Vladimir Novick

React Native - Building Mobile Apps with JavaScript

Build real-world iOS and Android native apps with JavaScript

Packt

React Native - Building Mobile Apps with JavaScript

Build real-world iOS and Android native apps with JavaScript

Vladimir Novick

BIRMINGHAM - MUMBAI

React Native - Building Mobile Apps with JavaScript

First published: August 2017

Production reference: 1230817

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78728-253-7

www.packtpub.com

Credits

Author
Vladimir Novick

Reviewer
Vijay Thirugnanam

Commissioning Editor
Smeet Thakkar

Acquisition Editor
Siddharth Mandal

Content Development Editor
Onkar Wani

Technical Editor
Akhil Nair

Copy Editor
Dhanya Baburaj

Project Coordinator
Devanshi Doshi

Proofreader
Safis Editing

Indexer
Mariammal Chettiyar

Graphics
Jason Monteiro

Production Coordinator
Shraddha Falebhai

About the Author

Vladimir Novick is developer, software architect, public speaker, and consultant. Coming from a web programming background, he started programming for mobile using React Native around 2015. Vladimir works in the web, mobile, VR, AR, and Internet of Things fields daily, developing complex enterprise-level software, consulting clients, contributing to open source, and teaching, and talking at various meetups and conferences. He is also the author of several courses and workshops.

Previously, Vladimir worked in the video, gaming, sports, and entertainment industries where he architected and developed large-scale web applications for hundreds of millions of users, each month.

I would like to thank several people who helped make this book a reality. First, I want to thank Tatyana Novick, my wife. Her encouragement and support is something that kept me going forward. Thanks to my kids for supporting me and understanding that sometimes, daddy is busy writing. Thanks to my dad for introducing me to programming back in the 90s. Thanks to my mom, even though I kept the writing process a secret from you until the release, the idea of presenting you a hardcopied book was one of the initial drivers that encouraged me to write.
Thanks to Onkar Wani, my content development editor, whose professionalism and help was invaluable.

About the Reviewer

Vijay Thirugnanam is a React developer from Bangalore, India. Before working as an independent developer, he worked for more than 16 years, delivering software solutions as an engineer, a consultant, and a manager. His past experience includes working in organizations such as ABB, Dell, and Microsoft. He holds a bachelor's degree from Indian Institute of Technology, Madras.

His current passion is developing apps with React and React native. Apart from that, he likes to take short breaks at resorts around Bangalore with his family, his wife Shalini and daughter Tanishka.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`. Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https:/ / www. packtpub. com/ mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https:/ / www. amazon. com/ dp/ 1787282538`. If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

For my wife and kids, who believed in me and supported all the way

Table of Contents

Preface 1

Chapter 1: Understanding Why React Native is the Future of Mobile Apps 9

What is React Native? 9

The history of React Native 10

It all begun with ReactJS 10

Hackathon, that grew bigger 11

React Native today 12

The motivation of creating React Native 12

Mobile development is too specific 12

Taking the React approach 12

Making developer experience state of the art 13

How React Native is different from hybrid applications? 14

What are hybrid applications? 14

Why Native applications lead the way? 14

Is React Native really a Native framework? 15

How the React Native bridge from JavaScript to Native world works? 15

Information flow 16

Architecture 16

Threading model 17

The benefits of React Native 18

Developer experience 18

Chrome DevTools debugging 18

Live reload 20

Hot reload 20

Component hierarchy inspections 20

Web-inspired layout techniques 22

Code reusability across applications 23

Summary 24

Chapter 2: Working with React Native 25

Setting up an environment for developing iOS and Android apps 26

Installing projects with native code 26

Installing iOS dependencies 28

Installing Android dependencies 28

Installing JDK 28

Installing Android studio 29

Installing SDK and built tools 31

Setting up the ANDROID_HOME environment variable 33

Creating an Android virtual device 33
The development environment 36
Creating your application 37
Application with create-react-app and Expo 39
Introduction to JSX and how it's used in React Native 45
What is JSX? 47
Children in JSX 48
JSX props 49
Stateful versus presentational components 50
Presentational components 51
Stateful components 52
React lifecycle methods 54
Mounting 54
Updating 55
Structuring React Native apps and their resemblance to HTML 57
Thinking in React 57
Reusable components 57
Containers 57
Basic React Native components 57
The folder structure 59
Summary 61
Chapter 3: Getting Familiar with React Native Components 62
Platform-independent components 63
Basic components 63
View 64
Layouting 65
Touch events 65
Accessibility 66
Text 67
StatusBar 68
Images and media 70
Basic user interaction 71
Button 72
TouchableOpacity 74
TouchableHighlight 75
TouchableWithoutFeedback 75
Getting feedback from your application 75
ActivityIndicator 76
Modal 79

Dealing with lists of data 80
ListView 80
ScrollView 83
RefreshControl 84
FlatList 84
SectionList 89
VirtualizedList 91
Embedding web content 91
Handling user input 92
TextInput 93
Restricted choice inputs 94
Platform-dependent components 96
Detecting specific platform 96
Extensions 97
DatePickerIOS 97
Progress bars 98
Additional controls 98
SegmentedControlIOS 98
TouchableNativeFeedback 99
Platform-specific navigation 99
Navigation in React Native 100
The Navigator component 100
Summary 105
Chapter 4: Debugging and Testing React Native 106
Debugging your React Native apps 107
Developer in-app menu 107
Reloading your app 109
Reloading 110
Live reload 110
Hot reloading 112
Remote debugging 114
Debug with Chrome DevTools 114
Debuging our device 116
Logging 117
In-app errors and warnings 118
Errors 118
Warnings 121
Inspecting React Native components 123
Performance monitoring 126
Testing 128

Introduction to the Jest testing framework 128
Setup 128
Preset system 128
Basic syntax 129
Snapshot testing your React Native components 131
Working with functions 133
Mocking modules 136
Summary 137
Chapter 5: Bringing the Power of Flexbox to the Native World 138
Flexbox styling concepts and techniques 138
The flexbox layout 139
Aligning elements 140
Item dimensions 142
How different is React Native styling from CSS or inline? 143
Laying out our app 145
Dealing with background images 146
Applying styles conditionally 148
Best practices and techniques for styling your React Native applications 150
Dimensions 151
Style sheet 151
Split your styles 152
Code your styles 152
Extract common components 153
Summary 153
Chapter 6: Animating React Native Components 154
Understanding animations 155
Conventions 155
How animation works 156
Using the LayoutAnimation API for simple animations 156
Basic syntax 160
The preset system under the hood 163
Using Animated API for complex animations 170
Animated values 170
Calculations 174
Animated functions 175
Interpolation 177
Extrapolation 178

Combining several animations to create complex sequences 181
Panning and scrolling animations 184
Summary 186
Chapter 7: Authenticating Your App and Fetching Data 187
Getting familiar with Firebase 188
What is Firebase? 189
Firebase setup 190
Creating real-time database 192
Managing permissions 195
Bringing Firebase to React Native and fetching data 195
Initializing your app 197
Setting up Firebase database listeners 199
Writing data to Firebase 200
Fetching data in React Native 207
Websockets support in React Native 208
Fetching and uploading files 208
Setting up authentication at Firebase 209
Creating functional Login and Sign Up screens 210
Login screen 210
The Sign Up screen 214
Authentication logic 214
Authenticating via social providers 220
Summary 220
Chapter 8: Implementing a Flux Architecture with Redux or MobX 221
What is Flux architecture? 222
MVC problem 223
The unidirectional data flow solution 224
How it works 224
Redux concepts and usage 225
What is Redux? 226
Redux in a nutshell 226
Three principles 226
A single source of truth 226
State is read-only 227
Changes are made with pure functions 227
Basic building blocks 227
Actions 228
Reducers 228
Store 230
Middleware 230

Data flow 231
Connecting Redux to your app 232
Containers versus Components 234
Provider 234
Using connect 235
Getting relevant state keys with selectors 237
Using redux-thunk for async actions 237
Async data flow 238
Adding redux-thunk 238
Centralizing side effects 241
Mobx- a functional reactive Flux implementation 241
What is Mobx? 242
Basic concepts 243
State 243
Derivations 243
Actions 243
Connecting MobX to our app 244
Summary 247
Chapter 9: Understanding Supported APIs and How to Use Them 248
Linking libraries and APIs with native code 249
Auto linking 250
Manual linking 250
Linking libraries 250
Configuring build phases 251
Getting familiar with a list of native APIs covered by React Native 253
Notification APIs 254
Alert 254
AlertIOS 256
ToastAndroid 256
ActionSheetIOS 257
Vibration 259
PushNotificationIOS 259
Information APIs 260
AccessibilityInfo 260
AppState 260
NetInfo 261
PixelRatio 261
Dimensions 261
Platform 262
Settings 262
Input related 264
Clipboard 264

Keyboard 264
DatePickerAndroid 265
TimePickerAndroid 265
App related 266
InteractionManager 267
Linking 267
Image related 268
ImageEditor 268
ImageStore 268
ImagePickerIOS 269
Style related 269
Other various APIs 269
BackHandler 270
PermissionsAndroid 270
AdSupportIOS 270
Share 271
Retrieving and saving photos with CameraRoll API 272
Getting your exact location with GeoLocation API 276
IOS 277
Android 277
Usage 278
Learning about persistence with AsyncStorage API 280
Responding to user gestures with PanResponder 281
The Gesture responder system 286
PanResponder 287
Combining it with Animated 289
Summary 291
Chapter 10: Working with External Modules in React Native 292
Diving deeper into react-navigation 293
Navigators explained 293
Navigation 295
Redux integration 295
MobX integration 297
Setting the app navigation structure for a real app. 298
The best open source packages to use 301
Visuals and animations 302
react-native-vector-icons 302
react-native-animatable 303
lottie-react-native 304
Shoutem UI 304
NativeBase 304
react-native-elements 306

Social providers 307
Facebook 307
OAuth 308
Additional APIs 308
ExpoKit 308
Maps 308
Image Picker 309
Video 311
Toasts 311
Camera 312
Data related packages 314
react-native-fetch-blob 314
react-native-firebase 316
react-native-push-notifications 317
react-native-i18n 320
Boilerplates 321
Writing your own Native modules 324
Diving into iOS and Objective-C 324
Android 328
Creating folder structure and files 328
Creating a native module Java class 329
Creating a native module package 330
Integrating React Native with the existing apps 332
Summary 333
Chapter 11: Understanding Application Development Workflow by Recreating Twitter 334
Defining your application requirements 335
Defining your application architecture using a desired design 336
Setting up functional navigation and a wireframe for your application 345
SplashScreen 347
The Login screen 347
Main flow 348
Home 350
Discover 351
Notifications 351
Profile 352
Tweet 352
Mock data and style application screens including animations 352
Bringing Redux or MobX to your application and moving data mocks to a centralized state 357
Redux 357

Refactoring all actions 360
MobX 364
Using the Twitter API to work with real data 368
Summary 371
Chapter 12: Deploying Your App to App Store or Google Play 372
Deploying iOS apps and how it's done in React Native 373
Join the Apple Developer program 374
Creating a certificate for your device 374
Signing your app to run on a device 377
Enable App Transport Security 380
Configure Release scheme 381
Registering our App ID 383
Archiving your app 385
Deploying Android apps and how it's done in React Native 392
Generating a Signed APK file 392
Distributing your APK 394
Introducing fastlane - automate your deployment workflow 396
Get to know Microsoft CodePush and integrate it with your application 398
So, how do we get started 398
Setting up your mobile client 399
Summary 399
Index 401

Preface

The world of web development is diverse and complicated. New technologies are introduced each year, creating competition between various libraries and frameworks for their place under the sun. Some of these technologies have emerged as web development community answers to problems introduced by constantly evolving user needs. Some were introduced by big corporations, such as Facebook or Google.

For React Native, it all started as an internal hackathon project between the walls of Facebook offices, and since then, it has grown to become one of the most popular frameworks. React Native did something that web developers had tried to do for several years preceding the hackathon--writing mobile apps in JavaScript. There were several ways to write mobile apps, but all of them tackled the idea of hybrid apps.

The main idea was to write apps in HTML, JavaScript, and CSS, and put them inside a thin native container, which would be responsible for rendering all elements. This was all about one code base written using web technologies and rendered on both iOS and Android. Many companies used this idea in production apps, but soon, everyone who wrote mobile hybrid apps figured out that the performance difference was noticeable and multiple bugs led to frustration among web developers trying to break into the mobile development world.

React Native had a different idea in mind. Its idea was to use techniques used in the React library, which came a few years before React Native, and create modular components using JavaScript, but without HTML or CSS involved. React Native uses an internal mechanism to transform JavaScript to native modules. With it, we are dealing with real native apps and not hybrid ones. Performance-wise, apps became identical to native ones, even for complex animations and interactions. React Native also wraps lots of iOS and Android-core APIs and provides a simple bridging API to wrap your own modules or even integrate React Native app inside your existing app.

This book will start by explaining the difference between hybrid apps and React Native and why it's one of the most popular frameworks, and looks like it's going to be the future of mobile apps. It will explain how React Native works under the hood--how JavaScript is compiled to native modules, and how this “bridge” is architected. After understanding the basic ideas and the architecture of React Native, the book will follow with software installation and setting up a React Native project. It will cover different aspects of the setup for iOS and Android, as well as community solutions and tooling to make the app development experience better.

Then, the book will overview the basic React concepts used in React Native. It will ensure that readers feel comfortable with JSX syntax and understand what it represents. Following these introductory sections, the book will dive deeper into explaining every single React Native component, API, and technique in real-world examples such as WhatsApp, Instagram, and YouTube. It will teach you how to set up your animations, authenticate your app with Firebase or an external server, manage your application state with the best state management libraries such as Redux and MobX, build complex navigation systems in your app, test and debug your applications using built-in as well as community-written tools. The book will not only focus on React Native, but will introduce readers to the whole ecosystem of useful npm packages that can be used in React Native. It will cover how to properly link such packages with native application code and even write your own native code in Java or Objective C to be available inside JavaScript. At the end of the book, all this knowledge will be summarized by creating a major part of Twitter application--pulling actual Twitter data. In the final chapter of the book, readers will also learn how to release their app to App Store or Play Store, and learn about the tools available to speed up the release process.

This book targets JavaScript developers who want to learn how to create native mobile apps using React Native. The reader must have basic JavaScript knowledge, preferably of the latest version of JavaScript--ES2015. If the reader lacks such knowledge, it's advised to walk through a fast tutorial available at `https:/ / babeljs. io/ learn- es2015/ .`

What this book covers

`Chapter 1`, *Understanding Why React Native is the Future of Mobile Apps*, introduces React Native as to the JavaScript framework for building native mobile apps. It describes the advantages of writing apps in React Native as well as how React Native works under the hood.

`Chapter 2`, *Working with React Native,* teaches the reader the basic building blocks of React Native. It covers how to set up your environment and create basic layouts for iOS and Android. Readers will get familiar with JSX syntax and Stateful and Presentational components, and will get a taste of creating a basic one-screen app and running it in an emulator.

`Chapter 3`, *Getting Familiar with React Native Components*, goes through all the components React Native supplies to us. This chapter will cover the components that can be reused across both Android and iOS as well as platform-specific components.

`Chapter 4`, *Debugging and Testing React Native*, explores debugging and testing React Native components. It will cover basic debugging techniques to debug your application while it's running, and it will explain how to write basic unit tests for your app.

`Chapter 5`, *Bringing the Power of Flexbox to the Native World*, informs the reader of the concept of flexbox styling and how it's applied to styling React Native apps. It covers the differences between styling with flexbox inside a browser and in a React Native environment.

`Chapter 6`, *Animating React Native Components*, focuses on animating UI components with APIs supplied by React Native. It will start with simple layout animations and will dive into more complex animation examples used in real apps.

`Chapter 7`, *Authenticating Your App and Fetching Data*, outlines how to fetch data from a remote server and post data to it. Readers won't have to set up their own server, but will get familiar with Firebase as a solution to "server as a service" architecture. Readers will learn how to retrieve and post data to Firebase as well as how to authenticate against Firebase. They will also learn how to authenticate an app with popular social networks.

`Chapter 8`, *Implementing a Flux Architecture with Redux or MobX*, dives into the flux architecture pattern and will dive deeper into Redux, the functional state management library, and flux architecture implementation, which is an industry standard nowadays. The chapter will cover the building blocks of Redux and will guide the readers through the process of connecting Redux to their React Native application and managing their application state with it. It will also explore an alternative library for state management--MobX as a functional reactive way to deal with state management.

`Chapter 9`, *Understanding Supported APIs and How to Use Them*, teaches that it's important to understand which core APIs are supported in React Native. This chapter will cover most native APIs, such as various notification, informational, and image-related APIs. Readers will learn how to retrieve photos from camera roll, get geolocation, and respond to gestures. At the end of the chapter, readers will build a Tinder App card, swiping behavior with complex animations and the PanResponder API.

`Chapter 10`, *Working with External Modules in React Native*, focuses on lots of community packages that are considered standard and make React Native app development much faster. It will dive deeper into navigation with the react-navigation package and will explain how it can be integrated both with Redux and MobX; then, it will list famous open source packages that will speed up your development, enhance mobile API coverage, and enrich your application with prebuilt animations and interactions. Then, it will cover how you start writing your own native modules and connect them to the React Native world. Finally, it will discuss integration between existing native apps and React Native.

`Chapter 11`, *Understanding the Application Development Workflow by Recreating Twitter*, presents a real-world application, Twitter, and will discuss how to create this application from scratch. It will follow all the techniques the book has taught to summarize the process of creating a fully functional production app.

`Chapter 12`, *Deploying Your App to App Store or Google Play*, shows that deploying your application to either App Store or Play Market can be a challenge for web developers who are not familiar with this process. The chapter will cover deploying iOS and Android apps and using instant deployment services to make your deployment even better than that of native apps.

What you need for this book

Since React Native is used for creating both iOS and Android apps, and iOS apps have to be developed on OSX, you will have to have a Mac, ideally MacBook Pro with atleast 8 GB+ RAM. If you want to develop only for Android, you can still use Windows PC and React Native to create android apps; however, it's not covered in the book and may result in unexpected behavior. While you will be able to follow most of the book, it will require additional troubleshooting if there are any errors.

You will need an iOS and an Android phone, preferably the latest version.

Who this book is for

This book is for JavaScript developers who want to learn how to create native mobile apps using React Native.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "If we take a look at our `index.ios.js` file, we will see that our `render` method returns content wrapped in the `View` component".

A block of code is set as follows:

```
<View style={styles.container}>
   <Text style={styles.welcome}>
     Welcome to React Native!
   </Text>
   <Text style={styles.instructions}>
     To get started, edit index.ios.js
   </Text>
   <Text style={styles.instructions}>
```

```
        Press Cmd+R to reload,{'n'}
        Cmd+D or shake for dev menu
      </Text>
    </View>
```

Any command-line input or output is written as follows:

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

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<View style={styles.container}>
   <Text style={styles.welcome}>
     Welcome to React Native!
   </Text>
   <Text style={styles.instructions}>
     To get started, edit index.ios.js
   </Text>
   <Text style={styles.instructions}>
     Press Cmd+R to reload,{'n'}
     Cmd+D or shake for dev menu
   </Text>
 </View>
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors` .

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http:/ / www. packtpub. com`. If you purchased this book elsewhere, you can visit `http:/ / www. packtpub. com/ support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https:/ / github. com/ PacktPublishing/ React- Native- Building- Mobile- Apps- with- JavaScript`. We also have other code bundles from our rich catalog of books and videos available at `https:/ / github. com/ PacktPublishing/` . Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https:/ / www. packtpub. com/ sites/ default/ files/ downloads/ ReactNativeBuildingMobileAppswithJavaScript_ ColorImages. pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http:/ / www. packtpub. com/ submit- errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https:/ / www. packtpub. com/ books/ content/ support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Understanding Why React Native is the Future of Mobile Apps

Welcome to the first chapter. In this chapter, you will get a basic understanding of what React Native actually is and what it brings out of the box. Then, we will dive deeper and get familiar with how it works under the hood. You will learn the differences between the Native and React Native apps and understand that React Native brings with it lots of advantages over conventional mobile development for iOS and Android.

In this chapter, you will learn the following topics:

- What is React Native?
- How the React Native bridge from JavaScript to Native world works?
- What are benefits of React Native?

What is React Native?

React Native is a framework developed by Facebook for building Native mobile apps in JavaScript. It's based on ReactJS, a Facebook library for building user interfaces.

React introduced some amazing concepts to build UI, and these concepts can be applied not only to web browser, but also to mobile. They include better state management techniques, a unidirectional data flow in applications, component-based UI construction, and much more. If you have a React background, you should be familiar with these concepts, but if you don't, no worries. You will see how these concepts are used by following the book.

React Native currently supports iOS and Android and there are plans to expand to other platforms. The change that React Native brings with it is that even while an application is written in JavaScript, it's compiled to Native code, so its performance is much better than so-called **hybrid apps**. These apps are written in JavaScript, HTML, and CSS and are executed in **WebView** (a browser embedded inside an app). Besides, React Native brings web-like developer experience, live reloading of your application during development, and much more--things mobile developers only dreams about.

The history of React Native

In order to understand the motivation behind creating React Native, it's important to understand what lead to the creation of React Native and what was the motivation for it.

It all begun with ReactJS

Before looking at the history of React Native, let's take a look at the history of ReactJS. After all, React Native is based on ReactJs and uses its concepts and best practices. For years in web development, there was a mantra: *Dom Manipulation is expensive*. Basically, it means that presenting your data to the browser or rendering it is a costly operation. **Document Object Model** (**DOM**) was built in the 80s to deal with simple HTML documents.

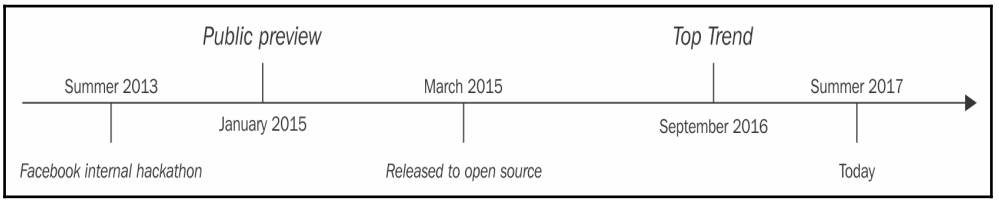
React is focused around the concept of Virtual DOM. Let's understand why this concept is called **virtual**. DOM stands for Document Object Model and means actual browser API to access HTML nodes structured in a tree-like structure. Virtual DOM is a representation of this same structure, but inside JavaScript without accessing HTML nodes directly.

React has an internal state, and when this state is updated, it knows to update DOM with batched updates. Also, React uses a **JSX-XML**-like syntax, which lets developers represent their React components hierarchies somewhat similar to writing XML, but doing it inside JavaScript.

Since React uses Virtual DOM, it's decoupled from exact browser implementation. This gave an idea to some developers at Facebook internal Hackathon to write a framework based on a solid React foundation, but that will render a UI not inside a browser, but instead will compile it to Native modules--Objective-C for iOS, and Java for Android.

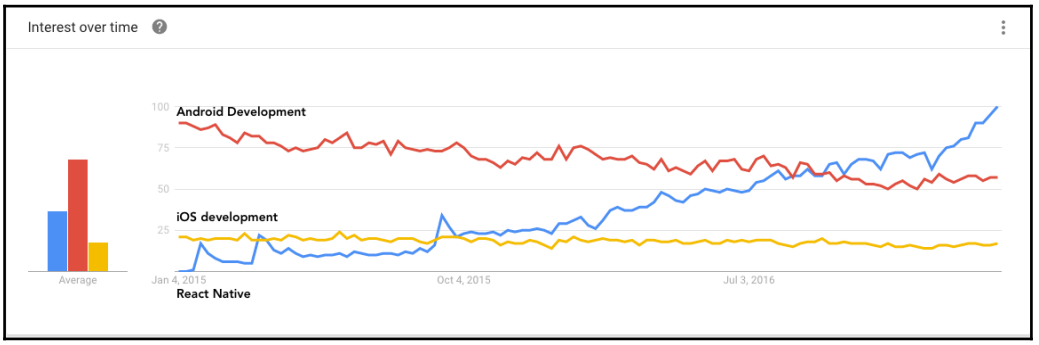
Hackathon, that grew bigger

Yes, you heard it right. Great things emerge when great minds start hacking. That was exactly the case in **Facebook internal Hackathon** in **Summer 2013**. After creating React Native, it was publicly reviewed in **January 2015** at the ReactJS conference. In **March 2015**, It was officially released by Facebook to open source and since then it has been available on GitHub:



Almost two years passed, and React Native grew to be the most popular and best framework for developing mobile apps. It even surpassed iOS and Android development according to Google Trends. It's a collective effort of React Native core team and the community, so after finishing the book and becoming a React Native professional, I encourage you to support this collective effort and contribute to React Native development.

Check out the following Google trends comparison between **React Native**, **Android development**, and **iOS development**:



React Native today

Today, React Native is considered to be the best choice for developing mobile applications since it gives you lots of advantages over traditional mobile development. It's not only a framework for external use. Facebook uses it for its Facebook groups, which is a combination of Native modules and React Native and for Facebook Ads Manager, which is built entirely using React Native.

The motivation of creating React Native

Apart from being a really cool *Hackathon* project, React Native had some solid foundation on why it was created in the first place. There were several things in mobile development that became a real bottleneck for companies creating mobile apps. Also, there were some things that the team behind React Native wanted to improve in mobile development workflow in general. Let's briefly look at them.

Mobile development is too specific

Let's talk about mobile development in general. iOS and Android are the two most popular platforms. Each platform uses a different programming language, has different APIs to deal with a device, and usually requires a different approach from the developer.

To summarize this, mobile development is coupled to a specific platform. If a company wants to create an Android application, it would need to hire an Android developer. If it needs to develop an iOS application, it will hire an iOS developer. These developers are not interchangeable. Application code, even when doing the same, can't be shared across platforms and basically needs to be written twice--once in Java, and once in Objective-C or Swift.

Taking the React approach

React introduced new concepts to web development, one of which is **unidirectional data flow**.

The meaning of unidirectional data flow this means can be expressed in the following formula:

$$\boxed{UI = f(state)}$$

It means that our UI is a function of an application state. That means that, instead of updating the UI directly in an imperative way, we change the state and the UI is updated accordingly by the internal React mechanism.

Let's check the following use case: A user clicks on a button, and it's color is updated. Usually, we attach some kind of event listener to the button click, and when it's performed, we update the button color directly. In React, instead of changing the button color directly, we change the component state and React propagates this change to the UI.

Another concept is splitting your UI to reusable components. Since you focus on each component in separation, it's easier to test and to reuse it across the app, meaning more **Don't Repeat Yourself** (**DRY**) code and less bugs.

The declarative nature of ReactJS components and their tree-like structure provide powerful feedback on how your application is laid out. This speeds up development, because code is easier to read and maintain.

The previously mentioned concepts were used for web development, but there were no reason not to reuse them for mobile development. So, these concepts were incorporated in React Native and used upto now. We will cover how this works in detail in later chapters.

Making developer experience state of the art

One of the important concepts in software development in general is a great developer experience. If your application is not working, you need to debug it. Also, when developing for mobile, each minor change means that you need to wait for your code to compile, restart your app, and watch whether it has changed. React Native was created with an idea of overcoming this short-coming to give the best developer experience, and it succeeded in lots of ways.

Since React Native is based on web development concepts, it allowed developers to use one of the most popular web development tools, Chrome Developer Tools. When you run React Native in the development mode, you can attach your application to Chrome browser and use its developer tools. You can debug your code, check network requests, profile memory usage, and do all the things you can do in Chrome DevTools when developing for the web.

Another new concept that has been introduced into web development and, with React Native was also got into mobile development, is a concept of Hot Module Reloading or HMR. HMR is a technique used by the JavaScript bundling system to "know" which modules were changed and reload them without doing a full page reload.

Introducing this together with live reload to the React Native ecosystem provides instant feedback during development and shortens development cycles.

In terms of layouting your application, layout engines for iOS and Android Native apps are different. React Native unites it under one way to layout your application using a modern web technique called **flexbox**. Also, you will see when we will get into the code that styles look very similar to inline styles used in web development.

How React Native is different from hybrid applications?

Was React Native the first framework used to build mobile applications in JavaScript? Of course not. For quite a while, there was the Cordova platform. On top of it, there were popular frameworks such as Ionic or Sencha Touch. Cordova created a new kind of application: hybrid apps.

What are hybrid applications?

Cordova gave a developer the ability to make calls to Native mobile APIs while writing their application using JavaScript, HTML, and CSS. The reason these types of apps are called **hybrid** is because they don't really create Native experiences. The resulting app runs inside a WebView as mentioned earlier and may look almost the same as a Native one, but there will always be small things that will matter, for example, scrolling, accessing keyboard, animations, and much more. There were lots of hybrid apps written and published to Play and App Stores, but none of them feel really Native.

So, you may ask, if they are all bad, then why were they created in the first place? There are several reasons:

- Code reuse across multiple platforms
- Development speed
- Web technologies used no platform-specific mobile developers

One of the main drivers of creating hybrid apps was code reuse. Write your code once in JavaScript, HTML, and CSS, and you are good to go. The application will work both on Android and iOS. Of course, it's undoubtedly faster to develop a single application for multiple platforms. For companies, hiring mobile developers for every platform was more costly than retraining web developers to develop for mobile.

Why Native applications lead the way?

When Cordova came out and hybrid apps started to emerge, it all looked very promising, but very fast hybrid apps showed their bad side.

They introduced new architectural complexities and more technologies to use in the project. While the motivation was to let developers create mobile experiences using web technologies, it was not as simple as that and always included involvement in Native technologies; thus, the overall experience was not the same.

Since hybrid applications are running inside a WebView, the level of interaction and performance is very limited. Animations can become laggy and do not compare to the Native approach.

Is React Native really a Native framework?

So how is React Native different? Well, it doesn't fall under the hybrid category because the approach is different. While hybrid apps are trying to make platform-specific features reusable between platforms, React Native has platform-independent features, but also has lots of specific platform implementations. Meaning that on iOS and on Android, code will look different, but somewhere between 70-90 percent of code will be reused.

Also, React Native does not depend on HTML or CSS. You write in JavaScript, but this JavaScript is compiled to platform-specific Native code using the React Native bridge. It happens all the time, but it's optimized in a way that the application will run smoothly in 60 fps.

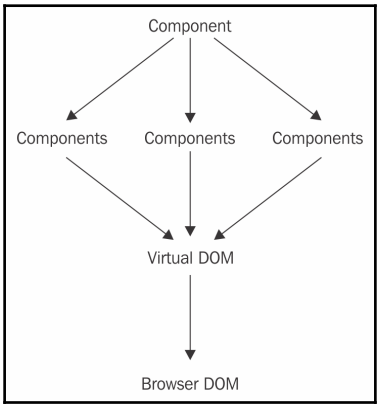
So, to summarize, React Native is not really a Native framework, but it's much closer to Native code than hybrid apps. Now, let's dive a bit deeper and understand how JavaScript gets converted into Native code.

How the React Native bridge from JavaScript to Native world works?

Let's dive a bit deeper and understand how React Native works under the hood, which will help us understand how JavaScript is compiled to a Native code and how the whole process works. It's crucial to know how the whole process works, so if you have performance issues some day, you will understand where it originates.

Information flow

So we've talked about React concepts that power up React Native, and one of them is that UI is a function of data. You change the state, and React knows what to update. Let's visualize now how information flows through the common React app. Check out the following diagram:



- We have a React component, which passes data to three child components
- What is happening under the hood is that a **Virtual DOM** tree is created, representing these component hierarchies
- When the state of the parent component is updated, React knows how to pass information to the children
- Since children are basically a representation of UI, React figures out how to batch **Browser DOM** updates and executes them

So now let's remove **Browser DOM** and think that instead of batching **Browser DOM** updates, React Native does the same with calls to Native modules.

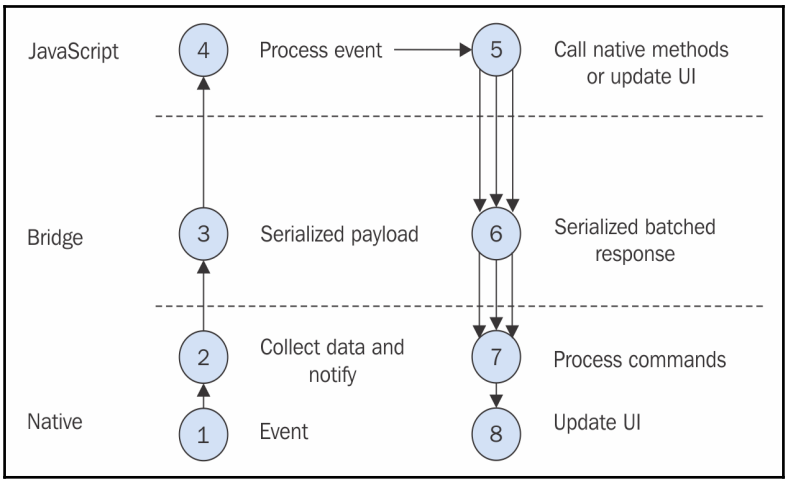
So what about passing information down to Native modules? It can be done in two ways:

- Shared mutable data
- Serializable messages exchanged between JavaScript and Native modules

React Native takes the second approach. Instead of mutating data on shareable objects, it passes asynchronous serialized batched messages to the React Native Bridge. The Bridge is the layer that is responsible for gluing together Native and JavaScript environments.

Architecture

Let's take a look at the following diagram, which explains how React Native architecture is structured and take a walk through the diagram:



In the preceding diagram, three layers are mentioned: **Native**, **Bridge**, and **JavaScript**. The Native layer is pictured last in the picture because it's the layer that is closest to the device itself. The Bridge is the layer that connects JavaScript and Native modules and is basically a transport layer that transports asynchronous serialized batched response messages from JavaScript to Native modules.

When an event is executed on the **Native** layer--it can be a touch, timer, or network request--basically, any event involving device Native modules. Its data is collected and is sent to the Bridge as a serialized message. The Bridge passes this message to the **JavaScript** layer.

The **JavaScript** layer is an event loop. Once the **Bridge** passes **Serialized payload** to **JavaScript**, **Event** is processed and your application logic comes into play.

If you update the state, triggering your UI to rerender, for example, React Native will batch **Update UI** and send it to the **Bridge**. **Bridge** will pass this **Serialized batched response** to the **Native** layer, which will process all commands that it can distinguish from a serialized batched response and will **Update UI** accordingly.

Threading model

Up till now, we've seen that there is lots of stuff going on under the hood of React Native. It's important to know that everything is done on three main threads:

- UI (the application's main thread)
- Native modules
- JavaScript runtime

The UI thread is the main Native thread where native-level rendering occurs. It is here, where your platform of choice, iOS or Android, does measuring, layouting, and drawing.

If your application accesses any Native APIs, it's done on a separate Native modules thread. For example, if you want to access the camera, geo location, photos, and any other Native API, planning and gestures in general are also done on this thread.

The JavaScript runtime thread is the thread where all your JavaScript application code will run. It's slower than the UI thread since it's based on a JavaScript event loop, so if you do complex calculations in your application that leads to a lot of UI changes, these can lead to bad performance. The rule of thumb is that if your UI changes slower than 16.67 ms, then your UI will appear sluggish.

The benefits of React Native

React Native brings with it many advantages for mobile development. We've covered some of them briefly before, but let's go over them now in more detail. These advantages are what has made React Native so popular and why it is trending now. Also, most of all, it helped web developers to start developing Native apps with a relatively short learning curve compared to overhead learning for Objective-C and Java.

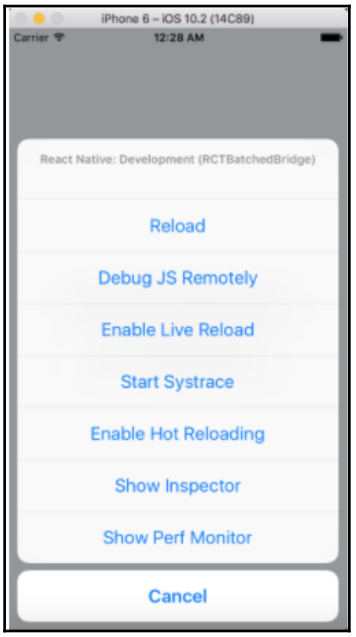
Developer experience

One of the amazing changes React Native brings to the mobile development world is enhancement of the developer experience. If we check the developer experience from the point of view of web developer, it's awesome. For a mobile developer, it's something that every mobile developer has dreamt of. Let's go over some of the features React Native brings for us out of the box.

Chrome DevTools debugging

Every web developer is familiar with **Chrome Developer tools**. These tools give us an amazing experience when debugging web applications. In mobile development, debugging mobile applications can be hard. Also, it's really dependent on your target platform. None of mobile application debugging techniques even come near the web development experience.

In React Native, we already know that a JavaScript event loop is running on a separate thread, and it can be connected to Chrome DevTools. By clicking on *Ctrl/Cmd+D* in an application simulator (we will cover all the steps of setting such a simulator in future chapters), we can attach our JavaScript code to Chrome DevTools and bring web debugging to a mobile world. Let's take a look at the following screenshot:



Here, you see a React Native debug tools. By clicking on **Debug JS Remotely**, a separate Google Chrome window is opened, where you can debug your applications by setting breakpoints, profiling CPU and memory usage, and much more.

The **Elements** tab in Chrome Developer tools won't be relevant, though. For that, we have a different option. Let's take a look at what we will get with Chrome Developer tools Remote debugger.

Currently, Chrome Developer Tools are focused on the **Sources** tab. You can note that JavaScript is written in ECMAScript 2015 syntax. For those of you who are not familiar with React JSX, you see a weird XML-like syntax. Don't worry, this syntax will also be covered in the book in the context of React Native.

If you put debugger inside your JavaScript code, or a breakpoint in your Chrome development tools, the app will pause on this breakpoint or debugger, and you will be able to debug your application while it's running.

Live reload

As you can see in the React Native debugging menu, the third row says *Live Reload*. If you enable this option, whenever you change your code and save, the application will be automatically reloaded.

This ability to live reload is something mobile developers only dream of. There is no need to recompile an application after each minor code change. Just save, and the application will reload itself in the simulator. This greatly speeds up application development and makes it much more fun and easy than conventional mobile development. The workflow for every platform is different, whereas the experience is the same in React Native--the platform for which you develop does not matter.

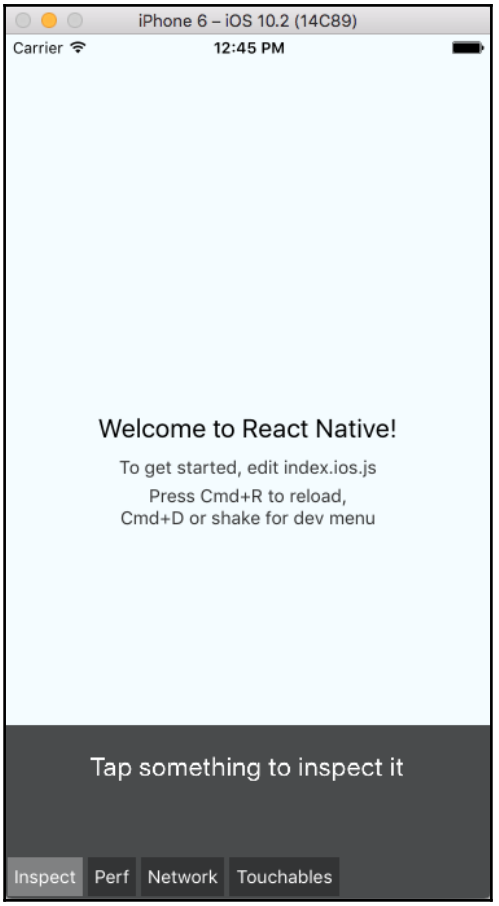
Hot reload

Sometimes, you develop a part of the application that requires several user interactions to get to. Consider, for example, logging in, opening the menu, and choosing an option. When we change our code and save, while live reload is enabled, our application is reloaded and we need to do these steps once again.

However, it does not have to be like that. React Native gives us an amazing experience of hot reloading. If you enable this option in React Native development tools and if you change your React Native component, only the component will be reloaded while you stay on the same screen you were on before. This speeds up the development process even more.

Component hierarchy inspections

I've said before that we cannot use the elements panel in Chrome development tools, but how do you inspect your component structure in React Native apps? React Native gives us a built-in option in development tools called **Show Inspector**. When you click on it, you will get the following window:



After the Inspector is opened, you can select any component on the screen and inspect it. You will get the full hierarchy of your components as well as their styling:

In this example, I've selected the **Welcome to React Native!** text. In the opened pane, I can see its dimensions, padding margin, and component hierarchy. As you can see, it's `IntroApp/Text/RCTText`.

`RCTText` is not a React Native JavaScript component, but a Native `Text` component, connected to React Native bridge. In that way, you also can see that this component is connected to a Native text component.

There are even more dev tools available in React Native that I will cover later on, but we can all agree that the development experience of React Native is outstanding.

Web-inspired layout techniques

Styling for Native mobile apps can be really painful sometimes. Also, it's really different between iOS and Android. React Native brings another solution. As you may have seen before, the whole concept of React Native is about bringing the web development experience to mobile app development.

That's also the case for creating layouts. The modern way of creating a layout for the web is using flexbox. React Native decided to adopt this modern technique for the web and bring it to the mobile world with a few differences.

In addition to layouting, all styling in React Native is very similar to using inline styles in HTML.

Let's take a look at an example:

```
const styles = StyleSheet.create({
container: {
  flex: 1,
  justifyContent: 'center',
  alignItems: 'center',
  backgroundColor: '#F5FCFF',
});
```

As you can see in this example, there are several properties of flexbox used as well as a background color. This really reminds us of CSS properties; however, instead of using `background-color`, `justify-content`, and `align-items`, CSS properties are named in a camel case manner in order to apply these styles to the `Text` component, for example. It's enough to pass them as follows:

```
<Text styles={styles.container}>Welcome to React Native </Text>
```

Styling will be discussed in the next chapters; however, as you can see from the preceding example, styling techniques are similar to web techniques. They are not dependent on any platform and are the same for both iOS and Android.

Code reusability across applications

In terms of code reuse, if an application is properly architectured (something that you will also learn in this book), around 80% to 90% of code can be reused between iOS and Android. This means that in terms of development speed, React Native beats mobile development.

Sometimes, even code used for the web can be reused in a React Native environment with small changes. This really brings React Native to the top of the list of the best frameworks to develop Native mobile apps.

Summary

So, we've discussed several things about React Native. We started by learning the motivation for creating React Native, that is, its history. Then, we dived deeper into how it works under the hood.

The whole explanation of React Native architecture and threads may sound intimidating, but I personally think it's important to understand how things work under the hood before starting to get into understanding bytes and bits of framework. To distinguish parallels between web and mobile and to explain the whole React Native bridge architecture in one sentence, I think the best way will be to say that React Native communication between JavaScript Runtime and Native modules is somewhat similar to client-server interaction: both send serialized data over some kind of transport. Finally, the chapter ends by covering some of the advantages of the React Native framework.

More and more companies have started to adopt React Native for their production apps. Facebook itself uses React Native for some of their applications, and the overall growth of React Native usage has lately become exponential.

It's important to understand that React Native bridges the gap between the Native mobile development and web development and removed lots of obstacles for web developers to enter the mobile development world. I hope that you believe that the future of mobile development is in learning React Native, and the fact that you are reading this means that you are ready to take this train to the future. Enjoy your ride!

What's next? I strongly believe that real knowledge comes from hands-on experience. So, in the next chapter, the first thing we will do is to set up a development environment so that you will be able to run your first React Native application. Then, you will get familiar with important concepts of React and React Native, and it will get you prepared for creating your first React Native applications.

2
Working with React Native

Welcome to the second chapter. We will start this chapter by setting our development environment for developing both iOS and Android apps. After that, we will set our environment, where we will learn the basics of JSX syntax and React libraries. We will get familiar with React component types and lifecycle methods and will cover the differences between React for web and React Native. We will also cover how structuring React Native apps can look somewhat similar to structuring HTML pages. This will build a solid foundation and prepare you for diving deeper into creating React Native apps in the subsequent chapters.

So, let's summarize what we will learn in this chapter:

- Setting up an environment for developing iOS and Android apps
- Introduction to JSX and how it's used in React Native
- Stateful versus presentational components
- React lifecycle methods
- Structuring React Native apps and their resemblance to HTML

We will use ES2015 or ES6 through the book since React is used in modern environments with the latest ES6 features. For ES6 reference please check MDN: `https://developer.mozilla.org/en-US/docs/Web/JavaScript`.

Setting up an environment for developing iOS and Android apps

Installation of React Native can be done in several ways: installing via `create-react-native-app` package or true native installation. If you check Facebook docs by default you will see installation using `create-react-native-app`. However, even though this installation is much faster, at some point you still need native dependencies so it's important to understand how they are installed. We will walk through installing them and later will overview `create-react-native-app` and **Expo XDE**.

Installing projects with native code

In this section, we will cover steps for setting up our development environment for both iOS and Android apps on macOS. All up-to-date instructions are found in the official documentation at `http://facebook.github.io/react-native`, under the tab *"Building Projects with Native Code"* but we will cover setting up our development environment here anyway. The reason the book only covers macOS setup is because macOS is the common development environment for React Native apps. iOS apps cannot be built on Windows or Linux machines, and most developers and companies want to use React Native to develop both Android and iOS apps at the same time.

In case you are interested in setting up your environment on Windows or Linux machine, detailed instructions are provided in the official docs at `https://facebook.github.io/react-native/docs/getting-started.html`.

For installing React Native dependencies for both iOS and Android, we will use **Homebrew** (`https://brew.sh/`)--a package manager for macOS. If you don't have it, use Homebrew installation instructions on the Homebrew site, but if you have it, make sure that it's up to date by running the following command:

```
brew update brew upgrade
```

If you will encounter errors in this command, it means that your Homebrew installation has been corrupted. By running **brew doctor**, you can pinpoint the problems with your local Homebrew installation.

After updating Homebrew, you need to install several React Native dependencies, as follows:

- Node.js
- Watchman

You've probably heard of Node.js, but not of Watchman. **Watchman** is a tool by Facebook for watching filesystem changes. It's really advised to install it for better performance.

The installation of these packages is also done via Homebrew by simply executing the following commands in the OS X terminal you use:

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

Note that if you already have Node.js installed, make sure that it's version is above 4.5 since it's the minimum required Node.js version for developing with React Native, however It's advised to use version 8 and above since 8 is entering Long term support at October 2017.

After installing all the packages, you are ready to install the React Native command-line interface. This is done by running the following command:

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

The Node.js that you've installed before comes with a **Node Package Manager** (**npm**), which we will use to install not only React Native but all our apps dependencies. Instead of npm, you can use a modern and faster package manager called **yarn** (`https://yarnpkg.com`). We won't cover its installation and usage in this book, but keep in mind that everything that is done with npm can be done with yarn with slight differences, for example, instead of running `npm install` we can run `yarn add`.

After installing React Native CLI tools that we will use later to create and run our applications, we need to take care of some platform-specific requirements. You will notice that there are fewer requirements for the iOS development than for Android. That is due to the fact that Android runs on a broader spectrum of devices, whereas iOS runs on a limited number of devices. Also, the fact that React Native was made available for iOS more than half a year earlier than for Android also contributes to this.

In case you encounter errors in installing node packages globally, you can either use nvm (`https://github.com/creationix/nvm`) to install node instead of using Homebrew or check the following npm doc in case you have permission problems: `https://docs.npmjs.com/getting-started/fixing-npm-permissions`

Installing iOS dependencies

In order to develop iOS apps, you will need to have an Apple development account. If you want to release your apps to the App Store, it will cost you around 99$ a year (rates may change, so make sure that you check them); however, if you want to develop only your iOS application, you can set up a free Apple developer account by following these steps:

1. By navigating to `https://developer.apple.com/` and clicking on the **Account** option in the menu, you will be prompted with your Apple ID credentials. If you don't have an Apple ID, you can create it by clicking on the **Create Apple ID** button:

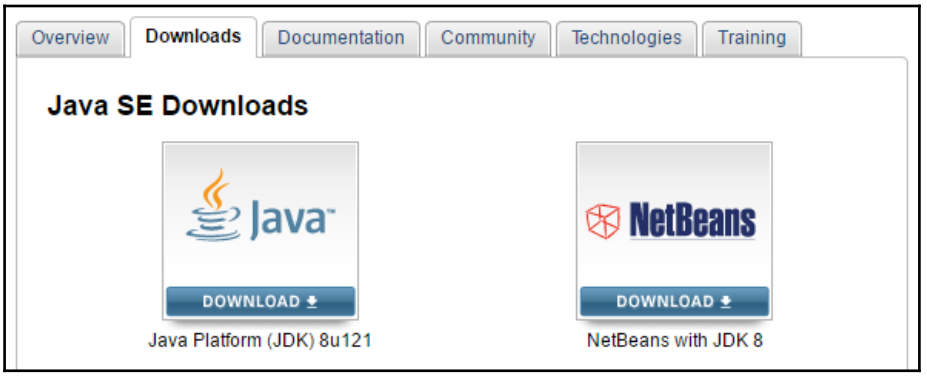
2. After you enter your credentials, you will see a license agreement, asking you to agree to the Apple developer license terms.
3. After accepting it, you can install XCode from the Mac App Store or find the download link on your account page in `https://developer.apple.com`. Xcode will install XCode IDE, iOS simulator, and all other tools needed to build and run your app.

Installing Android dependencies

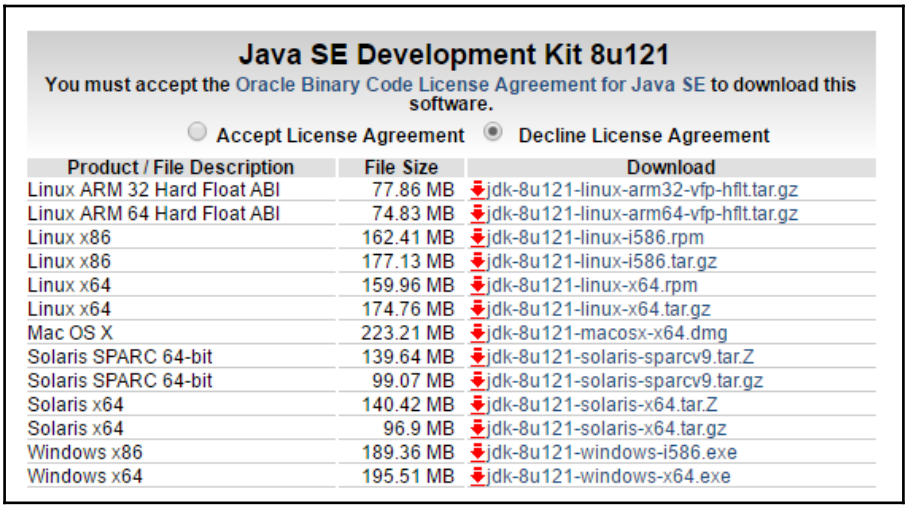
Android setup is a much more complex process. This book will cover multiple steps of installing Android dependencies and emulator, but it's important to check the official instructions on the React Native site, `https://facebook.github.io/react-native/docs/getting-started.html`. Again, you'll find it under the tab *Building Projects with Native Code*.

Installing JDK

The first requirement when developing Android applications is an up-to-date version of **JDK** (**Java SE Development Kit**). It can be downloaded by navigating to the following URL: `http://www.oracle.com/technetwork/java/javase/downloads`



By clicking on the **Java Platform (JDK)** icon, you will get to the download page, where you need to select **Accept License Agreement** and choose your operating system installer:



Java SE Development Kit 8u121

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement Decline License Agreement

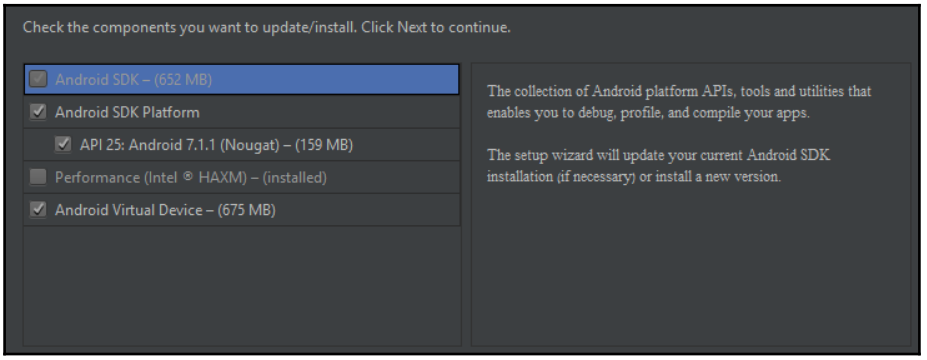
Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.86 MB	jdk-8u121-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.83 MB	jdk-8u121-linux-arm64-vfp-hflt.tar.gz
Linux x86	162.41 MB	jdk-8u121-linux-i586.rpm
Linux x86	177.13 MB	jdk-8u121-linux-i586.tar.gz
Linux x64	159.96 MB	jdk-8u121-linux-x64.rpm
Linux x64	174.76 MB	jdk-8u121-linux-x64.tar.gz
Mac OS X	223.21 MB	jdk-8u121-macosx-x64.dmg
Solaris SPARC 64-bit	139.64 MB	jdk-8u121-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.07 MB	jdk-8u121-solaris-sparcv9.tar.gz
Solaris x64	140.42 MB	jdk-8u121-solaris-x64.tar.Z
Solaris x64	96.9 MB	jdk-8u121-solaris-x64.tar.gz
Windows x86	189.36 MB	jdk-8u121-windows-i586.exe
Windows x64	195.51 MB	jdk-8u121-windows-x64.exe

After the download has finished, run the installer and you will get JDK installed on your system.

Installing Android studio

Navigate to `https://developer.Android.com/studio/` to download the latest version of Android studio. Android studio installation will install Android SDK and most other dependencies. We won't develop inside Android Studio, but we will use it to run the Android simulator.

After downloading the installer and installing Android studio, you will be asked whether you want to open it. When opening it for the first time, you will be asked whether you want the standard or custom setup. Ensure that you choose **Custom** and select all, as follows:



This contains following components:

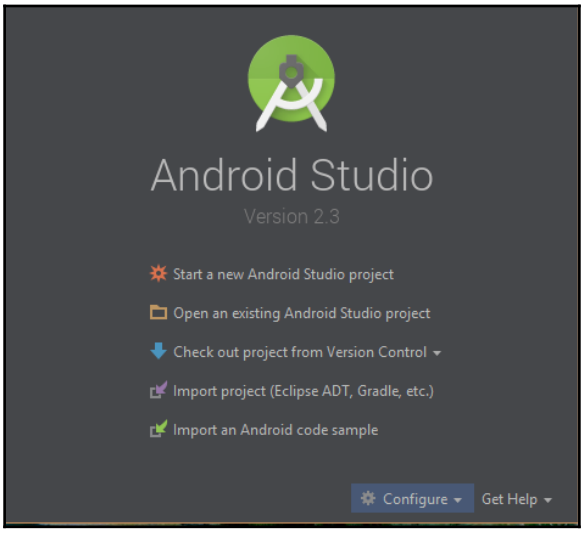
- **Android SDK**
- **Android SDK Platform**
- **Performance (Intel HAXM)**
- **Android Virtual Device**

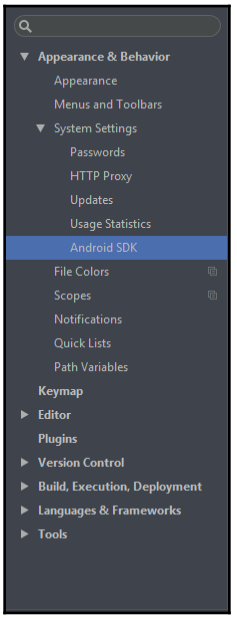
If you already have Android Studio installed before, some of its components will be marked as installed. Select all the remaining ones and click on **Next**.

In the preceding screenshot, you can see that under Android SDK Platform, Android Studio installs the latest Android API (Nougat, in my case). At the moment of writing this book, React Native required Android 6.0 (Marshmallow) SDK.

Installing SDK and built tools

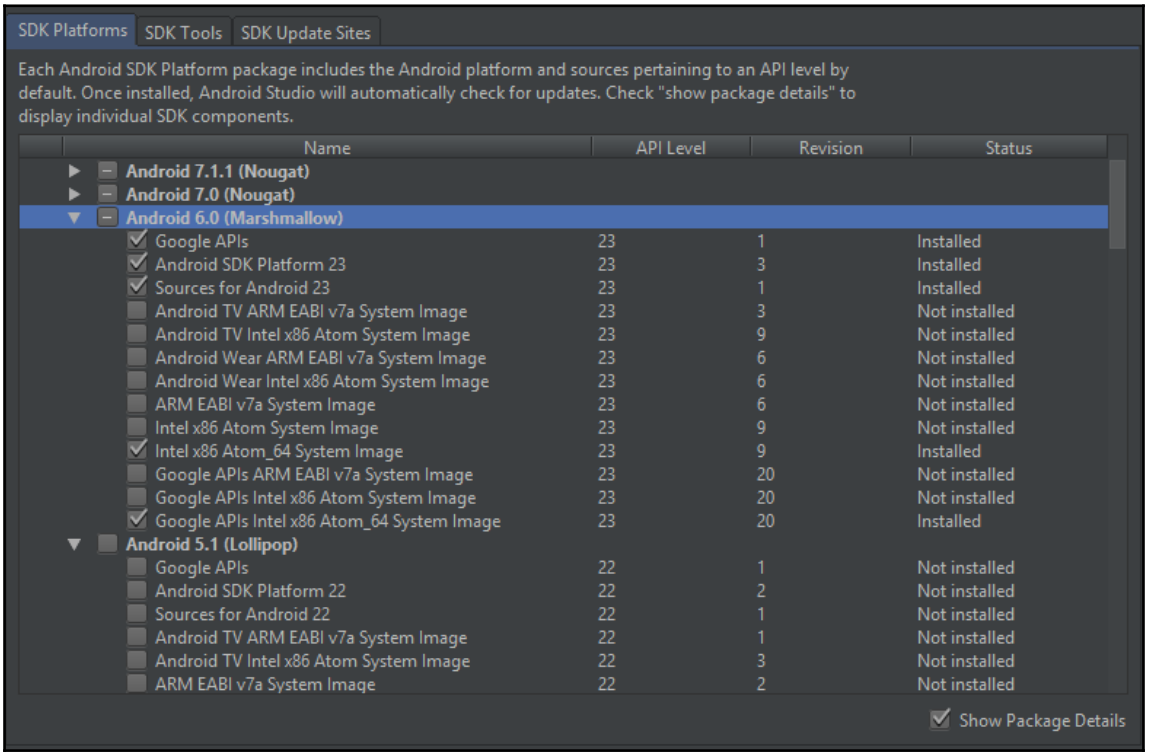
After the setup has been completed and **Android Studio** opens up, you will need to install Marshmallow SDK through the SDK Manager. It can be accessed through **Configure** | **Preferences** | **Appearance & Behavior** | **System Settings** | **Android SDK**:





You need to check the boxes next to the following under the SDK Platforms tab:

- **Google APIs**
- **Android SDK Platform 23**
- **Intel x86 Atom_64 System Image**
- **Google APIs Intel x86 Atom_64 System Image**



Under the SDK Tools tab, find **Android SDK Build Tools** and make sure that you select **Android SDK Build-Tools 23.0.1**. Ensure that you check **Show Package Details**, which is on the right-hand corner of the preceding screenshot.

Now, click on **Apply**, and SDK Manager will download and install all that you've selected.

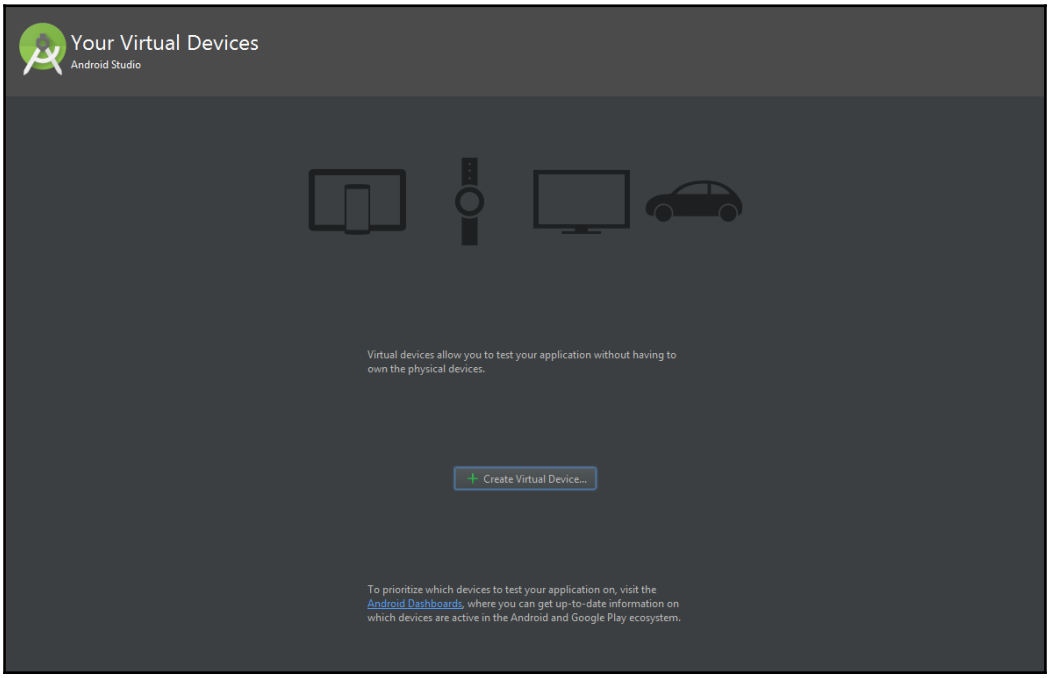
Setting up the ANDROID_HOME environment variable

The React Native CLI relies on the `ANDROID_HOME` variable to run an Android application. In order to set it up, you will need to add the following three lines to `~/.profile` or `~/.bashrc`:

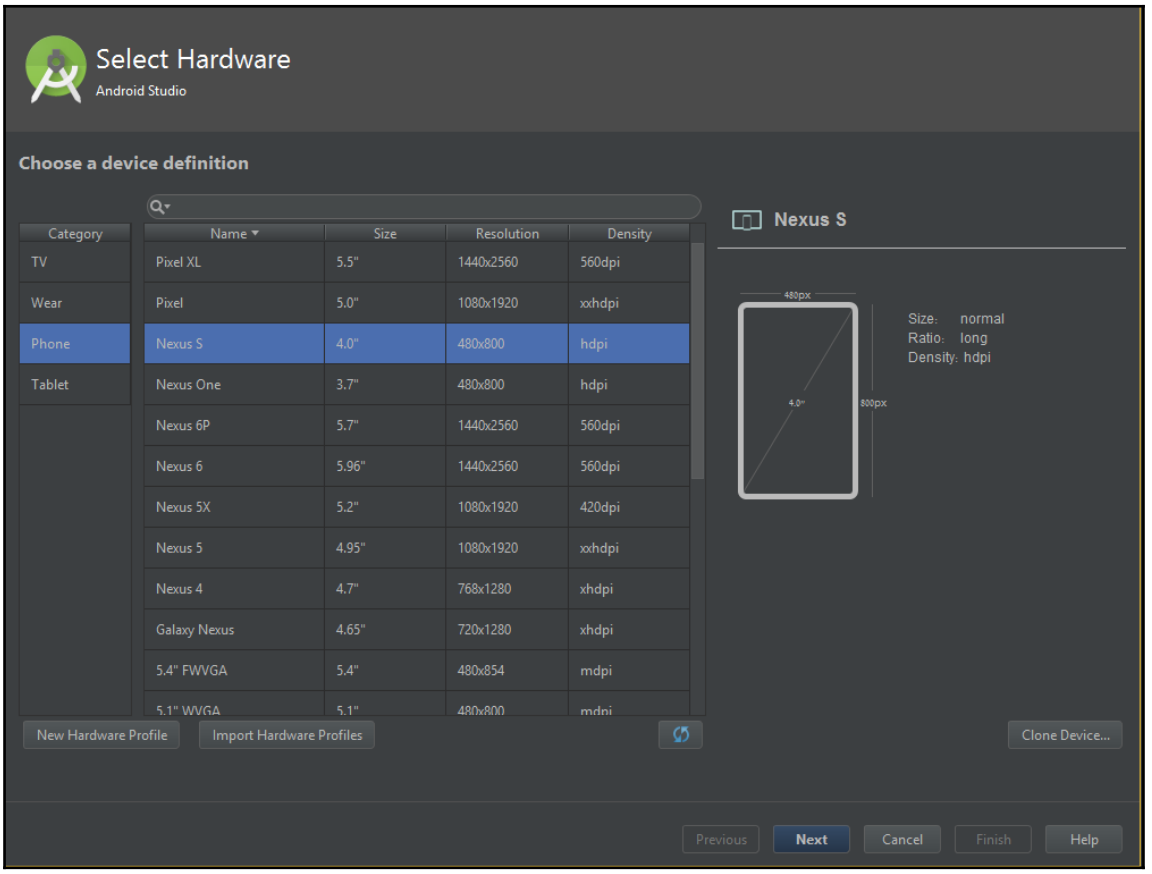
```
export ANDROID_HOME=${HOME}/Library/Android/sdk export
PATH=${PATH}:${ANDROID_HOME}/tools export
PATH=${PATH}:${ANDROID_HOME}/platform-tools
```

Creating an Android virtual device

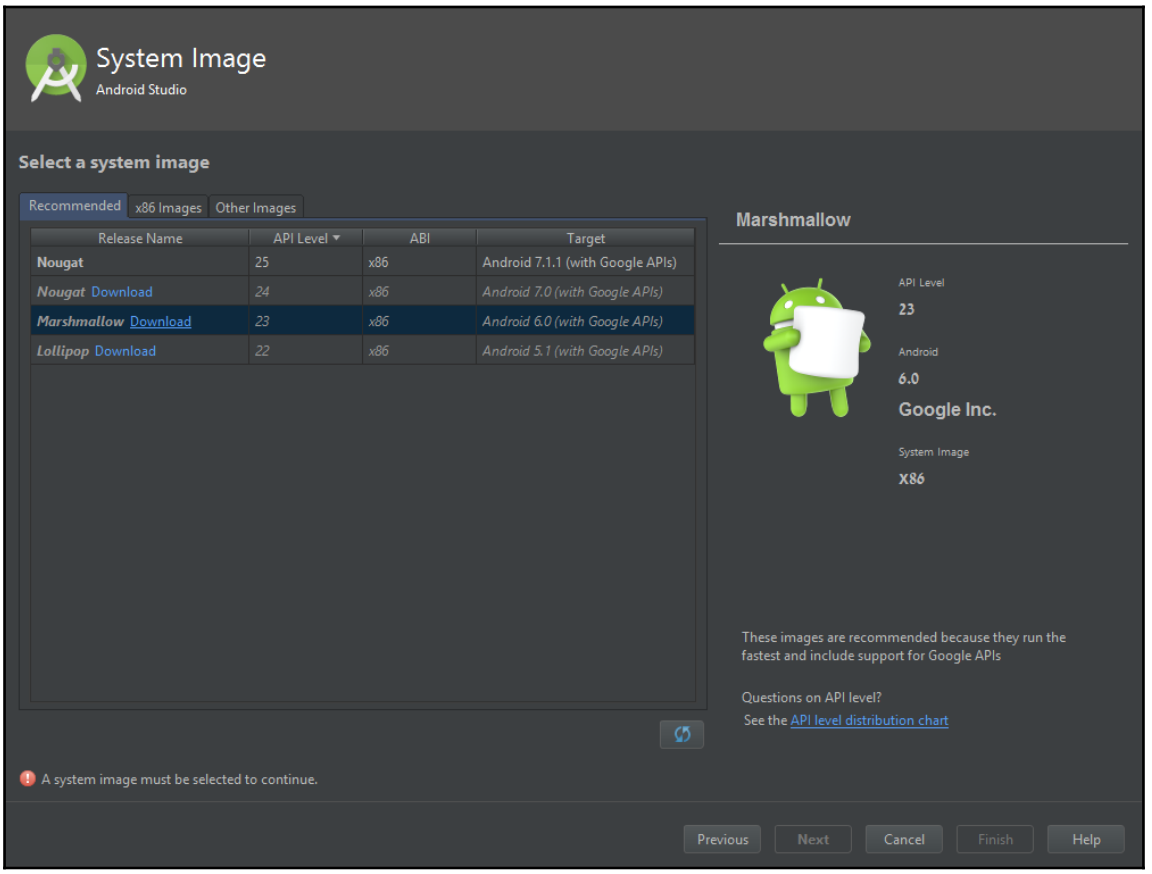
We want to be able to view our application on some kind of an Android Virtual device to speed up our development. We don't have to own an Android device to develop an application for it. Open your Android Studio and click on **Start a new Android Studio project**. It doesn't matter what you choose since the reason for creating a new project is just to get to main Android Studio menu toolbar. In the Android Studio menu toolbar, click on the  icon. This will open up the following window:



By clicking on **Create Virtual Device...**, you will get this screen asking which device you want:

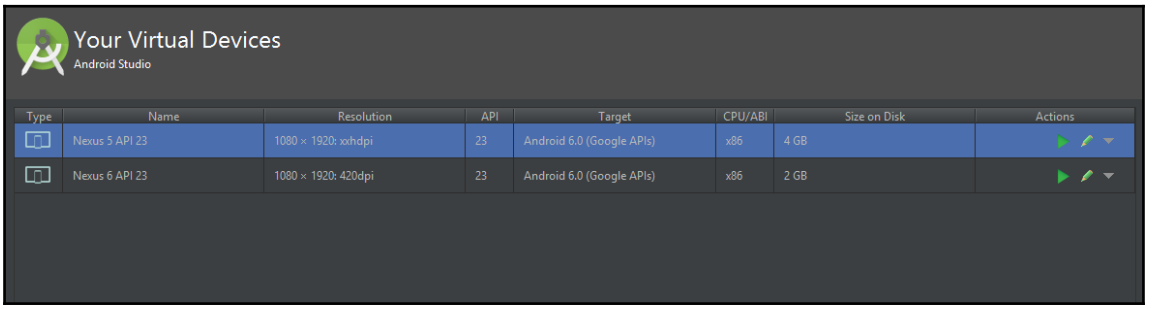


Select your device and click on **Next**. On the next screen, you will be prompted with **System Image** (the API version); select **Marshmallow**. AVD Manager also lets you download the API version if you haven't downloaded it yet:



If you encounter any errors during the AVD creation, check out the Android Studio official guide by following this link:
`https://developer.Android.com/studio/run/managing-avds.html`

Once your AVD has been created, it will be added to the AVD list. You will see the following screen:



There is another emulator commonly used in Android development and is much better than AVD. It's called **Genymotion** and you can check out how it can be set up by reading it's official documents here:

`https://www.genymotion.com`

The development environment

React Native can be developed in the text editor or IDE of your choice that supports modern JavaScript syntax and JSX syntax. In our examples, we will use the Nuclide IDE. It's a package on top of Atom. It's an open source editor that can be extended with multiple packages to have a similar feature set of IDE. The Atom editor can be downloaded at `https://atom.io/`.

You can read about Nuclide at `https://nuclide.io/`. Instructions on how to get started with it can be found at `https://nuclide.io/docs/quick-start/getting-started/`.

Also another commonly used tool for developing is VSCode with React Native tools package, which can be found here:

`https://marketplace.visualstudio.com/items?itemName=vsmobile.vscode-react-native`

VScode installation can be found here:

`https://code.visualstudio.com/`

Creating your application

Now, when we've set everything up, we can start creating React Native applications. At the beginning of this chapter when we've installed react native dependencies, we've installed the React Native command-line interface by running the following command:

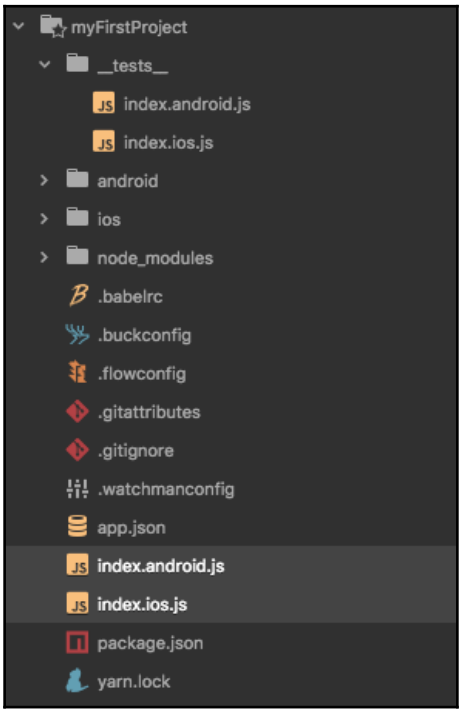
```
npm install -g react-native-cli
```

We will use it to both run our application and to create it.

First of all, run the following in the terminal:

```
react-native init myFirstProject
```

The preceding command will create a new folder in your current directory and will create a React Native project inside of it. Let's open up this folder and briefly take a look at what we've got:



In the following chapters, we will cover all the inner bits of this project structure, but you can note that we have the `__tests__` folder with two index JavaScript files for both Android and iOS.

The `android` and `ios` folders as seen in the picture above, contain Objective-C and Java code for iOS and Android native parts. The `node_modules` directory contains all installed npm packages. In the root folder there are various configuration files for babel, buck, flow, git, watchman, and yarn. Also, we have two index JavaScript files for both Android and iOS JavaScript entry points.

All this structure was created for us by the React Native CLI.

Now, let's use it to run our actual application. This is done simply by typing in one of the following two commands in the terminal:

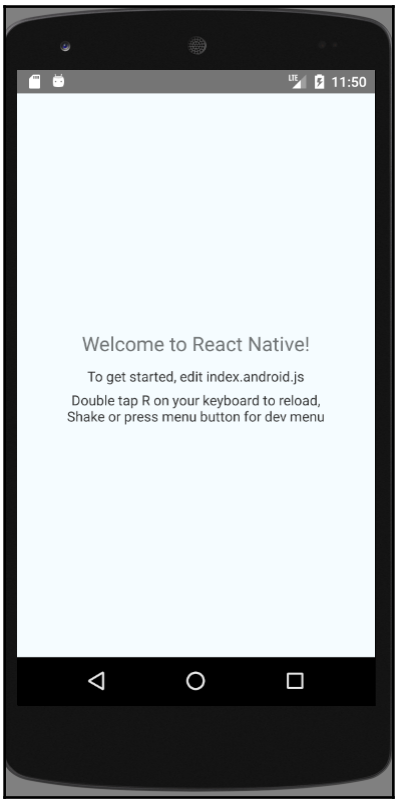
```
react-native run-ios
react-native run-android
```

The React Native CLI will run the packager, which is in charge of bundling your JavaScript files and will launch your emulator.

For Android, you need an emulator that is running; before running the React Native CLI.

Note that you need the packager running at all times while developing, so your code changes will be reflected in the application. If at some point packager crashes, you can rerun it by running `react-native start` from your project folder.

After launching your React Native app, this is what you will see in the end:



Application with create-react-app and Expo

Installing all native dependencies is a tedious, but a must process for developing your app. However, at some point you want to create your app fast and debug it on your physical device without walking through the tedious process of setting this up. For that use case there is a `create-react-native-app` package. This package is built on top of Exponent--a platform that extends React Native with additional functionality and APIs.

When using create-react-native-app you can run your application both on simulator or on real device. Make sure to install Expo XDE by following steps described in official docs:

`https://docs.expo.io/versions/latest/introduction/installation.html`

In order to create your React Native app, you first run the following:

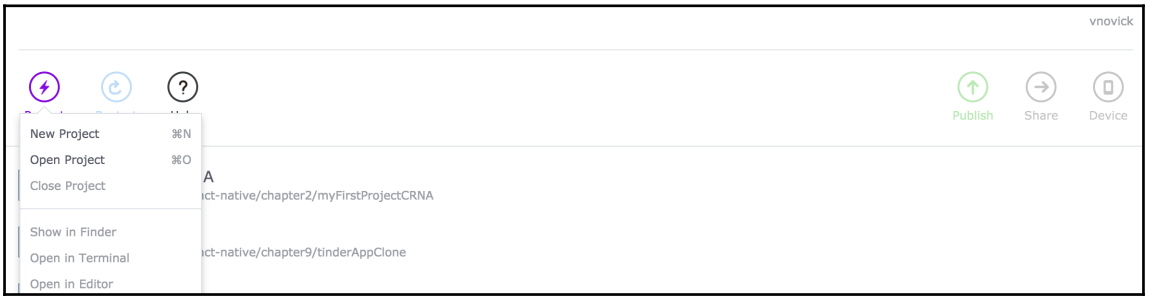
```
npm i -g create-react-native-app
```

Then you can create your app by running:

```
create-react-native-app myFirstProjectCRNA
```

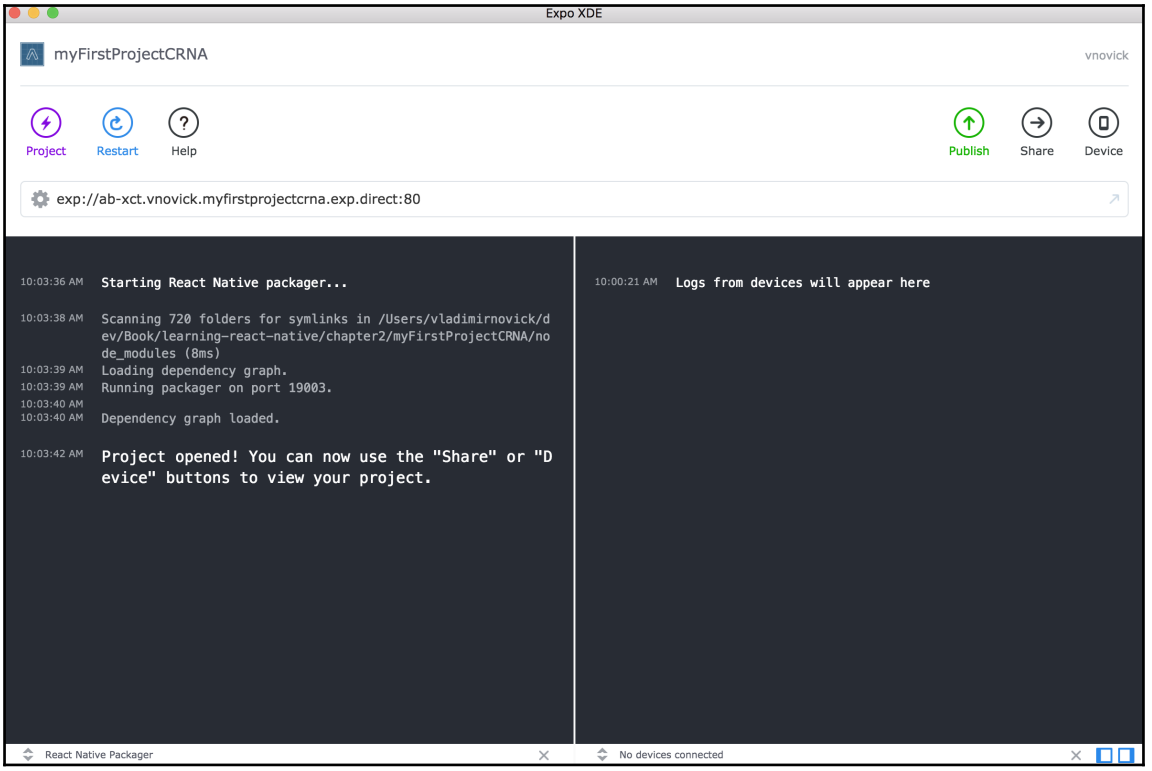
Now we can open our Expo client.

We will get the main XDE window where you can select recent projects or open them. Click on **Project** | **Open project**



When clicking on it, Finder will open up and choose there **myFirstProjectCRNA** - this folder was created when you ran create-react-native-app command.

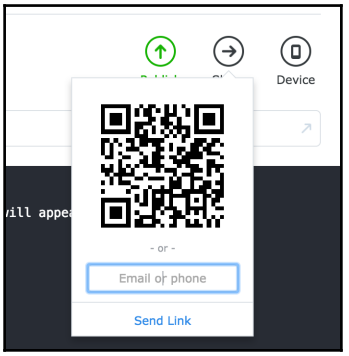
You will get the following window



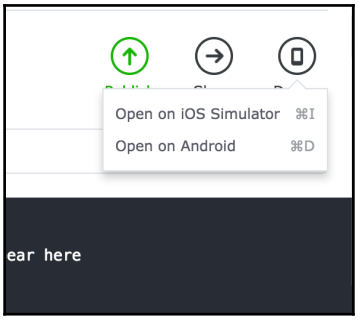
When opening your project, XDE will automatically start React Native packager. In right window you will see various logs while on the left you will see logs from device. In case React Native packager crashes, you can restart it by clicking on Restart button.

Now you have two options.

- By clicking **Share** on the right you will get a QR code, which you can read using Expo app installed on your device



- By clicking Device on the right, you can choose whether to run your project in emulator. We will click on **Open on iOS simulator** which will open iOS simulator and run our app.

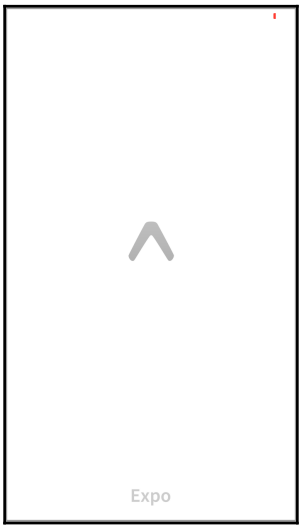


While XDE provides lots of benefits such as debugging on real device without additional overhead or by providing good packager management via Restart options and much more, you can alternatively run your application from project folder by using the following command

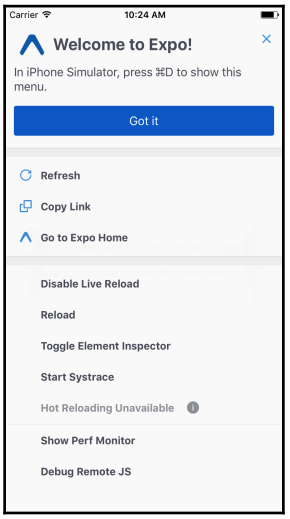
```
npm run ios
```

This will provide QR code in terminal, so you could open an app on real device using Expo app and will open iOS Simulator.

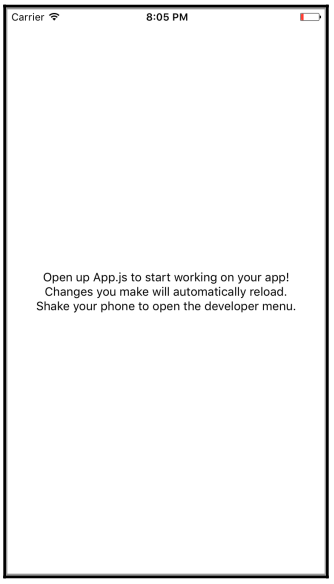
As you can notice, on the right you will get a prompt if you want to open your app in Expo. When you click open, you will see that Expo app is installed:



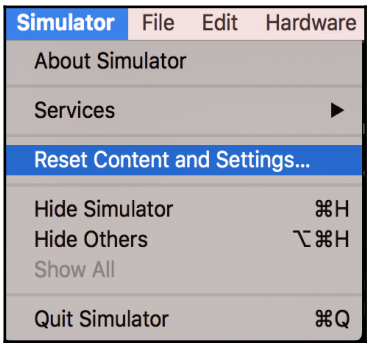
And then you will get the following Welcome screen and dev menu:



By clicking on **Got It**, you will see your actual app:



Note that if your simulator was closed incorrectly previously, you may experience "Failed to start simulator" error. In such case you need to go to Simulator Menu and **Reset Content and Settings**



It's much more convenient to use XDE to run your app if you use **create-react-native-app.** In addition to features described above there is much more you can use with XDE, so make sure to check official Exponent docs `http://docs.expo.io`

Also refer to exponent docs for troubleshooting in case you have errors. Exponent is constantly in development and is improving from day to day, so it's important to always check the docs

While Expo is great for fast prototyping and writing your app when only JavaScript code is involved, usually you will at some point add packages with native dependencies and as a result you will have to use React Native. `create-react-native-app` provides a mechanism of ejecting from Exponent by running `npm run eject` and continuing to run your application with `react-native run-ios` or `react-native run-android`.

Introduction to JSX and how it's used in React Native

In `Chapter 1`, *Understanding why React Native is the future of mobile apps*, I've mentioned that React Native came from ReactJS and that the ability of React to use Virtual DOM is one of the main features that led to the creation of the React Native framework. Let's recap the concept of Virtual DOM. Virtual DOM means storing DOM nodes or components inside JavaScript instead of accessing the real DOM.

In React Native, we don't have DOM. Also, our app components structure may look similar to HTML syntax, but that is due to JSX.

Let's look at an example. When we set up our environment and created our first application, there were two files that I've mentioned:

```
index.ios.js
index.android.js
```

Open the file relevant to the environment that you plan to develop. Basically both files will look the same, so let's take a look at the code of either of them:

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

React Native uses ECMAScript 2015 (ES6) spec for writing modern JavaScript.

In case you do not know ES6 it's advised to go through the following tutorial before continuing with the book: `https://babeljs.io/learn-es2015/` . In case you want more information on a specific feature, you can always check out **Mozilla Developer Network** (**MDN**) at `https://developer.mozilla.org`, or else you can check out **devdocs** `http://devdocs.io`. There, you can find information on not only the latest JavaScript features, but on React Native, React, and lots of other libraries. This site combines lots of documentation in one single searchable site.

In our preceding code, we can see that React and React component are imported from the React library. Then, we will see several other imports from the React Native library.

These important components are the base layout component that we will use to construct our application. We will cover these components and the creation and composition of React components themselves.

What we can see later on in the code is a strange combination of XML-like syntax and JavaScript:

```
render() {
 return (
  <View style={styles.container}>
    <Text style={styles.welcome}>
      Welcome to React Native!
```

```
        </Text>
        <Text style={styles.instructions}>
          To get started, edit index.ios.js
        </Text>
        <Text style={styles.instructions}>
          Press Cmd+R to reload,{'n'}
          Cmd+D or shake for dev menu
        </Text>
      </View>
    );
  }
```

This is JSX. Let's dive a bit deeper to understand what we can do with it.

What is JSX?

The following quote is written in the Facebook spec:

> *JSX is a XML-like syntax extension to ECMAScript without any defined semantics. It's NOT intended to be implemented by engines or browsers. It's not a proposal to incorporate JSX into the ECMAScript spec itself. It's intended to be used by various preprocessors (transpilers) to transform these tokens into standard ECMAScript.*

What we can understand from it is that JSX is intended to be used by preprocessors to transform HTML-like text found in JavaScript files into standard JavaScript objects that a JavaScript engine can parse.

As we also know from `Chapter 1`, *Understanding why React Native is the future of mobile apps*, these JavaScript objects are passed to the React Native bridge and are translated into actual native components.

If we look at the preceding example, we will see this:

```
<Text style={styles.welcome}>
  Welcome to React Native!
</Text>
```

`Text` and `View` are components imported from the React Native library. JSX is transpiled into React code, which creates a hierarchy of components.

Let's take a look at how the preceding JSX will look like after the transpilation:

```
React.createElement(
     Text,
     { style: styles.welcome },
     "Welcome to React Native!"
);
```

No magic as you can see, but it has a much better readability with JSX, especially when dealing with component hierarchies--for example, wrapping the `<Text>` component inside the `<View>` component like this:

```
<View>
  <Text style={styles.welcome}>
     Welcome to React Native!
  </Text>
</View>
```

This will be transpiled into the following:

```
React.createElement(
 View,
 null,
 React.createElement(
  Text,
  { style: styles.welcome },
  "Welcome to React Native!"
 )
);
```

The preceding code is less readable. Now, imagine complex hierarchies of several component levels--they are much more readable in JSX.

Children in JSX

The main JSX feature that allows component composition is that children can be passed to a React component. That is the case with the `View` component we've seen before:

```
<View>
  <Text style={styles.welcome}>
    Welcome to React Native!
  </Text>
</View>
```

Let's create our component and encapsulate the fact that the `Text` component is wrapped with the `View`:

```
class Container extends Component {
 render(){
  return (
  <View>{ this.props.children }</View>
  )
 }
}
```

Now, we can easily switch `View` to `Container` and get the same result:

```
<Container>
  <Text style={styles.welcome}>
    Welcome to React Native!
  </Text>
</Container>
```

JSX props

React has a system to pass data from one component to another. This ability is crucial for creating component hierarchies. As you have already seen in the `Text` component example, the following part of code looks a bit weird:

```
style={styles.welcome}
```

You have a style attribute and it is equal to some value wrapped in curly braces. Also, you may have noticed that, when we created the `Container` component, we passed to View `this.props.children` wrapped in curly braces. Let's address two use cases.

When passing style `={}`, we don't really create a style tag. We are not in the context of HTML, as it may seem. We pass a prop to the React component. This prop will be available inside the component by accessing `this.props.propName`. Children is the default prop passed to the React component as its reference component children. In our case, Container child is the `Text` component.

Let's do a little exercise. Let's--instead of `Text`--create the `Welcome` component that will get a welcoming message text as a prop and print it to screen:

```
<Container>
  <Welcome style={styles.welcome} text="Welcome to React Native!"/>
</Container>

class Welcome extends Component {
 render(){
   return (
    <Text style={this.props.style}>{ this.props.text }</Text>
    )
 }
}
```

Let's summarize:

- Props are passed to a component in a similar way to the way HTML attributes are passed, but the prop value should be wrapped with curly braces or in quotes for strings. If values are wrapped in curly braces, they can be JavaScript variables and expressions.
- Inside the JSX hierarchy, you can pass a JavaScript expression by putting them in curly braces similar to how `this.props.text` is wrapped in the preceding example. They also can be both variables and expressions.

There is much more in JSX, and we will address things as we go, but if you want to read an in-depth guide on JSX, I advise you to check out the following Facebook document for advanced JSX features:
`https://facebook.github.io/react/docs/jsx-in-depth.html`

Stateful versus presentational components

So we've seen that we can compose React Native components as well as create our own components and pass data from one to another. In React, there are two important component types, so let's understand what they are and how they are different, but first let's understand the terminology:

- **Stateful**: When something is stateful, it is a central point that stores information in memory about the app or component's state. It also has the ability to change it.
- **Stateless**: When something is stateless, it calculates its internal state but it never directly mutates it. This allows for complete referential transparency, meaning that given the same inputs, it will always produce the same output.

I've mentioned in `chapter 1`, *Understanding why React Native is the future of mobile apps*, how information is flown through the React application. We've seen how it's done via React props, but we haven't seen what the React state looks like and how to manipulate it. However, before looking at React state, let's understand why the React needs stateful and presentational components.

Presentational components

Most parts of UI are a function of received data. For example, a list of all your WhatsApp conversation is a function of data received from the server. This list item is also a function of data it receives, but this time not from the server, but from its parent in the list.

So if we look at the UI of any application, it consists of small blocks of UI that can be reused. Let's look at the welcome message example we've created:

```
class Welcome extends Component {
  render(){
   return (
     <Text style={this.props.style}>{ this.props.text }</Text>
   )
 }
}
```

In our case, it's simply a wrapper of the React Native `Text` component, but let's alter it a bit to be useful:

```
class Welcome extends Component {
  render(){
    return (
      <View style={styles.welcomePanel}>
        <Text style={styles.welcomeText}>
          { this.props.text }
        </Text>
      </View>
    )
  }
}
```

In this case, it has its own style, hence, the component can be reused in any place we need the welcome message. As you can see, this component doesn't have any state inside of it. This type of component is called **presentational component**. It's also sometimes called **functional component**. Also, this is due to the fact that this component is a pure function (function that does not mutate it's arguments) of props it receives from the parent. Writing this component as class extending--`React.Component`, in this case--is unnecessary.

Let's write it down as a pure functional component:

```
function Welcome(props){
 return (
 <View style={styles.welcomePanel}>
   <Text style={styles.welcomeText}>
   { props.text }
   </Text>
 </View>
 )
}
```

With the help of ES6 arrow functions and function argument destructuring, it can also be written as follows:

```
const Welcome = ({text}) => (
 <View style={styles.welcomePanel}>
   <Text style={styles.welcomeText}>
    {text }
   </Text>
 </View>
)
```

Stateful components

In the React world, the stateful component is the component that is in charge of changing the state and passing it down to props. It encapsulates UI logic in it and pass it to the children. Such a component can be reused anywhere in the application and will bring its logic with it. You can think of it as of widget of sorts.

Let's understand how such a component is created by walking through an example. We will create a simple timer that will tick every second.

We will begin with a basic structure of every React component:

```
class Timer extends Component {
 render(){
  return (
   <View>
     <Text>Timer</Text>
   </View>
  )
 }
}
```

Then, let's set our initial state. It can be done in the following two ways:

1. Setting it in the constructor:

```
constructor(props){
 super(props);
   this.state = {
     seconds: 0
   };
}
```

2. Using class properties. Class properties are a stage 2 proposal for future JS versions, but they are already included in React Native by default:

```
class Timer extends Component {
 state = {
  seconds: 0
 }
};
```

Now that we have seconds in the state, let's add two getters to ease getting hours and minutes based on the second's value:

```
get minutes(){
  return (this.state.seconds / 60).toFixed()
}

get hours(){
 return (this.state.seconds / 3600).toFixed()
}
```

Also, let's reference all three--seconds, minutes, and hours--inside our render function:

```
render(){
 return (
  <View>
    <Text>{`${this.hours}:${this.minutes}:${this.state.seconds}`}</Text>
  </View>
  )
 }
```

Here, we use an ES6 template strings to combine hours, minutes, and seconds values into the `0:0:0` format.

We have only one thing missing though. We need to somehow update our state. In React, the state is immutable and can be updated only by executing the `setState` function, which exists inside the React component.

In order to update seconds, we will need to do the following:

```
setInterval(()=>{
 this.setState({
  seconds: this.state.seconds + 1
  })
}, 1000;
```

Every second increase the seconds state by one.

There is one thing left to get a fully functional timer. We need to put `setInterval` somewhere. Ultimately, we need to start our interval once when component is rendered. After it starts, every second state will be updated with a new seconds state, and React will re-render the UI for us and figure out how to update it. If, for example, our seconds state will be passed deeper in the component hierarchy using props, React will update only those parts of the UI that are impacted by the seconds state changes.

Being able to set our interval after the React component is rendered brings us to the important topic of React lifecycle methods.

React lifecycle methods

When creating a stateful component in React, along with the `setState` method there are different lifecycle hooks that come together with the React component class. Let's take a look at the important ones.

Mounting

When a component is rendered to the page, it's called mounting. There are several methods in a React component that gives us the ability to execute custom logic when the component is mounted:

- `componentWillMount`: This method is invoked by React immediately before a component is mounted. It's called even before the `render()` method. There are very limited use cases for this since even in Facebook document it's advised to put things inside a constructor instead, if some pre-mounting logic is needed. However, it's important to know that this lifecycle hook exists. Note that changing state inside this method will lead to unexpected behavior since it won't trigger re-rendering.

- `componentDidMount`: This method is called on after a component is mounted. Initialization of timers, event listeners, fetching data, and everything that needs to be done on initial mounting of a component will go there. Triggering `setState` here will call re-rendering.

- `componentWillUnmount`: This method is invoked before the component is unmounted and destroyed. We will use this method for general cleanup, for example, getting rid of timers, canceling network requests, and so on.

Updating

As we've discussed before, components gets updated when their state is changed or their parent passes different props to the component. We can hook into this mechanism and execute our logic before a component gets updated. The following is the list of life cycle hooks dealing with updates:

- `componentWillReceiveProps`: This useful method is used mostly when you want to trigger a state change as a result of a props change. React may call this method if props haven't changed, so common a use case is to compare current and next props values like this:

```
componentWillReceiveProps(nextProps){
   If (nextProps !== this.props){
   // do something
   }
}
```

- `shouldComponentUpdate`: React usually re-renders your component on every state change; however, if you want to optimize your application and re-render only when state or props are different, you can use this lifecycle hook. Returning `false` from this method will lead to the component not being re-rendered. There are several other lifecycle hooks, which you can look up at `https://facebook.github.io/react/docs/react-component`. So, now after learning or refreshing your knowledge about notable React component lifecycle hooks, let's get back to our timer example and find a place to put in `setInterval`, which will update our state and ultimately lead to a working timer:

```
componentDidMount(){
 this.timerInterval = setInterval(()=>{
  this.setState({
   seconds: this.state.seconds + 1
```

```
      })
    }, 1000)
  }
```

As you can see here, we will put the preceding code, setting the interval inside the `componentDidMount` method since we will need to set the interval only once upon component mounting.

The second step is to clear the interval when the component will be destroyed:

```
componentWillUnMount(){
 clearInterval(this.timerInterval)
}
```

The following is the full code of the functional timer:

```
class Timer extends Component {

 constructor(props){
   super(props);
   this.state = {
     seconds: 0
   };
 }

 componentDidMount(){
  this.timerInterval = setInterval(()=>{
    this.setState({
     seconds: this.state.seconds + 1
    })
  }, 1000)
 }

 componentWillUnMount(){
   clearInterval(this.timerInterval)
 }

 get minutes(){
  return (this.state.seconds / 60).toFixed()
 }

 get hours(){
  return (this.state.seconds / 3600).toFixed()
 }
  render(){
   return (
     <View>
       <Text>{`${this.hours}:${this.minutes}:${this.state.seconds}`}</Text>
```

```
        </View>
      )
    }
  }
```

Structuring React Native apps and their resemblance to HTML

Up until now, we've looked at how React Native apps work under the hood, how React works, how to create both stateful and presentational components, but we have not yet discussed another way of thinking that React brings with it to mobile app development through React Native.

Thinking in React

React is heavily trying to promotes the idea of reusable components and composability. What does this mean in terms of app development? It means that when starting to develop an application, you need to look at the design and break it down to small pieces. Consider that every component should be responsible for only one thing. It's often called the **single responsibility principle**. The ultimate idea is to break your app to smaller pieces as much as you can and with reusability in mind.

Reusable components

You want your application to consist of lots of smaller components that you can reuse. That will be exactly your 80-90% of cross-platform code reuse. They don't have to be all presentational. If you have a component that is responsible for only one thing and can be used in several places in your application, go with that.

Containers

Containers are usually components that don't have any UI, but they are wrappers around blocks of UI, and they give those UI blocks their functionality. Usually, containers encapsulate logic inside of them and pass data/callbacks to child presentational components.

Basic React Native components

Agreeably, the JSX syntax looks somewhat similar to the HTML syntax, and though it's relevant to the web development world, in the native world, it can look a bit strange. After all, you don't have a `<div>` or `<img>` tags in Native applications. Instead, we have different base elements. Let's cover them, and draw some parallels between HTML elements and React Native ones:

- `<View/>`: This is a fundamental component in React Native. Basically, it's a container for anything in your application. It's usage is pretty similar to the `<div/>` usage in web development.

- `<Text/>`: This is similar to `<span/>` in web development.

- `<Image/>`: This is used to display various types of images, so is similar to `<img/>` on web. The image has to get a source prop, which has to be an object with the URI key. It can be a network URI or the local file resource. For the image, there are also various loading event handlers that can be passed as props, and they are as follows:
    - `onLoad`
    - `onLoadEnd`
    - `onLoadStart`
    - `onError`

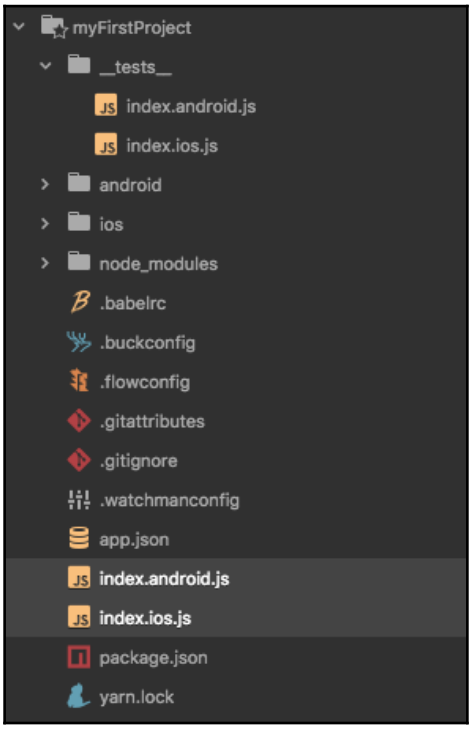
Additional data on image components can be found at `https://facebook.github.io/react-native/docs/image`.

- `<Button/>`: Button was introduced in 0.43 version to React Native. It can get title and color props as well as the `onPress` prop. Usually, `onPress` will be a callback function that will be executed once the Button component is pressed. Before Button component was introduced--and even now if you are not fond of the default button--it can be implemented with the use of `TouchableOpacity` or `TouchableNativeFeedback` components, which we will cover in `chapter 3`, *Getting familiar with React Native components*.

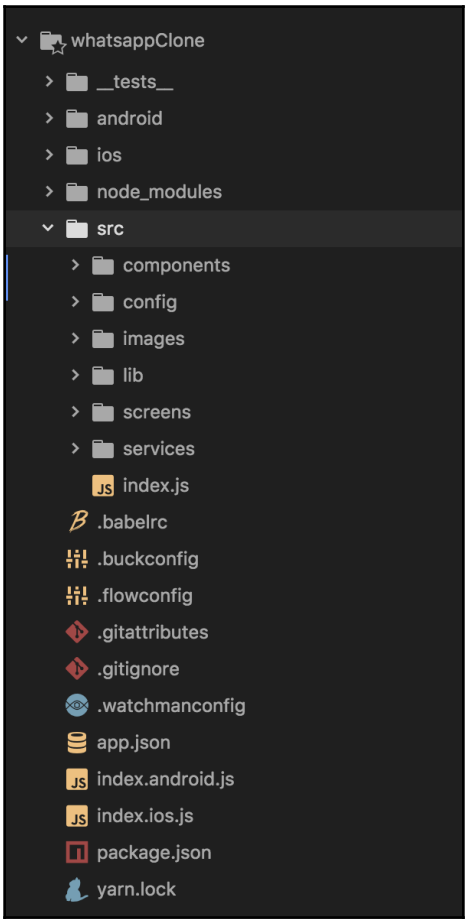
The folder structure

Structuring React Native components is important, but it's also very important to structure your folder properly. As your application grows and becomes more complex, it's harder to introduce new features without a proper folder structure.

Let's look again at the default folder structure that React Native gives us:



As we've discussed before, `index.Android.js` and `index.ios.js` are entry points for Android and iOS, respectively. We want our application to be as much cross-platform as we can, so the first thing we will do is create the following structure:



Now let's understand what is the purpose of each directory:

- `src/components`: All presentational components will go here. They can be put as separate files or organized into folders.
- `src/config`: All configuration will be put here, such as routes, image URL list, and more.
- `src/images`: Local images will be in this folder.
- `src/screens`: Here we will put our screen layouts.

- `src/lib`: All utility functions will be here.
- `src/services`: Sometimes, the application will need some business logic outside of container or component scope. If such need appears, it can be put inside service--for example, an API service.

`src/index.js` will be an entry point for cross-platform applications. Now, let's look at our `index.ios.js` and `index.android.js` files since they will be the same:

```
import React, { Component } from 'react';
import App from './src';

export default class myFirstProject extends Component {
 render() {
  return <App/>;
  }
}

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

Another important thing worth mentioning is `AppRegistry`. `AppRegistry.registerComponent` is the way in which an application root component is registered with the native world. The registration process is done by React Native itself and it connects between your specified component to `main.m` for iOS and to `MainActivity.java` and `MainApplication.java` for `android. main.m`. It can be found under `ios/yourAppName` folder and the Java files under your `android/src/main/java/com/yourAppName`.

Summary

In this chapter, we have covered all the basics you will need to know about how to create your React Native application. We covered React basics starting with JSX syntax and dived deeper into the difference between presentational and stateful components. We've covered how the state is updated and how the React lifecycle method helps us to achieve anything we want. Lastly, we've discussed best practices of structuring your React Native applications both in terms of folder structure and dividing components in.

3
Getting Familiar with React Native Components

Welcome to the third chapter. In this chapter, we will get to know most of the components you will need to know to develop amazing React Native apps. At first, you will get to know React Native components, which you can use on both iOS and Android. We will go over their usage and props and how to compose them together to create experiences you are used to seeing in most applications. Some of these components will have some extended capabilities on either iOS or Android. After that, we will go over all these components; we will take a close look at platform-specific components. We will start with iOS and continue with Android. After covering all these components, you will have a wide range of tools you can use to structure your apps properly. Then, we will dive into the important topic of navigation inside of your React Native apps. We will cover Navigator component and will understand the best practices we can use to make Navigation pretty easy.

So, let's summarize what we will learn in this chapter:

- Get to know platform-independent components, which you can reuse across iOS and Android
- Learn platform-specific components
- Understand how navigation in React Native works, learn best practices, and how to use it

Platform-independent components

One important thing React Native brings to us is that most components that will be reused in your application are platform independent. That means that it doesn't matter whether you use them on iOS or Android, they will work in the same manner. Some components will have an additional functionality, though, on some platforms, React Native core team work hard to unify these components even more. So, ensure that you check the official documentation during your app development, which can be found at: `https://facebook.github.io/react-native/docs/`.

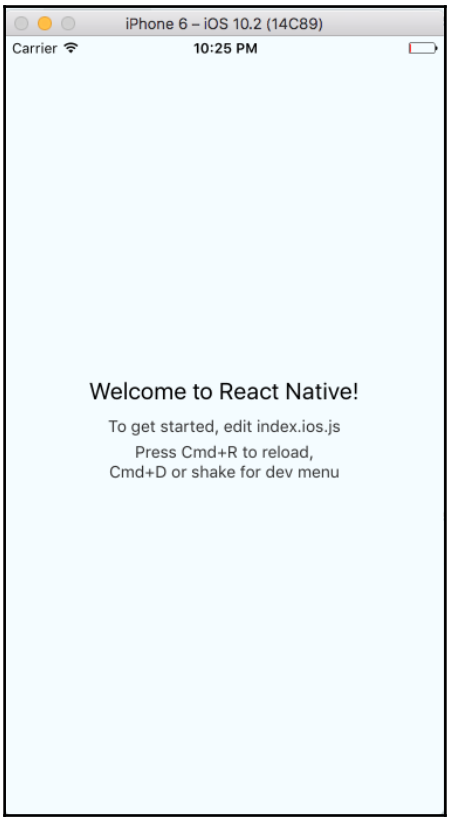
Basic components

Let's cover the most basic components that we will use across all of our apps. We've just mentioned two of them in our preceding chapter; however, we did this briefly and now I want to dive in to understand their purpose as well as capabilities.

Let's take a look at these components after creating our basic React Native project and look at the generated code:

- Run `react-native init componentPlayground` in terminal to create a basic *React Native* project
- Run `react-native run-ios` to run our application in simulator

You will get the following output:



In `Chapter 2`, *Working with React Native*, by looking at the code of this application at `index.ios.js` we've seen several imports from React Native:

```
Import { AppRegistry, StyleSheet, Text, View } from 'react-native'
```

We covered `AppRegistry` in the preceding chapter and will return to it in `Chapter 9`, *Understanding supported APIs and how to use them*, when we will dive into the React Native API--same for StyleSheet, we will take a look at it in `Chapter 5`, *Bringing the power of flexbox to native world*, when we will talk about styling your React Native apps. Now, in this section, we will focus on `View`, `Text`, `StatusBar` and `Image` components as some of the fundamental components of our application.

View

View is the most fundamental component for building our user interface. It's like a div in HTML, but in React Native it's much more. You will see that lots of components, which we will cover later, will get same props as the `View` component. If we take a look at our `index.ios.js` file, we will see that our render method returns content wrapped in the `View` component:

```
<View style={styles.container}>
  <Text style={styles.welcome}>
    Welcome to React Native!
  </Text>
  <Text style={styles.instructions}>
    To get started, edit index.ios.js
  </Text>
  <Text style={styles.instructions}>
    Press Cmd+R to reload,{'n'}
    Cmd+D or shake for dev menu
  </Text>
</View>
```

View is designed to be nested--similar to divs in HTML--and is used for layouting our application. It supports basic layouting through style prop, touch handling, and accessibility controls.

Layouting

View, as well as lots of other components, have an `onLayout` function, which is invoked immediately when layout has been calculated. It doesn't necessarily mean that layout exists on the screen. It only means that layout has been calculated.

`onLayout` is the function that gets an event argument in the following format:

```
{ nativeEvent: { layout: {x, y, width, height }}}
```

Touch events

There are basic touch events, but there are also different gestures, such as panning, sliding, and other gestures that you are familiar with from various iOS and Android apps. Touch events and gestures in React Native are dealt using the `PanResponder` API, which we will cover in `Chapter9`, *Understanding supported APIs and how to use them*.

There is an important View prop, though, that you should know and that we can use to limit touch events on a specific View:

```
pointerEvents
```

It can accept the following props:

- `box-none`: This indicates that View does not target touch events, but subviews do
- `none`: This indicates that no touch events on View nor on subviews
- `box`: This indicates no touch events in subviews only, but there are touch events for View itself
- `auto`: This indicates that the view can be target of touch events

Accessibility

For accessibility, there are several useful props on `View` component:

- `accessibilityLabel`
- `accessible`
- `onAccessibilityTap`
- `onMagicTap`
- `accessibilityComponentType` (Android)
- `accessibilityLiveRegion` (Android)
- `importantForAccessibility` (Android)
- `accessibilityTraits` (iOS)

Ensure that you check them out when you deal with accessibility:

`https://facebook.github.io/react-native/docs/view.html`

Accessibility is a separate topic that won't be covered in this book; however, it is recommended that you look into the official guide while dealing with it:
`https://facebook.github.io/react-native/docs/accessibility.html`

Text

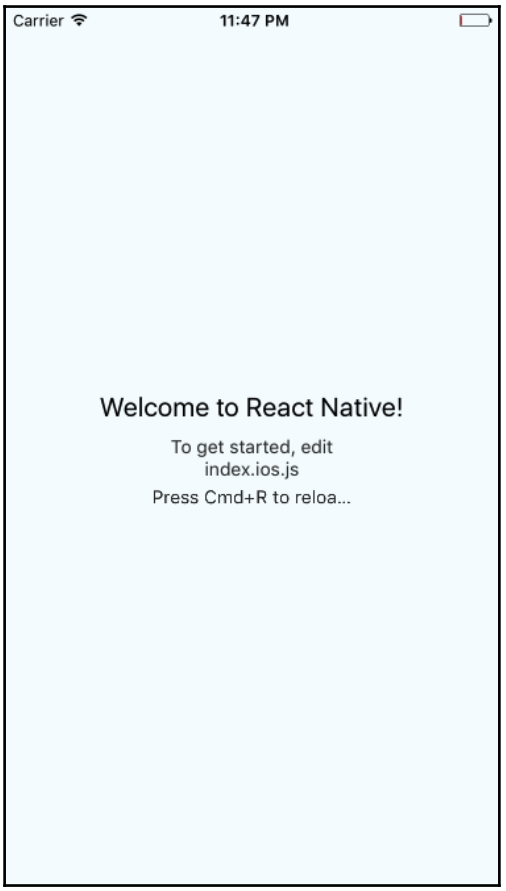
Text is used in React Native to display text on the screen. It can be nested to achieve different styling for different ranges of Text. For iOS, you can also nest a View inside the `Text` component. `Text` component supports styling, accessibility, and various touch options. Let's take a look at some important Text props:

- `onLayout`: same as for the `View` component
- `onPress`: This is the function callback called on pressing a text
- `onLongPress`: This is the function callback called when long pressing the text
- `selectable`: When true, lets the user select the text onscreen
- `numberOfLines`: This limits text to a specific number of lines and will show ellipsis, in the style specified by the `ellipsizeMode` prop
- `ellipsizeMode`: This is used only with `numberOfLines` and specifies different modes, such as follows:
    - `head`: "....text"
    - `middle`: "text....end"
    - `tail`: "text..."
    - `clip`: lines are not drawn past the edge of the text container.

Consider this example--let's limit one of our `Text` components to one line and set `ellipsizeMode` to `tail`:

```
<Text style={styles.instructions} selectable numberOfLines={1}
  ellipsizeMode="tail"
>
   Press Cmd+R to reload,
   Cmd+D or shake for dev menu
</Text>
```

The resulting screen will look like this:

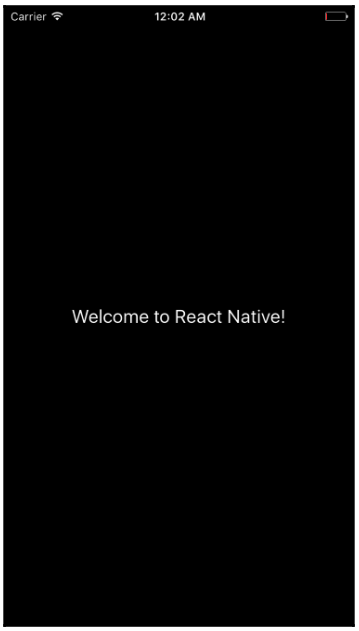


StatusBar

Regardless of whether this component is not a building block as View and Text, it controls our app's status bar component (the bar with the carrier icon, date and battery status).

In most apps, we really need this control--we need to animate it, set its color, or style.

For example, let's say we want one of our application screens to be black like this:



Also, we want to animate our `StatusBar` from black to white when we enter the preceding screen. The code for this screen will look like this:

```
<View style={styles.container}>
    <StatusBar animated barStyle="light-content"/>
     <Text style={styles.welcome}>
        Welcome to React Native!
     </Text>
</View>
```

Of course, don't forget to import the `StatusBar` component from React Native.

As you can see in the preceding code, we used an animated prop for `StatusBar` and `barStyle="light-content"` to specify that our status bar should be white.

The `barStyle` prop can be one of the following:

- `light-content`: White status bar
- `dark-content`: Dark status bar

The `barStyle` prop can be used only in iOS, whereas we can use `backgroundColor` and `translucent` props in Android. There is also the `hidden` prop that indicates whether `StatusBar` is hidden or not. For iOS, it can also be animated with `showHideTransition`.

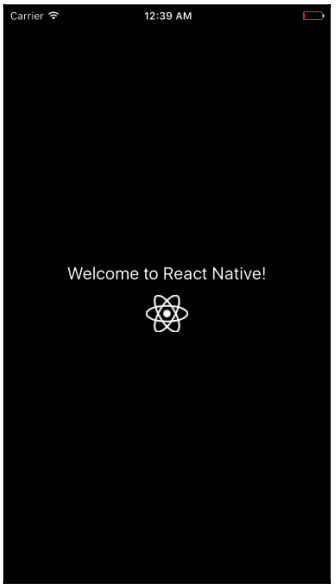
Sometimes, using `StatusBar` as a component is not an ideal solution and you need an additional functionality in `StatusBar` that is not exposed in component as props.

In that case, `StatusBar` provides a set of static functions that can be used without rendering an actual component. Note that the static API should be used only when not using props:
`https://facebook.github.io/react-native/docs/statusbar.html`.

Images and media

React Native provides a way to manage your images and media. Although, the component is called **Image**, it can be used to display not only images, but also different static resources. Specifically for images, React Native knows how to fetch images from the network also.

Let's say that we want to render a logo after the Welcome message as in the following screenshot. We will take React logo from the following React Native docs site:
`https://facebook.github.io/react-native/img/header_logo.png}`:



Let's take a look at the following code:

```
<View style={styles.container}>
  <StatusBar animated barStyle="light-content"/>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
  <Image style={styles.logo} resizeMode="contain"
     source={{ uri:
      'https://facebook.github.io/react-native/img/header_logo.png'
     }}
  />
</View>
```

As you can see in the preceding code, we use an `Image` component and give it some styles. We also pass a `resizeMode` prop in it as well as a `source` prop.

`resizeMode` can get the following values:

- `cover`: Scales image that maintains the image aspect ratio while image dimensions will be equal or higher than provided in style's width and height
- `contain`: Scales image that maintains image aspect ratio while image dimensions will be equal or lower than provided in style's width and height
- `stretch`:: Scales width and height without maintaining the aspect ratio
- `repeat` (only in iOS): repeat image
- `center`: center image

Images have additional lifecycle props (such as `onLoad`, `onLoadEnd`, or `onLoadStart`) to handle loading and accessibility options. In addition to props, images also have the following two static methods:

- `getSize`: You can get image size before rendering it by running the `Image.getSize` method
- `prefetch`: You can prefetch a remote image with this method and cache it to disk

In latest version of React Native there was a new Image type introduced: **ImageBackground**. It's used specifically for setting background image for a View. It has same properties as an image, however if you use image with children, you will get the following warning: "**Using <Image> with children is depreceated and will be an error in the near future. Please reconsider the layout or use <ImageBackground> instead**"

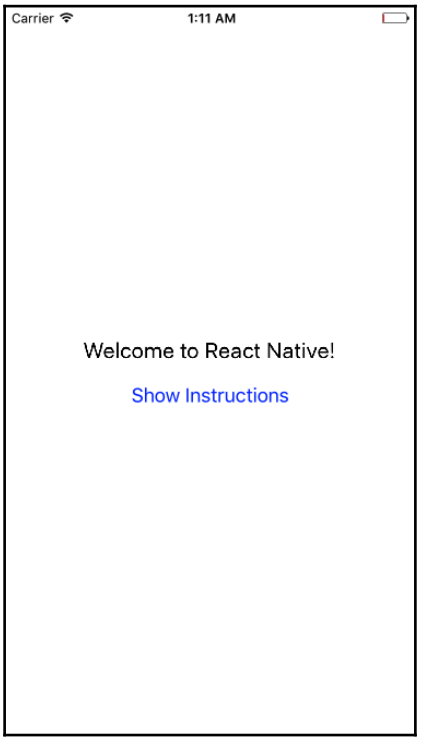
Basic user interaction

Mobile applications not only have to display content on the screen, but they should provide a great level of interaction for the user. The basis of this interaction is clicking buttons.

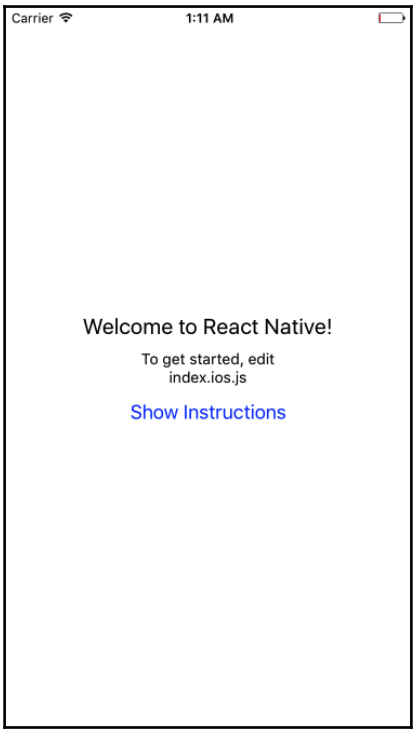
Button

Until recently, React Native hadn't had a `Button` implementation. Instead, it had various wrappers that could be used to get a button behavior. Owing to the high demand for the basic button, this component was created.

Let's create the following behavior; you will see the screen with the `ShowInstructions` button:



You will see the following instruction appear when you click on it:



First of all, we will need a state for interaction; let's define it using the `ES6` class property syntax:

```
class componentPlaygroud extends Component {
   state = {
     showInstructions: false
   }
// rest of code
```

We can also define our state inside constructor:

```
class componentPlaygroud extends Component {
   constructor(props){
   This.state = {
     showInstructions: false
   }
}
```

Now, let's get to our `render` method; it returns the following:

```
<View style={styles.container}>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
   { this.state.showInstructions ?
     <Text style={styles.instructions}>
       To get started, edit index.ios.js
     </Text>:
     false
   }
  <Button onPress={() =>
    this.setState({ showInstructions: true })}
  />
</View>
```

Let's walk through the preceding code. We conditionally show `Text` component with instructions when our `showInstructions` state is set to true. The interaction happens inside the `Button` component whenever a user clicks on it. As you can see from the code, we are passing the `onPress` callback prop, which is called when the user clicks on the button.

The default button is very basic and can be configured only by passing the following props:

- `color`: This sets the color of the button in iOS and sets its background color in Android
- `disabled`: You can disable interactions with this component by passing this boolean prop
- `title`: As you can clearly can see from the preceding code, it's used to pass the button caption

TouchableOpacity

This is basically the component that was used before the `Button` component was shipped. The `TouchableOpacity` component is a wrapper that wraps a child component with the View and lets this View respond to touches. Also, it changes opacity of wrapping View when touched. It has the same props as `TouchableWithoutFeedback`, which is the basic component for both `TouchableOpacity` and `TouchableHighlight`.

You can configure opacity with the `activeOpacity` prop. It defaults to `0.2` if no value is passed. This prop indicates the opacity that should be there when pressing a button.

Note that `TouchableOpacity` and all Touchable components should have ONLY one child.

TouchableHighlight

`TouchableHighlight` is a bit more complicated and configurable than `TouchableOpacity`. Like `TouchableOpacity`, it changes opacity of its child component wrapping View. In addition to that, on a touch event, the underlay color is shown by darkening or tinting the View. It has the following additional props dealing with underlay:

- `onHideUnderlay`: This function is called when underlay is hidden
- `underlayColor`: you can configure the color of the underlay, which will be shown through the View. By default it won't show if not configured.
- `onShowUnderlay`: This function is called when underlay is shown. This handler can be used, for example, triggering additional animations once underlay color of Touchable highlight is shown.

TouchableWithoutFeedback

This component is the basis of both `TouchableHighlight` and `TouchableOpacity`; however, it's advised not to use it on its own because, as the name suggests, it doesn't have any feedback for the user, which is really bad in terms of UX.

In addition to `onPress`, `TouchableWithoutFeedback` brings with it `onPressIn`, `onPressOut`, and some other props.

A full list of props can be found in the following official docs:
`https://facebook.github.io/reactnative/docs/touchablewithoutfeedback.html`

Getting feedback from your application

So, now we know how to create basic screens; however, sometimes we need to get feedback from our application. It can be rendering a loading icon, presenting some kind of message model, or even showing the refresh button. For that, we have the following set of components.

ActivityIndicator

`ActivityIndicator` is basically a spinning loader, which you are familiar with from dozens of apps. It's probably one of the smallest components in React Native. In addition to View props, it has the following three props:

- `animating`: Boolean prop that should be passed, so the component will determine whether it's animating or not
- `color`:: You can change the foreground color of the spinner
- `size`: You can pass a small or large string to indicate whether `ActivityIndicator` is small or large; for Android, you can pass size prop to configure exact dimensions

Let's take a look at what an activity indicator looks like. Let's use the same screen we used before with the `ShowInstructions` button, but this time let's delay showing instructions by 5 seconds.

We will introduce another state key-- `loading`--and use it to show the `ActivityIndicator`:

```
state = {
    showInstructions: false,
    loading: false
 }
```

Let's create a function that will change our state later on. We will call it `ShowInstructions`. Since we use the `ES6` classes, we will always need to bind our functions when passing them to props. So, our constructor will look like this:

```
constructor(props){
   super(props);
   this.showInstructions = this.showInstructions.bind(this);
 }
```

Now, we need to have a functionality that executes the `ShowInstructions` function. In this function we, first set our state to loading (to show activity indicator), and then after 5 seconds, disable both loading and the Button.

`ShowInstructions` will look like this:

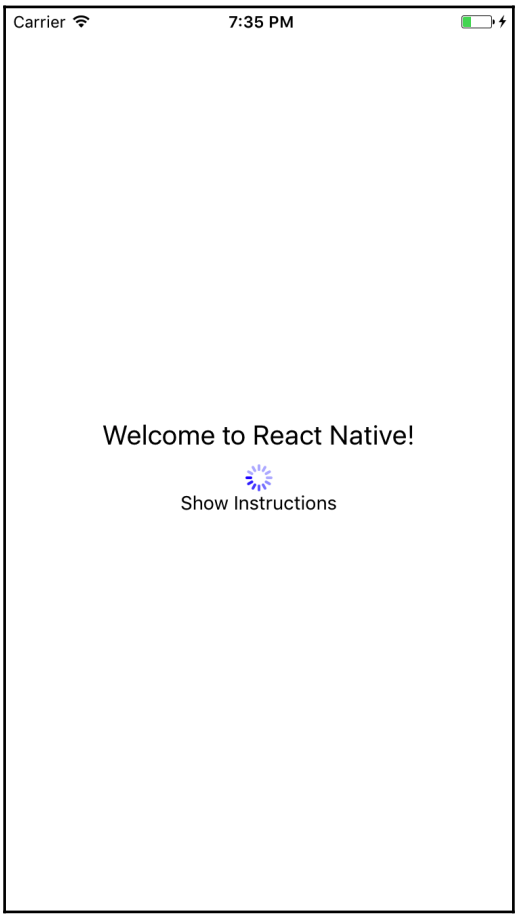
```
    showInstructions(){
      this.setState({
        loading: true
      });
      setTimeout(() => {
        this.setState({
          showInstructions: true,
          loading: false
        })
      }, 5000)
    }
```

The `render` method will look like this:

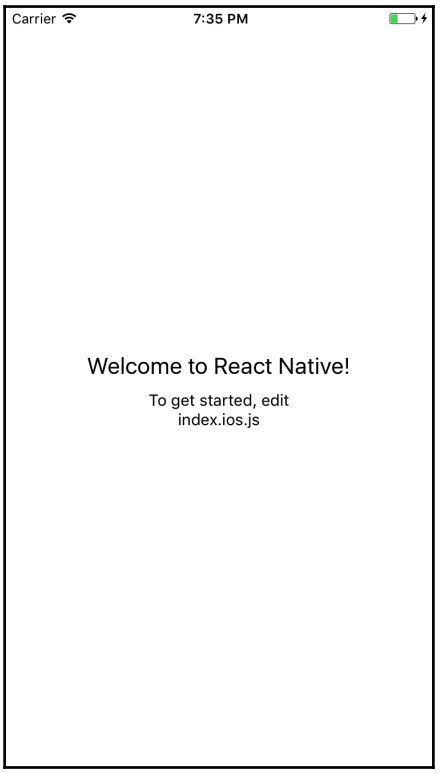
```
<View style={styles.container}>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
  {
    this.state.loading ?
      <ActivityIndicator color="blue" size="small" animating/> :false
  }
  { this.state.showInstructions ?
      <Text style={styles.instructions}>
        To get started, edit index.ios.js
      </Text> :
      <TouchableOpacity onPress={this.showInstructions}>
        <Text>Show Instructions</Text>
      </TouchableOpacity>
   }
</View>
```

Let's walk through the code. First of all, we define our view container and place the **Welcome to React Native** text in it. Then, we check whether our state is loading and render our small, blue, and animated `ActivityIndicator`. Then, we wait for 5 seconds and update loading state to `false` and `showInstructions` to `true`.

According to the `showInstructions` state, we will have instructions or a button rendered. Note that, in this example, I've used `TouchableOpacity` together with the `Text` components to get a custom `Button` component of my own. After clicking on the **Show Instructions** button, the result will look like this:



The following result will be seen 5 seconds after clicking:



Modal

Modal is used to present content above the enclosing view. It's used for various things. It can be success, feedback, or error messages; it can contain forms or images.

The `Modal` component gives you limited ability to present Modal view. In case you need more control over Modal view, it's advised to use a separate screen. The `Modal` component receives the following props:

- `animationType`: This defines animation type for Modal appearance. It can be none, slide (slides from the bottom), or fade.
- `Transparent`: If true, this will render the Modal background transparent. It's used to display Modal window on top of the application screen.
- `Visible`: This controls Modal visibility.

- `onShow`: You can pass a callback function to the `Modal` component that will be executed once Modal is opened.

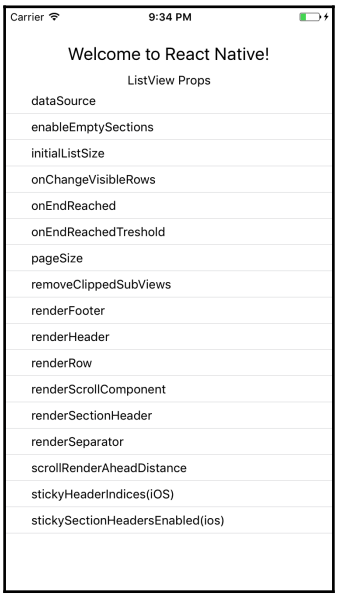
Dealing with lists of data

Up until now, we've seen the basic building blocks of every application. You have layout components and feedback components, but your application will be pretty empty without lists of data. Lists can be static and can be scrollable. We will take a look at both of them.

ListView

ListView is used to display a scrolling list of data. Usually, this data is subjected to change, but the important thing is that it should be similarly structured. ListView is optimized specifically for scrolling lists of changing data. It will calculate and render to the screen only elements, that can fit into view. If you have a long list of data, components beyond the screen will be rendered lazily when you scroll down.

Let's change our component to render a list of props of ListView when clicking on the **Show ListView Props** button. The resulting app should look like this:



We will simulate a network request with `setTimeout` in this example, since we will talk about network requests and deal with actual data later on.

So, how do we begin creating this list? First of all, `ListView` has to have `dataSource`. Based on the data it gets from the `dataSource` and `rowHasChanged` functions, it decides what content to render. Our constructor function from the preceding example will change into the following:

```
constructor(props){
  super(props);
  this.ds = new ListView.DataSource({
    rowHasChanged: (r1, r2) => r1 !== r2
  })
  this.state = {
    dataSource: this.ds.cloneWithRows([])
  };
  this.showList = this.showList.bind(this);
}
```

As you can see from the preceding code, we set up the `ds` property on the component class to be able to reference it later on. Then, we created `dataSource` by calling the static `DataSource` method on `ListView`. This method receives a `rowHasChanged` function, which basically is a comparator between row elements. The `r1` and `r2` arguments stand for row 1 and row 2 in `dataSource`. Calling this method will create a `DataSource` object, which we will use to create the actual `dataSource`. In our example, we only render rows to the screen, but we can do much more through `DataSource` methods. Its API is described in the official docs, but we will mostly work with other, newer list components that were introduced in the React Native 0.43 version. We will cover these components in a moment.

After creating an actual `dataSource` object and assigning it to state, we have to create `ListView`.

We will create a `getListView` class method, which will return the title and list of data:

```
getListView(){
  return [
    <Text key="listView props title">ListView Props</Text>,
    <ListView key="listView"
      style={styles.listView}
      dataSource={this.state.dataSource}
      renderRow={(rowData) => (
        <View style={styles.listRow}>
          <Text>{rowData}</Text>
        </View>
      )}
```

```
          />
        ]
      }
```

We take our `dataSource` state and pass it to ListView's `dataSource` prop. Also, we pass the `renderRow` function. This function will run for each item in `dataSource` and will act as a render method for each `dataSource` row.

In our example, it just renders the `View` with text inside of it. Let's look now at our render function:

```
<View style={styles.container}>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
  { this.state.showList ?
    this.getListView() :
    this.state.loading ?
      <ActivityIndicator color="black" size="small" animating/> :
      <Button color="black"
       onPress={this.showList}/>
  }
</View>
```

You can see that it looks somewhat similar to the Show Instructions example. The only change is that instead of rendering instructions, we call the `getListView` function.

When rendering our application for the first time, we will see a welcome message and a button. Let's emulate network request by clicking on this button. You can see in the preceding code that the `showList` method is called by pressing the button. Let's take a close look at it:

```
    showList(){
      this.setState({
        loading: true
      });
      setTimeout(() => {
        this.setState({
          showList: true,
          loading: false,
          dataSource: this.ds.cloneWithRows(LISTVIEW_PROPS)
        })
      }, 5000)
    }
```

Here, as you can see, we set the `loading` state to `true` to display `ActivityIndicator`, and then after 5 seconds, we update `dataSource`, taking content from the `LISTVIEW_PROPS` constant. This constant is an array of all the `ListView` prop names:

```
const LISTVIEW_PROPS =
 ['dataSource','enableEmptySections','initialListSize',
 'onChangeVisibleRows', 'onEndReached', 'onEndReachedThreshold',
 'pageSize', 'removeClippedSubviews', 'renderFooter', 'renderHeader',
 'renderRow', 'renderScrollComponent', 'renderSectionHeader',
 'renderSeparator', 'scrollRenderAheadDistance', 'stickyHeaderIndices',
 'stickySectionHeadersEnabled']
```

This is the basic example of `ListView`; however, `ListView` is much more versatile, as you can see by looking at its props list. It can support sections, sticky sections, headers, header, footer, callbacks, and much more. You can read more about all its props in the official docs; however, as I mentioned before, we will get back to this topic later on during development of more complex applications. Visit the following mentioned links for more information:

- Information on ListView Docs can be found at the following link:

  `https://facebook.github.io/react-native/docs/listview.html`

- Information on ListView `DataSource` Docs can be found at the following link:

  `https://facebook.github.io/react-native/docs/listviewdatasource.html`

ScrollView

ScrollView is a more generic component than ListView and is not very efficient performance wise. It wraps View with scrolling capabilities and provides integration with touches on various levels. The downside of ScrollView is that it renders all elements at once; it is different from ListView, which is much more optimized and can render elements lazily.

ListView is built for rendering similarly structured data, whereas you can render totally different components with ScrollView. Also, ScrollView does not use `dataSource` to render items. You can just pass child components to it, and it knows how to render them.

Note that since ScrollView renders all its children even if they are off screen, all views passed to ScrollView should have height. Having a child view without a specified height can lead to calculation errors for ScrollView.

I won't dive deeper into ScrollView props since there are lots of them, but we will use some of them in later chapters both for ScrollView and for ListView, which inherits them. There are several useful props there like, for example, `keyboardShouldPersistTaps`. This prop is all about dealing with keyboard. When passing 'always' it will ensure that taps are passed to children for handling. This is super useful.

For ScrollView Docs visit the following link:
`https://facebook.github.io/react-native/docs/scrollview`

RefreshControl

This is a fairly simple component that can be passed to a vertical ScrollView using the `refreshControl` prop or is set in `FlatList` or `SectionList` when passing the `onRefresh` prop.

This component is in charge of refreshing your list by pulling it down. This functionality is a common user experience in mobile applications. It's two major props are `onRefresh` and `refreshing`. `onRefresh` is a refresh callback, and `refreshing` is used to show or hide the refresh indicator.

FlatList

FlatList is a component used to render more performant flat lists, which was released in the React Native 0.43 version and does not require `dataSource` as in ListView. FlatList has lots of handy features, which makes it the number one choice for both basic and more complex lists in your app. It has a really impressive configurability, both in its appearance and functionality. Let's look at how FlatList is rendered and overview most of its props.

We will use the previous example of rendering ListView; however, we will include the following changes:

We don't need `dataSource` for FlatList anymore, so we will simply provide data state key with an array of our items like this:

```
constructor(props){
   super(props);
   this.state = {
     flatListProps: [
       {
         title: 'title',
         description: 'description'
       }
```

```
            ]
        };
        this.showList = this.showList.bind(this);
      }
```

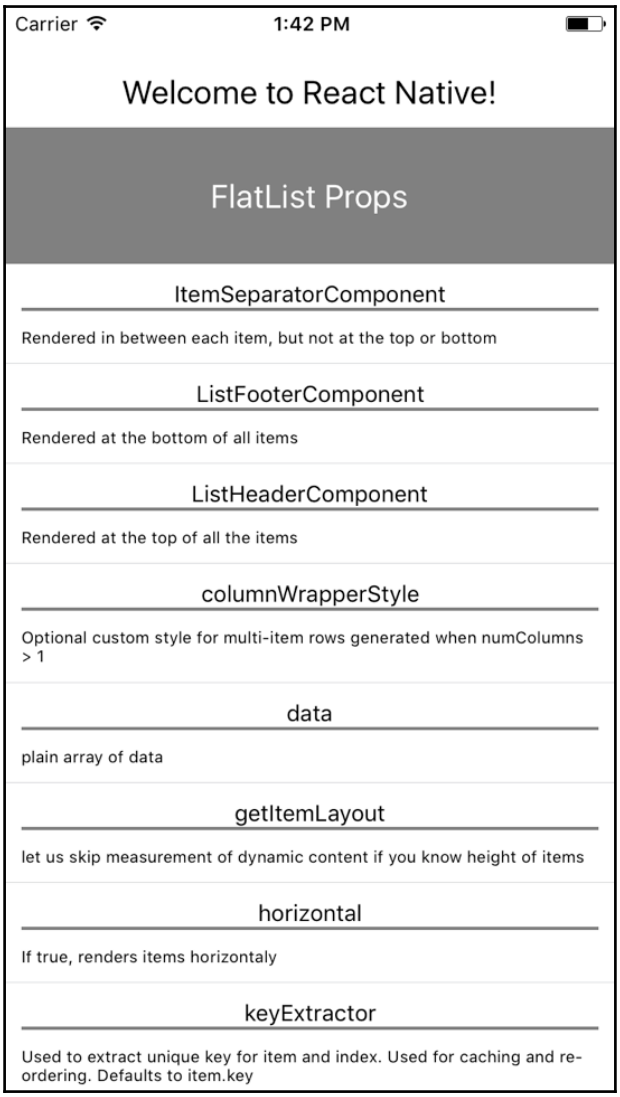
Now, let's get back to our `getListView` function and change it as follows:

```
    getFlatList() {
        return (
          <FlatList key="flatList"
            style={styles.flatList}
            data={this.state.flatListProps}
            ListHeaderComponent={() => (
              <Text style={styles.header} key="FlatList props">FlatList
              Props</Text>
            )}
            keyExtractor = {(item, index) => (`${item}--${index}`)}
            renderItem = {({ item, index }) =>
              <View style={styles.listRow}>
                  <View style={styles.rowHeader}>
                    <Text>{item.title}</Text>
                  </View>
                  <View>
                    <Text style={styles.description}>{item.description}
                    </Text>
                  </View>
              </View>
            } />
        )
      }
```

And of course, `render` method:

```
<View style={styles.container}>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
  { this.state.showList ?
    this.getFlatList():
    this.state.loading ?
      <ActivityIndicator color="black" size="small" animating/>:
      <Button color="black"
        onPress={this.showList}/>
  }
</View>
```

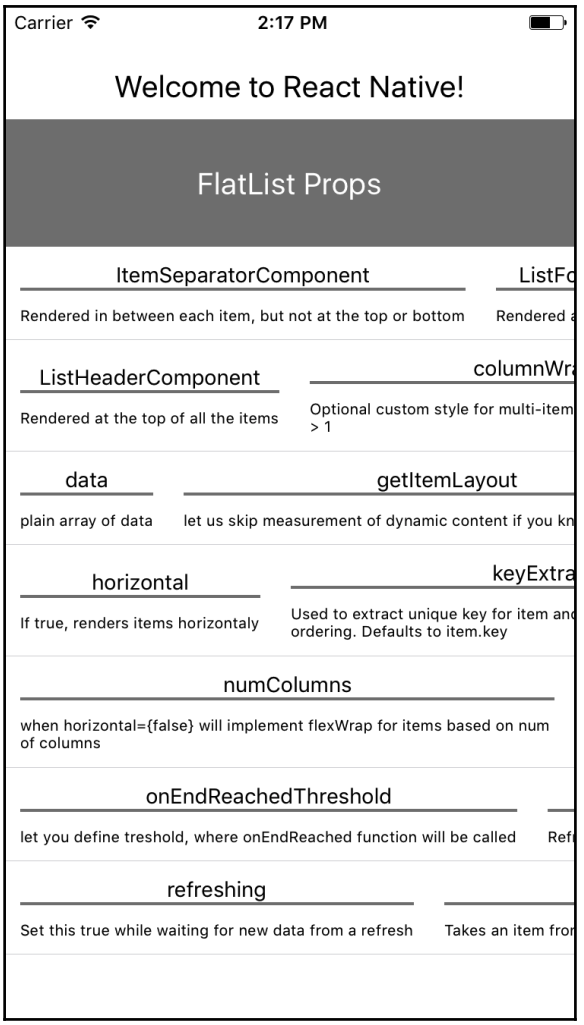
The result will look like this (I've added custom styles which you can check on github repo):



Let's look at the props that I've used to render this FlatList:

- `data`: This prop is a simple array of data that is passed to FlatList. It takes this array of data, and on each item, it executes a `renderItem` function.
- `ListHeaderComponent`: Using this prop, I've passed a list header. In the same manner, I can use `ListSeparatorComponent` or `ListSeparatorComponent` to render custom components for the header, separator, and footer.
- `keyExtractor`: By default, FlatList tries to add a key prop to every item in a data array using `item.key`. However, if you haven't specified a key in the data array, you can use the `keyExtractor` callback prop. This prop will run for each item in the data array, get an item and index as arguments, and return a string, which will be added as a key prop on the list item.
- `renderItem`: This callback function is executed for every item in a data array, and this is basically the place where you specify how your list item will be rendered. It gets an object with an item and index key as an argument. The returned React component will be rendered in the list.
- `columnWrapperStyle`: You can pass custom style here for multi-item rows when `numCulumns` > 1. For example, if we wanted to style our preceding example, we would have used this prop to pass it a custom style. This style would have been applied to every multi-item row.
- `onRefresh`: FlatList ships with a Pull to Refresh functionality if you pass it the `onRefresh` callback.
- `onEndReached`: This callback prop can be used to define lazy loading of data, when the user scrolls to the end of the list. Internally, the `onEndReachedThreshold` prop will be checked to determine whether the user scrolled to the boundaries of the threshold and if so, the prop will be executed.
- There are several other props that should be mentioned, making FlatList very configurable in a simple way.
    - `horizontal`: Specifying horizontal as true will lead to horizontal rendering of items.

- `numColumns`: When horizontal is false, you can specify number of columns, and you will get a zigzag-like View. In our example, if you pass `numColumns={2}`, you will get the following View:

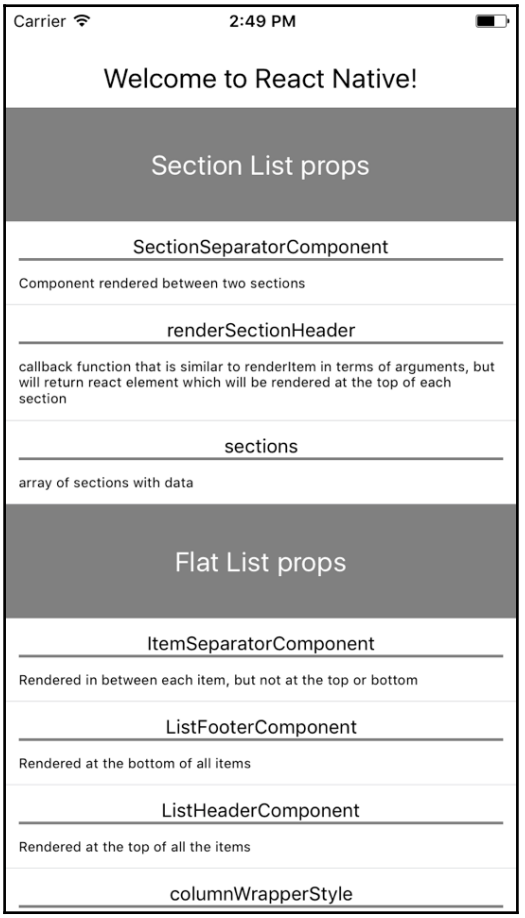


There are additional props, that can be used to configure your FlatList even more. You can check these props by checking official FlatList docs.

For FlatList Docs:
`https://facebook.github.io/react-native/docs/flatlist`

SectionList

`SectionList` is a component that extends the `FlatList` functionality even more. As the name suggests, it lets you render your list component with section headers. Let's change our preceding `FlatList` example to incorporate such behavior:



Now let's create this component. For that, we will need to introduce a new state key with section list props in our constructor in the same manner we've added `flatListProps` in the preceding example.

Now, let's change our component to render sections:

```
<SectionList key="listView"
  style={styles.listView}
  sections={[
    {
      data: this.state.sectionListProps,
      title: 'Section List props',
      key: 'slp'
    },
    {
      data: this.state.flatListProps,
      title: 'Flat List props',
      key: 'flp'
     }
   ]}
   renderSectionHeader={({section}) => (
     <Text style={styles.header}>{section.title}</Text>
   )}
   keyExtractor={//...no changes from FlatList example}
   renderItem={//...same code for rendering item  }
/>
```

As you can see, `keyExtractor` and `renderItem` props stayed the same. What did change is instead of a `data` prop, there is a `sections` prop, which is an array of objects. Each one of these objects has data and key. In data key, we pass an array of data for a specific section, and key will be used for React key when rendering components.

There is also the `renderSectionHeader` callback function prop, which is used to render a header for each section. In the same manner as the `renderItem` callback prop gets an object with item and index keys, `renderSectionHeader` gets an object with section and index keys as an argument. In the preceding example, you can see that I used the `ES6` function argument destructuring to get this section key, and then I rendered a `Text` component with its corresponding title.

Let's summarize `SectionList` props:

- `sections`: An array of section objects with data and key keys

- `renderSectionHeader`: A callback function that is executed for each section, which must return a valid React Native component
- `SectionSeparatorComponent`: A component rendered between two sections

For information on `SectionList` Docs, visit the following link:

`https://facebook.github.io/react-native/docs/sectionlist`

In the `SectionList` and `FlatList` examples, we used `FlatList` and `SectionList` props from the docs as a source of data. It's important to understand that this can be any data. `FlatList` provides a simple API of working with flat lists and `SectionList` extends it further to introduce the concept of sections. In more complex examples you can use ListView.

VirtualizedList

VirtualizedList is the base implementation of `FlatList` and `SectionList`. It should be used only in cases where you need to use immutable data structures instead of data arrays or other flexibility not covered by `FlatList` or `SectionList`. This component won't be covered in the book since it's use is for extreme edge cases, but you a can read more about it in official docs.

For VirtualizedList Docs visit the following link:

`https://facebook.github.io/react-native/docs/virtualizedlist`

Embedding web content

Sometimes, you want to embed the existing web page inside your application. In mobile development, it's usually done using WebView. React Native is not different, and it gives you component that will render into native WebView.

The `WebView` component has a variety of props that allow you to manipulate the web page rendered inside. It has various callbacks that provide an option to register to specific events, such as errors, responding to messages from WebView, and so on. For example, adding **WebView** to our application will look like this:

It will be as simple as adding the `WebView` component and passing its `source` prop; in the same fashion, we pass the source prop to the `Image` component when dealing with network images. In this example, we also pass the `scalesPageToFit` prop to scale the page to fit `WebView` dimensions:

```
<View style={styles.container}>
  <Text style={styles.welcome}>Welcome to React Native!</Text>
  <WebView style={styles.webView}
    scalesPageToFit
    source={{ uri:
      'https://facebook.github.io/react-native/docs/webview.html' }}/>
</View>
```

Handling user input

One important part of every application is handling the user input. This can be done in various ways, and we will see examples of more complex applications written in subsequent chapters. When looking at applications you are familiar with, you probably encounter various ways to handle the user input. Mostly, this can be generalized to handling text input and various pickers, sliders, or switches.

TextInput

TextInput is the basic component for inputting text. It can handle plain text and passwords, have dozens of various modes and props, and is simply too huge to be described in this book. We will look at some implementations of text input in our examples when we look at real-world apps and develop more complex apps ourselves.

A basic use case for rendering `TextInput` for the username field is as follows:

```
<TextInput style={styles.textInput}
  onChangeText={() => this.setState({ username })}
value={this.state.userName}
/>
```

Sometimes, when you render `TextInput` in your app, and when the user clicks on it, the keyboard, that pops out can cover the `TextInput`. For that specific use case, ensure that you use `KeyboardAvoidingView` instead of the regular `View`. This specific view will add proper padding to the view to move `TextInput` out of the way of the keyboard.

- For more information on `TextInput` docs, visit the following link:

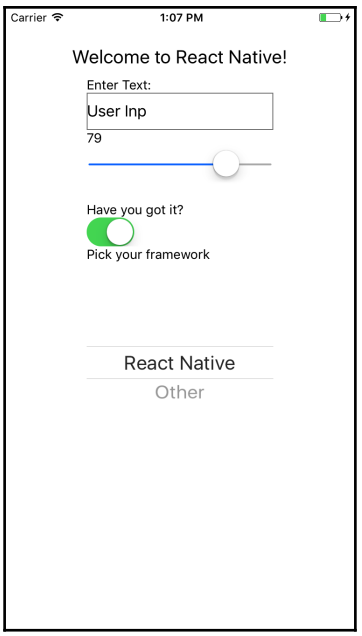
  `https://facebook.github.io/react-native/docs/textinput.html`

- For more information on `KeyboardAvoidingView` docs, visit the following link:

  `https://facebook.github.io/react-native/docs/keyboardavoidingview.html`

Restricted choice inputs

Restricted choice input components are used to let the user pick values from the list or in the range of predefined values. While in `TextInput`, the user can type anything they want, and only on submission, will the entered data be validated, in case of Picker, Slider, and Switch, the user has a limited choice. In terms of user experience, it's wise to let the user type as less as possible and use more of these three components since forms with lots of typing usually scare the user out. Let's take a look at the code we'll need to create the following form:



We have a TextInput, Slider, Switch, Picker, and some text labels in the preceding screenshot. Let's separate this form into several Views.

We will have a `TextInput` View:

```
<View>
   <Text>Enter Text:</Text>
   <TextInput
     style={styles.textInput}
     onChangeText={(text) => this.setState({text})}
     value={this.state.text}
   />
 </View>
```

Then, we will have a Slider View:

```
<View>
  <Text>{ this.state.sliderValue }</Text>
  <Slider
    step={1}
    style={ styles.controlWidth }
    value={this.state.sliderValue}
    maximumValue={100}
    onValueChange={(sliderValue) => this.setState({sliderValue})} />
</View>
```

For slider, we must define value as well as `maximumValue` to define the slider range. Also, we have to define the slider width in styles. Optionally, we can define it in steps. In our case, it means that the user can select only whole numbers because step is set to 1. Similar to how TextInput and Slider will take `onValueChange` function.

For Switch view, we will have the following code:

```
<View style={styles.withMargin}>
  <Text>Have you got it?</Text>
  <Switch
    value={this.state.gotIt}
    onValueChange={(value) =>
      this.setState({
        gotIt: !this.state.gotIt
      })
    }
  />
</View>
```

Similar to other form controls, you have the `onValueChange` function here as well as a value prop.

Switching your value should be boolean, because Switch knows how to work with boolean values. You won't get an error for providing a string, but you will get unexpected behavior from the `Switch` component.

For Picker view, we will have the following code:

```
<View>
  <Text>Pick your framework</Text>
  <Picker style={styles.controlWidth}
    selectedValue={this.state.framework}
    onValueChange={(framework) => this.setState({ framework} )}
  >
```

```
        <Picker.Item label='React Native' value='Awesome'/>
        <Picker.Item label='Other' value='undefined' />
      </Picker>
    </View>
```

Basically, instead of value in other form components, you have the `selectedValue` prop for Picker.

All `form controls` have additional props, so if you need some specific behavior, ensure that you check the official documentation for them since there are specific props for Android or iOS to match different user experiences:

- **Picker**: `https://facebook.github.io/react-native/docs/picker.html`
- **Slider**: `https://facebook.github.io/react-native/docs/slider.html`
- **Switch**: `https://facebook.github.io/react-native/docs/switch.html`

Platform-dependent components

React Native supplies a wide range of components that you can use in your applications, enough to build amazing experiences. However, there are some components that you want to use only on iOS or on Android. These components define the difference between two platforms. For example, Android drawer is something that you don't want to use in iOS since the user experience will be weird. iOS users are not accustomed to using Drawer. Similarly, using specific iOS controls on Android would look out of sync with the Android UI.

Detecting specific platform

When most of your application code is cross platform, you want some kind of system to properly detect when there are edge cases where you need to use platform-dependent components. React Native provides us with the following two ways of detecting the platform your app is running on.

First, you can use platform module. This module can be imported from React Native:

```
Import { Platfrom } from 'react-native'
```

- `Platform.OS`: Will return iOS or Android for current platform
- `Platform.Version`: Will return Android version (works only on Android)

Another way is to conditionally import your components based on their environment:

```
Platform.select({
  ios: () => require('ComponentIOS'),
  android: () => require('ComponentAndroid')
})
```

Extensions

React Native gives you a great way to load your platform-dependent component based on extensions. For example, you have `ComponentIOS` and `ComponentAndroid`. They are components implemented completely different with specific platforms in mind, but they should be loaded on the same screen. When that's the use case, you can rename both of these components to Component and add `.ios.js` or `.android.js` as file extensions:

```
Import { Component } from './component';
```

Now, if you import a component, React Native will load it with a specific extension. This is why initially you have `inde.ios.js` and `index.android.js` when bootstrapping a basic application. React Native wants to load an index file. Its extension mechanism tells React Native to load Android or iOS code.

DatePickerIOS

A date picker on iOS is as simple as adding the `DatePickerIOS` component. On Android, it's more complicated and requires a specific API; that's why it's usage is covered in React Native official API docs:
`https://facebook.github.io/react-native/docs/datepickerandroid.html`.

For iOS, though, we need to pass date prop for the currently selected date and `onDateChange` function to update our state when the date gets updated as a result of user selection. We also can pass `minimumDate` and `maximumDate` to define our date range and `timeZoneOffsetInMinutes` to define our timezone. The default timezone will be our device's timezone:

```
<DatePickerIOS
  date={this.state.date}
  onDateChange={(date) => this.setState({ date })}
/>
```

You can read more about all these props in official docs:
`https://facebook.github.io/react-native/docs/datepickerios.html`

Progress bars

In iOS and in Android, progress bars are completely different, so there are specific components for them--`ProgressViewIOS` and `ProgressBarAndroid`.

Both of them have a `progress` prop, however, their similarity ends there. For example, the iOS progress bar can be color or have an image displayed behind it, while Android cannot. For more information on the progress bar, refer to the following mentioned links:

- `ProgressViewIOS`: `https://facebook.github.io/react-native/docs/progressviewios.html`
- `ProgressBarAndroid`: `https://facebook.github.io/react-native/docs/progressbarandroid.html`

Additional controls

The user experience in Android and iOS has its differences. This is due to introducing different controls. Here are some of them that are supported in React Native.

SegmentedControlIOS

In iOS, there is a specific form control that is not found on Android. It's called segmented control, and you probably have seen it in dozens of apps:

Code rendering of this control looks as follows:

```
<SegmentedControlIOS values={['First', 'Second']}
   selectedIndex={this.state.segmentSelectedIndex }
   onChange={(event)=>
     this.setState({
       segmentSelectedIndex: event.nativeEvent.selectedSegmentIndex
     })
   }
/>
```

Instead of passing `onValueChange`, you can pass the `onChange` callback prop, which will receive an event argument. In the preceding example, we retrieve `selectedSegmentIndex` from `event.nativeEvent`.

We also can write in a different fashion. Instead of passing hardcoded values, you can pass a constant with first and second values. Then, instead of using the `onChange` function, we can use the `onValueChange` function and find the index of the value in a passed constant, as follows:

```
<SegmentedControlIOS values={SEGMENT_VALUES}
   selectedIndex={this.state.segmentSelectedIndex }
   onValueChange={(value)=>
   this.setState({ segmentSelectedIndex: SEGMENT_VALUES.indexOf(value)
}) } />
```

For docs, refer to:

`https://facebook.github.io/react-native/docs/segmentedcontrolios.html`.

TouchableNativeFeedback

On Android, TouchableNativeFeedback makes views respond to touches and lets the user define a custom background by passing a prop to this component. Since the experience on Android is different from iOS, this control provides you with Android native feedback responding to your touches.

You can read more about it at:
`https://facebook.github.io/react-native/docs/touchablenativefeedback.html`

Platform-specific navigation

Navigation differences between iOS and Android are much more than the different look and feel of components. The subsequent components have to be platform-specific, because they are unique to their platform. You would use `TabBarIOS` to create bottom tabs for your iOS apps, whereas your navigation on Android would be done using `DrawerLayoutAndroid`, `ToolbarAndroid`, and `ViewPagerAndroid`. Each one of them is unique and the use case for them are as follows:

- `TabBarIOS`: This renders bottom iOS tabs for navigation:
  `https://facebook.github.io/react-native/docs/tabbarios.html`

- `DrawerLayoutAndroid`: This creates Android drawer for your Android applications:
  `https://facebook.github.io/react-native/docs/drawerlayoutandroid.html`
- `ToolbarAndroid`: This creates an Android-specific top toolbar (usually having a hamburger icon on the left and settings icon on the right with a logo in between):
  `https://facebook.github.io/react-native/docs/toolbarandroid.html`
- `ViewPagerAndroid`: This creates a container that allows you to flip between child views. This UX is more natural on Android, hence, its implementation is only for Android:
  `https://facebook.github.io/react-native/docs/viewpagerandroid.html`

Navigation in React Native

One important thing to understand in React Native is how navigation is done. To set up basic navigation, we can use an amazing package that was created by the community and is used in lots of projects in production. It's called **React-navigation**. We will mainly use React-navigation for projects in this book, but before we take a look into it, let's understand how React Native Navigation was implemented before the community package.

The Navigator component

Navigator component is the component that was used in React Native to provide navigation to child views. It was released together with React Native in 2015 and was for 2 years, however much better implementations of navigation have emerged from a collective effort of the growing React Native community. One of the advised navigation implementations released by the community is the package, react-navigation, and it's available via npm.

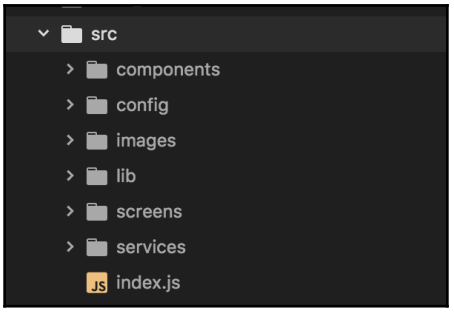
Navigator is a stack-based component that uses the screen stack for navigating through your application. When you navigate to a new screen, you push it into the stack, and when you navigate back, you pop it from the stack. Navigator was released before animated library, which enabled native animations in React Native, so all animations were done inside JavaScript. For iOS, there is a specific Navigator called `NavigatorIOS`, which implements native screen transitions differently from the Navigator component.

Now after understanding the history of navigation, let's set up basic navigation for the project we will be working on in the subsequent chapters: *whatsappClone*.

First of all let's create a new project and call it `whatsappClone`:

```
react-native init whatsappClone
```

Now, let's organize our folders a bit. We've talked about folder organization in the preceding chapter:



We create our JavaScript files in the `src` folder so our `index.ios.js` will look as simple as follows:

```
import React from 'react';
import App from './src';

AppRegistry.registerComponent('whatsappClone', () => App);
```

Now, let's create a `Home` component and `ChatScreen` component. Since, later on, we will add functionality and use lifecycle hooks, we will use class notation to create the component:

```
export default class Home extends React.Component {
  render(){
    return (
      <View style={styles.container }>
        <Button />
      </View>
    )
  }
}
```

We will do the same for `ChatScreen`, with just one difference:

```
<Button />
```

So now, let's get to navigation. What we want is to be able to navigate from the Home Screen to Chat Screen and back, by clicking buttons and using regular device navigation.

There are different navigation options, and while previously they could be done using React Native, now the best navigation options rely on external packages. So let's first of all install a navigation package:

```
npm i --save react-navigation
```

React Navigation is based around a concept of different Navigators. Let's import such a navigator in our `src/index.js` file and define it:

```
import { StackNavigator } from 'react-navigation';
import routes from './config/routes';
export default StackNavigator(routes);
```

As you can probably see, all our routing configuration resides in `src/config/routes.js` file:

```
import Home from '../screens/Home';
import ChatScreen from '../screens/ChatScreen';

 const routes = {
   home: { screen: Home },
   chat: { screen: ChatScreen }
 }

 export default routes;
```

Now on every screen, if we want our title to appear in the navigation bar, we should define the `navigationOptions` static property in the `screen` class, like this:

```
export default class Home extends React.Component {

static navigationOptions = {
  title: "Home Screen"
}

// rest of code
```

Now let's add actual navigation:

```
<Button
  onPress={()=> this.props.navigation.navigate('chat', { name: 'John' })}
/>
```

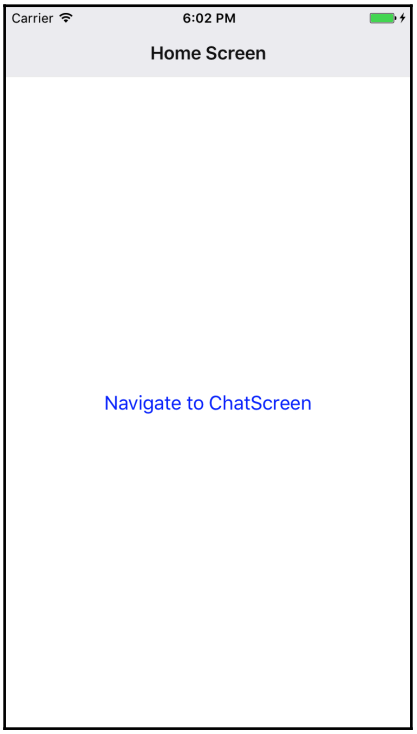
By calling the `navigate` function, we first pass an argument of the screen name as defined in the routes configuration and then pass the parameter that we want to pass to the next screen. Now, in our `ChatScreen`, we can reference this `param` by accessing:

```
this.props.navigation.state.params
```

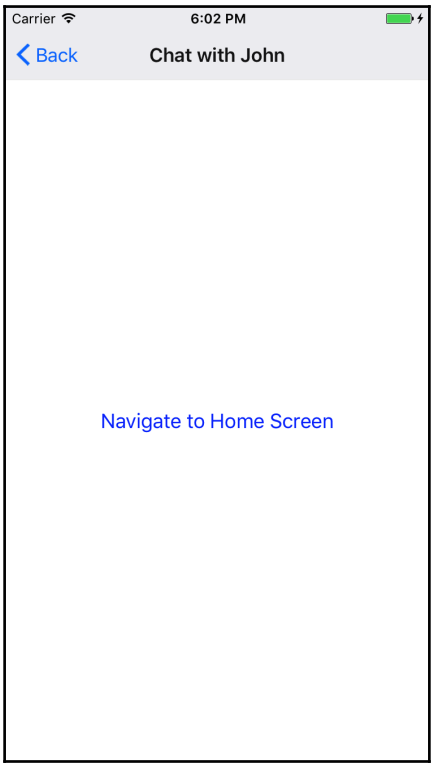
We also can reference it in our static property--`navigationOptions`--like this:

```
static navigationOptions = ({ navigation }) => ({
  title: `Chat with ${navigation.state.params.name}`
})
```

The resulting application screens will look like this:



The chat screen will look, as follows:



There is more that can be done with React navigation like nesting the Navigator, dealing with tabs and drawer without any need to use platform-specific components, and much more. We will gradually learn how to use it by looking at various samples and by building our *whatsAppclone* application; refer to the React Navigation official documentation-- `https://reactnavigation.org/docs`--for samples.

Summary

So, it was a long journey to explore all the components of the React Native framework. After exploring all components and lots of examples on how they can be used, we dove into learning Navigation in React Native. We've created a basic skeleton app for our future project, *whatsAppclone*. From now on, we will build real-world examples and apps. During this chapter, you've probably seen errors if you forgot to include some component or had a syntax error. Before we continue to style our *whatsAppclone* and working on actual screens in `Chapter 5`, *Bringing the power of flexbox to native world*, we need to pause and talk about debugging, performance, and testing of your React Native apps. That is what is awaiting you in the next chapter. So, buckle up and enjoy the ride.

4
Debugging and Testing React Native

Welcome to the fourth chapter. In this chapter, you will get yourself familiar with two important concepts. First of all, you will learn how to properly debug your apps using various set of tools built in React Native. Then, you will get yourself familiar with an app developer menu available in React Native apps. Then, we will go through a list of really helpful features that are not available for mobile app developers. You will be able to live reload your app, log things to the console, enable remote debugging with Chrome DevTools, which you are familiar with from web development, debug while running your app on an actual device, and inspect your React Native component structure, thanks to a built-in inspector, monitor performance with Chrome DevTools, Systrace, and Perf monitor. You will also understand how to deal with warnings and errors in your app, so your app development experience will be smooth and pleasant. The second concept, which is very important not only in React Native but also in development in general, is testing. React Native highly suggests usage of Jest testing framework built by Facebook for testing components, so you will get to know the basics of it, how to set it up, and how to begin testing your components. By the end of this chapter, you will get the knowledge to test every part of your application.

So, let's summarize. Here is what we will learn in this chapter:

- Debugging and remote debugging of your React Native apps
- Various Diagnostics tools and inspectors
- Introduction to the Jest testing framework
- Snapshot testing your React Native components
- Mocking modules

Debugging your React Native apps

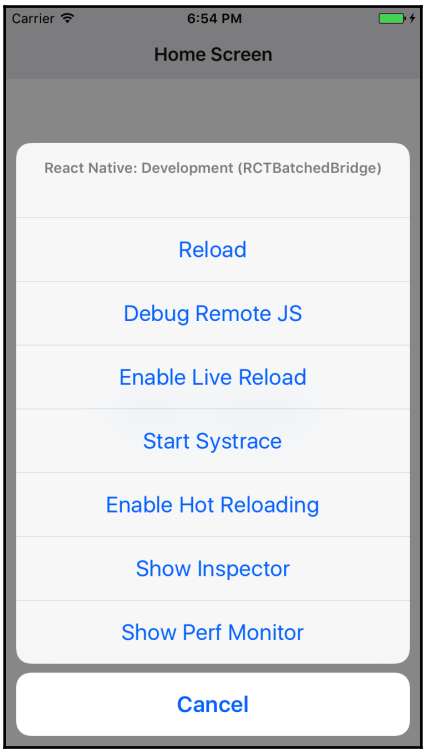
Debugging is a very important part of the development process. In mobile development, debugging sometimes can become tricky, so the React Native team approached this from a perspective of web development. In recent years, development experience or DX got a major breakthrough in the web development field. Web apps are now reloaded automatically while writing code; Chrome DevTools brought a lot of tools to debug, check performance, and inspect your written code inside a web app. In mobile development, DX is not that smooth. In React Native, we write in JavaScript, so why don't we use familiar tools from web development for the same DX, but for mobile apps? Well, we can use them. Let's take a look at what we get out of the box from React Native in terms of debugging. Ensure that you also check the official docs. New debugging options are introduced on a constant basis, and while the book is updated to the most recent React Native version and best practices on release date, there can be minor changes that can improve DX. All these changes are explained in official docs in debugging section:

```
https://facebook.github.io/react-native/docs/debugging
```

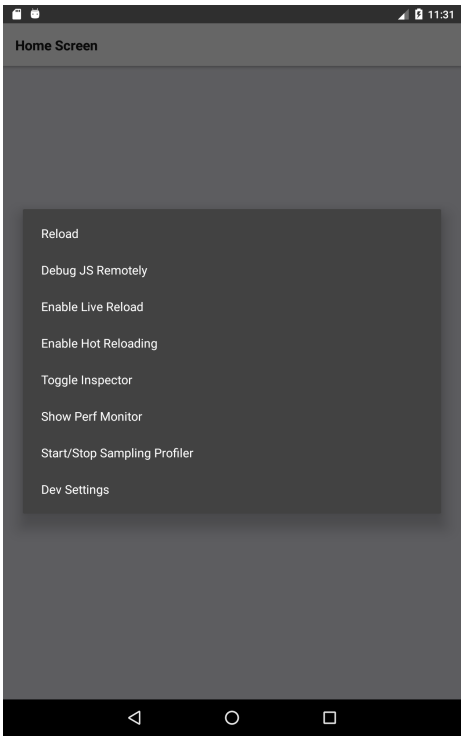
Developer in-app menu

In order to enable debug options in React Native, first of all you should bring an in-app developer menu. This menu has various options that we will cover in a bit, but they differ slightly in Android than iOS, and also in terms of bringing the menu itself. In both emulators, you can shake your device or use the *Command+D* keyboard shortcut for iPhone, or *Command+M* for Android.

The screen you will get in iOS is as follows:



For Android, the screen will look as follows:



As you can see from the previous screenshots, there is a difference in its usage between Android and iOS. On Android, there are additional **Dev Settings** option. **Dev Settings** has debugging and performance options

Reloading your app

While developing your application, you would always want to get the fastest feedback of how your application will look in the end. This can be done with reloading your application and checking whether changes you've done in the code are reflected in the actual app. In mobile development, if you want your application to reflect changes you've made to the app, you need to recompile and run it in an emulator or on a device. This can lead to long code change--recompile cycles and slow down developer experience. In recent years, in web development, we are used to reloading our browser and seeing the changes right away. React Native team brought the same DX to the mobile world. Let's check our reloading options in React Native apps.

Reloading

The first reloading option is fairly simple, and its functionality is basically the same as clicking on the browser's refresh button. You can tap **reload** from the developer menu as seen in the previous screenshots or press *Command+R* in the iOS simulator, or press *R* twice on Android devices. For that, you don't need to bring in the app developer menu.

Live reload

Automatic reloading for web applications has been there for several years, and now with React Native, this feature is in mobile development as well. When enabling it via the **In app developer** menu, every time you change your code and save it, your application is reloaded automatically.

Now, let's try it:

1. Open the *whatsappClone* application we've been writing in `Chapter 3`, *Getting familiar with React Native components*. Then, let's add mock data.
2. Create an `api.js` file under the `services` folder, and add the following code to it:

```
const mockMessages = [
  {
    incoming: true,
    message: 'Hi Vladimir'
  },
  // rest of messages in the same format.
]

export const getMockData = () => (
  new Promise(resolve => setTimeout(() =>
resolve(mockMessages),1000))
)
```

Let's understand what's going on here. I've added a `mockMessages` array with a list of messages and created a `getMockData` function, which simulates a network request by returning a promise. The result of the promise is the array of `mockMessages`.

3. Now let's start our app.

Note that any time your app crashes, you can restart only the packager without any need to run `react-native run-ios`. In order to restart packager you need to open a terminal window where the packager runs and press *Cmd+C*. Then, run `npm start` in your project folder. If it still does not help, you may consider to shut down your emulator and rerun your app with `react-native run-ios`. Make sure though, that packager is stopped.

4. Enable live reload through the in app developer menu.
5. Add the following code to `ChatScreen.js`.
6. At the top of the file, let's add our mock `api` service and `FlatList` to represent our messages:

```
import { getMockData } from '../services/api';
import { View, Text, StyleSheet, Button, FlatList } from 'react-
native';
```

7. Then, inside the `ChatScreen` class, let's add the following. First, let's define our state:

```
state = {
  messages: []
}
```

8. Then, let's represent our messages inside the render function. Let's navigate to Home Screen button, add the following code:

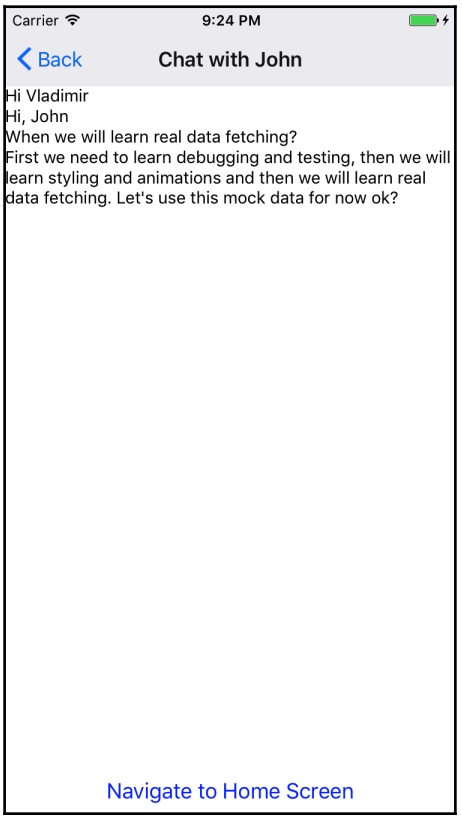
```
<FlatList
  data={this.state.messages}
  renderItem={({ item }) =>
    <View>
      <Text>{item.message}</Text>
    </View>
  }
  keyExtractor={(item, index) => (`message-${index}`)}
/>
```

9. Then, let's fetch our messages from the `api` we've just mocked:

```
componentDidMount(){
  getMockData().then((messages) => {
    this.setState({
      messages
    })
  });
```

```
}
```

10. Now, save the file. You will notice that the application is reloaded, and you get the home screen. Now, if you will navigate to the Chat Screen, you will get the list of messages after a 1-second delay:

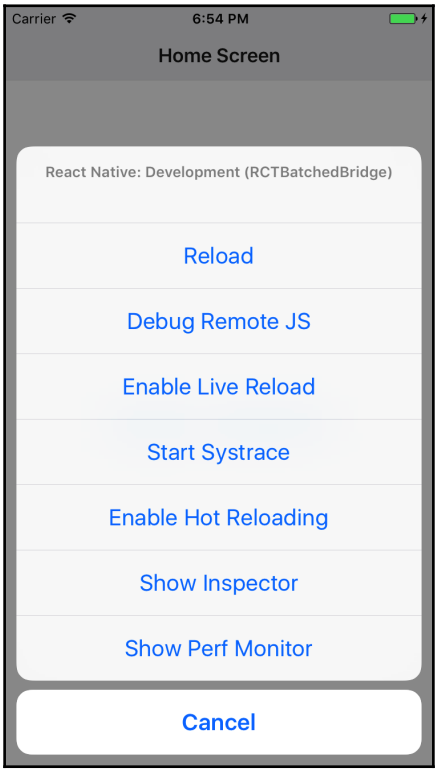


Hot reloading

In our *whatsAppclone* application, we have only two screens for now, but it means that every time you change the code on Chat Screen, your application is reloaded and you need to navigate back to Chat screen. In more complex apps, there are lots of screens, and navigating each time to the exact place in the application every time you change your code can be frustrating. In React Native, we have the best technology available from the web development world. One of the latest features introduced in the webpack bundling system is hot reloading. In React Native, you can also enable hot reload.

What hot reload means is that instead of fully reloading JavaScript bundle for your app (live reload), React Native packager is aware what parts of code were changed, and it replaces only these parts of code.

In order to enable hot reload, you need to simply select the **Enable Hot Reloading** option from the in-app developer menu, and Hot Reload will be enabled:



In practice, it means that if, for example, I want to add a header saying `Chat with John` on our Chat Screen, I can do so, by adding the following line's previous message list in the render function:

```
<View>
 <Text>Chat with John</Text>
</View>
```

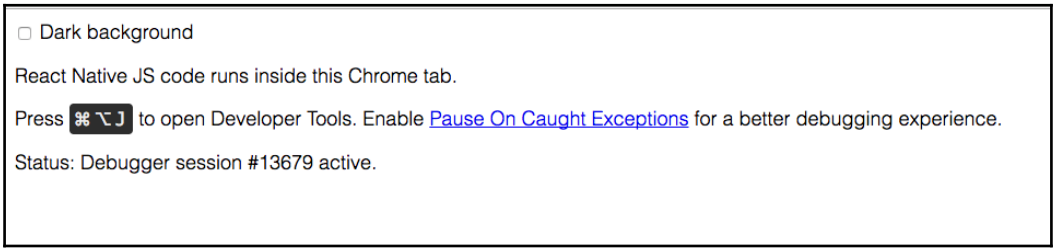
Now, when I save the file, instead of fully reloading an app, React Native packager reloads only the changed code. As a result, your changes are reflected right on screen without returning back to the home screen as in the live reload option.

Remote debugging

One of the most important development tools React Native gives us is the ability to debug our application remotely. It's the second option from the top in the development menu, and it allows several debugging options.

Debug with Chrome DevTools

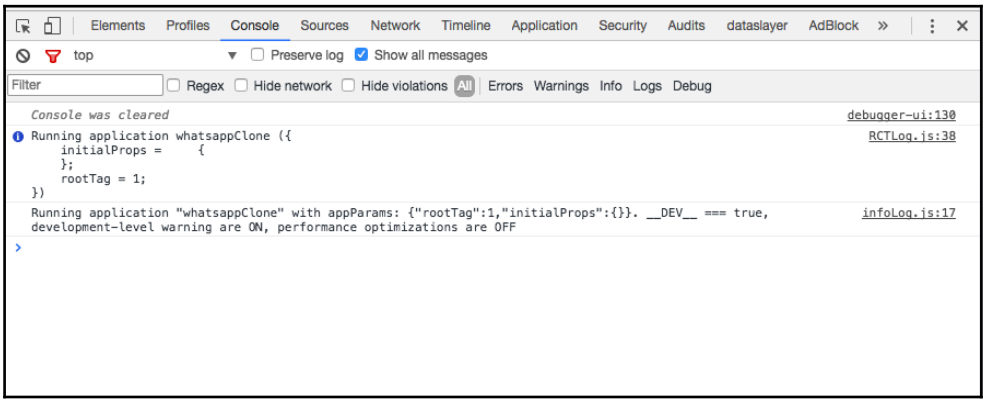
The first thing that happens when you click on **Debug Remote JS** is that Chrome window opens with the following text:



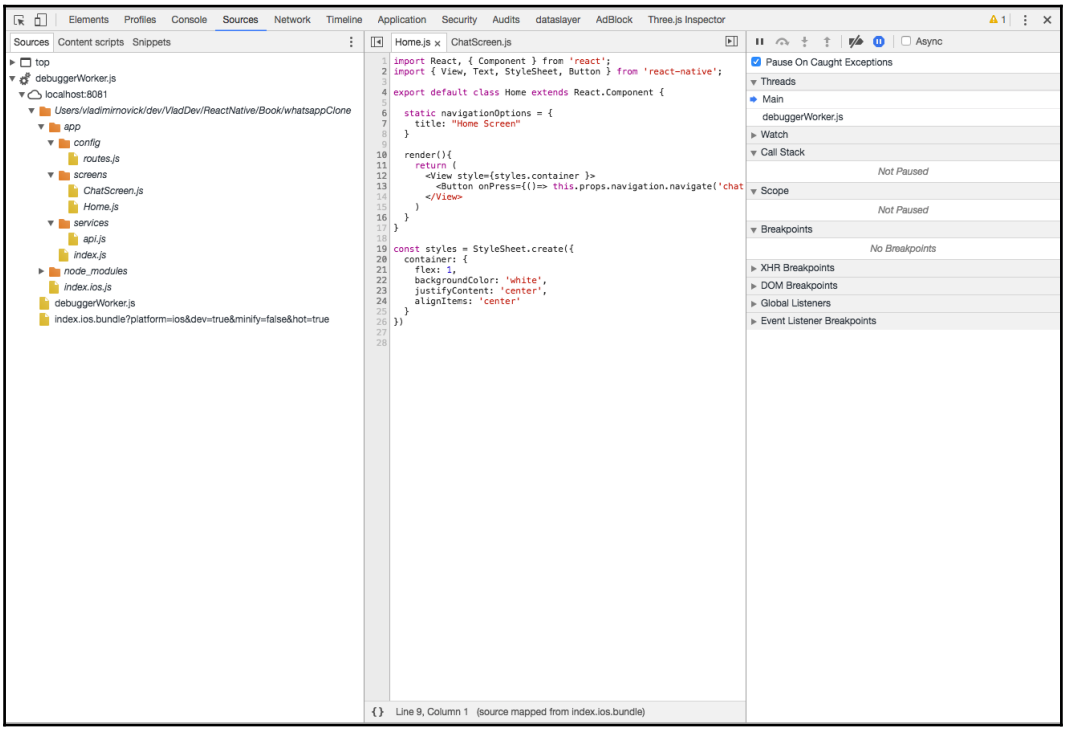
This text is for descriptional purposes, and the main usage of this newly opened Chrome Tab will be for Chrome DevTools. You can open it using *Command+Option+I* on Mac, *Ctrl+Shift+I* on Windows.

Instructions on how to open Chrome DevTools can be found at `https://developer.chrome.com/devtools`.

Let's open Chrome DevTools for our *whatsappClone* app. In the console, you will see the following:



These are the first things that are logged into the console when your application starts. All the default parameters of your application are logged here. Let's check out the **Sources** tab:



Here, you can see your application structure on the left; search for your JavaScript files. In our example, you can see `Home.js`. Note that you will get the original code, so you'll also be able to debug your original code. You can set watches on the left, pause on caught exceptions (recommended), look at call stack, and do other things that you are familiar with from web development.

Debuging our device

Sometimes it's important to run your application on a device especially when dealing with animations. Animations can look a bit sluggish in simulator since it's simulated environment after all. On the device you will see the real application behavior.

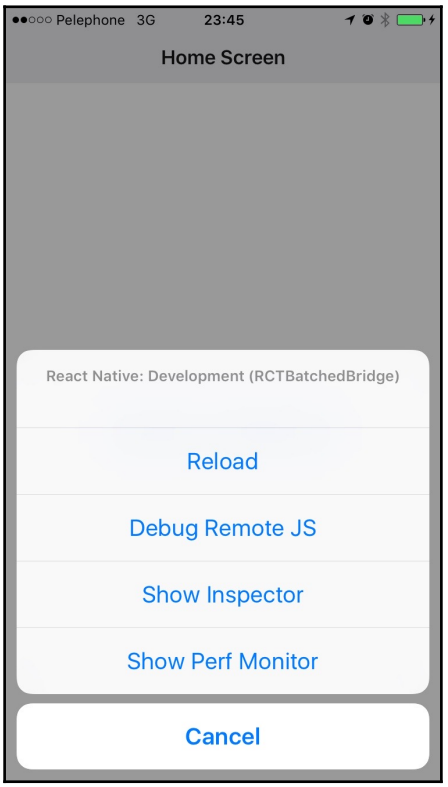
Since debugging on a device constantly changes for both iOS and Android with the release of new iOS/Android/React Native versions, we won't cover this in this book. The official docs are constantly updated, so I advise you to check there for up-to-date instructions. So, check at: `https://facebook.github.io/react-native/docs/running-on-device`.

If you've created your app by using **create-react-native-app**, you can debug your application on the device right away inside *Exponent* app. You can read about it in the Expo docs here:

`https://docs.expo.io/versions/latest/guides/debugging.html`

Note that running your app on iPhone requires an Apple developer account. Also, you have to be on the same Wi-Fi network as your Mac.

After you've built your app and deployed it to your device, open it in the app developer menu by shaking your device. When shaking it for the first time, you will get a limited in app menu as follows:



You need to choose **Debug Remote JS**, open Chrome on your computer, and navigate to `http://localhost:8081/debugger-ui`.
You will get the same experience as if you've been running your app in a simulator.

Logging

It's important to know what's happening in your application under the hood. Also, sometimes you want to know what exactly the error means or what is printed in the system log. You can access these logs by typing one of the following commands in terminal:

- **For iOS:** `react-native log-ios`

- **For Android:** `react-native log-android`

The other option is to access logs via a simulator menu on iOS by navigating to **Debug** | **Open System Log** or by typing the following command on Android, when the Android virtual device is running:

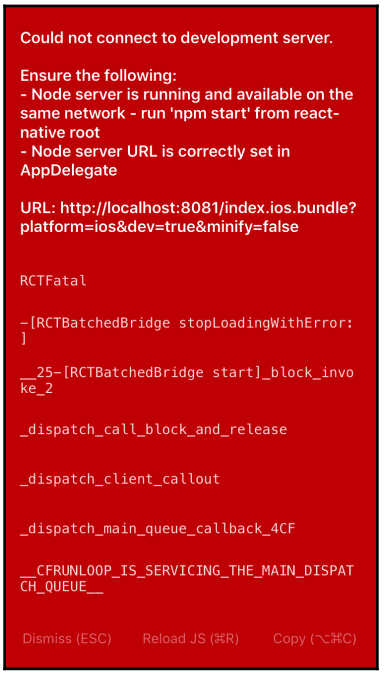
```
adb logcat *:S ReactNative:V ReactNativeJS:V
```

In-app errors and warnings

During development of your application, encountering errors and warnings is a common thing. In React Native, errors can look intimidating, but in fact they are really descriptive and useful.

Errors

Errors in React Native app are seen as full screen errors on a red background. They usually provide lots of details and all the crucial data we need to pinpoint the problem. Let's take a look, for example, at the following error:



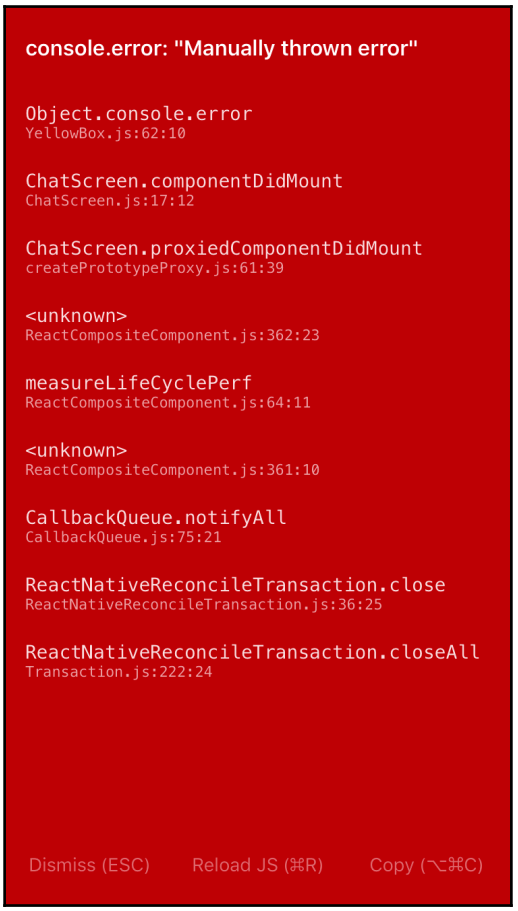
Not only do you get an idea of what the error is, you also have suggested ways to fix the problem. In this case, the error happens because the packager stopped running. You need to start it by running `npm start` as suggested in the error description.

Let's take a look at another example of an error:

Here, we see that the problem is because React Native is unable to resolve `react-navigation` module. This error usually happens when you import the npm package, but forgot to install it. As you can see in the error description, there are also three different suggestions of how to fix this error. Also, you have a link to the issue, to which an error can be connected.

Errors can be triggered manually by running `console.error()`.

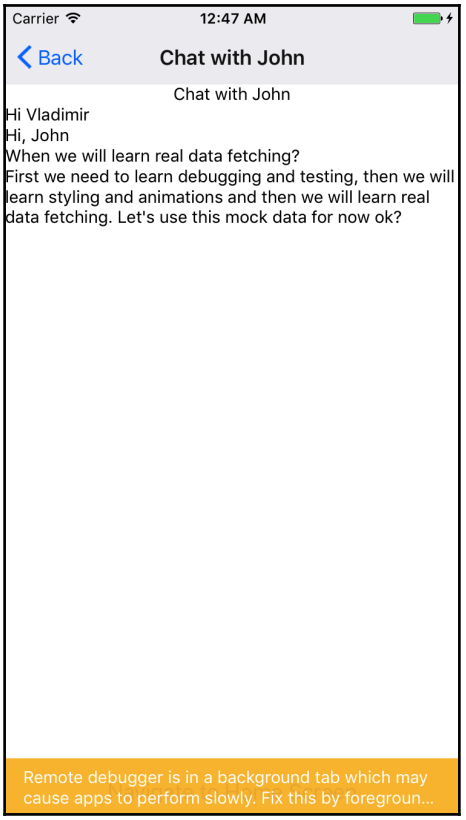
Let's take a look at it:



As you can see here, by looking at the errors shown in the preceding screenshot, we can tell exactly where it originated from. We can see the method it was thrown from as well as the file and line/column number. Every part of `stackTrace` displayed in the error is clickable, and when it will open the relevant file when running in the simulator. So, clicking on `ChatScreen.componentDidMount` will bring us to the exact line where the error occurred.

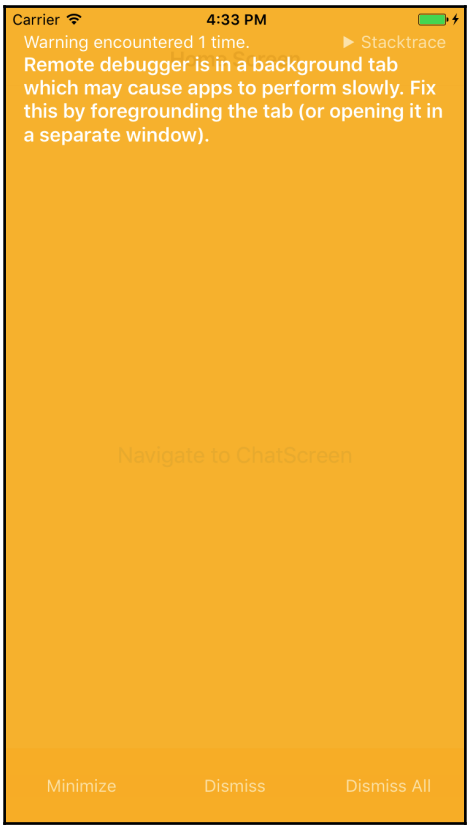
Warnings

Warnings are usually rendered using YellowBox, as follows:



YellowBoxes are also clickable, but clicking on it will lead it to be full screen, so you could see the whole description of the warning. These warnings, as well as errors, can be triggered with `console.warn` instead of `console.error` and usually also provide important information on how to get rid of the warning.

For example, Remote debugger warning when clicked will lead to the following screen:



In the previous example, the warning exists because remote debugger (Chrome DevTools) is in the background. This may cause apps to perform slowly, so always ensure that your remote debugger is running on the foreground tab. The best way will be to open it in a separate window.

YellowBoxes also can be disabled in development, though I don't encourage you to do so. The warnings are usually pretty important.

To disable warnings set, run the following command:

```
console.disableYellowBox = true
```

You also can ignore specific warning messages using:

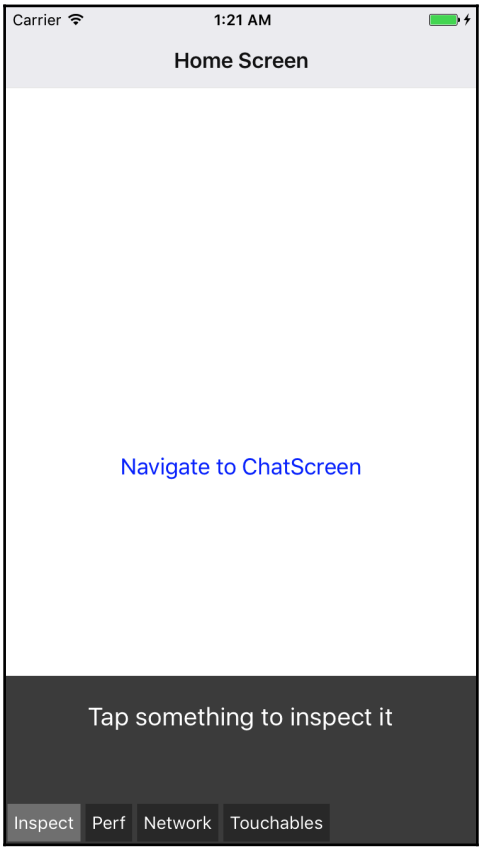
```
console.ignoredYellowBox = ['Remote debugger'];
```

In the previous example, we've disabled the Remote debugger warning.

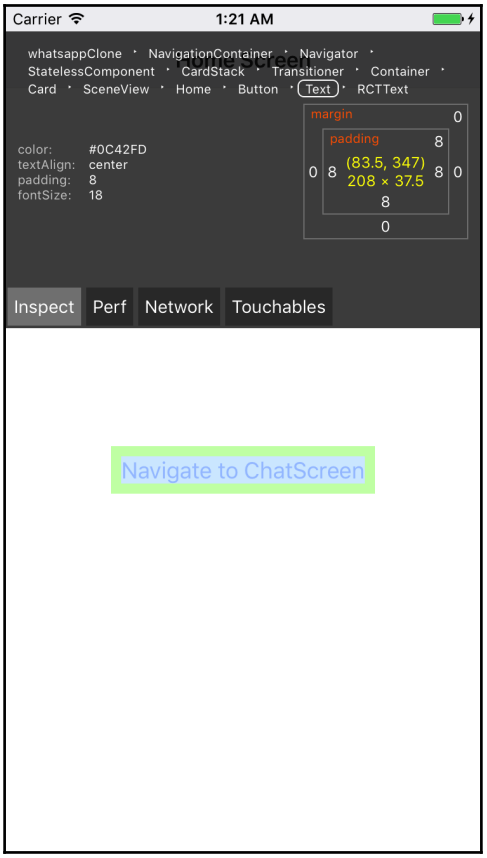
Note that RedBoxes and YellowBoxes are automatically disabled in release (production) builds.

Inspecting React Native components

When developing in the React ecosystem, it's important to know the component hierarchy. In web development, we are used to inspecting our HTML code using Chrome DevTools. In React Native, inspecting our app layout is also possible using a built-in inspector. To show inspector, select the **Show Inspector** option inside the developer menu. You will be presented with the following screen:



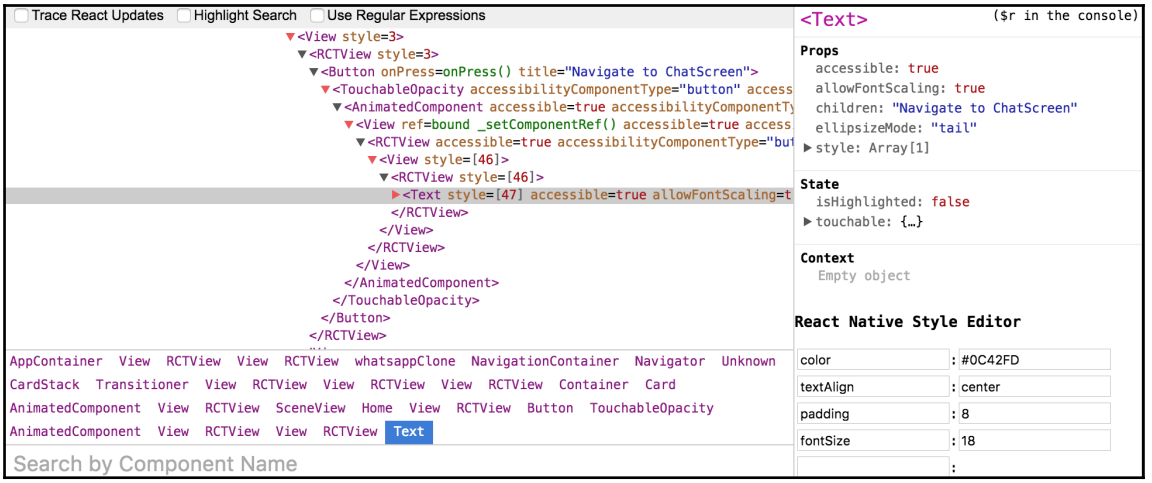
Let's, for example, follow the instructions shown on the screen and select the **Navigate to ChatScreen** button. We will be presented with the following screenshot:



As you can see from the preceding screenshot, we have "box model" for our button, showing what exact margins and padding it has. Also, we see all component hierarchies on top. We can click on each parent or child component and check its style similar to what we would do in web development.

There is another option to show a more robust inspector. If you want to use Atom editor and Nuclide IDE package on top of it for developing React Native apps, you can go to the **Menu** | **Nuclide** | **React** | **Toggle Inspector**.

You will see the following:



Here, you can use the combined functionality of Chrome DevTools and React Native in-app Inspector. In addition to being able to debug styles, you are able to set new styles on the fly, similar to web development.

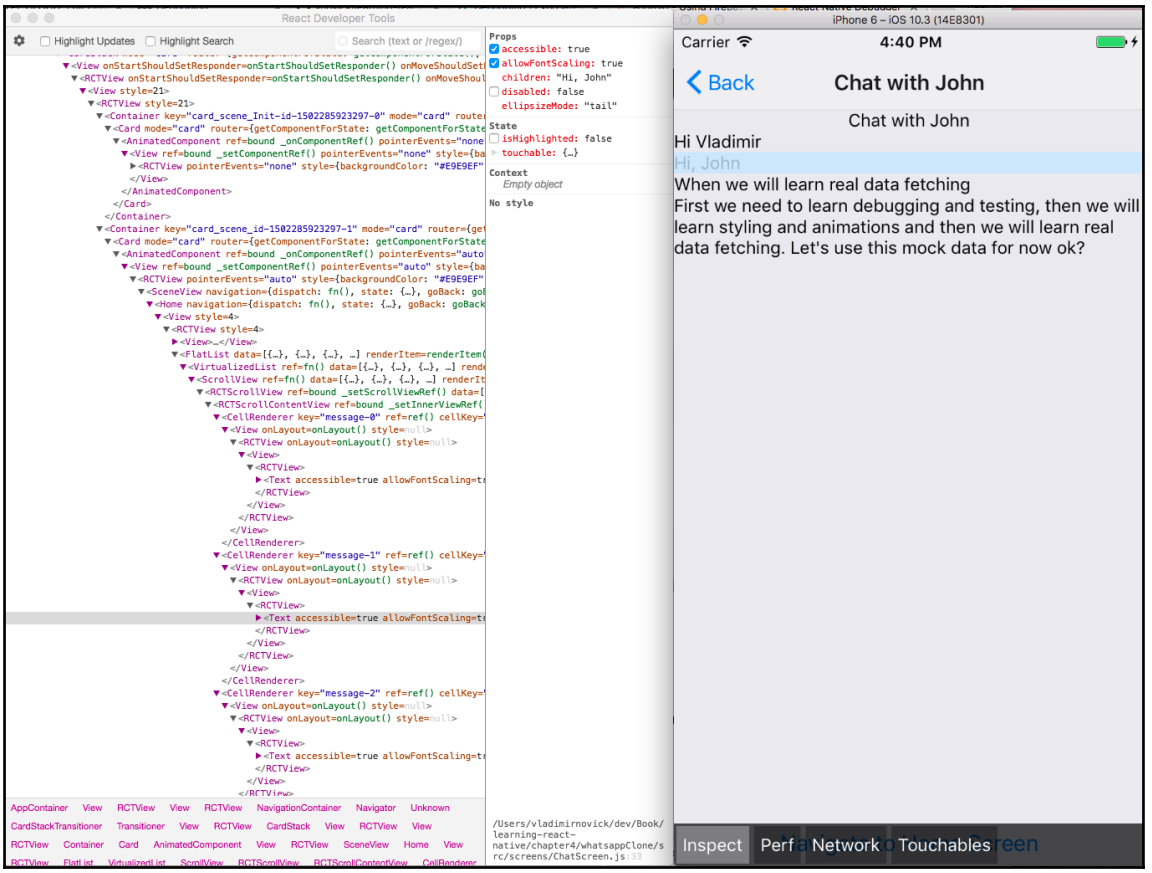
There is also a standalone dev tools project that you can use with React Native and in general with React applications. You can install it globally:

```
npm install -g react-devtools
```

And then open by running the following command in terminal:

```
react-devtools
```

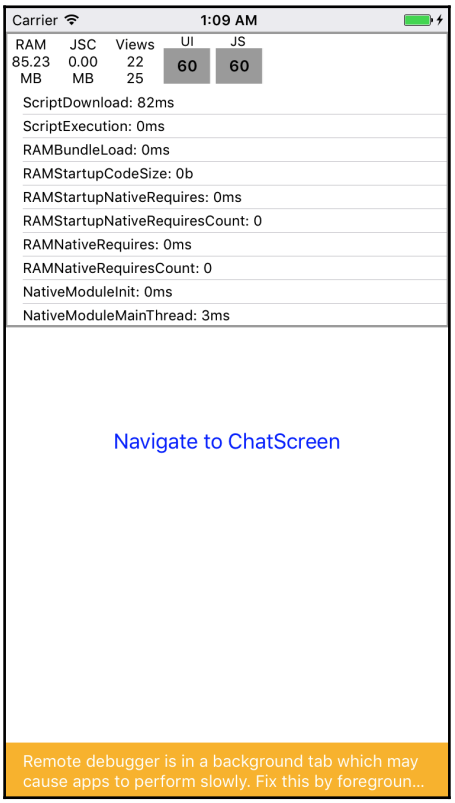
When it opens up, it connects to your React Native app and let's you inspect your React Native components in the same way we've inspected them inside Nuclide. For example, inspecting components of our *whatsappClone* app running in simulator (on the right) next to React Dev tools on the right will look like this:



Performance monitoring

Usually, when you deal with performance issues, you will check Chrome DevTools profiler; however, React Native also ships with a built-in performance monitor. You can access it by selecting Perf Monitor inside the in-app menu.

This monitor provides low-level performance metrics of your app, such as the RAM used, number of rendered Views, and so on. It's also expandable and shows more metrics when expanded:



Now, after you know the set of tools given to you by the React Native team, you can be confident that if you encounter an error, you will know how to fix it. Ensure that you check the official docs on the debugging topic, since new tools are added all the time:

- **Debugging**: `https://facebook.github.io/react-native/docs/debugging`
- **Performance**: `https://facebook.github.io/react-native/docs/performance`

Testing

Being able to test your application is crucial for smooth development workflow. You want to release features with as few bugs as possible, so it's important to write unit tests to test both your logic and rendering part. Usually, setting and configuring the testing framework properly is a nontrivial task; however, with React Native, we have a built-in testing mechanism. After you create your first app by running the following command: `react-native init myFirstApp`, you already have tests configured as well as several basic unit tests which reside in the `__tests__` folder. By running `npm test`, you can run tests using the Jest framework from Facebook.

Introduction to the Jest testing framework

Jest is a really powerful testing framework with several exclusive features that make our testing painless and straightforward, making development workflow smoother than ever.

In this section, we will cover important features of the Jest framework as well as useful community tools; however, testing is a stand-alone concept--which if you are not familiar with--requires more studying from books or the internet. I will provide you with most resources that you will need to know Jest testing as well as testing specifics for React and React Native.

Setup

In React Native, Jest is included by default inside your React Native installation. This means that as described previously, you can simply `run npm test` in your app folder, and your Jest will start your tests.

Preset system

Jest has an extensive preset system. These presets define how Jest will work in your environment. These presets are preconfigured environment or configuration options that can be imported into the Jest ecosystem. After initializing your React Native app, you can see the following code in `package.json`:

```
"jest": {
 "preset": "react-native"
}
```

This means that React Native is shipped with the `react-native` Jest preset. This preset basically is Node.js environment that mimics the React Native app. It does not load any DOM or browser APIs.

By default, React Native preset processes only project source files; however sometimes you need to import npm packages that should be processed as well. In that case, there is a `transformIgnorePatterns` configuration array that can be added next to "preset" in the `package.json` file.

Let's take a look at our *whatsappClone* example. When running npm test, we will get the following error:

```
MyPath/learning-react-native/chapter4/whatsappClone/node_modules/react-
navigation/src/navigators/StackNavigator.js:3
    import React from 'react';
    ^^^^^^
    SyntaxError: Unexpected token import
```

The error is caused because we have also to transpile react-navigation. Let's add`transformIgnorePatterns` to our project. `package.json` section for Jest will look like this:

```
"jest": {
  "preset": "react-native",
  "transformIgnorePatterns": ["node_modules/(?!react-native)/"]
}
```

Now running `npm test` will result in different errors, however now it's transpiled correctly. We will deal with the error in the following sections.

There are additional options when configuring your presets that you can check in the Jest official getting started guide for React Native:

`https://facebook.github.io/jest/docs/tutorial-react-native#preset-configuration`

Basic syntax

Jest is built on top of the Jasmine testing framework, so in case you are familiar with it, the syntax will be familiar to you. First, let's understand how Jest knows which tests to run. When you execute `npm test`, you basically execute `jest test` using the npm script. Jest looks for all files `under __tests__` directories or files with `.test.js` extension and runs them. Let's check our *whatsappClone* application and see what tests we have already generated by initializing React Native apps.

Taking a look at the folder structure, you can note that the `__tests__` directory has two files inside--`index.ios.js` and `index.android.js`.

Let's check the code of `index.ios.js`.

First, you will see set of imports that bring all needed dependencies as well as the component itself:

```
import 'react-native';
import React from 'react';
import Index from '../index.ios.js';

// Note: test renderer must be required after react-native.
import renderer from 'react-test-renderer';
```

The first test that was already written for us basically checks whether we have errors when rendering our root component to the screen. As you can see, similar to all testing frameworks out there, you wrap your test in the `it` function or in case of Jest you can also use `test` instead.

Then, we call `react-test-renderer`, which was installed together with our React Native installation. This renderer renders our component to Jest environment, so we can check it later on using the `expect` function:

```
it('renders correctly', () => {
  const tree = renderer.create(
    <Index />
  );
});
```

For our *whatsappClone*, these two tests won't work. That's because we import Index from `index.ios.js` and it does not have an export. Our index files just register our App from `src/index.js` folder.

Let's change the code so our tests will pass.

```
import 'react-native';
import React from 'react';
import App from '../src';

// Note: test renderer must be required after react-native.
import renderer from 'react-test-renderer';

it('renders correctly', () => {
  const tree = renderer.create(
    <App />
```

```
    );
});
```

Now let's create the test files. You can create files with a `.test.js` extension and put it adjacent to files you are testing or you can create a `__tests__` folder in the same place. Let's now create a `__tests__` folder.

Let's create the following files:

```
config/__tests__/routes.js
screens/__tests__/Home.js
screens/__tests__/ChatScreen.js
services/__tests__/api.js
```

Let's, first, check whether we have all needed routes. This test will go inside the `config/__tests__/routes.js` file. Here, we will import both routes and our screens and check whether the routes list matches what we would expect:

```
import 'react-native';
import React from 'react';
import routes from '../routes';
import Home from '../../screens/Home';
import ChatScreen from '../../screens/ChatScreen';

it('has all needed routes', () => {
  expect(routes).toEqual({
    home: { screen: Home },
    chat: { screen: ChatScreen }
  });
});
```

Let's now check explicitly whether each route has a proper component defined:

```
it('has home screen', () => {
  expect(routes.home.screen).toBe(Home);
})
it('has chat screen', () => {
  expect(routes.chat.screen).toBe(ChatScreen);
})
```

Our tests are passing, however, if we change our routes list, we need to go to the test and change our assertion too. This is fine for such a small component as routes, but for nested objects or even components, it can become difficult to check all structures. That's why, Jest came up with the snapshot testing idea.

Snapshot testing your React Native components

The idea behind snapshot testing is that instead of manually checking an object or component for equality with hardcoded data, we can execute the `toMatchSnapshot` function. As a result, the snapshot file will be generated on disk under the `__snapshots__` directory adjacent to your test. Snapshot file contains the structure of your tested component object or function.
If you later change the code and snapshot is different, then your test will fail, and you will need to manually update your snapshot or execute `npm test -- -u` to update all failing snapshots.

Let's refactor our routes test to use the snapshot testing feature:

```
it('has all needed routes', () => {
  expect(routes).toMatchSnapshot();
});
```

Note that when running this test for the first time, snapshot will be generated under `config/__tests__/__snapshots__/routes.js.snap`.

Snapshot testing also lets you properly test your components. Let's test our Home screen in our `screens/__tests__/Home.js` file:

```
it('renders correctly', () => {
  const tree = renderer.create(
    <Home />
  );
  expect(tree).toMatchSnapshot();
});
```

Here, we will check whether our component tree is rendered correctly. After running test, it will generate a snapshot file with all component hierarchy:

```
it('has proper navigation options', () => {
  expect(Home.navigationOptions).toMatchSnapshot();
})
```

We can also use snapshots to check simple objects. In this case, we check whether `navigationOptions` are defined correctly in home screen.

Let's take a look at snapshot of screens `/__tests__/__snapshots__/Home.js.snap`:

```
exports[`has proper navigation options 1] =
Object {
  "title": "Home Screen",
}
;
```

Working with functions

In our Home component test, we tested everything, except the fact that if you click on the **Navigate to Chat Screen** button, navigation is executed with proper parameters.

In order to do so and also for convenience, we will install a really useful package that lets us shallow render our components in a slight different way. It also gives us the ability to search inside our components. This is the use case that we need. We need to find the `Button` component inside our home screen and call the `onPress` function. We also need know if it was called with proper arguments:

```
npm i enzyme react-dom --save-dev
```

At the moment of book publishing, enzyme does not work out of the box and you will probably get the following error:

```
react-dom is an implicit dependency in order to support react@0.13-14.
Please add the appropriate version to your devDependencies.
```

In order to proceed, go to `package.json` and change `react-dom` version to match `react` version. In our case react version is **16.0.0-alpha.12**.

Change `react-dom` version to **16.0.0-alpha.12** and run:

```
npm install
```

Now, we can continue to write our test for Home screen. First of all lets add the following:

```
import { shallow } from 'enzyme'
```

And now the code:

```
it('navigate on press', () => {
  const navigate = jest.fn();
  const homeScreen = shallow(
    <Home navigation={{ navigate }}/>
  )
```

```
    homeScreen.find('Button').simulate('press');
    expect(navigate).toBeCalledWith("chat", {"name": "John"});
  })
```

Here, we use `jest.fn()` to create a Jest `spy` function. This function lets us check whether navigation was executed. Since inside our `onPress` function we use `this.props.navigation.navigate`. We pass it as a prop to our Home component.

Note that instead of `createRenderer` we use `shallow` provided by enzyme. By using it, we can now call `homeScreen.find('Button')` to specifically target Button inside our Home screen component hierarchy. We call props function to retrieve its props and call `onPress()`.

Since `onPress` uses `this.props.navigation.navigate`, our `spy` function will be called instead of the real one. Now, we can check whether it was called correctly using this:

```
expect(navigate).toBeCalledWith("chat", {"name": "John"})
```

Our `Home.js` test is done. Now, we will do the same for our `ChatScreen.js` test file.

```
it('navigate on press', () => {
  const goBack = jest.fn();
  const chatScreen = shallow(<ChatScreen navigation={{ navigate }} /> )
  chatScreen.find('Button').simulate('press');
  expect(goBack).toBeCalledWith("home");
})
```

Also we need to check that navigation props are passed correctly:

```
it('ChatScreen has proper navigation options', () => {
    expect(
        ChatScreen.navigationOptions({
            navigation: {
                state: {
                    params: {
                        name: 'John'
                    }
                }
            }
        }
    )).toMatchSnapshot();
})
```

Since `navigationOptions` is a static function we call it directly on ChatScreen class. We pass proper navigation state structure to this function and we expect to get "chat with John" heading.

Let's take a look at generated snapshot:

```
exports[`ChatScreen has proper navigation options 1`] = `
Object {
 "title": "Chat with John",
}
`;
```

We almost finished writing tests, however, we haven't checked our API call. Let's do it in two steps.

First of all, we need to write a test for the `api.js` file. We will put it under `services/__tests__/api`.

Here, we will also use Jest snapshot feature to check whether we get all messages correctly:

```
import 'react-native';
import React from 'react';
import { getMockData } from '../api';
it('resolves to mock data',async () => {
  const messages = await getMockData();
  expect(messages).toMatchSnapshot();
});
```

We get `getMockData` from `api.js` and then wait for the promise to resolve, then execute `toMatchSnapshot` to check whether the message list is correct.

Note that here we use ES7 async await. React Native uses not only ES6, but some selected features from ES7. Async await is one of them. Basically instead of calling `getMockData().then` as we would have done with a regular promise and by calling `done()` to ensure that test is complete.

Instead, we define our function as `async`, making it possible to call `await` inside of the function. If we would have written our test without async await it would have looked like this:

```
it('Api returns mock data correctly', () => {
    return getMockData().then(result => {
        expect(result).toMatchSnapshot();
    });
});
```

However, to be more clean, we will use `async await` from now on instead of `promise.then`, because it's considered as a best practice in React Native and makes your asyncronous code more descriptive as it's read synchronously.

So now, we have almost finished creating tests for our *whatsappClone*. However, there is one important test missing. We want to know that the `getMockData` function is actually called when we mount our component.

Mocking modules

Jest has an amazing mocking feature. You can both create your mocks inside the `__mocks__` directory adjacent to mocked file or create them programmatically.

Let's create the `services/__mocks__/api.js` file for our mock.

Let's export the spy function we would use in our test:

```
export const getMockDataSpy = jest.fn();
```

Now, let's create the actual mock:

```
export const getMockData = () => {
    return new Promise(resolve =>
      getMockDataSpy(() => resolve(mockMessages)))
}
```

Since our `getMockData` is returning a promise, we will return a `new Promise` and inside it--we will call `getMockDataSpy` passing a callback function and resolve.

Now, in our `ChatScreen` test, we will; need to do two things. First, enable our mock using `jest.mock('../../services/api');` this will tell Jest to include our mock file in every place where there is an import from `services/api`.

Second, we also import the `getMockDataSpy` function from `services/api`, and since Jest knows to include our mock file, it will import the correct `spy` function.

```
import { getMockDataSpy } from '../../services/api';
jest.mock('../../services/api')
```

All we are left to do is to write our test, which is pretty straightforward:

```
it('fetches data when component is rendered', () => {
    const chatScreen = renderer.create(<ChatScreen />);
    expect(getMockDataSpy).toBeCalled();
})
```

There is even more information available here, so ensure that you check out the Jest React Native guide at `https://facebook.github.io/jest/docs/tutorial-react-native`, Jest API docs at `https://facebook.github.io/jest/docs/api`, and Enzyme shallow rendering API at `http://airbnb.io/enzyme/docs/api/shallow`.

Summary

Congratulations! You've finished the first part of *Getting familiar with React Native*. You've learned a lot in previous chapters about how React Native works and how to set up your development environment for both iOS and Android. You've seen a comprehensive overview of all React Native components and now you've learned how to debug and test your application properly. You've learned various debugging techniques that make your application development workflow enjoyable and painless. In addition to understanding how to debug your app, you now know also how to test it properly using the Jest testing framework by Facebook as well as other tools. Now, when you have all these set of tools, it's time to get to the fun and visual stuff. In the next chapter, we will discuss styling your components. We will transform our app to look and feel like WhatsApp after we understand the main concepts of styling in React Native. So, see you in the next chapter.

5

Bringing the Power of Flexbox to the Native World

Welcome to the fifth chapter. In this chapter, we will start by introducing concepts of flexbox, the new layout system used in CSS for all modern browsers, and an overview of how we can position elements on screen using it. We won't dive into bytes and bits of flexbox for the web, but we will understand the important concepts. Then, we will learn how React Native leverages the flexbox system to layout your apps with CSS-like styles. We will provide an overview of the various style properties familiar to you from CSS and will understand how they are different in React Native. Then, we will cover best practices and techniques on how to build your styles in a more scalable and maintainable way. Throughout the chapter, we will transform our newly created WhatsApp application clone to be visually close to WhatsApp itself. After you finish reading this chapter entirely, I challenge you to style it further to achieve the look and feel you would have in WhatsApp.

So, let's summarize what you will learn in this chapter:

- Flexbox styling concepts and techniques
- How React Native styling is different from CSS or inline styles
- Best practices and techniques for styling your React Native applications

Flexbox styling concepts and techniques

In the mobile development world, styling has always been much harder than on the web, especially when modern web brought in flexbox layout system. React Native brought with it web style CSS-like styling. It uses parts of the flexbox layout system to layout elements on screen. Let's see an overview of its basic concepts.

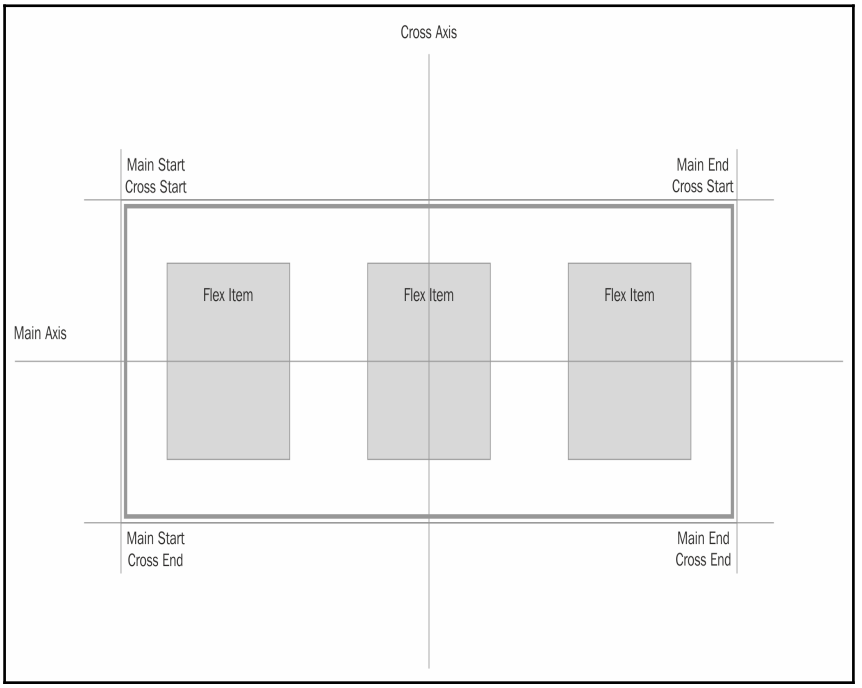
For those who want to learn more about flexbox for web or just for reference, check out `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Advanced_layouts_with_flexbox` (the Mozilla Development Network docs).

The flexbox layout

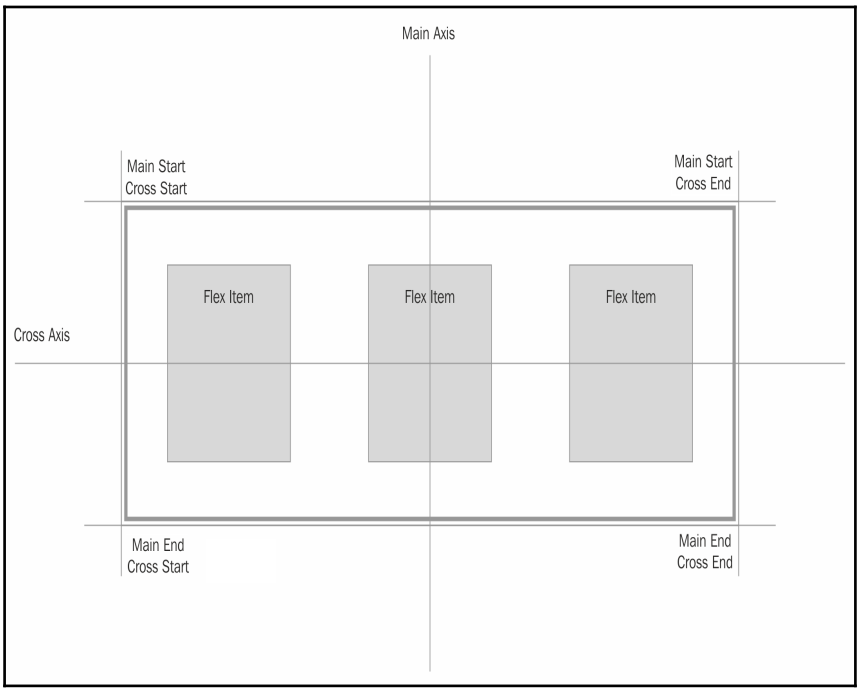
In web, flexbox is about creating an element with a `display: flex` property and adding children to it. When you do so, created element becomes flex container inside of which we can manipulate children with flexbox layout techniques.

When we create such a container, it lays out its children on two axis: **Main Axis** and **Cross Axis**. These are two axis perpendicular one to each other and they both create a coordinate system for the flexbox layout.

This axis can change as a result of the `flex-direction` property (by default, its value is `row`). This property tells browser to align items in flex container horizontally. Check out the following scheme:



If you change `flex-direction` to `column`, you basically switch between Main Axis and Cross Axis, so the scheme will look like this:



If you treat these axes like an X and Y axis in the Cartesian coordinate system and you have `flex-direction: row`, then Main Axis is X axis and Cross Axis is Y axis. If you switch to `flex-direction: column`, however, then Main Axis becomes Y and Cross Axis becomes X.

Aligning elements

Okay, so after we've got an idea of how Main Axis and Cross Axis looks, let's take a look at how we align our items. We have the following two properties:

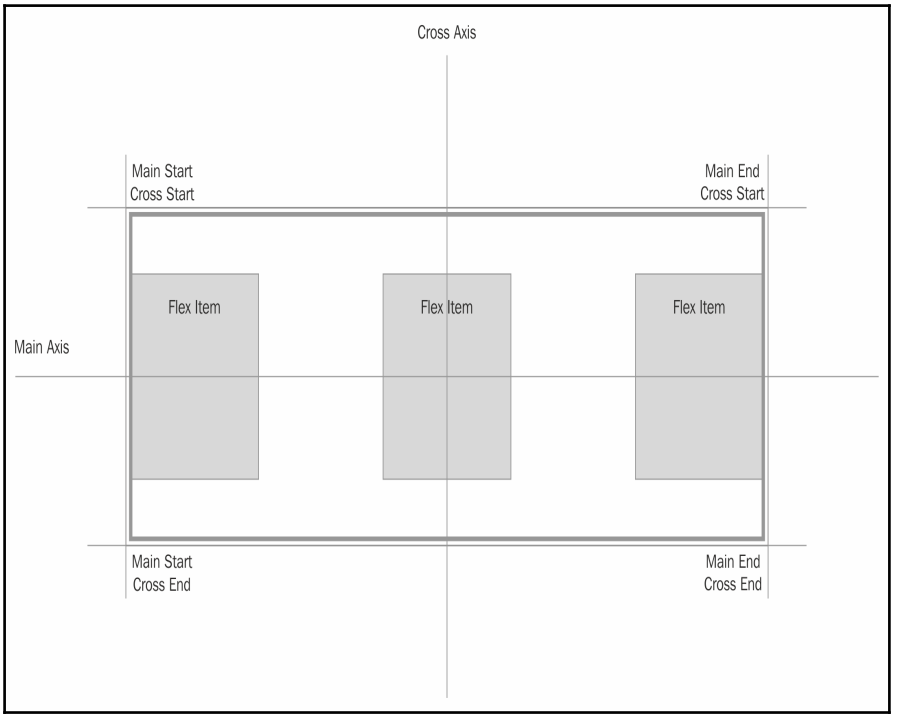
1. `justify-content`: Aligns child items across Main Axis
2. `align-items`: Aligns child items across Cross Axis

Both of these properties can get the following values:

- `flex-start`: Aligns items to Axis start
- `flex-end`: Aligns items to Axis end
- `center`: Aligns items to Axis center

In addition to aligning items to start, end, or center of the axis, we can distribute them evenly using the following values:

- `space-around`: Items are distributed evenly with equal spacing between them and the same space before the first and the last items as shown in the preceding scheme
- `space-between`: Items are distributed evenly with the first item starting on axis start and last item ending on axis end, as shown in the following image:



- `stretch`: Items are stretched to fit all of the available space

Sometimes, we want to position a specific element differently from what is defined in its parent. We can do so using the flexbox `align-self` property.

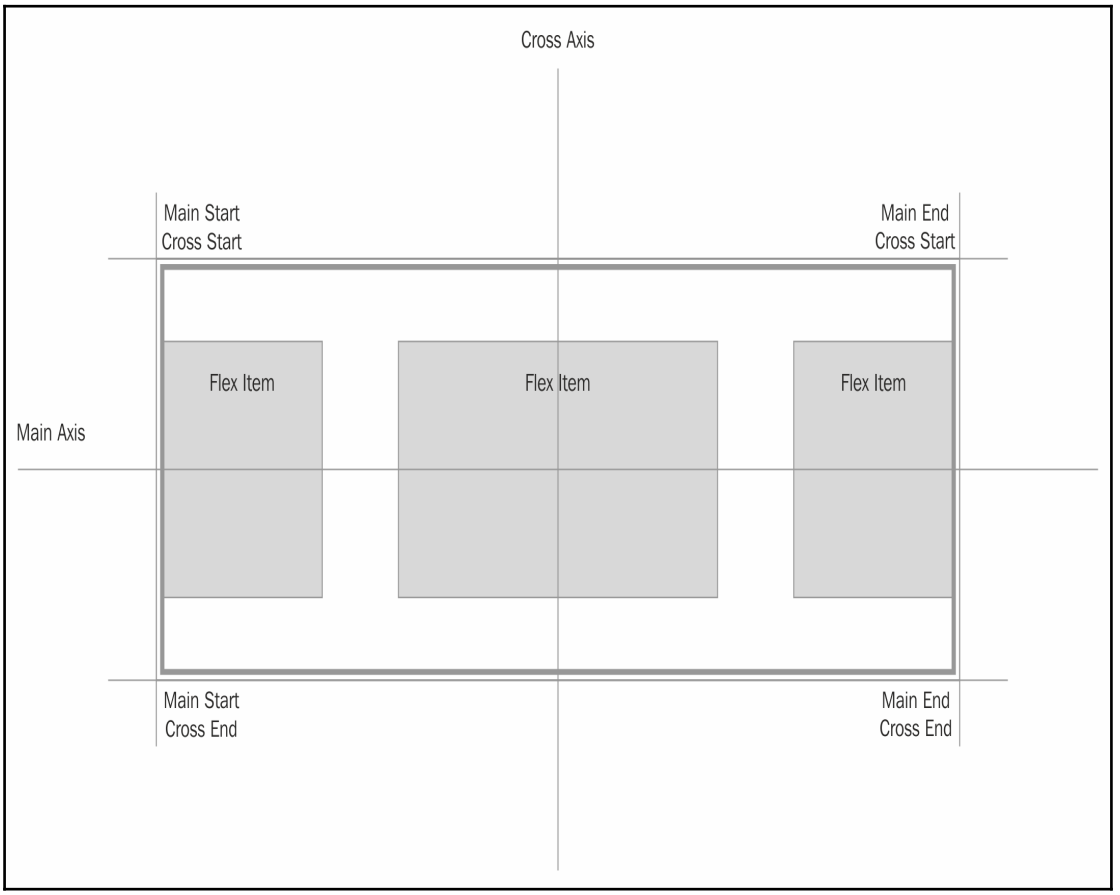
Item dimensions

With flexbox, we can specify the width of our items using the `flex` property. `flex` accepts numeric values, for example, setting:

```
flex: 1
```

This will expand an element to the full available width along the Main Axis.

For example, if in the preceding scheme we want any two flex items to take twice as much space as the other item's width, we will set `flex: 1` on the first and third items and `flex: 2` on the second item. The result will look like this:



There are also lots of other features in flexbox that can be used on Web, but since React Native use only a subset of flexbox layouting, understanding the basics would be enough to start talking about how flexbox in React Native is different from flexbox layout for web.

How different is React Native styling from CSS or inline?

Okay, so, we've talked about layouting things on Web with flexbox, and the tool that comes to our mind when thinking about styling on web in general is CSS. In React Native, we don't have CSS since we are dealing with native apps, which don't behave similar to the web. When we look at React Native styling for the first time, it looks much like inline styles, just as we use them in React on the web. Consider the following:

```
<Component style={{ borderWidth: 1, borderStyle: 'solid'}} />
```

This looks similar to CSS, but it's not. It has some key differences from CSS in addition to the fact that their properties also behave a bit differently. The differences between React Native styles and inline styles are as follows:

1. Style properties are dealt in the same way as in React. dash-delimited values are camelCased (for example, `background-color` turns into `backgroundColor`).
2. All React Native elements are flex containers, by default. It means that you will never see the `display: flex` style definition in React Native apps.
3. The `flex-direction` property is, by default, `row` on web, but in React Native it's, by default, `column`. That means that vertical axis is the Main Axis and horizontal axis is Cross Axis.
4. There are no such things as pixels, ems, rems, or other units in React Native. So, in the preceding example, `borderWidth: 1` is a valid code and `borderWidth: '1px'` is an invalid code for React Native.
5. Even though on web, `background-image` is a common CSS property; in React Native, you cannot use it.

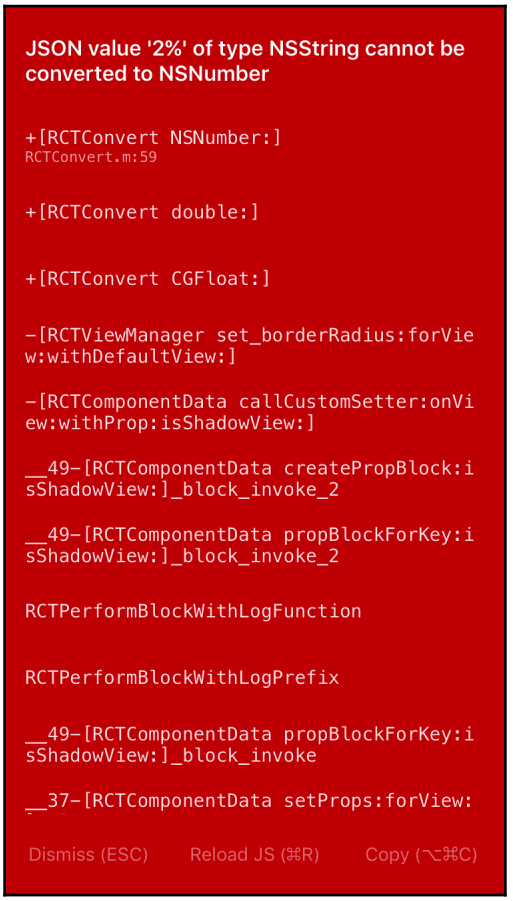
If you do need `background-image`, just do the following:

```
<Image source={...}>
   <YourComponent/>
</Image>
```

In latest React Native version you should use **ImageBackground** instead, so your code will look like this:

```
<ImageBackground source={...}>
  <YourComponent/>
</ImageBackground>
```

6. Percents are accepted only for `width` or `height`. You cannot pass percent to properties such as `borderRadius`. If you pass a border radius, such as `borderRadius: '2%'`, you will get the following error:



As a rule of thumb, if you get this kind of error when defining styles, it means that you haven't passed a style value properly. Ensure that you pass numeric values as numbers.

7. Text component customization has some platform-specific props that can be used only in Android or in turn only on iOS. For a complete reference, you can check out: `https://facebook.github.io/react-native/docs/text.html#style`.

8. You can pass an array to style a prop if you have multiple styles, as follows:

```
style={[styles.container, styles.active ]}
```

9. Transforms, such as `scale` and `transformX`, are passed as an array of objects where each object is a transform, as follows:

```
transform: [{
  scale: 1
}]
```

Shadows are not supported currently on Android, but it can be used on iOS with the props available at `https://facebook.github.io/react-native/docs/shadow-props`. For now, on Android, shadows are patched with community packages, for example, `react-native-shadow`.

Laying out our app

After we've talked about the differences, let's walk through our WhatsApp app clone and see how we can style our app.

First, let's take a look at our Home Screen component `render` function:

```
<View style={styles.container }>
   <Button onPress={ () => this.navigate() } />
</View>
```

As you can see here, we pass `styles.container` to style prop. Let's take a look at this object:

```
const styles = {
  container: {
    flex: 1,
    backgroundColor: 'white',
    justifyContent: 'center',
    alignItems: 'center'
  }
}
```

As you can see, it's a plain JavaScript object. We tell our container to expand to the full available width along the Main Axis. Since our Main Axis is a vertical one, our container will take full screen height.

Then, we simply change our background color to white.

The next step is to layout our navigate to the **ChatScreen** button. This is done with the following properties:

```
justifyContent: 'center',
alignItems: 'center'
```

As we discussed previously, it means to position it at the center of Main Axis (`justifyContent`) and at the center of Cross Axis (`align-items`).

Dealing with background images

Now, let's take a look at our **Chat with John** screen (we just removed the navigate back button):

```
<View style={styles.container }>
  <View>
   <Text>Chat with John</Text>
  </View>
  <FlatList
    data={this.state.messages}
    renderItem={({ item }) =>
      <View>
        <Text>{item.message}</Text>
      </View>
    }
    keyExtractor={(item, index) => (`message-${index}`)}
  />
</View>
```

First of all, let's make it have a background like WhatsApp. Usually, images should be in the same place as our React Native components, however, sometimes, like in this use case, images can be reused across the app. Let's create a folder and name it `assets`. Inside of it, let's put a folder named `imgs` and copy our background there.

In React Native, since we don't have the `background-image` style property, we will wrap the whole content of our `ChatScreen` inside an `ImageBackground` - a component introduced in latest versions of react-native. So instead of starting with the following:

```
<View style={styles.container} >
//...rest of ChatScreen code
```

We will write this:

```
<ImageBackground
  style={styles.container}
  source={require('../assets/imgs/background.png')}>
  //...rest of ChatScreen code
```

Now, let's take a look at the result:

The following styles are used to achieve this result:

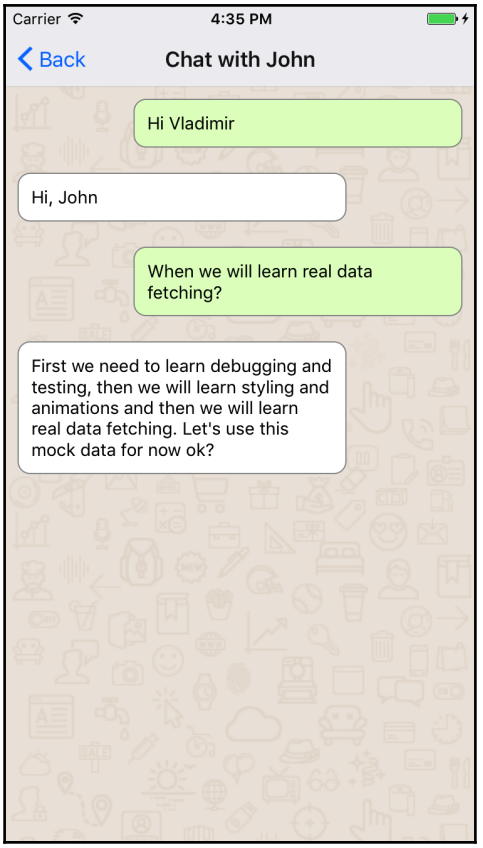
```
const styles = {
  container: {
    flex: 1,
    backgroundColor: 'transparent',
    justifyContent: 'center'
  }
```

```
}
```

Note that with images, as well as with new files you add to your application, you need to recompile it once again by running `react-native run-ios` or `react-native run-android`.

Applying styles conditionally

Now, let's add some messaging styling. Let's take a look at the result we want to achieve:



Let's think what information we can extract by looking at the design. We have a border, equal spacing between messages, margin from right and the left, border radius, and most importantly we have incoming messages on the right while outgoing are on the left.

First of all, we can clearly see that we don't need `justify-content` properties so we can remove them.

If you remember, in our messages data from the preceding chapter, we had the key incoming in our message object. That was for that particular reason, that is to align messages to right or to left..

We want to be able to make slight UI changes based on this flag. That means that we need to apply styles conditionally. Let's look at how we will do that in our `FlatList` component. We will also refactor it a bit to make it more readable:

```
<FlatList
      data={this.state.messages}
      renderItem={({ item }) =>
        this.getMessageRow(item)
      }
      keyExtractor={(item, index) => (`message-${index}`)}
/>
```

I've extracted the whole row rendering in a separate method:

```
getMessageRow(item){
  return (
    <View style={[
        styles.listItem, item.incoming ?
        styles.incomingMessage :
        styles.outgoingMessage
      ]}>
      <Text>{item.message}</Text>
    </View>
  )
}
```

Message rendering changed a bit. I still added `styles.listItem` to it, but this time I also checked the `item.incoming` flag and if it's true, then I will apply `styles.incomingMessage`; if not, I will apply `styles.outgoingMessage`.

The styles will be as follows:

```
listItem: {
    width: '70%',
    margin: 10,
    padding: 10,
    backgroundColor: 'white',
    borderColor: '#979797',
    borderStyle: 'solid',
    borderWidth: 1,
```

```
    borderRadius: 10
  },
  incomingMessage: {
    alignSelf: 'flex-end',
    backgroundColor: '#E1FFC7'
  }
```

The styles for `listItem` will be familiar to you if you know CSS. Note that `borderWidth` and `borderRadius` are exact numbers, though.

Even though we haven't passed the `alignItems` property, it's default is flex-start. That's why, for all our incoming messages, we need to override our `alignItems` with `alignSelf: flex-end` to position only incoming messages on the right.

As you can see, you can get a WhatsApp-like layout pretty fast just by following CSS rules with slight modifications. I challenge you to try and play with different properties and create different looks and feels for your WhatsApp-like screen.

For your reference, you can check the following links, which are a list of supported props:

- **Text style props**:
  `https://facebook.github.io/react-native/docs/text.html#style`
- **Layout style props**:
  `https://facebook.github.io/react-native/docs/layout-props.html`
- **Shadow props**:
  `https://facebook.github.io/react-native/docs/shadow-props`

Best practices and techniques for styling your React Native applications

In order to improve your styling skills and eliminate errors in your application, it's crucial to follow best practices. In the following paragraph, we will cover best practices you should always follow when styling your application. Some of them may not seem very important, but in big apps they've proven to be very helpful.

Dimensions

Setting hardcoded margin or padding values can be a good solution, but for item sizes, it's not so good unless you want to support different device orientations with different ratios. Usually, you want consistent UI language for both landscape and portrait mode.

Also, sometimes you just need to know what your screen size is. All of the above can be done using the React Native Dimensions API.

In order to get the height and width of your device, simply import `Dimensions` from React Native and get a height and width by calling:

```
const { height, width } = Dimensions.get('window');
```

It's advised to get and use height and width values as inline styles and not inside a style object passed to a style prop. You would usually want to get relevant dimensions even when they are updated.

You can also add an event listener to dimension change of your device to execute your custom logic:

`https://facebook.github.io/react-native/docs/dimensions`

Style sheet

React Native supplies an API to handle your styles even better than passing objects as we saw earlier. This can be done with the `StyleSheet` API. For the basic usage, it may seem seamless. Consider the following:

```
const styles = {
  container: {
    flex: 1,
    backgroundColor: 'white',
    justifyContent: 'center',
    alignItems: 'center'
  }
}
```

Instead of this, on our Home screen, we will import style sheet from React Native and then use it like this:

```
const styles = StyleSheet.create({
   container: {
     flex: 1,
     backgroundColor: 'white',
```

```
        justifyContent: 'center',
        alignItems: 'center'
      }
    })
```

We won't see differences for basic usage, since style sheet supplies us with lots of methods to be used for more advanced usages. For the list of methods, you can check out `https://facebook.github.io/react-native/docs/stylesheet`.

The most important thing you get from writing your styles with `Stylesheet.create` is that under the hood, an ID is created for your style and your style is cached; if you pass objects, your style object is created on every render, which is not a good thing performance wise.

In the future, there is a proposed feature to cache style sheet styles even further, so the styles would be sent only once through the Bridge.

Split your styles

While in React Native, you can pass your styles directly to styles prop; however, in terms of future maintenance and code readability, it's not considered a good practice. That's why, it's advised to separate your styles.

Consider doing following things as your application grows:

- Put your component-related styles in separate files or at the end of the same file
- Extract common styles to a different directory, extract all application colors to a different file, for example, `styles/colors.js` or `styles/variables.js`

Code your styles

Don't forget that your styles are written in JavaScript and not in CSS. That means that you can code them:

- Using Dimensions API as I've mentioned earlier, it's easy to calculate your styles based on screen dimensions
- Calculate your styles based on your component props
- Apply styles conditionally
- Use community packages to manipulate color, check out `https://www.npmjs.com/package/color for reference`

You have unlimited possibilities to make your component styles, not only work on both device orientations, but be robust, maintainable and reusable in your current and even future apps.

Extract common components

While building your application, you will notice that there are styles that repeat themselves. Don't forget that you can extract these styles for future use.

For example, you've already noticed that we had `flex: 1` repeated several times across our app. We can extract it to a separate style and use it across our application.

Summary

In this chapter, we've learned how to make your components visual compelling. We started with understanding the flexbox layout algorithm and followed with differences between React Native styling and CSS. They are very similar, however, we covered important differences between the two. Then, we worked a bit on our *whatsAppClone* application to make it more attractive visually. After completing the WhatsApp Chat screen look and feel, we've covered techniques and tips for styling our React Native applications. Now that we reached the end of this chapter, I challenge you to take all that you've learned by now and try to create the *whatsAppClone* app yourself. We focused only on the chat screen.

In `Chapter 6`, *Animating React Native components,* we will continue creating our WhatsApp application, but this time we will make it look even greater by learning how to use animations in React Native. However, before going into the next chapter, I suggest that you try out creating different screens with different layouts. After all, the best way of learning is by trying things yourself.

6
Animating React Native Components

Welcome to the sixth chapter. In this chapter, we will focus on how to animate your components with APIs supplied by React Native. We will start by understanding animation--how it works, what are animation-easing functions, and how they look like on the timing graph. Then, we will cover the LayoutAnimation API provided by React Native. This is a poorly documented React Native feature, yet very powerful for certain tasks, so we will dive into bytes and bits of how it works and see some examples from React Native source code on how it can be customized to fit our needs. After getting proficient with the LayoutAnimation API, we will dive into the juggernaut of the Animated API. This API lets you animate anything in any way you want. We will understand the concept and use case of animation interpolation and extrapolation, and in the end, we will be able to create really complex animations by combining several animations together. Throughout the chapter, instead of working on our WhatsApp clone, I will show in isolation various animations that you later can apply to your app UI and achieve animations like in Instagram, Facebook, or YouTube apps.

So, let's summarize what we will learn in this chapter:

- Animating React Native components when they are created or destroyed
- Animating components in response to user interactions
- Animating in response to scrolling
- Running animations in a sequence or in parallel to create complex animations

Understanding animations

There is no need to explain what is an animation. As a person living in the modern world, you've seen a lot of animations from the most subtle ones to entire screens animating in exaggerated and complex animations.

The key to make your app stand out is to add animation, but the most important one is also not to exaggerate your animations. Small, simple animations are engaging and even interesting. Even if the user does not notice them, they have a psychological effect on the user, making the content stand out and look more modern. Consider WhatsApp. When you send a simple WhatsApp message, there is a sending/sent icon on the right of the screen, which appears with a fade in animation. You may have never thought of this icon as being animated before or had noticed it before I told you, but these small simple animations make your app look more slick and modern. In contrast, think of the following use case. When a WhatsApp message arrives, it can be animated from the left, bouncing in and spinning until it gets into view. This effect would be really disturbing, right? It will make the user focus more on animation and less on content.

The key to a killer app is to make your animations very subtle and seemingly unnoticeable, and, yes, it can be a nonrewarding experience. You can spend lots of time on one animation, and it won't be noticeable to users at all. What will make the difference is that putting these animations in their right place will make the user experience much more pleasant.

Thanks to React Native, animations can be prototyped pretty fast, and due to the feature of hot reloading, it lets you see how the animation will look like before transferring your application to the phone.

Conventions

So, you may have purchased a printed copy or a Kindle version of this book. The first question I asked myself when I started to write this chapter was how I will show animations in print. I came up with some conventions to show you on paper how an animation should behave:

- **Animation states**: This will be presented as series of pictures, usually, the initial view and the end view. If an animation has several other states in between, you will see these, too.
- **Animation timing graph**: Since animations are not always linear in timing (as a matter of fact they usually are not linear) next to animation states, you will see a graph of the animation timing.

- **Description**: Of course, you will get the full description of how an animation works. In some recurring examples, I will omit an animation state or timing graph in favor of a more in-depth explanation of how the animation will behave.

I advise you that, in any case, even if the animation description is sufficient, try and code animation along with the book. There is nothing that can substitute seeing your animation working visually.

How animation works

Animations, generally speaking, are style values transformed over time. For example, the fade in animation has an opacity value that moves from 0 to 1. This transition can be done in various ways. This transition is specified by **easing functions**, which specify the rate of parameter change over time. There are various easing functions that you are probably familiar with from the web; you can check them at `http://easings.net/`.

Although not all of these functions are supported in React Native, the subset we get and the ability of interpolation and combining animations are enough to create really complex animations, but remember one thing; if you are doing a really complex animation, always ask yourself if you really want your animation to be so complex. Maybe, you will end up distracting users from the actual content.

React Native provides us with two systems to create stunning animations:

- LayoutAnimation
- Animated

The difference between the two is that LayoutAnimation is built for more simpler animations that happen on component render. It's much less customizable and not well documented; however, we will cover it, including the hiding in the source code options we can use to create really complex and powerful animations. LayoutAnimation is an animation that runs directly on the native UI thread, meaning that we get real native animations.

Animated, though, is much more customizable, but it runs on `requestAnimationFrame`, and even though it's very performant and optimized, if you want to trigger native animations, you need to specify it with the `useNativeDriver` option. We will cover that when we talk about the Animated API. However, before we do so, let's understand the much simpler system of LayoutAnimation.

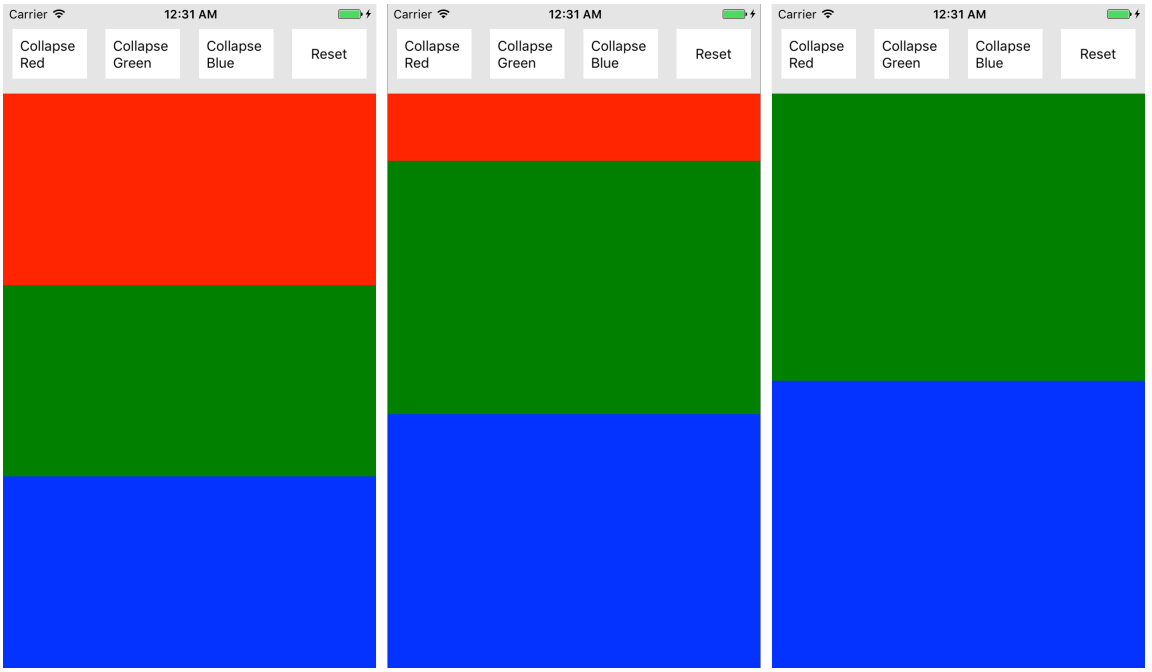
Using the LayoutAnimation API for simple animations

LayoutAnimation is a React Native API used to configure automatic animations of a view to its new position when the next layout happens. Basically, it gives you a system to configure animations of all component child views in one place, instead of configuring these animations separately for every component. LayoutAnimation handles all the interpolation for you. You just need to configure it in your parent component.

In Android, in order to make this work, we should do the following in the root of our React Native application (index.android.js):

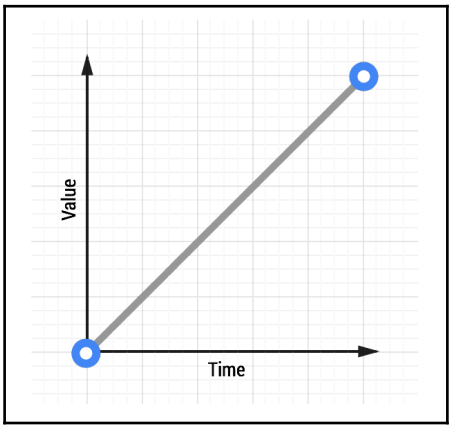
```
import { UIManager } from 'react-native'
UIManager.setLayoutAnimationEnabledExperimental &&
UIManager.setLayoutAnimationEnabledExperimental(true);
```

Let's create the following animation:



Let's overview what are we trying to make. We have a screen with four buttons at the top and three colored areas from the top: Red, Green, Blue and Reset.

When clicking on the **Collapse Red** button, first area (Red) is collapsed toward the top of the screen. When clicking on **Collapse Green**, the Red area reappears, and the Green (second) area collapses. When clicking on the Blue button, the Blue (third) area collapses, and all other areas, which are not seen, appear. In the image, you see the animation frames of the first behavior. Animation is linear and can be described with the following graph:



Now, let's write some code. I will skip app creation since by this point you should be familiar with it; if not, check out previous chapters for reference.

After creating an app and organizing it, I will create my simple app component. For buttons, I will use `TouchableOpacity` since instead of default buttons styling I want to be able to create buttons with a different style.

Our render function will look as follows:

```
<View style={styles.container}>
  <View style={styles.topBar}>
    <TouchableOpacity style={styles.button} onPress={()=>
    this.onPress(1)}>
      <Text style={styles.buttonText}>Collapse Red</Text>
    </TouchableOpacity>
    <TouchableOpacity style={styles.button} onPress={()=>
     this.onPress(2)}>
      <Text style={styles.buttonText}>Collapse Green</Text>
    </TouchableOpacity>
    <TouchableOpacity style={styles.button} onPress={()=>
     this.onPress(3)}>
      <Text style={styles.buttonText}>Collapse Blue</Text>
    </TouchableOpacity>
    <TouchableOpacity style={styles.button} onPress={()=>
     this.onPress(0)}>
```

```
        <Text style={styles.buttonText}>Reset</Text>
      </TouchableOpacity>
    </View>
    <View style={[styles.red, styles.area]}/>
    <View style={[styles.green,styles.area]}/>
    <View style={[styles.blue, styles.area]}/>
  </View>
```

As you can see from the preceding code, we have several styles for our buttons. Let's take a look at `styles.area` in detail.

It's simply:

```
area: {
  flex: 1
},
```

This means that areas are distributed evenly across a vertical container.

Now, let's pause for a second and think what collapsing our area means. It means that if we set our specific area to `flex: 0`, then the rest of the areas will be evenly spaced because every area is `flex: 1`.

Now, all we need is to do a couple of things:

1. Create a collapsed style. We will add the following to our styles:

```
collapsed: {
  flex: 0
}
```

2. Set some `activeIndex` state and update it accordingly in the `onPress` function:

```
state = {
  activeIndex: 0
}
onPress(activeIndex){
  this.setState({ activeIndex })
}
```

3. Get collapsed style based on `activeIndex` in our render function:

```
const {
  redStyle,
  greenStyle,
  blueStyle
} = this.getStyleByCollapsingIndex(this.state.activeIndex)
```

4. Return a style according to `activeIndex`:

```
getStyleByCollapsingIndex(index){
  return {
    redStyle: index === 1 && styles.collapsed,
    greenStyle: index === 2 && styles.collapsed,
    blueStyle: index === 3 && styles.collapsed
  }
}
```

5. Add a style to our area components:

```
<View style={[styles.red, styles.area, redStyle]}/>
<View style={[styles.green, styles.area, greenStyle]}/>
<View style={[styles.blue, styles.area, blueStyle]}/>
```

So now, it's working. The next thing we need to do is to add animations. This is much simpler to do than the creation of the whole component:

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

Inside the `onPress` function, before setting the state, add the following:

```
LayoutAnimation.linear();
```

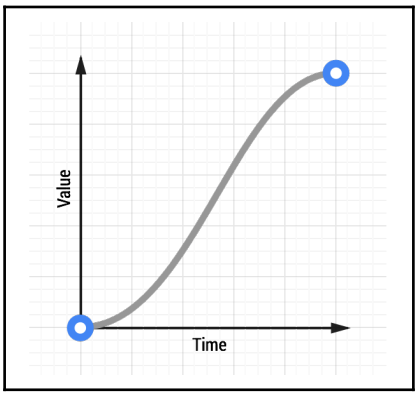
Now, you will get a linear animation for collapsing areas.

Basic syntax

Okay, so we saw that adding `LayoutAnimation` for predefined animations is pretty straightforward. We also can add fadeIn animation simply using `LayoutAnimation.linear()` inside the `componentDidMount()` lifecycle method. Automatically, without even setting any style, we will get fade in animation. LayoutAnimation will use the default animate opacity from 0 to 1.

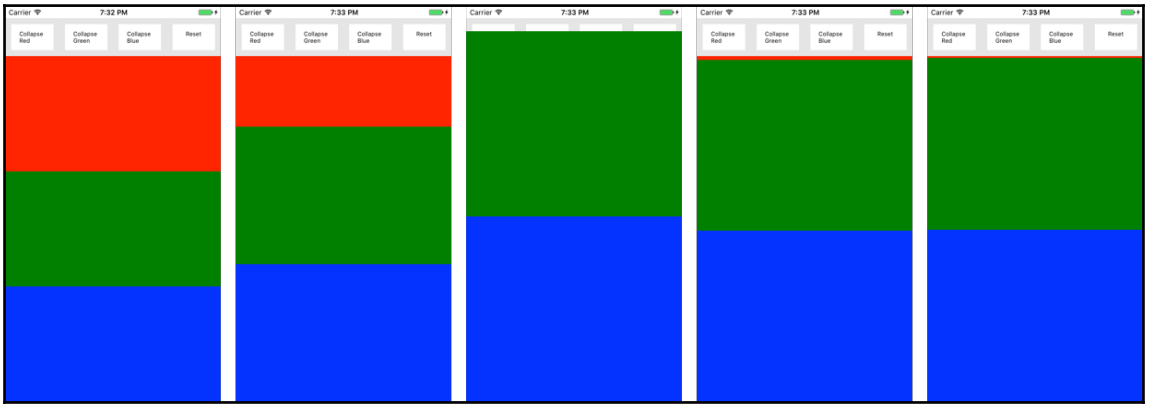
However, as I said before, we are not used to linear animations since there is no such thing in the physical world. We are more familiar with easing or spring animations. `LayoutAnimation` API supplies us with these functions. Instead of using `LayoutAnimation.linear()`, we can use `LayoutAnimation.easeInEaseOut()` or `LayoutAnimation.spring()`.

Let's take a look at what we get for `easeInEaseOut`. The resulting animation will be the same, however, the timing graph will be different:

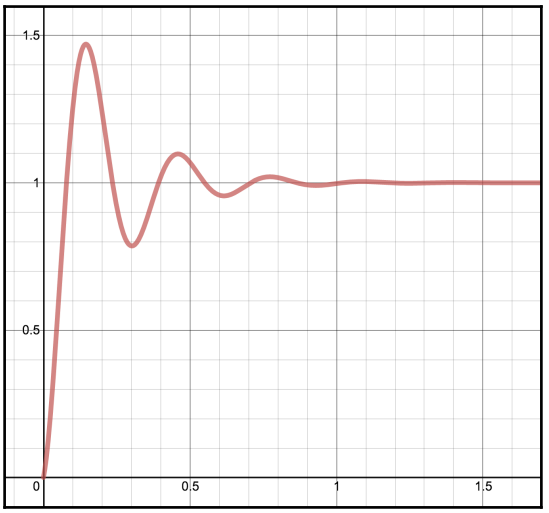


As you can see in the preceding graph, it accelerates slowly, keep a constant speed, then decelerates slowly.

In the case of spring, we have a bouncing effect. You are welcome to try it. It will look similar to this:



The timing of our animation can be described as on the following graph:



These three functions (`linear`, `spring`, and `easeInEaseOut`) are basically a shorthand for one of the most important functions, `configureNext`. This function takes two arguments: a config object and an optional `onAnimationDidEnd` callback, which is the function that will be executed when the animation is finished.

Let's write down all three previous examples using `configureNext` instead of animation functions:

```
LayoutAnimation.configureNext(LayoutAnimation.Presets.linear)
LayoutAnimation.configureNext(LayoutAnimation.Presets.easeInEaseOut)
LayoutAnimation.configureNext(LayoutAnimation.Presets.spring)
```

There are three presets already configured in `LayoutAnimation.Presets`, but LayoutAnimation also ships with the helper function to create such presets.

The preset system under the hood

For simple animations, we can pass the existing presets, but sometimes we would want to create our own animation configuration, starting from opacity to spring damping or other properties. It can be configured using the `LayoutAnimation.create` function. So, the pattern will be as follows:

```
const customAnimationConfig = LayoutAnimation.create(config)
LayoutAnimation.configureNext(customAnimationConfig)
```

Now, let's see what exactly we can pass inside `config`. For that, let's take a look at how `linear` and `easeInEaseOut` presets are configured in our source code:

- `easeInEaseOut` configuration is done in the following way (that's how it's done in the source code):

  - ```
    create(
      300, Types.easeInEaseOut, Properties.opacity
    )
    ```

- `linear` configuration is done in the following way (that's how it's done in the source code)

  - ```
    create(
      500, Types.linear, Properties.opacity
    )
    ```

`Types` and `Properties` are available via:

```
const { Types, Properties } = LayoutAnimation;
```

Let's walk through the arguments. The first one is the duration of the animation; the second is a `Types` enum--it can be one of the following:

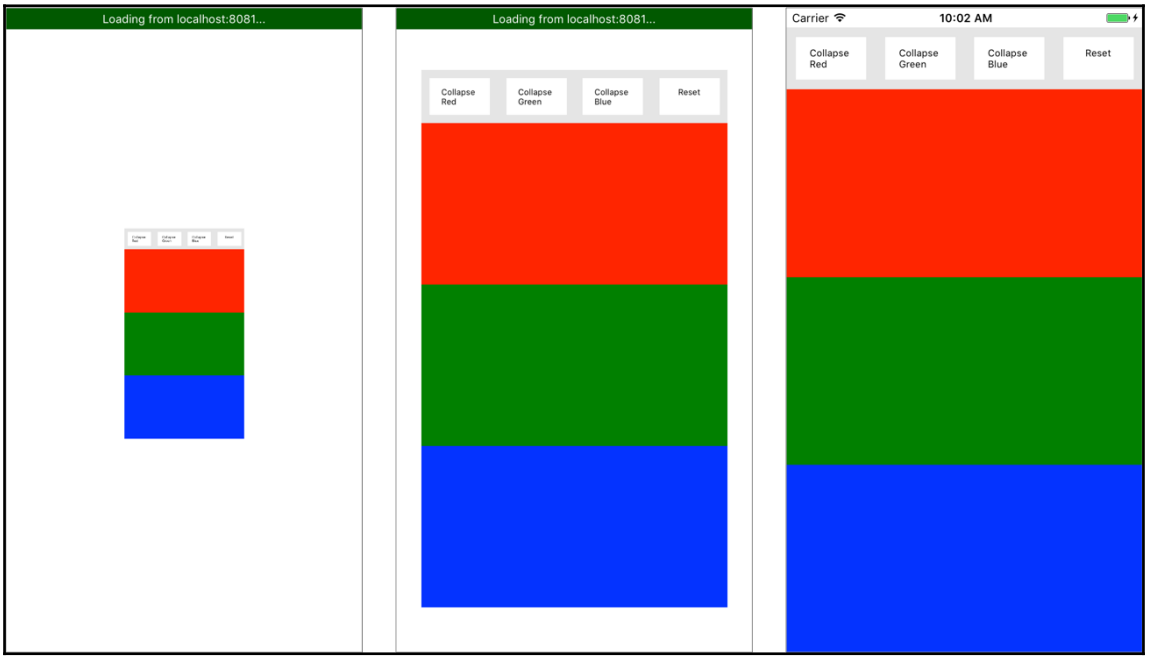
- `spring`
- `linear`
- `easeInEaseOut`
- `easeIn`
- `easeOut`
- `keyboard`

Although all of the above are self-explanatory, the `keyboard` animation type remains a mystery. It's not one of the known easing types. Since LayoutAnimation activates native animations, the keyboard animation activates same type of easing used for the keyboard. So, if you want to mimic keyboard animation on iOS or Android, use this type.

The third argument is `creationProperty`. In this case, we have `opacity`. The other option is `scaleXY`. This option is relevant only at component creation, and it means, as the name suggest, scale on the X and *Y* axes. For example, consider that we will use this:

```
componentDidMount(){
  const { configureNext, create, Properties, Types } = LayoutAnimation;
  configureNext(
    create(500, Types.linear, Properties.scaleXY)
  )
}
```

Then, we will get a linear animation with the following result on the initial mount of our component instead of fadeIn:

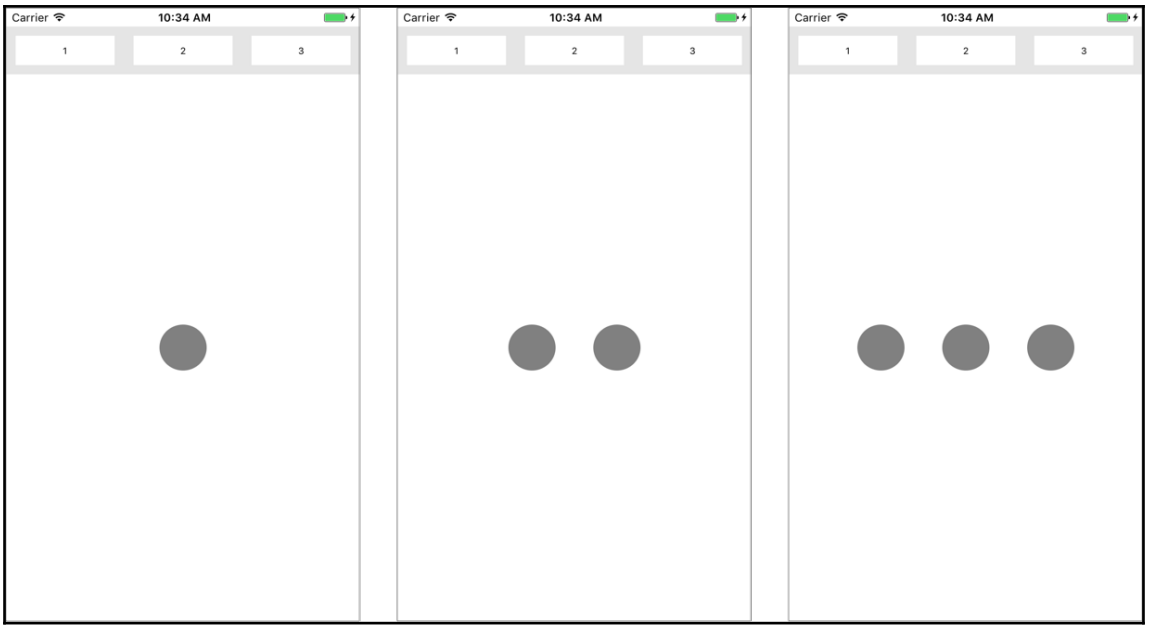


While custom animation can be also created by `create` helper, it also can be created using even more complex configuration objects. Let's take a look at the `spring` preset--how it's configured in the code:

```
spring: {
  duration: 700,
  create: {
    type: Types.linear,
    property: Properties.opacity,
  },
  update: {
    type: Types.spring,
    springDamping: 0.4,
  },
  delete: {
    type: Types.linear,
    property: Properties.opacity,
  },
}
```

`duration` is the duration of the animation.
Then, we have the following three configuration objects:

- **create**: This is the configuration of animation that happens on initial render of a component. This is relevant for the existing component if you use it in `componentDidMount()` or for components that are added as a result of the next `setState` call.
- **update**: This configuration corresponds for component updates, meaning that it will happen for every component re-render.
- **delete**: This is the animation configuration for the destruction of a component. Let's take a look at an example to illustrate how this can be used. Let's alter our code a bit. Instead of collapsing areas, we will add circles. First, let's do it without animation like this:



When clicking on a button shown in the preceding screenshot, we will have a corresponding number of circles.

First, let's change our `state`:

```
state = {
  items: 1
}
```

Then, let's change our `onPress` function to change this state:

```
onPress(items){
  this.setState({ items })
}
```

Our `render` function will look like this:

```
<View style={styles.container}>
  <ButtonsBar onPress={this.onPress.bind(this)}/>
  <View style={styles.area}>
    { this.items }
  </View>
</View>
```

As you can see, I moved the top buttons bar to separate a component without any changes.

Instead of the Red/Green/Blue area, I added a `View` tag with the item. area styles changed to the following:

```
area: {
  flex: 1,
  justifyContent: 'center',
  flexDirection: 'row',
  alignItems: 'center'
},
```

Let's look at the `item` getter:

```
get items(){
return Array(this.state.items)
  .fill(1)
  .map((item, index) =>
    <View style={styles.item} key={index}/>
  );
}
```

`item` style will be the following:

```
item: {
  width: 50,
  height: 50,
  backgroundColor: 'gray',
  borderRadius: 25,
  margin: 20,
  justifyContent: 'center',
  alignItems: 'center
}
```

As you can see, I created an array with the same number of items as written on the buttons and map on this array, returning a circle.

Now, let's add animation:

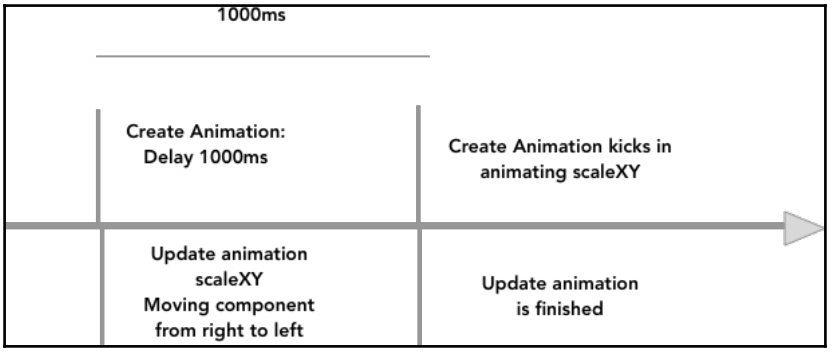
```
const CustomLayoutAnimation = {
  duration: 100,
  create: {
    delay: 100,
    type: LayoutAnimation.Types.spring,
    property: LayoutAnimation.Properties.scaleXY,
    springDamping: 0.2,
    initialVelocity: 1
  },
  update: {
    type: LayoutAnimation.Types.easeInEaseOut,
```

```
      property: LayoutAnimation.Properties.scaleXY
    },
    delete: {
      type: LayoutAnimation.Types.easeOut,
      property: LayoutAnimation.Properties.opacity
    },
  };
```

So, what is happening here? First of all, we can see that duration of an animation is 1,000 ms. Then, we see that when a component is created, we wait 1,000 ms by specifying the delay property. Then, with the spring type animation, we change the `scaleXY` of the circle. We also add `initialVelocity` and `springDamping` properties to configure our spring behavior.

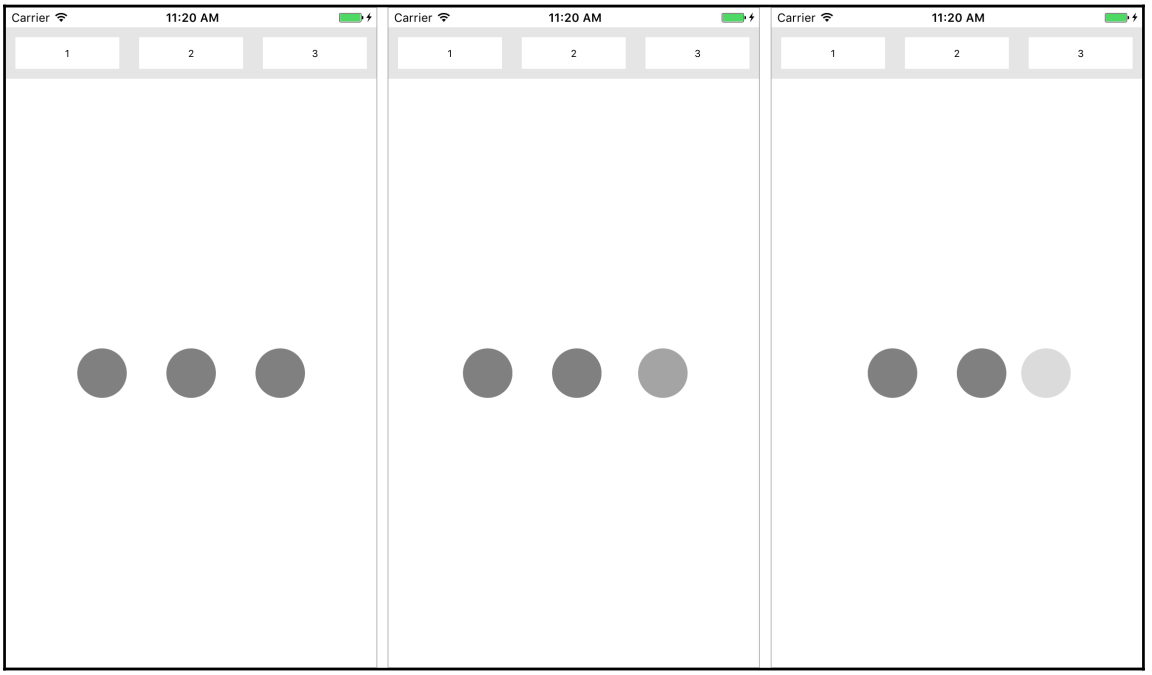
When our component is updated, we will get a scale XY transition, which corresponds for moving our component from right to left. After it's moved, `create` animation is triggered, and we have our new circle scaled up.

Let's look at the following schematic--how animations kick in:



Now, let's look at the actual animation frames:

Now, let's take a look at the delete animation:



When we click on a button with a fewer number of items than what we have on the screen, circles that are not required are destroyed by fading their opacity out using the `easeOut` animation.

In the previous example, we have three circles on the screen, and when we click on the **2** button, our component starts fading out while still using update animation and places the rest of the circles at the center of the screen.

LayoutAnimation is an awesome tool that can be used for lots of animations, even complex ones. It's much more performant than the rest of the animation options because it works directly with native animations. It's also pretty well configurable, however, poorly documented. When you want to implement animation, think whether it could be done with LayoutAnimation, and only if it can't be done or would require lots of work, then switch to Animated. Animated is extremely configurable and well documented. You can create really stunning animations with it.

The key difference, though, is that LayoutAnimation will globally configure, create, and update animations that will be used for all views in the next render/layout cycle. This means that it's use case is limited. Animated, on the other hand, can be used for nearly any animation you want.

Using Animated API for complex animations

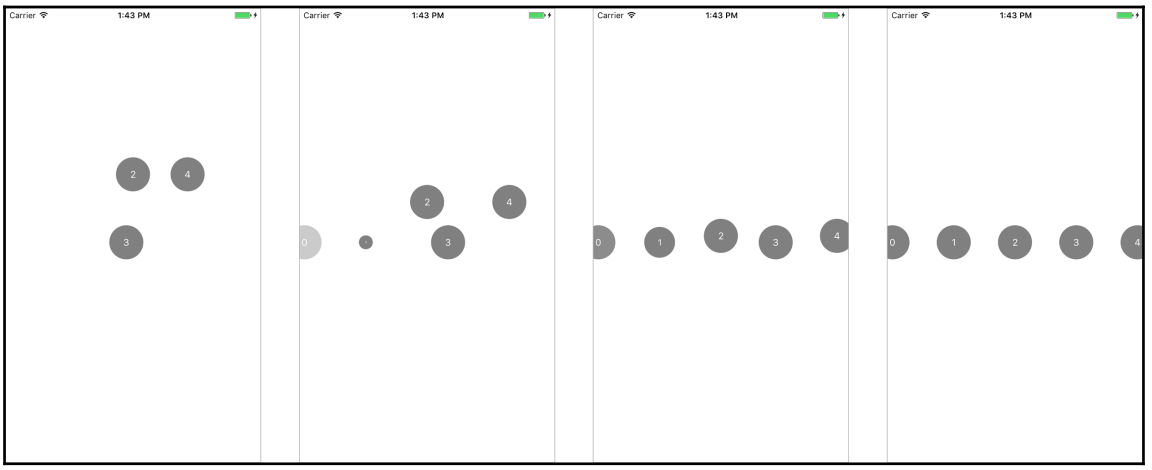
Animated API is an API that React Native supplies to create a wide range of animations and interaction patterns in a performant way. It focuses on relationships between inputs and outputs with configurable transforms in between and executing start and stop methods to control animation execution.

Animated has built-in Animated components and a function to create an Animated component. The exposed components are View, Text, Image, and ScrollView. Also, you can make almost any component Animated using the `Animated.createAnimatedComponent()` function. In order to start using animation, we will need to use `animatedValues` to declare our animation and then execute the Animated function to start/stop execution.

Animated values

In order to create a fadeIn effect, we will need to animate opacity from 0 to 1. We do so using both Animated values for setting animation and Animated functions for triggering it.

Let's take a look at the following example, which will work only with Animated and won't work with `LayoutAnimation`:



We are going to animate five circles using the Animated API. The first two circles are not initially visible. When the animation starts, the first circle fades in on the left while at the same time the second circle scales up. At the same time, the third cycle is moving down and the fourth circle moves to the right horizontally. The fifth cycle is moving down and right.

As you can see, this animation is pretty complex. Let's split it into pieces. First of all, we can see right away that there are two simple animations present. The first one is fadeIn of the first circle (0) and the second one is scaling up of the second circle (1) from the left. Then, we have the third circle (2) that is changing its position from -100 on *y* axis toward its position in the middle of the screen. At the same time, the fourth circle (3) is moving on the X axis from -100 to its position as seen in the diagram. Finally, the fifth circle (4) is moving from -100, 100 to its position, and is to be distributed evenly with the rest of the items.

Let's walk through the code and see how this animation can be achieved. Later, we will make this animation even more complex with the use of different animation functions.

First of all, we've differentiated three types of animations: fading, scaling, and sliding. That means we will have three different `Animated.Values`. Later, we will be able to use interpolation to use only one `Animated.Value`.

Interpolation will help us by transforming one Animated value to three different values. An ability to interpolate from one value to an other value is a very strong feature that gives us the ability to create really complex animations. We will dive deep into how it's done later on.

Let's assign them and walk through the `Animated.Value` API. So, here is our state:

```
state = {
  items: 5,
  fadeAnim: new Animated.Value(0),
  slideAnim: new Animated.Value(-100),
  scaleAnim: new Animated.Value(0)
}
```

By defining `fadeAnim`, `slideAnim`, and `scaleAnim` as Animated values, we make state changes much more optimized than by calling `setState` and re-rendering, which will lead to lagging. Animated will update these values using its internal mechanism.

We have two types of Animated values:

- `Animated.Value`
- `Animated.ValueXY`

One is one dimensional, and the second is two dimensional. ValueXY is useful for driving 2D animations, such as pan gestures. We will use ValueXY when we will get into the React Native PanResponder API to deal with gestures in a later chapter.

The important methods available on `Animated.Value` you should know are as follows:

- `setValue`: You can set values directly on `Animated.Value` using this method.
- `addListener`: Due to the asynchronous nature of animations, if you want to know what the Animated value is, you need to observe updates through a listener.
- `removeListener / removeAllListeners`: If you've added listeners using the `addListener` function, it's important to remove them when that component is unmounted.
- `stopAnimation`: This stops the animation execution. You can also pass an optional callback function that will be executed when the animation is stopped.
- `interpolate`: We will cover this method later in this chapter.

There are several other methods available for `Animated.Value` that are mentioned in the following official documentation:

`http://facebook.github.io/react-native/docs/animated.html#animatedvalue`

For `Animated.ValueXY`, there are also `getLayout` and `getTranslateTransform` methods.

`getLayout` converts X and Y into { left, top }, and `getTranslateTransform` converts X and Y into a useable translate transform. We will use it in later chapters when we deal with gestures.

Now, let's get back to our example. It's not enough only to set our Animated value. We need to style our views accordingly. We will add a switch case in the items getter to style the circles based on their index:

```
get items() {
        return Array(this.state.items)
            .fill(1)
            .map((item, index) => {
                switch (index) {
                    case 1:
                    // return Animated.View for relevant circle with
                    index 1
                    case 2:
                    // return Animated.View for relevant circle with
                    index 2
                    case 3:
                    // return Animated.View for relevant circle with
                    index 3
                    case 4:
                    // return Animated.View for relevant circle with
                    index 4
                    default:
                    // return Animated.View for all other circles
                }
            });
    }
```

Now, let's first set up our fading circle (number 0):

```
default:
  return (
    <Animated.View
      style={[styles.item, {
      opacity: this.state.fadeAnim
    }]} key={index}>
      <Text style={styles.itemText}>{ index }</Text>
    </Animated.View>
  );
```

We will use `Animated.View`, so Animated will be able to work with it properly and set opacity to the `fadeAnim` state key. It means that at the beginning its value is 0 as we've set it in our component state. Now, let's configure our scale-up component for the second circle.

Scale-up for the second circle by returning the following for `index === 1`.

```
case 1:
  return (
    <Animated.View
      style={[styles.item, {
      transform: [
        { scale: this.state.scaleAnim }
      ]}
    ]} key={index}>
      <Text style={styles.itemText}>{ index }</Text>
    </Animated.View>
);
```

It's the same pattern, but instead of `opacity`, we will reference the `scaleAnim` state key.

Moving down for the third circle will have:

```
transform: [
  { translateY: this.state.slideAnim }
]}
```

Moving right for the fourth circle will have:

```
transform: [
  { translateX: this.state.slideAnim }
]}
```

The last one will have two values Animated, both X and Y:

```
transform: [
  {
    translateX: this.state.slideAnim
  },
  {
    translateY: this.state.slideAnim
  }
]}
```

Calculations

`Animated.Value` is not a primitive, but a value wrapped by the Animated API. Sometimes, we will want to make simple calculations on these values. We can do so using Animated helper methods:

- `Animated.add`

- `Animated.divide`
- `Animated.modulo`
- `Animated.multiply`

Let's take a look at an example of their use:

```
const a = new Animated.Value(1);
const b = new Animated.Value(2);
Animated.add(a,b) = Animated.Value(3)
Animated.divide(a,b) = Animated.Value(0.5)
Animated.modulo(b,2) = Animated.Value(0)
Animated.multiply(a,b) = Animated.Value(2)
```

Animated functions

Okay, so we've defined our animations in state and created `Animated.View` with corresponding styles for each circle. Now, it's time to start animation. In order to do so, we will use Animated functions.

Let's start our animations on `componentDidMount()`:

```
componentDidMount(){
  const { timing } = Animated;
  const { scaleAnim, fadeAnim, slideAnim } = this.state;
  timing(
    fadeAnim, { toValue: 1 }
  ).start();
  timing(
    slideAnim, { toValue: 0 }
  ).start();
  timing(
    scaleAnim, { toValue: 1 }
  ).start();
}
```

Here, we will use the `Animated.timing` function. This function, as other Animated functions--which we will cover in a bit--is used to configure and compose our animations if needed. `Animated.timing` can be configured with a duration parameter and easing parameter. Default easing is `easeInOut`, but can be configured using Easing functions exported with React Native.

Easing has four predefined animations:

- `back`: This provides a simple animation to the object that goes back before moving forward
- `Bounce`: This provides a bouncing animation
- `ease`: This provides a simple inertial animation
- `elastic`: This provides a simple spring interaction

It also has lots of helper methods to create any curve you like. You can read more about Easing here:

`http://facebook.github.io/react-native/docs/easing.html`

Let's change our fade animation to use different easing:

```
timing(
  fadeAnim, {
    duration: 3000,
    delay: 400,
    easing: Easing.bounce,
    toValue: 1
  }
).start();
```

We changed the duration to three seconds, set the delay to 400 ms, and changed our Easing to bounce. Of course, before doing so, we imported easing from React Native.

We have one more important parameter that we can pass to any Animated function:

`useNativeDriver: true`

When the parameter is specified, we will send all animation configuration to native before starting animation. This means that animation will be done on the UI thread without passing through a bridge on every frame. It's pretty useful, especially for Scroll events. The problem with native driver is that not everything is currently supported. However, more and more is being added, so ensure that you check exactly what works with native driver and what doesn't by referring to official docs at `http://facebook.github.io/react-native/docs/animations.html#using-the-native-driver`.

Interpolation

I mentioned previously that both `Animated.Value` and `Animated.ValueXY` have the interpolate method. This method is used to map between `Animated.Value` ranges to different output ranges.

Let's take a look at our example and make it work with the power of extrapolation using only one `Animated.Value`--let's call it `animatedValue`:

```
state = {
  items: 5,
  animatedValue: new Animated.Value(0)
}
```

Now, our `componentDidMount` will look as simple as:

```
componentDidMount(){
  const { timing } = Animated;
  timing(
    this.state.animatedValue, { toValue: 1 }
  ).start();
}
```

Now, let's take a look at where we need to make changes. We have `this.state.fadeAnim` that is changed to `this.state.animatedValue`. We also have `scaleAnim`, which we can also change to `this.state.animatedValue`.

The range is the same, that is, 0 to 1, but what should we do with the `slideAnim` state? The range for the `slideAnim` state is -100 to 0. For that particular reason, we have interpolation. With the power of interpolation, we can define that when we have 0, it's mapped to -100, and when we have 1, it's mapped to 0. Animated will animate values between two new values instead of 0 to 1. This is done in the following manner:

```
const slideAnim = this.state.animatedValue.interpolate({
  inputRange: [0, 1],
  outputRange: [-100, 0]
})
```

Later on, we will just use `slideAnim` instead of `this.state.slideAnim` inside the view. You will get the same results as in the preceding example.

Interpolation is very useful when dealing with colors, for example, if you want to animate between colors, but in addition to that also animate the opacity, you can animate from 0 to 1 in your Animated value and use interpolation to define `outputRange` for use with colors.

Extrapolation

Sometimes, you will see that interpolation is not working as expected. This can happen because when interpolating values, they can extend beyond the input range. When setting interpolation, by default, it will extrapolate the curve beyond the given range, but you can also clamp the output value by setting the extrapolate option.

So, now, probably you are asking yourself what does that mean.

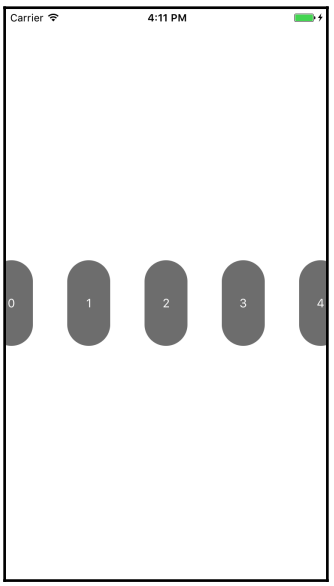
Consider this code:

```
const slideAnim = this.state.animatedValue.interpolate({
 inputRange: [0, .5],
 outputRange: [0, 50]
})
```

Note that my `inputRange` is 0 to 0.5, whereas `animatedValue` goes from 0 to 1. Sometimes, I want to interpolate only parts of `inputRange`. When rendering the component later on, I will do the following:

```
style={[styles.item, {
 height: slideAnim
}]}
```

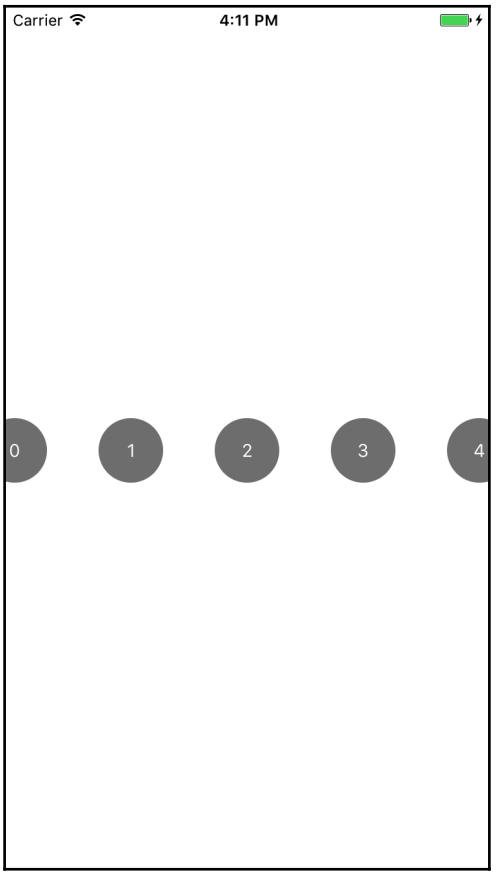
This is what I will get:



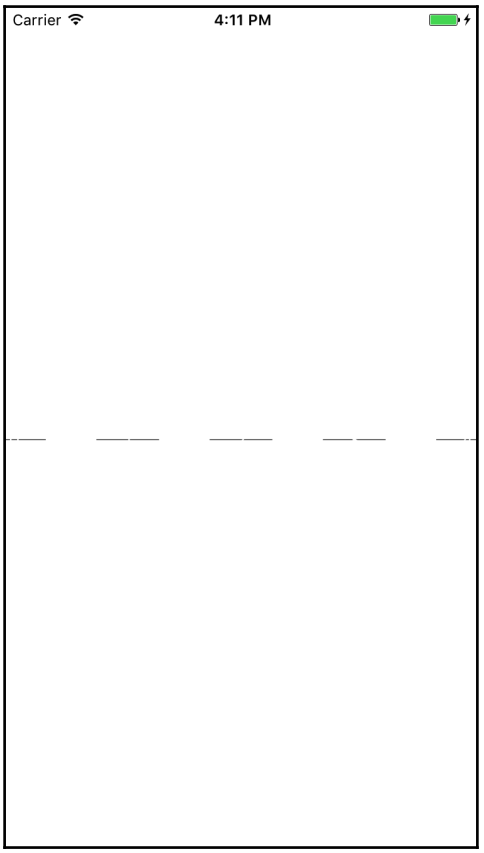
This happens because when my `animatedValue` goes beyond the 0.5 input range, the interpolate method extrapolates the curve up to its final value. This is the default option, but it can be configured in the following way:

```
const slideAnim = this.state.animatedValue.interpolate({
  inputRange: [0, .5],
  outputRange: [0, 50],
  extrapolate: 'clamp'
})
```

The result will be as expected:



The Extrapolate option can also get identity as a value. That means that when a value hits `inputRange`, it's assigned a value by passing interpolation. So in this case we will get an animation from 0 to 50, and then circles will get a value of 1 that looks like this:

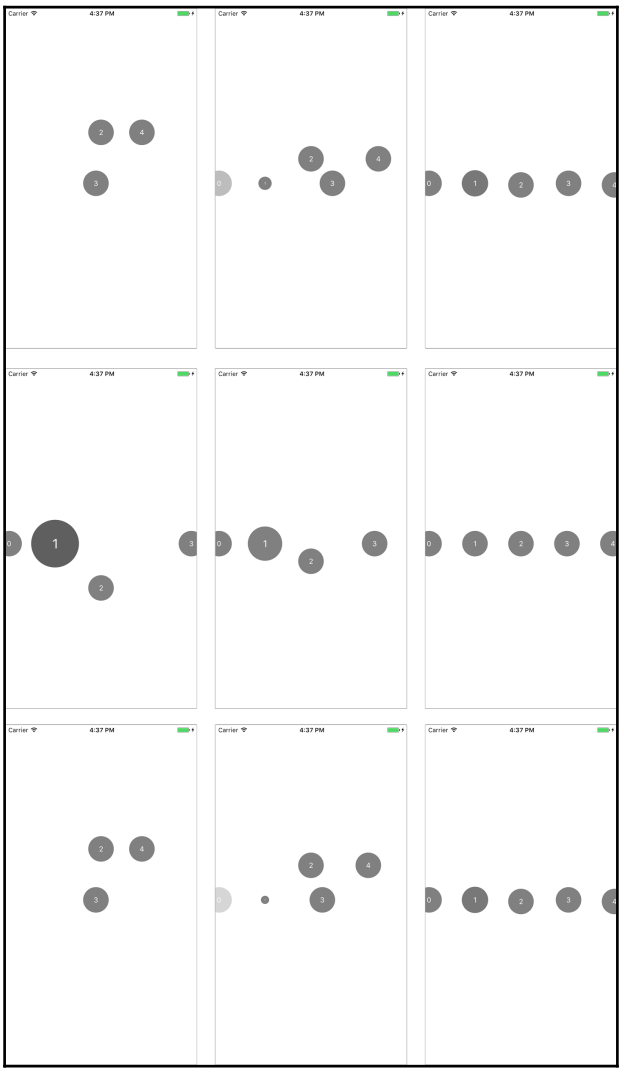


Interpolation is, by itself, an interesting option, and lots of animations can be achieved using interpolation. There is additional information regarding interpolation, which you can find in its official documentation. In the future, interpolation will have more configuration options, so always check the following documentation to stay up to date:

`http://facebook.github.io/react-native/docs/animations.html#interpolation`.

Combining several animations to create complex sequences

We saw that animations can be complex using LayoutAnimation or Animated, but they can become even more complex with the power of creating sequences of animations and running them in parallel. Let's take a look at the following animation and see exactly how it is implemented:



We want the following animation: after delay of 200ms, we will change `animatedValue` to 1 by using the `timing` function

```
timing(
 this.state.animatedValue, { toValue: 1 }
)
```

then wait for 200ms and will change it to `2` by using the following timing function:

```
timing(
  this.state.animatedValue, { toValue: 2 }
)
```

Our opacity is animated to `2`, which doesn't matter, but the scale of the second circle and positions (due to interpolation) of other circles change. After animating to `2`, we want to execute several animations in parallel: Change the color of the second circle from gray to black and back again. After 600ms from the beginning of this animation we want `animatedValue` to animate from `1` to `0` and back.

The main question is how to implement such a complex animation. We will do that by understanding several Animated functions that can help us in this, and then we will implement the complex animation I've shown in screenshots and just explained.

First of all, we have `delay`. That can be done simply using the `Animated.delay` function that gets as an argument timeout in milliseconds. Then, we have several animations running in sequence. This can be done using the `Animated.sequence` function. As an argument, it gets an array of animation functions that you would want to run one after another.

Then, we have animations running in parallel. This can be done using:

`Animated.parallel`, which accepts an array of animation functions and runs them in parallel, that is, differently from the sequence function.

We also have `Animated.stagger` function, which is much like a sequence; however, you can configure a delay between each animation. So, it gets timeout in milliseconds as its first argument and an array of animation functions as its second argument.

So, now let's take a look at our code. We've animated color, so we can introduce the new `animatedColor` state key, which will get a value of `0`:

```
animatedColor: new Animated.Value(0)
```

Later, we will animate it from 0 to 100, but we want it to change from gray to black and back; that can be achieved with interpolation:

```
const color = this.state.animatedColor.interpolate({
  inputRange: [0, 50, 100],
  outputRange: ["gray", "black", "gray"]
})
```

Adding this value to our second circle will look as follows:

```
style={[styles.item, {
  transform: [
  { scale: this.state.animatedValue }
  ], backgroundColor: color }
]}
```

Now, we are left with the task of animating everything. Let's do it also in `componentDidMount`.

First, let's get all the functions we need from Animated and all `animatedValues` from state:

```
const { timing, sequence, parallel, delay, stagger } = Animated;
const { animatedValue, animatedColor } = this.state;
```

Now, we will define a sequence and start an animation:

```
sequence([
  delay(200),
  timing(
    animatedValue, { toValue: 1 }
  ),
  delay(200),
  timing(
    animatedValue, { toValue: 2 }
  ),
  parallel([
    timing(animatedColor, { toValue: 100 }),
    stagger(600, [
      timing(
        animatedValue, { toValue: 1 }
      ),
      timing(
        animatedValue, { toValue: 0 }
      ),
      timing(
        animatedValue, { toValue: 1 }
      )
    ])
```

```
    ])
]).start();
```

We can write this sequence as shown in the following schematics:

1. Delay 200ms.
2. Animate `animatedValue` to `1`.
3. Delay 200ms.
4. Animate `animatedValue` to `2`.
5. Run animations in parallel:
    1. Animate `animatedColor` to 100.
    2. Run stagger animations with 600ms delay.
        1. Animate `animatedValue` to `1`.
        2. Animate `animatedValue` to `0`.
        3. Animate `animatedValue` to `1`.

As you can see, we can define really complex animations with multiple levels of nesting. The key to create stunning animations is to separate them to isolated parts, which can be implemented with a single Animated function. Then, combine them to run in parallel or in sequence or both.

Up until now, we saw only the `Animated.timing` function as a function that provides basic easing from value to value. In addition to timing, we also have the `spring` and `decay` functions. Let's take a look at their syntax:

- `Animated.decay`: Animates a value from an initial velocity to zero based on deceleration:

```
Animated.decay(value,{
  velocity: 100,
  deceleration: .5 // default 0.997
})
```

- `Animated.spring`: Spring animation based on `http://origami.design/` and `http://facebook.github.io/rebound/`:

```
Animated.spring(value, {
  friction: 10, //default 7
  tension: 10 // default 40
})
```

Panning and scrolling animations

Sometimes, during scrolling or panning, we need to extract arguments out of the event and set value on `Animated.Value` using the `setValue` function. Since setting `Animated.Value` is faster than changing state and re-rendering, it is a common practice to use Animated with gestures and scrolling. Animated gives us a helper function that does all this for us.

We can use `Animated.event`. As an argument, it gets an array of mapping and extracts values from each argument, then calls `setValue` on mapped outputs. Let's take a look at the following code:

```
onScroll={
  (e) => this._scrollX.setValue(e.nativeEvent.contentOffset.x)
}
```

Here we `setValue` on our `scrollX` `Animated.Value` by taking x from `e.nativeEvent.contentOffset.x`

Instead of writing this way, we can use `Animated.event`:

```
onScroll={Animated.event(
 [{ nativeEvent: {
    contentOffset: {
      x: this._scrollX
    }
  }
 }]
)}
```

Notice that `Animated.event` gets an array of mappers. `this._scrollX` will be set to `e.nativeEvent.contentOffset.x` in this example. We will dive deeper into panning when we get into the `PanResponder` API in the next chapters.

Summary

In this chapter, we started by understanding the basics of animations and easing as well as best practices to design your animations. Then, we looked into the LayoutAnimation API and looked at various simple and complex animations that can be achieved with it. We dived into React Native source code to understand how LayoutAnimation can be configured to achieve stunning results. Then, we continued focusing on the Animated API, which gives us much more flexibility and freedom to create any animation you want. After going through this chapter, you will now be confident enough to try and add some cool animations to your application.

This chapter concludes Part 2--Styling your React Native apps. We will start dealing with more complex apps from the next chapter, `Chapter 7`, *Authenticating your app and fetching data*, onward. In the next chapter, we will cover how to fetch actual data from a remote server and authenticate your app using Facebook or Twitter. I suggest that you start thinking about a killer app that you would like to develop after finishing this book because you are one step closer to learning how to create fully functional applications that can become best sellers. Good luck, and see you in the next chapter.

7
Authenticating Your App and Fetching Data

Welcome to the seventh chapter. In this chapter, you will learn how to deal with real data fetched from the server and how to authenticate your application both using username and password and Facebook authentication. We will start by understanding what Firebase is since it's a great service that lets you build your apps without thinking about investing time and effort in creating your own server. After understanding what Firebase is we will set up our Firebase account and will go through the basic options Firebase gives us. We will learn how to create our own database via Firebase and how to set permissions to it. Then, we will take a look at how fetching data in React Native is done and how it's different from fetching data on the web. We will fetch real data from the database by connecting it to our React Native *whatsappClone* app.

Our next step will be learning authentication. We will create functional Login and Sign Up screens using username and password fields. Then, we will discuss what steps should be taken to enable other social providers, such as Twitter, Google, or GitHub.

So, let's summarize what you will learn in this chapter:

- Getting familiar with Firebase
- Creating your database using the Firebase console
- Fetching data from Firebase
- Posting data to Firebase
- Enabling social authentication using various techniques

We've come a long way from understanding how React Native works to the point where we can create applications with production grade animations and behavior. Every application, which you use or have heard about, does not exist in the void. It connects over the network to a remote server to retrieve and post data. Also, most of the applications use some form of authentication. It can be username and password, or it can be much more popular, social authentication with providers, such as Facebook, Google, Twitter, or any other social provider.

It's not different from the world of web applications. In a similar fashion, we retrieve data from the server and send requests to the server to make our web application work.

Our server can be written into any language we want and, as soon it exposes API endpoints, we can call to get or post data, this same option is also valid for React Native.

We won't dive here in the world of backend development; however, we will consider another option, which is widely adopted as industry standard not only for prototyping things but full applications. It's called Firebase. Firebase is a mobile and web application development platform. Basically, you don't need to worry about backend development and you can use Firebase as a service that will provide for you real-time database, authentication, cloud storage, and much more. All you need to do is to focus on React Native app development, connect it to Firebase, and enjoy seamless integration. It's important to say that Firebase is not the only solution for your app's backend and of course your application can be connected to your own backend or other services.

Getting familiar with Firebase

In this chapter, you will work with Firebase both for authentication and as a real-time database. It's important to understand that React Native does not limit you to use Firebase in your project. All techniques used in this chapter to authenticate with Firebase and data retrieval will also work with any server you want (with slight changes since Firebase synchronizes your application automatically, whereas if you develop your own server, you have to synchronize it yourself).

What is Firebase?

Firebase is a platform that gives you several options. It's used as a service instead of your own server and provides you with all the things your server would provide you and even more.

Let's briefly overview the products that Firebase offers and the ones that are most useful for our React Native apps:

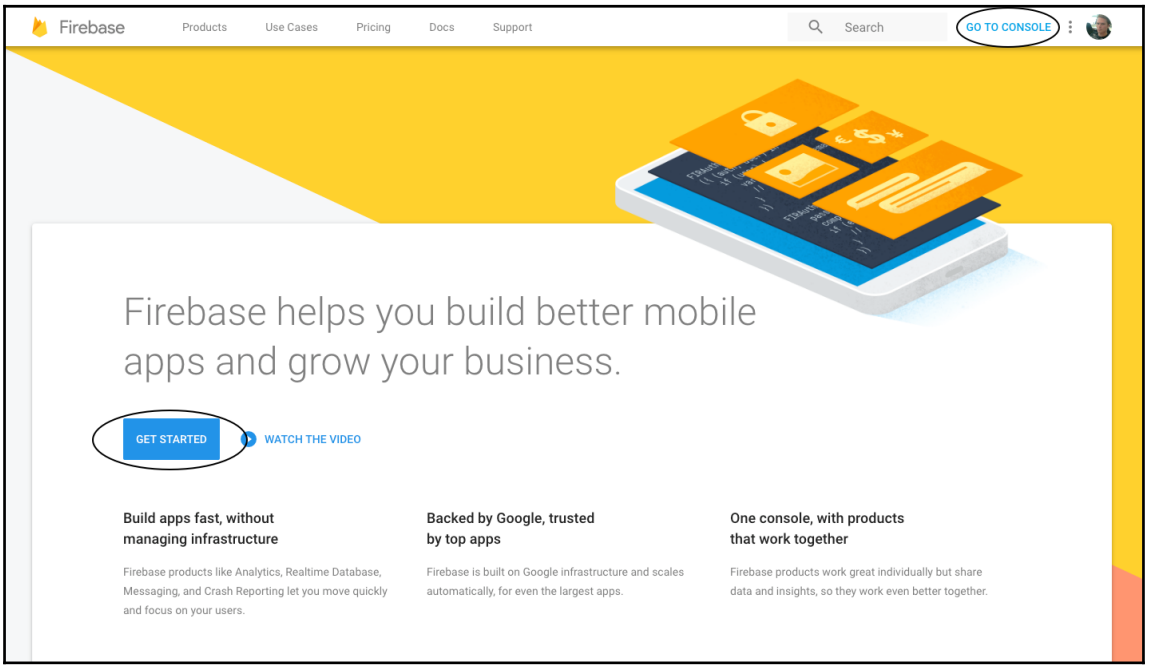
- **Real-time database**: It gives you an ability to store and sync your data in a NoSQL cloud database. Data is synced across all clients in real time whenever you update your database.
- **Authentication**: Firebase provides backend services and SDKs to deal with the authentication of your application. It supports a wide range of authentication options and popular social providers, such as Google, Facebook, and Twitter. It's based on industry authentication standards, OAuth 2.0 and OpenID Connect, so it can be also used with your own server.
- **Cloud storage**: It's usually used for uploading and downloading your files from and to clients. It also enables the client to retry the operation if the connection was terminated, so generally it will save time and bandwidth. Files are stored in a Google Cloud Storage bucket, so they can be accessible through Firebase or via Google Cloud. This gives you the ability to use a Google Cloud platform for image filtering, video transcoding, and much more.

Besides these, Firebase offers dynamic links to specific places in your app, analytics, adwords, admobs, crash reporting, notifications, testing, and so on. You can check all of these options inside the Firebase console in a minute after we set up our Firebase account.

Firebase setup

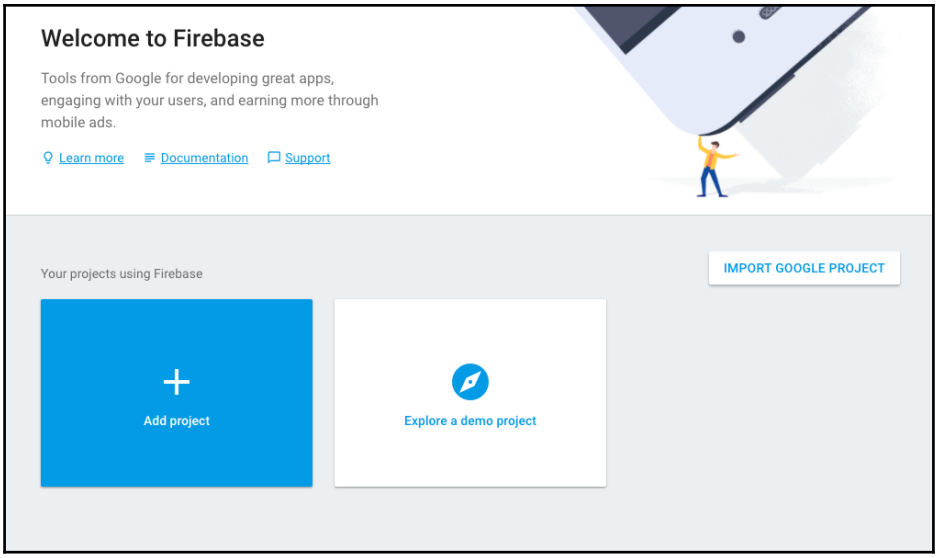
Let's take a look at the steps involved in setting up the Firebase:

1. First, let's create our Firebase account by logging in with your Google account at `https://firebase.google.com`:

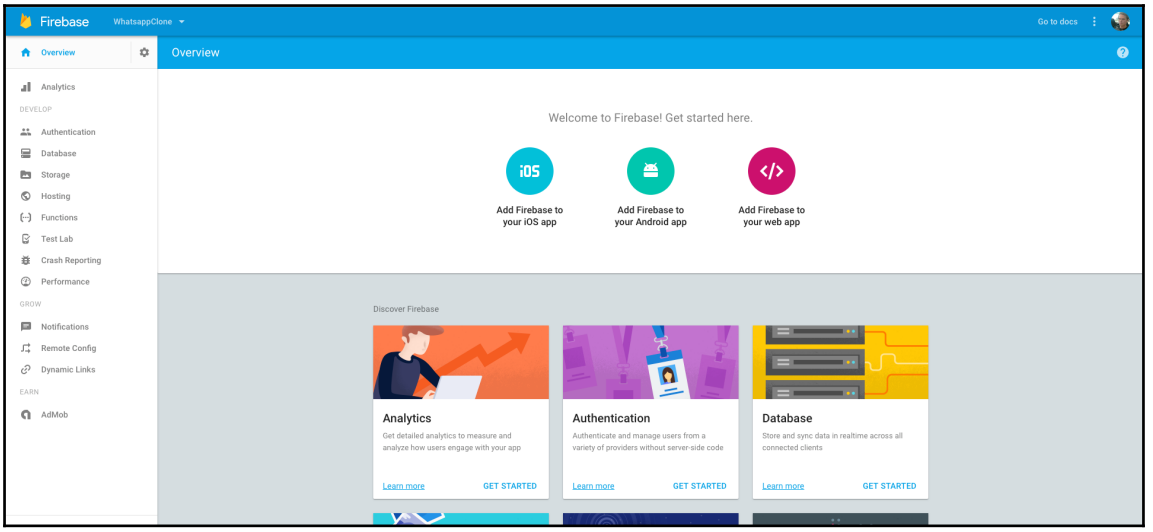


2. After signing in, you can click both **GET STARTED** or **GO TO CONSOLE** links, and you will be navigated to the Firebase console, which is the place you set all Firebase configurations.

3. It should look like this:



4. Now, it's time for you to create a new project. A popup will appear where you should enter your project name and select a country or region. After doing so, the new project will be created, and you will be navigated to the console dashboard:

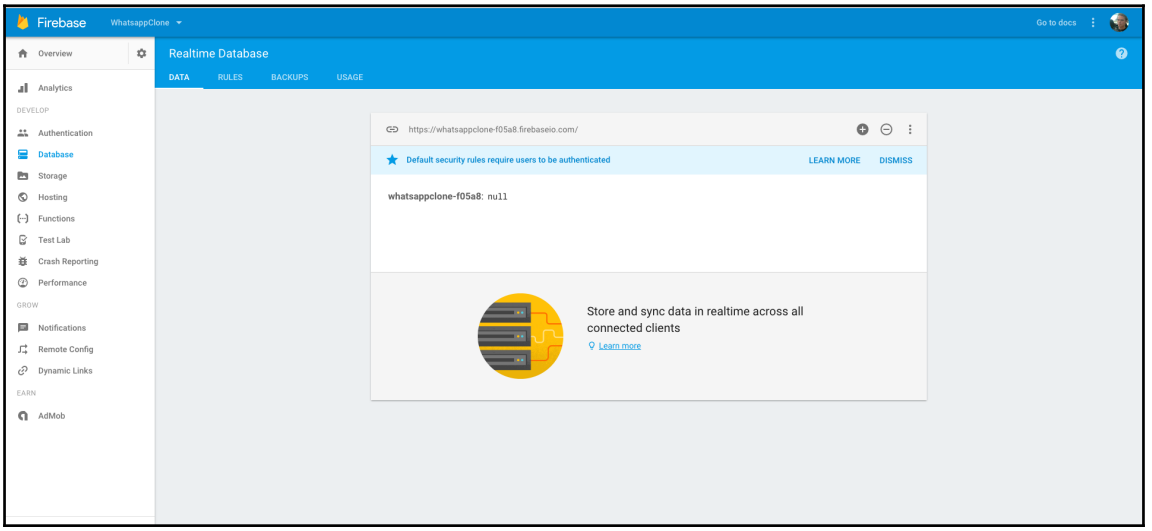


Here, you have lots of options as mentioned earlier. You can check every option and read about it. I won't cover all these options in this book since we are talking about React Native after all and not about Firebase; however, keep in mind that most of these options have npm packages with great documentation, so before trying to configure some specific option yourself, check whether there is an available package.

Creating real-time database

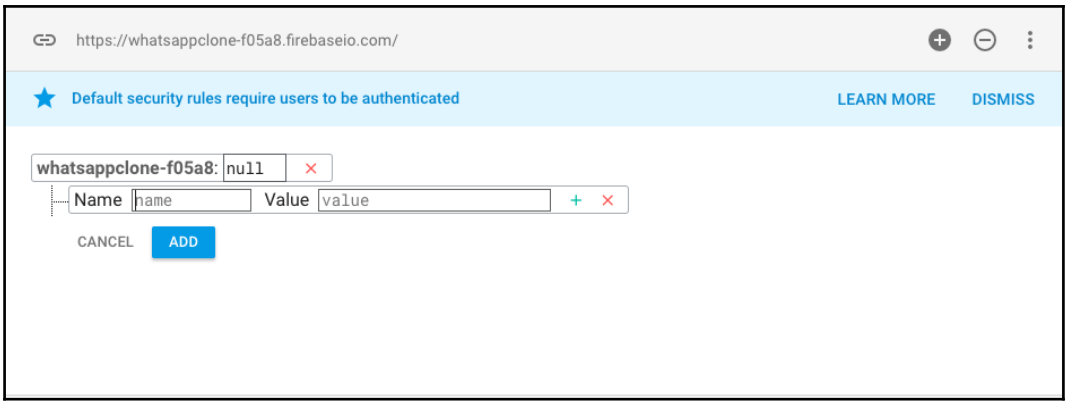
It's time to take our chat screen from the *whatsappClone* project and make it work using a Firebase real-time database. However, before we dive deep into the code and into setting everything up from the React Native end, let's set up our database.

In order to do so, we will select **Database** from the toolbar on the left, and we will be presented with a screen that shows our empty database:

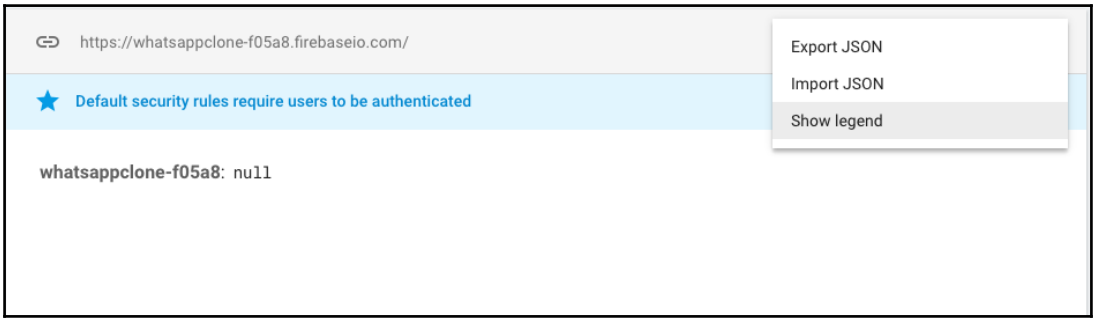


As you can see, our database is called *whatsappClone*--derived from the project name that I set up when I created the project and then an ID. We will use this ID later to retrieve our data from the *React Native* app.

Firebase real-time database stores data in a JSON format, so it's much like MongoDB and is very straightforward to web developers. Its API is simple and straightforward, and its management is very intuitive. You can create a new entry straight from the Firebase console:



Alternatively, you can simply import JSON:

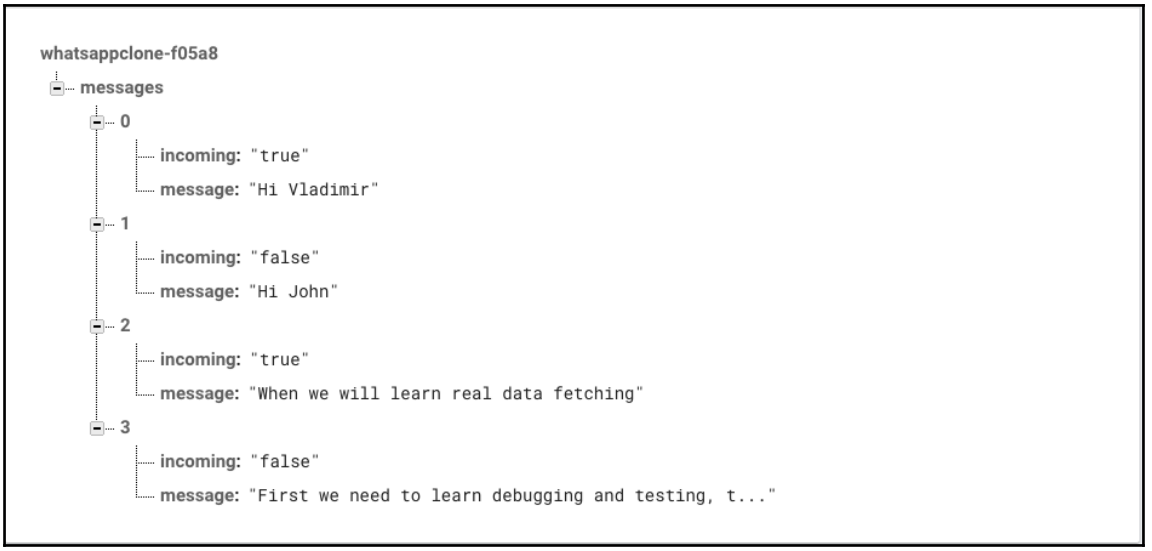


Let's do it. Let's take our message list from our *whatsappClone* app created in the previous chapters and add it into our database. First, let's change it to JSON:

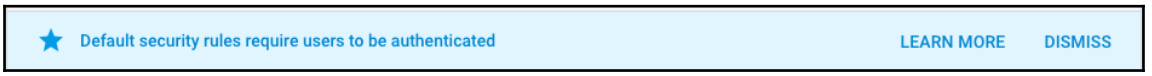
```
{
  "messages": [
     {
        "incoming": true,
        "message": "Hi Vladimir"
     },
     {
        "incoming": false,
        "message": "Hi John"
     },
```

```
        {
            "incoming": true,
            "message": "When we will learn real data fetching"
        },
        {
            "incoming": false,
            "message": "First we need to learn debugging and testing, then
                        we will learn styling and animations and then we
                        will learn real data fetching. Let's use mock data
                        for now ok?"
        }
    ]
}
```

Then, let's import it. After choosing **Import** from the menu on the right, we will choose our newly created JSON file, and after a couple of seconds, we will have our new database with a few messages created:



As you can probably see from a couple of earlier screenshots, when you view your database inside the Firebase console, you will see the following warning that will stay there until you click on **DISMISS**:

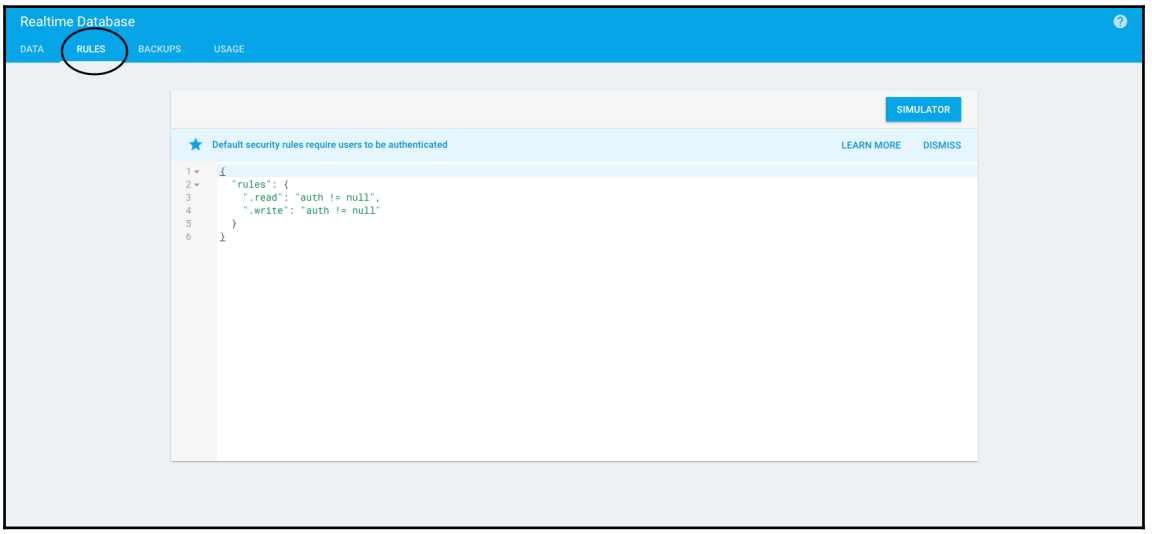


That brings me to another important topic, that is, Permissions.

Managing permissions

In Firebase, you can manage database access for your users, which were logged in through Firebase authentication. This can be read more in depth inside the Firebase security quickstart found at `https://firebase.google.com/docs/database/security/quickstart`.

However, let's take a look at how it looks in a nutshell. Under the **Realtime Database** title there is a navigation bar with **DATA**, **RULES**, **BACKUPS,** and **USAGE** tabs. Let's select the **RULES** tab. We will see the following:



This screen is used to set permissions for our application. Since we haven't dealt yet with authentication, let's make our rules public. This means that any user will be able to retrieve the data from the Firebase endpoint, without any authentication. For our *whatsappClone* app, let's keep it that way.

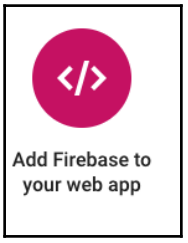
Let's change the rules to the following ones:

```
{
  "rules": {
  ".read": "true",
  ".write": "true"
  }
}
```

Bringing Firebase to React Native and fetching data

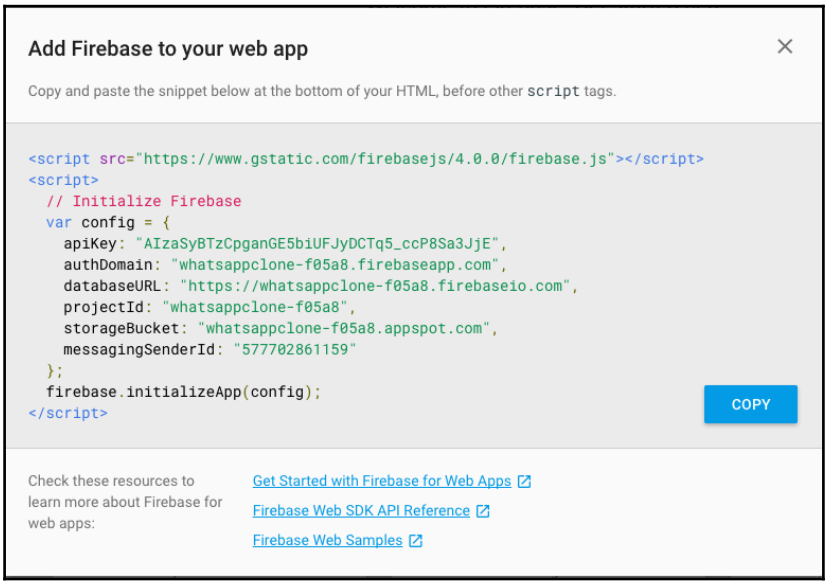
After configuring everything at the Firebase end, we will need to configure Firebase at our React Native end:

1. Let's go back to console dashboard by clicking on **Overview**, and then select **Add Firebase to your web app**:



   We are not dealing with a web app here, but this means that we are using JavaScript SDK to bring Firebase into our React Native app.

2. After clicking on it, the following modal will appear:



3. Copy the configuration values since we will need them to set up our Firebase connection.

Firebase runs on a JavaScript thread, and that means that with more complex applications, it can potentially affect the frame rate, causing jank with animations, touch events, and so on.
For better performance, you can use react-native-Firebase, which is a community package wrapped around Firebase SDK for Android and Firebase SDK for iOS.
It can be found here at `http://invertase.io/react-native-firebase/`.

Initializing your app

Let's begin integrating Firebase JavaScript SDK into our project:

1. First, let's install the official Firebase npm module:

```
npm install --save firebase
```

2. Now, let's create a new file in the services folder and call it `firebase.js`, and write the initialize function for our application:

```
import * as firebase from "firebase";

export const initialize = () => firebase.initializeApp({
   apiKey: "AIzaSyBTzCpganGE5biUFJyDCTq5_ccP8Sa3JjE",
   authDomain: "whatsappclone-f05a8.firebaseapp.com",
   databaseURL: "https://whatsappclone-f05a8.firebaseio.com",
   projectId: "whatsappclone-f05a8",
   storageBucket: "whatsappclone-f05a8.appspot.com",
   messagingSenderId: "577702861159"
})
```

3. The keys are taken from the Firebase console, as described in the previous section.

4. We will export our initialization as the `initialize` function. In our `src/api.js` we will export is as `initApi` function like this:

```
export const initApi = () => initialize();
```

Now, let's set up our Firebase at application root, `src/index.js`, but before doing so, let's take a look at how it looks:

```
import { StackNavigator } from 'react-navigation';
import routes from './config/routes';
export default StackNavigator(routes);
```

In `Chapter 4`, *Debugging and Testing React Native*, we've set it up using the `react-navigation` package. Now, let's alter this file to initialize our Firebase before mounting:

```
import React from 'react';
import { StackNavigator } from 'react-navigation';
import routes from './config/routes';
import { initApi } from './services/api';

const AppNavigator = StackNavigator(routes);

export default class extends React.Component {
  componentWillMount(){
    initApi();
}

render(){
  return (
    <AppNavigator />
    )
  }
}
```

We import the `initApi` function from `services/api` and, on `componentWillMount` lifecycle hook, we call it. In that way, we ensure our Firebase is set up before our application starts.

Setting up Firebase database listeners

Firebase SDK lets your database always be in sync with your client. That's why, instead of using regular data fetching in React Native, about which we will talk in a bit, we will need to set up database listeners to update our state whenever the database changes:

1. Inside our `firebase.js` file, we will create a new function:

```
export const setListener = (endpoint, updaterFn) => {
   firebase.database().ref(endpoint).on('value', updaterFn);
   return () => firebase.database().ref(endpoint).off();
}
```

   In this function, we will do two things. First, after passing `endpoint` and `updaterFn` as arguments, we will get a reference to our Firebase database endpoint using:

```
firebase.database().ref(endpoint)
```

2. Later on, we will pass `'messages'` as an endpoint, so we will get the following:

```
firebase.database().ref('messages')
```

3. Then, we will call on function and pass `updaterFn` to it, which we will pass to our `setListener` call.
4. Finally, we will return a function with `firebase.database().ref(endpoint).off();` so that later we can use it to remove the attached listener.

This is done for abstracting our database connection from our API service. In that way we can set up our database listeners inside the API service without worrying about Firebase specific implementation. In the future, if Firebase API changes, we'll simply change our `setListener` function inside `firebase.js` without worrying to update it elsewhere in code.

6. Now, in our `api.js` file, we will create a `getMessages` function and use `setListener` that we've just created:

```
export const getMessages = (updaterFn) => setListener('messages',
updaterFn);
```

7. The final step will be to import our `getMessages` function from `services/api` to our `ChatScreen`:

```
import { getMessages } from '../services/api';
```

Then in `ChatScreen` we set our state accordingly:

```
componentDidMount(){
   this.unsubscribeGetMessages = getMessages((snapshot) => {
   this.setState({
      messages: Object.values(snapshot.val())
    })
   })
  }
componentWillUnmount(){
  this.unsubscribeGetMessages();
}
```

We use `Object.values` on Firebase `snapshot.val()` because FlatList expects an array and firebase returns an object.

Since `getMessages` is calling `setListener`, which returns a remove listener function, `firebase.database().ref(endpoint).off();,`

We can set the `getMessages` result to `this.unsubscribeGetMessages`, and later call it in the `componentWillUnmount` lifecycle hook.

We also can get data only once without listening to updates. This can be done by calling

`firebase.database().ref(endpoint).once('value')` that will return a promise that can be treated in a regular wait or using `.then` or with `async await`.

Writing data to Firebase

Now, let's alter our application to be more interactive and enable posting messages to Firebase:

1. First, let's add a `pushData` method to the Firebase service:

```
export const pushData = (endpoint, data) => {
  return firebase.database().ref(endpoint).push(data);
}
```

We are using `push` method here because we want to push additional data. There are other methods--such as `set`, which overrides data and `update`, which updates a specific entry in a database.

You can read more about the use of these methods at `https://firebase.google.com/docs/database/web/read-and-write`.

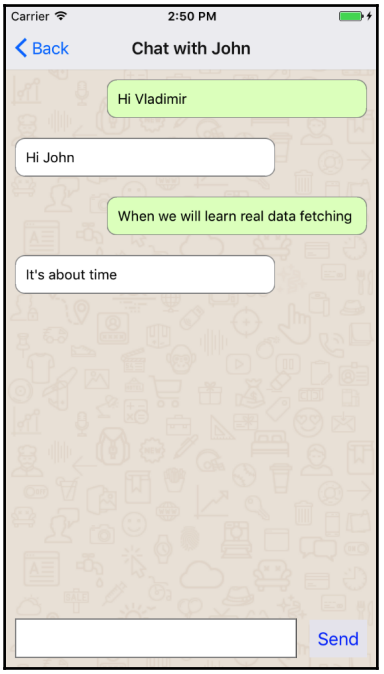
2. Now, let's add a proper API call in our API service:

```
export const postMessage = (message) => {
   if (Boolean(message)) {
     pushData('messages', {
        incoming: false,
         message
     })
   }
}
```

3. We check here whether a message is empty and use the `pushData` method we defined earlier.
4. It's time to refactor to our ChatScreen. Let's remove the `getMessageRow` function and move it together with corresponding styles to separate the component called message.
5. Also, let's create an empty `Compose` component, which we will fill in with a text input field and button. Our render function will look like this:

```
render(){
  return (
    <ImageBackground
      style={[ styles.container, styles.backgroundImage ]}
      source={require('../assets/imgs/background.png')}>
      <FlatList
        style={styles.container}
        data={this.state.messages}
        renderItem={Message}
        keyExtractor={(item, index) => (`message-${index}`)}
      />
      <Compose submit={postMessage} />
    </ImageBackground>
  )
 }
```

6. As you can see, we passed the `postMessage` function from the API as a submit prop to our `Compose` component.
7. We want our final result to look like the following screenshot or similar to it. Generally speaking, we want an input field at the bottom of the screen:



8. We will have two methods of submitting input, by entering text and by clicking on the `Send` button. We will need `React.class` and not a functional component, because we have an internal state. Also, we want to encapsulate our text state update inside our `Compose` component. Let's take a look at the code and go through it:

```
class Compose extends React.Component {

  state = {
      text: ''
  }

  submit(){
    this.props.submit(this.state.text);
    this.setState({
      text: ''
```

```
      })
      Keyboard.dismiss();
    }

  render(){
    return (
        <View style={styles.compose}>
          <TextInput
             style={styles.composeText}
             value={this.state.text}
             onChangeText={(text) => this.setState({text})}
             onSubmitEditing={(event) => this.submit()}
             editable = {true}
             maxLength = {40}
          />
           <Button
            onPress={this.submit}
          />
        </View>
      )
    }
}
```

9. We set up our internal state with a text key and wire up to the state update and the value of `TextInput`:

```
value={this.state.text}
onChangeText={(text) => this.setState({text})}
```

10. Then, we set `onSubmitEditing` to support the use case when a user types text and clicks on `return` on their phone keyboard:

```
onSubmitEditing={(event) => this.submit()}
```

11. We execute the `submit` function that we defined earlier. In the `submit` function, we do several things:
    1. First of all, we call the submit function passed from props and pass it to our `text` state value:

```
this.props.submit(this.state.text);
```

    2. Then, we use the `TextInput` method to clear the field of any text:

```
this.setState({ text: '' })
```

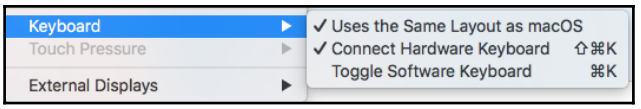
3. Then, we import `Keyboard` from React Native and execute the `dismiss` function to hide our Keyboard.
    1. Now, the next step will be to wire up `Button`. This is done by simply passing the `submit` function we've created in the `onPress` prop:

```
<Button
  onPress={this.submit}
  title="Send"
/>
```

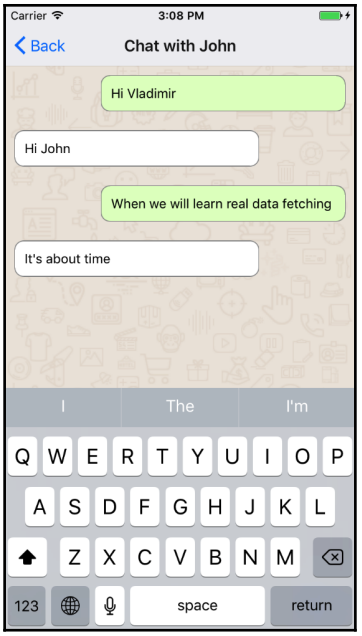
12. The final step will be to add styles:

```
const styles = StyleSheet.create({
   composeText: {
        width: '80%',
        paddingHorizontal: 10,
        height: 40,
        backgroundColor: 'white',
        borderColor: '#979797',
        borderStyle: 'solid',
        borderWidth: 1,
   },
   compose: {
      flexDirection: 'row',
      alignItems: 'center',
      justifyContent: 'space-between',
      margin: 10
  }
});
```

13. Now, it's time to test our app behavior. In order to enable **Keyboard** in the simulator go to **Menu** | **Hardware** | **Toggle Software Keyboard**:



14. We get some unexpected results after tapping on our input field:



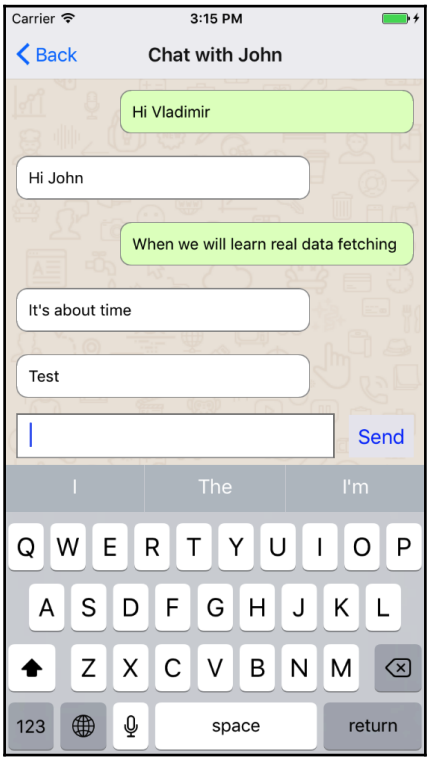
15. We will get Keyboard above our input field. We simply don't see what we are typing. That happens due to the fact that we should have used the `KeyboardAvoidingView` component to push our input field up when the Keyboard opens.
16. Let's do it. In `ChatScreen`, our render function now looks like this:

```
<ImageBackground
     style={styles.container}
     source={require('../assets/imgs/background.png')}>
     <KeyboardAvoidingView behavior="padding"
        keyboardVerticalOffset={this.keyboardVerticalOffset}
        style={styles.container}>
        <FlatList
          style={{ flex: 1 }}
          data={this.state.messages}
          renderItem={Message}
          keyExtractor={(item, index) => (`message-${index}`)}
        />
        <Compose submit={postMessage} />
      </KeyboardAvoidingView>
</Image>
```

We will pass padding behavior to our `KeyboardAvoiding` since we want it to add bottom padding to the last child component to be seen in the view. Next, we will pass `keyboardVerticalOffset` because on IOS we also have a text suggestion bar. Don't forget to define our `keyboardVerticalOffset` as a static property of `ChatScreen`:

```
//class definition before
   state = {
     messages: []
  }
   keyboardVerticalOffset = Platform.OS === 'ios' ? 60 : 0
   componentDidMount(){
//rest of code
```

Now, everything will work as expected:



Our application is now interactive. Our state is updated automatically when Firebase database changes due to the updater function we previously set up. This function updates our React state, and as a result, the `FlatList` component gets additional data.

We've covered a subset of all options Firebase in this section. Ensure that you check the official Firebase documentation to get more options tailored for your particular use case:

`https://firebase.google.com/docs/database/web/read-and-write`

Fetching data in React Native

React Native gives us HTML5 Fetch API for networking needs. In order to fetch content, we will use it in the following way:

```
fetch('https://myserver.com/api/data.json')
```

`Fetch` will return us a promise, so we can use it both using `Promise.then` or `async/await`, which is supported out of the box inside React Native.

Fetch can be configured with a second parameter, as in the following example:

```
fetch('https://myserver.com/endpoint/', {
   method: 'POST',
   headers: {
       'Accept': 'application/json',
      'Content-Type': 'application/json',
   },
   body: JSON.stringify({
     firstParam: 'yourValue',
     secondParam: 'yourOtherValue',
    })
 })
```

The important thing is that fetch response is a blob containing metadata on response itself, but not the actual data; so, in order to get a JSON, we should execute the JSON function from a response that also returns a promise. So, it will look similar to the following example when using promises:

```
function getMusicFromApi() {
  fetch('https://freemusicarchive.org/recent.json')
   .then(result => result.json())
   .then(musicData =>
     this.setState({
       musicData
     })
```

```
    )
    .catch(error => {
      console.error(error);
    });
}
```

Alternatively, it will look similar to the following example using `async/await`:

```
async function getDataFromApi() {
  try {
    let response = await fetch('https://freemusicarchive.org/recent.json');
    let musicData = await response.json();
    this.setState({
      musicData
    })
  } catch(error) {
    console.error(error);
  }
}
```

In the preceding two examples, we fetch data from `http://freemusicarchive.org` to display recent music albums.

In addition to the Fetch API, we can use third-party libraries, such as `axios` or `frisbee`, or just the plain `XMLHttpRequest` API since it's built in React Native.

Keep in mind though, that there is no CORS in native apps.

Websockets support in React Native

React Native supports WebSockets, and that is the reason why Firebase SDK for web also works for React Native since it's based on web sockets for pushing data to the client whenever the database changes.

You can use WebSockets in the same way you would use for the web, so you can push data from the server to the client without any polling mechanisms.

Fetching and uploading files

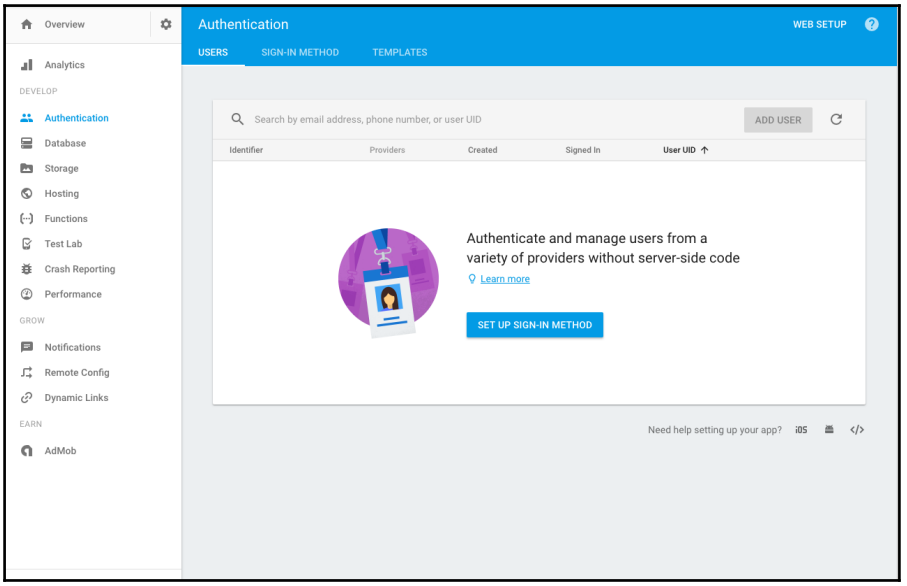
Downloading and uploading files is a complex task in React Native and not as straightforward as on the web; however, we have an amazing package, which gives us a great flexibility when dealing with files. It can even cache files for us in local storage and do many other useful things, which can be a long process to implement ourselves. I won't cover the use of this package in this book, however, I advise you to look into its documentation whenever the need for downloading or uploading files occurs:

```
https://github.com/wkh237/react-native-fetch-blob
```

Now, it's time to talk about authentication. In the preceding section, we made our *whatsappClone* application interactive. You can play around with it and add even more interactions, such as actual conversations, user groups, and more. Now, let's start by creating an *instagramClone* application, which we will start in this chapter and will continue in the subsequent chapters.

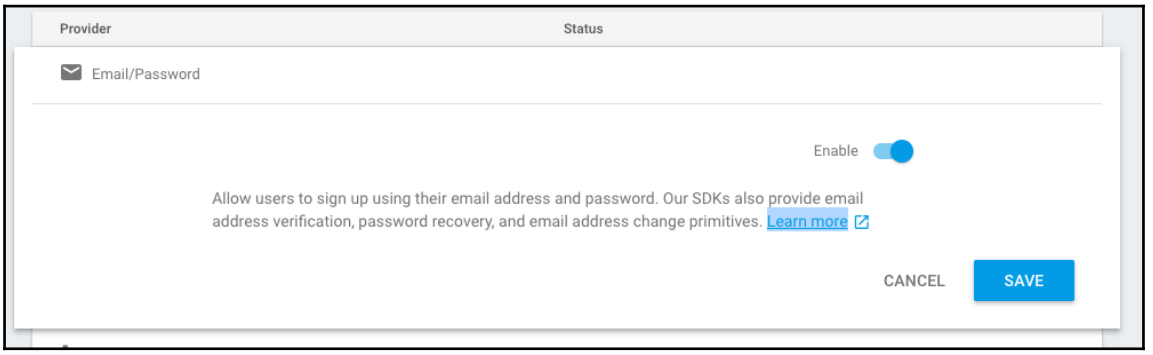
Setting up authentication at Firebase

In order to begin using Firebase authentication, first of all, we will need to set up our authentication by navigating to the **Authentication** tab in our Firebase console and click on **SET UP SIGN-IN METHOD**:



After clicking on it, let's select an **Email/Password** provider since we will be dealing with this type of provider now and click on the **Enable** switch on top-left corner to enable it.

Then, clicking on **SAVE** will save our configuration:



Creating functional Login and Sign Up screens

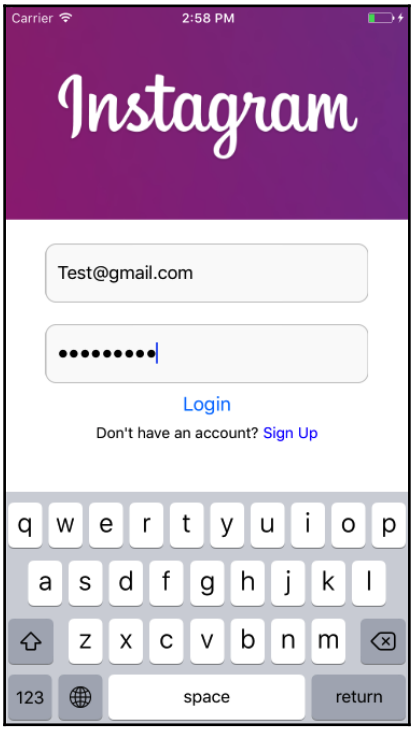
After everything is set on Firebase end, let's create our Login and Sign Up screens for our *instagramClone* application. Let's reuse our API and Firebase service from the *whatsappClone* app, but we will leave only Firebase authentication in our API service for now. In a Firebase service, we will need to change the configuration, which we've already learned how to obtain from the Firebase console.

Our `config/routes.js` file will now look like this:

```
import Login from '../screens/Login';
import Signup from '../screens/Signup';
import Home from '../screens/Home';
const routes = {
 login: { screen: Login },
 signup: { screen: SignUp },
 home: { screen: Home }
 }
export default routes;
```

Login screen

Let's now create our Login screen. Visually, it will look as in the following screenshot:



Let's take a look at the `render` function and then go through the code:

```
<View
       style={styles.container}>
       <Image
         style={[ styles.logo ]}
         source={require('../assets/imgs/InstagramLogo.png')}/>
       <ScrollView style={styles.container}>
         <TextInput
             ref={(textInput) => this._user = textInput }
             style={styles.inputField}
             value={this.state.text}
             onChangeText={(user) => this.setState({user})}
             onSubmitEditing={(event) => this._password.focus()}
             onFocus={() => this.clearValidationErrors() }
             editable={true}
```

```
              maxLength={40}
              multiline={false}
              placeholder="Phone number, username or email"
            />
            <TextInput
                ref={(textInput) => this._password = textInput }
                style={styles.inputField}
                value={this.state.text}
                onChangeText={(password) => this.setState({ password
                })}
                onSubmitEditing={(event) => this.submit()}
                editable={true}
                secureTextEntry={true}
                maxLength={40}
                multiline={false}
                placeholder="Password"
              />
                { this.state.error &&
                  <View style={styles.validationErrors}>
                    <Text style={styles.error}>{this.state.error}
                  </Text>
                  </View>
                }
            <Button onPress={() => this.submit() }  />
            <View style={styles.redirectLink}>
              <Text>Don't have an account? </Text>
              <TouchableOpacity onPress={() =>
              this.props.navigation.navigate('signup')}>
                <Text style={styles.link}>Sign Up</Text>
              </TouchableOpacity>
            </View>
          </ScrollView>
        </View>
```

You can see that we have an image for the Instagram logo, then we have two `textInput` fields, each one of them having its respective `ref`:

```
ref={(textInput) => this._user = textInput }
ref={(textInput) => this._password = textInput }
```

The `Password` field is defined with a `secureTextEntry` prop. This prop indicates whether to show the input field text or show circles instead.

Next to the input fields, we have a section where we will put the errors returned from Firebase:

```
{ this.state.error &&
 <View style={styles.validationErrors}>
 <Text style={styles.error}>{this.state.error}</Text>
 </View>
}
```

Below the section, we have `Button`, which will handle `submit` by calling the respective function when you click on it.

Note that for both inputs we also have a `submit` function passed to the `onSubmitEditing` prop on `TextInput`. On the first `TextInput`, though, we defined other behavior:

```
onSubmitEditing={(event) => this._password.focus()}
```

Basically, we tell our app to focus on the password field whenever an entry is submitted to the username field.

Under the `Button`, we have navigation button to the **Sign Up** screen.

Our submit function will look like this:

```
async submit(){
  try {
    const response = await login(this.state.user, this.state.password)
    this.clearAndNavigate('home')
  } catch ({ message }) {
    this.setState({
      error: message
    })
  }
}
```

We will call a login function from the API service and pass it a username and password. When we get an error, we will set our error state to display errors returned from Firebase. If there are no errors we will call the `clearAndNavigate` function:

```
clearAndNavigate(screen){
  this.setState({
    user: '',
    password: ''
   })
   Keyboard.dismiss();
   authService.isAuthenticated = true;
   this.props.navigation.dispatch(
```

```
      NavigationActions.reset({
       index: 0,
       actions: [
         NavigationActions.navigate({ routeName: 'home' })
       ]
      })
    );
  }
```

This function clears the username and password using the React `setState`. Then, we dismiss keyboard and set the `isAuthenticated` flag in `authService` to `true`, and we replace our navigation screen with a home screen. You can read more about how it's done in the react-navigation document. Basically, we don't want the user to be able to get back to the Login or SignUp page, so we will use a technique of dispatching navigation actions instead. We will talk more about react-navigation in upcoming chapters.

The Sign Up screen

Our Sign Up screen will look the same as the Login screen, with slight differences. It will have a bottom link navigating to **Login** instead of Sign Up, and its submit function will be different. It will call the Signup function from the API service instead.

Authentication logic

Now, it's time to dive into our authentication logic. We will create an `authService` file in our `src/services` folder:

```
class AuthService {
  _isAuthenticated = false
  set isAuthenticated(bool){
     this._isAuthenticated = bool;
    }
  get isAuthenticated(){
    return this._isAuthenticated
   }
  checkAuth(){
     return new Promise(resolve => setTimeout(() =>
        resolve(this.isAuthenticated), 1000))
  }
}
```

As you can see, it's a class with a setter and getter for authentication and a fake `checkAuth` function, which returns `isAuthenticated` after one second.

In later chapters, we will implement authentication persistency, so we will alter this `checkAuth` logic.

Let's create an instance of our service:

```
const authService = new AuthService();
```

Now, it's important to create `authDecorator`. Here we will use a React principle of higher order component.

Higher order component (**HOC)** is a component that transforms any component and adds additional behavior to it. Usually HOC will look like a function that receives a component and returns a new React class with additional logic. In this class render function it returns the passed component with additional logic.

`authDecorator` is a HOC that will wrap any component that we want to be accessed only after the authentication check:

```
export const authDecorator = (Component) => {
  return class AuthChecker extends React.Component {
    state = {
      auth: false
    }
    componentDidMount(){
      authService.checkAuth().then(isAuthenticated => {
        if (isAuthenticated) {
          this.setState({ auth: true })
        } else {
          this.props.navigation.dispatch(
            NavigationActions.reset({
              index: 0,
              actions: [
                NavigationActions.navigate({ routeName: 'login' })
              ]
            })
          );
        }
      });
    }

    render() {
      return this.state.auth ? <Component/> : <SplashScreen />
    }
  }
}
export default authService;
```

We will show or SplashScreen our component. Let's now wrap our Home component with this decorator:

```
export default authDecorator(Home)
```

When our application starts, the router will enter the home route, and after checking authentication, it will redirect to the Login screen. Now, we are left with a really small task, that is, connecting everything to Firebase. Remember that we called the Login and SignUp functions from the API.

Now, it's time to add them to the API service; however, there is nothing specific to our API there, so we can just pipe them from the Firebase server.

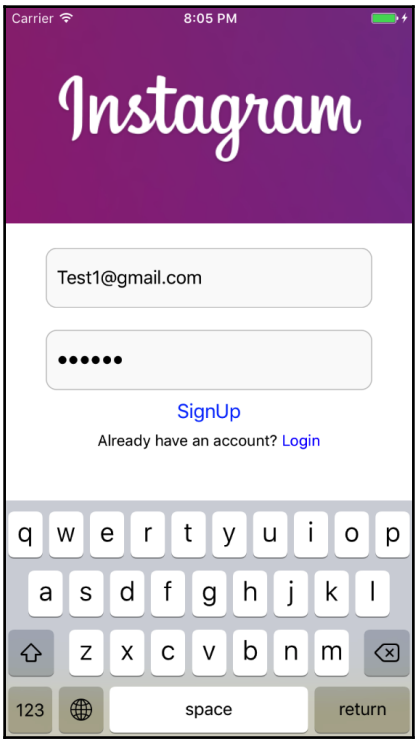
Our API service will now look as follows:

```
import { signup, login, initialize } from './firebase';
export const initApi = () => initialize();
export {
  login,
  signup
}
```

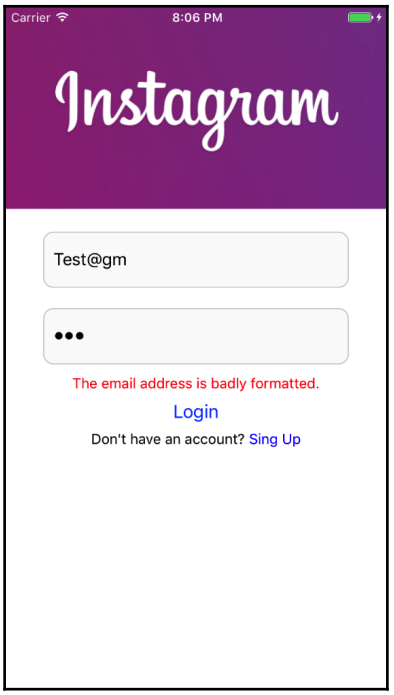
In the Firebase service, we will add login, logout and signup functions

```
export const login = (email, pass) =>
  firebase.auth()
    .signInWithEmailAndPassword(email, pass)

export const logout = () =>
  firebase.auth().signOut()

export const signup = (email, pass) =>
  firebase.auth().createUserWithEmailAndPassword(email, pass);
```

Now, if we try to **SignUp**, we will be redirected to Home after the process has completed:



Then, when we will try to **Login** and accidently enter an invalid email, validation will work, too. In fact, any error we get from Firebase will be displayed in our errors field:

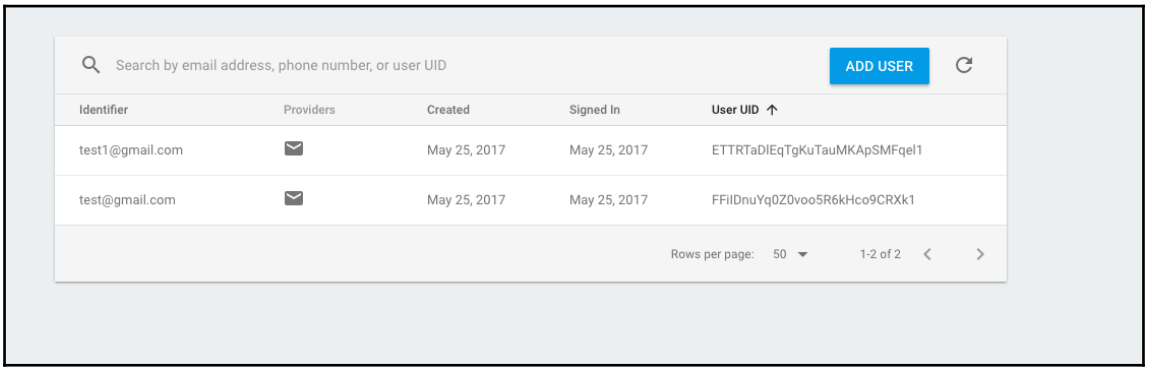


You can see that we've also styled our errors to be in red and have pretty nice looking input fields. This is done with the following styles:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: 'white'
  },
  logo: {
    width: '100%',
    height: 200
  },
  inputField: {
    marginTop: 20,
    alignSelf: 'center',
    height: 55,
    width: '80%',
    backgroundColor: '#FAFAFA',
    borderWidth: 1,
```

```
      paddingHorizontal: 10,
      borderRadius: 10,
      borderColor: "#CACACA"
    },
    redirectLink: {
      flex: 1,
      flexDirection: 'row',
      alignSelf: 'center'
    },
    link: {
      color: 'blue'
    },
    validationErrors: {
      flexDirection: 'row',
      justifyContent: 'center',
    },
    error: {
      marginTop: 10,
      textAlign: 'center',
      color: 'red'
    }
  })
```

Let's take a look at how newly created users look in the Firebase Console. For that, we simply access our **Authentication** tab in the console, and we will see list of all our users:



Authenticating via social providers

Authenticating via social providers can be tricky in React Native since you basically need to communicate with a social provider SDK for the particular platform you are developing on. However, there are several projects that you can use to make it possible. While setup is not as straightforward as adding Firebase authentication, It's pretty much detailed and well documented:

`https://github.com/fullstackreact/react-native-oauth`

Another amazing project, which also enables you to use any of the provided Facebook APIs bundled in Facebook SDK is `https://github.com/facebook/react-native-fbsdk`.

The reason I won't cover the exact steps for authenticating with social providers is mostly because SDK changes, the projects I mentioned before have changed, and IOS and Android versions change, so it's important that I guide you to the projects that are considered as the best practice in industry and then you can read more on your own.

In later chapters, we will specifically use `react-native-oauth` when we are creating our *Twitter clone* application.

Summary

In this chapter, we were introduced to Firebase, a service that we can use for several tasks, which usually require a server. We've set up our account in Firebase and got a basic understanding of what its console looks like. Then, we've changed our *whatsappClone* app to be fully interactive by connecting it to a Firebase real-time database. We've added to our *whatsappClone* app a text input field and submission logic, so our data can be both retrieved from and submitted to the Firebase database. Then, we overviewed other data fetching techniques that can be used in React Native and progressed toward the authentication topic. We used Firebase to authenticate our user with username and password and built Login and Singup screens for the *instagramClone* app we will continue working on in the next few chapters. We've created a decorator, which we will reuse for components that should be shown only for an authenticated user. In our use case, it's the Home component. Lastly, I mentioned two important packages that you should be familiar with if you want to start implementing authentication with social providers. In the next chapter, we will cover important techniques of state management. We will get familiar with Flux architecture in general and specifically with Redux, a functional Flux implementation. It doesn't matter if you already know Redux or you are just starting with React Native without prior experience in Redux, the following chapter will be equally important for you. So, get ready for some functional programming concepts ahead.

8
Implementing a Flux Architecture with Redux or MobX

Welcome to the eighth chapter. In this chapter, we will overview Flux architecture as the architectural standard for writing React applications in general. It's important to note that if you come with prior React development experience, this chapter may sound familiar to you. We will start the chapter by overviewing the Flux architecture pattern in general versus the **Model View Controller** (**MVC**) architectural pattern, and then we will dive deeper into Redux, a library that for several years has been standard for state management in React applications. React Native is not different. It's commonly used with Redux or MobX, an emerging solution for state management based on functional reactive programming concepts. After getting familiar with Redux and learning how you should connect your React Native app to it, we will overview MobX as an alternative solution. So, if you have prior experience with Redux and MobX, you might learn only few new bits. However, even if you are familiar with these libraries, it's important to recap on Redux and MobX core concepts so that you will be ready to create a full Twitter client app in `Chapter 11`, *Creating Twitter client app fully functional clone*. During the course of this chapter, we will work with our Whatsapp clone application and eventually move its state management from our components to Redux or MobX.

So, let's summarize what we will learn in this chapter:

- Getting familiar with the Flux architecture concept
- Getting familiar with the Redux basic concepts of Stores, Action creators, Reducers, and middleware
- Setting up Redux and connecting it to your application

- Triggering state updates by dispatching actions
- Getting familiar with another state management library--MobX--inspired by functional reactive programming paradigms

As mentioned at the beginning of this book, React introduced a new and amazing way of building UI for web and mobile. React itself has always been a View library. The React reconciliation mechanism (a mechanism of comparing changes and batching UIs updates) and the concept of Virtual DOM gave it an option to be used everywhere on mobile, desktop, and even on hardware devices.

React Native is much more than a library; it is a framework that consists of dozens of native device API's wrappers, which we will cover in the next chapter. However, it lacks a proper state management system. Our state is local to component, and in order to propagate it to child components, it should be passed via props or on a React context object. You can read more about the context at `https://facebook.github.io/react/docs/context.html`.

The idea behind React Native is using the same techniques we use on the web for writing React apps on mobile. So we will use the same ecosystem as we would have used on the web.

We will use Flux architecture as advised by Facebook and will use one of the libraries that became the de facto standard--Redux or MobX. However, first let's understand what is Flux Architecture, how it's different from MVC architecture, and what it comes to solve before we will dive deeper into what Redux is.

What is Flux architecture?

Flux architecture didn't come from a void. There were several factors responsible for its emergence. So, it's important to understand these factors.

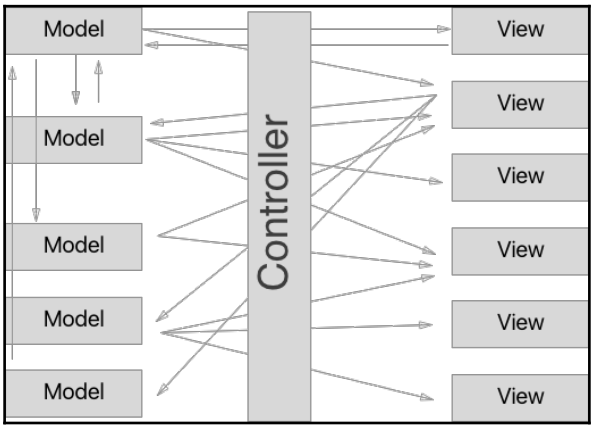
For years, the MV* pattern was popular among most major frameworks, such as Backbone, Angular, Knockout, Ember, and more. MV* is a popular shorthand for variations of MVC architectural pattern, which the above-mentioned frameworks tried to implement, not necessarily following the pattern strictly. The idea of all patterns was to separate Model, and View behind the Controller. Since it wasn't always the controller, but sometimes the Presenter or View model, the pattern is called **MV***.

Even though MVC architectural pattern is common and widely used, it introduces several problems.

MVC problem

In huge applications, such as Facebook, when working with tons of Models and Views, in standard MVC implementation, the controller acts as a data source for Views. It gets the data from the model as well as updates by providing action handlers for the Views. Models can communicate with each other too but in a limited way through a broadcast mechanism. Usually, in large applications, multiple MVCs are combined and at some point it gets to an unmaintainable complexity.

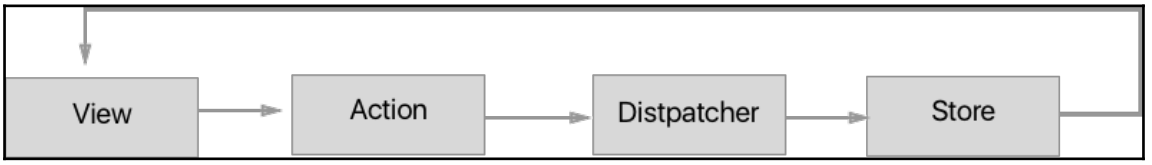
While View does not communicate directly with the Model and it's done via **Controller**, View can update the model, and as a result, a model will update an other Model and a View, cascading usually unnecessary updates. Since some of the changes happen asynchronously, the application can easily get to the state where the model is unpredictable because of the complex data flow. Of course, all this can be fixed; however, it's really hard to debug the data flow. So, let's illustrate what it can look like:



It's not even debuggable, and not readable. So, in Facebook, they came up with a solution--unidirectional data flow.

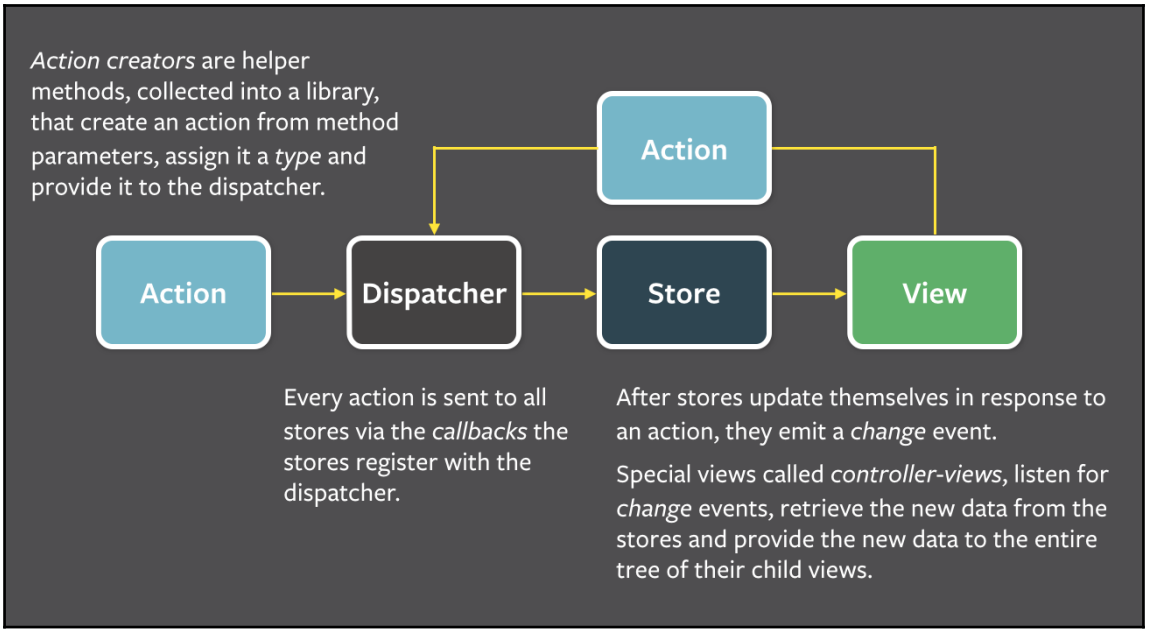
The unidirectional data flow solution

Consider the following diagram:



You can probably see--even without an explanation--that the preceding diagram is much clearer than the one in the preceding section.

When reading official Flux documentation, you will probably see the following diagram:



It, however, can get confusing,so let me briefly explain the main idea.

How it works

The main idea is dispatching actions. Actions are simple objects that are emitted to the world by dispatcher, and stores are the ones that listen to these actions; when a specific action is triggered, they update the View.

In that way, View cannot directly update any data on the model since it can only throw out an action.

Actions are usually thrown by action creators. Take a look, for example, at the following action creator:

```
function onClickAction(payload){
  return {
      type: "CLICK",
      payload
    }
 }
```

So, in our React component, our `onPress` callback can be something like this:

```
onPress={
   (e) => {
     dispatch(onClickAction(e))
   }
}
```

We use some kind of dispatch function, which can differ based on the Flux implementation and call action creator. As a result of calling it, an object (actual action) is dispatched. Store that listens to it will choose what to do and will then update the View.

You can read more about the Flux architectural pattern at `https://facebook.github.io/flux/docs/in-depth-overview.html`.

There are lots of libraries implementing Flux architecture; however, there are two that are considered standard--**Redux** and **MobX**. We will cover their concepts in this chapter.

Redux concepts and usage

In this section, we will take a look at one of Flux architecture implementations that is considered a standard in both Web and React Native development. It's called Redux. We will overview its concepts and usage before we actually connect Redux to our application. It's important to understand that, even though we will cover major aspects of Redux here, an in-depth coverage of Redux is out of the scope of this book, so if you want to learn more I suggest that you look in official Redux documentation for a more in-depth explanation at

`http://redux.js.org/`.

What is Redux?

Redux is a functional Flux implementation. It's based on *language and architecture*. The main idea of Redux is that there is an immutable application state that lives outside of the components. In response to actions, relevant parts of the state are overridden instead of modifying directly.

Redux in a nutshell

Think of the application state as a simple object like this:

```
{
 messages: [{ message: 'Hi Vladimir', incoming: true }]
}
```

State can be modified only by dispatching actions like this:

```
{ type: 'ADD_MESSAGE', payload: { message: 'Hi John', incoming: false }}
```

In order to modify the state we will modify it in the Redux Reducer which we will go over later. In the end, we will modify our messages state by returning a new messages array with a new message concatenated into it. At any point, we won't modify our state directly. In fact, if we do so, we will get an error from Redux.

Three principles

Redux is based around three important principles. Some of them may sound a bit weird, but in conjunction, they work pretty well.

A single source of truth

At the beginning of the chapter, we've talked about Flux architecture and multiple stores to store our state. In Redux, this is not the case. We have only one store, and the whole application tree is stored inside this store--this may sound weird. After all, it may sound like I need to determine the whole application state before starting to develop the app.

That's not the case. Due to the Redux Reducers mechanism, we can define our state as we introduce new modules to our app, so our state will become larger as the application grows. It's easier to debug and persist our state in development. Redux is famous for being introduced as part of the *Dan Abramov React Europe talk* about time travel in React. His idea was to implement undo - redo functionality for all user interactions.

This may sound a little complex; however, with Redux, it became trivial to implement by just persisting the state tree. Check this out at `https://www.youtube.com/watch?v=xsSnOQynTHs`.

State is read-only

As I've already mentioned before, you can't update your state directly, but only by emitting an action that will describe what is about to happen. Actions are simple objects, and, hence, can be serialized, stored, and logged for debugging purposes.

Changes are made with pure functions

In response to an action, the state tree is transformed with pure functions called **Reducers**. For example, for our Whatsapp clone, a basic Reducer will look like this:

```
const initialState = {
   messages: []
 }
 export default function messagesReducer(state = initialState, action){
   switch (action.type) {
     case actionTypes.ADD_MESSAGE:
       return {
         messages: [
           ...state.messages,
         action.payload
        ]
     }
   default:
     return state;
    }
 }
```

As you can see, message Reducer does not make any modifications on the state, but in response to the action, it returns a new array. `[ ...state.messages, action.payload ]` is an equivalent of

`state.messages.concat(action.payload)` in ES6.

Basic building blocks

After understanding the three principles of Redux, let's take a look at its core concepts: Actions, Reducers, Store, and Middleware.

Actions

Actions are basically descriptors of manipulation you want to perform or to trigger state updates. Sometimes, you will also use actions for logging purposes. They are plain objects with a type key, using which you differentiate between actions and payload. You can use the following naming convention: `{ type: 'ACTION_TYPE', payload: {}}` or `{ type: 'ACTION_TYPE', keyA: {}, keyB: {}}`.

It doesn't really matter. The most important thing is that you know how to find your action type and how to extract your action payload later on, whether it's a payload object or various keys you pass on an action object.

Since actions are simple objects, but the data they pass can be dynamic, actions are created by running an action creator function. In our example, when we want to send the `{ type: 'ADD_MESSAGE', payload: { message: 'Hi John', incoming: false }}` action, our message is not static, so we will wrap our action in the action creator:

```
function addMessage(message){
     return {
             type: 'ADD_MESSAGE',
             payload: {
                     message,
                     incoming: false
               }
        }
}
```

As you can see, message is passed dynamically to the `addMessage` action creator, and as a result, we have an action object.

In order to dispatch our action, we will use the `store.dispatch` function. In our example, it will look like this:

```
store.dispatch(addMessage('Hi John'));
```

Reducers

Actions describe what happened, however, someone has to modify the state in response to these actions. This is what Reducers do. As we've seen, it's a pure function, which gets a previous state and an action as an argument and returns a new state. We can write it in a general way like this:

```
(previousState = initialState, action) => newState
```

Now, let's compare it with our messages Reducer we've seen earlier:

```
export default function messagesReducer(state = initialState, action){
   switch (action.type) {
     case actionTypes.ADD_MESSAGE:
       return {
         messages: [
           ...state.messages,
           action.payload
         ]
       }
     default:
       return state;
   }
 }
```

As you can see in the preceding code, we get state as the first argument, which defaults to `initialState`, and an action as the second argument.

It's important to note that an application can, and usually will, have several Reducers, each one responsible for a specific part of application state tree. In order to combine these Reducers, you will use the `combineReducers` function from the Redux package and pass your Reducers to it, as in the following example:

```
import { combineReducers } from 'redux'

 const myApp = combineReducers({
   stateKeyA: reducerResponsibleForStateKeyA,
   stateKeyB: reducerResponsibleForStateKeyB
 })
```

The resulting application state will be shaped by an object passed to `combineReducers` and states of each Reducers. So, if, for example, `reducerResponsibleForStateKeyA` has the `{ keyC: []}` state and `reducerResponsibleForStateKeyB` has the `{ keyD: []}` state, then the whole app state will look as follows:

```
{ stateKeyA: { keyC: [] }, stateKeyB: { keyD: [] } }
```

Group your application state logic using different Reducers. Since Reducers are simple functions, you can nest your Reducers one inside of other.

You can read more on Reducers in official Redux documentation, at `http://redux.js.org/docs/basics/Reducers.html`.

Store

While actions define what happen and Reducers update our application state in response to these actions, store is the mechanism that glues everything together. Store holds the app state, allows access to the `getState()` function, which we can call directly from the store to get our state at any point in time, allows the state to be updated via `dispatch(action)`, and much more. In order to create store, we will import the `createStore` function from the Redux package and pass our Reducer to it. If there are several Reducers, we will pass the result of the `combineReducers` function.

Consider this example:

```
import { createStore } from 'redux'
import whatsappApp from './reducers'
let store = createStore(whatsappApp)
```

Now, in order to update our state, we can use `store.dispatch(addMessage('Hi'))`, as mentioned earlier. If we log `store.getState()` after running `store.dispatch`, we will see our state updated.

Middleware

Middleware is an amazing feature of Redux inspired by *Express* and *Koa* frameworks for NodeJS. They give Redux its extensibility. Generally speaking, middleware is an extension between dispatching an action and the moment it reaches the Reducer. It's used for lots of purposes, such as logging, reporting, routing, asynchronicity, and much more. In order to add middleware to our project, we can, for example, install the `redux-logger` package, which supplies logging middleware.

In order to add middleware to our store, we will supply them by calling the `applyMiddleware` function from Redux and passing it our middleware as arguments. Consider the following example for logger:

```
import { createStore, applyMiddleware } from 'redux';
 import logger from 'redux-logger';
 import rootReducer from './reducers';

export default createStore(rootReducer, applyMiddleware(logger));
```

Now, if we dispatch our action in the Chrome console, we will see the following:

Logging will give you the whole flow of your actions and will let you understand where your state failed to update or hasn't updated correctly. In this example, we saw that in the preceding state, we had an empty messages array, then we saw what action was thrown as well as its payload, and finally we saw that our messages array now has one message with the exact same data that was passed in an action. If the data is not in the same format, we can see it right away without additional debugging.

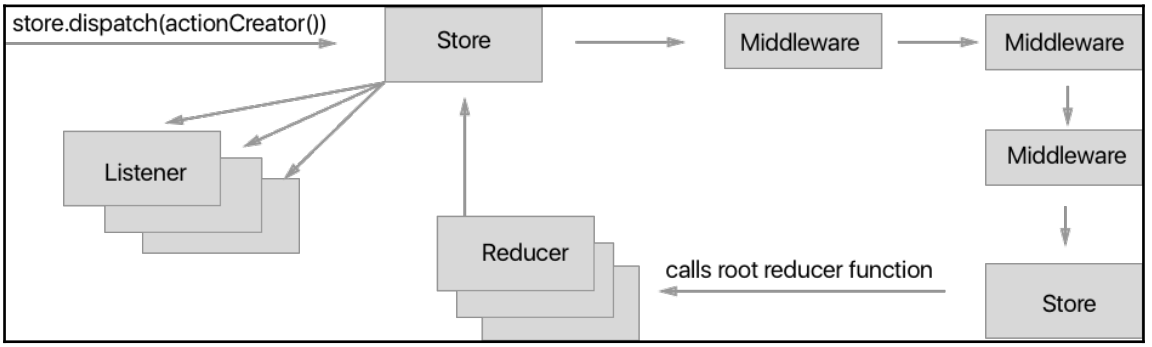
Middleware can be used as imported packages supplied to the `applyMiddleware` function or can be written by you. I won't cover it here, however, you can look at `http://redux.js.org` for more information.

Data flow

Let's summarize now how the data will flow in our application:

1. You use `store.dispatch` and pass it the result of `actionCreator`, an action object.
2. Store passes your action to the middleware chain if any middleware is configured.

3. After all middleware has been executed, store, calls root the Reducer function.
4. Root Reducer passes the execution to the combined Reducers.
5. Reducers return the relevant state for the chunk of state they are responsible to.
6. Whoever listens to a store using the `store.subscribe(listenerFunction)` method gets updates:



Connecting Redux to your app

Before connecting Redux to our application, let's set up Redux for our *whatsappClone* app:

1. First of all, let's create an `addMessage` action and put it inside the `app/actions.js` file:

```
export const actionTypes = {
   ADD_MESSAGE: 'ADD_MESSAGE'
 }

 export function addMessage(message){
   return {
     type: actionTypes.ADD_MESSAGE,
     payload: {
       message,
       incoming: false
     }
   }
 }
```

2. Next, let's create a `reducers` folder and add the `messagesReducer.js` and `index.js` files. Inside `index.js`, we will simply import and export `messagesReducer`; however, in a real-world app, it will usually export the `combineReducers` function result:

```
import { combineReducers } from 'redux';
import messages from './messagesReducer'

export default messages;
```

3. Our `messagesReducer.js` will be as follows:

```
import { actionTypes } from '../actions';

 const initialState = {
   messages: []
 }

 export default function messagesReducer(state = initialState,
 action){
   switch (action.type) {
     case actionTypes.ADD_MESSAGE:
       return {
         messages: [
           ...state.messages,
           action.payload
         ]
       }
     default:
       return state;
   }
 }
```

As you can probably see in the preceding code, instead of using strings for action types we've defined constant `actionTypes` in the `actions.js` file and use it whenever we need it.

4. Our next step will be to define our store. We will create the `app/store.js` file and add `rootReducer`, imported from the `reducers` folder, as well as logger and thunk middleware. We've seen how useful logger middleware can be, and later on, we will use thunk middleware to deal with async actions:

```
import { createStore, applyMiddleware } from 'redux';
 import logger from 'redux-logger';
 import thunk from 'redux-thunk';
 import rootReducer from './reducers';
```

```
const middlewares = [thunk];

if (__DEV__) {
  middlewares.push(logger);
}

export default createStore(
  rootReducer,
  applyMiddleware(...middlewares)
);
```

You can note that we use a `__DEV__` global variable to specify our logger middleware, only on a dev environment. It's a good practice to use the `__DEV__` global variable that React Native supplies to us to eliminate unnecessary logging.

Containers versus Components

In `Chapter 2`, *Working with React Native*, we've talked about Stateful and Presentational components. In Stateful components, we've used the React state to deal with all state updates. With Redux, we take most of the React state out of components into the application state and let Redux manage it for us. However, we should somehow connect our previously Stateful components to Redux. We use the same idea that we had with Stateful components, meaning that they are basically containers for other Presentational components and are in charge of passing the state into props of Presentational components. So, in Redux, we will call components connected to Redux Containers and components that are purely Presentational components.

In our example, ChatScreen is both a screen and a container because it's in charge of state updates as well as the basic layout of our screen. In our case, it's fine; however, you will note that in a more complex app, you won't necessarily have one container per screen. You can have several ones. If we need to create containers separately, we will usually stick to naming our files as `myComponent.js` or `myContainer.js`.

Provider

In order to properly connect our store to React, we will use the `react-redux` package:

The first step will be setting our store on React context by wrapping our whole application with the `Provider` component.

Let's take a look at our `app/index.js` file and wrap it with the `Provider` component from the `react-redux` package:

```
import { StackNavigator } from 'react-navigation';
 import routes from './config/routes';
 import { initApi } from './services/api';
 import React from 'react';
 import { Provider } from 'react-redux';
 import store from './store';
 const AppNavigator = StackNavigator(routes);

 export default class App extends React.Component {

   componentWillMount(){
     initApi();
   }

   render(){
     return (
       <Provider store={store}>
         <AppNavigator />
       </Provider>
     )
   }
 }
```

As you can see in the preceding code, it's as simple as wrapping our navigator with Provider. Speaking of `react-navigation` and Redux, it's totally fine to use your navigation outside of Redux; however, if you want to fully integrate your navigation with the Redux state, you can use the following guide: `https://reactnavigation.org/docs/guides/redux`.

After wrapping our application with Provider, we need to connect our containers to Redux using the connect function.

Using connect

Connect is a function supplied by the `react-redux` package, which lets us to connect our store passed to Provider to the actual component. Connect has the following signature:

```
connect(mapStateToProps, masDispatchToProps)(MyContainer);
```

It automatically re-renders our component when state updates happen. More specifically, it uses the `mapStateToProps` function to determine which state updates your container should be aware of and re-renders it only when these state updates happen.

Let's take a look at how we can connect our ChatScreen to Redux. After importing the `connect` function with `import { connect }` from `'react-redux';`, we will export the following instead of exporting the `ChatScreen` component:

```
export default connect(mapStateToProps,
mapDispatchToProps)(ChatScreen);
```

Now, we will need to implement both `mapStateToProps` and `mapDispatchToProps` functions.

The former is used to determine which parts of Redux state to pass into our container props. The latter is used to wrap our action creator into a dispatch call:

```
function mapStateToProps(state) {
   return {
     messages: state.messages
   }
 }
function mapDispatchToProps(dispatch){
   return {
     addMessage: (message) => {
       dispatch(addMessage(message))
     }
   }
 }
```

As you can see, `mapStateToProps` gets state as an argument and it returns an object. This object will be merged into container props. Whenever the Redux state is updated and messages are changed, the Container is rerendered.

`mapDispatchToProps` gets store dispatch function as an argument and returns an object, which also will be merged into container props. Later, when using `this.props.addMessage('hey')` it will call `store.dispatch(addMessage('hey'))` instead.

There is also a shorthand for `mapDispatchToProps`. Instead of passing a function, you can pass an object to get the exact same behavior:

```
export default connect(mapStateToProps, { addMessage })(ChatScreen);
```

Getting relevant state keys with selectors

In our `mapStateToProps` function, state retrieval is pretty straightforward; however, in a more complex application, this cannot be really simple. Consider that you want to see only `filteredMessages`. In that case, your container need to be aware of the state shape that is wrong. For that, Redux introduced a concept of selectors.

Consider the following:

```
function mapStateToProps(state) {
   return {
     messages: state.messages
   }
 }
```

Instead of the preceding code, we can write the following:

```
function mapStateToProps(state) {
   return {
     messages: getMessagesSelector(state)
   }
 }
```

`getMessagesSelector` can be in our Reducer or can reside in a separate file. It will look like this:

```
function getMessagesSelector(state) {
   state.messages
}
```

You can use selectors more efficiently using **Reselect library**, which introduce more advanced concepts; You can read more about it in its official Readme:

`https://github.com/reactjs/reselect`

Using redux-thunk for async actions

So, up to now, we've seen how we can dispatch actions and saw how our View is updated in response. However, there is a reason we haven't implemented a View update for our *whatsappClone* app, yet. We are dealing with asynchronous actions here. We want to trigger an action that is not a plain object. An action should be a function that updates a Firebase database. This is called a side effect. It means that an action is not only a description of what is about to happen to a state, but also does something else "on the side".

Async data flow

Typical data flow for async actions consist of several steps:

1. Throwing a simple action, indicating that we are starting an async flow.
2. Executing a side effect, such as making ajax calls or, in our case, setting Firebase listeners.
3. Throwing another action on success or on error.

While we have a mechanism of throwing actions, we lack a mechanism for executing side effects. It would be wrong to execute them from components as well as from Reducers. We won't write our own middleware when we need a side effect. In order to help us with that, we have redux-thunk.

Adding redux-thunk

Redux-thunk basically transforms our `actionCreators` to be capable of returning functions instead of objects. The return function will automatically get two arguments: dispatch and getState. This enables us to fire several actions from one action creator.

Let's add redux-thunk for our application.

First of all, let's change our container component a bit. We will remove the `componentDidMount` logic, which dealt with Firebase connection and instead will throw a custom action:

```
componentDidMount(){
  this.props.subscribeToGetMessagesFromServer();
}

 componentWillUnmount(){
   this.props.unSubscribeToGetMessagesFromServer();
 }
```

We will do the same for our `Compose` component `submit` callback:

```
<Compose submit={this.props.postMessageToServer} />
```

Also, don't forget to import all these actions and pass them to `connect`:

```
import {
      postMessageToServer,
      subscribeToGetMessagesFromServer,
      unSubscribeToGetMessagesFromServer
```

```
} from '../actions';

// ...
export default
     connect(mapStateToProps, {
             postMessageToServer,
             subscribeToGetMessagesFromServer,
             unSubscribeToGetMessagesFromServer
       })(ChatScreen);
```

Now, after removing all side effects from our container, we can focus on our action creators. Since we are using thunks, they become more complex than functions that return simple objects, but in a reasonable way.

First of all, let's set up our `actionTypes`:

```
export const actionTypes = {
  SUBSCRIBE_GET_MESSAGES_TO_FIREBASE: 'SUBSCRIBE_GET_MESSAGES_TO_FIREBASE',
  UNSUBSCRIBE_GET_MESSAGES_FROM_FIREBASE:
'UNSUBSCRIBE_GET_MESSAGES_FROM_FIREBASE',
  UPDATE_MESSAGES: 'UPDATE_MESSAGES',
  POST_MESSAGE: 'POST_MESSAGE',
  POST_MESSAGE_SUCCESS: 'POST_MESSAGE_SUCCESS',
  POST_MESSAGE_ERROR: 'POST_MESSAGE_ERROR'
}
```

Let's then take a look at our `postMessageToServer` action:

```
export const postMessageToServer = (message) => (dispatch) => {
   dispatch({
     type: actionTypes.POST_MESSAGE,
     payload: {
       message,
       incoming: false
     }
   })
   postMessage(message)
     .then(() => {
       dispatch({
         type: actionTypes.POST_MESSAGE_SUCCESS
       })
     })
     .catch((error) => {
       dispatch({
         type: actionTypes.POST_MESSAGE_ERROR,
         error
       })
     });
```

```
    }
```

As you can see, our `postMessageToServer` is a function that returns a function with a dispatch argument. The first thing we do is dispatch a `POST_MESSAGE` action for introspection and logging. Then, we update Firebase, and when it's done, we dispatch the success or error function.

When subscribing to Firebase, we will do the following:

```
export const subscribeToGetMessagesFromServer = () => (dispatch) => {
     dispatch({
       type: actionTypes.SUBSCRIBE_GET_MESSAGES_TO_FIREBASE
     })
     const unsubscribeFn = getMessages((snapshot) => {
       dispatch({
         type: actionTypes.UPDATE_MESSAGES,
         payload: {
           messages: Object.values(snapshot.val()),
           unsubscribeFn
         }
       });
     })
 };
```

We will dispatch the `SUBSCRIBE_GET_MESSAGES_TO_FIREBASE` action, then set our updater function to be a new dispatch of `UPDATE_MESSAGES`. You can probably see here that we also pass `unsubscribeFn`, as a payload. This is done in order to be able to remove our listener if our component unmounts:

```
export const unSubscribeToGetMessagesFromServer = () =>
(dispatch, getState) => {
   if (getState().unsubscribeFn) {
     getState().unsubscribeFn();
     dispatch({
       type: actionTypes.UNSUBSCRIBE_GET_MESSAGES_FROM_FIREBASE
     })
   }
 }
```

Here, you can also see that we are using the `getState` function to get `unsubscribeFn` from our application state.

Now, let's look at our Reducer:

```
import { actionTypes } from '../actions';

 const initialState = {
```

```
  messages: []
}
export default function messagesReducer(state = initialState, action){
  switch (action.type) {
    case actionTypes.UPDATE_MESSAGES:
      return action.payload
    case actionTypes.UNSUBSCRIBE_GET_MESSAGES_FROM_FIREBASE:
      return {
        messages: state.messages
      }
    default:
      return state;
  }
}

export const getMessagesSelector = (state) => state.messages
```

When calling the `UPDATE_MESSAGES` action, Reducer replaces our state with new messages `array` along with `unsubscribeFn`.

When calling `UNSUBSCRIBE_GET_MESSAGES_FROM_FIREBASE`, we return the new state with only a messages array, removing `unsubscribeFn` from our application state.

Centralizing side effects

In this example, we've used thunks along with action creators, and although it's okay for small applications, it can be hard to manage for large applications. In that case, it's advisable to centralize side effects to one place, that is, create reusable and more generic thunks or use redux-saga instead of thunks.

Check out more information on **redux-saga** at `https://github.com/redux-saga/redux-saga`.

Mobx- a functional reactive Flux implementation

Although Redux has been widely used in React and React Native since 2015, a new library has become so much popular recently that it's worth mentioning and even using in our projects. It's called **MobX** and is based around *functional reactive programming paradigm*.

If you want to understand more about what this paradigm is I suggest that you check out the following gist:

`https://gist.github.com/staltz/868e7e9bc2a7b8c1f754`

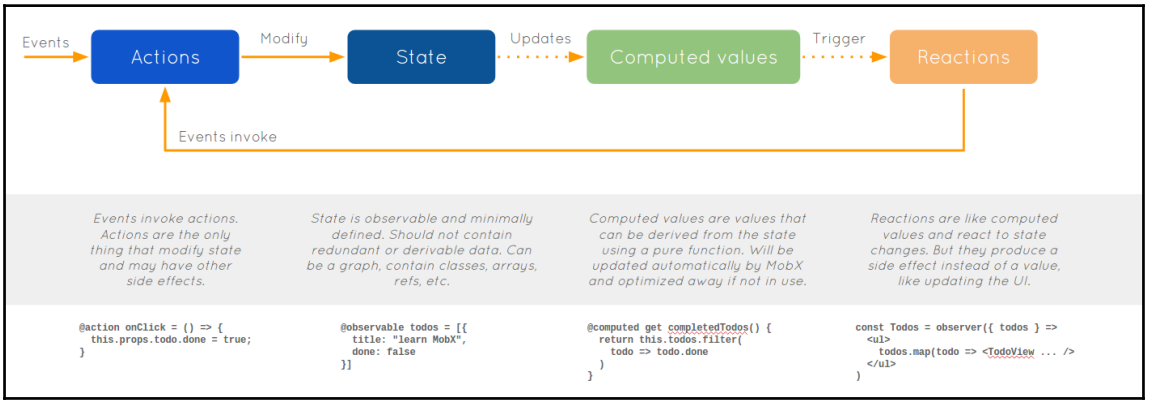
What is Mobx?

MobX is a library that simplifies state management by applying functional reactive programming. The philosophy around it is that anything that can be derived from an application state should be derived automatically. This includes UI, serialization, server and communication.

While in Redux you basically do everything manually, in MobX its done automatically by the library itself. Internally, MobX uses a *reactive virtual dependency state graph* that is only updated when strictly needed and is never stale.

This is really important for the React Native environment. Generally speaking, when your component is re-rendered, values go through the bridge. So, unnecessary re-renders can result in performance issues. In MobX, components are optimized to be re-rendered only when their actual state is updated.

Let's take a look at how MobX infrastructure looks like in the following chart taken from official docs:



Actions trigger state updates, which update computed values, and as a result, reactions are triggered.

Different from Redux, which has only one store, MobX can, and should, have several stores, each one responsible for its own logic. Usually actions, state, computed values, and side effects are grouped in the same place in that store.

Basic concepts

Let's overview basic concepts of MobX to understand the building blocks of this Flux implementation and the reason why it has become so popular lately. Here, we will explain only the basics, so it's advisable to look at the following official documentation for more information:

`https://mobx.js.org`

State

State is the data that drives your application. It can, and should, be organized in logical units and grouped together to create state trees grouped by domain logic in stores. State holds values that can be updated over time. As a result of this update, derivations are calculated automatically.

Derivations

Derivations in MobX can be of two types, **Computed Values** and **Reactions:**

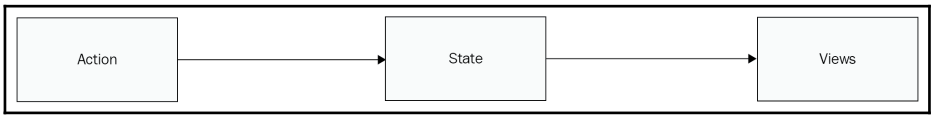
- Computed Values are values that can always be derived from a current observable state using pure functions
- Reactions are side effects that happen automatically if the state changes, for example, re-rendering a React component as a result of a state change is a reaction

Actions

An action is a code that changes the state. When using MobX in a strict mode (advisable), you cannot change the state outside of actions.

So, to summarize all these basic concepts, let's take a look at an excerpt from the official documentation at: `http://mobx.js.org`.

MobX supports a uni-directional data flow, where actions change the state, which in turn updates all affected Views:



All derivations are updated automatically and **atomically** when the state changes. As a result, it is never possible to observe intermediate values.
All derivations are updated synchronously, by default. This means that, for example, actions can safely inspect a computed value directly after altering the state.
Computed values are updated lazily. Any computed value that is not actively in use will not be updated until it is needed. If a View is no longer in use, it will be garbage collected automatically. All computed values should be pure. They are not supposed to change state.

Connecting MobX to our app

MobX widely uses `ES.next` decorators. So, first of all, let's add support for decorators to our React Native app. It's done by installing `babel-preset-react-native-stage-0`:

```
npm install babel-preset-react-native-stage-0 --save
```

Then, we add a decorator preset to `.babelrc`:

```
{
 "presets": ["react-native", "react-native-stage-0/decorator-support"]
 }
```

Now, we will be able to use decorators in our project. Don't forget to restart the packager by stopping and running the following command:

```
npm start reset-cache
```

First of all, we will set our MobX to strict mode inside our `app/index.js`, so we will follow the Flux architectural pattern. In strict mode, if we will change our state outside of actions, we will get an exception:

```
import { useStrict } from 'mobx';
useStrict(true);
```

Now, we will wrap our application with `Provider`, but this time from the `mobx-react` package:

```
<Provider messageStore={new MessageStore()}>
    <AppNavigator />
 </Provider>
```

As you can see, we will now create a new `MessageStore` instance. Our message store is imported like this:

```
import MessageStore from './store';
```

Now, it's time to construct our `MessageStore`.

It will encapsulate both our state and actions and deal with side effects:

```
import { observable, action, computed } from 'mobx';
 import { getMessages, postMessage } from './services/api';

 export default class Store {
 @observable
 messages = []

 @observable
 unsubscribeFn = null;

 @action
 postMessageToServer(message){
   postMessage(message)
 }

 @action
 updateMessages(snapshot){
   this.messages = Object.values(snapshot.val());
 }

 @action
 subscribeToGetMessagesFromServer(){
   this.unsubscribeFn = getMessages(this.updateMessages.bind(this))
 }

 @action
 unSubscribeToGetMessagesFromServer(){
   this.unsubscribeFn = this.unsubscribeFn();
 }
}
```

First, we define our state:

```
@observable
 messages = []

 @observable
 unsubscribeFn = null;
```

Decorating our state with `observable`, we will tell Redux to make this state reactive and to execute reactions once it's updated.

It is important that we should decorate all components that rely on MobX state with the `observer` decorator, so MobX could calculate updates efficiently.

Then, we will define our actions:

```
@action
 postMessageToServer(message){
   postMessage(message)
 }

 @action
 updateMessages(snapshot){
   this.messages = Object.values(snapshot.val());
 }

 @action
 subscribeToGetMessagesFromServer(){
   this.unsubscribeFn = getMessages(this.updateMessages.bind(this))
 }

 @action
 unSubscribeToGetMessagesFromServer(){
   this.unsubscribeFn = this.unsubscribeFn();
 }
```

The logic is the same as we've used in Redux, but with noticeably lesser boilerplate code.

Now, in order to connect our screen to MobX, we will need to do the following:

```
import { inject, observer } from 'mobx-react';
@inject("messageStore")
@observer
export default class ChatScreen extends React.Component {
```

Inject will add `messageStore` to your props, so you will have access to state and actions, and observer will create a reaction of re-rendering your component once the state keys that are used in it are changed.

As you can see, it has undoubtedly lesser boilerplate code than Redux. However, it's important to know both of these libraries since Redux is the de-facto standard, whereas MobX has only begun emerging recently. More and more people are switching to MobX, so I suggest that you try it, if you haven't already.

I strongly suggest that you read the official MobX documentation for more information on various edge cases in MobX: `http://mobx.js.org`

Summary

In this chapter, we've covered the Flux Architectural pattern and how it's different from the MVC pattern as well as what exact problems it's supposed to solve. Then, we dived deeper into Redux, a functional Flux implementation library. We overviewed its basic principles and concepts, and by altering our *whatsappClone* app to work with Redux, we've focused on several important concepts as well as how to manage state using Redux. We've used logging middleware and implemented async flow for our firebase connection, totally separating our state management to Redux. After getting an overview of Redux capabilities and making our app work, we looked at a new emerging state management library--MobX. After understanding the basic concepts of this library, we've changed our *whatsappClone* app to work with MobX instead of Redux. At this point, you gained solid knowledge of two state management systems you can use with React Native. One (Redux) involves much more boilerplate, however, it's structured around a functional programming paradigm and the second, (MobX) is a more functional reactive way of dealing with data. You can use whichever you prefer. Both are good, and both have a huge community that supports them.

In the next chapter, we will take a look at all those fancy things you always wanted to create with your app, such as getting geo location, saving data locally, using CameraRoll, and much more. Also, we will overview lots of native APIs you can use with React Native. While some will have only a brief overview, the most important ones will include a special explanation. For example, you will learn how to use user gestures to create Tinder-like draggable cards. So, get ready to dive deep into native APIs exposed by React Native to the JavaScript world.

9
Understanding Supported APIs and How to Use Them

Welcome to the ninth chapter. In this chapter, we will get an overview of supplied APIs in React Native framework. While we won't cover all of them, and new APIs get introduced with React Native on a regular basis, we will get familiar with the most important ones. Although, the list of APIs is up to date at the time of publishing this book, it's still important to check the official documentation before you start diving into these APIs. We will start this chapter by familiarizing ourselves with the concept of linking with react-native link which is used for connecting native API to our project since some APIs will require some interaction with native code. In the next chapter, we will see it more often when we will deal with community packages using native code. So, it's important to familiarize ourselves with this process before we dive into all APIs. Then, we will briefly go over basic APIs divided into categories by their use case: notification, informational, input-related, app-related, image-related, style-related APIs, and the ones that fall out of all these categories.

After getting familiar with these fairly simple APIs, we will dive deeper for widely used APIs, such as CameraRoll, GeoLocation, and AsyncStorage. At the end, we will focus on the most complex, but fun, PanResponder API, which will help us to capture user gestures and is essential for every app. In addition to overviewing its usage, we will create Tinder app card swipe behavior.

In this chapter, we won't cover APIs such as Animated or LayoutAnimation since we've already covered them in `Chapter 6`, *Animating React Native components*, but we will definitely use Animated for our card behavior, so if you are still unsure about how to use it, I suggest that you jump back to `Chapter 6`, *Animating React Native components*, where we discussed Animations in React Native and experiment with it. After all, the more you experiment, the better you learn.

So let's summarize what we will cover in this chapter:

- Getting familiar with the linking process of libraries or APIs with native code
- Getting familiar with a list of native APIs covered by React Native
- Learning how to work with CameraRoll, GeoLocation, ImageStore, and much more to interact with your phone's core APIs
- Learning how to use the PanResponder API to capture user gestures
- Learning how to use AsyncStorage as a local storage alternative

As we've seen up until now, React Native gives us lots of tools to write native code in JavaScript using best practices used on the web. In addition to it, there is a list of mobile APIs, which React Native wraps, so you can use it in JavaScript. Some of these APIs can be plugged and played and it's as simple as importing an API from React Native and then using it, but there are few that require native code to be linked manually.

Linking libraries and APIs with native code

In addition to APIs supplied by React Native, there are lots of community packages with native code that must be linked to your application before usage. There are several steps you should do when linking a package with API, and it differs from package to package and from API to API, so you should be familiar with the process before we dive deeper into APIs in this chapter and into packages in the next one.

First of all, it's important to understand that linking of libraries or APIs can be done only in an application generated by the `react-native init` command or by those made with Create React Native App and ejected. This is due to the fact that linking process registers native packages for iOS and Android. In `create-react-native-app`, since you don't have `ios` and `android` folder with native code, linking simply won't work. So in `create-react-native-app` we will run:

```
npm run eject
```

We do this before adding any libraries with native dependencies and linking them.

A more comprehensive guide for ejecting Create React Native App can be found at: `https://github.com/react-community/create-react-native-app/blob/master/EJECTING.md`

Auto linking

You won't find yourself doing automatic linking for APIs, but, for external packages, this will be a very common use case. The idea is that React Native gives you the a command to run in your terminal and will link all native dependencies for you, both for Android and IOS. This usually means adding the `xcodeproject` files to libraries in XCode, adding things to manifest on Android, and so on. The command to run for linking all native code dependencies automatically would be as follows:

```
react-native link
```

We will look at it again in the next chapter when we will talk about community packages with native dependencies.

Manual linking

Manual linking is more relevant for some of the IOS APIs. In Android, linking can be different from API to API, so there is no general rule of thumb on how to link in Android, but no worries, we will cover linking-specific APIs in a bit.

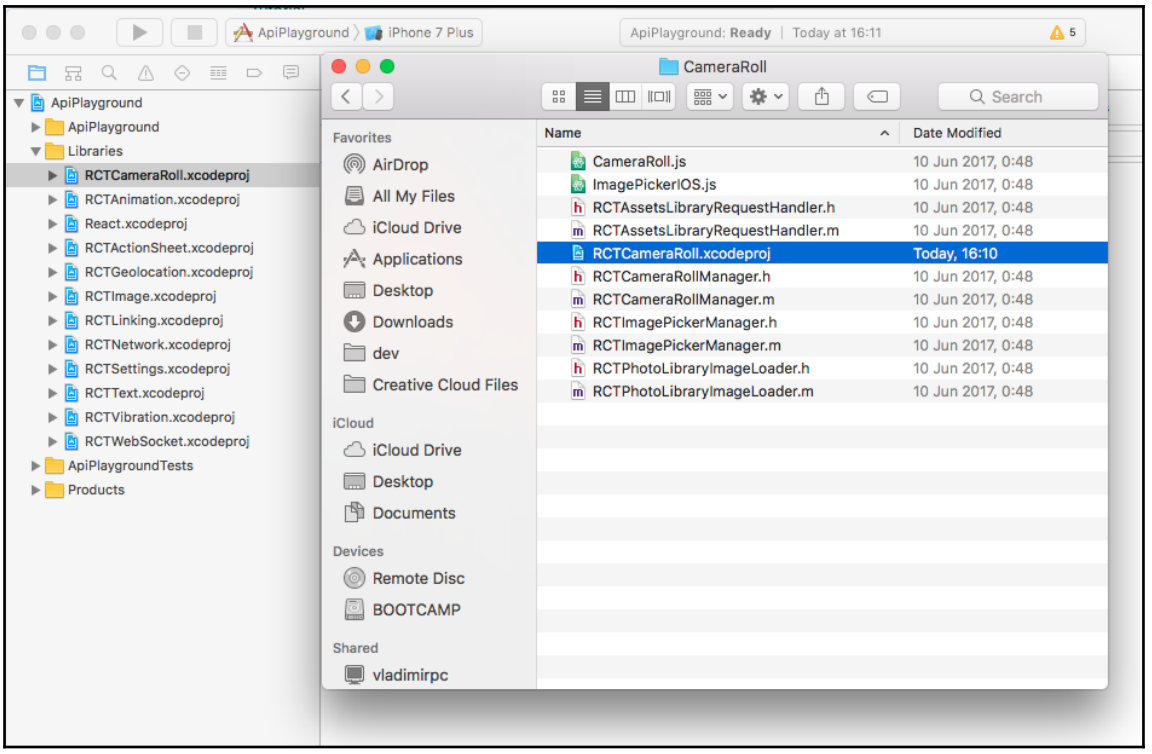
On IOS, linking has the following two phases mentioned in the next two sections.

Linking libraries

When an API or a library has a native code, it usually has a `.xcodeproj` file. In order to add it, you will need to locate the `xcodeproj` file of API or library you wish to add and add it into the `Libraries` folder group on XCode.

For example, let's take a look at the `CameraRoll` API.

We will go into `node_modules/react-native/Libraries/CameraRoll` and find the `RCTCameraRoll.xcodeproj` file. Then, we will simply drag it into our Libraries group in XCode, as shown in the following screenshot:

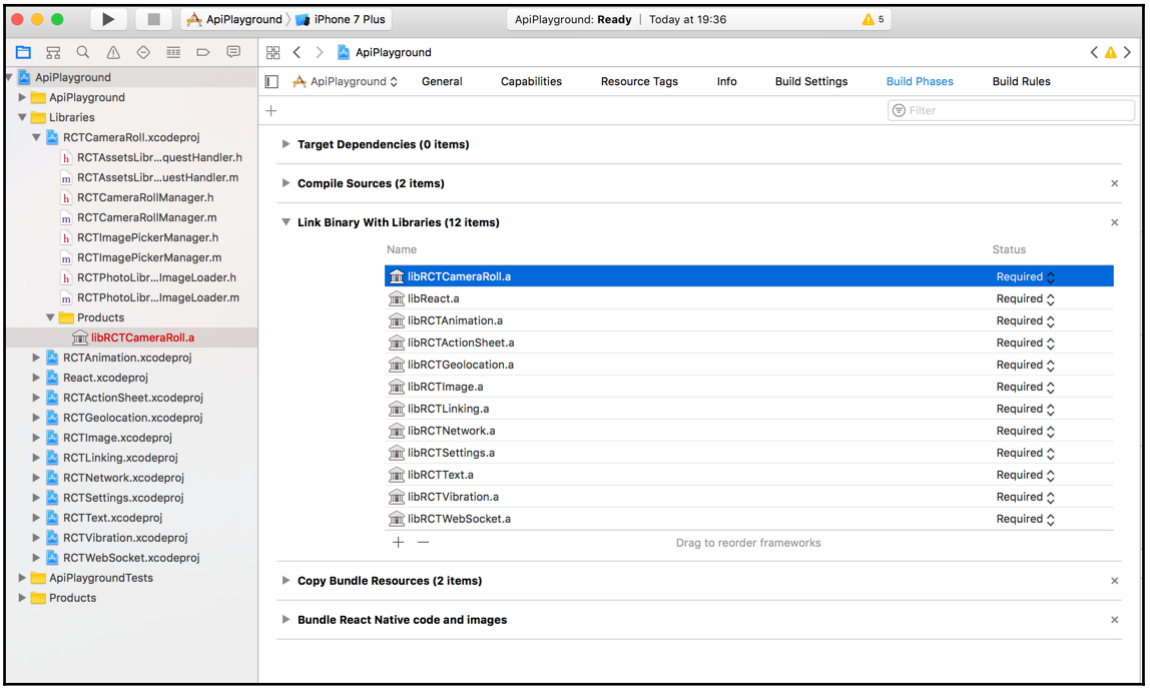


Configuring build phases

Another step we should do to manually link our library is to configure builds for the static library we just added to our library group:

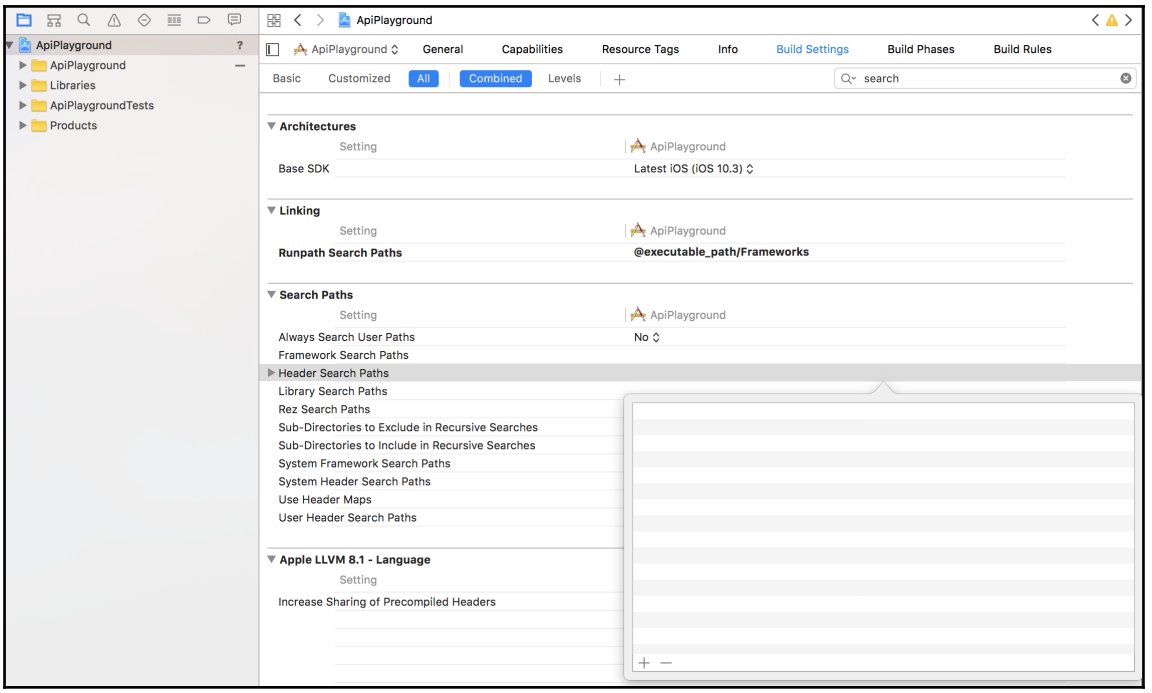
1. Click on the project name--**ApiPlayground**, in our case--and select Build Phases from the menu at the top. Then, expand **Link Binary with Libraries**.
2. Next, you will need to expand the library (`RCTCameraRoll.xcodeproj`, in our case) and select **Product**.

3. Then, drag and drop it into the **Link Binary with Libraries** list. Take a look at the following screenshot and note that `RCTCameraRoll.xcodeproj` and `Products` are expanded, and **Link Binary with Libraries** contains `libRCTCameraRoll.a`:



Some libraries require additional steps, such as calling specific methods from **AppDelegate**, the entry point of our IOS app. For that, they usually require headers to be imported into our application. While it may vary in each API, the general rule of thumb is to navigate to your **Build Settings**, select **All** instead of **Basic**, and search for **Header Search Paths**.

Then, you will need to add your `$(SRCROOT)/node_modules/react-native/Libraries/YourLibrary` library. Take a look at the following screenshot to see where you should add it:



Ensure that you check what documentation has to say about it before linking since APIs can change with future XCode or React Native versions: `https://facebook.github.io/react-native/docs/linking-libraries-ios.html`

Getting familiar with a list of native APIs covered by React Native

Now it's time to take an overview of React Native APIs and what we have out-of-the-box. Note that if you are using *Create react* app, in addition to React Native APIs, you have lots of APIs from Expo.

While there is a huge list of these APIs that is being added to on a regular basis, Expo is a community project, and even though it gives you access to lots of things, it limits you to having an Expo account and using Expo dev tools, so Expo won't be covered in this book, but you can read about it at `https://docs.expo.io/`.

For the sake of simplicity, I divided React Native APIs into several behavioral groups instead of dividing them by cross-platform, IOS, and Android APIs. There is no meaning to these groups in React Native and is used in the book only to group them logically.

Note that an API that has IOS in its name, for obvious reasons, won't work on Android, and the same goes for APIs that have Android in its name. Other cases will be specified in each API separately.

Notification APIs

This API group deals with all APIs that provide some kind of notification to their users. It can be visual, sound, or vibration.

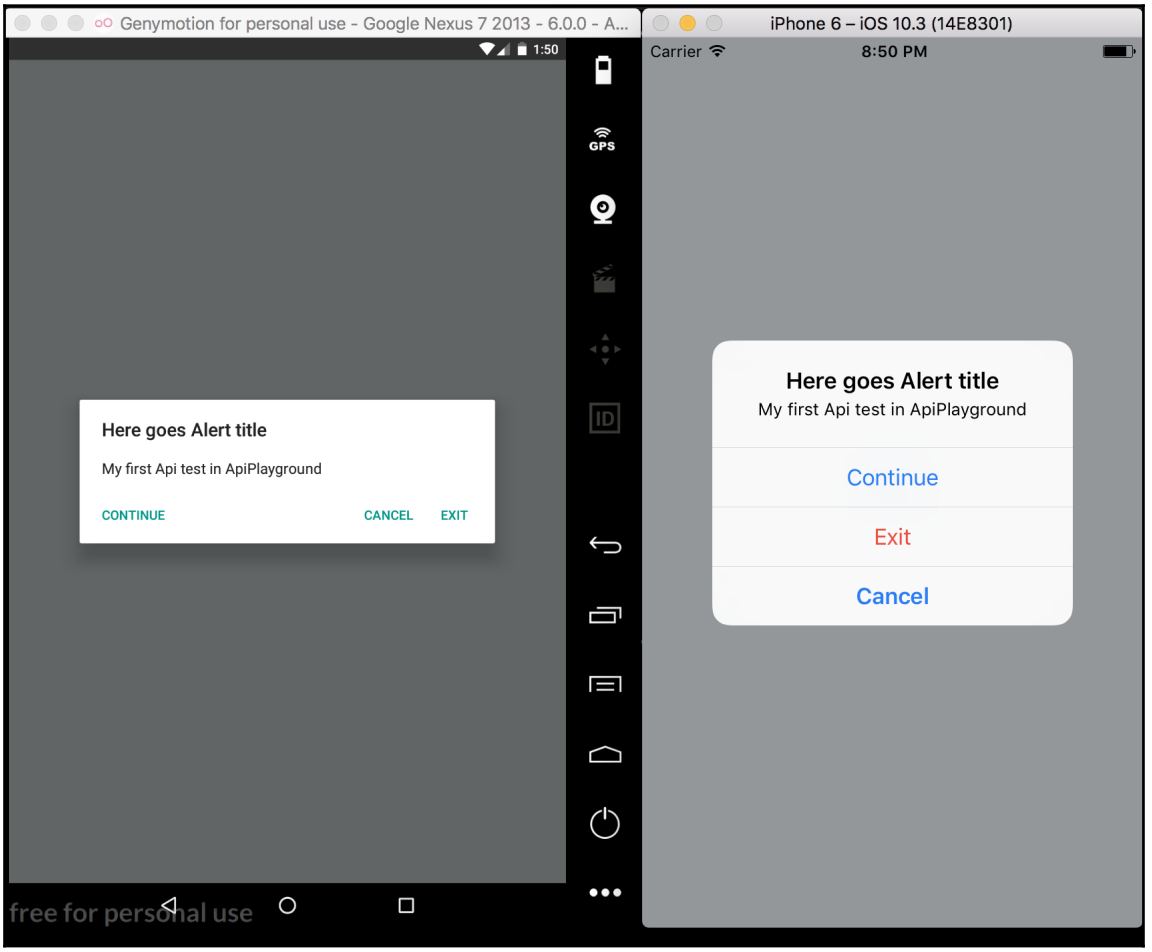
Alert

This is a cross-platform API, meaning that it will work on both Android and IOS. This is how it can be used after you import the `Alert` function from `react-native`:

```
alert(){
     Alert.alert(
       'Here goes Alert title',
       'My first Api test in ApiPlayground',
       [
         {text: 'Continue',
           onPress: () => console.log('Ask me later pressed')},
         {text: 'Cancel',
           onPress: () => console.log('Cancel Pressed'), style:
           'cancel'},
         {text: 'Exit',
           onPress: () =>
             Alert.alert('Are you sure?', '', [
               { text: 'Ok'},
               { text: 'Cancel', style: 'cancel'}]),
             style: 'destructive'
         }
       ],
       { cancelable: true }
```

```
    )
  }

  render(){
    return (
      <View>
        <Button  onPress={this.alert} />
      </View>
    )
  }
```

This is how it will look on Android and on IOS:



The `Alert` function gets a title, message, and an array of buttons as an arguement. Every button is described by an object with text, `onPress` callback, and `style`. On IOS, there are three style types: **default**, **cancel**, and **destructive.** On Android, there is a concept of positive, negative, and neutral buttons. They are not specified by a style object as in IOS, but they are defined by how many buttons you supply. One button will always be positive; two buttons will be positive and negative; and three means positive, negative, and neutral. Additionally, Android has an optional parameter--cancelable--as seen in the example: `{ cancelable: true }` which when set to true means that Alert can be canceled by taping out-of-the-box. You also can pass an `onDismiss` callback on Android to trigger anything you want when Alert is dismissed by tapping out of one of its boundaries.

You can read further about Alert in official docs found at `https://facebook.github.io/react-native/docs/alert.html`.

AlertIOS

This Alert can be used only on IOS and does not provide any additional functionality than Alert API, so it's advisable to use Alert for cross-platform compatibility.

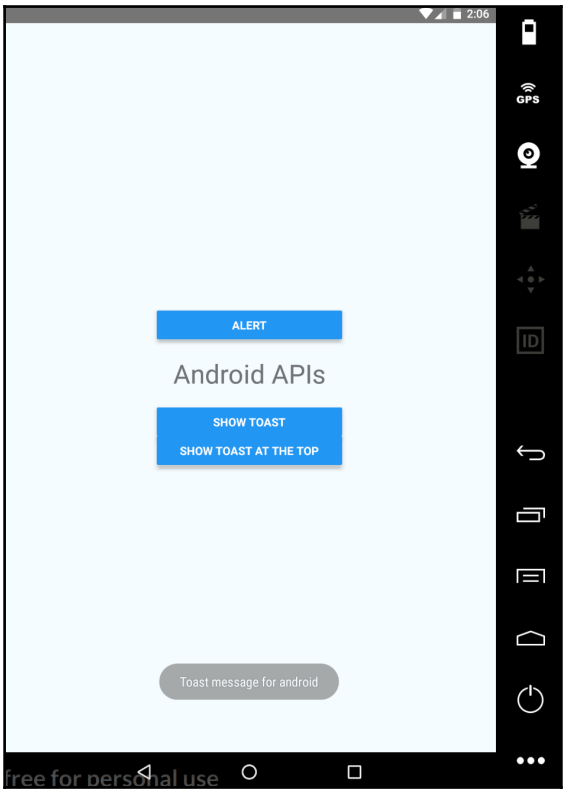
ToastAndroid

ToastAndroid is a very simple API for Android. It simply shows toast message. It can be used like this:

```
ToastAndroid.show('Toast message for android', ToastAndroid.LONG)
```

While the first argument is message text and the second is duration, which can be SHORT or LONG.

The result will look like this:



Toast can also be shown in the center of the screen, at the top, or at the bottom using the following function:

```
ToastAndroid.showWithGravity('Toast message for android',
ToastAndroid.LONG, ToastAndroid.TOP)
```

As you can probably guess, passing `CENTER` or `BOTTOM` will result in showing a toast at the bottom or in the center of the screen.

ActionSheetIOS

IOS action sheets are a commonly used option in IOS apps to provide multiple selection options to the user. **ActionSheetIOS** is used to create IOS action sheets similar to this one:

Its code looks like this:

```
<Button  onPress={() => {
     ActionSheetIOS.showActionSheetWithOptions({
       options: ['one', 'two', 'three', 'four', 'cancel', 'exit'],
       cancelButtonIndex: 4,
       destructiveButtonIndex: 5,
       title: 'Show case Action sheet',
       message: 'This is a simple action sheet show case'
       }, (args) => console.log(args))
 }} />
```

As you can see, `showActionSheetWithOptions` gets a configuration object, which is pretty descriptive. `options` is an `array` of the buttons, and `cancelButtonIndex` and `destructiveButtonIndex` are used to style buttons with provided index. `title` and `message` are for title and message of this action sheet.

You can read a bit more about `ActionSheetIOS` API at `https://facebook.github.io/react-native/docs/actionsheetios.html`.

Vibration

This API is probably the simplest to use in both iOS and Android; it will look like this:

```
import { Vibration } from 'react-native';
// ....
Vibration.vibrate([0, 500, 100, 200]);
Vibration.cancel();
```

The following is an explanation of its two functions:

- **vibrate**: Receives two arguments: a **pattern** and **repeat boolean**. A pattern is an array that represents vibration intervals. For Android, it looks like this: `[waitMillisecondsNumber, vibrateMillisecondsNumber, waitMillisecondsNumber, vibrateMillisecondsNumber]`, so, for example, passing [0, 500, 100, 200] will mean vibrate for 0.5 seconds, then wait for 0.1 second and then vibrate for 0.2 seconds. Then, repeat if the second argument is supplied as true, default to false. In our case there won't be repeat, because a second argument was not provided.

For IOS, the case is a bit different, however the syntax stays the same. You cannot specify how long your device should vibrate, so you can only pass an array of wait intervals. So, like the same pattern before, for IOS, it will mean the following: vibrate, then wait for 0.1 second, then vibrate, wait 0.5 seconds, vibrate, wait 0.2 seconds.

- **cancel**: This cancels vibration.

This API won't work in a simulator, but only on a physical device.

PushNotificationIOS

PushNotifications are part of lots of apps, however, React Native does not support PushNotifications for Android; it supports only IOS.

This is a complex API with tedious installation and linking. Moreover, it's not cross-platform compatible, so even though you can use it by referring to official docs at `https://facebook.github.io/react-native/docs/pushnotificationios.html`. I strongly advise that you use the community package `react-native-push-notification`, which is cross-platform and can be found at `https://github.com/zo0r/react-native-push-notification`.

Information APIs

This API group is used very often for obtaining information on your currently running app.

AccessibilityInfo

This API is used to request information on whether a screen reader is available. It has the following three methods:

- `fetch`: Used to get information on whether a screen reader is available; returns a `Promise` that results to a boolean
- `addEventListener`: Can be hooked to a change event and is used to notify when the screen reader state has changed
- `removeEventListener`: Removes a previously registered event listener

Official docs are available at `https://facebook.github.io/react-native/docs/accessibilityinfo.html`.

AppState

Similar to *AccessibilityInfo* that has `adEventListener` and `removeEventListener`, this API has only these two methods. An event listener can also be hooked to change event and is used to notify when the application state has changed. There are three application states:

- **active**: Indicates that the user is using the app
- **background**: Indicates that the user is in another app or on the Home screen
- **inactive**: Application is in transition between active state to background state or there is an incoming call while the user is in app.

Check out the official docs at `https://facebook.github.io/react-native/docs/appstate.html`.

NetInfo

This API is used to query network information from a simple online/offline status to more complex ones on Android.

For simple usage of querying, if our app is connected, we can simply do this:

```
componentDidMount(){
     NetInfo.isConnected.fetch().then(isConnected => {
       console.log(`App is ${isConnected ? 'connected': 'offline'}`);
     });
  }
```

We simply log in to to the console, regardless of whether our app is connected or offline. You can read more in the official docs at `https://facebook.github.io/react-native/docs/netinfo.html`.

PixelRatio

PixelRatio is used to get device pixel density. This can be used to retrieve larger images for devices with higher pixel density. While it's not used as often as the rest of APIs, it's important to know that there is such an API. Additional information on that API can be found at `https://facebook.github.io/react-native/docs/pixelratio.html`.

Dimensions

This API is used all the time to retrieve screen width and height for layout-related calculations. You usually use it by importing `Dimensions` from `react-native` and then getting `width` and `height`:

```
const { width, height } = Dimensions.get('window');
```

This is usually used for styling or animation calculations.

It's important to understand that dimensions can change due to device rotation, so I advise you to use width and height retrieved from `Dimensions` as inner styles and not use it inside a style sheet, because it will be cached and can result in unexpected behavior.

Platform

While it's not mentioned in the API docs at all, sometimes you need to query your application on whether you are on Android or on iOS. This is done using `Platform.OS === 'ios'`

or `Platform.OS === 'android'`.

Settings

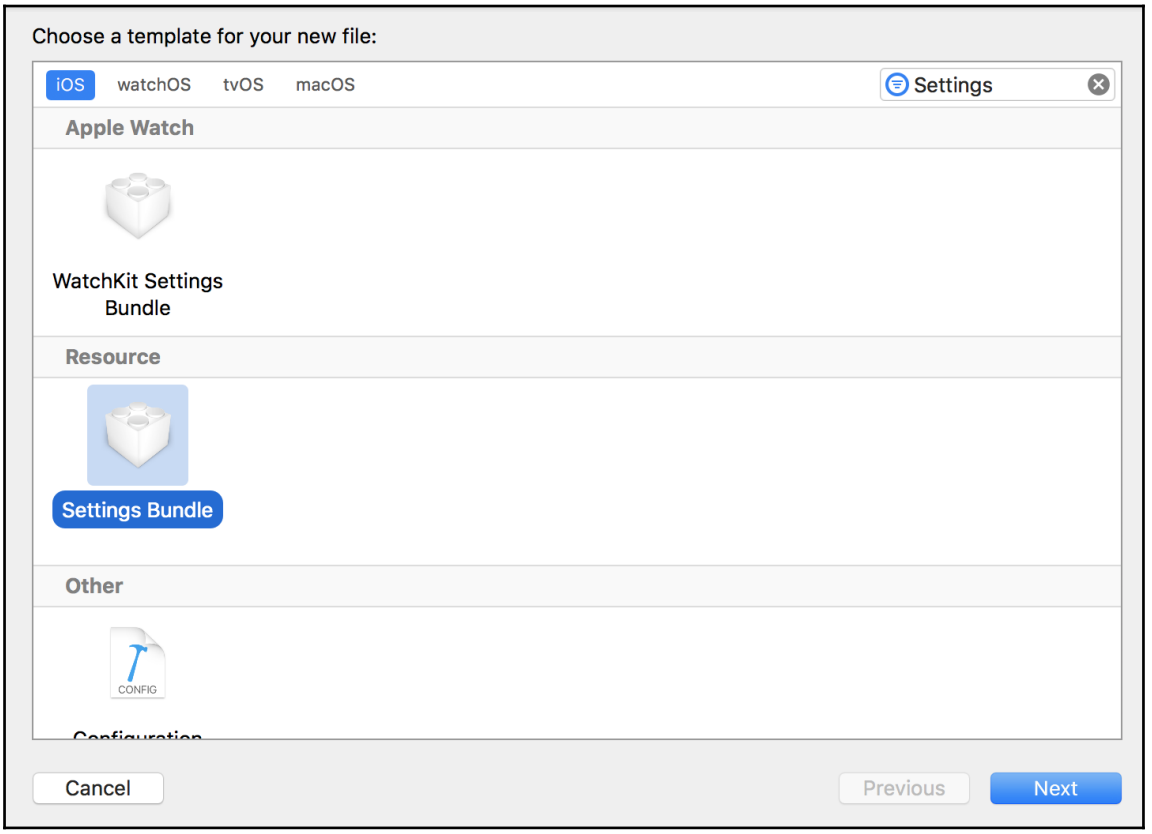
Settings is an API to store and retrieve keys for your application settings. It's currently implemented only for IOS and it has set and get methods as well as `watchKeys` and `clearWatch` methods for watching keys change and clearing `watch`. It's similar to the event listener. In order to start using **Settings**, you need to set up a settings bundle for your application by following the Apple developer docs here:

`https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/UserDefaults/Preferences/Preferences.html`

In a nutshell, you need to follow these steps:

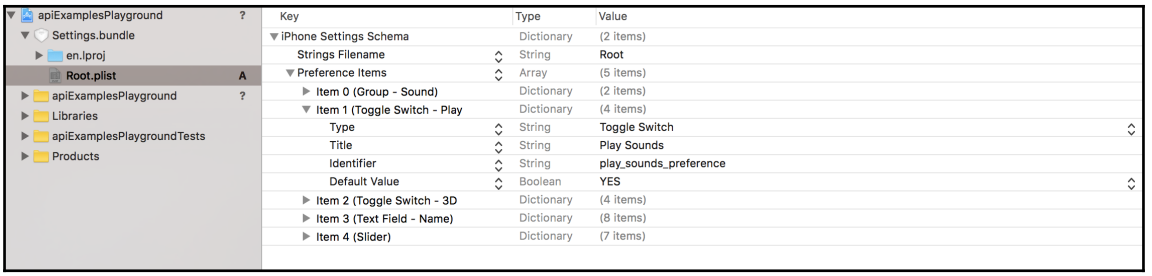
1. Open your `.xcodeproj` file in XCode

2. Go to **File** | **New** | **New File**. You will get a window where you can choose what file template you want:



Like in previous screenshot selected, you should choose **Settings Bundle** and name it `Settings.bundle`.

Now you can add values to `Settings.bundle` | **Preference Items**:



In our React Native code we will use now:

```
Settings.set({ play_sounds_preference: 'NO' })
```

You also can retrieve them by using:

```
Settings.get('play_sounds_preference')
```

Input related

This category is related to all the input taken from the user. It will explain about getting values from Clipboard, dealing with various Date and Time pickers for Android, and handling keyboard.

Clipboard

This API allows access to Clipboard using `getString` or `setString` methods. These methods return a Promise, so its simple use case will be like this:

```
<Button  onPress={
  async () =>
    await Clipboard.setString('get this text into clipboard')}/>
<Button  onPress={
    async () => {
      const result = await Clipboard.getString()
      console.log(result);
    }}
/>
```

When clicking the first button, we copy text to the Clipboard and we paste it by clicking on the **Next** button.

Keyboard

The **Keyboard** API is used to deal with keyboard. You may have encountered this API in previous examples using `Keyboard.dismiss()`, which dismisses Keyboard programmatically. Other useful methods in the Keyboard API are `addListener`, `removeListener`, and `removeAllListeners(eventName)`.

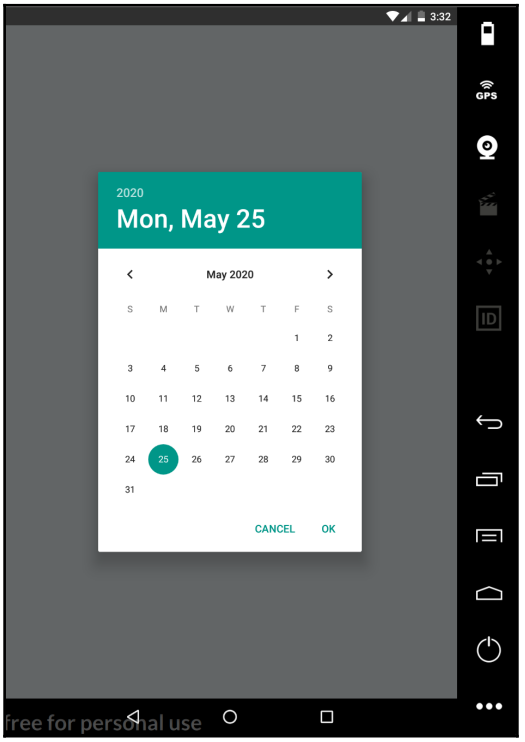
Listeners can be added to the following events:

- `keyboardWillShow`

- `keyboardDidShow`
- `keyboardWillHide`
- `keyboardDidHide`
- `keyboardWillChangeFrame`
- `keyboardDidChangeFrame`

DatePickerAndroid

This is basically an Android component, which is more an API rather than a component:

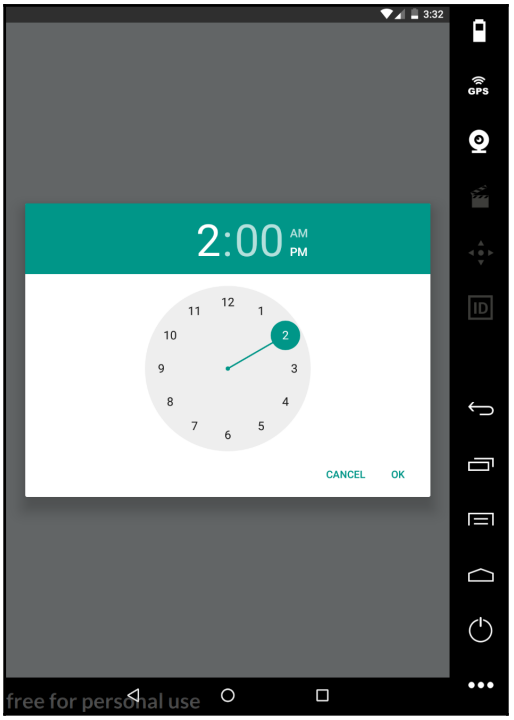


Instead of passing `<DatePickerAndroid />` as we would have expected with component, we will use an open function with various props. We won't cover all of these props, but everything is documented in the official docs, so make sure that you look into them:

`https://facebook.github.io/react-native/docs/datepickerandroid.html`

TimePickerAndroid

Time picker is an Android-specific component that can be used in your application to show specific native control for picking time. It can be used only via API and does not have a specific component for that. By using it, you will have the following component showing on the screen:



You can read more about it here:

`https://facebook.github.io/react-native/docs/timepickerandroid.html`

App related

In this section, we will mention APIs related to your App behavior. We won't dive deeply into each API, but it's important to know which APIs are available and what they do.

AppRegistry is a JavaScript entry point for running your app. When creating your application, you can see the `AppRegistry.registerComponent` function, which basically tells a native system to load a bundle and run your app. There are very limited use cases when you will need to use AppRegistry beyond its basic functionality (for example, running headless app without UI), so it won't be covered in this book. However, you can check all available methods on AppRegistry by taking a look at these docs:

`https://facebook.github.io/react-native/docs/appregistry.html`

InteractionManager

Sometimes, if we have heavy calculations that are done during animations or interactions, it can freeze our app. In that case, we can use `InteractionManager` to schedule task after interactions and animations are finished using:

```
InteractionManager.runAfterInteractions(() => {
 // some task
 });
```

InteractionManager also gives us the ability to notify that the interaction has started. In order to do so, we can use the `createInteractionHandle` method, and later when the interaction is finished, we will call `clearInteractionHandle`.

You can refer to official docs for more information at `https://facebook.github.io/react-native/docs/interactionmanager.html`.

Linking

The **Linking** API is used for deep linking to your application as well as opening external applications. For example, for opening WhatsApp from your application, you can use:

```
Linking.openURL('whatsapp://app')
```

Linking API and **PushNotificationIOS** are fairly complex to set up, so it's important that you check official docs-- `https://facebook.github.io/react-native/docs/linking.html`.

Image related

React Native provides an API to store, pick, and edit images. In this section, we will briefly cover these APIs.

ImageEditor

ImageEditor has only one method: the `cropImage(uri, cropData, successCallback, failureCallback)` method. It receives an image URI, and if it's a remote image, it will download it automatically. If it cannot download it, `failureCallback` will be executed. After downloading, ImageEditor will try to crop an image based on the `cropData` object passed to the method. The `cropData` keys are as follows:

- `offset: { x: number, y: number }`: The top-left corner of the cropped image, specified in the original image's coordinate space
- `size: { width: number, height: number }`: Size (dimensions) of the cropped image
- `displaySize: { width: number, height: number }`: (Optional) size to which you want to scale the cropped image
- `resizeMode: 'contain/cover/stretch'` (optional) resizing mode to use when scaling the image

When ImageEditor succeeds in cropping the image, it will store it automatically into ImageStore and execute `successCallback`, passing the URI that points to ImageStore.

You can, and should, use this URI to supply to Image. You don't need to retrieve it from ImageStore.

ImageStore

ImageStore is used to store images in-memory. For example, in ImageEditor, the cropped image is stored automatically in ImageStore. It's important to remove images we don't need from ImageStore so that we won't overload the memory. The most common use of ImageStore is image removal for images that are not needed with `removeImageForTag(uri)` and checking whether an image exists in the store using `hasImageForTag(uri, callback)`.

There are also Base64-related methods that can be checked in original docs. **Base64** is an encoding algorithm for binary data:

`https://facebook.github.io/react-native/docs/imagestore.html`

ImagePickerIOS

ImagePickerIOS is a specific iOS component that is used to retrieve images from your file system and pick one among them. It's supported currently only on iOS, and it's strongly advised that you use the community package instead. It's backed by the `react-community`, the creators of react-navigation and it's cross-platform:

`https://github.com/react-community/react-native-image-picker`

Style related

While we've discussed styles a while ago, it's important to know where exactly we can find all supported properties in a React Native style object. This can be found by looking at the following three APIs:

- **Style sheet**:
  `https://facebook.github.io/react-native/docs/stylesheet.html`
  Style sheet in addition to providing a level of caching for our style objects supplies several properties helper functions that can help with styling purposes
- **LayoutProps**: All style properties dealing with layout can be found at `https://facebook.github.io/react-native/docs/layout-props.html`
- **ShadowProps**: All style properties dealing with shadow can be found at `https://facebook.github.io/react-native/docs/shadow-props.html`

Other various APIs

The following are APIs that didn't fall into any of the preceeding categories.

BackHandler

On Android, in additon to interacting with the touch screen, the user can interact with the hardware back button. In order to implement this functionality, you will need to manually register it. It's done using:

```
BackHandler.addEventListener('hardwareBackPress', function() { });
```

In addition to adding a listener and, of course, removing it with `removeEventListener`, there is an `exitApp` method that can be used to exit from our app completely.

PermissionsAndroid

Since the introduction of Android M's new permission model (SDK 23 and later), previously automatically granted permissions now need to be requested from a user manually.

There are three methods in Permissions Android:

- `check`: To check permissions
- `request`: To request permissions from a user
- `requestMultiple`: To request multiple permissions

You can further read about various methods in PermissionsAndroid here:

`https://facebook.github.io/react-native/docs/permissionsandroid.html`

There is also a cross-platform package, compatible with the recent React Native version and with both IOS and Android permissions models, so make sure that you check it:

`https://github.com/yonahforst/react-native-permissions`

AdSupportIOS

While this API is available, it was supposed to be deprecated a long time ago. Basically, it gives you access to an advertising provider only on IOS, but as a result it can interfer with the AppStore submission process. In the next chapter, we will take a look at other ad solutions when using the `react-native-admob` package.

Share

For quite a while, sharing was not implemented as part of the React Native API; however, it has been added in the 0.39 version and now is available through a pretty simple usage.

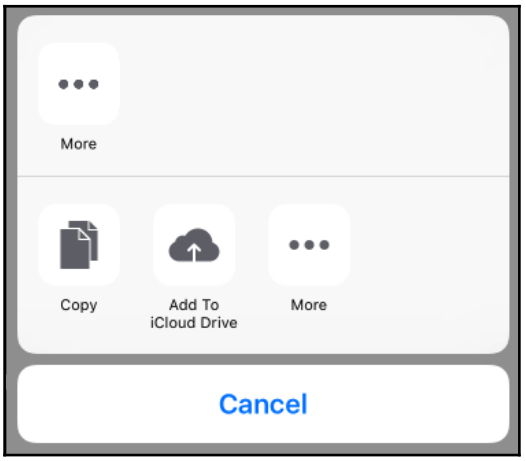
Let's create a simple share button:

```
<Button title="share" onPress={this.share} />
```

And now will call `Share.share` function with some content:

```
share(){
  Share.share({
    message: 'Sharing my awesome api',
    title: 'Api Playground'
  })
}
```

`share` function can receive a title, message and, on IOS, URI. It will look like this:



This API is mostly used for simple sharing, and for more advanced sharing on Facebook or on Twitter, dedicated packages are used with more options. We will review these packages in the next chapter.

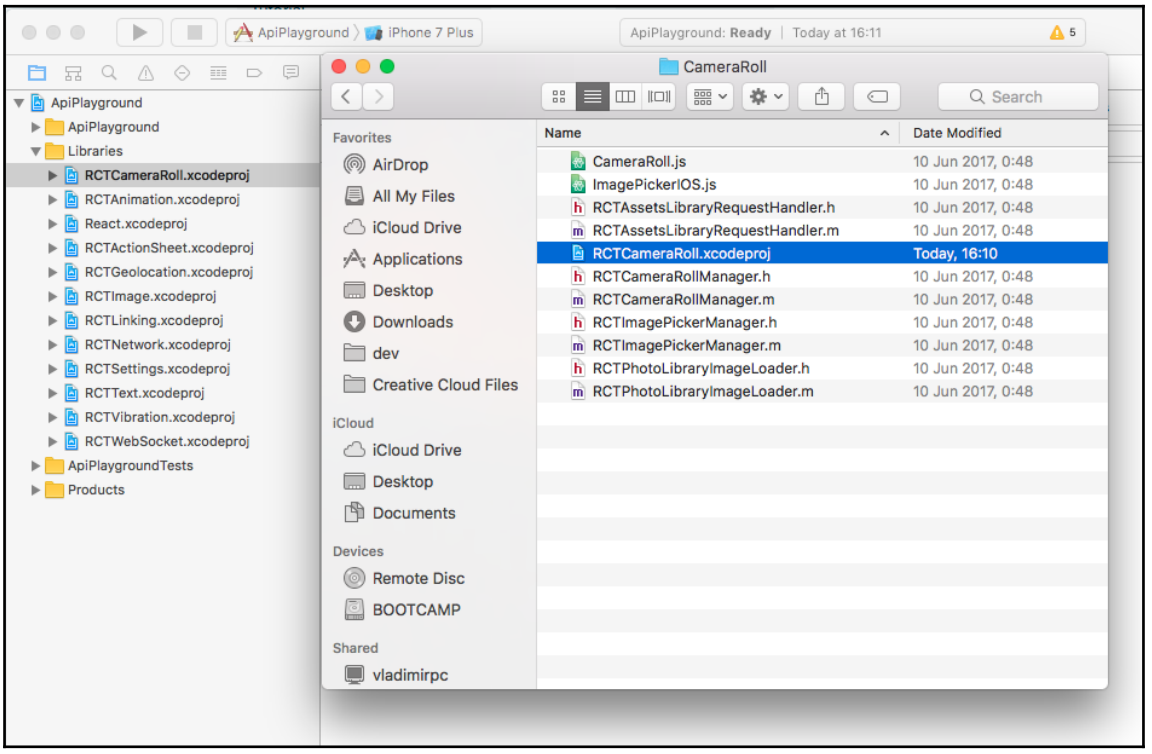
For more info, check the following link: `https://facebook.github.io/react-native/docs/share.html`

Retrieving and saving photos with CameraRoll API

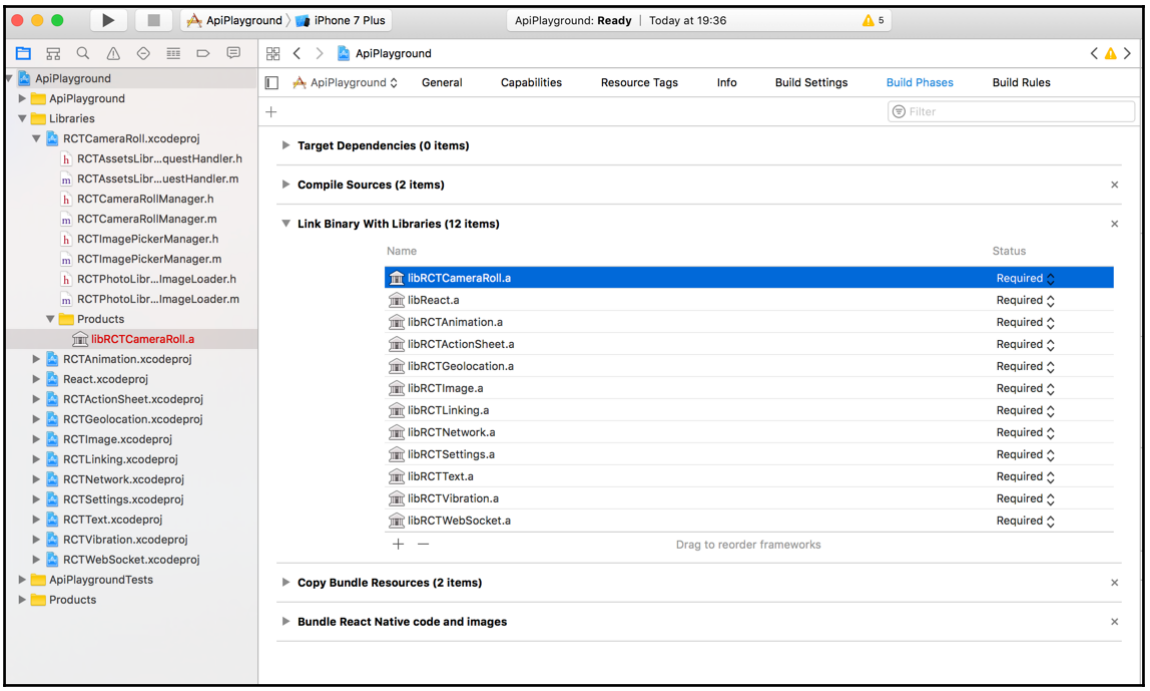
Okay, so we've looked at the list of React Native APIs, and now it's time to get to more complicated, but cool APIs. We will begin it with tackling the CameraRoll API. This API is used to get access to local photo gallery/camera roll, to obtain photos from your device.

It's important to know that you should properly "link", as described in the beginning of the chapter, your CameraRoll prior to using it, because it uses native modules. Fortunately, we've already linked CameraRoll, but for a brief overview, let's recall what we've done:

We brought `RCTCameraRoll.xcodeproj` from `node_modules/react-native/Libraries/CameraRoll` into our Libraries group in XCode:



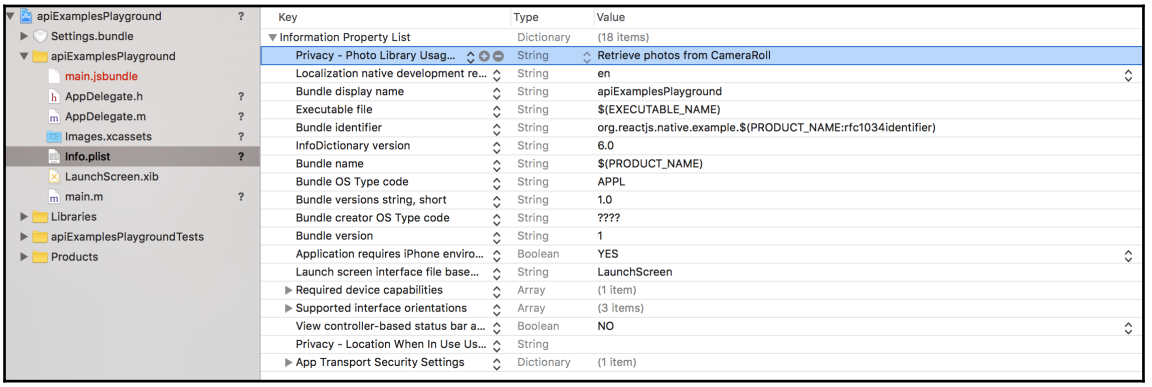
We configured our build phases by bringing `node_modules/react-native/Libraries/CameraRoll/Products/libRTCameraRoll.a` into **Build Phases** | **Link Binary with Libraries**:



In addition to that, if we develop our app for iOS 10 and later, we need to add some permissions to our app. Go into **iOS/Info.plist** and add the following:

```
<key>NSPhotoLibraryUsageDescription</key>
<string>Any string describing your usage</string>
```

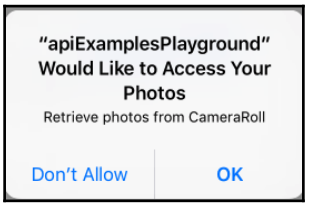
In XCode it will look like this:



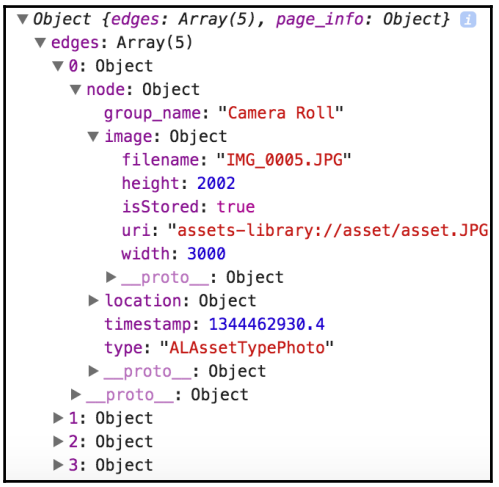
Now, we can use our CameraRoll:

```
getPhotos = () => {
  CameraRoll.getPhotos({
    first: 20,
    assetType: 'All'
  })
  .then(r => console.log(r))
}
```

When calling `getPhotos` for the first time you will get a permissions dialog:



After confirming, this is what you will get in the console:



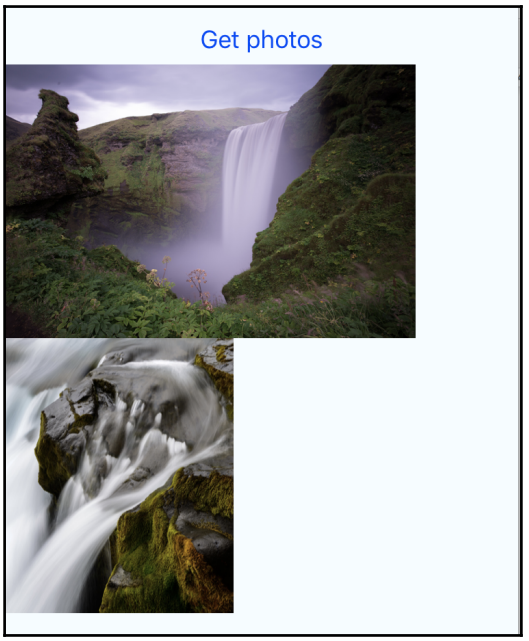
We need to update our photos in a component state instead of logging using

`setState({ photos: r.edges });`.

Then, we will add them to a view:

```
<FlatList
  data={this.state.photos}
  renderItem={({
    item: {
      node: {
        image: {
          height,
          width,
          uri
        }
      }
    }
  }) =>
    <Image
      style={{
        width: width * .1,
        height: height *.1
      }}
      source={{ uri }}/>
}/>
```

This will be our result:



There are lots of various parameters by which we can get our photos: types, categories, albums, and so on. All of them can be seen here:

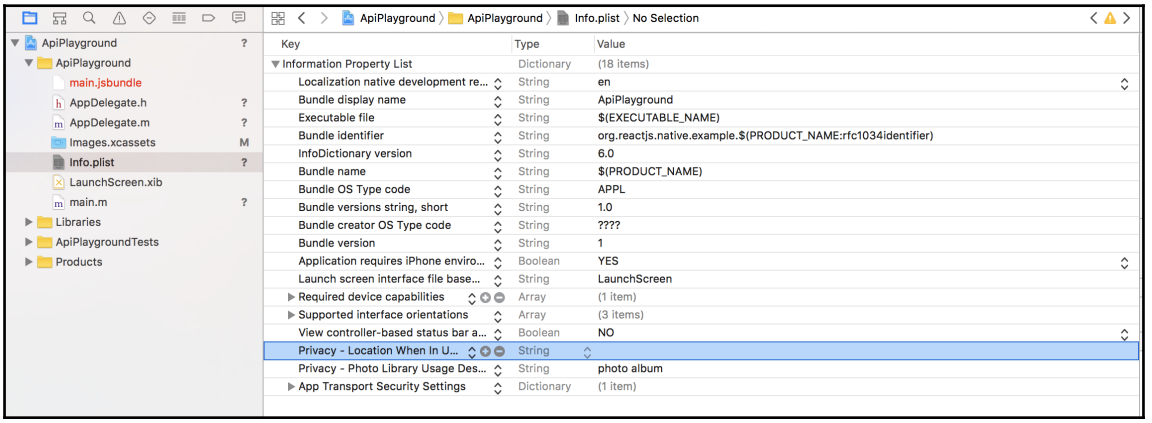
`https://facebook.github.io/react-native/docs/cameraroll.html`

Getting your exact location with GeoLocation API

There are several platform-specific settings you need to do prior to using GeoLocation API in your app.

IOS

When your application is installed using `react-native init`, GeoLocation will be enabled by default. If it's not, you need to set the `NSLocationWhenInUseUsageDescription` key into `Info.plist`. It will look like this:



Sometimes, you also want GeoLocation to be enabled when running your app in the background. If you want this functionality, you need to add another key--`NSLocationAlwaysUsageDescription`.

Note that when pasting the key in XCode, it will be automatically changed to a descriptive name. If you open `Info.plist` inside your editor though, you will see the following:

```
<key>NSLocationWhenInUseUsageDescription</key>

<key>NSLocationAlwaysUsageDescription</key>
```

Android

In Android, you also need to add some permission-related settings to `android/main/src/AndroidManifest.xml`. You should add the following:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"
/>.
```

Usage

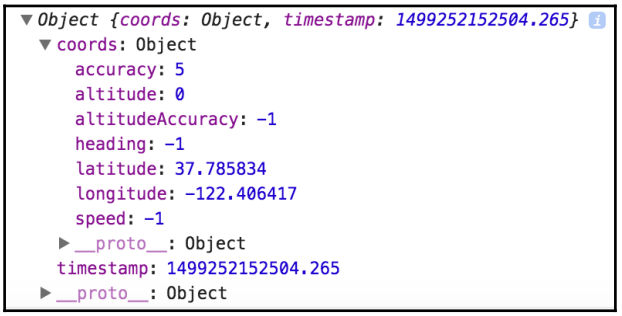
After all these steps are done, you can use GeoLocation on both Android and IOS; however, you don't need to import it. It exists on a global navigator object and basically is a polyfill of web GeoLocation. As such, it can be accessed using `navigator.geolocation` similar to how it's used on the web.

The web GeoLocation spec can be found here:

`https://developer.mozilla.org/en-US/docs/Web/API/Geolocation`

Basically, GeoLocation has four important methods: `getCurrentPosition`, `watchPosition`, `clearWatch`, and `stopObserving`.

- `getCurrentPosition`: Gets three arguments: `successCallback`, `errorCallback`, and `options` objects.
- `successCallback` will receive an object like this:



- `errorCallback` will receive an error message and options object can pass three parameters.
- `enableHighAccuracy (boolean)` will make GeoLocation retrieval slower, however, will retrieve a result with a higher accuracy.
- `timeout` defines how much time you should wait until an error is thrown.
- `maximumAge` is used for caching. It defines maximum age of location data stored on your device.
- `watchPosition` is the same as `getCurrentPosition`, however it's not triggered once, but whenever the user moves. It has an additional option that can be supplied--`distanceFilter`, which tells how many meters the user has to move, so the `watchPosition` will be triggered.

- `clearWatch` is the same as `clearInterval` function in JavaScript, but for `watchPosition`. It will clear a specific watch, that you've saved similar to how you would have done with `clearInterval`.
- `stopObserving` clears all `watchPosition` functions and basically tells GeoLocation API to stop observing any changes.

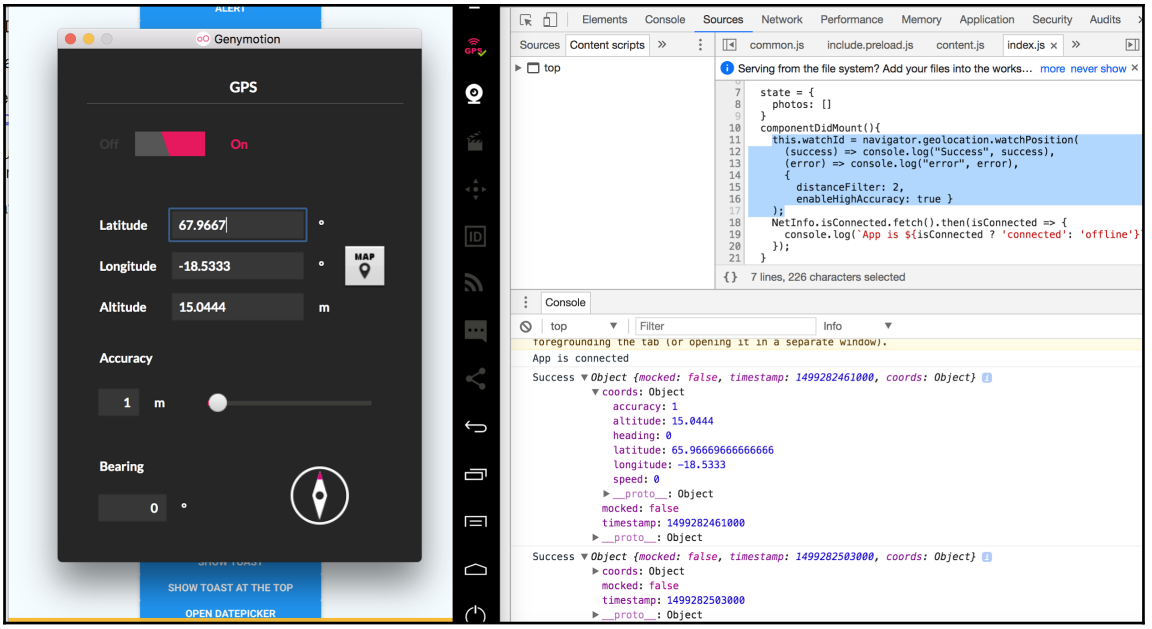
For example, we want to watch our position and simulate that on a device. This time we will do it on Android. I'm using the Genymotion emulator for this.

The code will look like this:

```
this.watchId = navigator.geolocation.watchPosition(
  (success) => console.log("Success", success),
  (error) => console.log("error", error),
  {
    distanceFilter: 2,
    enableHighAccuracy: true }
);
```

Note that I'm saving `watchId` to use it later in `clearWatch` whenever I want to unsubscribe from location changes.

Then, let's take a look at the following screenshot from Genymotion Android emulator:



In the preceding screenshot, I've opened my app and see in the console a location with latitude **65.9667**.

As shown in the preceding screenshot, I've opened the **GPS** tab in my Genymotion toolbox on the right and changed latitude from 65.96 to 67.96. The distance is much faster than 2 meters, so I get notified in the console about the change in the GeoLocation. If, for example, I change `distanceFilter` to 2000000 (2k km), I won't get notified for changing the latitude back to 65.96.

Learning about persistence with AsyncStorage API

There is not so much to write about `AsyncStorage`, the fact that it's used instead of `LocalStorage` on the web. You save authentication data or any other small data, you may probably need later on.

It's important to understand that `AsyncStorage` is unencrypted, so if you want to store passwords, make sure that you encrypt them before that with packages like `crypto-js` or `bcrypt`. Ideally your password should be encrypted on server.

Although it's possible to save data in `LocalStorage`, it's discouraged and advised to use `AsyncStorage` instead.

On the iOS side, `AsyncStorage` uses native code and stores key-value pairs in a serialized dictionary, when dealing with short values. Large values, though, are stored in files. On Android, it uses RocksDB or SQLite based on what's available.

AsyncStorage is promise-based, so every `AsyncStorage` method returns a Promise, hence usually, we will use it with the await keyword or using the Promise API.

The basic usage is using `setItem(keyName, value)`, `getItem(keyName)`, and `removeItem(keyName)` methods, which set, get, and remove keys from `AsyncStorage`. Similar to `localstorage`, the value should be a string, so in case you want to store JSON object, you should stringify it.

Another awesome method is `mergeItem(keyName, valueToMerge)`. It gives you the ability to automatically merge an existing key with a passed value, assuming it's also a stringified JSON, of course. This can be very helpful and is commonly used.

Let's take a look at the following example:

```
const dummyObj = {
    title: 'test',
    id: '1234'
  }
  const dummyObj2 = {
    description: 'this is a test object description'
  }

  let result = await AsyncStorage.setItem('OBJ1234',
    JSON.stringify(dummyObj)
  );

  result = await AsyncStorage.mergeItem('OBJ1234',
    JSON.stringify(dummyObj2)
  );

  result = await AsyncStorage.getItem('OBJ1234');
```

We define two dummy objects and add one of them into `AsyncStorage` with a key name `OBJ1234`. Then, we pass the other object to `AsyncStorage.mergetItem` and expect the two objects to be merged. Then, we get our `OBJ1234` key from `AsyncStorage`. This is what we will get in the console if we log our result:

```
{"title":"test","id":"1234","description":"this is a test object
description"}
```

There are several other helper functions on `AsyncStorage` that deal with multi-object merges, retrieval of all keys, and more. All of them can be found in official docs.:

```
https://facebook.github.io/react-native/docs/asyncstorage.html
```

Responding to user gestures with PanResponder

Till now, we've tackled almost every aspect of React Native API, excluding one of the most complex, but really important aspects: User gestures.

In application development, seamlessly responding to touches and swiping things around is very important, and understanding how this can be done is probably something you waited for from the beginning of this book. There is a lot to explain, but I wanted to explain it using a real-world example: the *Tinder* application.

Tinder is a location-based social search mobile app that facilitates communication between mutually interested users, allowing matched users to chat. The app is most commonly used as a dating app, but has branched out to provide more services, making it more of a general social media application. Matching is based on Facebook and Spotify profiles. For more information please check the following link: `https://en.wikipedia.org/wiki/Tinder_(app)`

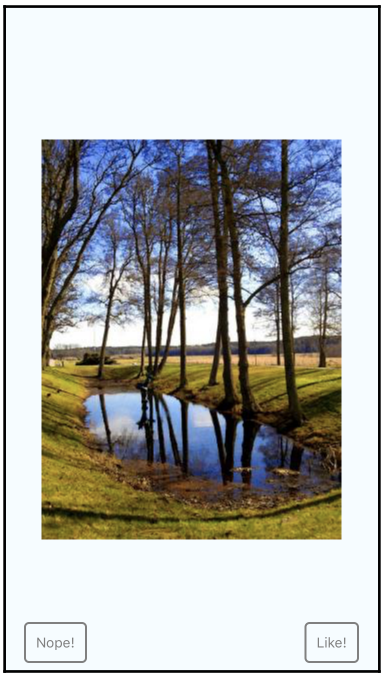
In Tinder, the app shows you photos of singles based on your preferences and around your GeoLocation. When you swipe left, the user profile card is tilted to the left and out of the screen with a nice animation. Same for the right, but swiping right will lead to tilting to the right and out of the screen. After any swipe, you switch to the next person card. Whenever you swipe left, this means you don't like the person, and swiping right means you do like the person.

In our example, we won't create a whole Tinder functionality, but we will implement the following behavior:

- Tinder swipe behavior
- When swiping, we will see the tilt and opacity changes
- When swiping to the corner, we will see a no or yes indicator appear at the bottom of the screen
- When we swipe for a specific threshold, our card will switch to the next one.

So, let's start. First of all, let's create our app with `create-react-native-app tinderAppClone`.

Then, we will define a basic card layout like this:



First of all, let's define our People array as a hardcoded array of random images and add it to our `App.js` file:

```
const profiles = [
 'https://lorempixel.com/300/400/',
 'https://lorempixel.com/300/400',
 'https://lorempixel.com/300/400/',
 'https://lorempixel.com/300/400/',
 'https://lorempixel.com/300/400/',
]
```

We use `lorempixel` service, which gets us random images. In a real app, you'll, of course, retrieve these images from API.

Let's add the first picture to our state, and later on, when we move from one card to another, we will update the state:

```
state = {
  person: profiles[0]
}
```

We will animate our card later on, so let's prepare everything for this. Our render function will look like this:

```
<View style={styles.container}>
   <Animated.Image source={{
       uri: this.state.person
     }}
     style={[styles.card, cardTransformStyle ]}>
   </Animated.Image>

   <Animated.View style={[styles.nope, nopeTransformStyle]}>
      <Text style={styles.nopeText}>Nope!</Text>
   </Animated.View>

   <Animated.View style={[styles.like, LikeTransformStyle]}>
     <Text style={styles.likeText}>Like!</Text>
   </Animated.View>
</View>
```

You can see here that in addition to a regular style object, I added `cardTransformStyle`, `nopeTransformStyle`, and `likeTransformStyle`.

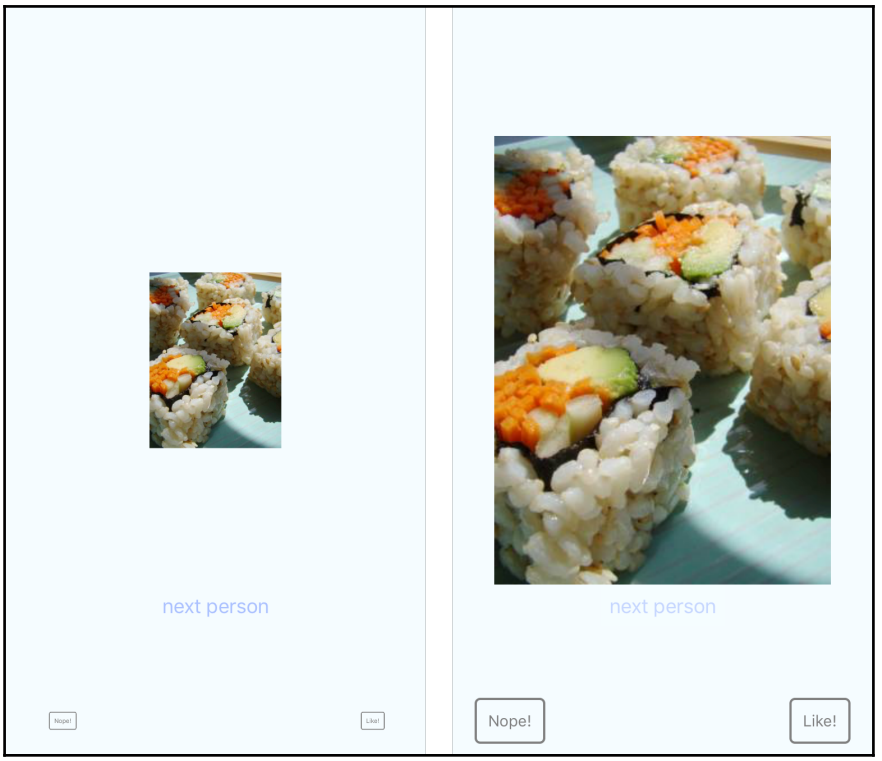
These style objects will later on be used to transform our styles when gestures are triggerred.

For now, let's put hardcoded values inside without transform, opacity 1, and no rotations:

```
let cardTransformStyle = {
       transform: [
         { translateX: 0 },
         { translateY: 0 },
         { rotate: '0deg' },
         { scale: 1 }
       ], opacity: 1
     };

     let nopeTransformStyle = {
       transform: [
         {scale: 1}
       ],
       opacity: 1
     }

     let likeTransformStyle = {
       transform: [{
         scale: 1
       }],
       opacity: 1
```

```
        }
```

Now, before tackling user gestures, let's add a simple scale-in animation when our new person enters and `nextPerson` switching behavior. The resulting animation will look like this:



In order to do that, let's introduce a new `AnimatedXY` state variable: scale. Our state will look like this:

```
state = {
    person: profiles[0],
    scale: new Animated.Value(0.5)
  }
```

The reason we use 0.5 is because we want subtle scale-up behavior for initial loading, opposing more "aggressive" scale-up animation when switching profiles.

Now let's create a scaling-up animation function and trigger it on `componentDidMount`:

```
componentDidMount(){
```

```
      this.scaleAnimationTrigger()
    }
  scaleAnimationTrigger(){
      Animated.spring(
        this.state.scale,
        { toValue: 1, friction: 7 }
      ).start();
  }
```

The next phase will be to create a function to switch between profiles:

```
  switchToNextProfile(){
    this.resetState();
    const currentPersonId = profiles.indexOf(this.state.person);
    let nextPersonId = currentPersonId + 1;
    this.setState({
      person: profiles[nextPersonId > profiles.length - 1 ? 0 : nextPersonId]
    });
    this.scaleAnimationTrigger();
  }

  resetState() {
    this.state.enter.setValue(0);
  }
```

Here, we first of all reset our scale state to 0, then calculate what will be the next profile, and at the end trigger the same animation we've triggered in `componentDidMount`. There is only one thing left: In our render function, change all scale: 1 to scale: `this.state.scale`.

So we've created a basic animation and layout for our Tinder app with random images taken from lorempixel. As you can see in the previous screenshots, I've added the next person button for debugging purposes, but now it's time to remove the button and implement it with gesture animations and a gesture responder system.

The Gesture responder system

The Gesture responder system is an internal system of React Native that manages the lifecycle of gestures in the system. It's crucial to know inside your application what the user intends to do, touch/scroll/swipe or something else. For this and more, this system was implemented, and you can use it to understand what your user is doing in your app.

Usually, this system will be used together with Animated, because as a rule of thumb, you want to show you user visual feedback on their interaction with an app. Also, it's advised to never create irreversible gestures. For example, if the user starts dragging to the left, they want to be able to cancel their gesture by simply removing their finger.

For example, in our case, if a user hit some threshold of swipe length, then we would switch to a new profile; however, if the user simply started to drag a card to the left and then changed their mind and released the finger, the card will return to its place using the `Animation.decay` function.

There is specific set of props you can pass to your View to make any View in your application respond to gestures. All these props are described in official docs; however, in most cases, you will use PanResponder:

`https://facebook.github.io/react-native/docs/gesture-responder-system.html`

PanResponder

PanResponder is a wrapper for a gesture responder system. In addition to that, it uses the `InteractionManager` handle to block long running JS calls that interrupt active gestures, making your gestures work well without any interference.

In order to use PanResponder, we should import it and create it. We will do it with the following:

```
componentWillMount() {
   this._panResponder = PanResponder.create(configurationObject)
}
```

Now, let's take a look at our configuration object. First of all, we have to pass two functions that return true:

```
onMoveShouldSetResponderCapture: () => true,
onMoveShouldSetPanResponderCapture: () => true,
```

By passing these two functions, we tell IOS that we allow movements and will be tracking them.

The next function is `onPanResponderGrant`, which runs when we start touching. Here, we will set our initial values both for `x`, `y`, and `offset`:

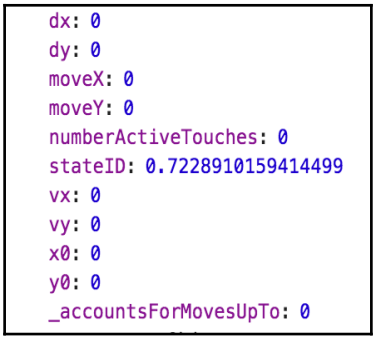
```
onPanResponderGrant: (e, gestureState) => {
  this.state.pan.setOffset({
    x: this.state.pan.x._value,
    y: this.state.pan.y._value
  });
  this.state.pan.setValue({x: 0, y: 0});
},
```

As you can see, we use `this.state.pan` in our code, but we haven't set it up yet. We will set it up with person and scale states:

```
pan: new Animated.ValueXY()
```

Here we are using `Animated.ValueXY` because we will use both `x` and `y` for our gestures.

Note that here, we get an event argument and a `gestureState` argument, which looks like this:



Next, we set up our actual move handler: `onPanResponderMove(event, gestureState)`

Here, we will extract `dx` and `dy` from `gestureState` and set our `pan.x` and `pan.y` with new values:

```
onPanResponderMove: (event, { dx, dy }) => {
  this.state.pan.x.setValue(dx)
  this.state.pan.y.setValue(dy)
}
```

Lastly, we will implement the functionality for moving to the next profile when hitting a threshold in the `onPanResponderRelease` function:

```
onPanResponderRelease: (e, {vx, vy}) => {
  this.state.pan.flattenOffset();
  if (Math.abs(this.state.pan.x._value) > SWIPE_THRESHOLD) {
    Animated.decay(this.state.pan, {
      velocity: {x: vx, y: vy},
      deceleration: 0.98
    }).start(() => this.switchToNextProfile())
   } else {
     Animated.spring(this.state.pan, {
       toValue: {x: 0, y: 0},
```

```
          friction: 4
        }).start()
      }
    }
```

Here, we check the swipe threshold. It's 120 in our case, and if it's bigger, we decay our pan values and then switch to the next profile. If not, then we spring back to the initial location of the card.

Combining it with Animated

Now, it's time to assign pan values to actual styles using Animated interpolation, but before we do that, let's beautify our `onPanResponderMove` function to:

```
onPanResponderMove: Animated.event([
  null, {dx: this.state.pan.x, dy: this.state.pan.y},
]
```

Using `Animated.event`, let's us assign `dx` and `dy` to our Animated pan value more easily.

Now interpolate `pan.x` values both for opacity and for rotation.

For rotation we will write the following interpolation:

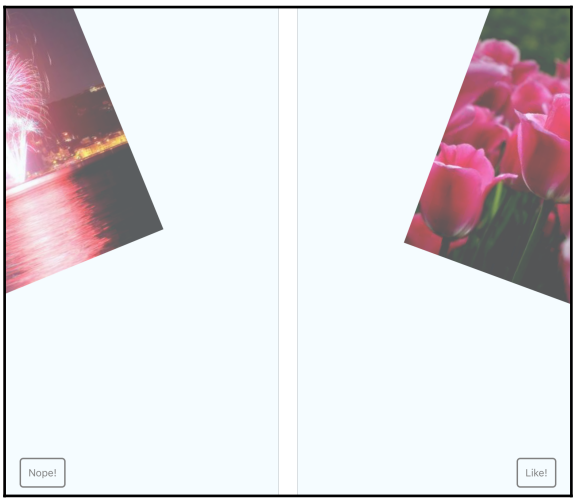
```
 const rotate = pan.x.interpolate({
   inputRange: [-250, 0, 250], outputRange: ["-30deg", "0deg", "30deg"]
 });
```

What we are saying here is that when our card moves left, tilt it toward -30 degrees, and if it moves right, tilt it to 30 degrees.

For opacity, we will use these interpolations:

```
const opacity = pan.x.interpolate({
  inputRange: [-250, 0, 250], outputRange: [0.5, 1, 0.5]})

const likeOpacity = pan.x.interpolate({
  inputRange: [0, 150], outputRange: [0, 1]});

const nopeOpacity = pan.x.interpolate({
  inputRange: [-150, 0], outputRange: [1, 0]});
```

Here, we fade out the card when dragging it to the left and fadeIn **Nope** text. When dragging a card to the right, `fadeout` the card and `fadeIn` the **Like** button.



The Final touches will be to add **Like!** and **Nope!** button scale:

```
const likeScale =
  pan.x.interpolate({inputRange: [0, 150], outputRange: [0.5, 1],
  extrapolate: 'clamp'});

const nopeScale =
  pan.x.interpolate({inputRange: [-150, 0], outputRange: [1, 0.5],
  extrapolate: 'clamp'});
```

Then, let's update our styles:

```
let cardTransformStyle = {
   transform: [
     { translateX },
     { translateY },
     { rotate },
     { scale }
   ], opacity
 };

 let nopeTransformStyle = {
   transform: [
     {scale: nopeScale }
   ],
   opacity: nopeOpacity
```

```
    }

    let likeTransformStyle = {
      transform: [{
        scale: likeScale
      }],
      opacity: likeOpacity
    }
```

In the end, we get the full experience of Tinder app swipe cards as we've tried to accomplish.

PanResponder is a really flexible API, and by using it together with Animated, you can achieve amazing things. You can read more about PanResponder in official docs here:

`https://facebook.github.io/react-native/docs/panresponder.html`

Summary

In this chapter, we covered various React Native APIs. We started by familiarizing ourselves with the process of linking libraries with native code and continued with getting to know a list of all React Native APIs. We divided them into groups: notification, information, input-related, app-related, and image-related APIs. Then, we focused on how to work with CameraRoll and image gallery, retrieve our position using GeoLocation, and store our data in AsyncStorage for persistency. Finally, we discussed two very important React Native APIs--Gesture Responder System and PanResponder. In addition to learning it's bytes and bits, we saw how it is fairly easy to create swipe cards similar to the *Tinder* application.

Now after we've almost finished our journey through React Native concepts, it's important to understand that together with React Native, there is the whole npm packages ecosystem that is brought in. There are lots of community packages, one of which we've seen briefly in the beginning--react-navigation. In the next chapter, we'll go through the most significant community packages you should be familiar with and will give an overview of the use cases of each one of them.

We will dive deeper into react-navigation and will finish with learning techniques of connecting your React Native app to native code or even writing native API wrappers yourself. So, get ready for a trip into the wild world of React Native npm packages, navigation, and native code.

10
Working with External Modules in React Native

Welcome to the tenth chapter. In this chapter, we will get to know how to get the best out of the npm ecosystem in order to install external packages to enhance our React Native apps. We will cover a list of the best community packages available out there at the moment of writing this book, and we will dive deeper into some of these packages, for example, `react-navigation`, which we used earlier in the book. We will learn more about how to use react-navigation while structuring our apps and will dive deeper into how it can be integrated with state management libraries, such as **Redux** or **MobX**. Then, we will cover a list of visual packages that will tremendously improve our apps. We will see how to use icons, easier animations, and even full-styled visual component kits from **Shoutem** and **Native Base** or `react-native-elements`. We will take a look at social provider packages, such as Facebook and Twitter. We will use one of them in the next chapter for authentication and data retrieval for a Twitter client app clone. Next, we will move on to the list of packages with additional APIs that are not supplied by React Native, but which are widely used in lots of apps.

In the last chapter, we learned how to link external packages with native code; in this chapter, we will learn how these packages can be created in the first place. At the end of the chapter, we will look at some basic Objective-C and Java code. We will deal with really basic examples since I don't assume that you know Objective-C and Swift for iOS, or Java and Kotlin for Android; however, it's important to understand that bridging between native worlds can be done pretty easily. At the end, we will learn that React Native can be integrated inside the existing native apps and vice versa.

So, let's summarize. Here's what we will learn in this chapter:

- Using `react-navigation` navigators and integrating them with Redux or MobX
- Getting to know lots of community packages that will make your app outstanding
- Getting familiar with writing your own native modules
- Integrating your React Native applications with the existing native app

Diving deeper into react-navigation

In the earlier chapters, we used the `react-navigation` package for several demo apps we created; however, we touched on it only briefly. In this section, we will dive deeper into `react-navigation` and not only understand how it should be used with Redux or MobX, but also how your app routing should be structured.

Navigators explained

While mobile navigation and the web seem similar, they are quite different from each other. Sometimes we want back functionality and sometimes we don't, but it always exists on the web. Basically, on the web, we always have a history stack, and we push or pop from this stack, sometimes replacing values and so on. In mobile, that's not the case; sometimes we have tabs. We don't have back functionality for tabs, and we even have a drawer-like menu, which is also a kind of navigation that is conceptually different. In `react-navigation`, we see three types of navigators:

- **StackNavigator**: This is the one you've seen in the *whatsappClone* app we've been creating so far. `StackNavigator` is, as the name suggests, a stack-like navigator. It means that every screen is laid on top of another screen.

  `StackNavigator` is defined in the following way:

  ```
  StackNavigator(RouteConfigs, StackNavigatorConfig)
  ```

  We saw that `RouteConfig` looks like the following:

  ```
  {
    screenName: { screen: screenReactComponent }
    screenName2: { screen: screen2ReactComponent }
  }
  ```

`StackNavigatorConfig` can receive lots of props, including custom headers, different display modes, and much more.

- `TabNavigator`: As the name suggests, this navigator is used for inserting tabs in our application. By default, they will be rendered at the bottom of the screen on iOS and at the top on Android. Even though the default usage of `TabNavigator` is for tabs, it's commonly used for any type of navigation that doesn't need stack functionality. So for example, if you have a splash screen and then authentication, you can consider putting both of them in `TabNavigator` and pass `tabBarVisible` in screen navigation props. The API for `TabNavigator` is pretty much similar to `StackNavigator`; however configuration props are a bit different. Generally speaking, if you want to create `TabNavigator`, you write it as follows:

```
TabNavigator(RouteConfigs, TabNavigatorConfig)
```

- `DrawerNavigator`: This is used for only one purpose-- setting up drawer navigation. `DrawerNavigator` is usually used at the top level because if it is not, it will be rendered inside another navigation, which can lead to weird results. `DrawerNavigator` is basically in charge of rendering the appearing side menu or the main view. The API is also similar:

```
DrawerNavigator(RouteConfigs, DrawerNavigatorConfig)
```

All navigators' configuration are passed in a similar way. First of all, you define your navigators. They eventually return a React component, so it's totally possible to nest navigators; in fact, it's done all the time. We will see an example in a bit. Second, in our screen component, we use the static `navigationOptions` property, which the navigator is using for configuring a particular screen. Third, the React component that we create using `StackNavigator(RouteConfig, options)` or any other navigator can receive the `screenProps` prop. The object passed in this prop gets merged into `this.props` of the `Screen` component. Say that we have the `Home` component:

```
const Home = ({ name }) => (
      <View>
           <Text>Hello ${name}</Text>
     </View>
)
```

Our navigator is created in the following way:

```
const AppNavigator =
      StackNavigator({ home: { screen: Home }}, options);
<AppNavigator screenProps={{ name: 'Vladimir Novick'}} />
```

At the end, we will get **Hello Vladimir Novick** on the home screen and on the other screen, you can use a name prop for other purposes.

Navigation

Navigation is done in several ways. First of all, you can use the navigation prop that is passed down to your screen. It has a `navigate` helper to link the screens like so: `this.props.navigation.navigate('home');`.

Generally speaking, the signature of this helper is `navigate(routeName, params, action)`, where `action` is an action that can be run in a sub-router screen if a screen is actually a nested navigator. It's an advanced topic and won't be covered here. You can read more about actions at `https://reactnavigation.org/docs/navigators/navigation-actions`.

Alternatively, you can use `NavigationActions.navigate`. This is an action creator that creates an action that can be passed to `this.props.navigation.dispatch`.

Consider the following:

```
import { NavigationActions } from 'react-navigation';
const navigateHomeAction = NavigationActions.navigate({ routeName: 'Home',
params: {});
```

You can use it by passing it to `this.props.navigation.dispatch(navigateHomeAction)`.

Basically, `NavigationActions` supplies a set of action creators: `navigate`, `reset`, `back`, and `setParams`.

Redux integration

React-navigation can be easily integrated with Redux.

First, let's define our basic navigator:

```
const AppNavigator = StackNavigator({
  Home: { screen: 'Home' }
});
```

Then, we need to create a specific reducer for our navigator. First, let's get the initial state for it; we can use the following code to get it:

```
const initialState =
  AppNavigator
    .router.getStateForAction(
     AppNavigator.router.getActionForPathAndParams('Home')
    );
```

The reducer itself will be very basic. We don't need to create any switch case for action types; instead, we will use `AppNavigator.router.getStateForAction(action, state)`:

```
const navReducer =
(state = initialState, action) => {
  const nextState =
      AppNavigator.router
  .getStateForAction(action, state);
  return nextState || state;
};
```

Of course, we will add our reducer, along with other reducers, to our `rootReducer` and pass it to `store`, as in a regular Redux setup.

We now can add navigation helpers to the `AppNavigator` navigation prop by importing the `addNavigationHelpers` function from `react-navigation` and using it like this:

```
class App extends React.Component {
  render() {
    return (
      <AppNavigator navigation={addNavigationHelpers({
        dispatch: this.props.dispatch,
        state: this.props.nav,
      })} />
     );
   }
 }
```

Finally, we need to wrap our app with `connect`:

```
connect(({navReducer}) => ({ nav }))(App)
```

Now, all navigation is done by Redux dispatch and the navigation state is stored in the Redux store.

MobX integration

MobX integration is done in a similar way. We also need to pass navigation helpers:

```
<AppNavigator navigation={addNavigationHelpers({
   dispatch: this.props.navStore.dispatchAction,
   state: this.props.navStore.navigationState,
 })} />
```

As you can see here, we have almost the same code, except that we reference `dispatchAction` from MobX `navStore`, as well as `navigationState`. When integrating with MobX, we don't need to create a reducer, but we need to create `navigationState`, as well as a dispatch action, in our store:

```
@observable.ref navigationState = {
    index: 0,
    routes: [
      { key: "Index", routeName: "Home" },
    ],
 };
```

We need to pass our routes to the `navigation` state. Also, we need to create a custom `dispatchAction` in our nav store:

```
@action dispatchAction =
  (action, stackNavState = true) => {
    const previousNavState = stackNavState ?
      this.navigationState : null;
       return this.navigationState = AppNavigator
        .router
        .getStateForAction(action, previousNavState);
 }
```

That's it. We've set up our `react-navigation` with MobX. As you can see, you can easily set it up with pretty much anything. After all, you only override the `state` and `dispatch` function using the `addNavigationHelpers` function.

Setting the app navigation structure for a real app.

Let's take a look at the *YouTube* app's navigation flow.

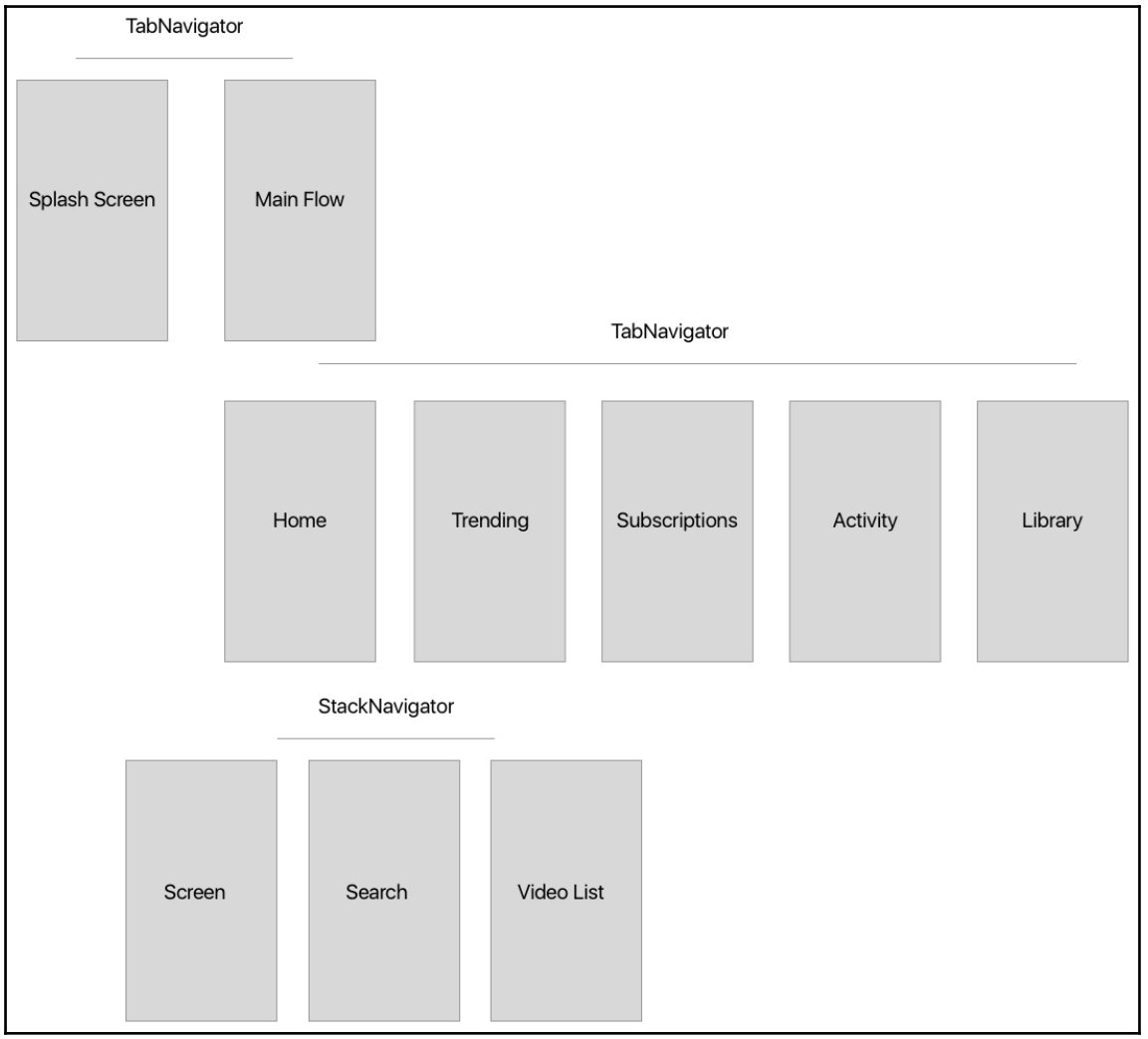
We start with the Main screen, and then you have tabs. Logically, it looks like we have `TabNavigator` at the root level (the splash screen with the YouTube logo on it is a tab, as well as the rest; we will call first the **Splash Screen** and second the **Main Flow**). We will probably also have the authentication screen on the same level.

Then, we have the Home, Trending, Subscriptions, Activity, and Library screens with the corresponding tabs.

We can see that on each screen, if we click on a video, it appears at the top of our screen, so we will probably need to create a component that will show a video or screen, but it won't be a navigation component. We need lots of custom animations here, as well as `PanResponder` to enable us to drag this video down, minimize it, and so on.

In addition to video on the Trending and Subscription pages, if we click on channels at the top of the screen, we get to the video list page, and we can see that we have a back button. So, it will be a nested navigator.

So basically, our navigation will look like this:



First, we will set up our `AppNavigator`:

```
const AppNavigator = TabNavigator(routesConfig, {
  navigationOptions: {
     tabBarVisible: false
     }
  });
export default class App extends React.Component {
  render() {
```

```
      return (
        <AppNavigator />
      );
    }
  }
```

As you can see, we use `TabNavigator`, and we pass it `routesConfig` and `navigationOptions`. The `navigationOptions` can be also passed as configuration objects to the navigator if you want the same options for all the screens in the navigator.

Our `routesConfig` resides in the `config/routes.js` file and looks like this:

```
const routes = {
  splash: { screen: SplashScreen },
  mainFlow: { screen: MainNavigator }
}
```

This happens when `SplashScreen` is actually a React component with an actual screen and `MainNavigator` is our nested `TabNavigator`.

Before writing it, note that we have Search, Video List, and Screen themselves on every tab of our application; so we will create a function that will create a nested `StackNavigator` for each screen:

```
const getStackNavigatorForScreen = (Screen) => StackNavigator({
  main: { screen: Screen },
  search: { screen: Search },
  videoList: { screen: VideoList }
})
```

Then, let's create our nested `TabNavigator` as in the previous scheme:

```
const MainNavigator = TabNavigator({
   home: { screen: getStackNavigatorForScreen(Home) },
   trending: { screen: getStackNavigatorForScreen(Trending) },
   subscriptions: { screen: getStackNavigatorForScreen(Subscriptions) },
   activity: { screen: getStackNavigatorForScreen(Activity) },
   library: { screen: getStackNavigatorForScreen(Library) }
})
```

As you can see, before organizing routes in our application, it's important to understand how an application behaves, and which views need back functionality and which views don't.

When dealing with nested TabNavigators in Android we have to pass disable swiping and animations on nested navigator if we want it to be displayed properly. This means that in order to be cross platform compatible our MainNavigator will look like this:

```
const MainNavigator = TabNavigator({
   home: { screen: getStackNavigatorForScreen(Home) },
   trending: { screen: getStackNavigatorForScreen(Trending) },
   subscriptions: { screen: getStackNavigatorForScreen(Subscriptions) },
   activity: { screen: getStackNavigatorForScreen(Activity) },
   library: { screen: getStackNavigatorForScreen(Library) }
},{
    swipeEnabled: false,
    animationEnabled: false
})
```

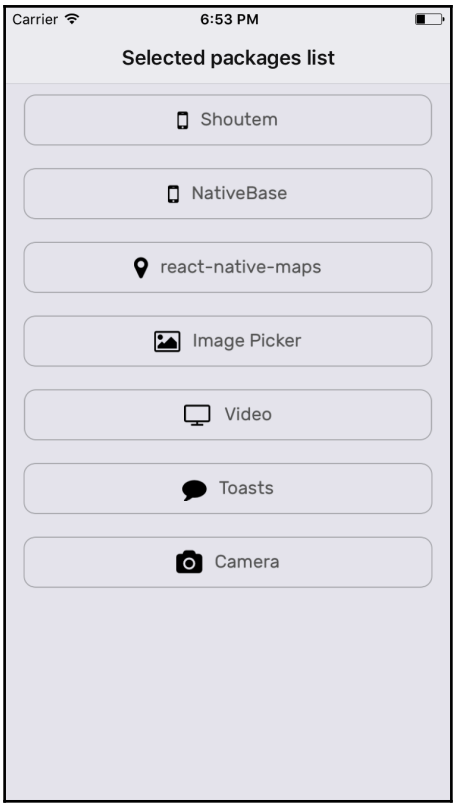
The best open source packages to use

There are dozens of open source packages that make your application development smoother.

In this section, I won't dive deeper into explaining each one of them, but I will mention the mostknown packages at the time of writing this book.

When you create your own application, it's important to search for the packages available for each functionality that you want to implement. Perhaps some behavior or animation you wish to incorporate into your application already exists.

In this section, we will create an application to showcase various open source packages, so make sure to check the code on `github repository` of this book. Essentially, in this application we will have the following Main screen with a list of packages to explore:



Since the code examples in `packagesPlayground` are dependent on the installation of lots of native modules, it's important that you "link" properly each package properly with the following docs. After doing so, you can use the code samples provided in each screen to recreate the same behavior.

Visuals and animations

While there are dozens of packages dealing with visuals and animations, there are some worth mentioning that are part of nearly every application, and some show how far React Native can go with stunning visuals.

react-native-vector-icons

As you've probably seen, we haven't used any icons yet--not anymore. In every React Native app, you will use this package. It gives you over three thousand icons from famous font libraries, such as `Entypo`, `EvilIcons`, `FontAwesome`, `Foundation`, `Ionicons`, `MaterialIcons`, `MaterialCommunityIcons`, `OctiIcons`, `Zocial`, and `SimpleLineIcons`.

The package is installed by running this:

```
npm install react-native-vector-icons --save
```

It is linked according to the readme `https://github.com/oblador/react-native-vector-icons`.

The usage is also explained there, but in a nutshell, you import the `Icon` component and use it like this:

```
import Icon from 'react-native-vector-icons/FontAwesome';
 const MobileIcon = (<Icon name="mobile" size={20} color="#900" />)
```

You can search for icon names using the following page:

`https://oblador.github.io/react-native-vector-icons/`

The `react-native-vector-icons` can also be used to import your own custom fonts.

react-native-animatable

In lots of applications, animations are repetitive swipes, slides, bouncing, and so on; `react-native-animatable` provides preconfigured animated components that you can use without rewriting commonly used animations yourself.

The repository, as well as the documentation and showcase, can be found at `https://github.com/oblador/react-native-animatable`.

For example, for simple pulse animation, instead of wiring it up with an animated API we can simply use the following:

```
<Animatable.View animation="pulse" easing="ease-out"
iterationCount="infinite">
   // View content
</Animatable.View>
```

lottie-react-native

One of the examples of how React Native can be used with really complex native libraries, **Lottie** is a mobile library for Android and iOS that parses *Adobe After Effects* animations exported as JSON. Airbnb created a React Native wrapper.

For Lottie documentation and examples, refer to `https://github.com/airbnb/lottie-react-native`.

Shoutem UI

The Shoutem organization is a mobile app maker built on top of React Native. However, they also created an open source UI library. It has lots and lots of reusable components that every app needs. You are welcome to check it out yourself, and in case you find (and you probably will) a component you need, you are free to use it. In addition, there is a theming option in the Shoutem UI to style your components in more CSS-like styles, providing a theme object that looks similar to the following:

```
const theme = {
  'shoutem.ui.Card': {
  '.dark': {
    backgroundColor: '#000'
  }
}
```

As you can see, Shoutem resembles CSS even more than React Native default styling. Here you have a naming system similar to that used in class names on the web. In addition to styling, you can use declarative Shoutem animations by using its animated components, such as `FadeIn`, `FadeOut`, `ZoomIn`, `ZoomOut`, and `Parallax`.

You can read more about the toolkit, theming, and animations at `http://shoutem.github.io/docs/ui-toolkit/introduction`.

NativeBase

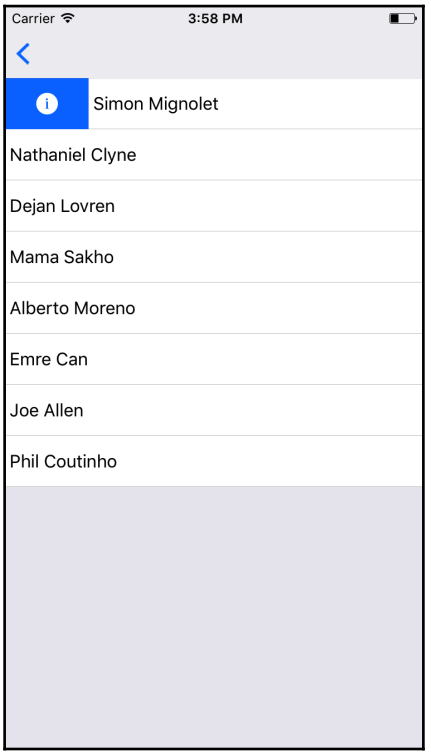
While the Shoutem UI is an open source toolkit used in the Shoutem mobile app builder, NativeBase is also a set of premade and customizable UI components that can be used in your application. NativeBase uses its own layout and theming techniques, like Shoutem UI.

You are welcome to check their official site at `https://nativebase.io/`.

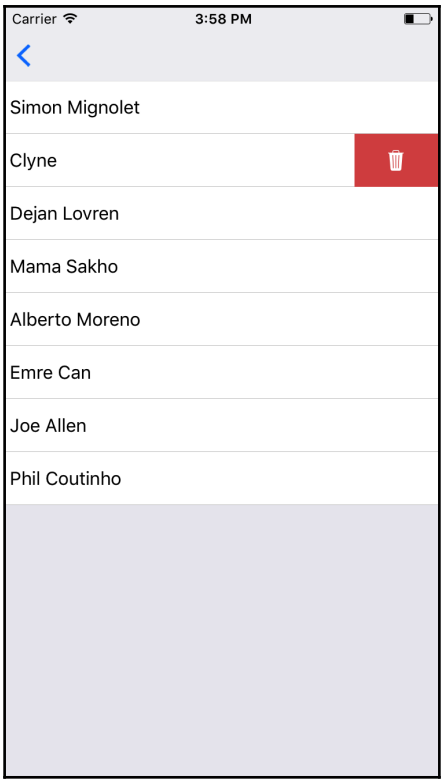
You can also clone the following repository to run the NativeBase `KitchenSink` app. This app essentially showcases all components available in NativeBase:

```
https://github.com/GeekyAnts/NativeBase-KitchenSink
```

Also, in the `packagesPlayground` application that we have in the book repo, you can check out a simple example of NativeBase swipable list usage, which looks like the following image, but without providing any styles. When swiping to the right, you will get an *info* icon on the left:



When swiping to the left, you will get the following delete icon on the right:



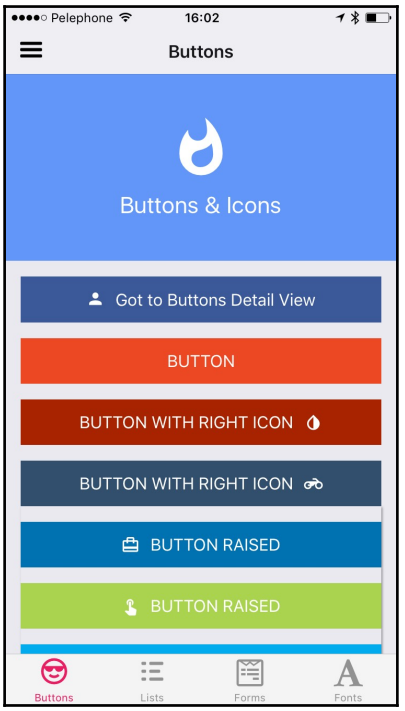
Note that this is done only by NativeBase, without any custom styles or animations, and dealing only with `PanResponder` API.

react-native-elements

The `react-native-elements` is another UI toolkit you can use. The difference between Shoutem and NativeBase is that `react-native-elements` is not restrictive in terms of how you should structure your layout. It lacks layout, theming, and animation solutions; however, it provides several useful components that are commonly used between apps. In the next chapter, we will use `react-native-elements` for our Twitter clone app. You can check more about `react-native-elements` at `https://react-native-training.github.io/react-native-elements/`.

You can also use the Expo app to play with a `react-native-elements` example app at `https://expo.io/@monte9/react-native-elements-app`.

It looks like this:



Social providers

In this section, we will list several packages from known social networking sites. It's important that you integrate at least basic social behavior in your app to enhance user engagement and make your app more popular.

Facebook

Facebook is an integral part of any application nowadays. Many apps are able to use a Facebook login, share actions on a profile page, or retrieve data through the Facebook Graph API. The `react-native-fbsdk` package wraps both the iOS and Android Facebook SDK. It's not so easy to set up, and will require reading Facebook integration docs for both iOS and Android. Ensure that you follow all the steps listed in the package readme and in the official Facebook documentation at `https://github.com/facebook/react-native-fbsdk`.

OAuth

This package provides OAuth 1 and 2 support for React Native. As a result, it supports the following providers:

- Twitter
- Facebook
- Google
- GitHub
- Slack

More information can be found at `https://github.com/fullstackreact/react-native-oauth`.

Additional APIs

In npm, there is a huge list of various APIs ported to React Native that gives your React Native project's additional functionalities. I will mention a few of them now.

ExpoKit

Expo, as mentioned in the early chapters, gives its users lots of additional APIs.

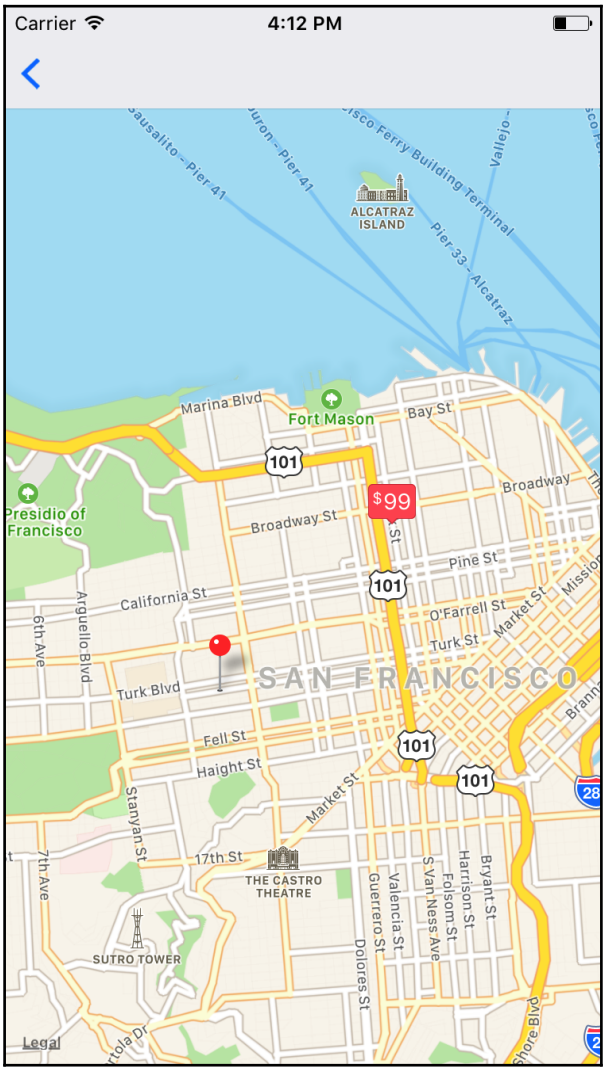
There are two ways of developing an ExpoKit project. One is by using XDE and the second one is by using `create-react-native-app`. Generally, when creating exponent apps, you are limited to ExpoKit APIs, and you cannot add modules that have native code dependencies. So in order to add them, you basically need to *eject*--if we used `create-react-native-app`--by running `npm run eject` or *detatch*--if we've used XDE to create our project. Then we detach by running the `expo detach` command.

You can read more about Expo in its official docs at `https://docs.expo.io`.

Maps

Maps were once available in React Native; however, they are not used now. When you need a map in your application, consider using `react-native-maps` from Airbnb. It's used in dozens of bestseller apps, and is stable and reliable. You can find the maps at `https://github.com/airbnb/react-native-maps`.

In our example app, we have an example for animated markers. The map view you will see will look like this:

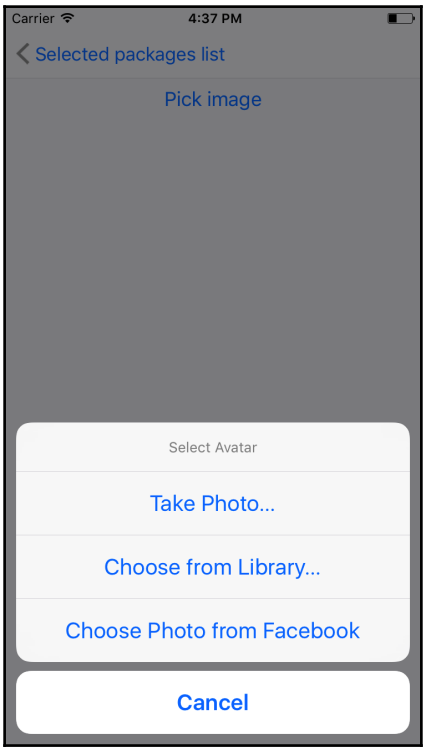


Note that the price marker can be dragged around. You can open a Chrome console and you will see all the logged data when you drag the marker around.

Image Picker

The **Image Picker** package is a better solution for using Image Picker capabilities in your app. Image Picker can be downloaded at `https://github.com/react-community/react-native-image-picker`.

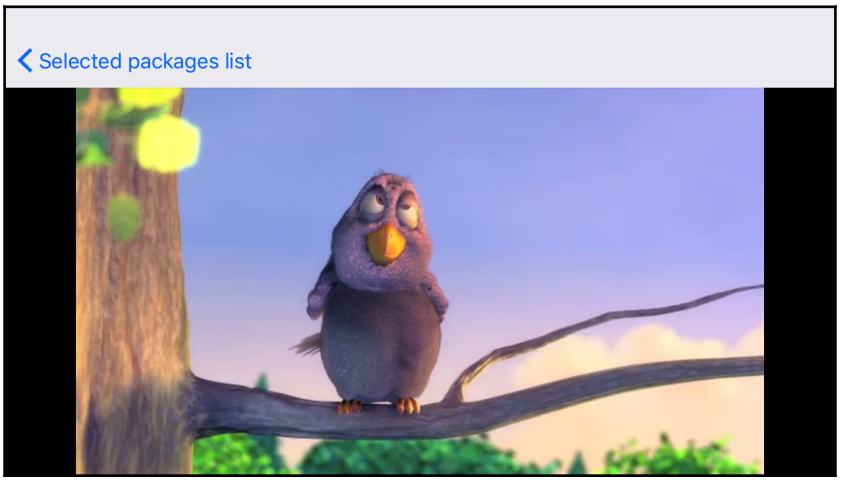
Image Picker will look like the following, and it will give you lots of options, including adding custom buttons, as shown in the following use case (The **Choose Photo from Facebook** button). You can check the code in `Chapter 10`, *Working with External Modules in React Native*, ngrx/store + ngrx/effects for state management/ `packagesPlayground` app in the book `repo` folder.



Video

While not all apps will use video, there are quite a few that will need them. Instead of implementing videos yourself, use `react-native-video` from the creators of `react-navigation`; you can find it at `https://github.com/react-native-community/react-native-video`.

The video component can stream videos from both local files and from URLs. In our *packagesExample* app, we stream video from `https://www.quirksmode.org/html5/videos/big_buck_bunny.mp4`.



As you can see from the picture, the video component also supports screen rotation and various other parameters that can be checked in official documents.

Toasts

Some apps take the path of making a unified UI between Android and iOS. Some behaviors, such as toasts, are specific for Android apps, and iOS does not have a dedicated API for that. As such, it's not implemented in React Native. In case you choose to create Android-like toasts in your React Native app, this can be done using the `https://github.com/crazycodeboy/react-native-easy-toast` package.

Camera

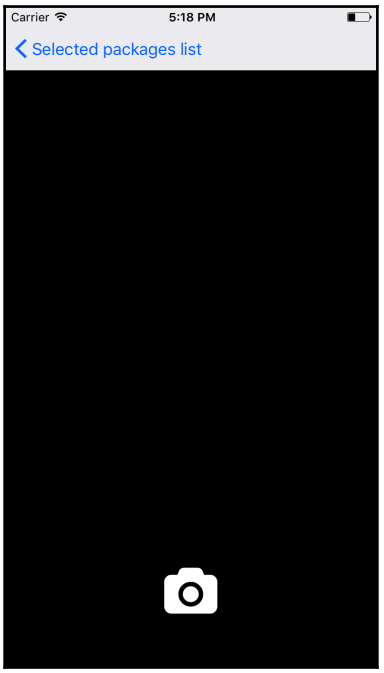
The `https://github.com/lwansbrough/react-native-camera` package is your solution for using a camera in your app.

Note that the camera won't work in a simulator, rendering only a black screen, so you should test it on an actual device.

Let's create a basic camera component in our `packagesPlayground` app.

```
export default class RNCamera extends Component {


render() {
  return (
    <View style={styles.container}>
      <Camera ref={(cam) => { this.camera = cam; }}
        style={styles.preview}
        aspect={Camera.constants.Aspect.fill}>
          <Icon size={50}
                style={styles.capture}
                onPress={() => this.takePicture()} name="camera" />
      </Camera>
    </View>
  );
}

takePicture() {
  this.camera.capture()
    .then((data) => {
      console.log(data)
    })
    .catch(err => console.error(err));
  }
}
```

We will end up with the following screen in the simulator:



The styles for it will be as follows:

```
const styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center'
    },
    preview: {
        flex: 1,
        width: '100%',
        justifyContent: 'flex-end',
        alignItems: 'center'
    },
    capture: {
        flex: 0,
        backgroundColor: 'transparent',
        borderRadius: 5,
        color: '#fff',
        padding: 10,
        margin: 40
```

```
    },
    image: {
        width: '100%',
        height: '100%'
    }
});
```

When pressing on the icon, picture will be added to `CameraRoll` and we will log the following to the console:

```
{
  mediaUri:"assets-library://asset/asset.JPG?id=BBA19DDD-852C-4314-
BD15-01B21EF5DEFC&ext=JPG"
  path:"assets-library://asset/asset.JPG?id=BBA19DDD-852C-4314-
BD15-01B21EF5DEFC&ext=JPG"
}
```

The `mediaUri` is not something we can use straight out of the box, however the next package we will look at will solve this problem for us.

Data related packages

There are several data-related packages that can be used to manipulate data inside your React Native app, but I want to introduce two of them.

react-native-fetch-blob

The `react-native-fetch-blob` is a huge package focusing on data transfer. It supports different ways of file transfer, including storage, Base64, file streaming, caching, and more. This library is usually used in most of the apps. It has extensive documentation, which is available in the `https://github.com/wkh237/react-native-fetch-blob` package repository. Let's, for example, alter our RNCamera screen to show a proper image preview once the image has been captured.

First of all, let's add a new state to our `RNCamera` component:

```
state = {
  imageUri: false
}
```

Now let's alter our render function to be the following:

```
  render() {
        return (
```

```
            <View style={styles.container}>
                { this.getCameraOrImage() }
            </View>
        );
    }
```

Then let's create the `getCameraOrImage` function:

```
getCameraOrImage() {
  return this.state.imageUri ?
    <Image style={styles.image} source={{ uri: this.state.imageUri }} />:
    <Camera ref={(cam) => { this.camera = cam; }}
      style={styles.preview}
      aspect={Camera.constants.Aspect.fill}>
      <Icon size={50}
         style={styles.capture}
         onPress={() => this.takePicture()} name="camera" />
    </Camera>
}
```

Now it's time to get our image from `CameraRoll` when it's saved:

```
this.camera.capture()
            .then((data) => {
                RNFetchBlob.fs.readFile(data.mediaUri,
               'base64').then(data => {
                    let base64Image = `${data}`;
                    this.setState({
                        imageUri: base64Image
                    })
                })
            })
            .catch(err => console.error(err));
```

We call the `RNFetchBlob.fs.readFile` function and pass it the `mediaUri` that we've received from camera. We read it in Base64 format.

In order to pass it properly to the `Image` component, we have to add a `data:image/jpeg;base64` prefix.

Then we set our state variable `imageUri` to be newly created base64Image data. Eventually, we will get the following image rendered in our app:

It's important to know that `react-native-fetch-blob` is your one-stop solution for all fetching and filesystem needs, and you will probably use it a lot during development, so it's important to familiarize yourself with its APIs.

The following is the current list of capabilities that `react-native-fetch-blob` supports:

- Transfer data directly from/to storage without Base64 bridging
- The file API supports regular files, asset files, and `CameraRoll` files
- Native-to-native file manipulation API, reducing JS bridging performance loss
- File stream support for dealing with large files
- Blob,file, and XMLHttpRequest polyfills that make browser-based libraries available in RN (experimental)

The official docs can be found at `https://github.com/wkh237/react-native-fetch-blob`.

react-native-firebase

In the previous chapters, we saw that we can use the Firebase web SDK for basic Firebase functionality; however, the mobile Firebase SDK provides many more features, and is more performant. Here's the list of Firebase features available in `react-native-firebase`. As you can see, it's much more than what is available on the web:

Firebase Features	v1	v2	Web SDK
AdMob	❌	✅	❌
Analytics	✅	✅	❌
App Indexing	❌	❌	❌
Authentication	✅	✅	✅
Cloud Messaging	✅	✅	❌
Crash Reporting	✅	✅	❌
Dynamic Links	❌	❌	❌
Invites	❌	❌	❌
Performance Monitoring	✅	✅	❌
Realtime Database	✅	✅	✅
- Offline Persistance	✅	✅	❌
- Transactions	✅	✅	✅
Remote Config	✅	✅	❌
Storage	✅	✅	❌

For more information go to `https://github.com/invertase/react-native-firebase`

react-native-push-notifications

`react-native-push-notification` is an amazing package that enables you to add push notifications to your application. In our example, we will add only push notifications to the iOS part. For Android, you can go through the official documentation on package github repo: `https://github.com/zo0r/react-native-push-notification/`.

In order to install it, run:

```
npm i -S react-native-push-notification
```

Then you have to link it using a `react-native` link command. But that's not all. Since `react-native-push-notification` is based on the `PushNotificationiOS` API, you have to follow the instructions in official the documents to set up all the required certificates, add capabilities to your app, and so on. The reason we didn't cover it in the API section is that this API constantly changes and will be subject to improvements in the near future. To set up it properly, follow the docs at `https://facebook.github.io/react-native/docs/pushnotificationios.html#content`.

In our example, we will also use `AppState` API to send notifications, but only if our app is in the background. Let's define our component state:

```
state = {
  appState: AppState.currentState,
  title: 'Exit to Home screen and wait for 5 seconds'
}
```

Let's take a look at the `render` function:

```
render(){
  return (
    <View>
      <Text>Exit to Home screen and wait for 5 seconds</Text>
    </View>
  )
}
```

As you can see, there is nothing here in the `render` function.

Now let's get to `PushNotifications`:

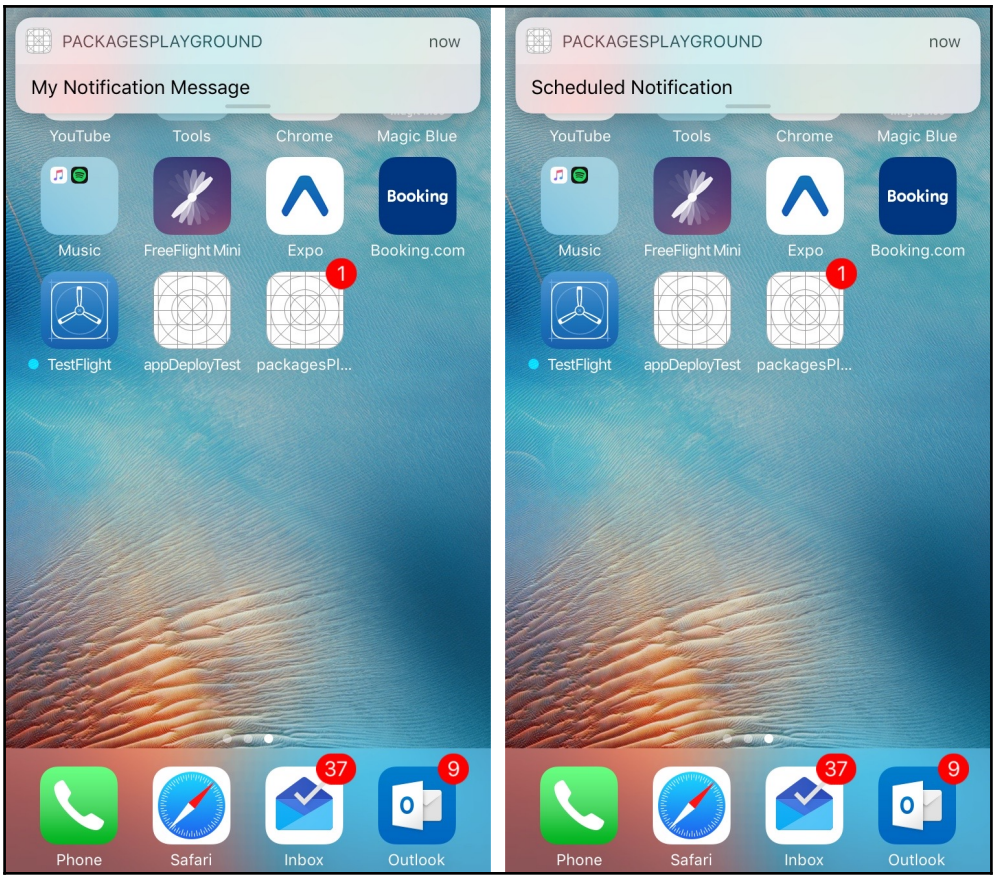
```
componentDidMount(){
  PushNotification.configure({
    onNotification: (notification) => {
      console.log('NOTIFICATION:', notification);
   }
  });
  AppState.addEventListener('change',
    this.changeAppState.bind(this));
  }

componentWillUnmount() {
  AppState.removeEventListener('change',
    this.changeAppState.bind(this));
}
```

On our `componentDidMount()` lifecycle hook, we will configure our `PushNotification` by passing the `onNotification` callback. We simply log our notification object to the console.

Then we set the `AppState` change listener to make sure that when we exit to the background, we will send a notification:

```
changeAppState(nextAppState){
        if (nextAppState === 'background') {
            PushNotification.localNotification({
              number: 0,
              message: "My Notification Message",
              playSound: false
            });
            PushNotification.localNotificationSchedule({
                number: 0,
                message: "Scheduled Notification",
                date: new Date(Date.now() + 10000)
            })
        } else {
            PushNotification.cancelAllLocalNotifications();
        }
        this.setState({ appState: nextAppState });
}
```

Here, we call `PushNotification.localNotification` to trigger immediate notification and also set a **scheduled** notification to activate in 5 minutes. In the end, when you enter this screen and exit it to the home screen, you will see the following notification:



react-native-i18n

This package is a wrapper for i18n internalization for React Native. It supports both passing locales implicitly and fallbacks to default locales, and even getting the language by device locale.

For more information, go to `https://github.com/AlexanderZaytsev/react-native-i18n`.

Boilerplates

There are several boilerplates out there, but no boilerplate stands out as much as Ignite from *Infinite Red* company. Ignite is not just a boilerplate, but a CLI tool that helps you with creating your screens. Ignite works with Redux out of the box, so if Redux is your preferred way of state management, you are welcome to use it as is. In case you prefer MobX, you probably should alter it to your needs.

To install Ignite, you must install it globally:

```
npm i -g ignite-cli
```

Then, if you wish to create your application, run:

```
ignite new igniteTestApp
```

It will ask us several questions:

- Whether we want Ignite development screens--Ignite development screens are examples of various behaviors that you can use in your app. Essentially, we want to install them only to have valuable examples.
- Whether we want to install `react-native-vector-icons`. Usually you will need `react-native-vector-icons`, but it's a good option to have flexibility in case we don't want to use this package.
- Whether we want to install `react-native-i18n`.
- Whether we want to install `react-native-animatable`.

The Ignite CLI tool and boilerplate at the time this book was released uses the 0.45.1 version of React Native, while the official version is 0.47.1, so take that in mind if you use the latest features available only in the 0.47.1 version.

While installing and after finishing the installation, you will see the following screen:

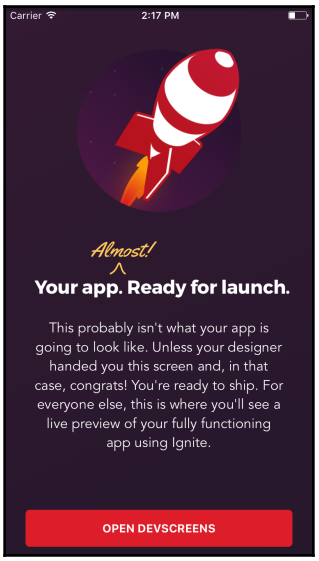
```
~/dev/Book/learning-react-native/chapter10(master*) » ignite new igniteTestApp
------------------------------------------------
  (                  )   (
  )\ )   (        ( /(   )\ )    *   )
 (()/(   )\ )     )\()) (()/(  ` )  /(   (
  /(_)) (()/(    ((_)\   /(_)) ( )(_))  )\
 (_))    /(_))_   _((_) (_))   (_(_())  ((_)
 |_ _|  (_)) __| | \| | |_ _|  |_   _|  | __|
  | |     | (_ | | .` |  | |     | |    | _|
 |___|     \___| |_|\_| |___|    |_|    |___|
------------------------------------------------

An unfair headstart for your React Native apps.
https://infinite.red/ignite

------------------------------------------------

🔥 igniting app igniteTestApp
✔ using the Infinite Red boilerplate v2 (code name 'Andross')
✔ added React Native 0.45.1 in 33.59s
? Would you like Ignite Development Screens? Yes
? What vector icon library will you use? react-native-vector-icons
? What internationalization library will you use? react-native-i18n
? What animation library will you use? react-native-animatable
✔ added ignite-ir-boilerplate in 9.27s
✔ added ignite-dev-screens in 17.73s
✔ added ignite-vector-icons in 14.51s
✔ added ignite-i18n in 12.78s
✔ added ignite-animatable in 10.99s
✔ added ignite-standard in 35.51s
✔ configured git
✔ ignited igniteTestApp in 387.98s

    Ignite CLI ignited igniteTestApp in 387.98s

    To get started:

      cd igniteTestApp
      react-native run-ios
      react-native run-android
      ignite --help

    Read the walkthrough at https://github.com/infinitered/ignite-ir-boilerplate/blob/master/readme.md#boilerplate-walkthrough

    Need additional help? Join our Slack community at http://community.infinite.red.

    Now get cooking! 🍽
```

As you can see, we get links to the official documents, as well as various commands we can run. One of these commands is `ignite --help`.

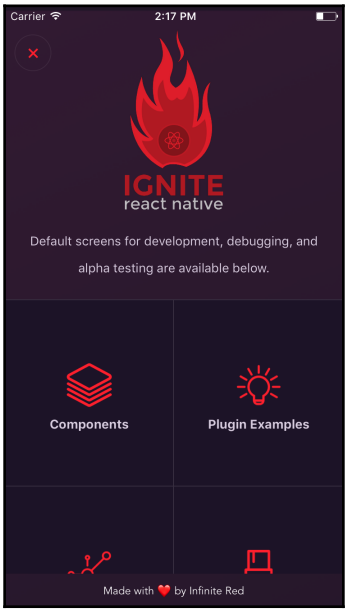
After running it, you will see the following available commands list:

```
add (a)          Adds an Ignite plugin.
attach           Attaches Ignite CLI to an existing project.
doctor           Checks your dev environment for dependencies.
generate (g)     Generates some files.
info             Displays info about a given Ignite plugin.
list             Lists known Ignite plugins.
new (n)          Generate a new React Native project with Ignite CLI.
plugin (p)       Manages Ignite plugins
remove (r)       Removes an Ignite CLI plugin.
search           Searches known Ignite plugins.
spork            Copy templates as blueprints for this project
version (v)      Prints current version of installed Ignite
```

If we run our app, we will see the following launch screen:



The DevScreen will look like this:



It will contain valuable examples of features you will probably want in your application, so make sure that you check them out.

Also, we will see everything set up for starting development: folder structure, Redux, Navigation, application styles organized in the best way possible, and so on. We won't cover everything about `ignite cli` in this book, so make sure to check out the official Ignite website at `https://infinite.red/ignite`.

You can check out the official docs at `https://github.com/infinitered/ignite/blob/master/docs/README.md`.

Writing your own Native modules

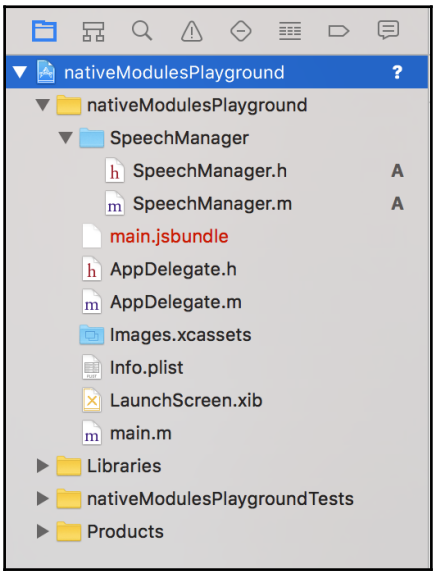
Writing your own native modules might sound intimidating if you come from a web development background; however, it's probably the first question people ask when they start with React Native. At some point, you want to add custom functionality beyond the JavaScript world and wrap a native API. In this section, we will cover the basic steps you should know when wrapping native APIs. Here, we will see both Objective-C and Java code; you can also use Swift and Kotlin.

Diving into iOS and Objective-C

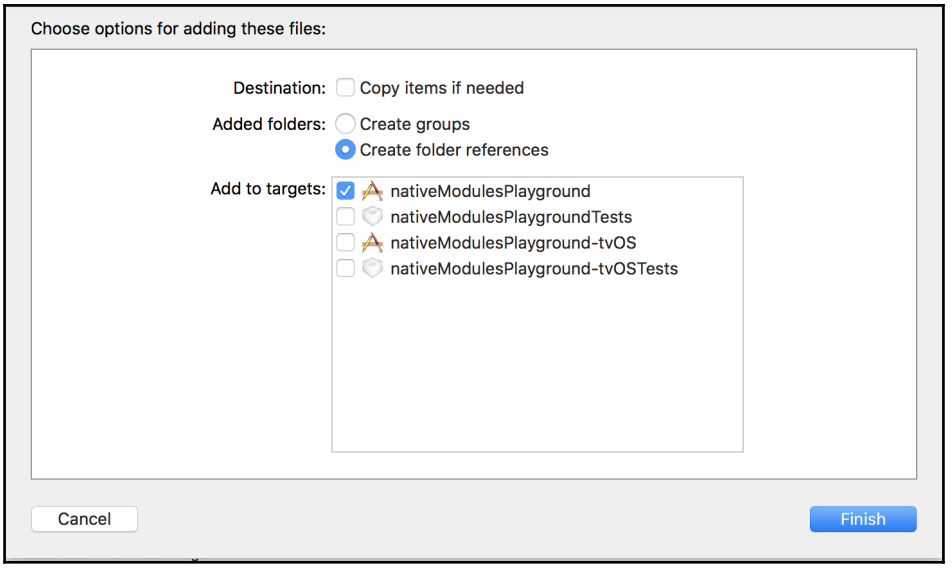
Let's, for instance, give our app the ability to use authorize with Siri. On iOS, by default, we are using Objective-C. When working in Objective-C, we need to perform the following steps:

1. Create the `ios/SpeechManager` folder.
2. Create a header file. In our example, we will simply create the `SpeechManager` native module, which will use the iOS speech API for transforming text to speech.
3. We will create a folder under the `iOS` folder and call it `SpeechManager`.
4. Create an Objective-C file for our `SpeechManager` implementation; we will call it `SpeechManager.m`.

5. Open a XCode project and drag and drop the `SpeechManager` files into the project:



6. When we drop them, XCode will display the following prompt, asking us whether we want to reference the folder where files reside in the XCode project:



Alternatively, you can select the `nativeModulesPlayground` folder and add the `SpeechManager` files by going into **File | Add Files to nativeModulesPlayground.** In that case, the options prompt won't be displayed since XCode will automatically create folder references.

Now, let's write some code.

The `SpeechManager.h` file will look like this:

```
// SpeechManager.h
 #import <React/RCTBridgeModule.h>

 @import AVFoundation;


 @interface SpeechManager : NSObject <RCTBridgeModule>
 @property (nonatomic, strong) AVSpeechSynthesizer *manager;
 @end
```

We import the `RCTBridgeModule` header file and create the `SpeechManager` interface, which is an implementation of the `RCTBridgeModule` protocol.

`RCTBridgeModule` is what we use in iOS native code to talk with React Native. RCT is short for React. Essentially, we import this module in our headers (`.h`) files.

We also import `AVFoundation` and create `property *manager` , which is specific to the iOS Speech API.

Our `SpeechManager.m` file will look like this:

```
// SpeechManager.m

 #import "SpeechManager.h"


 @implementation SpeechManager;

 // To export a module named SpeechManager

 RCT_EXPORT_MODULE();

 RCT_EXPORT_METHOD(talk:(NSString *)text)
 {
   NSString *voiceLanguage = @"en-US";
   AVSpeechUtterance *utterance = [[AVSpeechUtterance alloc]
   initWithString:text];
   utterance.voice = [AVSpeechSynthesisVoice
```

```
    voiceWithLanguage:voiceLanguage];
    utterance.rate = 0.4;
    self.manager = [[AVSpeechSynthesizer alloc] init];
    self.manager.delegate = self;
    [self.manager speakUtterance:utterance];
}

@end
```

We first import the header file:

```
#import "SpeechManager.h"
```

Then, we implement the interface we created in the header file:

```
@implementation SpeechManager;
```

Next, we'll call the `RCT_EXPORT_MODULE();` function, which registers our native module with React Native. By default, our module gets the same name as our file; however, we can pass an optional string argument to the `RCT_EXPORT_MODULE` function to rename it.

Then, with `RCT_EXPORT_METHOD`, we create the `talk` method that we will call from our JavaScript.

Now, in order to call this method from JavaScript in our `App.js`, we do the following:

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

export default class nativeModulesPlayground extends Component {

 talkThroughText(){
    const { SpeechManager } = NativeModules;
    SpeechManager.talk('Welcome to React Native!');
 }

 render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Welcome to React Native!
        </Text>
        <Button  onPress={() => this.talkThroughText()} />
      </View>
    );
  }
}
```

After building this, our `SpeechManager` will be available on `NativeModules`, which we will import from React Native. As you can see, we use the `talk` method we defined in Objective-C.

As you can see, writing your own native module is pretty easy; however, you need to know Objective-C and iOS to do so. Don't worry--because of the npm ecosystem, there are packages for almost anything you can think of. If you are interested in learning more about how to write native modules, there is a more detailed guide in the official documentation at `https://facebook.github.io/react-native/docs/native-modules-ios.html`.

Note that you probably won't need to write native modules unless you want to implement something very specific. For most use cases, there are community packages out there on npm that you can check and incorporate in your app.

Android

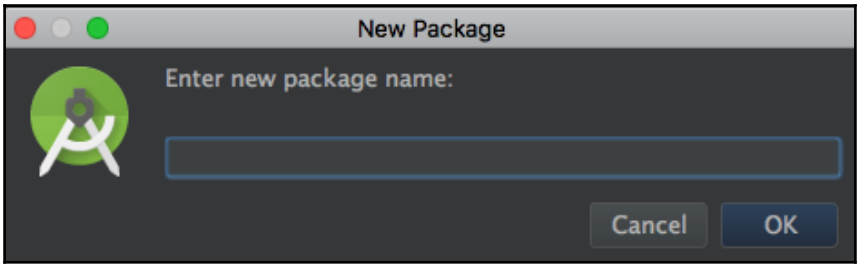
Android is harder to bridge to the React Native world because of large amount of boilerplate code you have to write; however, it's important that you abide by the following rules in order to connect the Android native module to React Native.

Let's implement the equivalent of the `SpeechModule` in Android.

Creating folder structure and files

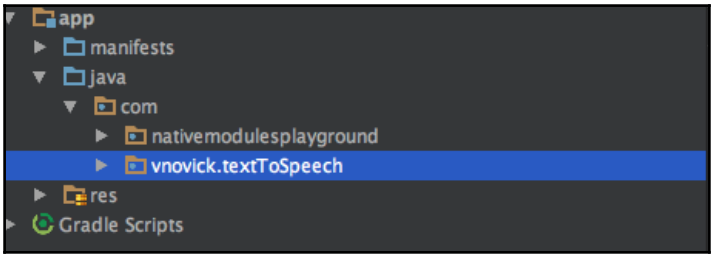
First of all, let's open Android Studio and open our Android directory. Navigate to the `app/java` folder and through the menu, create a new package by going into **File** | **New** | **New Package**.

You will be prompted with the following window:



Enter your package name with whatever namespace you want. I've created mine under `com.vnovick.textToSpeach`.

This is what I will get in the Android Studio project hierarchy:



Now it's time to create two files. The first one will be your actual module implementation and the second will be the package. Let's call the first `TextToSpeechManager` and the second `TextToSpeechPackage`.

Creating a native module Java class

In order to create native module you need to extend `ReactContextBaseJavaModule`:

```
public class TextToSpeechManager extends ReactContextBaseJavaModule {
   public TextToSpeechManager(ReactApplicationContext reactContext) {
      super(reactContext)
   }
}
```

Next, you need to override the `getName` method. This method will return a string, which is basically the name of your module inside `NativeModules`.

Now, in order to expose the methods to React Native, we should decorate them with `@ReactMethod`. In our case, we will have the following method:

```
@ReactMethod
public void translate(String message) {
    // specific implementation.
}
```

Before we proceed with `TextToSpeechPackage` file, let's implement the actual text to speech behavior. First of all, we import `TextToSpeech` from `android.speech.tts`:

```
import android.speech.tts.TextToSpeech;
```

Then we will set a private `textToSpeechManager` variable inside our class:

```
TextToSpeech textToSpeechManager;
```

Then we will transform our constructor to initialize the `textToSpeachManager` properly according to the Android API. What is important here is that, instead of using `getApplicationContext()` as described in the Android API, we will call `getReactApplicationContext()`. The rest is the same usage as in a regular Android application:

```
public TextToSpeechManager(ReactApplicationContext reactContext){
    super(reactContext);
    textToSpeechManager = new TextToSpeech(getReactApplicationContext(),
       new TextToSpeech.OnInitListener() {
          @Override
          public void onInit(int status){
            if (status != TextToSpeech.ERROR) {
                textToSpeechManager.setLanguage(Locale.US);
            }
          }
      }
    );
}
```

Now let's implement our `translate` method:

```
@ReactMethod
public void translate(String message) {
    textToSpeechManager.speak(message, TextToSpeech.QUEUE_FLUSH, null);
}
```

Creating a native module package

The next step will be to create a package and register our module there.

Inside our `TextToSpeachPackage` file we will use the following boilerplate:

```
public class TextToSpeechPackage implements ReactPackage {

    @Override
    public List<ViewManager> createViewManagers(ReactApplicationContext
reactContext) {
        return Collections.emptyList();
    }

    @Override
```

```
        public List<NativeModule> createNativeModules(
                ReactApplicationContext reactContext) {
            List<NativeModule> modules = new ArrayList<>();

            modules.add(new TextToSpeechManager(reactContext));

            return modules;
        }

    }
```

As you can see, in the `modules.add` function we create a new instance of our `TextToSpeechManager` class and pass `reactContext` to it.

Registering the package

The last steps we need to take in the native world is to register our package inside `MainApplication.java`. For that, under the `android/app/src/main/java/com/nativemodulesplayground` folder or `app/java/com/nativemodulesplayground`, as can be seen in Android Studio, we need to open `MainApplication.java` and find the `getPackages` method. We need to add our module to the packages list:

```
@Override
protected List<ReactPackage> getPackages() {
  return Arrays.<ReactPackage>asList(
      new MainReactPackage(),
      new TextToSpeechPackage()
  );
}
```

Now it's time to consume our new native module in the JavaScript world. Let's add it in a similar way to what we did with `SpeechManager` in iOS, but this time in `index.android.js`:

```
export default class nativeModulesPlayground extends Component {

  talkThroughText(){
    const { TextToSpeechManager } = NativeModules;
    TextToSpeechManager.translate('Welcome to React Native!')
  }

  render() {
    return (
      <View style={styles.container}>
```

```
          <Text style={styles.welcome}>
            Welcome to React Native!
          </Text>
          <Button  onPress={() => this.talkThroughText()} />
        </View>
      );
    }
  }
```

Now run `react-native run-android` with an open emulator and enjoy the Android speech API saying **Welcome to React Native!**.

You can check out more details on integrating Android native modules in the official documents at `https://facebook.github.io/react-native/docs/native-modules-android.html`.

Integrating React Native with the existing apps

Due to the extensible structure of React Native, it can be easily integrated with the existing apps. There are a set of steps that should be performed when integrating your existing application with React Native:

- Create a new folder for your project with the iOS or Android folder inside. Copy your iOS or Android existing apps inside the iOS or Android folders.
- `run npm init` in the root folder to generate the npm package in your folder.
- Add the following commands to your `package.json`:

```
{
  "name": "MyReactNativeApp",
  "version": "0.0.1",
  "private": true,
  "scripts": {
     "start": "node node_modules/react-native/local-cli/cli.js start"
  }
}
```

Then install all your JavaScript dependencies:

```
npm install --save react react-native
```

After setting the directory structure and React Native dependencies, you should install and configure your desired platform dependencies. For iOS, that would be the installation and configuration of CocoaPods and for Android it will be the Maven configuration.

Next, we will be adding `RCTRootView` to your native iOS app and `ReactRootView` to your Android app. This is a container view that hosts all of your React Native application. After adding it, you will be able to write your React Native code in a regular way and see whether it shows up in your `RCTRootView` or `ReactRootView`.

A more detailed explanation can be found in the official documentation at `https://facebook.github.io/react-native/docs/integration-with-existing-apps.html`.

Summary

In this chapter, we learned three things--we dove deeper into navigating, got familiar with lots of open source libraries, and learned how we can bridge the native world with the JavaScript world using React Native's extensible nature. We started by learning about navigators, how they are used, and how exactly we structure navigation for huge applications, such as YouTube. Then, we took a look at the list of best open source packages. We saw packages dealing with the UI, animations, icons, data, social providers, additional APIs, and so on. After we got familiar with a small subset of packages available on npm, we dove deeper into Objective-C and Java code to understand how native modules can be exposed to the JavaScript world. Finally, we discussed what steps should be taken if you want to integrate a React Native app as a view inside your existing application.

Now you have got to the point where you can confidently write enterprise-level applications. The examples you've seen throughout this book lead us to this moment. Bringing it all together and understanding the process of how to create a fully-functional app from start to finish will be covered in the next chapter, where you will create a Twitter client app clone. We will go through the whole process, from looking at design, prototyping, creating mocks, and connecting to actual data, to the point where it's fully functional. Now, it's time to bring everything you've learned together and write our app from start to finish.

11
Understanding Application Development Workflow by Recreating Twitter

Welcome to the eleventh chapter. In this chapter, we will combine everything we've learned so far and create a functional Twitter clone. The Twitter app is really complex, so we won't be creating all the screens; however, we will create infrastructure and architecture, so you can add more screens as a home assignment. We will start by looking at Twitter design and will map screen navigation and common behaviors so that we will be able to tackle them later. We will discuss what screens we will be dealing with in this book and how we will address the rest of the app. After creating high-level design of our app, we will start by creating functional screens routing. After creating navigation structure, we will begin by creating common elements across the screens as well as basic styling. During this process, we will implement several animations as well as use the existing modules from npm to make our app structure process faster. At the next stage, we will investigate the Twitter API to understand in which format we get the data from it. We will create mocks inside our components and implement basic functionality with these mocks.

The next step will be to connect Redux state management to your application. We will create relevant reducers and actions for parts of the application we chose to clone. Then, we will take a look how the same is done with MobX. The final stage will be to connect our app to the real data from Twitter. This includes authentication, data retrieval, and persistence. At the end, you will get an idea of how the process of creating an app looks on a larger scale. Working on a huge app like Twitter will give you the understanding of what consideration you should take when you start developing enterprise-level apps.

So, let's summarize. Here's what we will learn in this chapter:

- How to design proper architecture for your app based on design
- How to start with functional navigation and the wireframe of your application
- How to gradually introduce styles and animations to your application screens by mocking data
- How to move mocked data to Redux or Mobx state and connect it to application
- How to authenticate and retrieve data from Twitter
- How to post new Tweets using the Twitter API

Defining your application requirements

In the following section, we will take a look at the Twitter application and will define what exactly we want to create in this book. When looking at *Twitter* app on your phone, it does look fairly simple, Splash screen, then Login or Signup, and then four tabs, but that is only at first glance. In fact, we have lots and lots of various screens and complex navigations in Twitter, so we will create several things for our app:

- An app won't have the signup screen flow and will assume that you have a Twitter account
- An app will remember a user if they're authenticated (we will need `AsyncStorage` for that)
- A user will be able to navigate between tabs
- A user will be able to compose and post a new tweet
- A user will be able to access his profile
- Each tweet will be parsed to make links, hashtags(#), and profiles(@) clickable:
    - Clicking on a profile will lead you to a profile page and pass relevant profile data to a page
    - Clicking on a hashtag will lead you to a hashtag page and pass relevant hashtag data to a page
    - Clicking on a link will load it inside a WebView
- Each tweet will have counters for retweets and likes
- Each tweet will have the reply, retweet, and like actions
- Each tweet will be able to show embedded images

- By clicking on a tweet, we will be able to view the tweet on a separate screen
- There will be a navigation header across all the screens, which will change according to the screen it resides on
- On the profile page, we will see a horizontal carousel with various Twitter feed types
- We will be able to browse through these tweets and click on each tweet/it's hashtags, mentions, URLs, and so on

In terms of styling and design, we will design the following parts:

- Common header component with back, link to profile, and new tweet functionality
- Home page with a list of tweets
- Tweet page with relevant tweet data
- Profile page with feed carousel
- New tweet page with animations

The Hashtag, Discovery and Notification pages will be implemented as wireframe as well as their header. Their final implementation is up to you based on the same principles we will use when we will create the rest of the app. You may find that part of the functionality won't be implemented since it's out of the scope of this chapter. Take it as a challenge; implement the Twitter client app with all the functionalities that you want.

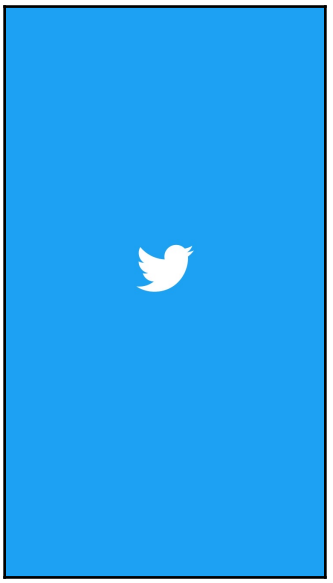
The idea of the chapter is to explain the techniques of bringing Application from design to prototype and to final implementation.

We will focus on iOS app, but if you wish to implement Android app instead, the changes will be minor mostly in react-navigation settings. Make sure to check official docs of react-navigation for specific android settings

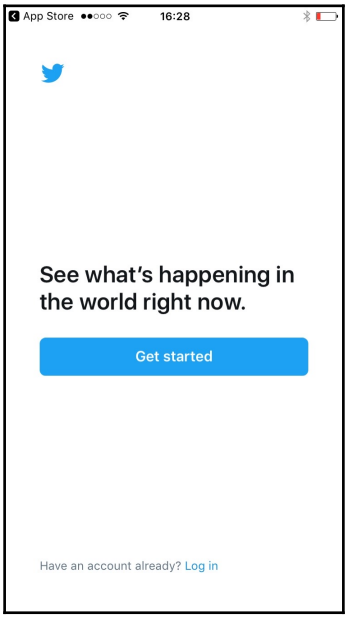
Defining your application architecture using a desired design

We've talked about different screens; let's now take a look at the actual Twitter design and then create a navigation flow architecture for our app based on this design.

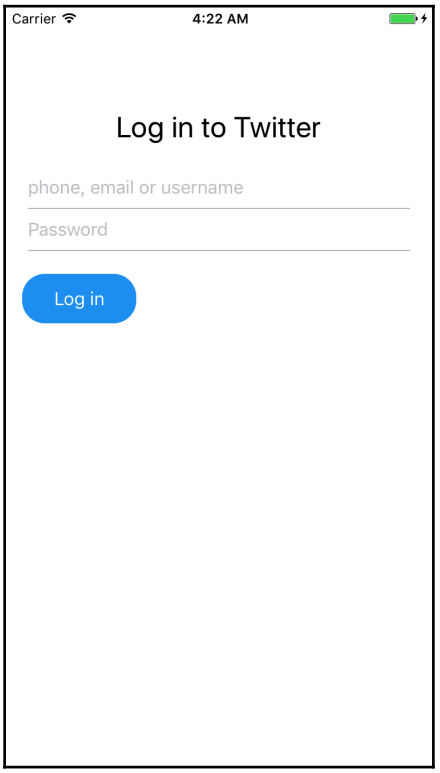
It all starts with SplashScreen:



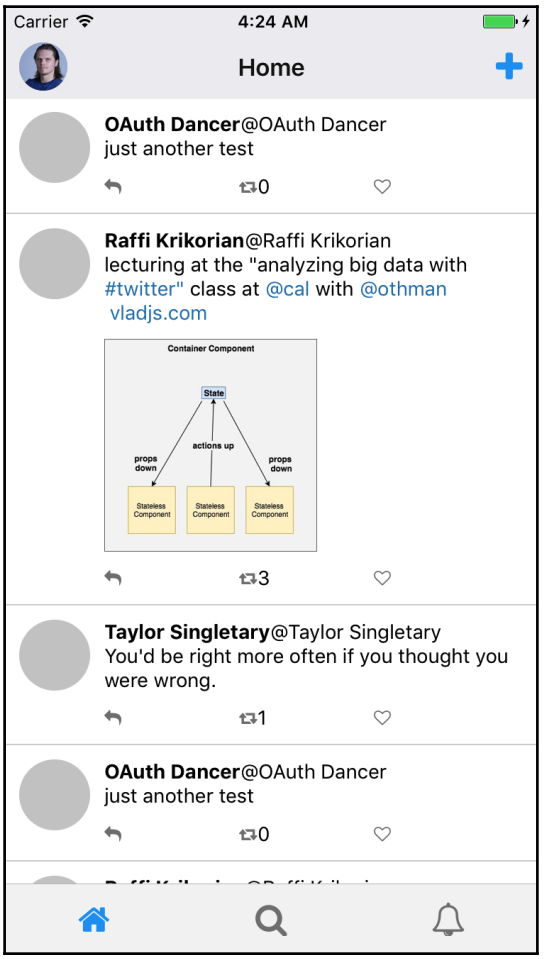
and Log in screen if the user is logged out:



As mentioned earlier, instead of implementing the Signup/Signin functionality, we will keep an app simpler just by creating a Login screen:

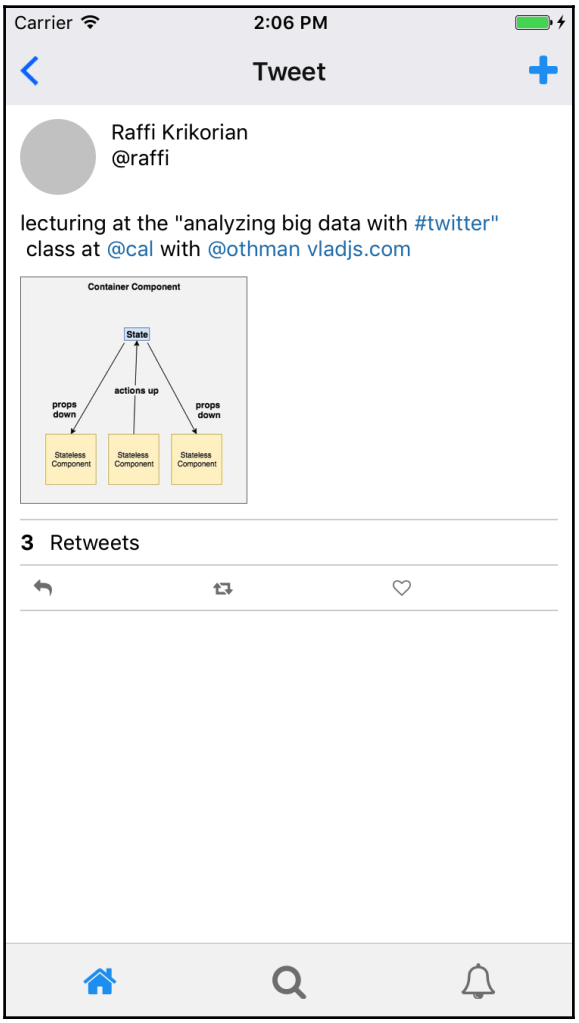


Then, after completing **Log in**, we will get the following screen:



As you can see, we have only three tabs in our app: Home, Discover, and Notifications.

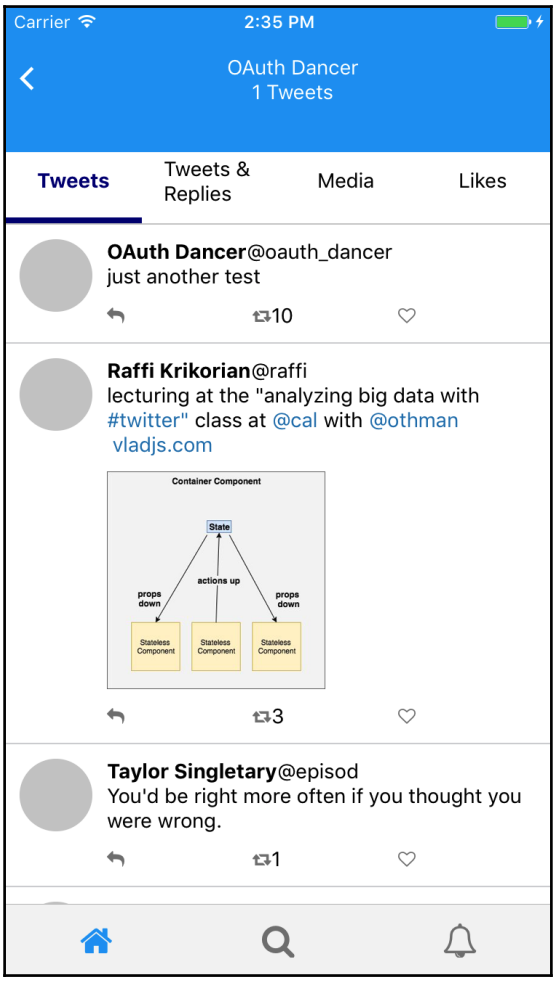
By clicking on a **Tweet**, we will get to the **Tweet** screen:



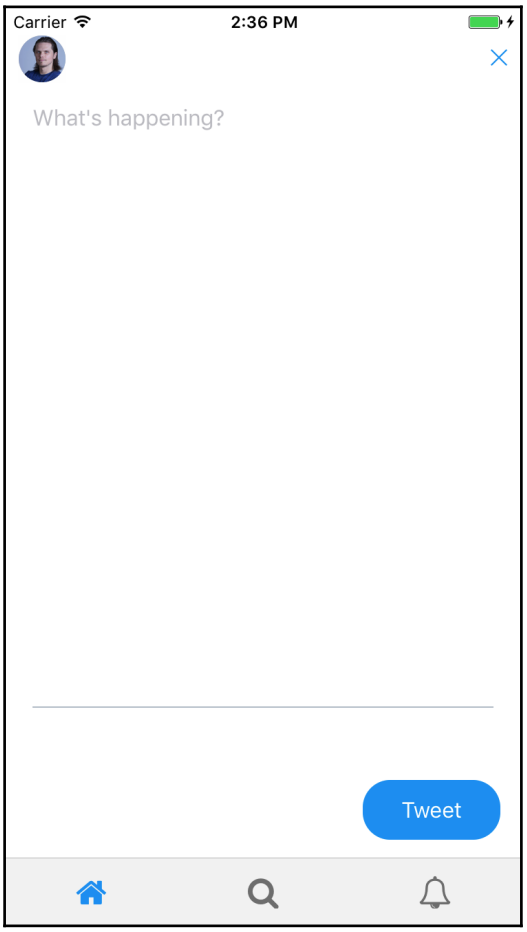
As you can see, you have a back button instead of a profile picture here, speaking of which, you will get to the Profile page with relevant parameters passed through navigation by clicking on the profile picture.

Accessing a profile can be done by clicking on the profile picture. Accessing a profile other than yours can be done by clicking on the profile picture or on the @ signed link.

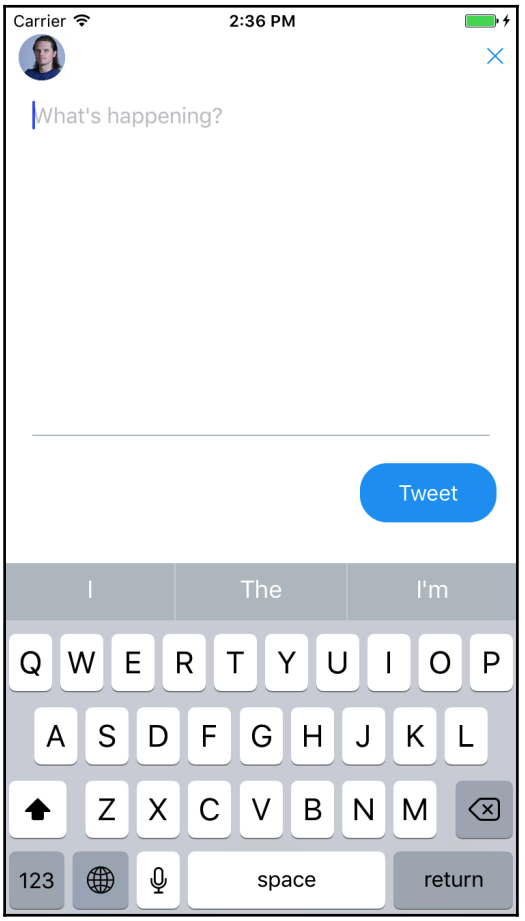
The Profile page will look like this:



In addition to these screens, we also want an ability to post a tweet. This can be done by clicking on the new **Tweet** button in the top-right corner. When clicking on this button, the following screen will appear:

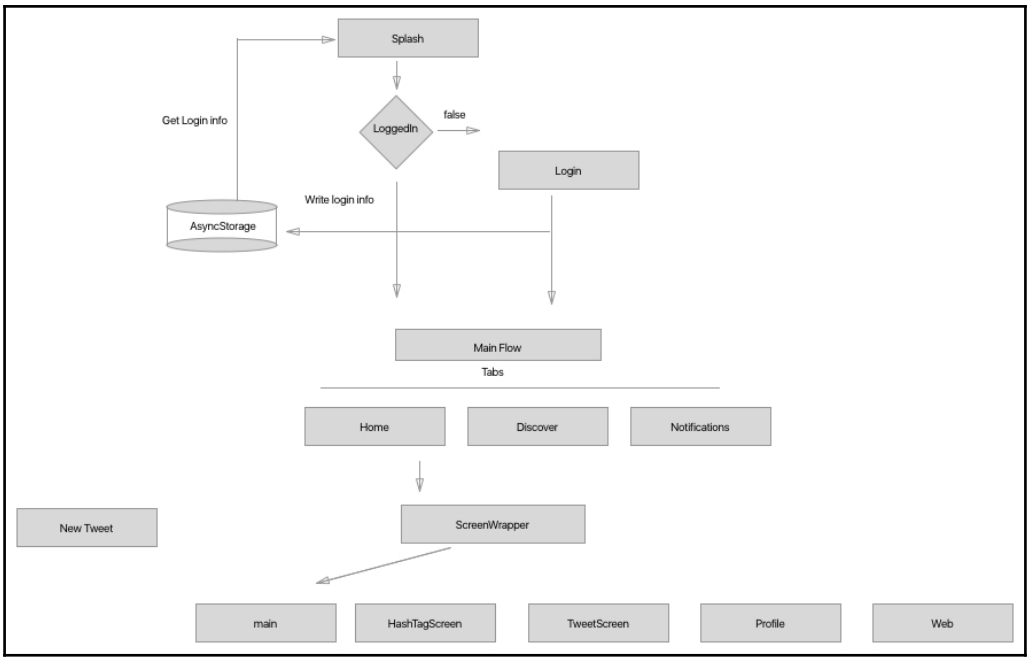


When clicking on the TextInput field, we will have a keyboard sliding up, pushing the **Tweet** button up:



Now you probably feel that it can get complicated, so what we will do now is open Twitter on our phone and map all behaviors that are in the scope of our application.

At the end, we will get the following diagram:



Let's describe what we have here:

- Our application starts with SplashScreen. During splash screen, we retrieve an `auth` token from `AsyncStorage` and check whether our user is logged in.
- If the user is logged out, we navigate to the Login screen. After the user clicks on submit, an `auth` key is written to `AsyncStorage`, and the user gets to the Main flow; the same happens if the user is already logged in.
- When we get to the Main flow, we have three tab components; however, we have similarities between them. Each one has a navigation header that has minor changes between each screen. That's the reason we define a ScreenWrapper, which will be responsible to serve the rest of our screens. Essentially it will be able to navigate between a timeline of tweets, tweets filtered by hashtag, screen for single tweet, profile screen, and a screen for displaying links.
- A new tweet is a separate component. At the beginning, it will be implemented as a sibling of the new tweet button and later on, when connecting to Redux or MobX, it will be moved to a higher level in the hierarchy.
- In addition, we have Tweet and Profile pages.
- As you can see from the screens before, you can navigate to and from these screens.

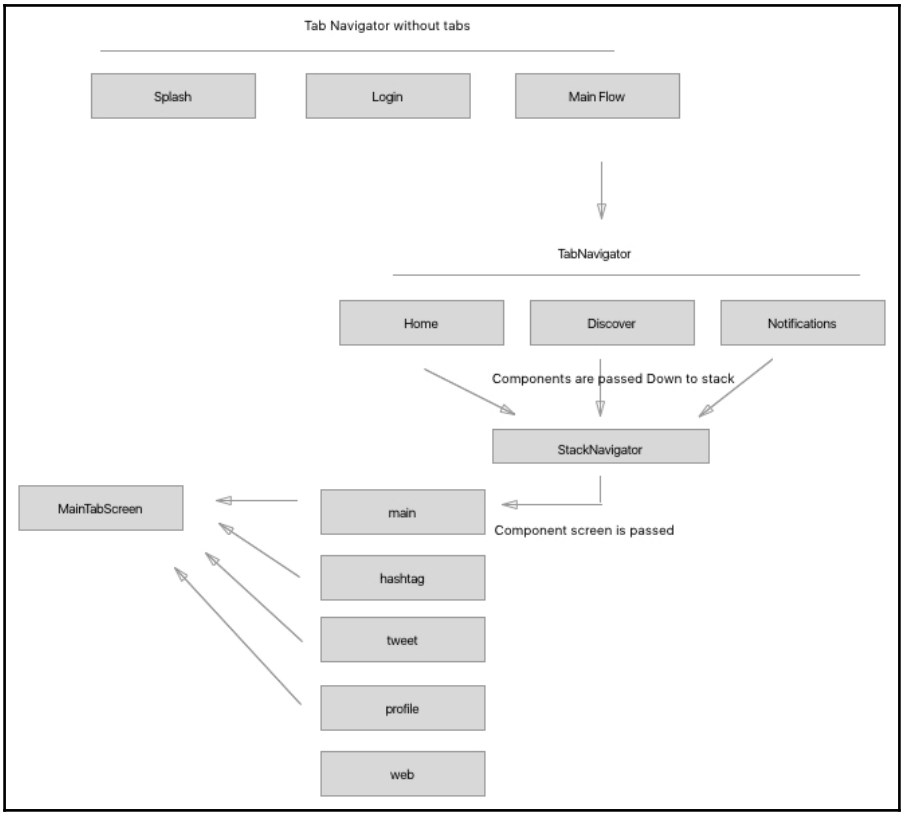
Setting up functional navigation and a wireframe for your application

Now, it's time to write some code, but let's create our application first. We will use `create-react-native-app` for that, and we will eject from it later on when we add the Twitter client API that has native code in it:

```
create-react-native-app twitterClone
```

Now, it's time to set some folder structure. We will create the `src` folder and put the `assets`, `components`, `config`, `core`, `screens`, and `utils` folders there.

Now, let's think of the navigation structure before diving into implementation:



Let's take a look at what is going in here.

First of all, SplashScreen, Login, and Main flow can be grouped by lack of one functionality; none of them needs the back functionality. You don't go from Login to Splash screen, or by getting to Splash screen, you don't need to press the back button to get to Login.

That's why they will be grouped by `TabNavigator`, though without tabs.

Let's start creating this navigation from our `App` component:

```
import React from "react";
 import { TabNavigator } from "react-navigation";
 import routesConfig from "./src/config/routes";

 const AppNavigator = TabNavigator(routesConfig, {
   navigationOptions: {
     tabBarVisible: false
   }
 });

 export default class App extends React.Component {
   render() {
     return <AppNavigator />;
   }
 }
```

We are using `TabNavigator` and passing the `tabBarVisible` prop to it to disable the tabs visibility completely.

After the `Root` component is set, it's time to set up `routesConfig`, which will hold our application navigation settings. At first, let's set up two screens--Splash and Login--and then we will continue to the Main flow:

```
const routes = {
 splash: { screen: Splash },
 login: { screen: Login }
};
```

Splash and Login will be our new screens, which we will put in the `screens` folder.

SplashScreen

Our SplashScreen view is fairly simple. Basically, it's just a view with an image:

```
<View style={styles.container}>
   <Image style={styles.logo} source={logo} />
 </View>
```

However, its main purpose is to load user data from `AsyncStorage` and, based on this data, to decide whether we should be redirected to the Main flow or to Login screen.

At this stage, we won't be dealing with the real data, only mocks, so we assume that we will always be redirected to Login. Even though we will still create proper authentication flow, we are laying the basis for future integration with Twitter API and real authentication. We will add this to the `componentDidMount()` function and navigate using the `props.navigation.navigate` function, which is passed from main `TabNavigator` we created in `config/routes.js`:

```
async componentDidMount(){
   const { navigation } = this.props;
   try {
    const authToken = await
AsyncStorage.getItem('@TwitterClient:authToken');
    if (authToken !== null){
      this.props.navigation.navigate("mainFlow");
    } else {
      setTimeout(() => {
      this.props.navigation.navigate("login");
     }, 3000)
    }
   } catch (error){
    setTimeout(() => {
      this.props.navigation.navigate("login");
    }, 3000);
   }
 }
```

The Login screen

Now, it's time to create the Login screen. We won't create the whole screen; instead, we will only add a button to navigate to our `mainFlow` since at this stage, we will be dealing with setting up our navigation:

```
    <Button style={styles.login}
      borderRadius={20}
```

```
        backgroundColor="#1DA1F3"
        onPress={() => this.props.navigation.navigate("mainFlow")} />
```

Note that button props are quite different. That's because in our app, I will be using the `react-native-elements` project for several reusable components, such as buttons and avatar. Even though `react-native-elements` depends on `react-native-vector-icons`, which in turn depends on native modules and linking, you may ask how exactly I can use it with **create-react-native-app** (**CRNA**) without ejecting. The answer is that the exponent (which is behind CRNA project) also depends on `react-native-vector-icons` and therefore using `react-native-elements` inside the exponent or CRNA project is permitted. Regarding ejecting, we will check out how it's done later. Make sure to follow the process.

Main flow

Earlier, when we created a flowchart of what our navigator structure will look like, we defined our Main flow as `TabNavigator`, which has a `ScreenNavigator` inside each tab. The reason we did so is because on every screen, there are some flows that needed back functionality, but always in the context of the same tab. For example, on the Home screen, clicking on a tweet will open a tweet, but will show a back button. If instead of clicking on a back button, I choose the Discover tab and then get back to the Home tab, I expect to be on the same "level in the stack navigator" as I was earlier.

Let's go to our `routes.config` and add the `mainFlow` screen:

```
mainFlow: {
  screen: TabNavigator({
     home: { screen:
       MainScreenNavigator(Home, 'home')},
     discover: { screen:
       MainScreenNavigator(Discover, 'search') },
     notifications: { screen:
       MainScreenNavigator(Notifications,'bell-o')}
   }, {
   swipeEnabled: false,
   animationEnabled: false,
   tabBarOptions: {
     showLabel: false,
     activeTintColor: '#1DA1F3',
     inactiveTintColor: 'gray'
    }
  })
}
```

Here, we define a nested `TabNavigator` and provide it with several styling options regarding how to display tab icons and active tab icon colors.

Here, you can note that every screen is basically a result of this:

```
MainScreenNavigator(ComponentName,iconName)
```

Now, let's take a look at the definition of the `MainScreenNavigator` function:

```
const MainScreenNavigator = (Component,tabIcon) =>
  StackNavigator({
    main: { screen: TabScreen(Component, tabIcon)},
    hashtag: { screen: TabScreen(Hashtag, tabIcon) },
    tweet: { screen: TabScreen(Tweet, tabIcon) },
    profile: { screen: TabScreen(Profile, tabIcon) },
    web: { screen: WebBrowser }
 });
```

So, what's happening here? For every tab screen, there is the main screen represented by "main" and hashtag, and profile and web screens. You can think of it as every tab screen having its own separate stack.

The idea of using the `MainScreenNavigator` function is for keeping the code more DRY (Don't repeat yourself) and clean. You can note that we pass the `Component` function, which is relevant only for the main screen. It's the actual main tab component and `tabIcon`. The `tabIcon` name is used for displaying tab icon by this name. This is done to define all our configurations in one place.

Then, there is the `TabScreen` function. The concept of `TabScreen` is the concept of higher order components, very famous in React. The idea is to create a function that takes some configuration and returns a component that uses this configuration:

```
import React from 'react';
 import { View, Text } from 'react-native';
 import { NewTweetButton } from "../components/NewTweetButton";
 import { Icon } from 'react-native-elements';

 export const TabScreen = (Component, tabIcon) =>
   class MainTabScreen extends React.Component {
     static navigationOptions = ({ navigation }) => ({
       ...Component.navigationOptions({ navigation }),
       headerRight: <NewTweetButton navigation={navigation}/>,
       tabBarLabel: null,
       tabBarIcon: ({ tintColor }) => (
         <Icon name={tabIcon} type="font-awesome" color={tintColor}/>
       )
```

```
        });

        render() {
          return <Component navigation={this.props.navigation} />
        }
    };
```

So, what do we have here? We get `Component` and `tabIcon` and in the render function, we render the component that is passed along with the navigation prop. However, that's not all; there are lots of `navigationOptions` applied here. You can treat the `TabScreen` Higher Order Component as basically a layout for our tab page. In our `navigationOptions`, we define common behavior and merge with `Component.navigationOptions` specific behavior. This way, every screen will have its defined common header and tab configurations, but will be able to have it's own configurations.

All we are left with is to create the actual screens with its options. We will take a look at the specific screens in the next section and will see what exact navigation options we will have for each screen and how they are different.

Home

The Home screen will have all our timeline tweets. It will have a Home title in Navbar and profile avatar to the left. Also, in options, we will specify that there won't be "Home" written near the back button if we go down the stack of our Navigator. This means that when we will link our Tweet, we will see only the back button without "Home" written near it:

```
static navigationOptions = ({ navigation}) => ({
   title: "Home",
   headerBackTitle: null,
   headerLeft: <ProfileHeaderButton navigation={navigation} />,
 });
```

Here and in several other spaces, we will define `ProfileHeaderButton` as a `headerLeft` component.

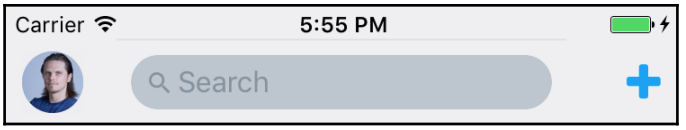
This corresponds to the user avatar in the top-right corner we will be looking at later. By defining `headerBackTitle: null` we tell `StackNavigator` not to display the previous screen name near the back button.

Discover

This screen will not be implemented in our client app, but let's style a header similar to how it looks in the real *Twitter* app:

```
static navigationOptions = ({ navigation }) => ({
   headerTitle: <SearchBar
                  lightTheme
                  round
                  containerStyle={styles.search}
                  placeholder="Search" />,
   headerLeft: <ProfileHeaderButton navigation={navigation} />,
   tabBarIcon: () => (
     <Icon name="search" />
   )
});
```

This tab, instead of having a title, will have a search bar from `react-native-elements`:



Notifications

The Notifications screen won't be implemented, so we won't dive into its implementation. You can try to implement it yourself. If that's your choice, Notifications need to display a notifications list similar to how the Home screen displays tweets.

Profile

The Profile screen won't have a header. Instead, it will have a custom header with almost the same functionality as a regular header:

Tweet

On the Tweet screen, we will display only the Tweet title and back button. So, `navigationOptions` will be pretty straightforward here:

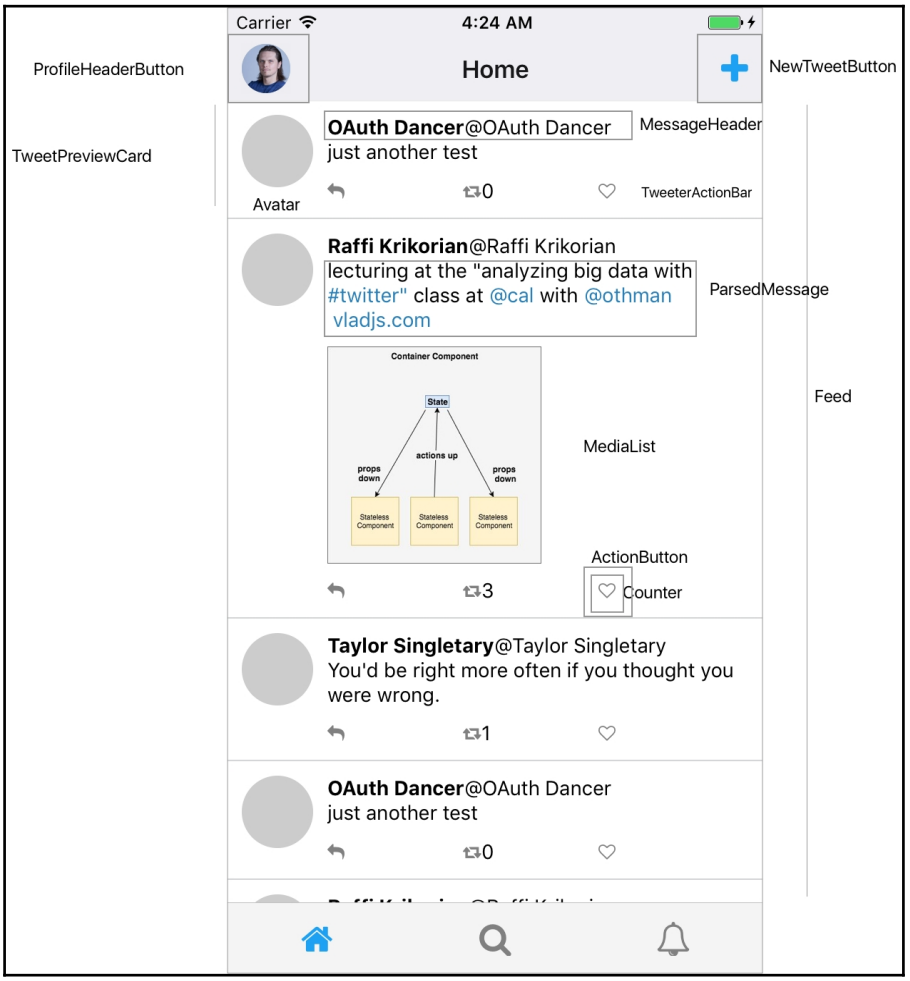
```
static navigationOptions = ({ navigation }) => ({
   title: "Tweet"
 });
```

We will have only the title and the rest of the options will fall back to the default ones defined in `TabScreen`.

Mock data and style application screens including animations

So now, after creating the basic navigation wireframe, let's style our screens accordingly. In order to do so, we need to get real data from the Twitter API, so our mocks will match the real use case once we connect our app to the Twitter API. I've started with a simple mock by accessing `https://dev.twitter.com/rest/reference/get/statuses/home_timeline` and pasting the example data inside the `Home` component. In addition, I also added a media key with the corresponding media array to the entities object. This was found after some searching for the media API in Twitter dev docs.

So now, after creating the data mock, let's take a look at our app and divide it by components:



ProfileHeaderButton is fairly simple. It's basically an avatar from `react-native-elements` with the following:

```
onPress={() =>
  navigation.navigate("profile")
}
```

NewTwitterButton, on the other hand, is much more complicated, so we will leave it for the end of this section.

Our `Home` component will look very simple:

```
<View style={{ flex: 1, backgroundColor: 'white'}}>
   <Feed data={this.state.data} navigation={navigation}/>
</View>
```

In fact, all the `Feed` components will use `Feed` with data prop and navigation to render the same structure--a `FlatList` of `TwitterPreviewCards`:

```
export const Feed = ({ data, navigation }) => (
    <FlatList data={data}
      renderItem={({ item }) => (
        <TweetPreviewCard key={item.id_str} navigation={navigation}
          data={item} />
      )}
      keyExtractor={(item, index) => `tweet-${index}-${item.id}`}
    />
)
```

`TwitterPreviewCards` will be constructed from `Avatar`, `MessageHeader`, `ParsedMessage`, `MediaList`, and `TweetrActionBar`:

```
export const TweetPreviewCard = ({ navigation, data }) => (
  <TouchableOpacity
    onPress={() => navigation.navigate("tweet",data)}>
      <View style={styles.container}>
         <Avatar
           medium rounded source={{ uri:
           data.user.profile_image_url }}
           onPress={() =>
             navigation.navigate("profile", { user: data.user })}
         />
         <View style={styles.tweetView}>
           <MessageHeader data={data}/>
           <ParsedMessage data={data} navigation={navigation}/>
           <MediaList media={data.entities.media} />
           <TweeterActionBar
             replyAction={() => {
               navigation.navigate("tweet",{
                 startReply: true, ...data
               })
             }}
             favorited={data.favorited}
             counters={{
               retweet: data.retweet_count,
               likes: data.favourites_count
             }}
           />
```

```
            </View>
          </View>
      </TouchableOpacity>
  );
```

`ParsedMessage` will parse a string by regex using a helper method I wrote (you can check the code from the repository with code samples) and wrap everything in the `Hyperlink` component taken from `react-native-hyperlink`.

`TweeterActionBar` is a view with several `ActionButtons`, and `ActionButton` is a `TouchableOpacity` around the `Counter` component.

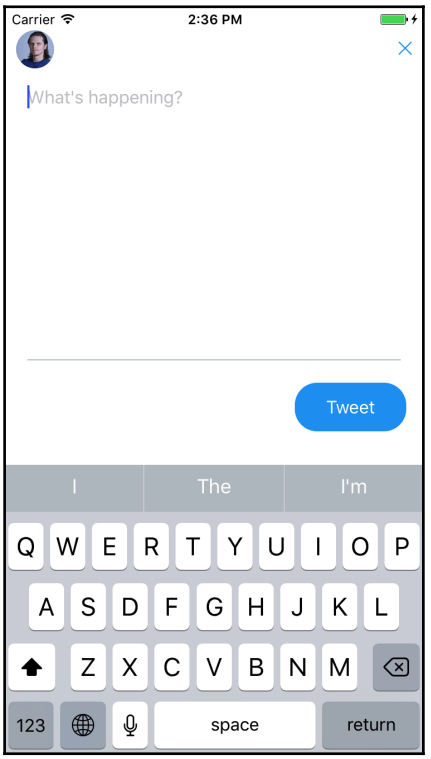
In the `Profile` component, we use the same reusable `Feed` components and wrap them inside `ScrollableTabView`, which we brought from the `react-native-scrollable-tab-view` package.

Finally, we need to create a **New Tweet** functionality. Here, we will have a basic `slideOut` animation. While we can use several techniques, both using external packages or using Animated API, we will use core React Native API to achieve this.

`NewTweetButton` is a plus button on the right and, in terms of code, it will look like this:

```
export class NewTweetButton extends React.Component {
   state = {
     showNewTweet: false
     }
   render(){
    return (
    <View style={{ marginRight: 10 }}>
    <TouchableOpacity onPress={() => this.setState({
    showNewTweet: true })}>
     <Icon
      name="plus"
      type="font-awesome"
      color={this.props.color || "#1DA1F3" } />
      </TouchableOpacity>
      { this.state.showNewTweet &&
      <NewTweet
       navigation={this.props.navigation}
       close={() => this.setState({ showNewTweet:
       false})}/> }
      </View>
      )
    }
}
```

Here, we show the `NewTweet` component once the button is clicked on, and will hide it once the `close` callback is called:



We want animation in `NewTweet`, so we will start by defining `Animated.Value` on our component:

```
transformY = new Animated.Value(Dimensions.get('window').height);
```

We want it to be animated on Mount, so we will use `componentDidMount` to start an animation:

```
componentDidMount() {
  const { height } = Dimensions.get('window');
  Animated.timing(this.transformY, {
    toValue: 0,
    duration: 300,
    useNativeDriver: true
  }).start();
}
```

We use `nativeDriver` because we want animations to be passed to the UI thread and be more performant.

In order to animate the view transform, we must wrap our component in `Animated.View`:

```
// inside a render function
const { width, height } = Dimensions.get('window');
return (
   <Animated.View ref="view" style={[
      styles.newTweetContainer,
      {
         width,
         height,
         transform: [{ translateY: this.transformY }]
      }]}
    >
//...
```

In order to make the Tweet button appear even when our keyboard is open, remember to use `KeyboardAvoidingView` (you can check the example code for specific usage).

Basically, creating a wireframe based on mocks with actual data gives you the ability to style your screen according to the desired look and only deal with data and complex state management later. The idea of creating such mocks originates from TDD and even though we haven't started creating this app using Test driven development methodologies, because of time and space constraints (only a small portion of applications fit into the book), it's highly advisable to do so. Even though TDD methodology says that you should start writing tests, I personally advise you to do the same brainstorming and high-level design as we did at the beginning of this chapter. This is crucial so that you won't make architectural mistakes and find challenges early.

Bringing Redux or MobX to your application and moving data mocks to a centralized state

Redux and **MobX** are two important state management systems we discussed in the earlier chapters. In this example, we will look at both the implementations.

Redux

Before switching to Redux, let's define the shape of our state.

We have feed data, the current user data (which was not implemented in mocks yet), and navigation data.

So generally speaking, we are talking about three reducers here: `feedReducer`, `userReducer`, and `navReducer`.

Let's define each one of them, but let's create a proper folder structure first.

Inside our core folder, we will put `actions folder`, `reducers folder`, `rootReducer.js`, and `store.js`.

Let's, first of all, define our reducers:

```
import { actionTypes } from '../actions';

 const initialState = {
     data: []
 };

 export default function feedReducer(state = initialState, action) {
   switch (action.type) {
     default:
       return state;
   }
 }
```

User reducer will look the same, but it will have a different `initialState`:

```
const initialState = {
     userData: {}
 };
```

`navReducer` will be more complicated. In order to create it, we first need to separate our `AppNavigator` from our `App.js` file. Let's put it under `core/AppNavigator.js`:

```
import { TabNavigator } from "react-navigation";
 import routesConfig from "../config/routes";

export const AppNavigator = TabNavigator(routesConfig, {
 navigationOptions: {
  tabBarVisible: false
  }
});
```

There's nothing new here. Now, let's create `navReducer`:

```
import { AppNavigator } from '../AppNavigator';
 const initialState = AppNavigator.router.getStateForAction(
   AppNavigator.router.getActionForPathAndParams("splash")
 );
 const navReducer = (state = initialState, action) => {
   const nextState = AppNavigator.router.getStateForAction(action,
   state);
   return nextState || state;
 };
 export default navReducer;
```

This is the same code used in the `https://reactnavigation.org/docs/guides/redux` `react-navigation guide`.

Now, it's time to combine reducers. Let's take a look inside our `rootReducer.js` file:

```
import { combineReducers } from "redux";
import feed from './reducers/feedReducer';
import user from './reducers/userReducer';
import nav from './reducers/navReducer';

 export default combineReducers({
  feed,
  user,
  nav
 });
```

Then, let's take a look at the `store.js` file. Note that we install logger and thunk middleware for introspection and async actions handling the following:

```
import { createStore, applyMiddleware } from "redux";
import thunk from 'redux-thunk';
import logger from 'redux-logger';
import rootReducer from "./rootReducer";

export default createStore(rootReducer, applyMiddleware(logger,thunk));
```

What's left is to wire everything up to the root component. Our `App.js` will be changed a bit:

```
import React from "react";
 import { Provider } from 'react-redux';
 import { AppNavigator } from './src/core/AppNavigator';
 import { addNavigationHelpers } from 'react-navigation';
 import { connect } from 'react-redux';
 import store from './src/core/store';
```

```
const AppWithInternalState = connect(({ nav }) => ({ nav }))(({ nav })
=> (
  <AppNavigator navigation={
    addNavigationHelpers({
      dispatch: store.dispatch,
      state: nav
    })}/>
));

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

Here, we change the `AppNavigator` dispatch and state function to be connected to the Redux store instead of internal state.

Refactoring all actions

Now, it's time to do some refactorings. First, we will remove mock data from the `Home` component and alter the `Feed` component to be in charge of data retrieval through emitting Redux actions. The `Feed` component will be used in the `Home` component in the following matter:

```
<Feed endpoint="statuses/home_timeline" navigation={navigation} />
```

As you can see, it's generic enough to retrieve any feed from any endpoint. The feed will change to the following:

```
class FeedContainer extends React.Component {

    componentDidMount() {
        this.props.getFeedData(this.props.endpoint);
    }
    render(){
        const { data, navigation } = this.props;
        return (
            data.length > 0 ?
            <FlatList data={data} renderItem={({ item }) =>
                <TweetPreviewCard
                key={item.id_str}
```

```
                    navigation={navigation}
                    data={item} />}
                    keyExtractor={(item, index) =>
                    `tweet-${index}-${item.id}`}
                /> :
                <View style={styles.container}>
                    <ActivityIndicator />
                </View>

            )
        }
    }
```

It will be exported in the following way:

```
export const Feed = connect(({ feed }) => ({ data: feed.data }), {
getFeedData })(FeedContainer)
```

Our feed is a connected component, and we use `getFeedData` to retrieve feed mock. We should create a corresponding action though and add the following statement to `feedReducer`:

```
case actionTypes.GET_FEED_DATA_SUCCESS:
 return { ...state, ...action.payload }
```

The action will look like this:

```
export const getFeedData = (endpoint) => (dispatch) => {
     dispatch({
         type: actionTypes.GET_FEED_DATA,
         payload: {
             endpoint
         }
     })

     setTimeout(() => {
         dispatch({
             type: actionTypes.GET_FEED_DATA_SUCCESS,
             payload: {
                 data: []
             }
         })
     }, 2000)
 }
```

We are still dealing with mocks, so we simulate an API call using `setTimeout`.

Now, we will give an overview of what actions we should add to our app and their purpose:

- `getLoggedInfoFromAsyncStorage`: This action will be in charge of retrieving data from `AsyncStorage`, and if there is a token, it will initiate the `getUser` action:

```
export const getLoggedInfoFromAsyncStorage = () => (dispatch) => {
    dispatch({
        type: actionTypes.GET_LOGGED_INFO__STARTED
    })
    AsyncStorage.getItem('@TwitterClient:authToken')
    .then(result => {
        if (result !== null) {
            dispatch({
                type: actionTypes.GET_LOGGED_INFO__SUCCESS,
                payload: result
             })
            dispatch(getUser());
        } else {
            dispatch(
              NavigationActions.navigate({ routeName: "login" })
            )
        }
    }).catch(error => {
        dispatch({
          type: actionTypes.GET_LOGGED_INFO__ERROR,
          payload: error
        });
    });
}
```

- `getUser`: This action will retrieve user data and, upon completion, will redirect to `mainFlow`:

```
export const getUser = () => (dispatch) => {
  dispatch({
    type: actionTypes.GET_USER_DATA,
    payload: {
      user: '@VladimirNovick'
    }
  });
  setTimeout(() => {
    dispatch({
       type: actionTypes.GET_USER_DATA_SUCCESS,
       payload: {
```

```
            user: {}
          }
      });
      dispatch(NavigationActions.navigate({
        routeName: "mainFlow"
      }));
    }, 3000)
  }
```

- `login`: This action will log you in to Twitter using the provided credentials. On success, it will save data to `AsyncStorage` and redirect you to `mainFlow`:

```
export const login = (data) => (dispatch) => {
  dispatch({
    type: actionTypes.LOGIN,
    payload: data
  })
  setTimeout(() => {
    dispatch({
      type: actionTypes.LOGIN_SUCESS,
      payload: {
        user: {}
      }
    })
    dispatch(NavigationActions.navigate({
      routeName: "mainFlow"
    }));
  },300)
}
```

- `postTweet`: This action will post tweets to Twitter:

```
export const postTweet = (text, closeModalCallback) => (dispatch)
=> {
    dispatch({
        type: actionTypes.POST_TWEET,
        payload: {
            text
        }
    })
    closeModalCallback();
    setTimeout(() => {
        dispatch({
            type: actionTypes.POST_TWEET_SUCCESS,
            payload: {
                text
            }
        })
    }, 3000)
}
```

All actions are working and can be seen in the `twitterClientRedux` example app.

We already changed our feed reducer to update the state accordingly. What's left is to add a case statement to our user reducer, so it will look like this:

```
import { actionTypes } from '../actions';
const initialState = {
 userData: {}
};
export default function userReducer(state = initialState, action) {
  switch (action.type) {
    case actionTypes.GET_USER_DATA_SUCEESS:
      return { ...state, isLoggedIn: true, userData:
      action.payload.user }
      default:
      return state;
  }
}
```

MobX

Now, after we've seen how Redux can be hooked up to our app, let's take a look at how we can do this with MobX. First of all, as Redux has three reducers, we will have three stores in MobX: `NavigationStore`, `UserStore`, and `FeedStore`.

Let's set up the Navigation store. It will look like this:

```
import { AppNavigator } from '../AppNavigator';
 import { observable, action } from 'mobx';
 import { NavigationActions } from 'react-navigation';

 class NavigationStore {
     @observable.ref navigationState = AppNavigator
         .router
         .getStateForAction(NavigationActions.init({}))

     @action dispatch = (action, stackNavState = true) => {
         const previousNavState = stackNavState ? this.navigationState
         : null;
         return this.navigationState = AppNavigator
             .router
             .getStateForAction(action, previousNavState);
     }
   }
 export const navStore = new NavigationStore();
```

We define an observable reference to store our `navigationState`, and since we don't know the structure of our internal navigation state, nor do we want to write our routes twice, we will call `AppNavigator.router.getStateForAction` and pass it the `NavigationActions.init` function's result. This will construct our internal navigation state.

Then, we will add the dispatch function on `NavigationStore`. This action is similar to what we did with Redux.

At the end, we will export a new instance of our store and, by doing so, we will ensure that our store is a singleton.

We will put all our stores under the `core/stores` folder and export them from the `index.js` file:

```
export * from './NavigationStore';
 export * from './FeedStore';
 export * from './UserStore';
```

We will ensure that we run MobX in the strict mode. This mode won't let us modify observable state outside of actions:

```
useStrict(true);
```

Our app will look like this. Similar to Redux, however, instead of one store, there are several of them:

```
export default class App extends React.Component {
   render() {
     return (
       <Provider {...stores}>
         <AppWithInternalState />
       </Provider>
     );
   }
 }
```

Then, we use the MobX inject function to inject `navStore` to our props and then wrap the component with observer to make it reactive. Similar to Redux where we've added navigation helpers, we will do the same, but we will point our state and dispatch to the `NavigationStore` function this time:

```
const AppWithInternalState = inject("navStore")(observer(({ nav, navStore
}) => (
   <AppNavigator navigation={
     addNavigationHelpers({
```

```
          dispatch: navStore.dispatch,
          state: navStore.navigationState
        })} />
    )));
```

After moving navigation state to MobX, let's do the same with feed and user states. `FeedStore` will look like this:

```
class FeedStore {
     @observable data = [];

     @action
     setData(data){
         this.data = data;
     }

     @action getFeedData(endpoint){
         setTimeout(() => {
             this.setData(mockData)
         }, 2000);
     }

     @action postTweet(text, closeModalCallback){
         closeModalCallback();
         setTimeout(() => {
         },2000)
     }
 }

 export const feedStore = new FeedStore();
```

Th `UserStore` will have the same initial state we had with Redux as well as the same actions; however, we don't need to throw several actions when doing an API call or a change this time:

```
class UserStore {

     @observable userData = {};
     @action tryToLogIn(){
         AsyncStorage.getItem('@TwitterClient:authToken').then((result)
         => {
             if (result === null) {
                 this.getUser()
             } else {
                 navStore.dispatch(NavigationActions.navigate({
                 routeName: "login" }))
             }
         }).catch(result => {
```

```
                navStore.dispatch(NavigationActions.navigate({ routeName:
                "login" }))
            })
        }

        @action
        setUserData(data){
            this.userData = data
        }

        @action login() {
            setTimeout(() => {
                this.setUserData(mockData);
                navStore.dispatch(NavigationActions.navigate({
                    routeName: "mainFlow"
                }));
            }, 300)
        }

        @action.bound getUser(){
            setTimeout(() => {
                this.setUserData(mockData);
                navStore.dispatch(NavigationActions.navigate({
                    routeName: "mainFlow"
                }));
            }, 3000)
        }
    }

    export const userStore = new UserStore();
```

At the end, to connect your app components to MobX, all you need is to search Redux connect we added earlier and instead of it, add inject with the corresponding store and observer. Also, keep in mind that data and actions reside on an injected store object and not directly on props. For example, this will be the change for the `Feed` component:

```
@inject('feedStore')
 @observer
 export class Feed extends React.Component {

     componentDidMount() {
         this.props.feedStore.getFeedData(this.props.endpoint);
     }
     render(){
         const { feedStore: { data }, navigation } = this.props;
```

You can overview the whole example from the books repository under the `twitterClientMobX` folder.

Using the Twitter API to work with real data

Now, it's about time we connect our app to the actual Twitter API, but before we do that, we need to eject from CRNA. Connection to Twitter API will be done by the npm package with native modules dependencies, and we will need to link them properly:

```
npm run eject
```

Now, the app will fail if you run it, because as I mentioned earlier, `react-native-elements` has a dependency on `react-native-vector-icons`. When we worked inside an environment with expokit installed, all was good, but as soon as we ran eject into react native app, we need to install and link this dependency.

So, let's run this command:

```
npm i -S react-native-vector-icons
```

And then link vector icons properly by running:

```
react-native link
```

Now we will be able to use icons in the same way we used them earlier.

In our next step, we will switch part of the mocked state to the real one. You are welcome to fork the repository and play around with the code to implement the whole Twitter client functionality. As an example, we will be implementing several steps here.

Lots of apps and, especially social providers, use the Ouath 1 and Ouath 2 mechanism for authenticating. This means that we basically won't need the Login window since we will get it out-of-the-box.

What we will implement is authentication and user timeline retrieval.

In order to authenticate, we will use the `react-native-oauth` package.

Ensure that you follow through all the steps of setting it up. This package has quite a few native dependencies and requires a tedious process of wiring things up.

After everything is set, we will add another file to our project--`core/oauth.js`:

```
import OAuthManager from 'react-native-oauth';
 const manager = new OAuthManager('learningreactnative')
 manager.configure({
   twitter: {
     consumer_key: 'CONSUMER_KEY',
     consumer_secret: 'CONSUMER_SECRET'}
 })
 export {
     manager
 }
```

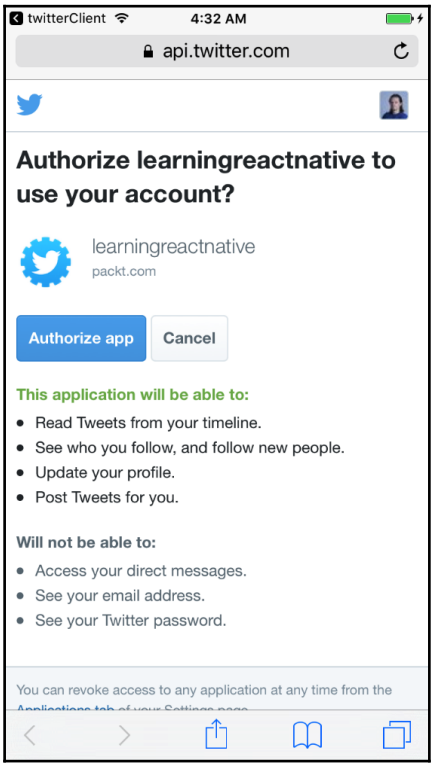
As you can probably see, we configure our auth manager with key and secret, which we have retrieved from the Twitter API dashboard. All these steps are also described in the `react-native-oauth` package.

Then, in `UserStore`, let's alter the `tryToLogIn` function:

```
@action tryToLogIn(){
   AsyncStorage.getItem('@TwitterClient').then((result) => {
   if (result !== null) {
     this.getUser()
    } else {
    manager.authorize('twitter').then(({ response }) => {
     AsyncStorage.setItem('@TwitterClient',
     JSON.stringify(response))
     this.getUser();
    });
   }
  })
 }
```

We check with `AsyncStorage` whether we are authorized and if not, we call the `manager.authorize` method.

This will open the following window if you are logged in to Safari, Twitter, and a window with the username/password fields if not:



When finished authorizing, the promise will be resolved, and we will retrieve user data. As you can see, we don't use the Login screen anymore, so we can remove it.

In our `FeedStore`, we will alter `getFeedData` to display real data instead of mocks:

```
@action getFeedData({ endpoint, params }){
    manager.makeRequest('twitter',
      https://api.twitter.com/1.1/${endpoint}.json`, {
      params
    }).then(resp => {
      console.log(resp.data);
      this.setData(resp.data.statuses || resp.data)
    }).catch(error => {
      console.log(error)
   })
 }
```

In the same way, you can wire up the rest of the actions to the real data. It's important that you understand the process of how to do it. Implementation details are described in Twitter API docs at `https://dev.twitter.com/docs`.

Summary

Congratulations, you completed everything you should know to start working with React Native. Even though there are lots of advanced concepts and some of them were covered only briefly in this book, the main idea is to give you the set of tools to work with as well as the best practices to create your next bestseller app. In this chapter, we combined the skills you gained through the book to walk you through the process of creating an app from design to final implementation. Well, it's not final. There is a lot of stuff to add to the app, but hopefully you've got an idea of how the process of structuring apps works. Also, it does not matter whether you are writing an app alone or you work as part of a team, the set of steps you learned in this chapter should help you create your next bestseller app. Let's overview what exactly we did in this chapter. We started by defining the requirements for our *Twitter client* app. Then, we continued to the high-level design of architecture and navigation. Then, we implemented a main navigation screen wireframe. In the next stage, we created data mocks that we used across the app to style everything and include some small animations. After getting this done, we looked at both Redux and MobX and looked at the process of moving your application state there. As you can probably clearly see, you should start with basic mocks and navigation and only when your state gets complex, should you move to a state management system. Finally, we used `react-native-oauth` to introduce Twitter authentication, persistence, and data retrieval for some parts of our application state.

So now you probably ask yourself--if you say that you've learned almost everything, what else is there? Well, we haven't finished yet. In the next chapter, we will give an overview various of techniques of distributing your app to the App Store and Play Market. So, get ready to learn about great tools and techniques to make your app deployment easy.

12
Deploying Your App to App Store or Google Play

Welcome to the twelfth chapter. Up till now, you have learned all the techniques needed to create amazing React Native apps, which are on a par with the best selling apps. We've covered almost every aspect of app development, from the beginning to create complex apps with animations, authentication, data retrieval, native APIs, and more. Now you can create your apps confidently, however, there is a small part missing. You know how to create amazing apps, but you don't know how to deploy them... yet. In this chapter, we will start by introducing you to the official process of submitting your app to iOS App Store and Android Google Play. We will walk through the basic steps you need to do to make it happen. If you come from a native development background, these steps are probably familiar to you, however, there are some nuances specific to React Native development. After we have covered these techniques, we will take a look at fastlane--an automation tool for continuous delivery. We will overview what it gives us and cover the basic steps to get started with it. Then, we will get to know another amazing service--Microsoft CodePush - a cloud tool that allows you to update your application in real time without the need of deploying. After finishing this chapter, you will know techniques and technologies to deploy your application confidently.

We won't deploy real applications in this chapter, however, we will overview the process and will give you all the knowledge you need to deploy your application.

So, let's summarize. Here is what you will learn in this chapter:

- Building your application for release
- Deploying iOS apps and how it's done in React Native
- Deploying Android apps and how it's done in React Native
- Introducing fastlane--automate your deployment workflow
- Get to know Microsoft CodePush and integrate it with your application

Deploying iOS apps and how it's done in React Native

Before diving into how to deploy the React native app, let's understand what the process of submitting regular iOS apps to the App Store is. A detailed explanation is available in Apple docs at:

`https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/Introduction/Introduction.html`.

But, let's overview the general steps to be familiar with the process:

- Joining the Apple Developer program
- Adding services to your app
- Preparing your app for distribution
- Testing apps on numerous devices and releases
- Submitting and releasing your app on the Store

In addition to these steps, along the process of deploying your app, you need to manage and maintain certificates, identities, and so on.

In React Native, the process of adding services to your app is a process that is done during the development phase when you add an external native package. When you do so, there will be an explanation in the package Readme of how to add a specific service.

All these steps, certificates, and identities can get really confusing, so let's take a look at the exact steps we should perform when we want to deploy our React Native app to the App Store.

Join the Apple Developer program

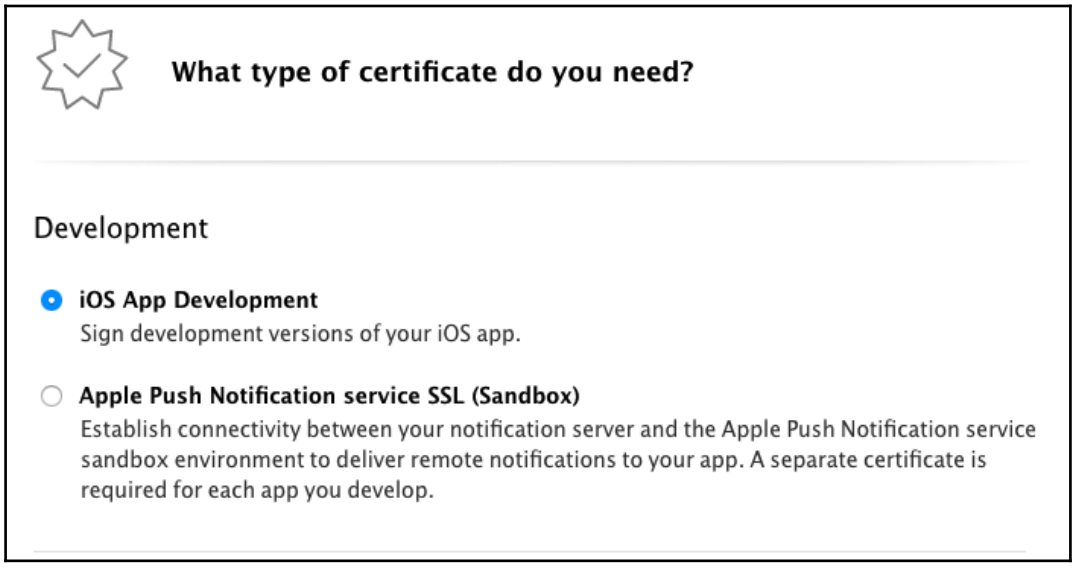
Well, you probably have done this before, when we've talked about running your application on an actual device, however, let's overview it once again.

Go to `https://developer.apple.com/account/` and access your Apple Developer dashboard. If you haven't joined the Apple Developer program yet, for this stage, it's crucial to do so. Before we submit our app, we want to build it for release and run it on an actual device.

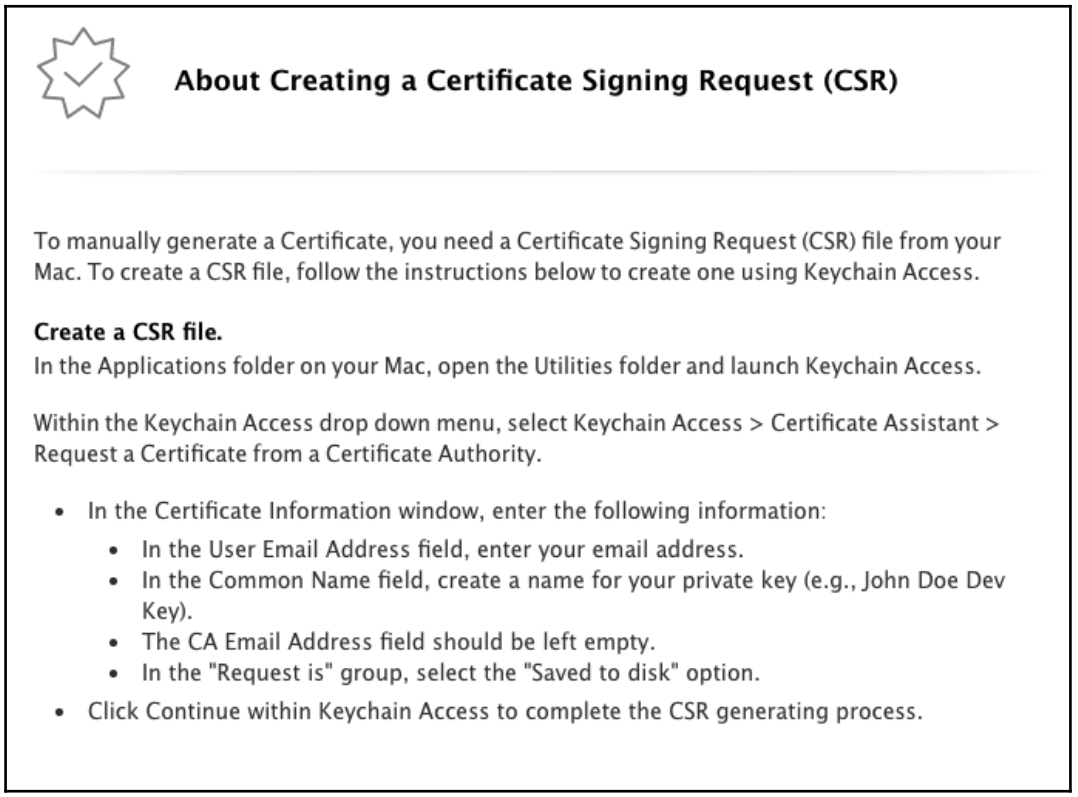
Creating a certificate for your device

To run applications on an actual iOS device, you need to create a certificate. This can be done by going to `https://developer.apple.com/account/ios/certificate/`.

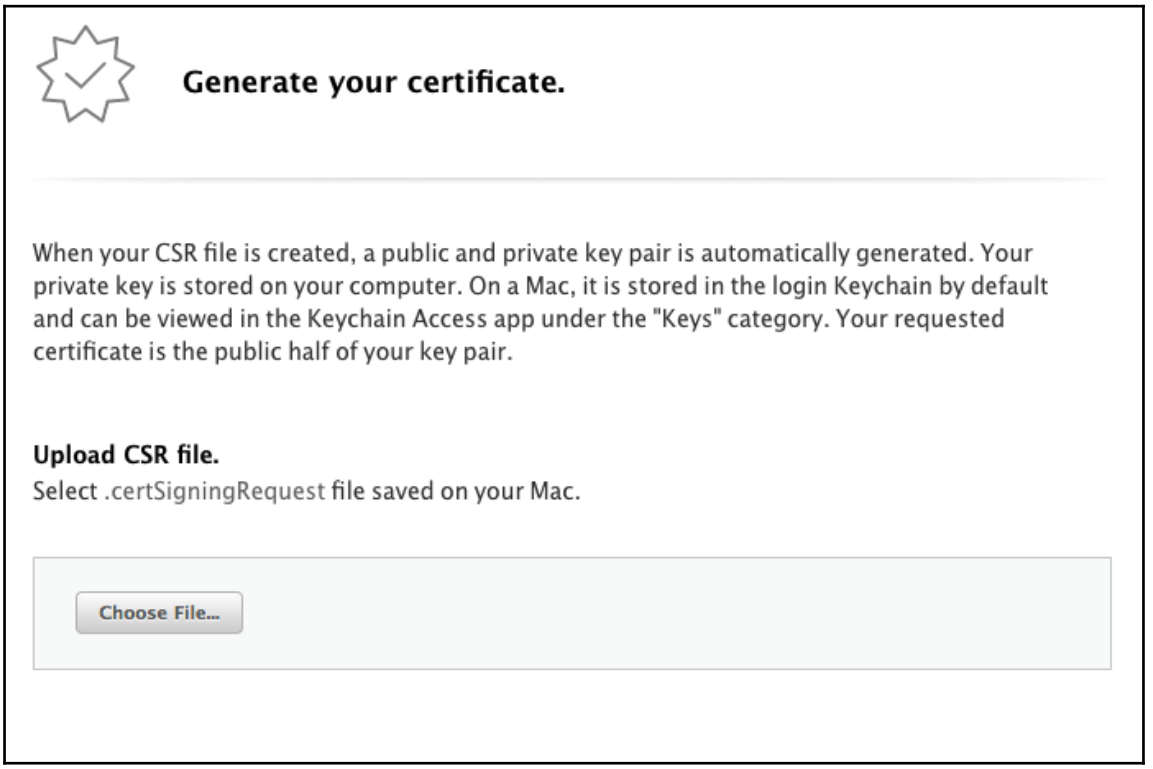
Then, choose Development from the panel on the left. You will be presented with the following screen:



You choose **iOS App Development** and hit **Continue**. You will be presented with the next screen:

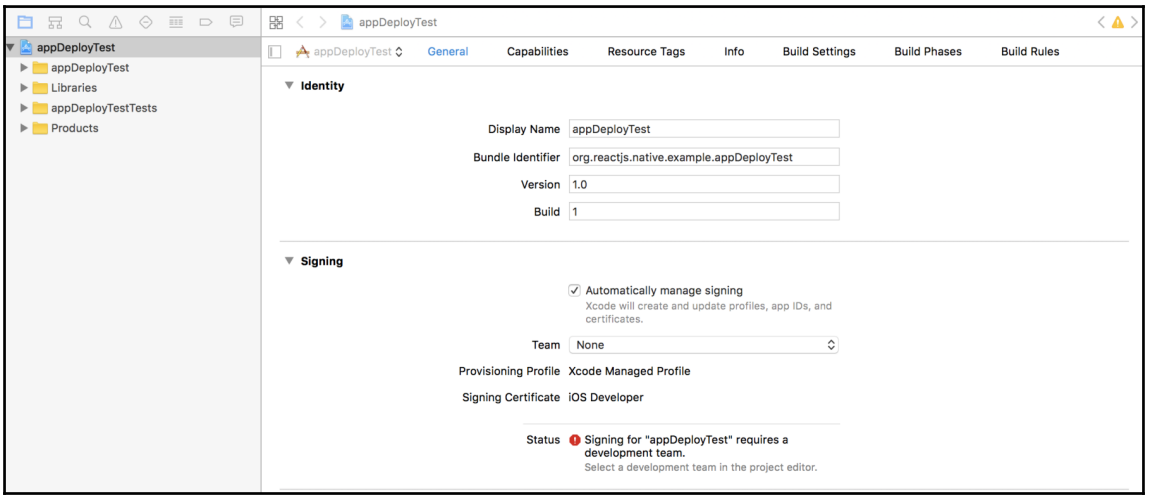


The following instructions get a CSR file from Keychain and upload a CSR file to the web interface, as seen in the following image:



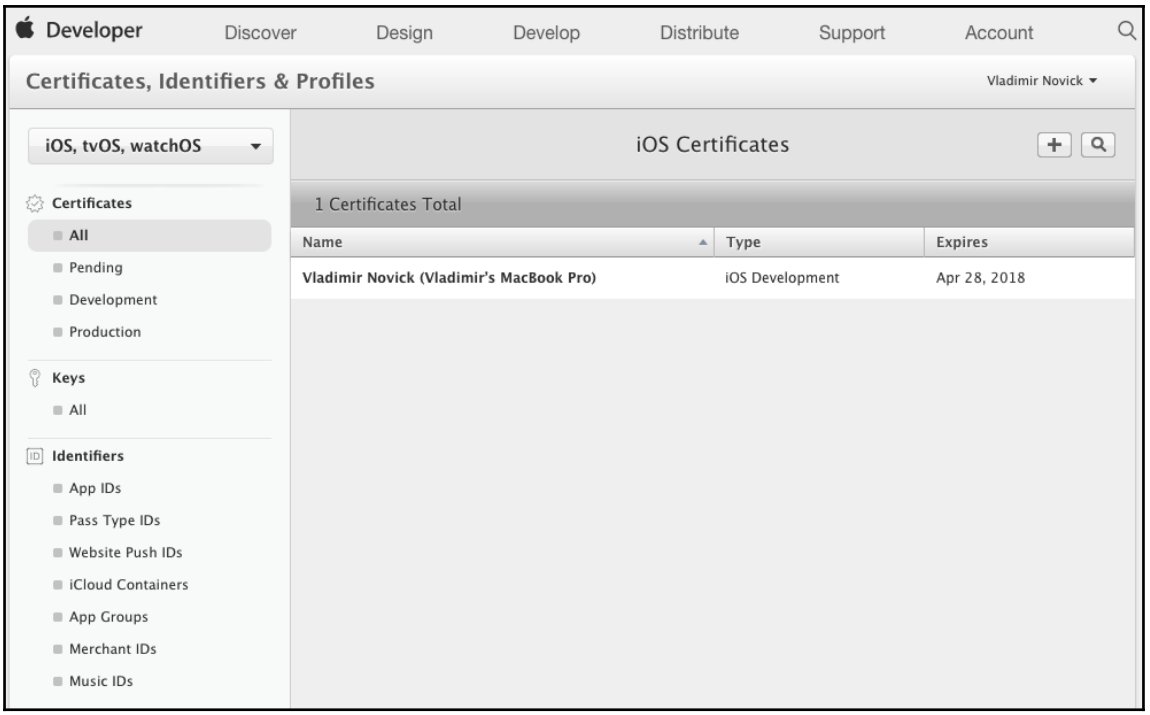
Signing your app to run on a device

After plugging in our iOS device via the USB cable and opening Xcode, select the project name, and under the **General** tab, you will see that the device requires signing:



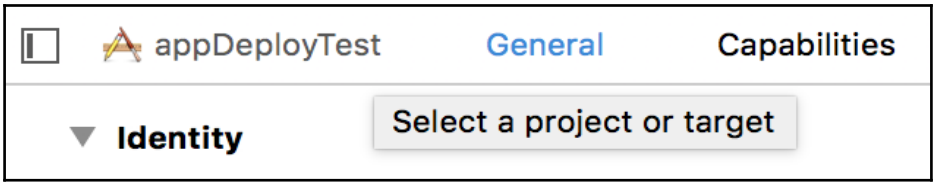
From here, I selected a certificate from the **Team** dropdown.

The certificate can be found at `https://developer.apple.com/account/ios/certificate/`:

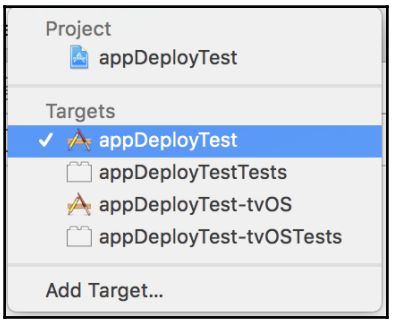


Also, don't forget to sign tests too. If you don't do so, building your app will result in an error.

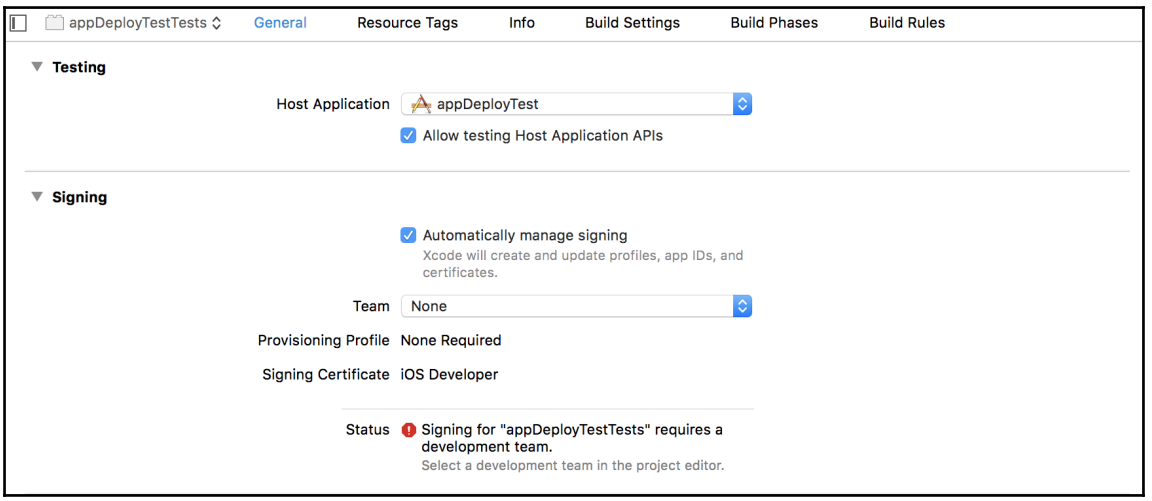
You can choose app tests from the drop-down menu on the left of the **General** tab:



By clicking on **appDeployTest**, you will be presented with the following menu:



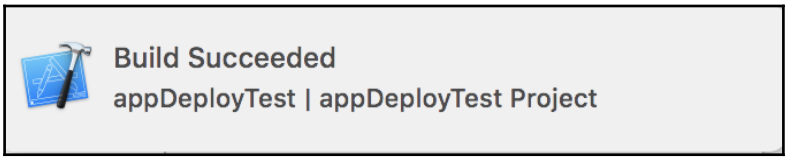
When selecting **Tests**, you will get the following screen, where you should also sign the correct team:



When selecting the correct certificate, you will see your device listed under the **Build Target** in the XCode toolbar:

Now, you can run and test it on a device.

You will get the following message and your app will open on your device:



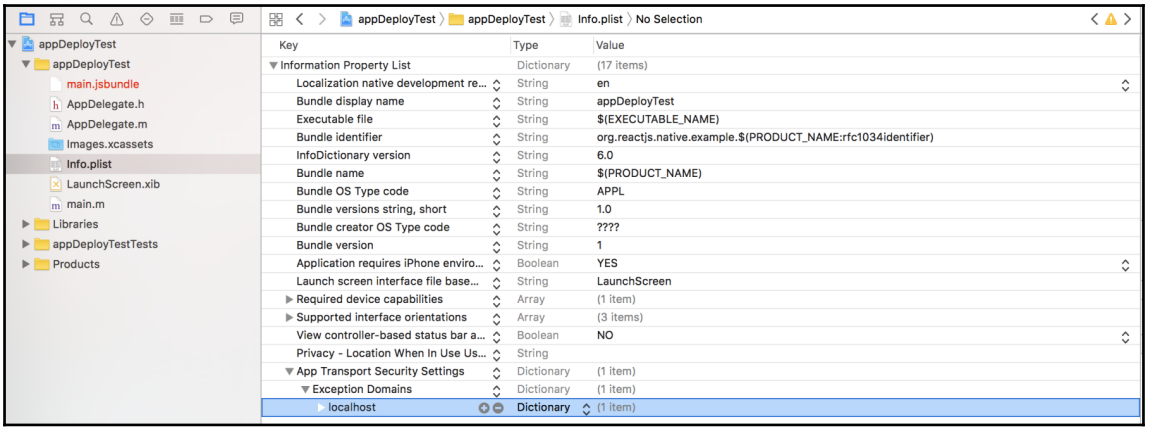
You get your app running. Sounds great, but now we want to deploy a release version.

Enable App Transport Security

By default, your React Native app is using the HTTP protocol for requests; however, after iOS 9, the **App Transport Security** (**ATS**) feature was introduced to iOS, and it basically limits you to sending and receiving data over HTTPS. ATS is disabled by default, so you can develop using localhost. You should re-enable security or modify `Info.plist` in Xcode or in your favorite editor. Open the `Info.plist` file and remove the localhost under `NSExceptionDomains` under the following dictionary:

```
<key>NSAppTransportSecurity</key>
        <dict>
               <key>NSExceptionDomains</key>
               <dict>
                      <key>localhost</key>
                      <dict>
                          <key>NSExceptionAllowsInsecureHTTPLoads</key>
                          <true/>
                      </dict>
               </dict>
        </dict>
```

In Xcode, it will look like this:

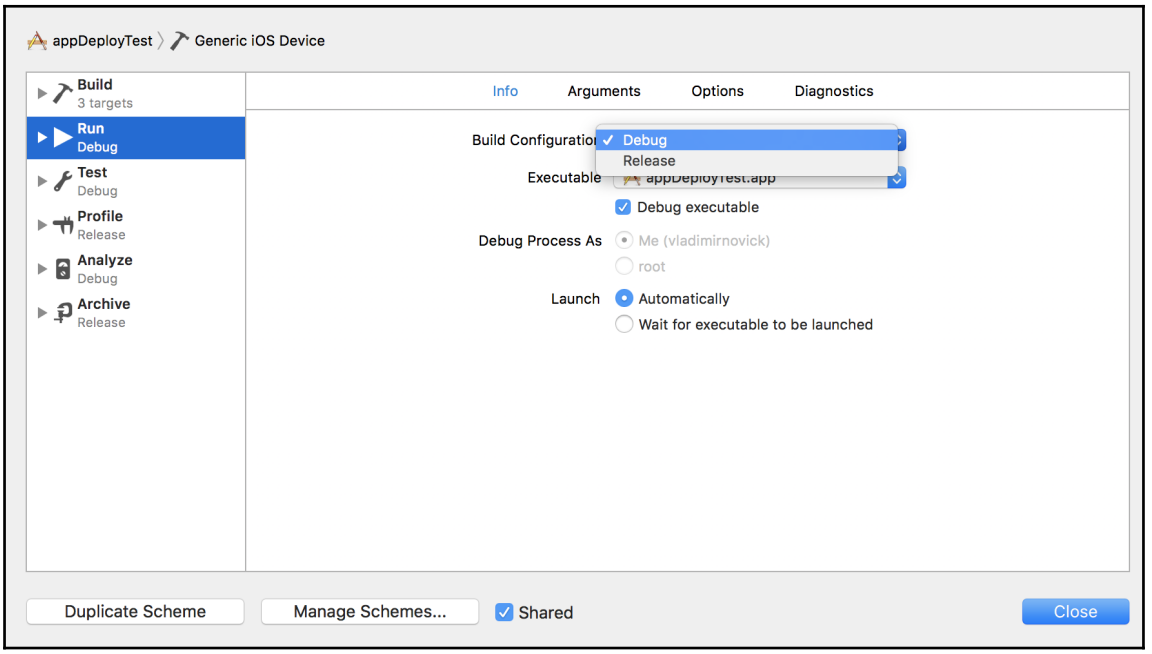


Configure Release scheme

Together with our app, we get the developer menu, which is good for development, but is completely unnecessary in production. In the Release scheme, we also want to bundle JavaScript locally and not serve it over the network.

To bundle our app in release mode, we can change our scheme to release by navigating to **Product** | **Scheme** | **Edit Scheme**.

It will open up the following window. In Build Configuration, select **Release** from the dropdown:



Finally, we can run *Cmd+B* or **Product** | **Build** to build our app:

Alternatively, instead of changing our scheme, we can just run:

```
react-native run-ios --configuration Release
```

Okay, so we've built our app. Now what?

It's time to test a bundled version on our test devices. In our developer account dashboard, where we've managed certificates, we can also manage our test devices by choosing the device type from the menu on the left:

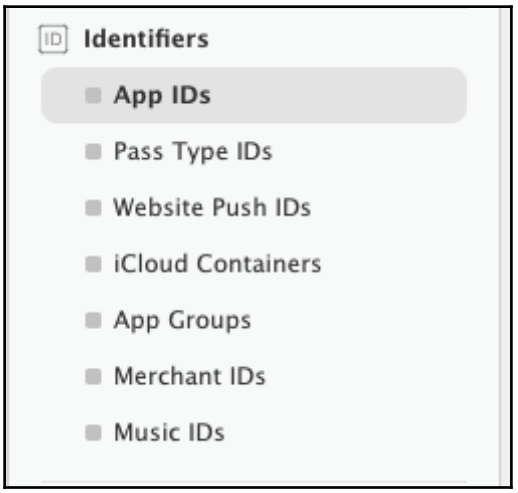
You will probably have your device there automatically if you've tried to run an app on your device. To register additional devices, you can read Apple documentation here:

`https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingProfiles/MaintainingProfiles.html#//apple_ref/doc/uid/TP40012582-CH30-SW10`

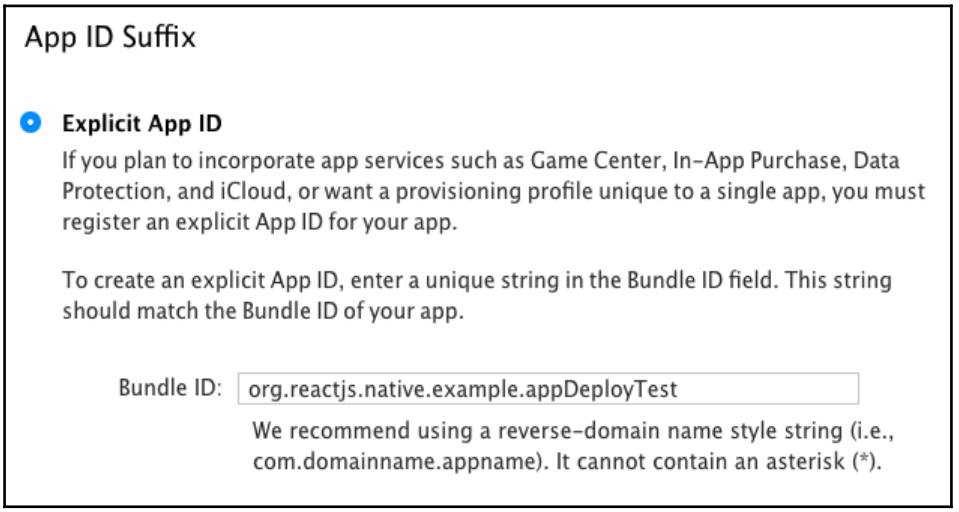
Registering our App ID

Apple enforces you to create an App ID for both deployment and release configurations. This can be done in the Apple Developer account dashboard. Let's overview it.

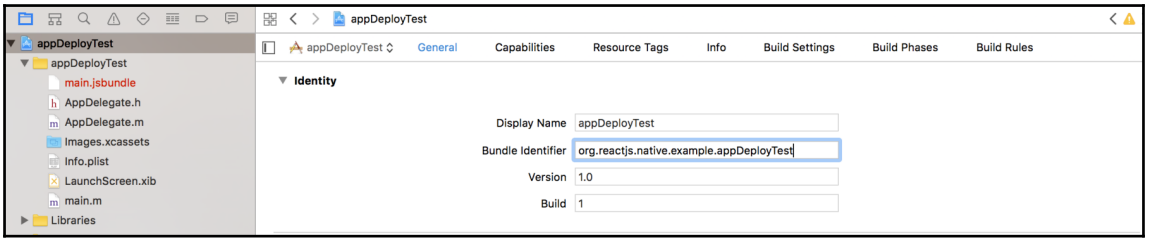
On our Apple Developer account dashboard, let's choose the **App IDs** section:



Here, we should add our App ID with the same name as our bundle identifier in Xcode:



Bundle identifiers are supposed to match.



Your Bundle Identifier is written inside the `Info.plist` file in the following format:

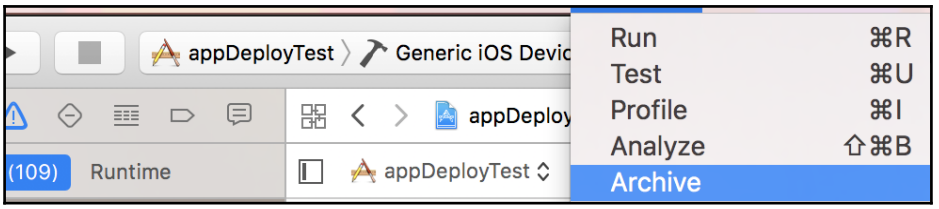
```
<key>CFBundleIdentifier</key>
<string>org.reactjs.native.example.$(PRODUCT_NAME:rfc1034identifier)</strin
g>
```

When you create your React Native app, by default it's prefixed with `org.reactjs.native.example`, however, you should change it to the proper name of your bundle register in your Apple Developer account.

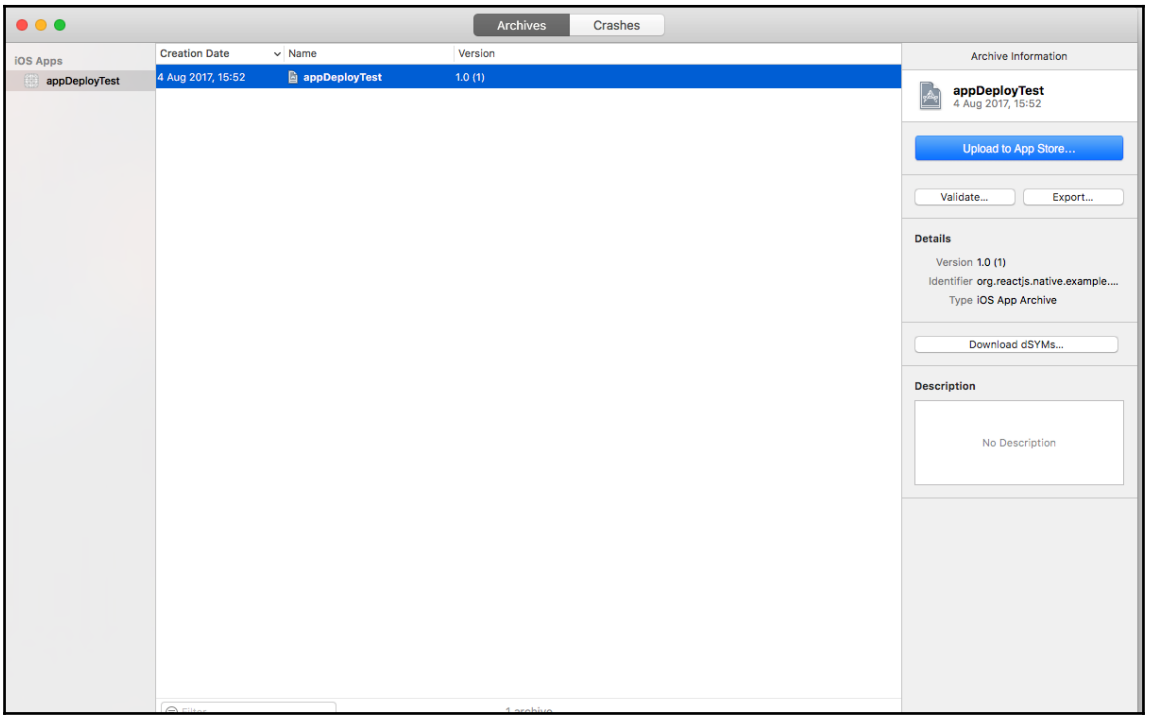
Archiving your app

Archiving your app is an essential process. Only after archiving your app do you get an `.ipa` file which you can distribute to a test device or eventually to the App Store.

In Xcode, navigate to to **Product** | **Archive** while selecting Generic iOS device in active scheme:



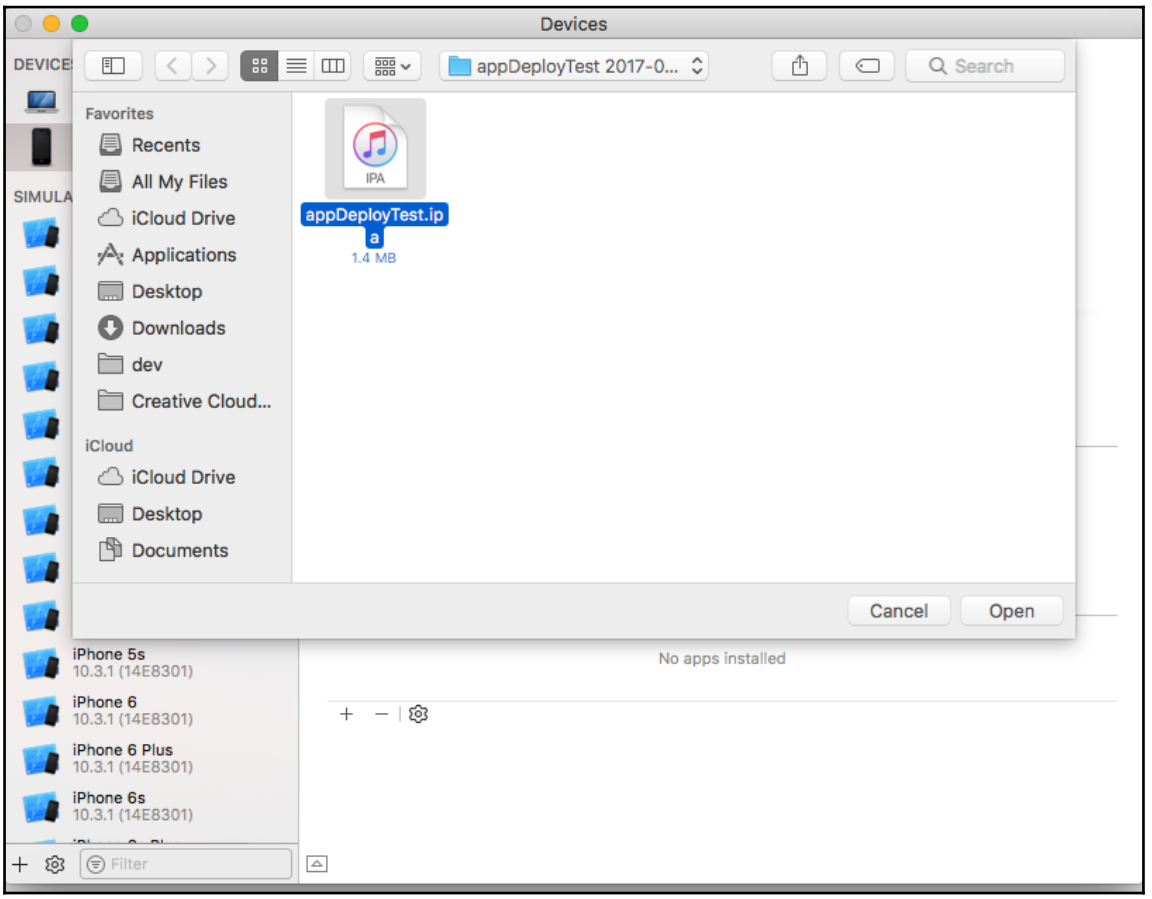
After the archiving process has finished, navigate to **Window** | **Organizer** to view your archives.



Here, you can either export your archive to test it on your phone or can **Upload to the App Store** by clicking on the blue button on the right.

Click on **Export** and select **Ad Hoc Deployment**. Then, export it for all compatible devices, or just choose a specific device. You will be asked to choose the folder where your `ipa` file can be saved. Now, you can upload it to your device by connecting your iOS device to Mac, and in XCode, navigate to **Window** | **Devices**.

When choosing your device, you can click on **+** at the bottom of the screen and add your `.ipa` file:

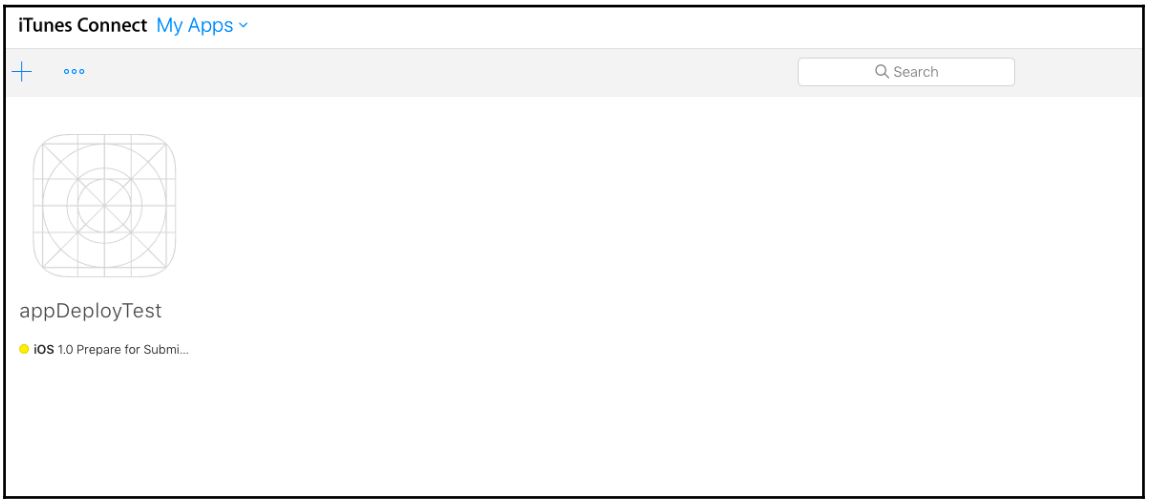


Now, disconnect your device and see how your application performs. Notice that it will be much faster because JavaScript is bundled, local on the device, and it is not fetched over the network. Shake your device. You won't see a developer menu.

Now, it's time to upload an app to the AppStore or TestFlight; this is basically a tool to invite beta testers to your app. But, before doing so, you need to create an app in iTunes Connect:

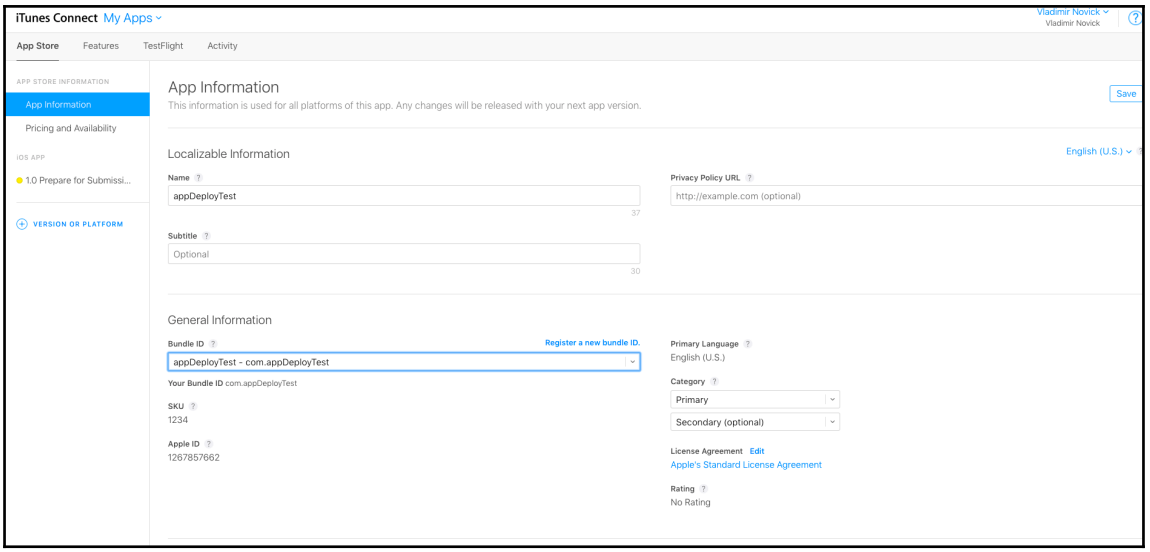
```
https://itunesconnect.apple.com/
```

You need to go under **My Apps** and click on the **+** button:



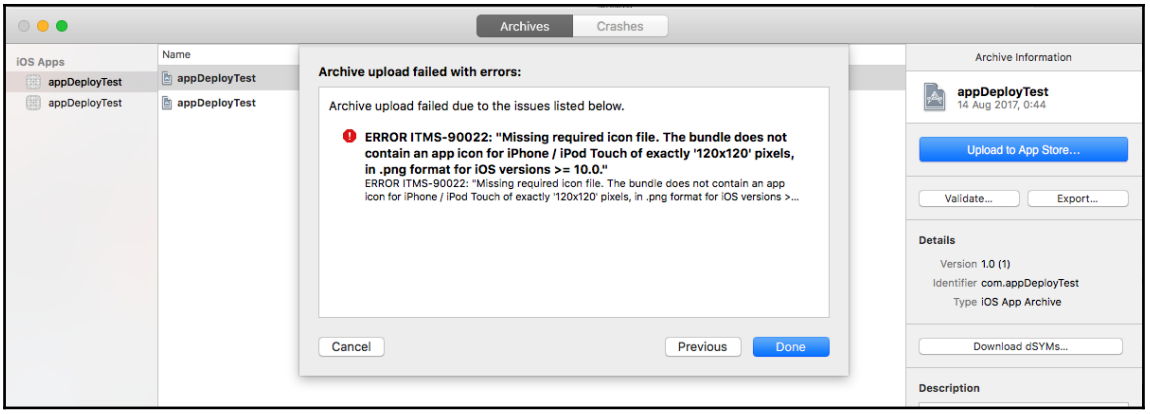
When you create your app, it's super important to choose the correct bundle Identifier for your app. If your bundle identifier is wrong, your app will fail to upload. The following is the main screen of your app in iTunes Connect, where you add all App information. Note that you have to add a Privacy Policy URL to your website too. This is what Apple says:

> *A URL that links to your organization's privacy policy. Privacy policies are required for apps that are Made for Kids or offer auto-renewable or free subscriptions. They are also required for apps with account registration, apps that access a user's existing account, or as otherwise required by law. Privacy policies are recommended for apps that collect user- or device-related data.*



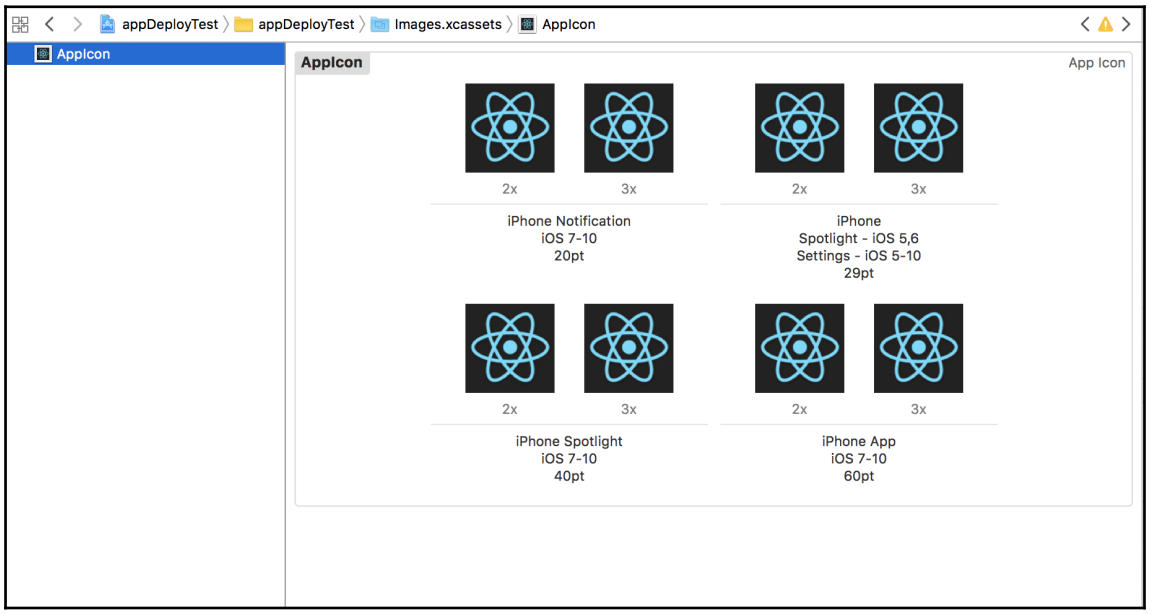
Under your IOS App, you need to fill in screenshots, information about your app, icons, and more. All this information has to be there so that you can submit your app for review. This process can be tedious if done manually, that's why services such as fastlane emerged. We will talk about fastlane in a bit.

After you have configured everything in your iTunes Connect and tried to **Upload to App Store...** you will probably encounter the following error:



That's because Apple restricts the uploading of anything to App Store without App Icons. To add them, go under the **General** tab and find the following section:

Here, by clicking the small arrow on the right near the **App Icons Source** dropdown, you will get to a screen where you can upload your icons by dragging and dropping them in icon slots.



Now, after adding App Icons, we can Build , Archive, and Upload our app once again.

You can use the `http://makeappicon.com` tool to create your icons. After your app has been accepted, it will end up in the App Store.

The entire building and submission process is long and tedious, and is well-known and documented for iOS developers in Apple Docs, however, it's crucial that you know all the major steps of the process.

After describing how deploy works for Android, we will see how this tedious process can be shortened with continuous integration tools.

Deploying Android apps and how it's done in React Native

In Android, the process of releasing an application to Google Play is much easier than in iOS. But, as in iOS, first we need to debug our application on the device so that we can be sure that everything is running properly.

At first, we need to enable USB debugging. This is done by navigating to **Settings** | **About the Device** and tapping several times on the Build Number field. Then, if we go back to the Settings menu, we will see Developer options appear. We need to select them and enable USB debugging there. Now it's time to connect a device via USB.

Let's check that our device is connected. In the console, type `adb devices`; you will get a list of devices attached. Only one device should be attached. It's possible that you will see both your device and an emulator. In that case, stop the emulator.

Now, in order to generate a release bundle, we need to generate a signed APK.

Note that if your application fails to deploy, make sure you check that you followed all the steps for signing your application for release. Check updated docs here: `https://facebook.github.io/react-native/docs/signed-apk-android.html`

Also, for information on how to update your apps, make sure you check out the official Google Play documentation here: `https://support.google.com/googleplay/android-developer/answer/113476?hl=en`

Generating a Signed APK file

This is done using the keytool utility from the terminal. Simply run the following command:

```
keytool -genkey -v -keystore appDeployTest.keystore -alias appDeployTest -
keyalg RSA -keysize 2048 -validity 10000
```

What is happening here is that you generate a keystore file with the `appDeployTest.keystore` name. The file will contain a single key valid for 10,000 days. It will be named `appDeployTest`.

When running this command, you will be asked twice to provide a password. Remember the password.

Next, move your file to the `android/app` folder and edit `.graddle.properties`,adding the following lines:

```
MYAPP_RELEASE_STORE_FILE=appDeployTest.keystore
 MYAPP_RELEASE_KEY_ALIAS=appDeployTest
 MYAPP_RELEASE_STORE_PASSWORD=*******
 MYAPP_RELEASE_KEY_PASSWORD=*******
```

Make sure you store your keystore because you won't be able to republish your app if you lose it without losing all ratings, comments, and so on.

The next step is updating the `build.gradle` file by adding the following section between `defaultConfig` and `buildTypes`:

```
signingConfigs {
 release {
  if (project.hasProperty('APPDEPLOYTEST_RELEASE_STORE_FILE')) {
   storeFile file(APPDEPLOYTEST_RELEASE_STORE_FILE)
   storePassword APPDEPLOYTEST_RELEASE_STORE_PASSWORD
   keyAlias APPDEPLOYTEST_RELEASE_KEY_ALIAS
   keyPassword APPDEPLOYTEST_RELEASE_KEY_PASSWORD
        }
    }
}
Under buildTypes -> release add:
signingConfig signingConfigs.release
```

Now, it's time to test our app in the project folder `run`:

```
react-native run-android --variant=release
```

This will build your React Native bundle in the release mode and deploy your APK to the device.

Distributing your APK

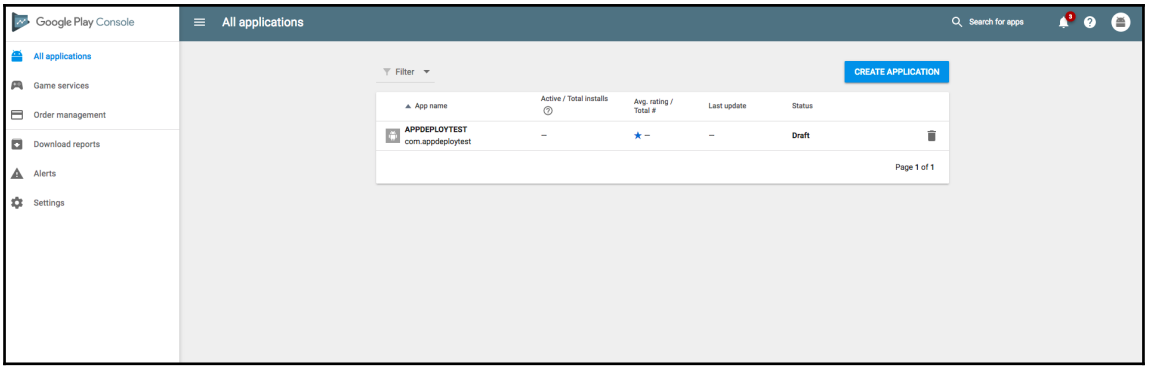
Your APK file can be generated from your Android folder by running:

```
./gradlew assembleRelease
```

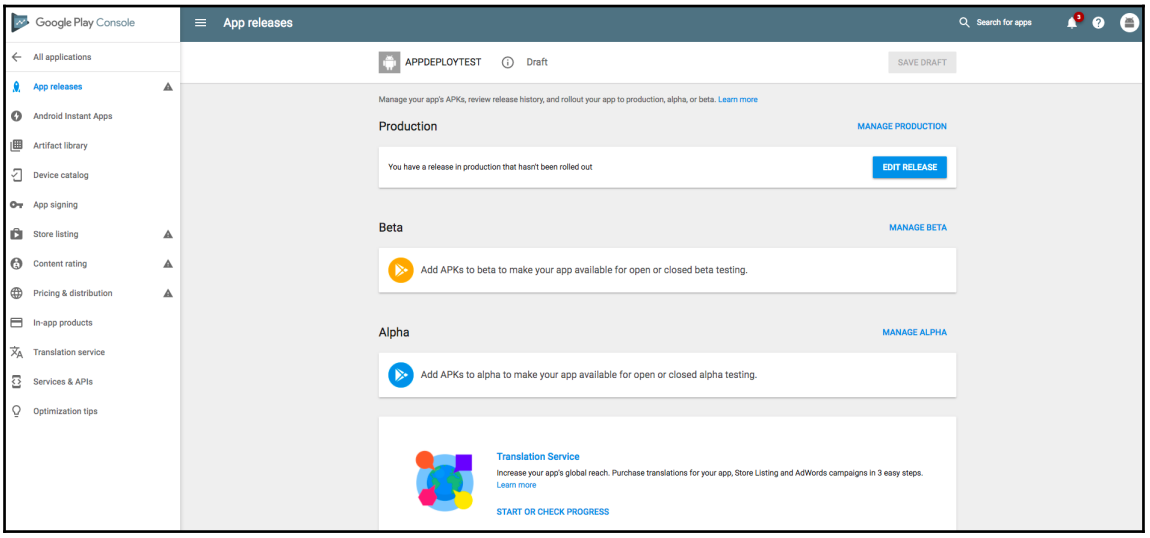
This will create the `app-release.apk` file under `android/app/build/outputs/apk`.

Now, sign into the Google Play console `https://play.google.com/apps/publish`.

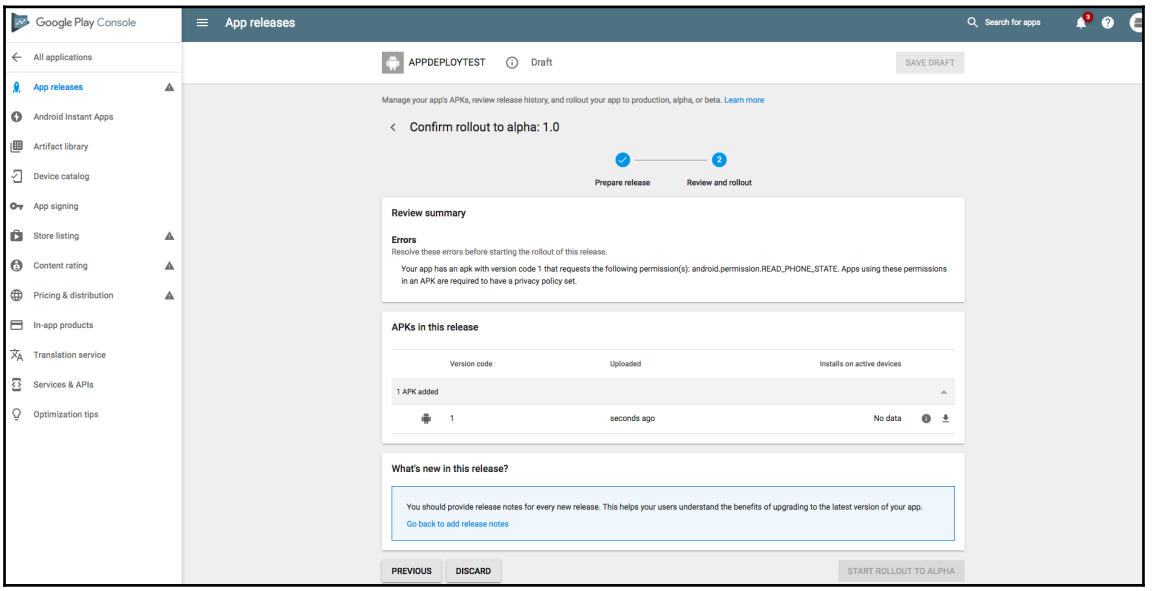
There, you need to create an application:



Then, you can choose whether you want to upload your application to **Production** or to **Alpha**/**Beta** testing:



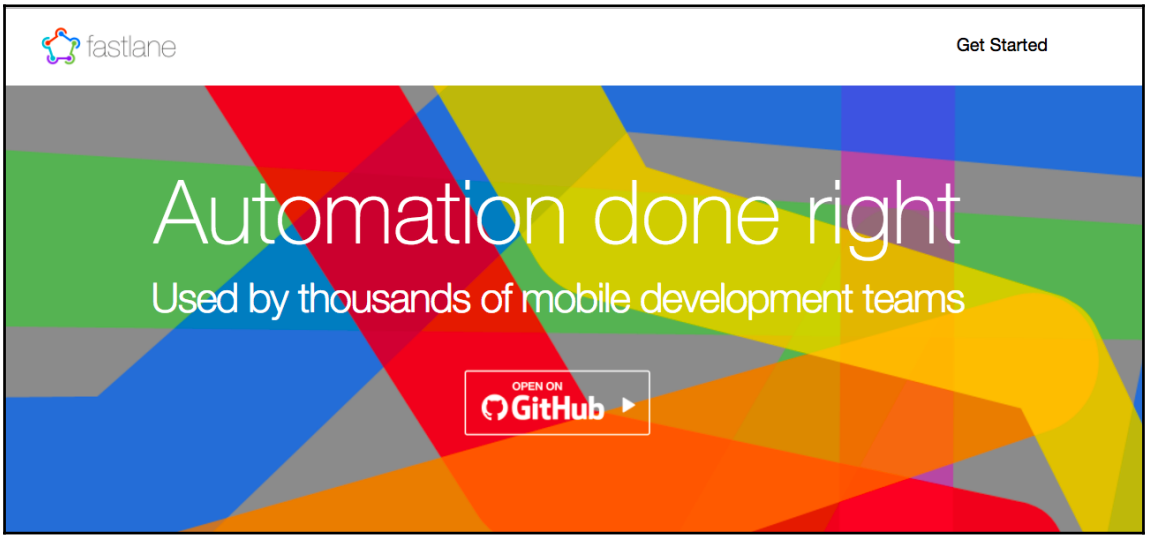
In the Store listing menu, you have to provide all the information about the app, screenshots, description, and so on. After you choose your **Beta**/**Alpha** or **Production** path, it's time to approve your app and it will get into the PlayStore:



As you can see, the process of deploying Android apps is much shorter than the process for iOS apps; however, both of them share the same things. You should bundle your React Native app with the release option and produce quality assets for your application listing in the App Store. Both of them need you to generate some sort of certificate. In Android, it's much, much simpler than in iOS, however, both have these security measures.

Introducing fastlane - automate your deployment workflow

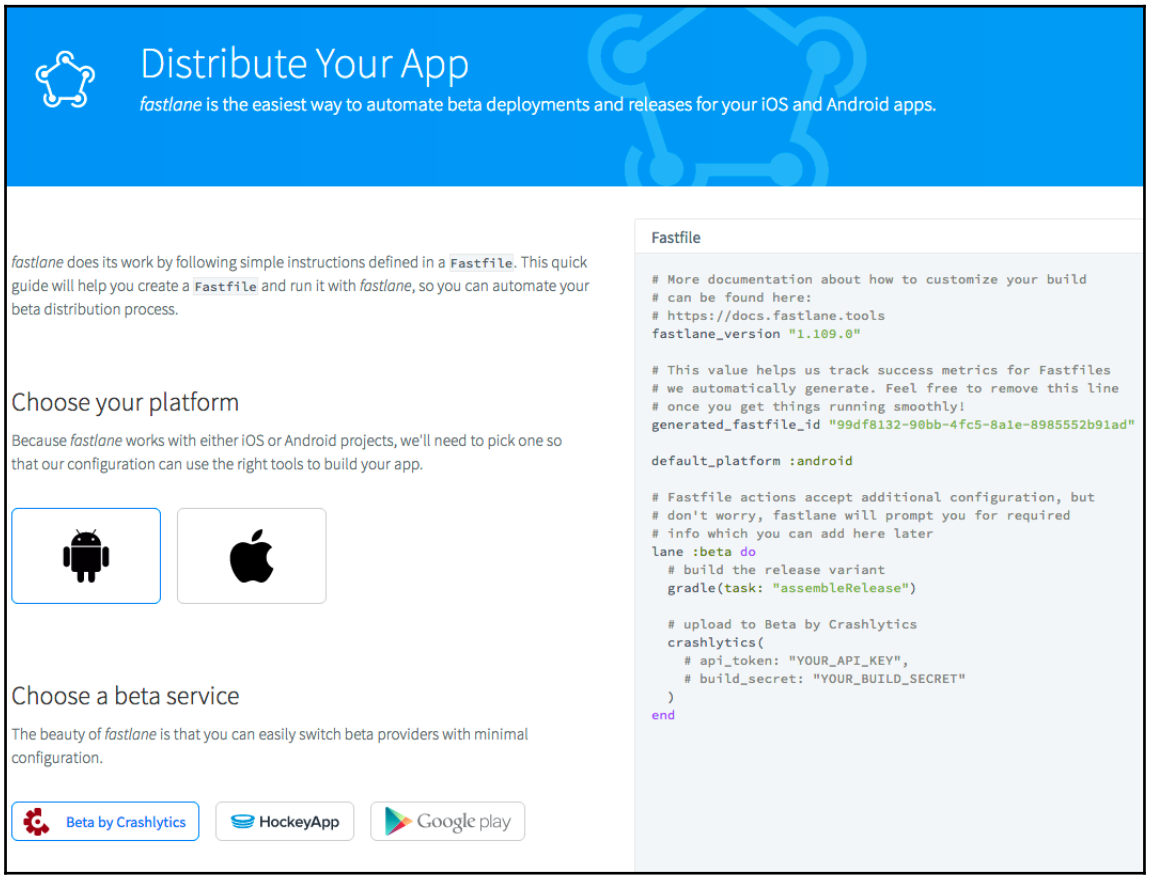
In addition to deployment of your iOS and Android apps, there is a testing phase where you have to manage your certificates and keystores properly. You have to manage screenshots for your apps, revisions, and much more. In order to do all these things with ease, there is a service called **fastlane**--a continuous delivery tool for Android and iOS development:



Fastlane brings with it several things: version control, automatic deployment, bringing together iOS and Android apps deployment into one automated process.

This is not something specific to React Native. Fastlane was born out of the need for companies that develop both iOS and Android for ease of common repetitive tasks such as generating screenshots, signing apps, deploying, testing, and so on.

If you go to `https://fastlane.tools/` and click on **Get Started,** you will find a very descriptive explanation of how to set this up exactly for both iOS and Android:



The main steps will be to get fastlane for iOS and Android, set them separately as one in an iOS folder and the second in an Android folder, and then use the npm scripts to automate the process. So, deploying is as simple as `npm run deploy-ios` or `npm run deploy-android`.

Get to know Microsoft CodePush and integrate it with your application

So, after looking briefly into the continuous integration tool fastlane, which is not unique to the React Native ecosystem, let's take a look at an absolute must have tool for your React Native app: **Microsoft CodePush**.

CodePush is a service that allows us to update our JavaScript bundle and images in our **already deployed** React Native app. This means that you can change your UI/UX, fix potential bugs, and introduce new features without going through deployment and review processes in both stores. The fact CodePush gives you this ability is due to the fact that the React Native app consists of bundled JavaScript served over the bridge. This means that, potentially, we can update this bundle over the network.

The limitations of CodePush are that we can do instant updates **only** with JavaScript code and assets. We cannot carry out such updates for any packages that contain native code; however, since most of our code is in JavaScript, this is not really a problem. In addition to the ability to push your bundle over the network, you can also rollback to a prior version, since CodePush stores it in case you change your mind and want to revert.

So, how do we get started

Let's install CodePush CLI globally:

```
npm i -g code-push-cli
```

Then, register `code-push register`. This will open Mobile Center in a browser where you have to sign in with your GitHub or Microsoft account and you will get a token.

In the terminal, you will be asked the following:

Please login to Mobile Center in the browser window we've just opened.
Enter your token from the browser:

So, paste the token you will get from the browser there. Let's now register our app with CodePush:

```
code-push app add appDeployTest ios react-native
```

Setting up your mobile client

Install CodePush client for react native:

```
npm i -S react-native-code-push
```

Run the following setup instructions in Readme:

`https://github.com/Microsoft/react-native-code-push`

In order to use CodePush in your application root file, you wrap it into the `CodePush` function like this:

```
import codePush from 'react-native-code-push'
// ...
AppRegistry.registerComponent('appDeployTest',
   () =>
     codePush(
       codePushOptions
     )(appDeployTest)
 );
```

In `codePushOptions`, you can specify various options of how exactly to check for bundle updates.

We won't dive deep into all these options here, but it's important that you know that there is such a service and understand it's basic usage.

Everything else can be easily found in it's documentation at `http://microsoft.github.io/code-push/docs/getting-started.html`.

Summary

Congratulations! You have successfully completed all the chapters of this book. You've learned all aspects of React Native developer, as well as useful tricks and techniques along the way. In the end, we've covered techniques that iOS and Android developers use for deploying their apps, and how we can make it work even better with Microsoft CodePush and fastlane.

So, you may ask, now what? What are my next steps? Make sure to check the following site for tons of resources on React Native, talks, tutorials, news and much more:

`http://www.awesome-react-native.com`

Also make sure to check my personal site in case you want to connect with me, book a consultation or workshop or just get to know me better:

`http://vnovick.com`

In addition to all of the above I strongly advise you to go over the book again and enhance all the apps we've created throughout the book. Add more functionality to them, change their design, and add native APIs. After all, the best way of learning is to develop more and more apps. I hope that the book has given you the confidence to start building your own apps for fun and profit.

Index

A

additional APIs
  camera package 312
  ExpoKit 308
  Image Picker 310
  maps 308, 309
  react-native-video 311
  toast package 311
Android apps, deploying
  about 392
  apk, distributing 394, 395
  signed apk file, generating 392
Android dependencies
  Android studio, installing 30
  Android Virtual device, creating 33, 35, 36
  ANDROID_HOME environment variable, setting up 33
  built tools, installing 31
  installing 28
  JDK, installing 29
  SDK, installing 31
Android studio
  URL 30
Animated API
  Animated values, using 170
  used, for creating complex animations 170
  values, calculating 174
Animated functions
  extrapolation 178
  interpolation 177
  using 175
Animated.Value
  interpolate method 172
  reference 172
  removeAllListeners 172
  removeListener method 172
  setValue method 172
  stopAnimation method 172
animations
  combining, for complex sequences 181
  conventions 155
  creating, with LayoutAnimation API 157
  overview 155
  panning 185
  scrolling 185
  working with 156
App Icons
  URL 391
app related APIs
  about 266
  InteractionManager 267
  Linking API 267
App Store
  URL 373
App Transport Security (ATS) 380
application
  base elements 58
  breaking down 57
  containers, using 57
  creating 37, 38
  creating, with create-react-native-app 39
  creating, with Expo 39
  resemblance to HTML 57
  reusable components, using 57
  structuring 57
AppRegistry 267
architecture, Flux
  about 222
  Model View Controller (MVC) 223
  reference 225
  unidirectional data flow 224
architecture, React Native
  about 17

Bridge layer 17
JavaScript layer 17
Native layer 17
async actions
data flow 238
redux-thunk, using 237
AsyncStorage API
persistence, observing 280
Atom
URL 36
authentication
logic 214
setting up, at Firebase 209
via social providers 220
AVD creation
reference 36

B

base elements 58
Base64 269
boilerplates 321
Bridge layer 17
Browser DOM 16

C

camera package
about 312
URL 312
CameraRoll API
photos, retrieving 272
photos, saving 272
Chrome Developer tools
debugging 19, 20
complex animations
creating, with Animated API 170
component types
stateful 50
stateless 50
components, React Native apps
accessibility 66
images 70
layouting 65
media 70
StatusBar 68
text 67
touch events 65
view 65
controls
about 98
SegmentedControlIOS 98
TouchableNativeFeedback 99
conventions, animations
animation states 155
descriptions 156
timing graph 155
create-react-native-app
used, for creating application 39
Cross Axis 139
CSS styles
versus inline styles and React Native styles 143
custom native module
files, creating 328
folder structure, creating 328
native module Java class, creating 329
native module package, creating 330
package, registering 331
writing 324
writing, in Android 328
writing, in iOS 324
writing, in Objective-C 324

D

data-related packages
about 314
react-native-fetch-blob 314
react-native-firebase 317
react-native-i18n 320
react-native-push-notifications 317
DatePickerIOS
about 97
URL 97
deployment workflow
automating, with fastlane 396, 397
devdocs
URL 46
development environment
Android dependencies, installing 28
iOS dependencies, installing 28
projects, installing with Native code 26
setting up, for Android apps 26

setting up, for iOS apps 26, 36
Document Object Model (DOM) 10
Don't Repeat Yourself (DRY) 13
DrawerLayoutAndroid
URL 100

E

easing functions
about 156
URL 156, 176
Expo XDE
URL 45
Expo
URL 254
used, for creating application 40
ExpoKit
about 308
URL 308
extrapolation
using 178

F

Facebook APIs
URL 220
Facebook
about 307
reference 307
fastlane
about 390, 396
deployment workflow, automating 396, 397
URL 397
feedback, obtaining from application
ActivityIndicator 75
modal 79
Firebase
about 188, 189
app, initializing 197
authentication 189
authentication, setting up 209
Cloud Storage 189
configuring, to React Native 196
data, fetching 196
data, writing 200
real-time database 189
real-time database, creating 192
reference 195, 207
setting up 190
URL 190
FlatList
about 84, 88
URL 88
flexbox
about 14
elements, aligning 140
item dimensions 142
layout techniques 139
reference 139
styling concepts 138
Flux
architecture 222
folder structure
about 59
src/components directory 60
src/config directory 60
src/images directory 60
src/lib directory 61
src/screens directory 60
src/services directory 61
functional component 51
functions
working with 133

G

Genymotion
URL 36
GeoLocation API
location, obtaining 276
URL 278
usage 278
using, with Android 277
using, with iOS 277
Gesture responder system
about 286
URL 287
Google Play
URL 394

H

Hackathon
history 11

higher order component (HOC) 215
Homebrew
  URL 26
Hot Module Reloading (HMR) 13
hybrid applications
  about 14
  limitations 15
  React Native, differentiating from 14, 15
hybrid apps 10

I

Image 70
image components
  reference 58
Image Picker
  about 310
  URL 310
image related APIs
  about 268
  ImageEditor 268
  ImagePickerIOS 269
  ImageStore 268
ImageBackground 144
information APIs
  about 260
  AccessibilityInfo 260
  AppState 260
  dimensions 261
  NetInfo 261
  PixelRatio 261
  platform 262
  settings 262
inline styles
  versus CSS styles and React Native styles 143
input related APIs
  about 264
  Clipboard 264
  DatePickerAndroid 265
  Keyboard 264
  TimePickerAndroid 266
InteractionManager 267
interpolation
  reference 180
  using 177
iOS apps
  App ID, registering 383, 385
  App Transport Security (ATS), enabling 380
  app, archiving 385, 387, 389, 390
  Apple Developer program, joining 374
  certificate, creating 374, 376
  deploying 373
  release scheme, configuring 383
  signing up 377, 378, 380
iOS dependencies
  installing 28
iTunes Connect
  URL 388

J

JavaScript layer 17
JDK (Java SE Development Kit)
  installing 29
  URL 29
Jest React Native
  reference 137
Jest testing framework
  about 128
  preset system 128
  reference 129
  setting up 128
  syntax 129
JSX-XML-like syntax 10
JSX
  about 45, 47
  children component 48
  props 49, 50
  reference 50
  using, in React Native 45

L

LayoutAnimation API
  advantages 170
  preset system, using 163
  syntax 160
  used, for creating animations 157
libraries
  and APIs, linking with native code 249
  auto linking 250
  build phases, configuring 251
  linking 250

manual linking 250
lifecycle methods
about 54
componentDidMount 55
componentWillMount 54
componentWillReceiveProps 55
componentWillUnmount 55
for mounting 54
for updating 55
reference 55
shouldComponentUpdate 55
Linking API 267
lists of data
dealing 80
FlatList 84, 88
ListView 80, 83
RefreshControl 84
ScrollView 83
SectionList 89, 91
VirtualizedList 91
ListView
about 80
references 83
Live Reload 20
logging
about 117
errors 118
erros 120
in-app errors 118
warnings 118, 121
Login screen
creating 210, 211
Lottie 304
lottie-react-native
about 304
URL 304

M

Main Axis 139
Main Flow 298
maps
URL 308
Microsoft Codepush
about 398
installing 398
integrating, with application 398
mobile client, setting up 399
URL 399
mobile development
bottlenecks 12
developer experience state 13
specific platform 12
with React approach 12
MobX
about 241, 242, 292
actions 243
concepts 243
connecting, to WhatsApp app clone app 244
derivations 243
implementing 357, 364, 365, 366, 368
react-navigation package, integrating 297
state 243
URL 242, 243, 247
Modal
about 79
animationType 79
onShow 80
transparent 79
visible 79
Model View Controller (MVC) 221
modules
mocking 136
Mozilla Developer Network (MDN)
URL 46

N

native APIs
AdSupportIOS 270
BackHandler 270
information APIs 260
notification APIs 254
overview 253
PermissionsAndroid 270
share 271
native driver
reference link 176
Native layer 17
NativeBase
about 304
URL 305

navigation
  performing 295
  reference 235, 295
navigators
  creating 293
  DrawerNavigator 294
  StackNavigator 293
  TabNavigator 294
Node Package Manager (npm) 27
notification APIs
  about 254
  ActionSheetIOS 257
  Alert 254
  AlertIOS 256
  PushNotificationIOS 259
  ToastAndroid 256
  vibration 259
Nuclide
  URL 36

O

OAuth based packages
  about 308
  URL 308
open source packages
  additional APIs 308
  based on data 314
  based on OAuth 308
  boilerplates 321
  for animations 302
  for visuals 302
  related to social providers 307
  using 301

P

PanResponder
  about 287
  combining, with Animated pan 289
  Gesture responder system 286
  URL 291
  user gestures, responding to 281
permissions
  managing 195
platform-dependent components
  about 96
  extensions 97
  specific platform, detecting 96
platform-independent components 63
platform-specific navigation
  about 99
  DrawerLayoutAndroid 100
  TabBarIOS 99
  ToolbarAndroid 100
  ViewPagerAndroid 100
platform-specific props
  reference 145
presentational components
  about 51
  versus stateful components 50
Progress Bars
  about 98
  references 98
props
  reference 150

R

React Native apps
  app, reloading 109
  components 63
  debugging 107
  hot reloading 112
  in-app developer menu 107
  live reload 110
  reference 63
  reloading 110
  URL, for accessibility 66
  URL, for StatusBar 70
React Native components
  inspecting 123
  performance monitoring 126
  snapshot testing 132
React Native styles
  versus CSS styles and inline styles 143
React Native, benefits
  about 18
  code reusability, across applications 23
  developer experience 18
  web-inspired layout techniques 23
React Native, features
  Chrome Developer tools, debugging 19

component hierarchy, inspecting 21, 22
hot reload 20
Live Reload 20
React Native
about 9
architecture 17
data, fetching 207
differentiating, from hybrid applications 14, 15
evolution 12
files, fetching 209
files, uploading 209
Firebase, configuring 196
history 10
information flow 16
integrating, with existing app 332
navigation 100
Navigator component 100, 104
need for 12
process 15
support for Websockets 208
threading model 18
react-native-animatable
about 303
URL 303
react-native-elements
about 306
URL 307
react-native-Firebase
URL 197
react-native-i18n
about 320
URL 320
react-native-vector-icons
about 303
URL 303
react-native-video
about 311
URL 311
React-navigation 100
react-navigation package
app navigation structure, setting 298
exploring 293
integrating, with MobX 297
integrating, with Redux 295
navigation, performing 295
ReactJS
history 10
Readme
URL 399
real-time database
creating 192
Reducer function 227
Reducers
reference 229
redux-saga
URL 241
redux-thunk
adding 238
side effects, centralizing 241
using, for async actions 237
Redux
about 226, 292
Actions 228
actions, refactoring 360, 362, 363
concepts 225
connect, using 235
connecting, to WhatsApp app clone app 232
containers, versus components 234
core concepts 227
data flow 231
data mocks, moving to centralized state 358, 359, 360
implementing 357
Middleware 230
principles 226
Provider component 234
pure functions, using 227
react-navigation package, integrating 295
read-only state 227
Reducers 228
relevant state keys, obtaining with selectors 237
state, defining 226
state, modifying 226
Store 230
URL 225, 231
usage 225
RefreshControl 84
remote debugging
about 114
device, debugging 116

reference 116
with Chrome DevTools 114
Reselect library
URL 237

S

ScrollView
about 83
URL 84
SectionList
about 89
URL 91
SegmentedControlIOS
about 98
reference 99
serialized batched response 17
Shoutem UI
about 304
URL 304
Show Inspector 21
Sign Up screen
creating 210, 214
single responsibility principle 57
social provider related packages
about 307
Facebook 307
Splash Screen 298
stateful components
about 52
versus presentational components 50
stateless components 50
style related APIs 269
styling, best practices
about 150
common components, extracting 153
dimensions, modifying 151
style sheet, using 151
styles, coding 152
styles, splitting 152

T

TabBarIOS
URL 99
testing 128
TextInput
about 93
references 93
Tinder application
URL 282
toast package
about 311
URL 311
ToolbarAndroid
URL 100
TouchableNativeFeedback 99
Twitter API
URL 371
Twitter client app
architecture, defining with desired design 336, 338, 339, 340, 343, 344
data, mocking 352, 354, 355, 357
functional navigation, setting up 345, 346
requirements, defining 335
screen, styling with animations 352, 354, 355, 357
Twitter API data, using 368, 369, 370
wireframe, setting up 345, 346

U

unidirectional data flow
about 12, 224
actions, dispatching 224
user gestures
responding to, with PanResponder 281
user input
handling 93
restricted choice inputs 94
TextInput 93
user interaction
about 72
button 72
TouchableHighlight 75
TouchableOpacity 74
TouchableWithoutFeedback 75

V

ViewPagerAndroid
URL 100
Virtual DOM 10, 16
VirtualizedList

about 91
URL 91
visuals related packages
lottie-react-native 304
NativeBase 304
react-native-animatable 303
react-native-elements 306
react-native-vector-icons 303
Shoutem UI 304
VSCode
URL 36

W

Watchman 27
web content
embedding 91
WebSockets
using, with React Native 208
WebView 10
WhatsApp app clone app
background images, customizing 146
laying out 145
MobX, connecting 244
Redux, connecting 232
styles, applying 148
wireframe, Twitter client app
Discover screen, creating 351
Home screen, creating 350
Login screen, creating 347
Main flow, defining 348, 349
Notifications screen, creating 351
Profile screen, creating 352
setting up 345
SplashScreen, creating 347
Tweet screen, creating 352

Y

yarn
URL 27