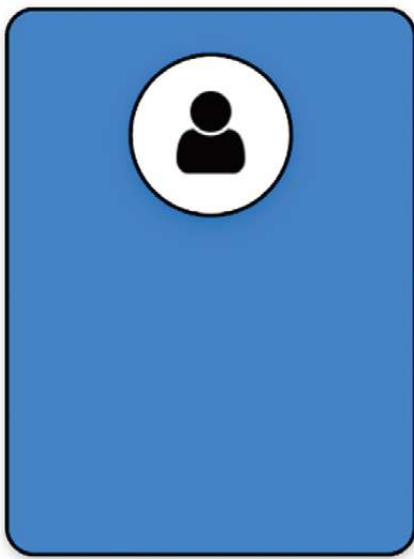


Figure 4.13 The Profile Card component after all the View components have been lined up properly

Now, the major View components for the Profile Card are in place. By using the techniques discussed so far, you've built a nice-looking foundation for the component, but you're not finished. You need to add information about the person: name, occupation, and a brief profile description. All that information is text based, so the next thing you'll learn is how to style Text components.

4.3 Styling Text components

In this section, we'll discuss how to style Text components. After you have a working knowledge of how to make text look great, we'll take another look at the Profile Card and add some information about the user. Figure 4.14 is the finished Profile Card component with the user's name and occupation and a brief profile description. But before we revisit the Profile Card, let's look at the styling techniques that will enable you to finish building it.

4.3.1 Text components vs. View components

With the exception of flex properties, which we have yet to cover, most of the styles applicable to View elements will also work as expected with Text elements. Text

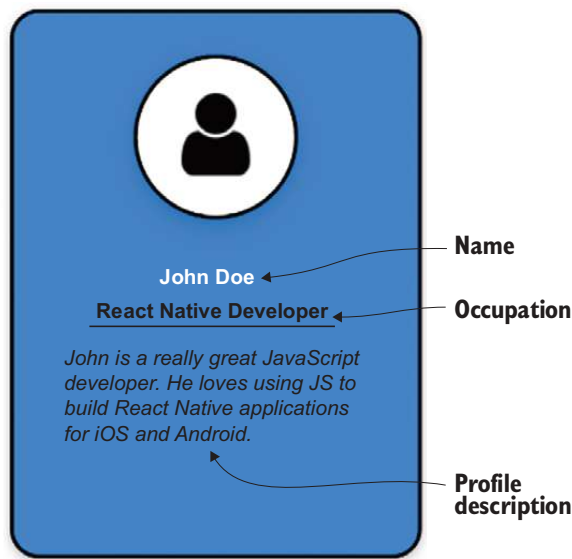


Figure 4.14 The completed Profile Card with the user's name and occupation and a brief profile description

elements can have borders and backgrounds and are affected by layout properties like margin, padding, and position.

The reverse can't be said. Most of the styles Text elements can use won't work for View elements, which makes perfect sense. If you've ever used a word processor, you know you can use different fonts for text and change the font color; that you can resize, bold, and italicize the text; and that you can apply decorations like underlines.

Before we get into text-specific styling, let's talk about color, a style common to both Text and View components. Then you'll use color along with everything you've learned thus far to start adding text to the Profile Card.

COLORING TEXT

The color property applies to Text components in exactly the same way as it does to View components. As expected, this property specifies the color of the text in a Text element. All the color formats listed in table 4.1 still apply—even transparent, although I can't imagine how that's of benefit. By default, the text color is black.

Figure 4.14 showed three Text elements in the Profile Card:

- Name
- Occupation
- Profile description

Using what you've already learned, you can center and position the text, change the color of the name from black to white, and add a simple border to separate the occupation from the description. Figure 4.15 shows what you'll end up with by applying techniques in your arsenal.

By this point, you should be able to follow along with listing 4.15 and understand everything that's going on. Don't feel bad if you don't—if necessary, go back and re-read the appropriate sections.

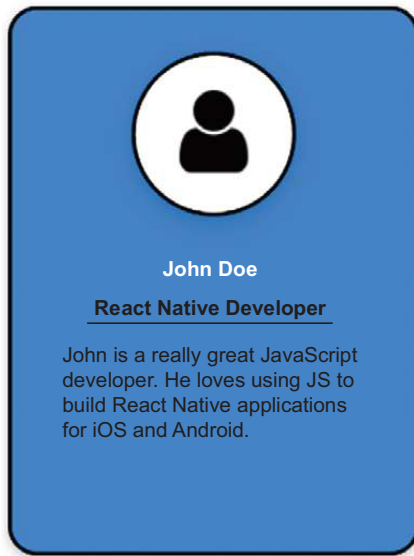


Figure 4.15 The Profile Card with `Text` elements added using text styling defaults and the `color` property for the name set to white

Listing 4.15 Adding text to the Profile Card

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

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.cardContainer}>
          <View style={styles.cardImageContainer}>
            <Image style={styles.cardImage}
              source={require('./user.png')} />
          </View>
          <View>
            <Text style={styles.cardName}>
              John Doe
            </Text>
          </View>
          <View style={styles.cardOccupationContainer}>
            <Text style={styles.cardOccupation}>
              React Native Developer
            </Text>
          </View>
          <View>
            <Text style={styles.cardDescription}>
              John is a really great JavaScript developer. He
              loves using JS to build React Native applications
              for iOS and Android.
            </Text>
          </View>
        </View>
      </View>
    );
  }
}
```

Imports the `Text` component from `react-native`

Text component that renders the person's name

Container around the occupation text that sets a bottom border separating the occupation from the description

Text component that renders the occupation

Text component that renders the profile description

```

    }
  }

  const profileCardColor = 'dodgerblue';

  const styles = StyleSheet.create({
    container: {
      flex: 1,
      justifyContent: 'center',
      alignItems: 'center'
    },
    cardContainer: {
      alignItems: 'center',
      borderColor: 'black',
      borderWidth: 3,
      borderStyle: 'solid',
      borderRadius: 20,
      backgroundColor: profileCardColor,
      width: 300,
      height: 400
    },
    cardImageContainer: {
      alignItems: 'center',
      backgroundColor: 'white',
      borderWidth: 3,
      borderColor: 'black',
      width: 120,
      height: 120,
      borderRadius: 60,
      marginTop: 30,
      paddingTop: 15
    },
    cardImage: {
      width: 80,
      height: 80
    },
    cardName: {
      color: 'white',
      marginTop: 30,
    },
    cardOccupationContainer: {
      borderColor: 'black',
      borderBottomWidth: 3
    },
    cardOccupation: {
      marginTop: 10,
      marginBottom: 10,
    },
    cardDescription: {
      marginTop: 10,
      marginRight: 40,
      marginLeft: 40,
      marginBottom: 10
    }
  });

```

Styles for the name Text component;
the color is 'white'.

Styles for the occupation container

Styles for the occupation text (currently
only positional styling)

Styles for the profile description

At this point, you have all the content for the Profile Card, but it's pretty plain. In the next couple of sections, we'll talk about how to set font properties and add decorative styles to text.

4.3.2 Font styles

If you've ever used a word processor or written an email with rich text capabilities, you've been able to change fonts, increase or decrease the font size, bold or italicize the text, and so on. These are the same styles you'll learn how to change in this section. By adjusting these styles, you can make text more compelling and attractive to the end user. We'll discuss these properties: `fontFamily`, `fontSize`, `fontStyle`, and `fontWeight`.

SPECIFYING A FONT FAMILY

The `fontFamily` property is deceptively simple. If you stick with the defaults, it's easy; but if you want to use a specific font, you can run into trouble quickly. Both iOS and Android come with a default set of fonts. For iOS, a large number of available fonts can be implemented out of the box. For Android, there's Roboto, a monospace font, and some simple serif and sans serif variants. For a full list of Android and iOS fonts available out of the box in React Native, go to <https://github.com/dabit3/react-native-fonts>.

If you wanted to use a monospaced font in an application, you couldn't specify either of the following:

- `fontFamily: 'monospace'`—The 'monospace' option isn't supported on iOS, so on that platform you'll get the error "Unrecognized font family 'monospace'." But on Android, the font will render correctly without any problems. Unlike CSS, you can't supply multiple fonts to the `fontFamily` property.
- `fontFamily: 'American Typewriter, monospace'`—You'll again get an error on iOS, "Unrecognized font family 'American Typewriter, monospace'." But on Android, when you supply a font it doesn't support, it falls back to the default. That might not be true in every version of Android, but suffice it to say neither approach will work.

If you want to use different fonts, you'll have to use React Native's `Platform` component. We'll discuss `Platform` in more detail in chapter 10, but I want to introduce it, so you can see how to work around this dilemma. Figure 4.16 shows the American Typewriter font rendered on iOS and the generic monospace font used on Android.

The following listing shows the code that produced this example. Pay attention to how the `fontFamily` is set using `Platform.select`.

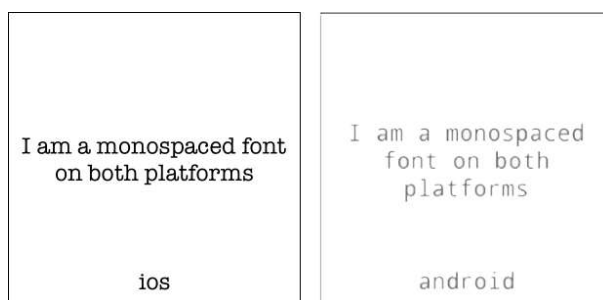


Figure 4.16 An example of rendering monospaced fonts on both iOS and Android

Listing 4.16 Displaying monospaced fonts on iOS and Android

```

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

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.row}>
          <CenteredText>
            I am a monospaced font on both platforms
          </CenteredText>
          <BottomText>
            {Platform.OS}
          </BottomText>
        </View>
      </View>
    );
  }
}

const CenteredText = (props) => (
  <Text style={[styles.centeredText, props.style]}>
    {props.children}
  </Text>
);

const BottomText = (props) => (
  <CenteredText style={[{position: 'absolute', bottom: 0},
    props.style]}>
    {props.children}
  </CenteredText>
);

const styles = StyleSheet.create({
  container: {
    width: 300,
    height: 300,
    margin: 40,
    marginTop: 100,
    borderWidth: 1
  },
  row: {
    alignItems: 'center',
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center'
  },
  centeredText: {
    textAlign: 'center',
    margin: 10,
    fontSize: 24,
    ...Platform.select({
      ios: {

```

Imports the Platform component from react-native

Platform.OS can also tell you what OS the code is running on.

Takes advantage of your absolute positioning knowledge

Uses Platform.select to pick the styles for the appropriate platform

```

        fontFamily: 'American Typewriter'
      },
      android: {
        fontFamily: 'monospace'
      }
    })
  }
});

```

This example shows how to select fonts based on the OS, but the set of fonts at your disposal is still limited to what comes with React Native out of the box. You can add custom fonts to a project using font files (TTF, OTF, and so on) and linking them to your application as assets. In theory the process is simple, but success varies greatly depending on the OS and the font files being used. I want you to know it's possible to do, but if you want to give it try, break out your search engine of choice and look into [react-native link](#).

ADJUSTING TEXT SIZE WITH FONT SIZE

`fontSize` is pretty simple: it adjusts the size of the text in a `Text` element. You've used this quite a bit already, so we won't go into much detail other than the fact that the default `fontSize` is 14.

CHANGING FONT STYLES

You can use `fontStyle` to change the font style to italic. The default is `'normal'`. The only two options at this moment are `'normal'` and `'italic'`.

SPECIFYING FONT WEIGHTS

`fontWeight` refers to the thickness of the font. The default is `'normal'` or `'400'`. The options for `fontWeight` are `'normal'`, `'bold'`, `'100'`, `'200'`, `'300'`, `'400'`, `'500'`, `'600'`, `'700'`, `'800'`, and `'900'`. The smaller the value, the lighter/thinner the text. The larger the value, the thicker/bolder the text.

Now that you know how to change the font styles, you can almost finish the Profile Card component. Let's change some font styles and see how close you can get to the final product, as shown in figure 4.17. The next listing shows how to change the styles from listing 4.16 to achieve this look.

Listing 4.17 Setting font styles for Text elements in the Profile Card

```

...
cardName: {
  color: 'white',
  fontWeight: 'bold',
  fontSize: 24,
  marginTop: 30,
},
...
cardOccupation: {
  fontWeight: 'bold',
  marginTop: 10,
  marginBottom: 10,
},
cardDescription: {

```

Changes the font weight of the name text to bold

Changes the font size of the name text to 24

Bolds the occupation text

```

    fontStyle: 'italic',      ← Italicizes the description text
    marginTop: 10,
    marginRight: 40,
    marginLeft: 40,
    marginBottom: 10
  }
  ...

```



Figure 4.17 The Profile Card with font styles applied to the Name, Occupation, and Description texts

Modifying the font styles for the name, occupation, and description text helps differentiate each of the sections, but the name still doesn't stand out much. The next section covers some decorative ways to style text and how to use those techniques to make the name stand out in the Profile Card.

4.3.3 Using decorative text styles

In this section, you'll go beyond the basics of changing font styles and start applying decorative styles to text. I'll show you how to do things like underline and strike-through text and add drop shadows. These techniques can add a lot of visual variety to applications and help text elements stand out from one another.

Here are the properties we'll cover in this section:

- *iOS and Android*—`lineHeight`, `textAlign`, `textDecorationLine`, `textShadowColor`, `textShadowOffset`, and `textShadowRadius`
- *Android only*—`textAlignVertical`
- *iOS only*—`letterSpacing`, `textDecorationColor`, `textDecorationStyle`, and `writingDirection`.

Notice that some of the properties only apply to one OS or another. Some values that can be assigned to the properties are also OS-specific. It's important to keep this in mind, especially if you're relying on a specific style to highlight a particular element of text on the screen.

SPECIFYING HEIGHT OF TEXT ELEMENTS

`lineHeight` specifies the height of the Text element. Figure 4.18 and listing 4.18 show an example of how this behaves differently on iOS versus Android. A `lineHeight` of 100 is applied to the Text B element: the height of that line is significantly greater than the others. Also notice how iOS and Android position the text within the line differently. On Android, the text is positioned at the bottom of the line.

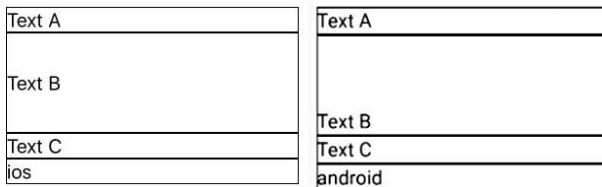


Figure 4.18 Example of using `lineHeight` on iOS and Android.

Listing 4.18 Applying `lineHeight` to a Text element in iOS and Android

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

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <TextContainer>
          <LeftText>Text A</LeftText>
        </TextContainer>
        <TextContainer>
          <LeftText style={{lineHeight: 100}}> ← Sets lineHeight to 100
            Text B
          </LeftText>
        </TextContainer>
        <TextContainer>
          <LeftText>Text C</LeftText>
        </TextContainer>
        <TextContainer>
          <LeftText>{Platform.OS}</LeftText>
        </TextContainer>
      </View>
    );
  }
}

const LeftText = (props) => (
  <Text style={[styles.leftText, props.style]}>
    {props.children}
  </Text>
);

const TextContainer = (props) => (
  <View style={[styles.textContainer, props.style]}>
    {props.children}
  </View>
);
```

```

    </View>
  );

  const styles = StyleSheet.create({
    container: {
      width: 300,
      height: 300,
      margin: 40,
      marginTop: 100
    },
    textContainer: {
      borderWidth: 1
    },
    leftText: {
      fontSize: 20
    }
  });

```

Setting a border lets you easily see the height of the line.

ALIGNING TEXT HORIZONTALLY

`textAlign` refers to how the text in the element will be horizontally aligned. The options for `textAlign` are 'auto', 'center', 'right', 'left', and 'justify' ('justify' is iOS only).

UNDERLINING TEXT OR ADDING LINES THROUGH TEXT

Use the `textDecorationLine` property to add either an underline or a line through the given text. The options for `textDecorationLine` are 'none', 'underline', 'line-through', and 'underline line-through'. The default value is 'none'. When you specify 'underline line-through', a single space separates the values in quotes.

TEXT-DECORATION STYLES (iOS ONLY)

iOS supports several text-decoration styles that Android doesn't. The first is `textDecorationColor`, which allows you to set a color for `textDecorationLine`. iOS also supports styling the line itself. On Android, the line is always solid, but on iOS `textDecorationStyle` lets you specify 'solid', 'double', 'dotted', and 'dashed'. Android will ignore these additional styles.

To use the additional iOS decoration styles, specify them in conjunction with the primary `textDecorationLine` style. For example:

```

{
  textDecorationLine: 'underline',
  textDecorationColor: 'red',
  textDecorationStyle: 'double'
}

```

ADDING SHADOWS TO TEXT

You can use the `textShadowColor`, `textShadowOffset`, and `textShadowRadius` properties to add a shadow to a Text element. To create a shadow, you need to specify three things:

- The color
- The offset
- The radius

The offset specifies the position of the shadow relative to the component casting the shadow. The radius basically defines how blurry the shadow appears. You can specify a text shadow like this:

```
{
  textShadowColor: 'red',
  textShadowOffset: {width: -2, height: -2},
  textShadowRadius: 4
}
```

CONTROLLING LETTER SPACING (iOS ONLY)

letterSpacing specifies the spacing between text characters. It's not something you'll use every day, but it can produce some interesting visual effects. Keep in mind that it's iOS only, so use it if you need it.

EXAMPLES OF TEXT STYLES

We've gone through a lot of different styles in this section. Figure 4.19 shows various styles applied to Text components.

Here's a quick rundown of the styles being used for each example in figure 4.19:

- A is italic text using {fontStyle: 'italic'}.
- B shows text decoration with an underline and a line through the text. The style for this is {textDecorationLine: 'underline line-through'}.
- C expands on example B by also applying some iOS-only text styles, {textDecorationColor: 'red', textDecorationStyle: 'dotted'}. Notice how these styles have no effect in Android.
- D applies a shadow using {textShadowColor: 'red', textShadowOffset: {width: -2, height: -2}, textShadowRadius: 4}.
- E uses the iOS-only {letterSpacing: 5}, which doesn't affect Android.
- The text *ios* and *android* is styled using {textAlign: 'center', fontWeight: 'bold'}.

Use listing 4.19 as a starting point, and see how modifying the styles affects the result.

A) <i>Italic</i>	A) <i>Italic</i>
B) <u>Underline and Line Through</u>	B) <u>Underline and Line Through</u>
C) <u>Underline and Line Through</u>	C) <u>Underline and Line Through</u>
D) Text Shadow	D) Text Shadow
E) Letter Spacing	E) Letter Spacing
ios	android

Figure 4.19 Various examples of styling text components

Listing 4.19 Examples of styling Text components

```

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

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <LeftText style={{fontStyle: 'italic'}}>
          A) Italic
        </LeftText>
        <LeftText style={{textDecorationLine: 'underline line-through'}}>
          B) Underline and Line Through
        </LeftText>
        <LeftText style={{textDecorationLine: 'underline line-through',
                          textDecorationColor: 'red',
                          textDecorationStyle: 'dotted'}}>
          C) Underline and Line Through
        </LeftText>
        <LeftText style={{textShadowColor: 'red',
                          textShadowOffset: {width: -2, height: -2},
                          textShadowRadius: 4}}>
          D) Text Shadow
        </LeftText>
        <LeftText style={{letterSpacing: 5}}>
          E) Letter Spacing
        </LeftText>
        <LeftText style={{textAlign: 'center', fontWeight: 'bold'}}>
          {Platform.OS}
        </LeftText>
      </View>
    );
  }
}

const LeftText = (props) => (
  <Text style={[styles.leftText, props.style]}>
    {props.children}
  </Text>
);

const styles = StyleSheet.create({
  container: {
    width: 300,
    height: 300,
    margin: 40,
    marginTop: 100
  },
  leftText: {
    fontSize: 20,
    paddingBottom: 10
  }
});

```

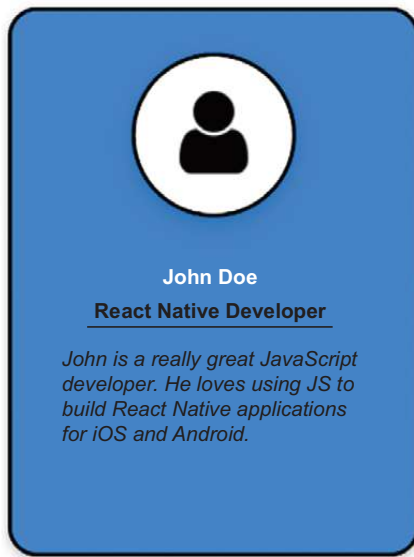


Figure 4.20 The completed Profile Card example. Textual information has been added about the person using the text styling techniques covered in this section.

Now that you know how to create a shadow effect, let's add a shadow to the person's name so it stands out from the other text. Figure 4.20 shows the desired result.

The completed code for the Profile Card is provided next. You only have to add a tiny snippet to set the text shadow for the name.

Listing 4.20 Completed Profile Card example

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

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.cardContainer}>
          <View style={styles.cardImageContainer}>
            <Image style={styles.cardImage}
              source={require('./user.png')} />
          </View>
          <View>
            <Text style={styles.cardName}>
              John Doe
            </Text>
          </View>
          <View style={styles.cardOccupationContainer}>
            <Text style={styles.cardOccupation}>
              React Native Developer
            </Text>
          </View>
          <View>
            <Text style={styles.cardDescription}>
              John is a really great JavaScript developer.
              He loves using JS to build React Native
              applications for iOS and Android.
            </Text>
          </View>
        </View>
      </View>
    );
  }
}
```



```

        </Text>
      </View>
    </View>
  </View>
);
}
}

const profileCardColor = 'dodgerblue';

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  cardContainer: {
    alignItems: 'center',
    borderColor: 'black',
    borderWidth: 3,
    borderStyle: 'solid',
    borderRadius: 20,
    backgroundColor: profileCardColor,
    width: 300,
    height: 400
  },
  cardImageContainer: {
    alignItems: 'center',
    backgroundColor: 'white',
    borderWidth: 3,
    borderColor: 'black',
    width: 120,
    height: 120,
    borderRadius: 60,
    marginTop: 30,
    paddingTop: 15
  },
  cardImage: {
    width: 80,
    height: 80
  },
  cardName: {
    color: 'white',
    fontWeight: 'bold',
    fontSize: 24,
    marginTop: 30,
    textShadowColor: 'black',
    textShadowOffset: {
      height: 2,
      width: 2
    },
    textShadowRadius: 3
  },
  cardOccupationContainer: {
    borderColor: 'black',
    borderBottomWidth: 3
  }
});

```

Sets the shadow color to black on the Title text component
 Sets the shadow offset to be down and to the right
 Sets the shadow radius

```

    },
    cardOccupation: {
      fontWeight: 'bold',
      marginTop: 10,
      marginBottom: 10,
    },
    cardDescription: {
      fontStyle: 'italic',
      marginTop: 10,
      marginRight: 40,
      marginLeft: 40,
      marginBottom: 10
    }
  }
});

```

There's a lot you could do to this basic example to make it even better, but the goal was to show how beneficial it is to understand styling concepts. You don't have to be a fantastic graphic designer to make a nice-looking component—a few simple techniques can make your application look great.

We covered a lot of ground in this chapter, but believe it or not, this has been a short introduction! We'll explore some additional advanced topics in chapter 5.

Summary

- Styles can be applied inline with components or by creating stylesheets that can be referenced by components.
- Styles should be organized in the same file as the component after the component definition or externalized into a separate `styles.js` file.
- Styles are code. The fact that JavaScript is a complete language with variables and functions affords many advantages over traditional CSS.
- View components are the main building blocks of a UI, and they have many styling properties.
- You can use borders in many ways to enhance the look of components. You can even use borders to create shapes, such as circles.
- You can use margins and padding to position components relative to one another.
- Absolute positioning lets you place a component anywhere within the parent container.
- Clipping can occur on Android devices, depending on how you set borders, margins, and padding.
- Specifying fonts other than the defaults can be tricky. Use the `Platform` component to select the appropriate font for the OS.
- Use general font styles like color, size, and weight to change the size and appearance of `Text` components.
- There are rendering differences between OSs, such as how the line height behaves differently between iOS and Android.
- Text-decorating styles can add underlines or drop shadows to text. The set of available styles differs from one OS to another.

5

Styling in depth

This chapter covers

- Platform-specific sizes and styles
- Adding drop shadows to components
- Moving and rotating components on the x- and y-axes
- Scaling and skewing components
- Using flexbox for layout

Chapter 4 introduced styling React Native components. It showed how to style `View` and `Text` components, styles you'll likely use every day and that mostly affect the look of a component. This chapter continues the discussion and goes into more depth with platform-specific styles; drop shadows; manipulating components with transformations such as translation, rotation, scaling, and skewing; and dynamically laying out components with flexbox.

Some of these topics may feel familiar. You used platform-specific styles and flexbox in several of the examples in chapter 4. We didn't cover them in detail, but you saw them in a few code listings.

This chapter expands on those topics. Transformations give you the power to manipulate components in two or three dimensions. You can translate components from one position to another, rotate components, scale components to different sizes, and skew components. Transforms are useful in their own right, but they will play a much bigger role in chapter 7, which discusses animation in detail.

We'll continue talking about some of the differences between platforms and look more deeply at flexbox. Because flexbox is a fundamental concept, it's important to properly understand it so you can create layouts and UIs in React Native. You'll probably use flexbox in every application you create. You'll use some of your new styling techniques to continue building new features into the `ProfileCard` example from the previous chapter.

5.1 *Platform-specific sizes and styles*

You've seen how to use the `Platform.select` utility function to choose fonts available only on iOS or Android. You used `Platform.select` to choose a monospaced font supported by each platform. You might not have thought much of that at the time, but it's important to keep in mind that you're developing for two different platforms. The styles you apply to a component may look or behave differently between the two OSs or even between different versions of iOS and Android.

You aren't coding for a single device; you're not even coding for a single OS. The beauty of React Native is that you're using JavaScript to create applications that can run on both iOS and Android. If you look through the React Native documentation, you'll see many components suffixed with `IOS` or `Android`, such as `ProgressBarAndroid`, `ProgressViewIOS`, and `ToolbarAndroid`, so it should come as no surprise that styles can be platform specific too.

You may not have noticed that you've never specified a size in pixels for anything, like `width: 300` vs `width: '300px'`. That's because even the concept of size is different between the iOS and Android operating systems.

5.1.1 *Pixels, points, and DPs*

Size can be a confusing topic, but it's important to keep in mind if you need to be absolutely precise when positioning components on the screen. Even if you're not trying to produce a high-fidelity layout, it will be useful to understand the concepts in case you encounter small discrepancies in your layouts from one device to another.

Let's start from the beginning and define a pixel. A *pixel* is the smallest unit of programmable color on a display. A pixel is typically made up of red, green, and blue (RGB) color components. By manipulating the intensity of each RGB value, the pixel emits a color you see. A pixel doesn't tell you anything until you start looking at the physical properties of the display: screen size, resolution, and dots per inch.

The *screen size* is the diagonal measurement of the screen, from one corner to another. For example, the original screen size of the iPhone was 3.5 inches, while the screen size of the iPhone X is 5.8 inches. Although the iPhone X is considerably bigger, the size doesn't mean anything until you understand how many pixels fit within that screen size.

Resolution is the number of pixels in the display, which is most typically expressed as the number of pixels along the width and height of the device. The original iPhone was 320×480 , while the iPhone X is 1125×2436 .

Screen size and resolution can then be used to calculate the pixel density: *pixels per inch* (PPI). You'll often see this expressed as *dots per inch* (DPI), which is a holdover term from the printing world, where a dot of color was printed on the page. PPI and DPI are often used interchangeably even though that's not exactly correct, so if you see DPI used in reference to a screen, know that PPI is what's truly being discussed.

The PPI gives you a measure of image sharpness. Imagine if two screens had the same resolution, 320×480 (half VGA). What would the same image look like on the 3.5-inch iPhone display versus a 17-inch HVGA monitor display? The same image would look much sharper on the iPhone, because it has 163 PPI versus the CRT monitor, which has 34 PPI. You can fit nearly five times as much information in the same physical space on the original iPhone. Table 5.1 compares the diagonal size, resolution, and PPI of the two devices.

Table 5.1 Comparison of a 17-inch HVGA monitor's PPI vs. the original iPhone's PPI

	HVGA monitor	Original iPhone
Diagonal size	17 inches	3.5 inches
Resolution	320×480	320×480
PPI	34	163

Why does this matter? Because neither iOS nor Android uses the actual physical measurements to render content to a device's screen. iOS uses an abstract measurement of points, and Android uses a similar abstract measurement of density-independent pixels.

When the iPhone 4 came on the scene, it had the same physical size as its predecessors; but it had a fancy new Retina screen with a resolution of 640×960 , quadrupling the resolution of the original device. If the iPhone had rendered images from existing apps at a 1:1 scale, everything would be drawn at a quarter size on the new Retina display. It would have been an insane proposition for Apple to make such a change and break all the existing apps.

Instead, Apple introduced the logical concept of a *point*. A point is a unit of distance that can be scaled independently of a device's resolution, so a 320×480 image that took up the entire screen on an original iPhone could be scaled up 2x to fully fit within the Retina display. Figure 5.1 provides a visualization of pixel density for several iPhone models.

The original iPhone's 163 PPI is the basis for the iOS point. An iOS point is $1/163$ of an inch. Without going into more detail, Android uses a similar measure called a *device-independent pixel* (DIP, often abbreviated DP). An Android DP is $1/160$ of an inch.

When defining styles in React Native, you use the logical concept of a pixel, a point on iOS, and a DP on Android. When working at the native level, you occasionally may need to work with device pixels by multiplying the logical pixels by the screen scale (for example, 2x, 3x).

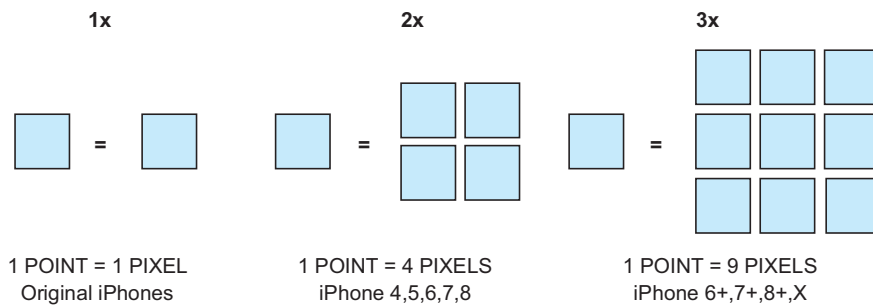


Figure 5.1 A visualization of points compared to pixel density for iPhones. The original iPhone had a resolution of 320×480 . The iPhone 4 had a resolution of 640×960 , quadruple the resolution of the original device. The iPhone 4 has twice the PPI (326 vs. 163), so images are said to be scaled up 2x.

5.1.2 Creating drop shadows with `ShadowPropTypesIOS` and `Elevation`

In chapter 4, you used the text shadow properties to add a drop shadow to the `ProfileCard` title. Both iOS and Android support adding a drop shadow to a `Text` component. It would be nice to spruce up more of the `ProfileCard` by adding drop shadows to the card and circular image container, but there isn't a common style property for `View` components to use between the two platforms.

That doesn't mean all is lost. The `ShadowPropTypesIOS` style can be used to add a drop shadow on iOS devices; it doesn't affect the z-order of the component. On Android, you can use the `Elevation` style to simulate a drop shadow, but it *does* affect the z-order of the component.

CREATING DROP SHADOWS IN iOS WITH `SHADOWPROPTYPESIOS`

Let's look at how to use `ShadowPropTypesIOS` styles to add drop shadows to a few view components. Figure 5.2 shows various shadow effects that can be achieved. Table 5.2 lists the specific settings used to achieve each shadow effect. The important takeaways are as follows:

- If you don't supply a value for `shadowOpacity`, you won't see a shadow.
- Shadows offsets are expressed in terms of width and height, but you can think of this as moving the shadow in the x and y directions. You can even specify negative values for width and height.
- A `shadowOpacity` of 1 is completely solid, whereas a value of 0.2 is more transparent.
- A value for `shadowRadius` effectively blurs the edges of the shadow. The shadow is more diffuse.

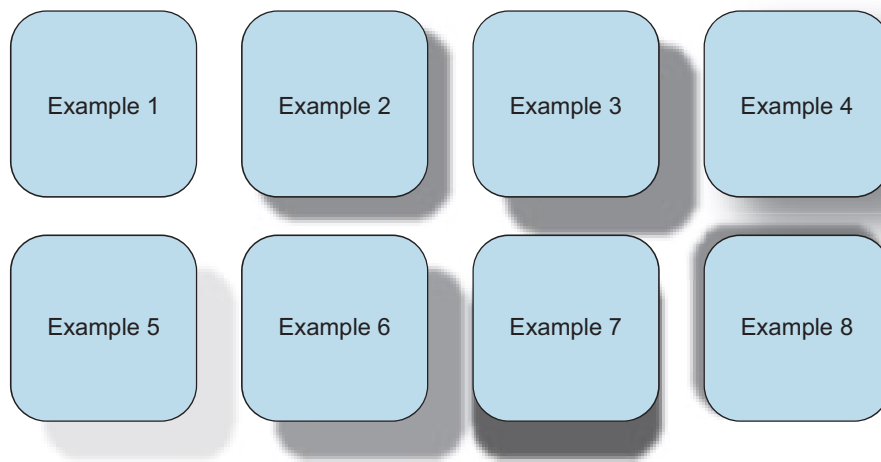


Figure 5.2 iOS-specific examples of how to apply `ShadowPropTypesIOS` styles to `View` components. Example 1 has a shadow applied but no opacity set, which causes the drop shadow to not be displayed. Example 2 has the same shadow effect but with opacity set to 1. Example 3 has a slightly larger shadow, and example 4 has the same size shadow with a shadow radius. Example 5 has the same shadow size, but opacity is changed from 1 to 0.2. Example 6 changes the color of the shadow. Example 7 shows the shadow applied in only one direction, and example 8 shows the shadow applied in the opposite direction.

Table 5.2 Shadow properties used to create the examples in figure 5.2

Example	shadowOffset				
	shadowColor	width (x)	height (y)	shadowOpacity	shadowRadius
1	Black	10	10		
2	Black	10	10	1	
3	Black	20	20	1	
4	Black	20	20	1	20
5	Black	20	20	0.2	
6	Red	20	20	1	
7	Black		20		
8	Black	-5	-5	1	

The code for this figure can be found in the git repository under `chapter5/figures/Figure-5.2-ShadowPropTypesIOS`. If you run the code for this example, remember to run it in the iOS simulator. On an Android device, you'll just see eight boring squares with rounded corners. `ShadowPropTypesIOS` styles are ignored on Android.

APPROXIMATING DROP SHADOWS ON ANDROID DEVICES WITH ELEVATION

How do you get the same effect on Android devices? The truth is, you can't. You *can* use Android's `elevation` style to affect the z-order of components. If two or more

components occupy the same space, you can decide which one should be in front by giving it the larger elevation and therefore the larger z-index, which will create a small drop shadow, but it isn't nearly as striking as the shadow effects you can achieve on iOS. Note that this only applies to Android, because iOS doesn't support the `elevation` style and will gladly ignore it if it's specified.

Nevertheless, let's see `elevation` in action. To do so, you'll create a View component with three boxes, each of which is positioned absolutely. You'll give them three different elevations—1, 2, and 3—and then you'll reverse the assignment of the elevations and see how that affects the layout. Figure 5.3 shows the results of these elevation adjustments.

Table 5.3 shows the absolute positions and elevations used for each group of boxes. Notice that nothing has changed except the elevation assigned to each of the boxes. iOS ignores the style and always renders box C on top of box B, and box B on top of box A. But Android respects the style and flips the order in which it renders the boxes, so box A is now on top of box B, and box B is on top of box C.

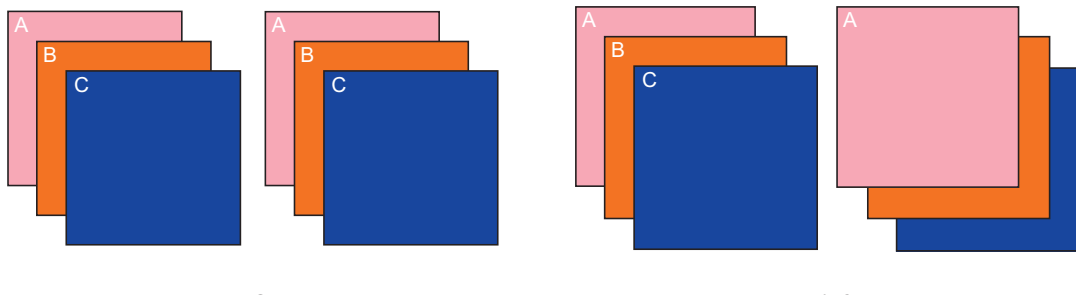


Figure 5.3 Examples of using the `elevation` style on iOS and Android. On iOS, elevation is ignored; all components retain the same z-order, so whatever component is last in the layout is on top. On Android, elevation is used, and the z-order is changed; in the second example, where the elevation assignments are reversed, A is on top.

Table 5.3 Elevation settings for figure 5.3

Example	color	top	left	elevation
A	Red	0	0	1
B	Orange	20	2	
C	Blue	40	40	3
A	Red	0	0	3
B	Orange	20	20	2
C	Blue	40	4	

5.1.3 Putting it into practice: drop shadows in the ProfileCard

Let's go back to the ProfileCard example from the last chapter and add some drop shadows that will look great on iOS and not so great on Android. You'll add a drop shadow to the entire ProfileCard container and to the circular image container. Figure 5.4 shows what you're shooting for on iOS and what you'll get on Android.

Notice that even with elevation applied on Android, you don't see much of a shadow. The reality is, on Android you'll never get close to the shadow effects that can be produced on iOS with React Native out of the box. If you really must have drop shadows on Android, then I suggest looking for a component on npm or yarn that does what you need. Experiment with different components, and see if you can get the Android version looking as sharp as the iOS version. I don't have any recommendations; I stay away from drop shadows or accept the differences.

The code in this chapter begins with listing 4.20: the completed ProfileCard example from chapter 4. Listing 5.1 only shows the changes needed to apply the drop shadows to the component. You don't need to add a lot of code to get the drop shadows on iOS. Look at the listing on an Android device, and see how the elevation setting causes the faintest of shadows.

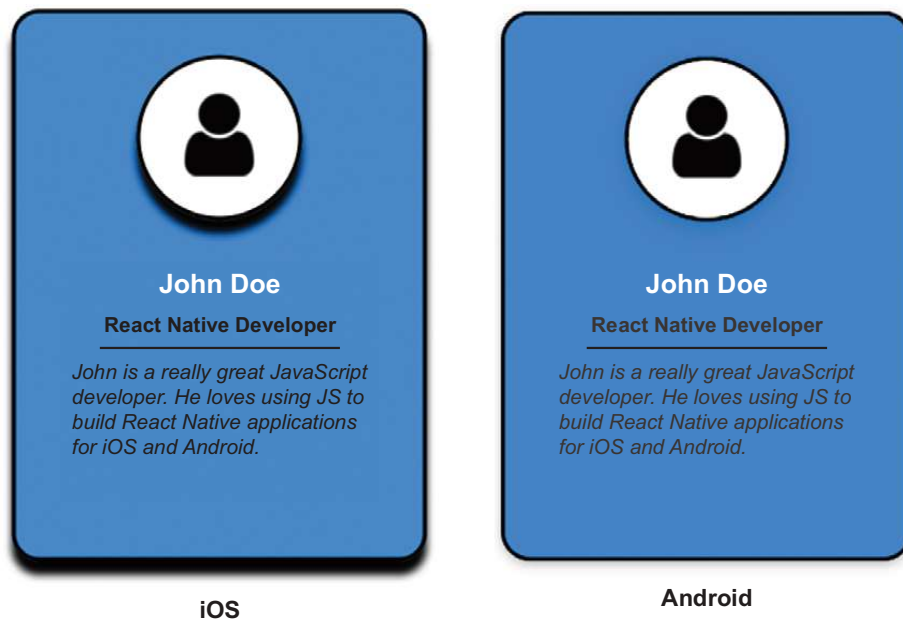


Figure 5.4 The ProfileCard on iOS and Android after drop shadows have been added to the card container and the circular image container. The drop shadows on iOS are created using the iOS-specific shadow properties: `shadowColor`, `shadowOffset`, and `shadowOpacity`. On Android, the `elevation` property is used to try to create depth. It produces only a minor shadow effect, far inferior to the shadows produced on iOS.

Listing 5.1 Adding drop shadows to the ProfileCard

```

import React, { Component } from 'react';
import { Image, Platform, StyleSheet, Text, View } from 'react-native';
...
cardContainer: {
  ...
  height: 400,
  ...Platform.select({
    ios: {
      shadowColor: 'black',
      shadowOffset: {
        height: 10
      },
      shadowOpacity: 1
    },
    android: {
      elevation: 15
    }
  })
},
cardImageContainer: {
  ...
  paddingTop: 15,
  ...Platform.select({
    ios: {
      shadowColor: 'black',
      shadowOffset: {
        height: 10,
      },
      shadowOpacity: 1
    },
    android: {
      borderWidth: 3,
      borderColor: 'black',
      elevation: 15
    }
  })
},
...

```

Imports the Platform utility component to programmatically select styles based on the platform

Adds a drop shadow to the card container based on the platform

Adds a drop shadow to the circular image container

Just as with font selection in chapter 4, you use the `Platform.select` function to apply different styles to components based on the platform: iOS or Android. In some cases, like the drop shadow, one platform may perform much better than the other; but in most cases the styles will behave the same on both platforms, which is an amazing benefit of React Native.

5.2 *Using transformations to move, rotate, scale, and skew components*

Up to this point, the styles we've discussed have mostly affected the appearance of components. You learned how to set properties like the style, weight, size, and color of borders and fonts. You applied background colors and shadow effects, and you saw how to manipulate the appearance of components relative to one another by using

margins and padding. But we haven't explored how to manipulate a component's position or orientation on the screen independent of everything else. How do you move a component on the screen, or rotate a component in a circle?

The answer is *transformations*. React Native provides a number of useful transforms that allow you to modify the shape and position of a component in 3D space. You can move components from one position to another, rotate components about all three axes, and scale and skew components in the x and y directions. Alone, transformations can produce some interesting effects, but their true power comes from sequencing them together to form animations.

This section will give you a firm understanding of transforms and how they affect the components to which they're applied. If you clearly understand what they do, you'll be better able to link them together to create meaningful animations later.

The `transform` style takes an array of transform properties that define how to apply a transformation to a component. For example, to rotate a component 90 degrees and shrink it by 50%, apply this transform to the component:

```
transform: [{rotate: '90deg', scale: .5}]
```

The `transform` style supports the following properties:

- `perspective`
- `translateX` and `translateY`
- `rotateX`, `rotateY`, and `rotateZ` (`rotate`)
- `scale`, `scaleX`, and `scaleY`
- `skewX` and `skewY`

5.2.1 3D effects with perspective

`perspective` gives an element 3D space by affecting the distance between the z plane and the user. This is used with other properties to give a 3D effect. The larger the `perspective` value, the greater the z-index of a component, which makes it appear closer to the user. If the z-index is negative, the farther away the component appears.

5.2.2 Moving elements along the x- and y-axes with `translateX` and `translateY`

The translation properties move an element along the x (`translateX`) or y (`translateY`) axis from the current position. This isn't very useful in normal development because you already have `margin`, `padding`, and other position properties available. But this becomes useful for animations, to move a component across the screen from one position to another.

Let's look at how to move a square using the `translateX` and `translateY` style properties. In figure 5.5, a square is placed in the center of the display and then moved in each of the four cardinal and four ordinal directions: NW (upper left), N (top), NE (upper right), W (left), E (right), SW (bottom left), S (bottom), and SE (bottom right). In each case, the center of the square is moved by 1.5 times the square's size in the x or y direction or in both directions.

When studying geometry, you typically see the positive y-axis drawn going up instead of down. But on mobile devices, the convention is to have the positive y-axis go down the screen, which reflects the most common interaction of scrolling down the screen to view more content. Coupled with that bit of knowledge, it's pretty easy to see how moving the center square in figure 5.5 in the positive x direction and in the positive y direction results in the square ending up in the bottom right corner. By combining `translateX` and `translateY`, you can move components in any direction in the Cartesian plane (x-y plane).

There's no corresponding translation for movement in the z plane. The z-axis is perpendicular to the face of the device, which means you're looking straight at it. Moving a component forward or backward would be imperceptible without some corresponding size change. The perspective transform is intended to handle this type of visual effect.

In the next section, we'll use the same example and focus on the center row, where the center square was translated to the left and to the right. You'll see what happens when you rotate components along each of the three axes.

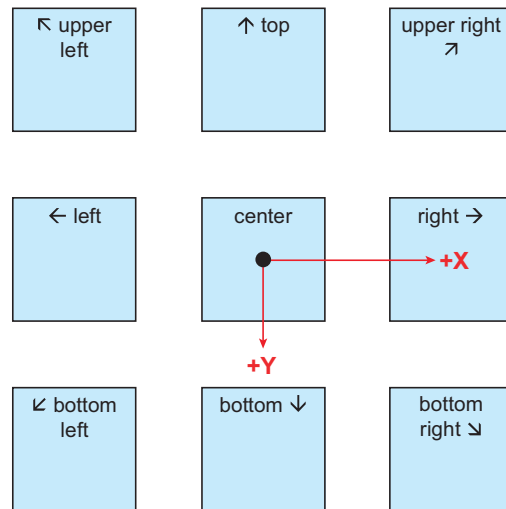


Figure 5.5 A depiction of a center square being moved in each of the four cardinal and four ordinal directions: NW (upper left), N (top), NE (upper right), W (left), E (right), SW (bottom left), S (bottom), and SE (bottom right)

5.2.3 Rotating elements with `rotateX`, `rotateY`, and `rotateZ` (`rotate`)

The rotation properties do exactly what it sounds like they would: they rotate elements. Rotation occurs along an axis: x, y, or z. The origin of the rotation is the center point of the element before any transformations are applied, so if you use `translateX` or `translateY`, keep in mind that the rotation will be around the axis at the original location. The amount of rotation can be specified in either degrees (`deg`) or radians (`rad`). The examples use degrees:

```
transform: [{ rotate: '45deg' }]
transform: [{ rotate: '0.785398rad' }]
```

Figure 5.6 shows the positive and negative direction of rotation for each axis. The `rotate` transform does the same thing as the `rotateZ` transform.

Let's rotate a 100×100 square about the x-axis in increments of 35° , as shown in figure 5.7. A center line is drawn through each square, so it's easier to see how the squares are rotating. You can visualize rotation about the x-axis in the positive direction as the square rotating from the top into the page. The bottom is coming closer to you as the top moves further away.

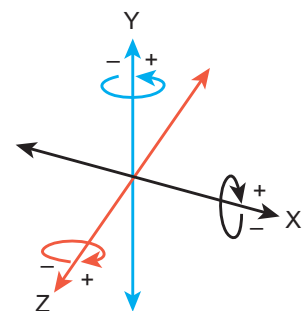


Figure 5.6 The positive and negative direction of rotation for each axis

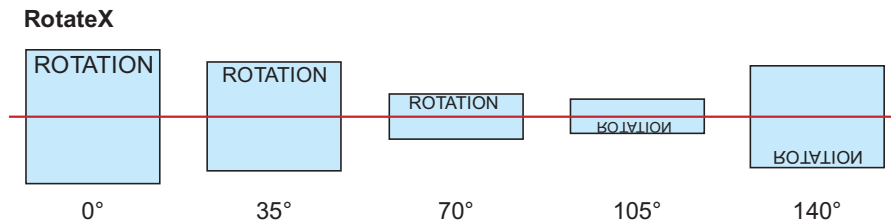


Figure 5.7 Rotating a 100×100 square about the x-axis in increments of 35° . After 90° , the “ROTATION” label can be seen through the element, upside down.

At 90° , you’re looking at the square on its edge (because it doesn’t have any thickness, you don’t see a thing). After the square has rotated past the 90° mark, you start to see the back of the square. If you look closely in figure 5.7, you can see the “ROTATION” label is upside down, because you’re looking through what was the back of the square.

The next example will rotate the same 100×100 square about the y-axis, instead continuing to use increments of 35° to demonstrate the rotation (see figure 5.8). Picture the right side of the square moving away from you, into the page. After the square has rotated beyond the 90° mark, you can see the “ROTATION” label through the component. Because you’re looking through the back of the component, the text appears backward.

Compare figure 5.8 to figure 5.7. Fundamentally, rotation about the y-axis is no different than rotation about the x-axis. I aligned the squares in figure 5.8 vertically so you can easily see the axis of rotation. I like to visualize rotation in the y-axis by picturing a book opening and closing: if you’re opening a book, the cover is rotating in the negative direction. If you’re closing the book, then you’re rotating the cover in the positive direction.

Rotation about the z-axis is the easiest to visualize. Rotation in the positive direction spins the object in a clockwise fashion, and rotation in the negative direction spins the square in a counterclockwise fashion. For this example, shown in figure 5.9, the axis of rotation is

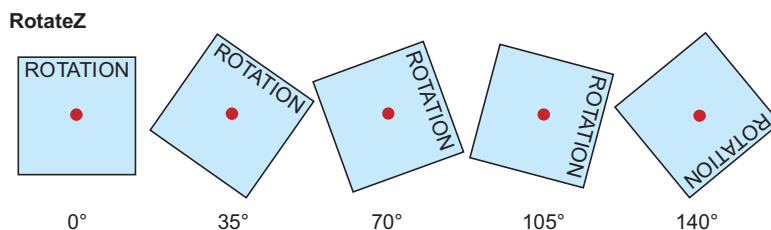


Figure 5.9 Rotating a 100×100 square about the z-axis in increments of 35° . The positive rotation is clockwise, and the negative rotation is counterclockwise.

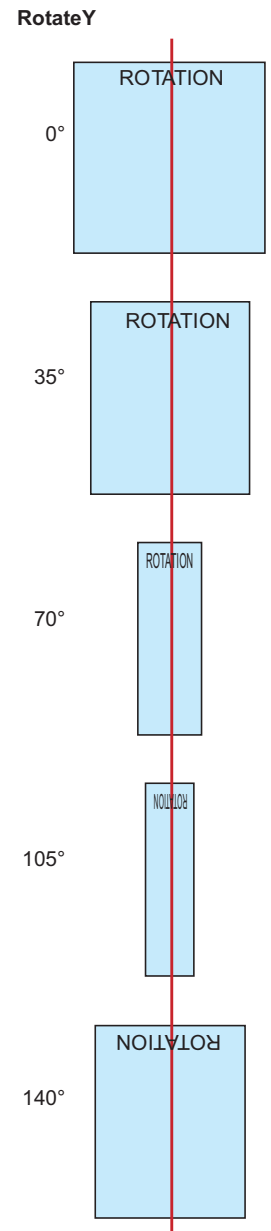


Figure 5.8 Rotating a 100×100 square about the y-axis in increments of 35° . After 90° , the “ROTATION” label can be seen through the element, backward.

represented as a dot in the center of the square, because the z-axis is, basically, your line of sight; it goes straight into the screen.

Hopefully, it's fairly obvious now how the rotation transforms work. Understanding the direction in which positive and negative rotations affect an object is probably the most complex part. But when you start combining other transforms in conjunction with rotation, you may be surprised by the results. Remember that the transform property is an array of transforms, so multiple transforms can be supplied at once, and order matters! Specifying a transform and switching the order of the elements in the array will yield different results.

Let's investigate how altering the order in which the transforms are specified affects the final layout. Let's apply three different transforms to a square: translate in the y direction 50 points, translate in the x direction 150 points, and rotate the square 45°. Figure 5.10 specifies the transform in the order just described. The original/previous position of the square has a dotted border and the new position of the square has a solid outline, so you can see how the transformation affects the position and orientation of the original square.

The result in figure 5.10 is pretty much as expected, but what will happen if you apply the rotation after moving the square in the y direction? Look at figure 5.11 and find out.

Whoa, what happened? The square is completely off the screen after the transformations are applied! It might not be immediately obvious what happened, which is why figure 5.11 is annotated with the new axis orientation.

After the rotation, the +x- and +y-axes are no longer oriented vertically and horizontally on the screen: they're rotated by 45°. When the `translateX` transform is applied, the square is moved 150 points in the +x direction, but the +x direction is now at a 45° angle from the original x-axis.

The next section shows another interesting aspect of rotational transforms.

transform: [{translateY: 50},{translateX: 150},{rotate: '45deg'}]

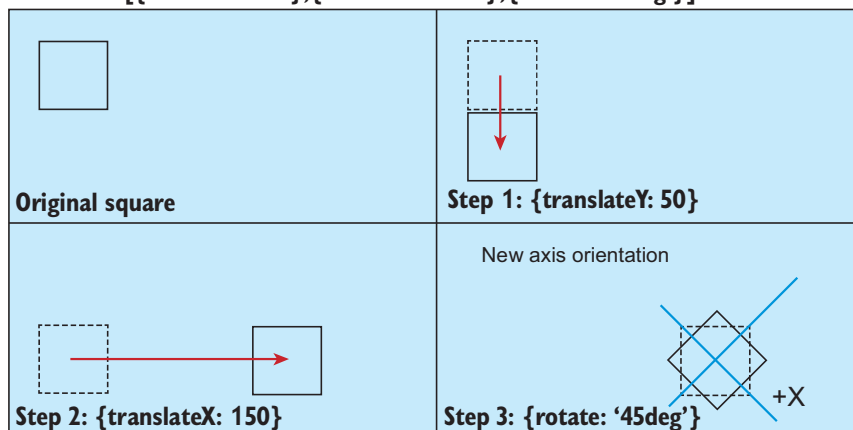


Figure 5.10 Applying transform: [{translateY: 50},{translateX: 150},{rotate: '45deg'}] to the original square

transform: [{translateY: 50},{rotate: '45deg'},{translateX: 150}]

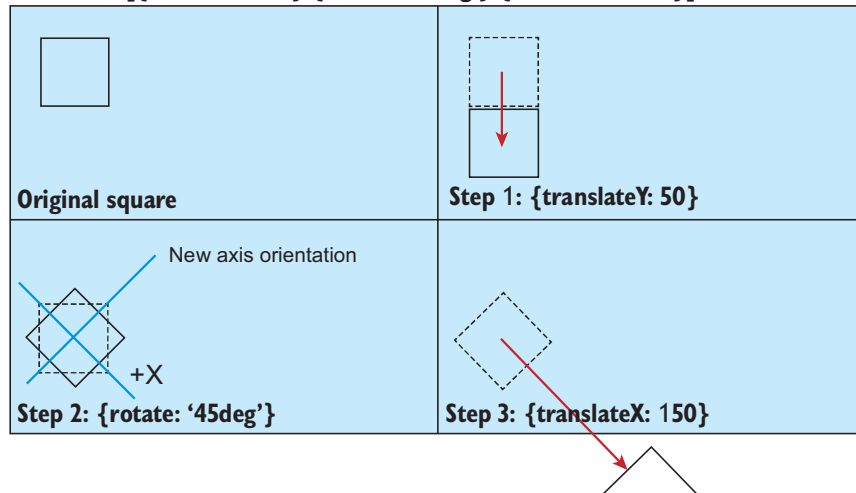


Figure 5.11 Applying transform: [{translateY: 50},{rotate: '45deg'},{translateX: 150}] to the original square. Rotating the square changes the orientation of the x- and y-axes, so when the square is translated 150 points in the +x direction, it's moved diagonally down and out of the viewport.

5.2.4 Setting visibility when rotating an element more than 90°

If you look back at figures 5.7 and 5.8, when you rotate the square about the x- or y-axis and go beyond the 90° point, you can still see the text that was on the front face of the square. The `backfaceVisibility` property dictates whether an element is visible when the element is rotated more than 90°. This property can be set to either 'visible' or 'hidden'. This property isn't a transform, but it gives you the ability to hide or show elements when viewing the back face of an object.

The `backfaceVisibility` property defaults to 'visible', but if you changed `backfaceVisibility` to 'hidden', you wouldn't see the element at all once the component rotated more than 90° in either the x or y direction. In figures 5.7 and 5.8, the squares corresponding to the 105° and 140° rotations would disappear. If that sounds confusing, look at figure 5.12.

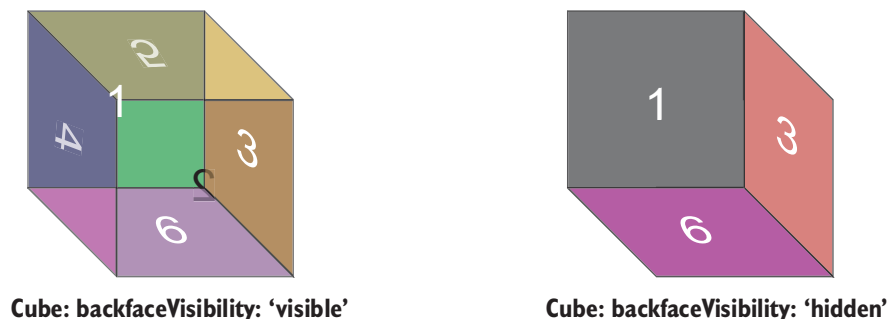


Figure 5.12 A demonstration of how setting the `backfaceVisibility` property to 'hidden' hides elements that have rotated beyond 90°. The cube on the left shows faces 2, 4, and 5, all of which have rotated 180°. The cube on the right has hidden those faces.

In the figure, you can easily see the effect of setting `backfaceVisibility` to 'hidden'. It's also easy to see how this behavior might be beneficial during animations. When the faces of the cube rotate out of sight, you want them to be hidden.

5.2.5 Scaling objects on the screen with `scale`, `scaleX`, and `scaleY`

This section talks about scaling objects on the screen. There are many practical uses for scaling and many patterns that take advantage of its capabilities. For instance, scaling can be used to create thumbnails of objects. You've seen this in many applications; the user taps a thumbnail, and an animation gradually scales the object back up to full size. It's a common transition technique that provides a nice visual effect.

You'll learn the basics of scaling objects and then use those skills to create a thumbnail of the `ProfileCard` that opens to full size when pressed. Later, this chapter discusses flexbox and how it can be used to manage a bunch of `ProfileCard` thumbnails in a gallery interface, from which you can press profiles to view them in more detail.

`scale` multiplies the size of the element by the number passed to it, the default being 1. To make an element appear larger, pass a value larger than 1; to make it appear smaller, pass a value smaller than 1.

The element can also be scaled along a single axis using `scaleX` or `scaleY`. `scaleX` stretches the element horizontally along the x-axis, and `scaleY` stretches the element vertically along the y-axis. Let's create a few squares to show the effects of scaling: see figure 5.13.

Nothing unusual happens; scaling an object is pretty straightforward. Listing 5.2 shows how simple it is.

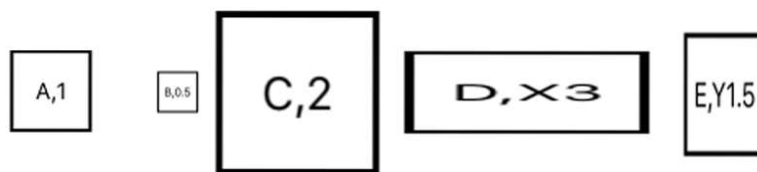


Figure 5.13 Examples of how scaling transforms the original square. All the squares start the same size and shape as A, which has the default scale of 1. B scales the square by 0.5, shrinking it. C scales the square by 2, enlarging it. D uses `scaleX`, transforming the square along the x-axis by 3x. E uses `scaleY`, transforming the square along the y-axis by 1.5x.

Listing 5.2 Scaling squares using `scale`, `scaleX`, and `scaleY`

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



```

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <Example style={{}}>A,1</Example>
        <Example style={{transform: [{scale: 0.5}]}>B,0.5</Example>
        <Example style={{transform: [{scale: 2}]}>C,2</Example>
        <Example style={{transform: [{scaleX: 3}]}>D,X3</Example>
        <Example style={{transform: [{scaleY: 1.5}]}>E,Y1.5</Example>
      </View>
    );
  }
}

const Example = (props) => (
  <View style={ [styles.example, props.style] }>
    <Text>
      {props.children}
    </Text>
  </View>
);

const styles = StyleSheet.create({
  container: {
    marginTop: 75,
    alignItems: 'center',
    flex: 1
  },
  example: {
    width: 50,
    height: 50,
    borderWidth: 2,
    margin: 15,
    alignItems: 'center',
    justifyContent: 'center'
  },
});

```

Scales the default square only in the x direction, stretching it horizontally

Default 50 × 50 square with no scaling applied

Scales the default square by 0.5, shrinking it

Scales the default square by 2, making it larger

Scales the default square only in the y direction, stretching it vertically

5.2.6 Using the scale transform to create a thumbnail of the ProfileCard

Now that you've seen scaling in action, let's use this technique to create a thumbnail of the ProfileCard. Normally you'd animate what I'm about to show you, to avoid flickering, but let's see how to use scaling in a practical way. Figure 5.14 shows a small, scaled-down version of the ProfileCard component—a thumbnail. If you press the thumbnail, the component will return to full size. If you press the full-size component, it will collapse back down into a thumbnail view.

Begin with the code from listing 5.1. As far as styles go, you only need to add one new style to do the scaling transform from full size to thumbnail. The remainder of the code reorganizes the component's pieces into a more reusable structure and provides the touch capabilities to handle the onPress events.

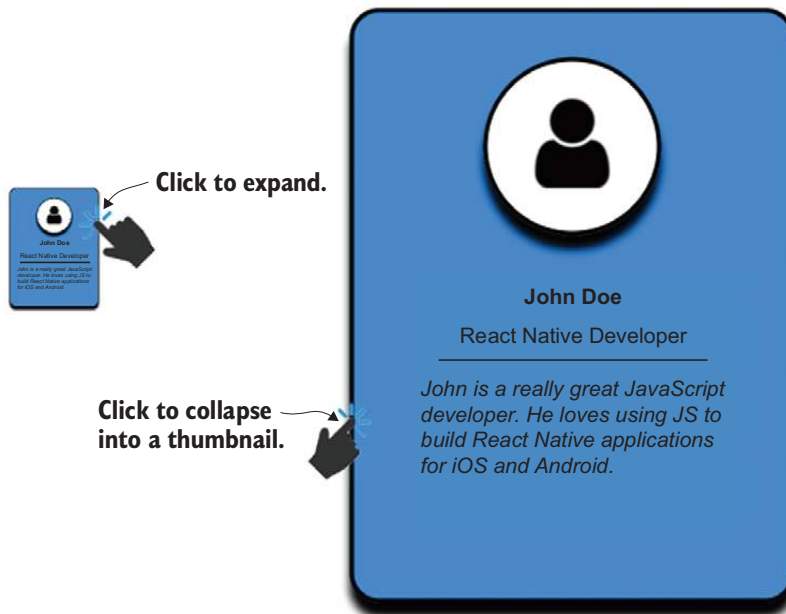


Figure 5.14 Scaling the full-sized `ProfileCard` down 80% into a thumbnail image. Pressing the thumbnail restores the `ProfileCard` to its original size, and pressing the full-sized component collapses the component into a thumbnail.

Listing 5.3 Scaling `ProfileCard` from full size to thumbnail

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
import update from 'immutability-helper';
import { Image, Platform, StyleSheet, Text,
  TouchableHighlight, View } from 'react-native';

const userImage = require('./user.png');

const data = [{
  image: userImage,
  name: 'John Doe',
  occupation: 'React Native Developer',
  description: 'John is a really great Javascript developer. ' +
    'He loves using JS to build React Native applications ' +
    'for iOS and Android',
  showThumbnail: true
}];

const ProfileCard = (props) => {
  const { image, name, occupation,
    description, onPress, showThumbnail } = props;
  let containerStyles = [styles.cardContainer];
```

The immutability helper function update lets you update a specific piece of the component's state.

`PropTypes` lets you specify what properties the `ProfileCard` component can accept.

The `TouchableHighlight` component enables touch processing.

Data elements have been extracted to generalize the component.

The `ProfileCard` component is now separated from the App code.

```

if (showThumbnail) {
  containerStyles.push(styles.cardThumbnail);
}

return (
  <TouchableHighlight onPress={onPress}>
    <View style={containerStyles}>
      <View style={styles.cardImageContainer}>
        <Image style={styles.cardImage} source={image}/>
      </View>
      <View>
        <Text style={styles.cardName}>
          {name}
        </Text>
      </View>
      <View style={styles.cardOccupationContainer}>
        <Text style={styles.cardOccupation}>
          {occupation}
        </Text>
      </View>
      <View>
        <Text style={styles.cardDescription}>
          {description}
        </Text>
      </View>
    </View>
  </TouchableHighlight>
)
};

ProfileCard.propTypes = {
  image: PropTypes.number.isRequired,
  name: PropTypes.string.isRequired,
  occupation: PropTypes.string.isRequired,
  description: PropTypes.string.isRequired,
  showThumbnail: PropTypes.bool.isRequired,
  onPress: PropTypes.func.isRequired
};

export default class App extends Component<{}> {

  constructor(props, context) {
    super(props, context);
    this.state = {
      data: data
    }
  }

  handleProfileCardPress = (index) => {
    const showThumbnail = !this.state.data[index].showThumbnail;
    this.setState({
      data: update(this.state.data,
        {[index]: {showThumbnail: {$set: showThumbnail}}})
    });
  };

  render() {
    const list = this.state.data.map(function(item, index) {

```

If showThumbnail is true, the component is scaled down by 80%.

Processes presses to minimize and maximize the component

Component state is maintained in the higher-order App component.

Handler function to process onPress events

List (array) of ProfileCard components

```

const { image, name, occupation, description, showThumbnail } = item;
return <ProfileCard key={'card-' + index}
    image={image}
    name={name}
    occupation={occupation}
    description={description}
    onPress={this.handleProfileCardPress.bind(this, index)}
    showThumbnail={showThumbnail}/>
}, this);

return (
  <View style={styles.container}>
    {list}
  </View>
);
}
}
...
cardThumbnail: {
  transform: [{scale: 0.2}]
},
...

```

← Renders the list in the overall container

← The cardThumbnail style reduces the component's size by 80%.

By reorganizing the structure of the component, you can better handle adding more ProfileCard components to the application. In section 5.3, you'll add more ProfileCards and see how to organize them into a gallery layout.

5.2.7 *Skewing elements along the x- and y-axes with skewX and skewY*

Before we leave transforms and talk about layout, let's look at the skewX and skewY transformations. In the source code that produced the cubes for the backfaceVisibility example shown in figure 5.12 (github chapter5/figures/Figure-5.12-BackfaceVisibility), you can see that skewing the squares was essential to producing the three-dimensional affect for the cube faces. Let's discuss what skewX and skewY do, so when you explore the source code in detail, you'll understand what you're seeing.

The skewX property skews an element along the x-axis. Similarly, the skewY property skews an element along the y-axis. Figure 5.15 shows the results of skewing a square as follows:

- Square A has no transformation applied to it.
- Square B is skewed along the x-axis by 45°.
- Square C is skewed along the x-axis by -45°.
- Square D is skewed along the y-axis by 45°.
- Square E is skewed along the y-axis by -45°.

As with scaling, skewing an element is relatively simple: provide an angle, and specify the axis. The next listing gives all the details.

NOTE At the time of writing, the skewX transform doesn't work correctly on Android.

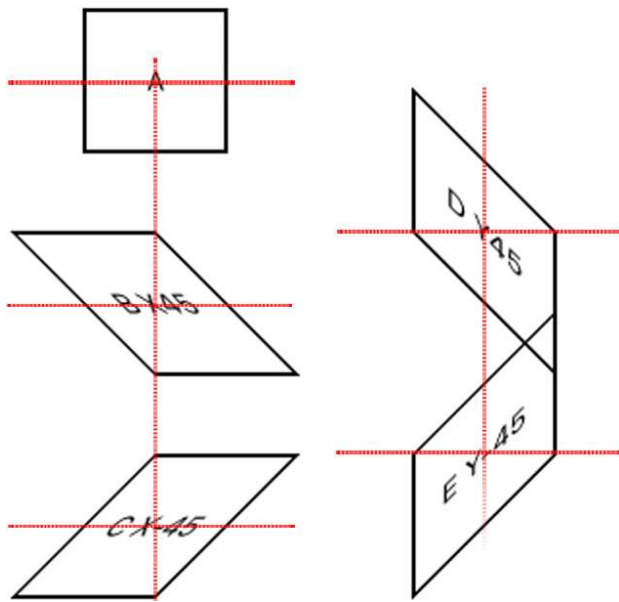


Figure 5.15 Examples of skewing a square along the x- and y-axes on iOS. Square A has no transformation applied. Square B is skewed along the x-axis by 45°. Square C is skewed along the x-axis by -45°. Square D is skewed along the y-axis by 45°, and square E is skewed along the y-axis by -45°.

Listing 5.4 Examples showing how skewing transforms a square

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

```
export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <Example style={{}}>A</Example>
        <Example style={{transform: [{skewX: '45deg'}]}}>
          B X45
        </Example>
        <Example style={{transform: [{skewX: '-45deg'}]}}>
          C X-45
        </Example>
        <Example style={{transform: [{skewY: '45deg'}]}}>
          D Y45
        </Example>
        <Example style={{transform: [{skewY: '-45deg'}]}}>
          E Y-45
        </Example>
      </View>
    );
  }
}
```

Skews the square 45°
along the x-axis

Skews the square -45°
along the x-axis

Skews the square 45°
along the y-axis

Skews the square -45°
along the y-axis

```

    }
  }

  const Example = (props) => (
    <View style={ [styles.example, props.style] }>
      <Text>
        {props.children}
      </Text>
    </View>
  );

  const styles = StyleSheet.create({
    container: {
      marginTop: 50,
      alignItems: 'center',
      flex: 1
    },
    example: {
      width: 75,
      height: 75,
      borderWidth: 2,
      margin: 20,
      alignItems: 'center',
      justifyContent: 'center'
    }
  });

```

5.2.8 Transformation key points

We've covered a lot of transformative ideas in this section! Some of them were relatively simple, while others may have been hard to visualize at first. I didn't show many examples that combine transforms, so you could focus on what individual transforms do. I encourage you to take any of the examples and include additional transformations, to experiment and see what happens.

In chapter 7, when we discuss animation, you'll see how transformations can make things come alive. For now, take away these key points:

- The origin of the x- and y-axes is at upper left, meaning the positive direction for y is down the screen. You saw this with absolute positioning in the previous chapter, but it's likely the opposite of what you're used to, which can make it hard to reason about what a transformation will do.
- The origin for rotations and translations is always at the element's original location. You can't translate an object in the x or y direction and then rotate it about a new center point.

Transformations are a great way to move components around the screen, but you won't use them on an everyday basis. Most often, you'll use Yoga, a layout engine that implements much of the W3C's flexbox web specification. In the next section, we'll discuss Yoga's flexbox implementation in detail.

5.3 Using flexbox to lay out components

Flexbox is a layout implementation that React Native uses to provide an efficient way for users to create UIs and control positioning. The React Native flexbox implementation is based on the W3C flexbox web specification but doesn't share 100% of the API. It aims to give you an easy way to reason about, align, and distribute space among items in a layout, even when their size isn't known or is dynamic.

NOTE Flexbox layout is only available for use on View components.

You've already seen flexbox used in many of the examples. It's powerful and makes laying out items so much easier than alternative methods that it's difficult *not* to use it. You'll benefit greatly by taking time to understand the material in this section. Here are the alignment properties used to control the flexbox layout: `flex`, `flexDirection`, `justifyContent`, `alignItems`, `alignSelf`, and `flexWrap`.

5.3.1 Altering a component's dimensions with flex

The `flex` property specifies the ability of a component to alter its dimensions to fill the space of the container it's in. This value is relative to the `flex` properties specified for the rest of the items in the same container.

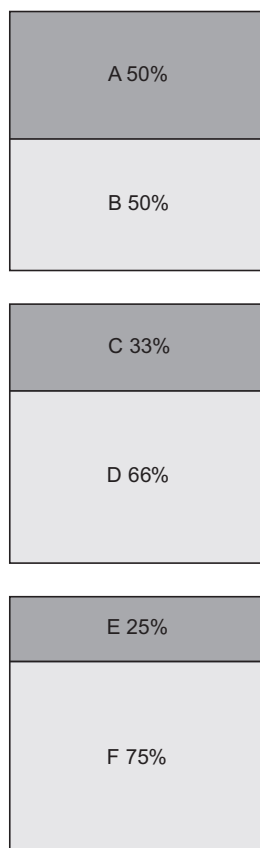


Figure 5.16 Three examples of layouts using the `flex` property. The top example is 1:1, with `A = {flex: 1}` and `B = {flex: 1}`, resulting in each taking up 50% of the space. The middle example is 1:2, with `C = {flex: 1}` and `D = {flex: 2}`, resulting in C taking up 33% of the space and D taking up 66%. The bottom example is 1:3, with `E = {flex: 1}` and `F = {flex: 3}`, resulting in E taking up 25% of the space and F taking up 75% of the space.

If you have a View element with a height of 300 and a width of 300, and a child View element with a property of `flex: 1`, then the child view will completely fill the parent view. If you decide to add another child element with a `flex` property of `flex: 1`, each view will take up equal space in the parent container. The `flex` number is only important relative to the other `flex` items occupying the same space.

Another way to look at this is to think of the `flex` properties as being percentages. For example, if you want the child components to take up 66.6% and 33.3%, respectively, you can use `flex:66` and `flex:33`. Rather than `flex:66` and `flex:33`, you can specify `flex:2` and `flex:1` and achieve the same layout effect.

To better understand how this works, let's look at a few examples shown in figure 5.16. These are easily achieved by setting the appropriate `flex` value on the individual elements. The following listing shows the steps necessary to create such a layout.

Listing 5.5 Flex views with 1:1 ratio, 1:2, and 1:3 ratios

```
...
render() {
  return (
    <View style={styles.container}>
      <View style={ [styles.flexContainer] }>
        <Example style={ [{flex: 1}, styles.darkgrey] }>A 50%</Example>
        <Example style={ [{flex: 1}] }>B 50%</Example>
      </View>
      <View style={ [styles.flexContainer] }>
        <Example style={ [{flex: 1}, styles.darkgrey] }>C 33%</Example>
        <Example style={ {flex: 2} }>D 66%</Example>
      </View>
      <View style={ [styles.flexContainer] }>
        <Example style={ [{flex: 1}, styles.darkgrey] }>E 25%</Example>
        <Example style={ {flex: 3} }>F 75%</Example>
      </View>
    </View>
  );
}
...
```

The items have the same flex value, so they take up the same amount of space in their parent container.

C takes up 1/3 of the total space, and D takes up 2/3 of the total space.

E takes up 1/4 of the total space, and F takes up 3/4 of the total space.

5.3.2 Specifying the direction of the flex with `flexDirection`

In the previous examples, the items in flex containers are laid out in a column (y-axis), meaning top to bottom. A is stacked on B, C is stacked on D, and E is stacked on F. Using the `flexDirection` property, you can change the primary axis of the layout, and therefore change the direction of the layout. `flexDirection` is applied to the parent view that contains

All that's needed to achieve the layout in figure 5.17 is to add a single line of code to the `flexContainer` style, which is the parent container for each of the example components. Changing `flexDirection` on this container affects the layout of all its flex children. Add `flexDirection: 'row'` to the style, and see how it changes the layout.

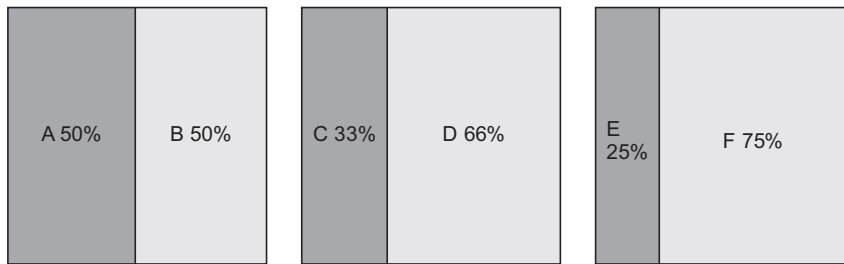


Figure 5.17 The same example as in figure 5.16, but with `flexDirection` set to `'row'`. Now the items take up space horizontally within the row rather than vertically within the column.

Listing 5.6 Adding `flexDirection: 'row'` to the parent container

```
flexContainer: {
  width: 150,
  height: 150,
  borderWidth: 1,
  margin: 10,
  flexDirection: 'row'
},
```

flexContainer is the parent container of each example.

Causes the children to be laid out horizontally

The child elements now appear left to right. There are two options for `flexDirection`: `'row'` and `'column'`. The default setting is `'column'`. If you don't specify a `flexDirection` property, content will be laid out in a column. This property is something you'll use a lot when developing apps in React Native, so it's important to grasp it and understand how it works.

5.3.3 Defining how space is used around a component with `justifyContent`

Using the `flex` property, you can specify how much space each component takes up in its parent container; but what if you're not trying to take up the entire space? How can you use flexbox to lay out components using their original size?

`justifyContent` defines how space is distributed between and around flex items along the primary axis of the container (the flex direction). `justifyContent` is declared on the parent container. Five options are available:

- `center` causes the children to be centered within the parent container. The free space is distributed on both sides of the clustered group of children.
- `flex-start` groups the components at the beginning of the flex column or row, depending on what value is assigned to `flexDirection`. `flex-start` is the default value for `justifyContent`.
- `flex-end` acts in the opposite manner: it groups items together at the end of the container.
- `space-around` attempts to evenly distribute space around each element. Don't confuse this with distributing the elements evenly in the container; the space is distributed around the elements. If it were based on the elements, you'd expect $space - element - space - element - space$

Instead, flexbox allocates the same amount of space on each side of the element, yielding

space – element – space – space – element – space

In both cases, the amount of whitespace is the same; but in the latter, the space between elements is greater.

- `space-between` doesn't apply spacing at the start or end of the container. The space between any two consecutive elements is the same as the space between any other two consecutive elements.

Figure 5.18 demonstrates how each of the `justifyContent` properties distributes space between and around the flex elements. Every example uses two elements to help depict what is happening.

Listing 5.7 shows the code used to generate figure 5.18. Look at it carefully, to understand how it works, and then try to do the following: add more elements to each example to see what happens as the number of items increases; and set `flexDirection` to `row` to see what happens when the items are laid out horizontally instead of vertically.

Listing 5.7 Examples showing the `justifyContent` options

```
...
render() {
  return (
    <View style={styles.container}>
      <FlexContainer style={[{justifyContent: 'center'}]}>
        <Example>center</Example>
        <Example>center</Example>
      </FlexContainer>
      <FlexContainer style={[{justifyContent: 'flex-start'}]}>
        <Example>flex-start</Example>
        <Example>flex-start</Example>
      </FlexContainer>
      <FlexContainer style={[{justifyContent: 'flex-end'}]}>
        <Example>flex-end</Example>
        <Example>flex-end</Example>
      </FlexContainer>
      <FlexContainer style={[{justifyContent: 'space-around'}]}>
        <Example>space-around</Example>
        <Example>space-around</Example>
      </FlexContainer>
      <FlexContainer style={[{justifyContent: 'space-between'}]}>
        <Example>space-between</Example>
        <Example>space-between</Example>
      </FlexContainer>
    </View>
  );
}
...
```

Uses the `justifyContent: 'center'` option

Uses the `justifyContent: 'flex-start'` option

Uses the `justifyContent: 'flex-end'` option

Uses the `justifyContent: 'space-around'` option

Uses the `justifyContent: 'space-between'` option

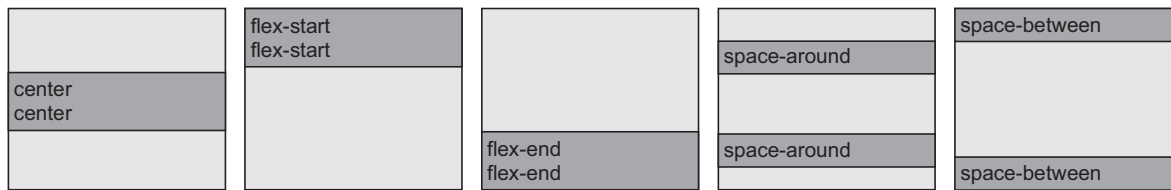


Figure 5.18 Examples of how `justifyContent` affects the distribution of space between flexible child elements for each of the supported options: `center`, `flex-start`, `flex-end`, `space-around`, and `space-between`.

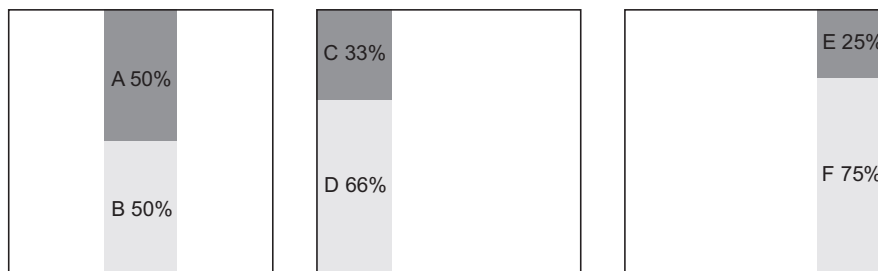


Figure 5.19 Modified examples from figure 5.16, using the non-default `alignItems` properties: `center`, `flex-start`, and `flex-end`

5.3.4 Aligning children in a container with `alignItems`

`alignItems` defines how to align children along the secondary axis of their container. This property is declared on the parent view and affects its flex children just as `flexDirection` did. There are four possible values for `alignItems`: `stretch`, `center`, `flex-start`, and `flex-end`.

`stretch` is the default, used in figures 5.17 and 5.18. Each example component is stretched to fill its parent container. Figure 5.19 revisits figure 5.16 and shows what happens with the other options: `center`, `flex-start`, and `flex-end`. Because a precise width isn't specified for the example components, they only take up as much space horizontally as is necessary to render their contents rather than stretching to fill the space. In the first case, `alignItems` is set to `'center'`. In the second case, `alignItems` is set to `'flex-start'`. And last `alignItems` is set to `'flex-end'`. Use listing 5.8 to change the alignments on each of the examples from listing 5.5.

Listing 5.8 Using non-default `alignItems` properties

```
render() {
  return (
    <View style={styles.container}>
      <View style={ [styles.flexContainer,
                    {alignItems: 'center'} ] }>
        <Example style={ [styles.darkgrey] }>A 50%</Example>
        <Example>B 50%</Example>
      </View>
    </View>
  )
}
```

Changes the `alignItems` property to `center`

```

</View>
<View style={ [styles.flexContainer,
               {alignItems: 'flex-start'} ] }>
  <Example style={ [styles.darkgrey] }>C 33%</Example>
  <Example style={{flex: 2}}>D 66%</Example>
</View>
<View style={ [styles.flexContainer,
               {alignItems: 'flex-end'} ] }>
  <Example style={ [styles.darkgrey] }>E 25%</Example>
  <Example style={{flex: 3}}>F 75%</Example>
</View>
</View>
);
}

```

Changes alignItems to flex-start

Changes alignItems to flex-end

Now that you've seen how to use the other alignItems properties and their effects on the default column layout, why don't you set flexDirection to 'row' and see what happens?

5.3.5 Overriding the parent container's alignment with alignSelf

So far, all the flex properties have been applied to the parent container. alignSelf is applied directly to an individual flex child.

With alignSelf, you can access the alignItems property for individual elements within the container. In essence, alignSelf gives you the ability to override whatever alignment was set on the parent container, so a child object can be aligned independently of its peers. The available options are auto, stretch, center, flex-start, and flex-end. The default value is auto, which takes the value from the parent container's alignItems setting. The remaining properties affect the layout in the same way as their corresponding properties on alignItems.

In figure 5.20, the parent container doesn't have alignItems set, so it defaults to stretch. In the first example, the auto value inherits stretch from its parent container. The next four examples lay out exactly as you'd expect. The final example has no alignSelf property set, so it defaults to auto and is laid out the same as the first example.

Listing 5.9 does something a little different. Rather than supply the style directly to the Example element, you create a new component property: align. It's passed down to the Example component and used to set alignSelf. Otherwise, the example is the same as many others in this chapter; it explores the effects of each value applied to the style.

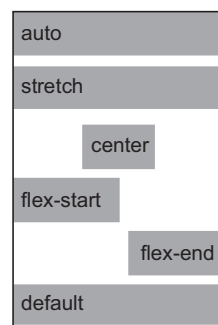


Figure 5.20 How each alignSelf property affects the layout when its parent container's alignItems property is set to the default value of stretch

Listing 5.9 Using alignSelf to override the parent's alignItems

```

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

```

```

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <FlexContainer style={[]}>
          <Example align='auto'>auto</Example>
          <Example align='stretch'>stretch</Example>
          <Example align='center'>center</Example>
          <Example align='flex-start'>flex-start</Example>
          <Example align='flex-end'>flex-end</Example>
          <Example>default</Example>
        </FlexContainer>
      </View>
    );
  }
}

const FlexContainer = (props) => (
  <View style={[styles.flexContainer, props.style]}>
    {props.children}
  </View>
);

const Example = (props) => (
  <View style={[styles.example,
    styles.lightgrey,
    {alignSelf: props.align || 'auto'},
    props.style
  ]}>
    <Text>
      {props.children}
    </Text>
  </View>
);

const styles = StyleSheet.create({
  container: {
    marginTop: 50,
    alignItems: 'center',
    flex: 1
  },
  flexContainer: {
    backgroundColor: '#ededed',
    width: 120,
    height: 180,
    borderWidth: 1,
    margin: 10
  },
  example: {
    height: 25,
    marginBottom: 5,
    backgroundColor: '#666666'
  }
});

```

Sets alignSelf explicitly to stretch

Sets alignSelf to auto, which picks up the parent container's value of stretch

Sets alignSelf to center

Sets alignSelf to flex-start

Sets alignSelf to flex-end

The default value for alignSelf is auto.

Uses the align property to set the Example component's alignItems style

5.3.6 Preventing clipped items with flexWrap

You learned earlier in this section that the `flexDirection` property takes two values: `column` (the default) and `row`. `column` lays out items vertically, and `row` lays out items horizontally. What you haven't seen is a situation in which items flow off the screen because they don't fit.

`flexWrap` takes two values: `nowrap` and `wrap`. The default value is `nowrap`, meaning items will flow off the screen if they don't fit. The items are clipped, and the user can't see them. To work around this problem, use the `wrap` value.

In figure 5.21, the first example uses `nowrap`, and the squares flow off the screen. The row of squares is chopped off at the right edge. The second example uses `wrap`, and the squares wrap around and start a new row. Listing 5.10 shows the code.

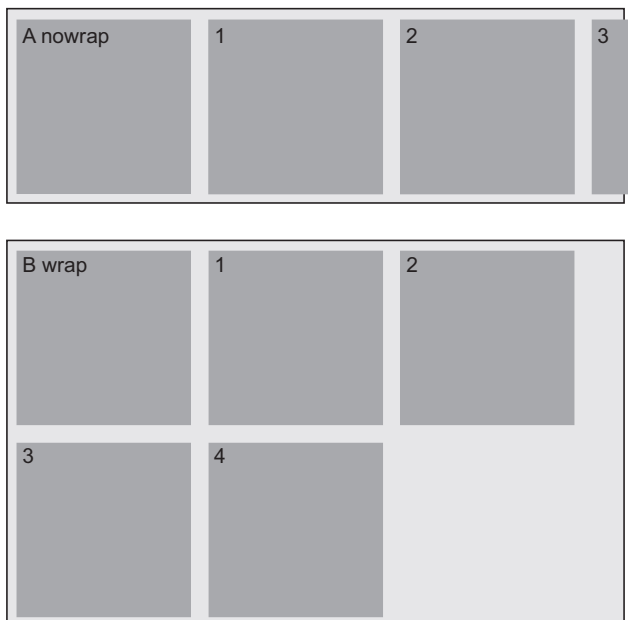


Figure 5.21 An example of two overflowing containers: one with `flexWrap` set to `nowrap` and the other with `flexWrap` set to `wrap`

Listing 5.10 Example of how `flexWrap` values affect layout

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

export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <NoWrapContainer>
          <Example>A nowrap</Example>
```

flexWrap is set to nowrap: the squares overflow off the screen.

```

        <Example>1</Example>
        <Example>2</Example>
        <Example>3</Example>
        <Example>4</Example>
      </NoWrapContainer>
      <WrapContainer>
        <Example>B wrap</Example>
        <Example>1</Example>
        <Example>2</Example>
        <Example>3</Example>
        <Example>4</Example>
      </WrapContainer>
    </View>
  );
}
}

const NoWrapContainer = (props) => (
  <View style={ [styles.noWrapContainer, props.style] }>
    {props.children}
  </View>
);

const WrapContainer = (props) => (
  <View style={ [styles.wrapContainer, props.style] }>
    {props.children}
  </View>
);

const Example = (props) => (
  <View style={ [styles.example, props.style] }>
    <Text>
      {props.children}
    </Text>
  </View>
);

const styles = StyleSheet.create({
  container: {
    marginTop: 150,
    flex: 1
  },
  noWrapContainer: {
    backgroundColor: '#ededed',
    flexDirection: 'row',
    flexWrap: 'nowrap',
    borderWidth: 1,
    margin: 10
  },
  wrapContainer: {
    backgroundColor: '#ededed',
    flexDirection: 'row',
    flexWrap: 'wrap',
    borderWidth: 1,
    margin: 10
  },
});

```

flexWrap is set to wrap:
the row of squares wraps
around to start a new line.

Uses the noWrapContainer
style for the first example

Uses the wrapContainer
style for the second example

Sets flexDirection to row
and flexWrap to nowrap

Sets flexDirection to row
and flexWrap to wrap

```

example: {
  width: 100,
  height: 100,
  margin: 5,
  backgroundColor: '#666666'
},
});

```

It's easy to see which behavior is preferable when laying out tiles, but you may come across a situation in which `nowrap` will serve you better. Either way, you should now have a clear understanding of flexbox and the many ways it can help you build responsive layouts in React Native.

Summary

- When sizing items for display, iOS uses points and Android uses density-independent pixels. The systems of measurement are different but should have little impact on development unless you need pixel-perfect graphics.
- Some styles are only available on one platform or another. `ShadowPropTypesIOS` is only available on iOS, and `elevation` is only recognized on Android.
- Components can be moved in the x and y directions using the `translateX` and `translateY` transforms.
- Components can be rotated about the x-, y-, and z-axes using `rotateX`, `rotateY`, and `rotateZ`. The point of rotation is the original location of the object before any transforms have been applied.
- Components can be scaled in the x and y directions to make the components grow or shrink.
- Components can also be skewed in the x and y directions.
- Several transformations can be applied at the same time, but the order in which they're specified matters. Rotating a component changes the orientation of the component for subsequent transformations.
- The `flexDirection` property defines the primary axis, the default being `column` (y-axis).
- The `justifyContent` property defines how items should be laid out along the primary axis.
- The `alignItems` property defines how items should be laid out along the secondary axis.
- The `alignSelf` property can be used to override the `alignItems` property specified by a parent container.
- The `flexWrap` property tells flexbox how to handle items that would typically overflow off the screen.

6

Navigation

This chapter covers

- Navigation in React Native vs. the web
- Navigating using tabs, stacks, and drawers
- Managing nested navigators
- Passing data and methods between routes

One of the core pieces of functionality in any mobile application is navigation. Before building an application, I recommend that you spend some time strategizing how you want the app to handle navigation and routing. This chapter covers the three main types of navigation typical to mobile applications: tab-based, stack-based, and drawer-based navigation.

Tab-based navigation typically has tabs either at the top or bottom of the screen; pressing a tab takes you to the screen that correlates with the tab. Many popular apps like Twitter, Instagram, and Facebook implement this type of navigation on their main screens.

Stack-based navigation transitions from one screen to another, replacing the current screen, and usually implements some sort of animated transition. You can then go backward or continue moving forward in the stack. You can think of stack-based navigation like an array of components: pushing a new component into the array

takes you to the screen of the new component. To go back, you pop the last screen from the stack and are navigated to the previous screen. Most navigation libraries handle this popping and pushing for you.

Drawer-based navigation is typically a side menu that pops out from either the left or right side of the screen and shows a list of options. When you press an option, the drawer closes, and you're taken to the new screen.

The React Native framework doesn't include a navigation library. When building navigation in a React Native app, you have to go with a third-party navigation library. A few good navigation libraries are available, but in this chapter, I use React Navigation as the navigation library of choice to build out the demo app. The React Navigation library is recommended by the React Native team and is maintained by many people in the React and React Native community.

React Navigation is a JavaScript-based navigation implementation. All the transitions and controls are handled by JavaScript. Some teams prefer a native solution for many reasons: for instance, they may be adding React Native to an existing native app and want navigation to be consistent throughout the app. If you're interested in a native navigation solution, check out React Native Navigation, an open source React Native navigation library built and maintained by the engineers at Wix.

6.1 *React Native navigation vs. web navigation*

Because the paradigm of navigation on the web is much different than that of React Native, navigation is a stumbling block for many developers new to React Native. On the web, we're used to working with URLs. There are many ways to navigate to a new route, depending on the framework or environment, but typically you want to send the user to a new URL and maybe add some URL parameters if needed.

In React Native, routes are based around components. You load or show a component using the navigator you're working with. Depending on whether it's tab based, stack based, drawer based, or a combination of these, the routing will also differ. We'll walk through all this when you build the demo app in the next section.

You also need to keep up with the data and state throughout the routes and possibly access methods defined elsewhere in the app, so having a strategy around data and method sharing is important. You can manage data and methods either at the top level, where the navigation is defined, or using a state-management library such as Redux or MobX. In the example, you'll manage data and methods in the class at the top level of the app.

6.2 *Building a navigation-based app*

In this chapter, you'll learn how to implement navigation by building out an app that uses both tab-based and stack-based navigation. The app you'll create is called Cities; it's shown in figure 6.1. It's a travel app that lets you keep up with all the cities you visit or want to visit. You can also add locations in each city you want to visit.

The main navigation is tab based, and one of the tabs includes a stack-based navigation. The left tab shows the list of cities you've created, and the right tab contains a form to create new cities. On the left tab, you can press an individual city to view it as well as view and create locations in the city.

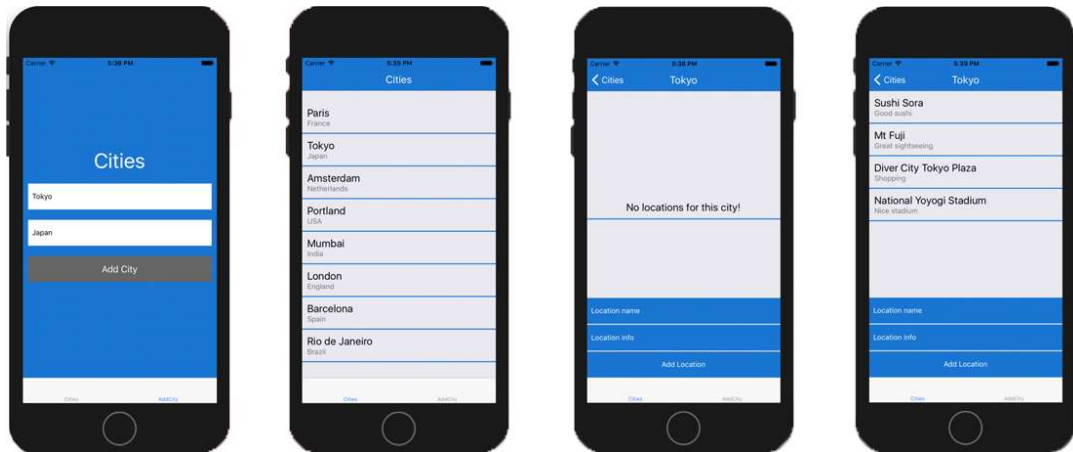


Figure 6.1 Completed Cities app with screens for adding a city, listing cities, viewing city details, and viewing locations within the city

To get started, create a new React Native application. In your terminal, navigate to an empty directory, and install the new React Native application using the React Native CLI:

```
react-native init CitiesApp
```

Next, navigate into the new directory, and install two dependencies: React Navigation and uuid. React Navigation is the navigation library, and uuid will be used to create unique IDs for cities in order to identify them uniquely:

```
cd CitiesApp
npm install react-navigation uuid
```

Now, let's get to work creating components! Create a new main directory called `src` in the root of the application. This directory will hold most of the new code for the app. In this new directory, add three main subdirectories: `Cities`, `AddCity`, and `components`.

Because the main navigation is tab based, you'll separate the main application into two main components (`Cities` and `AddCity`), each having its own tab. The `AddCity` folder will only contain a single component, `AddCity.js`. The `Cities` folder will contain two components: `Cities.js` to view the list of cities, and `City.js` to view an individual city. The `components` folder will hold any reusable components; in this case, it will hold a single component.

You'll also have `src/index.js` and `src/theme.js` files. `src/index.js` will hold all the navigation configuration, and `theme.js` will be where you keep themeable configuration—in this case, a primary color configuration. Figure 6.2 shows the project's complete folder structure.

Now that you've created the folder structure and installed the necessary dependencies, let's write some code. The first file you'll work with is `src/theme.js`. Here, you'll set the primary color and make it exportable for use in the app. The theme color I've chosen for the app is blue, but feel free to use any color you want; the app will work the same if you change the color value in this file.

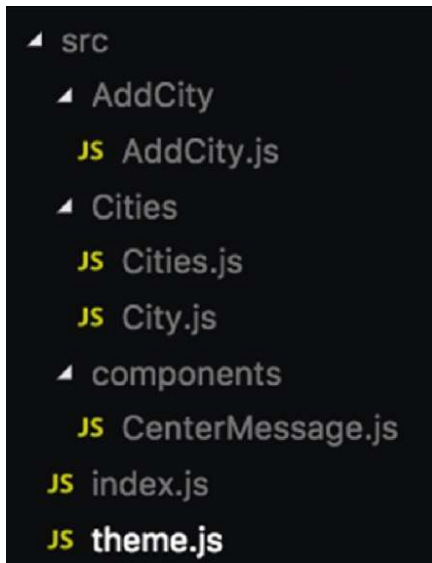


Figure 6.2 The complete src folder structure

Listing 6.1 Creating a theme file with a primary color

```

const colors = {
  primary: '#1976D2'
}

export {
  colors
}
  
```

You can import this primary color throughout the application if you wish, and change it in one place if you choose to do so.

Next, edit `src/index.js` to create the main navigation configuration. You'll create both navigation instances here: the tab-based navigation and the stack-based navigation.

Listing 6.2 Creating the navigation configuration

```

import React from 'react'

import Cities from './Cities/Cities'
import City from './Cities/City'
import AddCity from './AddCity/AddCity'

import { colors } from './theme'

import { createBottomTabNavigator,
  createStackNavigator } from 'react-navigation'

const options = {
  navigationOptions: {
    headerStyle: {
      backgroundColor: colors.primary
    },
  },
}
  
```

Imports the three components to have in the scope of the file

Imports colors from the theme

Imports the two navigators to access from React Navigation

Creates an options object to hold configuration for the stack navigator

```

    headerTintColor: '#fff'
  }
}

const CitiesNav = createStackNavigator({
  Cities: { screen: Cities },
  City: { screen: City }
}, options)

const Tabs = createBottomTabNavigator({
  Cities: { screen: CitiesNav },
  AddCity: { screen: AddCity }
})

export default Tabs

```

← Creates the first navigation instance

← Creates the tab navigator using the CitiesNav stack navigator for one tab and the AddCity component for the second tab

When you create the options object, the stack navigator automatically places a header at the top of each route. The header is usually where you'll have the title of the current route as well as buttons like a Back button. The options object also defines the background color and the tint color of the header.

For the first navigation instance, `createStackNavigator` takes two arguments: the route configuration and any configuration regarding things like styling to apply to the navigation. You pass in two routes as the first argument, and the options object as the second argument.

Next, update `App.js` to include the new navigation and render it as the main entry point. In addition to rendering the navigation component, `App.js` will contain and control any methods and data to be made available to the application.

Listing 6.3 Updating `App.js` to use the navigation configuration

```

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

import Tabs from './src'

export default class App extends Component {
  state = {
    cities: []
  }

  addCity = (city) => {
    const cities = this.state.cities
    cities.push(city)
    this.setState({ cities })
  }

  addLocation = (location, city) => {
    const index = this.state.cities.findIndex(item => {
      return item.id === city.id
    })
  }
}

```

← Imports the navigation from `src/index.js`

← Creates an initial state of cities, an empty array

← Adds a new city to the existing list of cities stored in the state

← Adds a location to the array of locations in a chosen city

```

const chosenCity = this.state.cities[index]
chosenCity.locations.push(location)
const cities = [
  ...this.state.cities.slice(0, index),
  chosenCity,
  ...this.state.cities.slice(index + 1)
]
this.setState({
  cities
})
}
render() {
  return (
    <Tabs
      screenProps={{
        cities: this.state.cities,
        addCity: this.addCity,
        addLocation: this.addLocation
      }}
    />
  )
}

```

Returns the Tabs component and passes in a screenProps object containing the cities array, the addCity method, and the addLocation method

App.js has three main pieces of functionality. It creates the initial state of the app: an empty array called `cities`. Each city will be an object and will have a name, country, ID, and array of locations. The `addCity` method lets you add new cities to the `cities` array stored in the state. The `addLocation` method identifies the city you want to add a location to, updates the city, and resets the state with the new data.

React Navigation has a way to pass these methods and the state down to all the routes being used by the navigator. To do this, pass a prop called `screenProps` containing whatever you want access to. Then, from within any route, `this.props.screenProps` gives access to the data or methods.

Next, you'll create a reusable component called `CenterMessage`, which is used in `Cities.js` and `City.js` (`src/components/CenterMessage.js`). It displays a message when the array is empty. For example, when the app first starts, it won't have any cities to list; you can display a message as shown in figure 6.3, instead of just showing a blank screen.

Listing 6.4 CenterMessage component

```

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

import { colors } from '../theme'

const CenterMessage = ({ message }) => (
  <View style={styles.emptyContainer}>

```

```

    <Text style={styles.message}>{message}</Text>
  </View>
)

const styles = StyleSheet.create({
  emptyContainer: {
    padding: 10,
    borderBottomWidth: 2,
    borderBottomColor: colors.primary
  },
  message: {
    alignSelf: 'center',
    fontSize: 20
  }
})

export default CenterMessage

```

This component is straightforward. It's a stateless component that receives only a message as a prop and displays the message along with some styling.

Next, in `src/AddCity/AddCity.js`, create the `AddCity` component that will allow you to add new cities to the `cities` array (see figure 6.4). This component will contain a

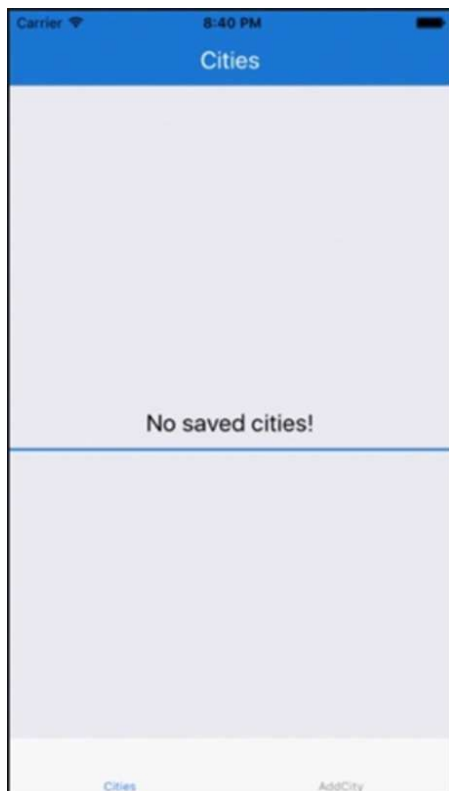


Figure 6.3 The reusable `CenterMessage` component displays a message centered within the display.

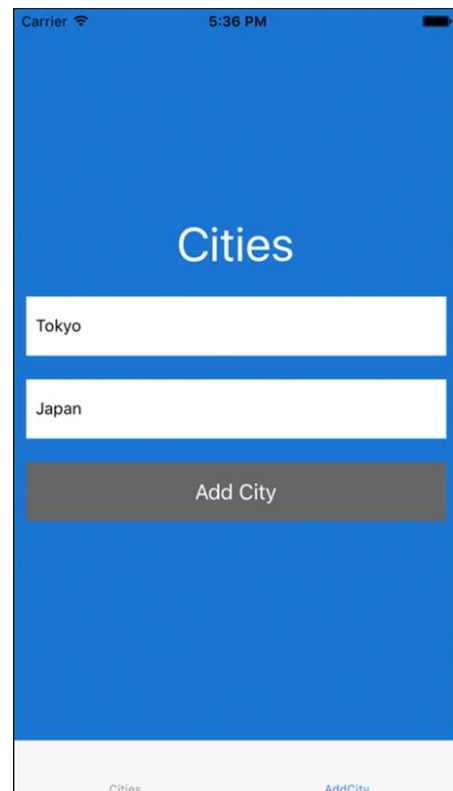


Figure 6.4 `AddCity` tab allows the user to enter a new city name and the country name.

form with two text inputs: one to hold the city name and one to hold the country name. In addition, a button will call the `addCity` method from `App.js`.

Listing 6.5 AddCity tab (functionality)

```
import React from 'react'
import {
  View,
  Text,
  StyleSheet,
  TextInput,
  TouchableOpacity
} from 'react-native'

import uuidV4 from 'uuid/v4'

import { colors } from '../theme'

export default class AddCity extends React.Component {
  state = {
    city: '',
    country: ''
  }
  onChangeText = (key, value) => {
    this.setState({ [key]: value })
  }
  submit = () => {
    if (this.state.city === '' || this.state.country === '') {
      alert('please complete form')
    }
    const city = {
      city: this.state.city,
      country: this.state.country,
      id: uuidV4(),
      locations: []
    }
    this.props.screenProps.addCity(city)
    this.setState({
      city: '',
      country: ''
    }, () => {
      this.props.navigation.navigate('Cities')
    })
  }
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.heading}>Cities</Text>
        <TextInput
          placeholder='City name'
          onChangeText={val => this.onChangeText('city', val)}
          style={styles.input}
          value={this.state.city}
        />
      </View>
    )
  }
}
```

Holds much of the functionality for this component

The initial state holds a city name and a country name, both initially set as empty strings.

Updates the state with either the city or name value. This will be attached to the TextInput and will fire whenever the input value changes.


```

    />
    <TextInput
      placeholder='Country name'
      onChangeText={val => this.onChangeText('country', val)}
      style={styles.input}
      value={this.state.country}
    />
    <TouchableOpacity onPress={this.submit}>
      <View style={styles.button}>
        <Text style={styles.buttonText}>Add City</Text>
      </View>
    </TouchableOpacity>
  </View>
)
}
}

```

First, you check to make sure neither the city nor the country is an empty string. If either or both are empty, you return, because you don't want to store the data unless both fields are filled out. Next, you create an object to hold the city being adding to the cities array. Take the existing city and country values stored on the state, and add an ID value using the `uuidV4` method and an empty locations array. Call `this.props.screenProps.addCity`, passing in the new city. Next, reset the state to clear out any values stored in the state. Finally, navigate the user to the Cities tab to show them their list of cities with the new city added, by calling `this.props.navigation.navigate` and passing in the string of the route to navigate to—in this case, 'Cities'.

Every component that's a screen in a navigator automatically has access to two props: `screenProps` and `navigation`. In listing 6.3, when you created the navigation component, you passed in three `screenProps`. In the `submit` method, you called `this.props.screenProps.addCity`, accessing and invoking this `screenProps` method. You also access the `navigation` prop by calling `this.props.navigation.navigate`. `navigate` is what you use to navigate between routes in React Navigation.

Next, add the styles for this component. This code goes below the class definition in `src/AddCity/AddCity.js`.

Listing 6.6 AddCity tab (styling)

```

const styles = StyleSheet.create({
  button: {
    height: 50,
    backgroundColor: '#666',
    justifyContent: 'center',
    alignItems: 'center',
    margin: 10
  },
  buttonText: {
    color: 'white',
    fontSize: 18
  },
},

```

```

heading: {
  color: 'white',
  fontSize: 40,
  marginBottom: 10,
  alignSelf: 'center'
},
container: {
  backgroundColor: colors.primary,
  flex: 1,
  justifyContent: 'center'
},
input: {
  margin: 10,
  backgroundColor: 'white',
  paddingHorizontal: 8,
  height: 50
}
})

```

Now, create `src/Cities/Cities.js` to list all the cities the app is storing and allow the user to navigate to an individual city (see figure 6.5). The functionality is shown in the following listing, and the styling is in listing 6.8.

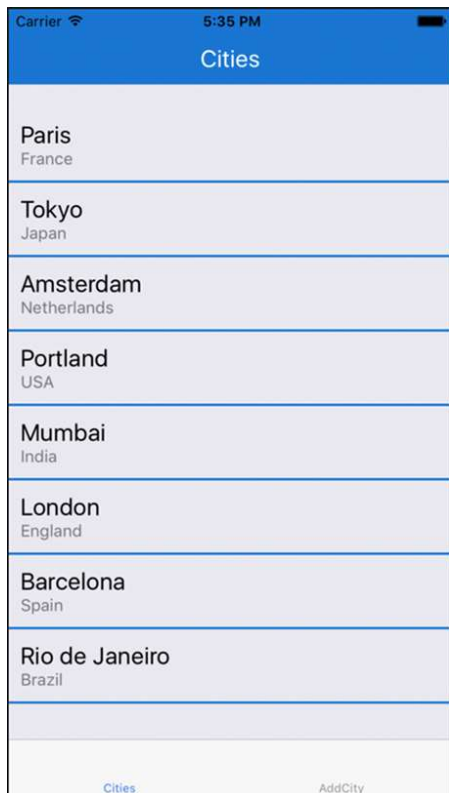


Figure 6.5 `Cities.js` displays a list of cities that have been added to the application.

Listing 6.7 Cities route (functionality)

```

import React from 'react'
import {
  View,
  Text,
  StyleSheet,
  TouchableWithoutFeedback,
  ScrollView
} from 'react-native'

import CenterMessage from '../components/CenterMessage'

import { colors } from '../theme'

export default class Cities extends React.Component {
  static navigationOptions = {
    title: 'Cities',
    headerTitleStyle: {
      color: 'white',
      fontSize: 20,
      fontWeight: '400'
    }
  }
  navigate = (item) => {
    this.props.navigation.navigate('City', { city: item })
  }
  render() {
    const { screenProps: { cities } } = this.props
    return (
      <ScrollView contentContainerStyle={!cities.length && { flex: 1 }}>
        <View style={!cities.length &&
          { justifyContent: 'center', flex: 1 }}>
          {
            !cities.length && <CenterMessage message='No saved cities!'/>
          }
        </View>
        <View>
          <ScrollView>
            cities.map((item, index) => (
              <TouchableWithoutFeedback
                onPress={() => this.navigate(item)} key={index} >
                <View style={styles.cityContainer}>
                  <Text style={styles.city}>{item.city}</Text>
                  <Text style={styles.country}>{item.country}</Text>
                </View>
              </TouchableWithoutFeedback>
            ))
          </ScrollView>
        </View>
      </ScrollView>
    )
  }
}

```

Imports the CenterMessage component created in listing 6.4

Declares a static navigationOptions property on the class and declares the configuration for this route

Accesses and destructures the cities array from the screenProps prop available in the component

Passes in the city as the second argument to this.props.navigation.navigate

Checks if the cities array is empty. If so, shows the user a message that there are no cities currently in the app.

Maps over all the cities in the array, displaying the city name and country name. Also attaches the navigate method to the TouchableWithoutFeedback component.

In this listing, you first import the CenterMessage component. React Navigation has a way to control certain options around the navigation within a route. To do so, you can declare a static navigationOptions property on the class and declare the configuration

for a route. In this case, you want to set a title and style the title, so give the configuration a `title` and `headerTitleStyle` property.

The `navigate` method calls `this.props.navigation.navigate` and passes in the route name as well as the city to access to in the `City` route. Pass in the city as the second argument; in the `City` route, you'll have access to this property in `props.navigation.state.params`. The `render` method accesses and destructures the `cities` array. It also includes logic to check whether the `cities` array is empty; if it is, show the user an appropriate message. You map over all the cities in the array, displaying the city name and country name. Attaching the `navigate` method to the `TouchableWithoutFeedback` component lets users navigate to the city by pressing anywhere on the city.

Listing 6.8 Cities route (styling)

```
const styles = StyleSheet.create({
  cityContainer: {
    padding: 10,
    borderBottomWidth: 2,
    borderBottomColor: colors.primary
  },
  city: {
    fontSize: 20,
  },
  country: {
    color: 'rgba(0, 0, 0, .5)'
  },
})
```

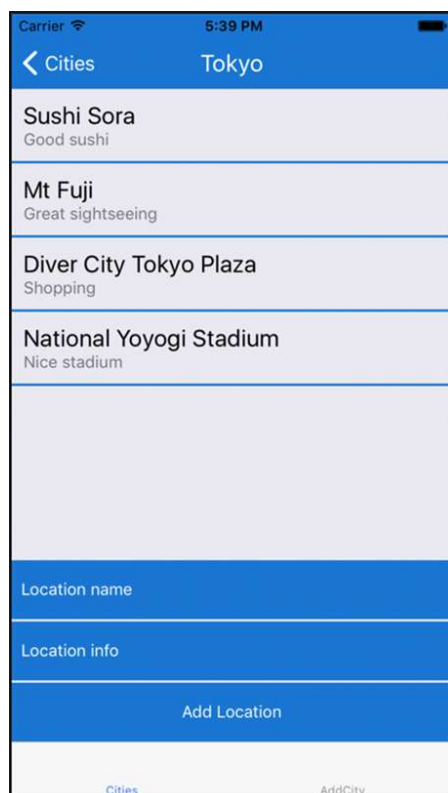


Figure 6.6 City.js shows locations within the city.

Next, create the City component (src/Cities/City.js) to hold the locations for each city as well as a form that lets users create a new location in a city; see figure 6.6. This component will access the cities from screenProps and will also use the addLocation method from screenProps to add a location to the city.

Listing 6.9 City route (functionality)

```
import React from 'react'
import {
  View,
  Text,
  StyleSheet,
  ScrollView,
  TouchableWithoutFeedback,
  TextInput,
  TouchableOpacity
} from 'react-native'

import CenterMessage from '../components/CenterMessage'
import { colors } from '../theme'

class City extends React.Component {
  static navigationOptions = (props) => {
    const { city } = props.navigation.state.params
    return {
      title: city.city,
      headerTitleStyle: {
        color: 'white',
        fontSize: 20,
        fontWeight: '400'
      }
    }
  }
  state = {
    name: '',
    info: ''
  }
  onChangeText = (key, value) => {
    this.setState({
      [key]: value
    })
  }
  addLocation = () => {
    if (this.state.name === '' || this.state.info === '') return
    const { city } = this.props.navigation.state.params
    const location = {
      name: this.state.name,
      info: this.state.info
    }
    this.props.screenProps.addLocation(location, city)
    this.setState({ name: '', info: '' })
  }
  render() {
    const { city } = this.props.navigation.state.params
    return (
      <View style={{ flex: 1 }}>
```

Creates the static navigationOptions property as in Cities.js

Deconstructs the city object, creates a location object, and calls this.props.screenProps.addLocation to add the location and reset the state

Deconstructs city

```

<ScrollView
  contentContainerStyle={
    [!city.locations.length && { flex: 1 }]
  }>
  <View style={[
    styles.locationsContainer,
    !city.locations.length && { flex: 1,
      justifyContent: 'center' }
  ]}>
    {
      !city.locations.length &&
      <CenterMessage message='No locations for this city!' />
    }
    {
      city.locations.map((location, index) => (
        <View key={index} style={styles.locationContainer}>
          <Text style={styles.locationName}>{location.name}</Text>
          <Text style={styles.locationInfo}>{location.info}</Text>
        </View>
      ))
    }
  </View>
</ScrollView>
<TextInput
  onChangeText={val => this.onChangeText('name', val)}
  placeholder='Location name'
  value={this.state.name}
  style={styles.input}
  placeholderTextColor='white'
/>
<TextInput
  onChangeText={val => this.onChangeText('info', val)}
  placeholder='Location info'
  value={this.state.info}
  style={[styles.input, styles.input2]}
  placeholderTextColor='white'
/>
<View style={styles.buttonContainer}>
  <TouchableOpacity onPress={this.addLocation}>
    <View style={styles.button}>
      <Text style={styles.buttonText}>Add Location</Text>
    </View>
  </TouchableOpacity>
</View>
</View>
)
}
}

```

Creates the form

Maps over the cities in the cities array, and returns a component that displays the city's name and information

This code first creates the `navigationOptions` property. You use a callback function to return an object instead of just declaring an object, because you need access to the props in order to have access to the city information passed down by the navigation. You need to know the city title for use as the route title instead of a hard-coded string.

The `addLocation` method destructures the city object available from `this.props.navigation.state.params` for use later in the function. You then create a location

object holding the location name and info. Calling `this.props.screenProps.addLocation` adds the location to the city you're currently viewing and then resets the state. Again, destructure `city` from the navigation state. You need `city` in order to map over the locations in the city and also to use as an argument when creating a new location, to identify the city you're referencing. Finally, you map over the cities, returning a component that displays both the city name and city information, and create the form with two text inputs and a button.

6.3 Persisting data

You're finished and should be able to run the app. Play around with the app, add cities and locations, and then refresh it. Notice that all the cities disappear when you refresh. This is because you're only storing the data in memory. Let's use `AsyncStorage` to persist the state, so if the user closes or refreshes the app, their data remains available.

To do so, you'll work in the `App` component in `App.js` and do the following:

- Store the cities array in `AsyncStorage` every time a new city is added.
- Store the cities array in `AsyncStorage` every time a new location is added to a city.
- When the user opens the app, check `AsyncStorage` to see whether any cities are stored there. If so, update the state with those cities.
- `AsyncStorage` only accepts strings for stored values. So, when storing a value, call `JSON.stringify` on the value if it isn't already a string, and `JSON.parse` if you want to parse the stored value before using it.

Open `App.js` and make the changes:

- 1 Import `AsyncStorage`, and create a key variable.

```
import {
  #omitting previous imports
  AsyncStorage
} from 'react-native';

const key = 'state'

export default class App extends Component {
  #omitting class definition
```

- 2 Create a `componentDidMount` function that will check for `AsyncStorage` and get any item stored there with the key value you set:

```
async componentDidMount() {
  try {
    let cities = await AsyncStorage.getItem(key)
    cities = JSON.parse(cities)
    this.setState({ cities })
  } catch (e) {
    console.log('error from AsyncStorage: ', e)
  }
}
```

- 3 In the `addCity` method, store the `cities` array in `AsyncStorage` after the new `cities` array has been created:

```
addCity = (city) => {
  const cities = this.state.cities
  cities.push(city)
  this.setState({ cities })
  AsyncStorage.setItem(key, JSON.stringify(cities))
    .then(() => console.log('storage updated!'))
    .catch(e => console.log('e: ', e))
}
```

- 4 Update the `addLocation` method to store the city array after `setState` has been called.

```
addLocation = (location, city) => {
  #previous code omitted
  this.setState({
    cities
  }, () => {
    AsyncStorage.setItem(key, JSON.stringify(cities))
      .then(() => console.log('storage updated!'))
      .catch(e => console.log('e: ', e))
  })
}
```

Now, when the user opens the app after closing it, their data will still be available.

6.4 **Using `DrawerNavigator` to create drawer-based navigation**

We've gone over how to create stack-based and tab-based navigation. Let's look at the API for creating drawer-based navigation.

The drawer navigator has an API very similar to that of the stack and tab navigators. You'll use the `createDrawerNavigator` function from `React Navigation` to create a drawer-based navigation. First define the routes to use:

```
import Page1 from './routeToPage1'
import Page2 from './routeToPage2'
```

Next, define the screens you want used in the navigator:

```
const screens = {
  Page1: { screen: Page1 },
  Page2: { screen: Page2 }
}
```

Now you can define the navigator using the screen configuration and use it in the app:

```
const DrawerNav = createDrawerNavigator(screens)

// somewhere in our app

<DrawerNav />
```


Summary

- Before building an application, spend time strategizing how you want it to handle navigation and routing.
- Many navigation libraries are available for React Native, but the two most recommended are React Navigation and React Native Navigation. React Navigation is a JavaScript-based navigation library, and React Native Navigation is a native implementation.
- There are three main types of navigators:
 - Tab-based navigation typically has tabs either at the top or bottom of the screen. When you press a tab, you're taken to the screen that correlates with that tab. For example, `createBottomTabNavigator` creates tabs at the bottom of the screen.
 - Stack-based navigation transitions from one screen to another, replacing the current screen. You can go backward or continue moving forward in the stack. Stack-based navigation usually implements some sort of animated transition. You create stack-based navigation using the `createStackNavigator` function.
 - Drawer-based navigation is typically a menu that pops out from either the left or right side of the screen and shows a list of options. When you press an option, the drawer closes and you're taken to the new screen. You create drawer-based navigation using the `createDrawerNavigator` function.
- Depending on which kind of navigation you use—tab-based, stack-based, drawer-based, or a combination of these—the routing will also differ. Every route or screen managed by the React Navigation library has a `navigation` prop you can use to control the navigation state.
- Use `AsyncStorage` to persist state so if the user closes or refreshes the app, their data is still available.

7 Animations

This chapter covers

- Creating basic animations using `Animated.timing`
- Using interpolation with animated values
- Creating animations and in parallel
- Staggering animations using `Animated.stagger`
- Using the native driver to offload animations to the native UI thread

One of the great things about React Native is the ability to easily create animations using the Animated API. This is one of the more stable and easy to use React Native APIs, and it's one of the few places in the React Native ecosystem where, unlike areas such as navigation and state management, there's almost 100% agreement on how a problem should be solved.

Animations are usually used to enhance the UI of an application and bring more life to the existing design. Sometimes, the difference between an average and above-average user experience can be attributed to using the right animations at the right time, thus setting an app apart from other, similar apps.

Real-world use cases that we cover in this chapter include the following:

- Expanding user inputs that animate when focused
- Animated welcome screens that have more life than a basic static welcome screen
- A custom animated loading indicator

In this chapter, we dive deeply into how to create animations. We'll cover everything you need to know to take full advantage of the Animated API.

7.1 Introducing the Animated API

The Animated API ships with React Native, so to use it, all you have to do is import it as you would any other React Native API or component. When creating an animation, you always need to do the following four things:

- 1 Import Animated from React Native.
- 2 Create an animatable value using the Animated API.
- 3 Attach the value to a component as a style.
- 4 Animate the animatable value using a function.

Out of the box, four types of animatable components ship with the Animated API:

- View
- ScrollView
- Text
- Image

The examples in this chapter work exactly the same across any of these components. In section 7.5, we also cover how to create a custom animated component using any element or component with `createAnimatedComponent`.

Let's take a quick look at what a basic animation might look like using Animated. In the example, you'll animate the top margin of a box (see figure 7.1).

Listing 7.1 Using Animated and updating the `marginTop` property

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

export default class RNAnimations extends Component {
  marginTop = new Animated.Value(20);
  animate = () => {
    Animated.timing(
      this.marginTop,
      {
        toValue: 200,
        duration: 500,
```

Imports the Animated API from React Native

Creates a class property called `marginTop` and assigns it to an animated value, passing in the starting value (20 in this case)

Creates a function that will animate the value

```

    }
    ).start();
  }
  render() {
    return (
      <View style={styles.container}>
        <Button
          title='Animate Box'
          onPress={this.animate}
        />
        <Animated.View
          style={[styles.box, { marginTop: this.marginTop } ]} />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 10,
    paddingTop: 50,
  },
  box: {
    width: 150,
    height: 150,
    backgroundColor: 'red'
  }
});

```

Attach the animate method to an onPress handler so you can call it

Use the Animated.View component instead of the regular View component



Figure 7.1 Animating the top margin of a square box using Animated

This example uses the `timing` function to animate a value. The `timing` function takes two arguments: a starting value and a configuration object. The configuration object is passed a `toValue` to set the value the animation should animate to, and a duration in milliseconds to set the length of the animation.

Rather than a `View` component, you use an `Animated.View`. `Animated` has four components that can be animated out of the box: `View`, `Image`, `ScrollView`, and `Text`. In the styling of the `Animated.View`, you pass in an array of styles consisting of a base style (`styles.box`) and an animated style (`marginTop`).

Now that you've created a basic animated component, you'll create a few more animations using real-world use cases that may come in handy.

7.2 Animating a form input to expand on focus

In this example, you'll create a basic form input that expands when the user focuses it, and contracts when the input is blurred. This is a popular UI pattern.

Along with the props that you've used with the `TextInput` component so far in this book, such as `value`, `placeholder`, and `onChangeText`, you can also use `onFocus` and `onBlur` to call functions when the inputs are focused and blurred. That's how you'll achieve this animation (shown in figure 7.2).

Listing 7.2 Animating a `TextInput` to expand when the input is focused

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

export default class RNAnimations extends Component {
  animatedWidth = new Animated.Value(200);

  animate = (value) => {
    Animated.timing(
      this.animatedWidth,
      {
        toValue: value,
        duration: 750,
      }
    ).start()
  }

  render() {
    return (
      <View style={styles.container}>
        <Animated.View style={{ width: this.animatedWidth }}>
          <TextInput
            style={styles.input}
            onBlur={() => this.animate(200)}
            onFocus={() => this.animate(325)}
          />
        </Animated.View>
      </View>
    );
  }
}
```

Creates an initial value for the animation, calling it `animatedWidth`

Creates an animate function that will animate the animated value of `animatedWidth`

Attach the `animatedWidth` value to the style of the container `View` holding the Input component.

Attach the `animate` method to the `onBlur` and `onFocus` handlers, passing in the desired width for when each event is fired.

```

        ref={input => this.input = input}
      />
    </Animated.View>
    <Button
      title='Submit'
      onPress={() => this.input.blur()}
    />
  </View>
);
}
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    padding: 10,
    paddingTop: 50,
  },
  input: {
    height: 50,
    marginHorizontal: 15,
    backgroundColor: '#ededed',
    marginTop: 10,
    paddingHorizontal: 9,
  },
});

```

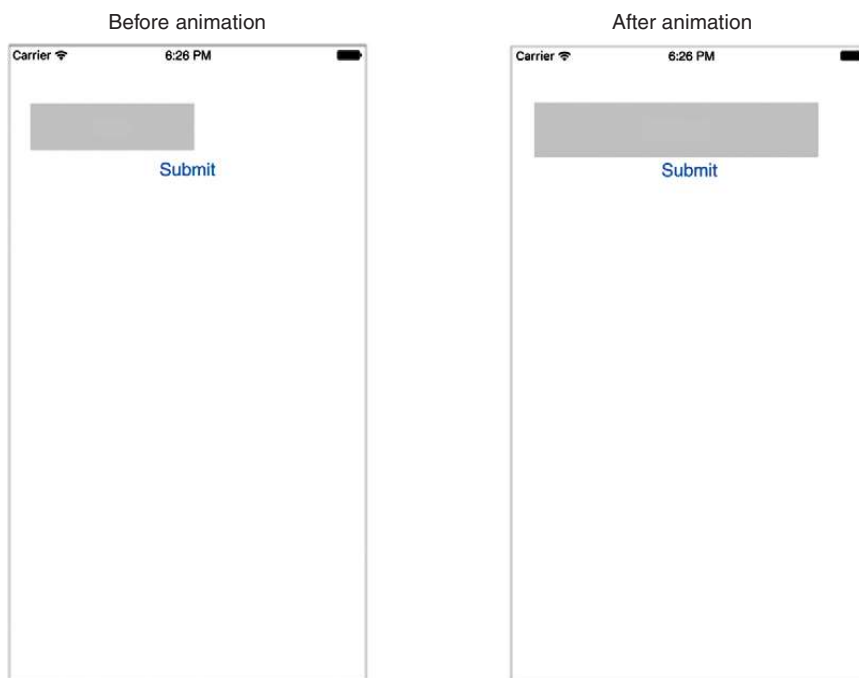


Figure 7.2 Animating a `TextInput` component when the input is focused

7.3 Creating a custom loading animation using interpolation

Many times, you need to create animations that are infinite loops, such as loading indicators and activity indicators. One easy way to create such animations is to use the `Animated.loop` function. In this section, you use `Animated.loop` along with the `Easing` module to create a loading indicator, spinning an image in an infinite loop!

So far, we've only looked at calling an animation using `Animated.timing`. In this example, you want the animation to run continuously without stopping. To do this, you'll use a new static method called `loop`. `Animated.loop` runs a given animation continuously: each time it reaches the end, it resets to the beginning and starts again.

You'll also deal with styling a little differently than in the past. In listings 7.1 and 7.2, you used the animated value directly in the `style` prop of the component. In subsequent examples, you'll store these animation values in variables and interpolate the values before using the new interpolated variables in the `style` prop. Because you're creating a spinning effect, you'll use strings instead of numbers: for example, you'll reference a value such as `360deg` for `style`.

`Animated` has a class method called `interpolate` that you can use to manipulate animated values, changing them into other values that you can also use. The `interpolate` method takes a configuration object with two keys: `inputRange` (array) and `outputRange` (also an array). `inputRange` is the original animated values you work with in a class, and `outputRange` specifies the values the original values should be changed to.

Finally, you'll change the easing value of the animation. `Easing` basically allows you to control the animation's motion. In this example, you want a smooth, even motion for the spin effect, so you'll use a linear easing function.

React Native has a built-in way to implement common easing functions. Just as you've imported other APIs and components, you can import the `Easing` module and use it along with `Animated`. `Easing` can be configured in the configuration object where you set values like `toValue` and `duration`, in the second argument of `Animated.timing`. Let's look at an example with an animated value called `animatedMargin`. Setting `animatedMargin` to 0 and animating the value to 200 would normally achieve the easing effect by directly animating the value between 0 and 200 in the timing function. Using interpolation, you can instead animate a value between 0 and 1 in the timing function and later interpolate the value by using the `Animated.interpolate` class method, saving the value into another variable, and then referencing *that* variable in the `style`, usually in the render method:

```
const marginTop = animatedMargin.interpolate({
  inputRange: [0, 1],
  outputRange: [0, 200],
});
```

Now, use interpolation to create the loading indicator. You'll show the indicator when the application loads; in `componentDidMount`, you'll call `setTimeout`, which cancels the loading state after 2,000 milliseconds (see figure 7.3). The icon used here is located at <https://github.com/dabit3/react-native-in-action/blob/chapter7/assets/35633-200.png>; feel free to use it or any other image you want.

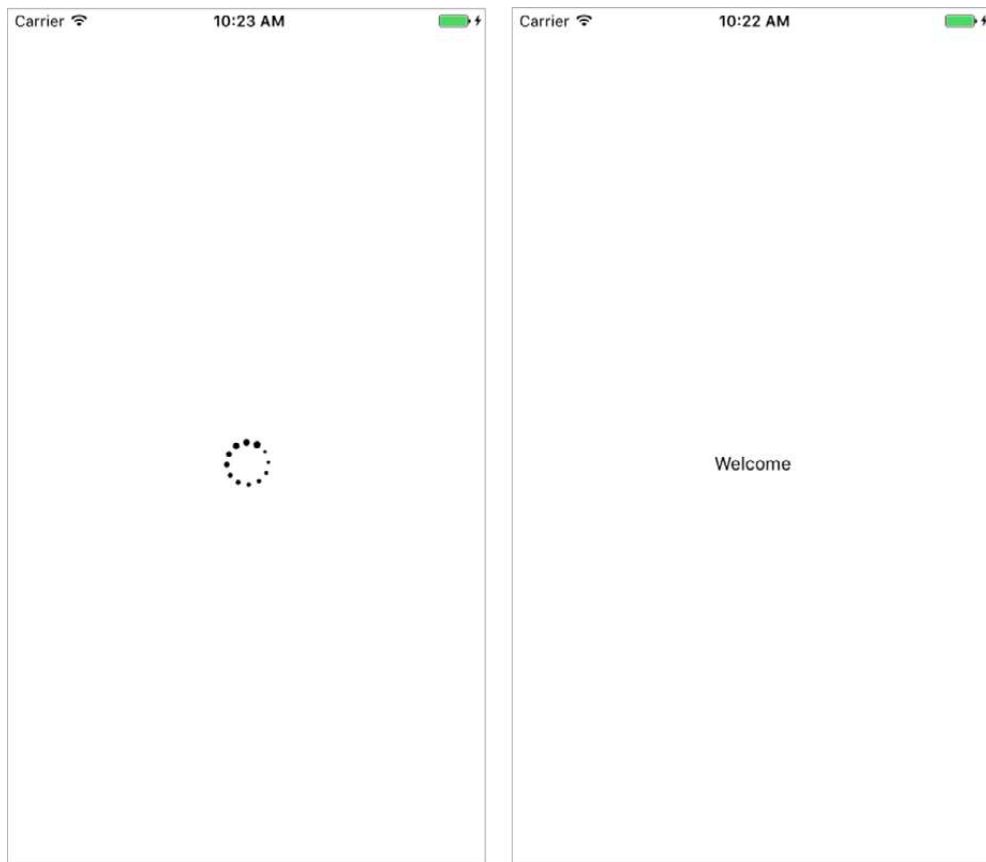


Figure 7.3 Creating a spinning loading indicator using interpolation and an animated loop

Listing 7.3 Creating an infinitely spinning loading animation

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

export default class RNAnimations extends Component {
  state = {
    loading: true,
  }
  componentDidMount() {
    this.animate();
  }
}
```

← Initializes the state with a Boolean loading value of true

← Triggers the animation by calling this. animate, and invokes a setTimeout function to set loading to false in the state after 2 seconds


```

    setTimeout(() => this.setState({ loading: false })), 2000)
  }
  animatedRotation = new Animated.Value(0);
  animate = () => {
    Animated.loop(
      Animated.timing(
        this.animatedRotation,
        {
          toValue: 1,
          duration: 1800,
          easing: Easing.linear,
        }
      )
    ).start()
  }
  render() {
    const rotation = this.animatedRotation.interpolate({
      inputRange: [0, 1],
      outputRange: ['0deg', '360deg'],
    });
    const { loading } = this.state;
    return (
      <View style={styles.container}>
        {
          loading ? (
            <Animated.Image
              source={require('./pathtoyourimage.png')}
              style={{ width: 40, height: 40, transform: [{ rotate: rotation }] }}
            />
          ) : (
            <Text>Welcome</Text>
          )
        }
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    padding: 10,
    paddingTop: 50,
  },
  input: {
    height: 50,
    marginHorizontal: 15,
    backgroundColor: '#ededed',
    marginTop: 10,
    paddingHorizontal: 9,
  },
});

```

Passes in the animation's beginning and end values (0 and 1)

Sets an initial animatedRotation value of 0

Creates an animate class method that passes Animated.timing into a call to Animated.loop

Uses the animatedRotation value to create a new rotation value using the interpolate method

Passes in the values for inputRange to map to

Checks whether loading is true, and responds accordingly

The `animate` class method passes `Animated.timing` into a call to `Animated.loop`. In the configuration, you set `toValue` to 1, `duration` to 1800, and `easing` to `Easing.linear`, to create a smooth spinning movement.

The `animatedRotation` value creates a new value called `rotation`, using the `interpolate` method. `inputRange` gives the animation's beginning and end values, and `outputRange` gives the values `inputRange` should map to: a beginning value of 0 degrees and a final value of 360 degrees, creating a full 360-degree rotation.

In the return statement, first check to see whether `loading` is `true`. If it is, show the animated loading indicator (update this path to that of the image in your application); if it's `false`, show a welcome message. Attach the `rotation` variable to the `transform rotate` value in the styling of `Animated.Image`.

7.4 Creating multiple parallel animations

Sometimes you need to create multiple animations at once and have them run simultaneously. The `Animated` library has a class method called `parallel` you can use to do this. `parallel` starts an array of animations at the same time.

For example, to make a welcome screen with two messages and a button all appear to move into the screen at once, you could create three separate animations and

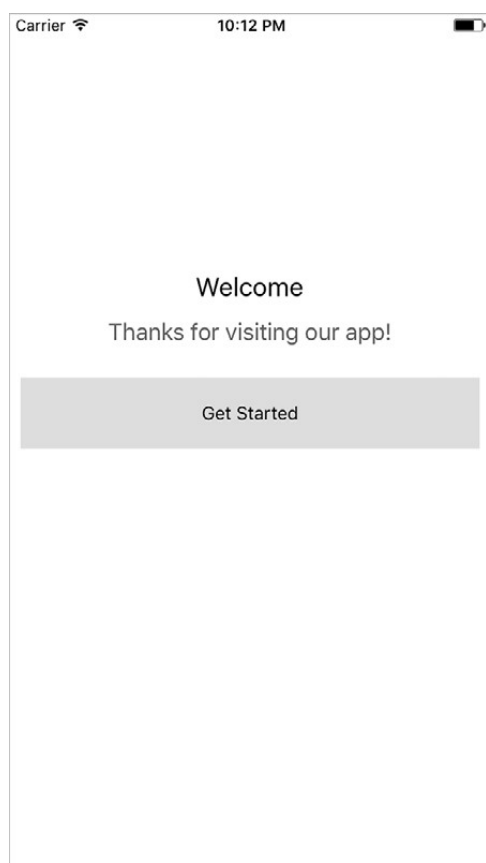


Figure 7.4 Welcome screen using parallel animations (shown after the animations are complete)

call `.start()` on each of them. But a more efficient way would be to use the `Animated.parallel` function and pass in the array of animations to run at the same time.

In this example, you'll create a welcome screen that animates in two messages and a button when the component mounts (see figure 7.4). Because you're using `Animated.parallel`, all three animations will begin at exactly the same time. You'll add a `delay` property to the configuration to control the start time of two of the animations.

Listing 7.4 Creating an animated welcome screen

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

export default class RNAnimations extends Component {
  animatedTitle = new Animated.Value(-200);
  animatedSubtitle = new Animated.Value(600);
  animatedButton = new Animated.Value(800);

  componentDidMount() {
    this.animate();
  }

  animate = () => {
    Animated.parallel([
      Animated.timing(
        this.animatedTitle,
        {
          toValue: 200,
          duration: 800,
        }
      ),
      Animated.timing(
        this.animatedSubtitle,
        {
          toValue: 0,
          duration: 1400,
          delay: 800,
        }
      ),
      Animated.timing(
        this.animatedButton,
        {
          toValue: 0,
          duration: 1000,
          delay: 2200,
        }
      )
    ]).start();
  }
}
```

When you create the class, also create three new animated values.

Calls the `animate()` method on `componentDidMount`

Calls `Animated.parallel` and passes in three `Animated.timing` animations to trigger all three animations to start at once

Calls `Animated.parallel` and passes in three `Animated.timing` animations to trigger all three animations to start at once

```

render() {
  return (
    <View style={styles.container}>
      <Animated.Text style={ [styles.title,
                             { marginTop: this.animatedTitle} ]}>
        Welcome
      </Animated.Text>
      <Animated.Text style={ [styles.subTitle,
                             { marginLeft: this.animatedSubtitle } ]}>
        Thanks for visiting our app!
      </Animated.Text>
      <Animated.View style={{ marginTop: this.animatedButton }}>
        <TouchableHighlight style={styles.button}>
          <Text>Get Started</Text>
        </TouchableHighlight>
      </Animated.View>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  title: {
    textAlign: 'center',
    fontSize: 20,
    marginBottom: 12,
  },
  subTitle: {
    width: '100%',
    textAlign: 'center',
    fontSize: 18,
    opacity: .8,
  },
  button: {
    marginTop: 25,
    backgroundColor: '#ddd',
    height: 55,
    justifyContent: 'center',
    alignItems: 'center',
    marginHorizontal: 10,
  }
});

```

Attach the animated values to each component you're animating.

7.5 Creating an animated sequence

An animated *sequence* is a series of animations that occur one after another, with each animation waiting for the previous animation to complete before it begins. You can create an animated sequence with `sequence`. Like `parallel`, `sequence` takes an array of animations:

```

Animated.sequence([
  animationOne,

```

```

    animationTwo,
    animationThree
  ]).start()

```

In this example, you'll create a sequence that drops the numbers 1, 2, and 3 into the screen, 500 milliseconds apart (figure 7.5).

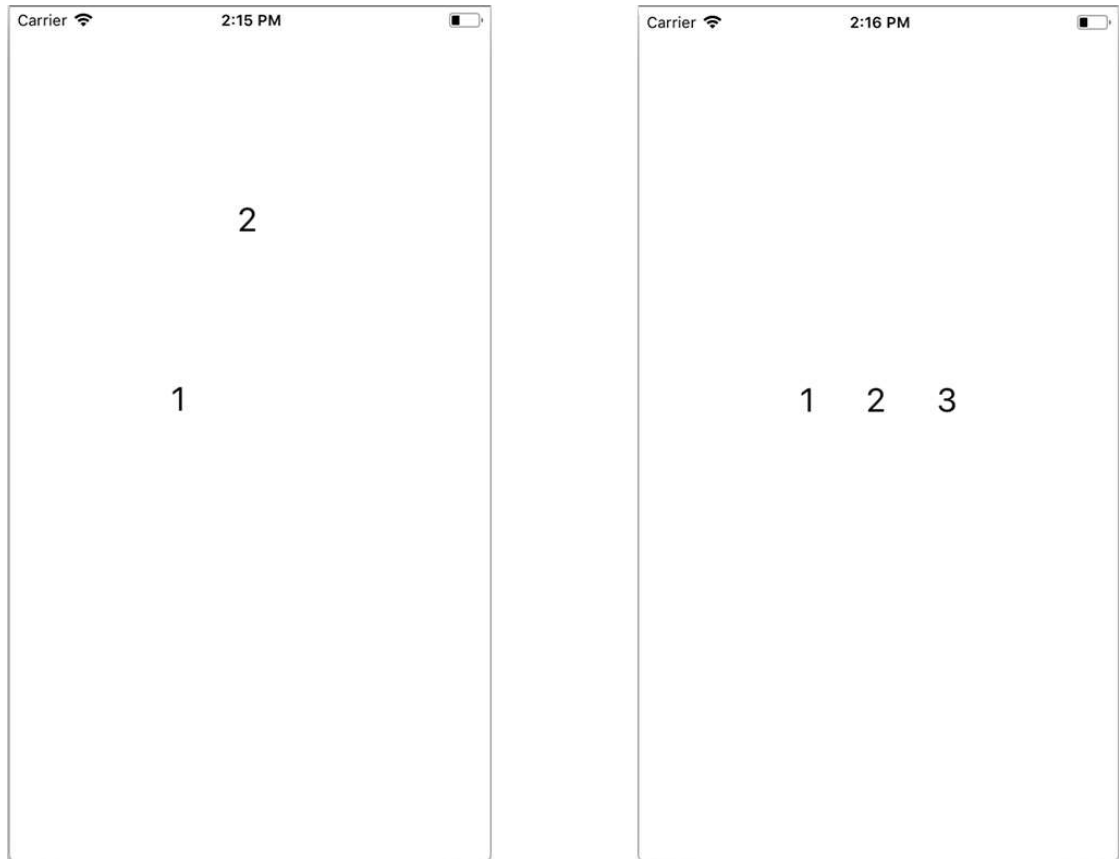


Figure 7.5 Creating an animated sequence of numbers

Listing 7.5 Creating a sequence of animations

```

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

export default class RNAnimations extends Component {
  componentDidMount() {
    this.animate();
  }
}

```

Imports Animated from React Native

Calls the animate function when the component mounts

```

AnimatedValue1 = new Animated.Value(-30);
AnimatedValue2 = new Animated.Value(-30);
AnimatedValue3 = new Animated.Value(-30);
animate = () => {
  const createAnimation = (value) => {
    return Animated.timing(
      value, {
        toValue: 290,
        duration: 500
      })
  }
  Animated.sequence([
    createAnimation(this.AnimatedValue1),
    createAnimation(this.AnimatedValue2),
    createAnimation(this.AnimatedValue3)
  ]).start()
}
render() {
  return (
    <View style={styles.container}>
      <Animated.Text style={styles.text,
        { marginTop: this.AnimatedValue1}}>
        1
      </Animated.Text>
      <Animated.Text style={styles.text,
        { marginTop: this.AnimatedValue2}}>
        2
      </Animated.Text>
      <Animated.Text style={styles.text,
        { marginTop: this.AnimatedValue3}}>
        3
      </Animated.Text>
    </View>
  );
}
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    flexDirection: 'row',
  },
  text: {
    marginHorizontal: 20,
    fontSize: 26
  }
});

```

Creates three animated values, passing in -30 for the beginning value

Creates a createAnimation function as a helper for making a new timing animation

Starts the sequence, calling createAnimation once for each animated value

Passes the animated values to the three Animated.Text components

Passes the animated values to the three Animated.Text components

This example uses beginning animated values of -30 because they're the `marginTop` values for the text elements: the text is pulled off the top of the screen and hidden before the animation begins. The `createAnimation` function also receives an animated value as its argument.

7.6 Using *Animated.stagger* to stagger animation start times

The last type of animation we'll go over is *Animated.stagger*. Like *parallel* and *sequence*, *stagger* takes an array of animations. The array of animations starts in parallel, but the start time is staggered equally across all the animations. Unlike *parallel* and *sequence*, the first argument to *stagger* is the stagger time, and the second argument is the array of animations:

```
Animated.stagger(  
  100,  
  [  
    Animation1,  
    Animation2,  
    Animation3  
  ]  
) .start()
```

In this example, you'll dynamically create a large number of animations that are used to stagger a series of red boxes onto the screen (figure 7.6).

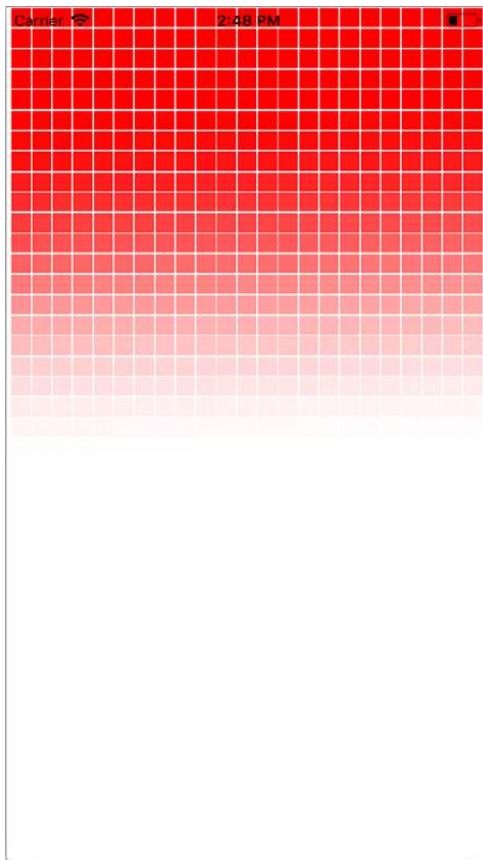


Figure 7.6 Using *Animated.stagger* to create an array of staggered animations

Listing 7.6 Using Animated.stagger to stagger a series of animations

```

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

export default class RNAnimations extends Component {
  constructor () {
    super()
    this.animatedValues = []
    for (let i = 0; i < 1000; i++) {
      this.animatedValues[i] = new Animated.Value(0)
    }
    this.animations = this.animatedValues.map(value => {
      return Animated.timing(
        value,
        {
          toValue: 1,
          duration: 6000
        }
      )
    })
  }
  componentDidMount () {
    this.animate()
  }
  animate = () => {
    Animated.stagger(15, this.animations).start()
  }
  render () {
    return (
      <View style={styles.container}>
        {
          this.animatedValues.map((value, index) => (
            <Animated.View key={index}
              style={[{opacity: value},
                styles.box]} />
          ))
        }
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    flexDirection: 'row',
    flexWrap: 'wrap'
  },
  box: {

```

Imports Animated from React Native

Creates an array that will contain 1,000 animated values of 0

Creates an array of Animated.timing animations referencing the animated values created in the animatedValues array

Calls the animate method

Calls Animated.stagger().start(), passing in the timing of 15 ms and array of animations

Maps over the animations, creating an Animated.View for each item in the array


```

    width: 15,
    height: 15,
    margin: .5,
    backgroundColor: 'red'
  }
})

```

7.7 Other useful tips for using the Animated library

In addition to the parts of the Animated API we've already covered, a few more techniques are useful to know about: resetting an animated value, invoking callbacks, offloading animations to the native thread, and creating custom animatable components. This section takes a quick look at each of these.

7.7.1 Resetting an animated value

If you're calling an animation, you can reset the value to whatever you want by using `setValue(value)`. This is useful if you've already called an animation on a value and need to call the animation again, and you want to reset the value to either the original value or a new value:

```

animate = () => {
  this.animatedValue.setValue(300);
  #continue here with the new animated value
}

```

7.7.2 Invoking a callback

When an animation is completed, an optional callback function can be fired, as shown here:

```

Animated.timing(
  this.animatedValue,
  {
    toValue: 1,
    duration: 1000
  }
).start(() => console.log('animation is complete!'))

```

7.7.3 Offloading animations to the native thread

Out of the box, the Animated library performs animations using the JavaScript thread. In most cases, this works fine, and you shouldn't have many performance problems. But if anything is blocking the JavaScript thread, you may see issues like frames being skipped, causing laggy or jumpy animations.

There's a way around using the JavaScript thread: you can use a configuration Boolean called `useNativeDriver`. `useNativeDriver` offloads the animation to the native UI thread, and the native code can then update the views directly on the UI thread, as shown here:

```

Animated.timing(
  this.animatedValue,

```

```

    {
      toValue: 100,
      duration: 1000,
      useNativeDriver: true
    }
  ).start();

```

Not every animation can be offloaded using `useNativeDriver`, so be sure to check the Animated API documentation when you use it. As of this writing, only non-layout properties can be animated using this method; flexbox properties as well as properties like margins and padding can't be animated.

7.7.4 Creating a custom animatable component using `createAnimatedComponent`

We mentioned in section 7.1 that the only animatable components out of the box are `View`, `Text`, `Image`, and `ScrollView`. There's also a way to create an animated component from any existing or custom React Native element or component. You can do this by wrapping the component in a call to `createAnimatedComponent`. Here's an example:

```

const Button = Animated.createAnimatedComponent(TouchableHighlight)

<Button onPress={somemethod} style={styles.button}>
  <Text>Hello World</Text>
</Button>

```

Now you can use the button just like a regular React Native component.

Summary

- The built-in Animated API is the recommended way to create animations in React Native.
- `Animated.timing` is the main method to use to create animations using the Animated library.
- The only components that are animatable out of the box are `View`, `Text`, `ScrollView`, and `Image`, but you can create custom animatable components using `createAnimatedComponent`.
- To interpolate and reuse animated values, use the `Animated.interpolate` method.
- To create and trigger an array of animations at the same time, use `Animated.parallel`.
- To create an infinitely looping animation, use `Animated.loop`.
- Use `Animated.sequence` to create a sequence of animations that execute one after another.
- Use `Animated.stagger` to create an array of animations that happen in parallel, but whose start times are staggered based on the time passed in.

8

Using the Redux data architecture library

This chapter covers

- How the React context API works
- Creating a Redux store
- How to use Redux actions and reducers to manage global state
- Reducer composition using `combineReducers`

When building React and React Native applications in the real world, you'll quickly learn that the data layer can become complex and unmanageable if it isn't handled very precisely and deliberately. One way to handle data is to keep it in component state and pass it around as props, as we've done throughout this book. Another way is to use a data architecture pattern or library. This chapter covers the Redux library: it's the most widely adopted method of handling data in the React ecosystem, and it's maintained by Facebook, the same team that maintains both React and React Native.

8.1 What is Redux?

In the Redux documentation, the library is described as “a predictable state container for JavaScript apps.” Redux is basically a global state object that's the single source of truth in an application. This global state object is received as props into

React Native components. Any time a piece of data is changed in the Redux state, the entire application receives this new data as props.

Redux simplifies application state by moving it all into one place called a *store*; this makes it much easier to reason about and understand. When you need the value of something, you'll know exactly where to look in a Redux application and can expect the same value to be available and up-to-date elsewhere in the application, too.

So how does Redux work? It takes advantage of a React feature called *context*, a mechanism for creating and managing global state.

8.2 Using context to create and manage global state in a React application

Context is a React API that creates global variables that can be accessed anywhere in the application, as long as the component receiving the context is a child of the component that created it. Normally you'd have to do this by passing props down each level of the component structure. With context, you don't need to use props. You can use the context anywhere in the app and access it without passing it down to each level.

NOTE Although context is good to understand and is used in numerous open source libraries, you probably won't need to use it in apps unless you're building an open source library or can't find another way around a problem. We're discussing it in order for you to fully understand how Redux works under the hood.

Let's look at how to create context in a basic component structure of three components: Parent, Child1, and Child2. This example shows how to apply application-wide theming from a parent level, which could make it possible to control the styling of an entire application if needed.

Listing 8.1 Creating context

```
const ThemeContext = React.createContext()

class Parent extends Component {
  state = { themeValue: 'light' }
  toggleThemeValue = () => {
    const value = this.state.themeValue === 'dark' ? 'light' : 'dark'
    this.setState({ themeValue: value })
  }
  render() {
    return (
      <ThemeContext.Provider
        value={{
          themeValue: this.state.themeValue,
          toggleThemeValue: this.toggleThemeValue
        }}
      >
        <View style={styles.container}>

```

```

        <Text>Hello World</Text>
      </View>
    <Child1 />
  </ThemeContext.Provider>
);
}
}

const Child1 = () => <Child2 />

const Child2 = () => (
  <ThemeContext.Consumer>
    { (val) => (
      <View style={ [styles.container,
        val.themeValue === 'dark' &&
        { backgroundColor: 'black' } ] }>
        <Text style={styles.text}>Hello from Component2</Text>
        <Text style={styles.text}
          onPress={val.toggleThemeValue}>
          Toggle Theme Value
        </Text>
      </View>
    ) }
  </ThemeContext.Consumer>
)

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  text: {
    fontSize: 22,
    color: '#666'
  }
})

```

Stateless function that returns a component, demonstrating that you aren't passing props between Parent and Child2

Stateless function that returns a component wrapped in a ThemeContext.Consumer

The Child2 stateless function returns a component that's wrapped in a ThemeContext.Consumer. ThemeContext.Consumer requires a function as its child. The function receives an argument containing whatever context is available (in this case, the val object containing two properties). You can now use the context values in the component.

When you use Redux with React, you'll take advantage of a function called connect, which basically takes pieces of context and makes them available as props in the component. Understanding context should make learning Redux much easier!

8.3 Implementing Redux with a React Native app

Now that you've know the fundamentals of what Redux is and have seen what's going on under the hood with context, let's create a new React Native app and start adding

Redux. You'll be creating a basic list app you can use to keep up with books you've read (see figure 8.1). Follow these steps:

- 1 Create a new React Native application, and call it RNRedux:

```
react-native init RNRedux
```

- 2 Change into the new directory:

```
cd RNRedux
```

- 3 Install the Redux-specific dependencies you'll need:

```
npm i redux react-redux --save
```

- 4 In the root of the directory, create a folder called `src`, and add to it the following files: `Books.js` and `actions.js`. Also, in `src`, create a folder called `reducers`, containing two files: `bookReducer.js` and `index.js`. The `src` folder structure should now look like figure 8.2.

The next thing to do is create the first piece of Redux state. You'll do this in `bookReducer.js`. In section 8.1, I described Redux as a global object. To create this global object, you'll piece together smaller objects using what are known as *reducers*.



Figure 8.1 Completed book list application



Figure 8.2 RNRedux `src` folder structure

8.4 Creating Redux reducers to hold Redux state

A reducer is a function that returns an object; when combined with other reducers, they create the global state. Reducers can be more easily thought of as data stores. Each store contains a piece of data, which is exactly what reducers do in the Redux architecture.

In the reducers folder are two files: `bookReducer.js` and `index.js`. In `index.js`, you'll combine all the reducers in the app to create the global state. The app will have only one reducer to start with (`bookReducer`), so the global state object will look something like this:

```
{
  bookReducer: {}
}
```

You've yet to decide what to put in `bookReducer`. An array in which to store a list of books will be a good start. This reducer will create and return a piece of state that you'll access later from the Redux store. In `reducers/bookReducer.js`, create your first reducer. This code creates a function whose only purpose (for now) is to return the state.

Listing 8.2 Creating a reducer

```
const initialState = {    #A
  books: [{ name: 'East of Eden', author: 'John Steinbeck' }]
}    #A
```

Creates the initialState object

```
const bookReducer = (state = initialState) => {
  return state
}
```

Returns the state

Takes a state argument and sets the default to the initial state

```
export default bookReducer
```

The `initialState` object will hold the beginning state. In this case, that's an array of books that you'll populate with objects containing name and author props. You create a function that takes an argument, `state`, and sets the default value to the initial state. When this function is first called, `state` will be undefined and will return the `initialState` object. At this time, the function's only purpose is to return the state.

Now that you've created the first reducer, go into `rootReducer` (`reducers/index.js`) and create what will be the global state. The root reducer gathers all the reducers in the application and allows you to make a global store (state object) by combining them.

Listing 8.3 Creating a root reducer

```
import { combineReducers } from 'redux'
import bookReducer from './bookReducer'
```

Imports the combineReducers function from Redux

```
const rootReducer = combineReducers({
  bookReducer
})
```

Imports the bookReducer reducer

Creates a root reducer containing all the reducers; in this case it contains the single property bookReducer

```
export default rootReducer
```

Next, to hook this all together, you'll go into `App.js`, create the Redux store, and make the store available to all child components using a couple of Redux and React-Redux helpers.

8.5 Adding the provider and creating the store

In this section, you'll add a *provider* to the app. A provider is usually a parent component that passes data of some kind along to all child components. In Redux, the provider passes the global state/store to the rest of the application. In App.js, update the code as follows.

Listing 8.4 Adding the provider and store

```
import React from 'react'

import Books from './src/Books'
import rootReducer from './src/reducers'

import { Provider } from 'react-redux'
import { createStore } from 'redux'

const store = createStore(rootReducer)

export default class App extends React.Component {
  render() {
    return (
      <Provider store={store}>
        <Books />
      </Provider>
    )
  }
}
```

Imports the Books component (created in listing 8.5)

Imports rootReducer

Imports the Provider wrapper from react-redux

Imports createStore

Creates a store, passing in the rootReducer

Returns the Books component wrapped in a Provider component, passing in the store as a prop to the Provider

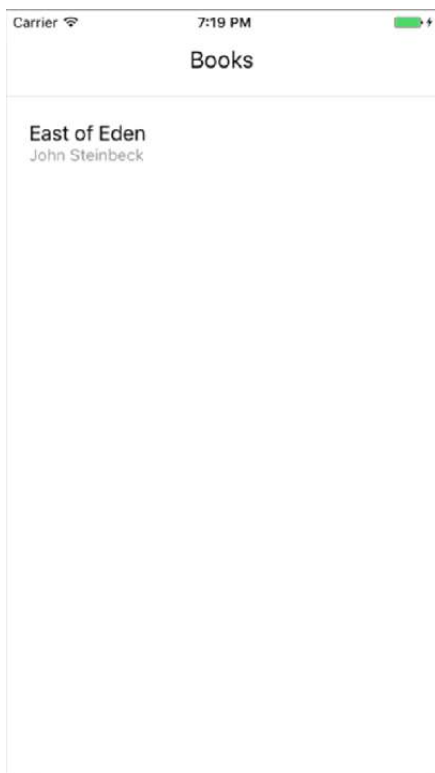


Figure 8.3 Rendering the list of books from the Redux store

The Provider wrapper is used to wrap the main component. Any child of Provider will have access to the Redux store. `createStore` is a utility from Redux that you use to create the Redux store by passing in the `rootReducer`. You're finished with the basic Redux setup, and you can now access the Redux store in the app.

In the Books component, you'll hook into the Redux store, pull out the books array, and map over the books, displaying them in the UI (figure 8.3). Because Books is a child of Provider, it can access anything in the Redux store.

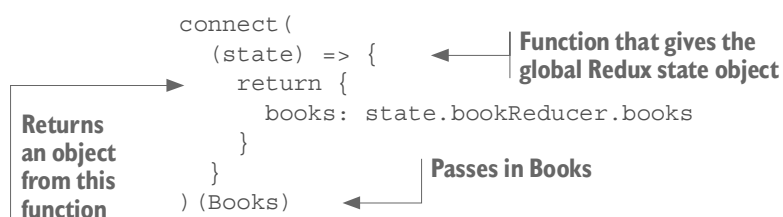
8.6 Accessing data using the connect function

You access the Redux store from a child component by using the `connect` function from `react-redux`. The first argument to `connect` is a function that gives you access to the entire Redux state. You can then return an object with whatever pieces of the store you want access to.

`connect` is a *curried* function, meaning in the most basic sense a function that returns another function. You'll have two sets of arguments, and a blueprint that looks something like this: `connect (args) (args)`. The properties in the object returned from the first argument to `connect` are then made available to the component as props.

Let's see what this means by looking at the `connect` function you'll use in the Books.js component.

Listing 8.5 `connect` function in Books.js



```

connect (
  (state) => {
    return {
      books: state.bookReducer.books
    }
  }
) (Books)

```

Annotations in the diagram:

- An arrow points from the text "Function that gives the global Redux state object" to the first argument `(state) => { ... }`.
- An arrow points from the text "Returns an object from this function" to the `return { ... }` block.
- An arrow points from the text "Passes in Books" to the second argument `(Books)`.

The first argument to `connect` is a function that gives the global Redux state object as an argument. You can then reference this state object and have access to anything in the Redux state. You return an object from this function. Whatever keys are returned in the object become available as props in the component you're wrapping: in this case, Books. You pass in Books as the only argument to the `connect` function's second function call.

Often, you'll separate this function and store it in a variable to make this easier to read:

```

const mapStateToProps = state => ({
  books: state.bookReducer.books
})

```

In this connected component is a new property called `this.props.books`, which is the books array from `bookReducer`. Tie all this together, access the books array, and map over the books to display them in the UI, as shown in the following listing (Books.js).

Listing 8.6 Accessing the Redux store and bookReducer data

```

import React from 'react'
import {
  Text,
  View,
  ScrollView,
  StyleSheet
} from 'react-native'

import { connect } from 'react-redux'

class Books extends React.Component<{}> {
  render() {
    const { books } = this.props

    return (
      <View style={styles.container}>
        <Text style={styles.title}>Books</Text>
        <ScrollView
          keyboardShouldPersistTaps='always'
          style={styles.booksContainer}
        >
          {
            books.map((book, index) => (
              <View style={styles.book} key={index}>
                <Text style={styles.name}>{book.name}</Text>
                <Text style={styles.author}>{book.author}</Text>
              </View>
            ))
          }
        </ScrollView>
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1
  },
  booksContainer: {
    borderTopWidth: 1,
    borderTopColor: '#ddd',
    flex: 1
  },
  title: {
    paddingTop: 30,
    paddingBottom: 20,
    fontSize: 20,
    textAlign: 'center'
  },
  book: {
    padding: 20
  },
  name: {

```

Imports connect from react-redux

Because the books array was returned from the connect function (at the bottom of the code listing), you have access to it as props.

Maps over the array, displaying the name and author of each book

```

    fontSize: 18
  },
  author: {
    fontSize: 14,
    color: '#999'
  }
})

const mapStateToProps = (state) => ({
  books: state.bookReducer.books
})

export default connect(mapStateToProps)(Books)

```

Takes the Redux state, and returns an object with a key containing the books array

Exports the connect function

You begin by importing `connect` from `react-redux`. In listing 8.5, you wrote the function returning the props inline. This listing separates it and names it `mapStateToProps`, following the convention of the Redux ecosystem. This naming convention makes a lot of sense, because you're essentially mapping Redux state to component props. This function takes the Redux state as an argument and returns an object with one key containing the books array from `bookReducer`. Finally, you export the `connect` function, passing in `mapStateToProps` as the first argument to `connect` and `Books` as the only argument in the second set of arguments to `connect`.

After launching the application, you should see a basic list of books, as shown earlier in figure 8.3.

8.7 Adding actions

Now that you have access to the Redux state, a logical next step is to add some functionality that will allow you to add books to the books array Redux store. To do this, you'll use *actions*. Actions are basically functions that return objects that send data to the store and update reducers; they're the only way to change the store. Each action should contain a `type` property in order for reducers to be able to use them. Here are a couple of examples of actions:

```

function fetchBooks() {
  return {
    type: 'FETCH_BOOKS'
  }
}

function addBook(book) {
  return {
    type: 'Add_BOOK',
    book: book
  }
}

```

Actions, when called using a Redux dispatch function, are sent to all reducers in the application as the second argument to the reducer. (We'll cover how to attach the Redux dispatch function later in this chapter.) When the reducer receives the action, you

check the action's type property and update what the reducer returns based on whether the action is one that it's listening for.

In this case, the only action you need for the next step is `addBook`, to add additional books to the array of books. In `actions.js`, create the following action.

Listing 8.7 Creating the first action

```
export const ADD_BOOK = 'ADD_BOOK'

export function addBook (book) {
  return {
    type: ADD_BOOK,
    book
  }
}
```

Creates and exports an `ADD_BOOK` constant for reuse in reducers

Creates the `addBook` function, which takes a single book object and returns an object containing a type and the passed-in book

Next, wire up `bookReducer` to use the `addBook` action.

Listing 8.8 Updating `bookReducer` to use the `addBook` action

```
import { ADD_BOOK } from '../actions'

const initialState = {
  books: [{ name: 'East of Eden', author: 'John Steinbeck' }]
}

const bookReducer = (state = initialState, action) => {
  switch(action.type) {
    case ADD_BOOK:
      return {
        books: [
          ...state.books,
          action.book
        ]
      }
    default:
      return state
  }
}

export default bookReducer
```

Imports the `ADD_BOOK` constant from the actions file

Adds a second argument to `bookReducer`: the action

Creates a switch statement that will switch on the action type

If the action type equals `ADD_BOOK`, returns a new books array

If the switch statement doesn't hit, returns the existing state

In the listing, if the action type is equal to `ADD_BOOK`, you return a new books array containing all the previous items in the array. You do so by creating a new array, using the spread operator to add the contents of the existing books array to the new array, and adding to the array a new item that's the book property of the action.

That's all you need to do in the Redux configuration to get this working. The last step is to go into the UI and wire it all together. To get the user's book info, you need to create a form. Figure 8.4 shows what the UI will look like.

This form has two inputs: one for the book name and one for the author name. It also has a submit button. When the user types into the form, you need to keep up with the values in the local state. You can then pass those values on to the action when the user clicks the submit button.

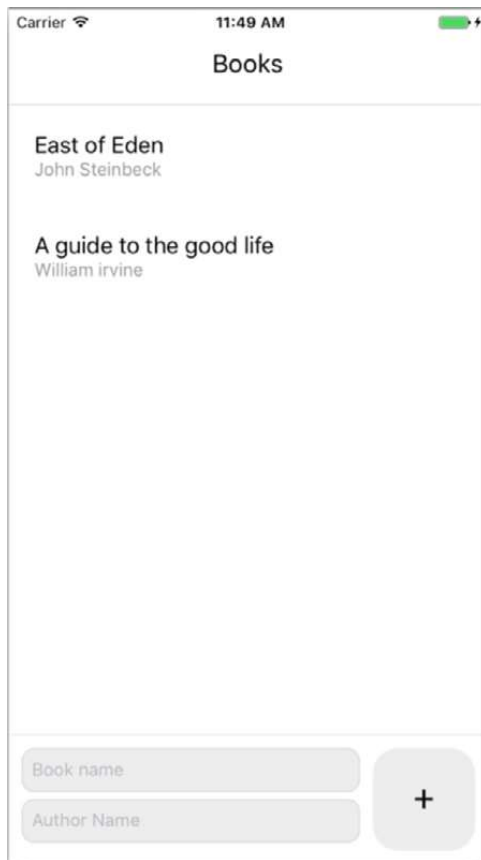


Figure 8.4 UI with added text inputs to capture the book and the author name

Open `Books.js`, and import the additional components needed for this functionality, as well as the `addBook` function from the actions. You'll also create an `initialState` variable to use as the local component state.

Listing 8.9 Additional imports in `Books.js`

```
import React from 'react'
import {
  Text,
  View,
  ScrollView,
  StyleSheet,
  TextInput,
  TouchableOpacity
} from 'react-native'
import { addBook } from '../actions'

import { connect } from 'react-redux'

const initialState = {
  name: '',
  author: ''
}

...
```

Imports TextInput and TouchableOpacity

Imports the addBook function from the actions file

Creates an initialState object containing name and author fields

Next, in the body of the class, you need to create three things: the component state, a method that keeps up with the component state when the `textInput` values change, and a method that will send the action to Redux containing the book values (name and author) when the submit button is pressed. Before the render method, add the following code.

Listing 8.10 Adding state and class methods to Books.js

```
class Books extends React.Component {
  state = initialState
  updateInput = (key, value) => {
    this.setState({
      ...this.state,
      [key]: value
    })
  }
  addBook = () => {
    this.props.dispatchAddBook(this.state)
    this.setState(initialState)
  }
  ...
}
```

Gives the component state the value of the initialState variable

Creates an updateInput method that takes two arguments: key and value. You'll update the state by using the spread operators to add the existing state key-value pairs to the new state and then adding the new key-value pair.

Calls dispatchAddBook, accessible as props from the connect function

The `addBook` method calls a function that you have access to as props from the `connect` function: `dispatchAddBook`. This function accepts the entire state as an argument, which is an object with `name` and `author` properties. After the dispatch action has been called, you then clear the component state by resetting it to the `initialState` value.

With the functionality in place, you can create the UI and hook these methods up to it. Under the closing tag of the `ScrollView` in `Books.js`, add the form UI.

Listing 8.11 Adding the UI for the form

```
class Books extends React.Component {
  ...
  render() {
    ...
  }
  ...
  </ScrollView>
  <View style={styles.inputContainer}>
    <View style={styles.inputWrapper}>
      <TextInput
        value={this.state.name}
        onChangeText={value => this.updateInput('name', value)}
        style={styles.input}
        placeholder='Book name'
      />
      <TextInput
        value={this.state.author}
        onChangeText={value => this.updateInput('author', value)}
        style={styles.input}
        placeholder='Author Name'
      />
    </View>
  </View>
}
```

Receives the updateInput method as the property of onChangeText, passing 'name' or 'author' as the first argument and the value of TextInput as the second argument

```

    <TouchableOpacity onPress={this.addBook}>
      <View style={styles.addButtonContainer}>
        <Text style={styles.addButton}>+</Text>
      </View>
    </TouchableOpacity>
  </View>
</View>
}
}

const styles = StyleSheet.create({
  inputContainer: {
    padding: 10,
    backgroundColor: '#ffffff',
    borderTopColor: '#ededed',
    borderTopWidth: 1,
    flexDirection: 'row',
    height: 100
  },
  inputWrapper: {
    flex: 1
  },
  input: {
    height: 44,
    padding: 7,
    backgroundColor: '#ededed',
    borderColor: '#ddd',
    borderWidth: 1,
    borderRadius: 10,
    flex: 1,
    marginBottom: 5
  },
  addButton: {
    fontSize: 28,
    lineHeight: 28
  },
  addButtonContainer: {
    width: 80,
    height: 80,
    backgroundColor: '#ededed',
    marginLeft: 10,
    justifyContent: 'center',
    alignItems: 'center',
    borderRadius: 20
  },
  ...
})

const mapDispatchToProps = {
  dispatchAddBook: (book) => addBook(book)
}

export default connect(mapStateToProps, mapDispatchToProps)(Books)
}

```

← Calls the addBook method. TouchableOpacity wraps the View component, allowing it to respond properly to touches.

← Adds new styles

← Creates a mapDispatchToProps object

← Passes in mapDispatchToProps as the second argument to connect

In the `mapDispatchToProps` object, you can declare functions you want access to as props in the component. You create a new function called `dispatchAddBook` and have it call the `addBook` action, passing in `book` as an argument. Similar to how `mapStateToProps` maps state to component props, `mapDispatchToProps` maps actions (that need to be dispatched to reducers) to component props. In order for an action to be recognized by the Redux reducers, it must be declared in this `mapDispatchToProps` object. You pass in `mapDispatchToProps` as the second argument to the `connect` function.

Now you should be able to easily add books to the book list.

8.8 Deleting items from a Redux store in a reducer

The next logical step is to add a way to remove books you've already read. Given everything you've put together, this won't require too much more work (figure 8.5).

The first thing to think about when removing an item from an array such as this is how to identify a book as being unique. Right now, a user could have multiple books with the same author or multiple books with the same name, so using the existing properties won't work. Instead, you can use a library such as `uuid` to create unique identifiers on the fly. To begin setting this up, from the command line, install the `uuid` library into `node_modules`:

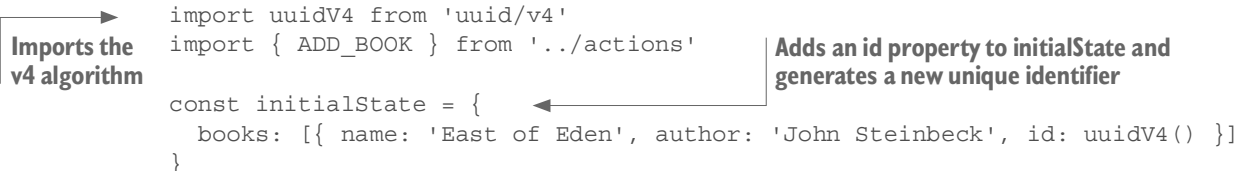
```
npm i uuid --save
```



Figure 8.5 Adding the Remove button to the Books.js UI

Next, you'll implement a unique identifier in the reducer for the items in the `initialState` books array. In `reducers/bookReducer.js`, update the imports and `initialState` to look like the next listing.

Listing 8.12 Importing and using uuid



```
import uuidV4 from 'uuid/v4'
import { ADD_BOOK } from '../actions'

const initialState = {
  books: [{ name: 'East of Eden', author: 'John Steinbeck', id: uuidV4() }]
}
```

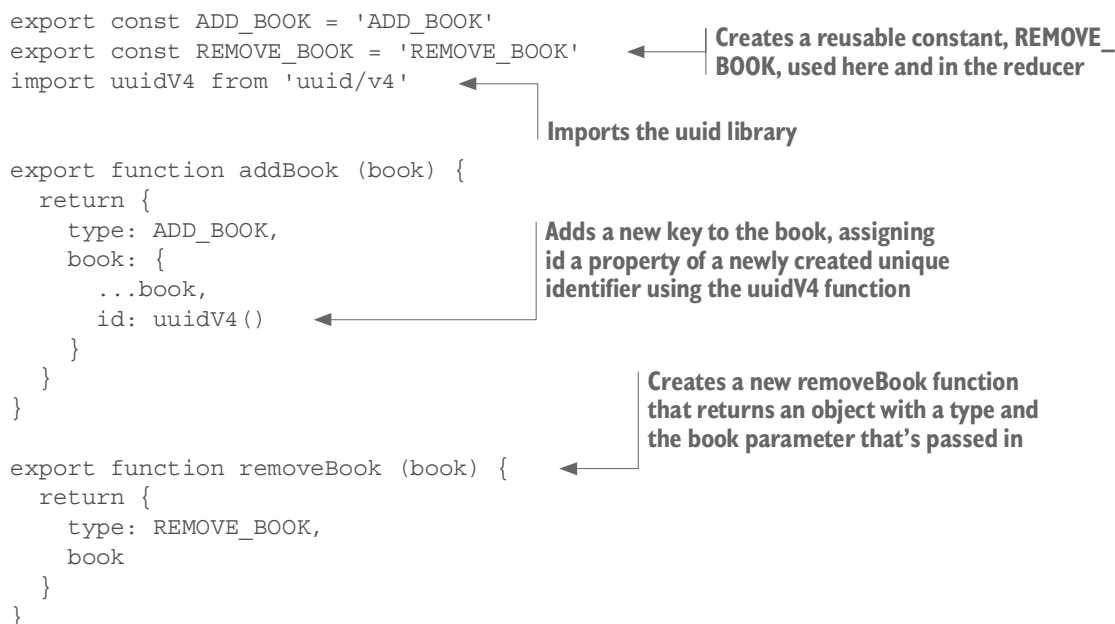
Imports the v4 algorithm

Adds an id property to initialState and generates a new unique identifier

The uuid library has a few algorithms to choose from. Here, you import only the v4 algorithm, which creates a random 32-character string. Then you add a new property to the `initialState` books array, `id`, and generate a new unique identifier by calling `uuidV4()`.

Now that you have a way to uniquely identify the items in the books array, you're ready to move forward with the rest of the functionality. The next step is to create a new action in `actions.js`; you'll call it when you want to remove a book. You also need to update the `addBook` action to add an ID to newly created books.

Listing 8.13 Creating the `removeBook` action



```
export const ADD_BOOK = 'ADD_BOOK'
export const REMOVE_BOOK = 'REMOVE_BOOK'
import uuidV4 from 'uuid/v4'

export function addBook (book) {
  return {
    type: ADD_BOOK,
    book: {
      ...book,
      id: uuidV4()
    }
  }
}

export function removeBook (book) {
  return {
    type: REMOVE_BOOK,
    book
  }
}
```

Creates a reusable constant, REMOVE_BOOK, used here and in the reducer

Imports the uuid library

Adds a new key to the book, assigning id a property of a newly created unique identifier using the uuidV4 function

Creates a new removeBook function that returns an object with a type and the book parameter that's passed in

Next, the reducer needs to be aware of the new action. In `reducers/bookReducer.js`, create a new type listener, this one for `REMOVE_BOOK`, and add the necessary functionality to remove a book from the array of books stored in the Redux state.

Listing 8.14 Removing an item from an array in a Redux reducer

Returns a new array containing the first and second half of the existing books array, leaving out the index of the book to be removed

```
import uuidV4 from 'uuid/v4'
import { ADD_BOOK, REMOVE_BOOK } from '../actions'

const initialState = {
  books: [{ name: 'East of Eden', author: 'John Steinbeck', id: uuidV4() }]
}

const bookReducer = (state = initialState, action) => {
  switch(action.type) {
    ...
    case REMOVE_BOOK:
      const index = state.books.findIndex(
        book => book.id === action.book.id
      )

      return {
        books: [
          ...state.books.slice(0, index),
          ...state.books.slice(index + 1)
        ]
      }
    ...
  }
}

export default bookReducer
```

Imports the new REMOVE_BOOK constant from the actions folder

Adds a new case to the switch statement that listens for the REMOVE_BOOK action type

Finds the index of the book to be deleted

The last thing to do is implement this new removeBook functionality in the UI of the Books component (Books.js). You'll import the removeBook action, add a remove button to each rendered item, and wire the remove button up to the removeBook action.

Listing 8.15 Adding removeBook functionality

```
...
import { addBook, removeBook } from './actions'
...
removeBook = (book) => {
  this.props.dispatchRemoveBook(book)
}
...

{
  books.map((book, index) => (
    <View style={styles.book} key={index}>
      <Text style={styles.name}>{book.name}</Text>
      <Text style={styles.author}>{book.author}</Text>
      <Text onPress={() => this.removeBook(book)}>
        Remove
      </Text>
    </View>
  ))
}
```

Adds removeBook as an import from the actions file

Creates a new class method removeBook, calling this.props.dispatchRemoveBook as a new key in mapDispatchToProps

Returns a new Text component and attaches removeBook to its onPress event

```
const mapDispatchToProps = {  
  dispatchAddBook: (book) => addBook(book),  
  dispatchRemoveBook: (book) => removeBook(book)  
}  
...
```

← Adds the new `dispatchAddBook` function to `mapDispatchToProps`

Summary

- With context, you can pass properties and data to children in a React Native application without explicitly passing the properties to each individual child.
- Reducers are similar to a traditional data store in the sense that they keep up with and return data, but also allow you to update data in the store.
- You can create and use actions to update a Redux store.
- With the `connect` function, you can access data from the Redux state as props and also create dispatch functions that interact with reducers using actions.
- Any time data needs to be changed in a reducer, it must be done by using an action.

