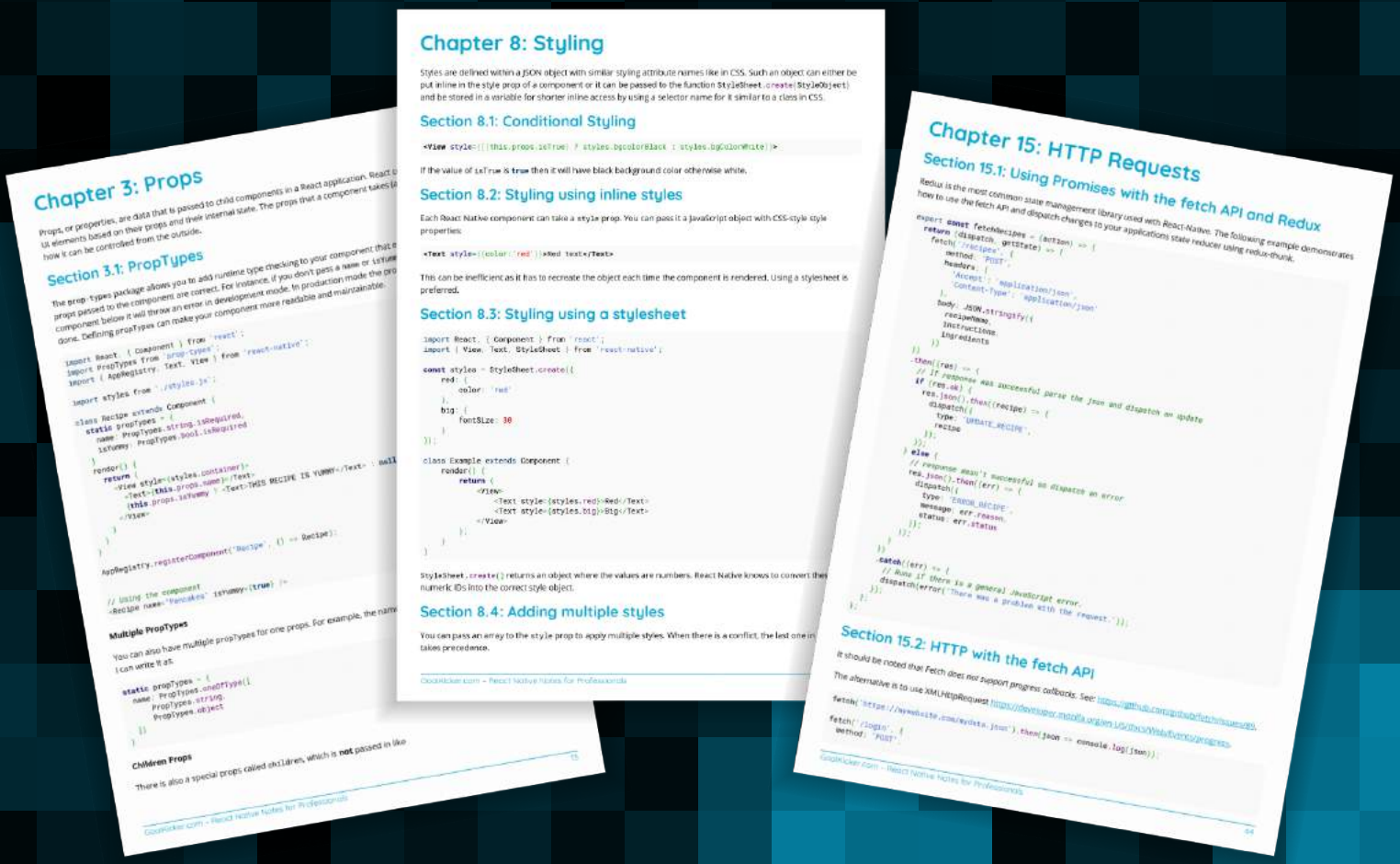
React Native

Notes for Professionals



80+ pages

of professional hints and tricks

GoalKicker.com

Free Programming Books

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
http://GoalKicker.com/ReactNativeBook

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with React Native

Section 1.1: Setup for Mac

Installing package manager Homebrew `brew`

Paste that at a Terminal prompt.

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Installing Xcode IDE

Download it using link below or find it on Mac App Store

https://developer.apple.com/download/

> **NOTE:** If you have **Xcode-beta.app** installed along with production version of **Xcode.app**, make sure you are using production version of `xcodebuild` tool. You can set it with:
>
> ```
> sudo xcode-select -switch /Applications/Xcode.app/Contents/Developer/
> ```

Installing Android environment

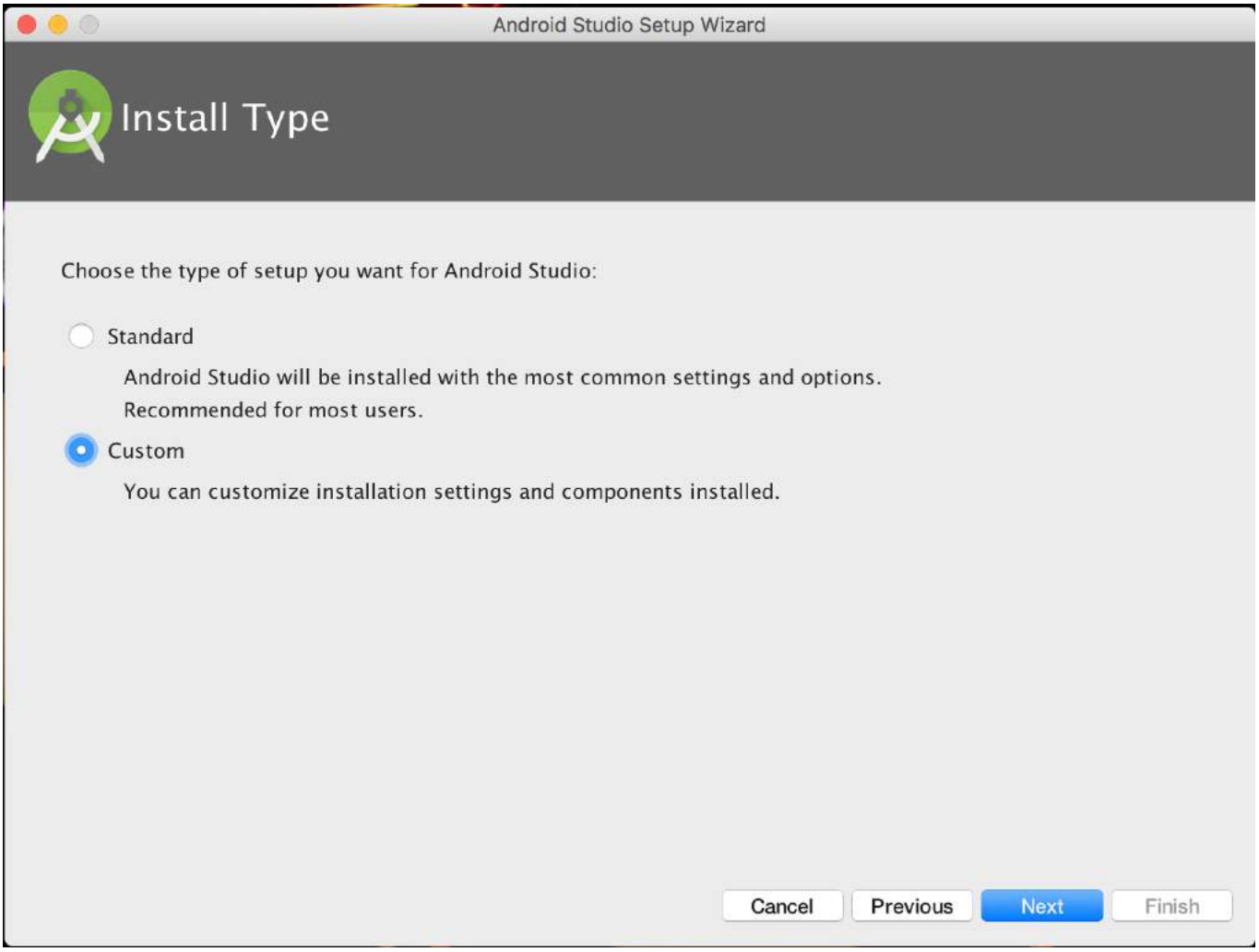
- Git **`git`**

  *If you have installed XCode, Git is already installed, otherwise run the following

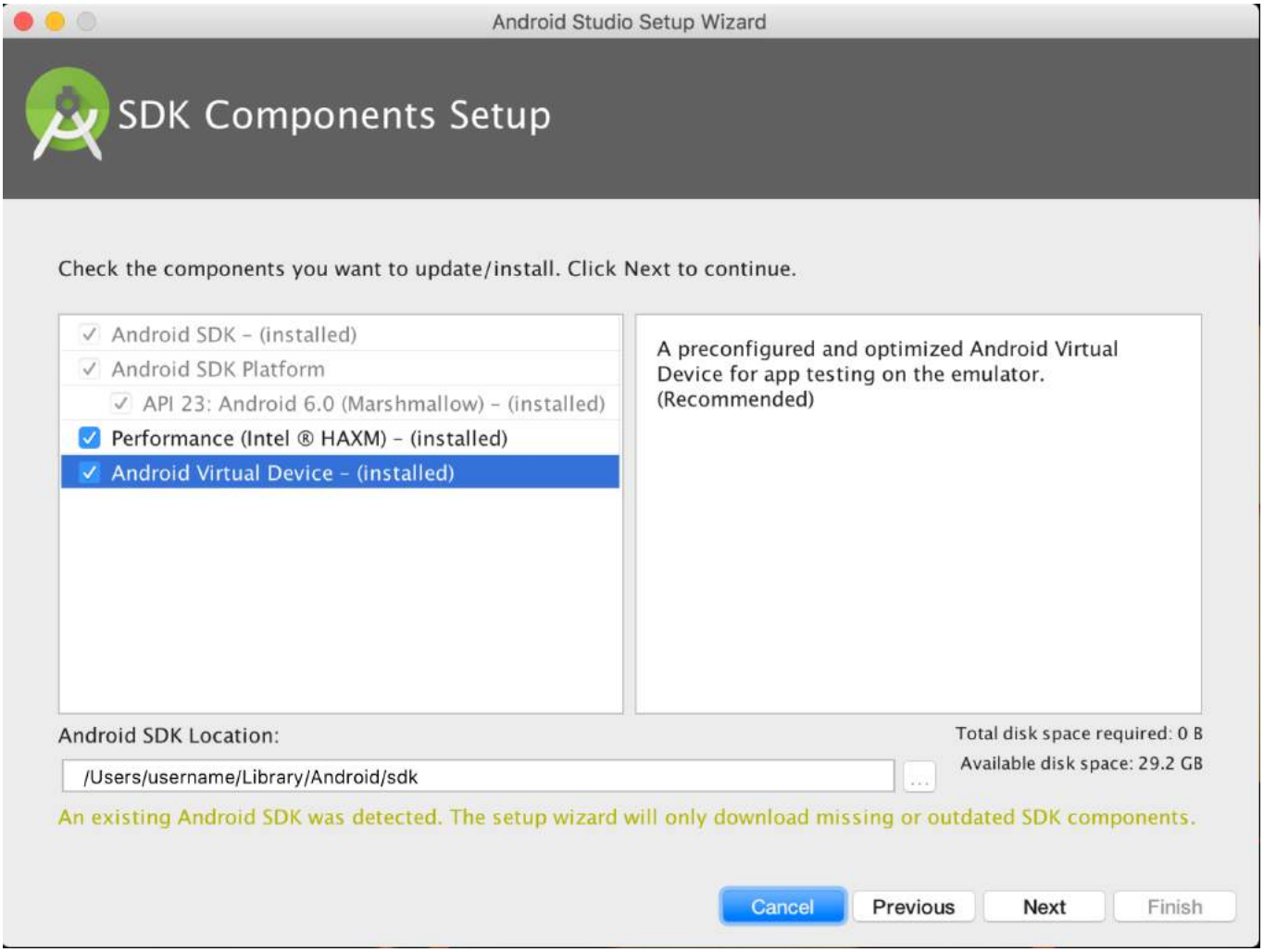
  ```
  brew install git
  ```

- Latest JDK
- Android Studio

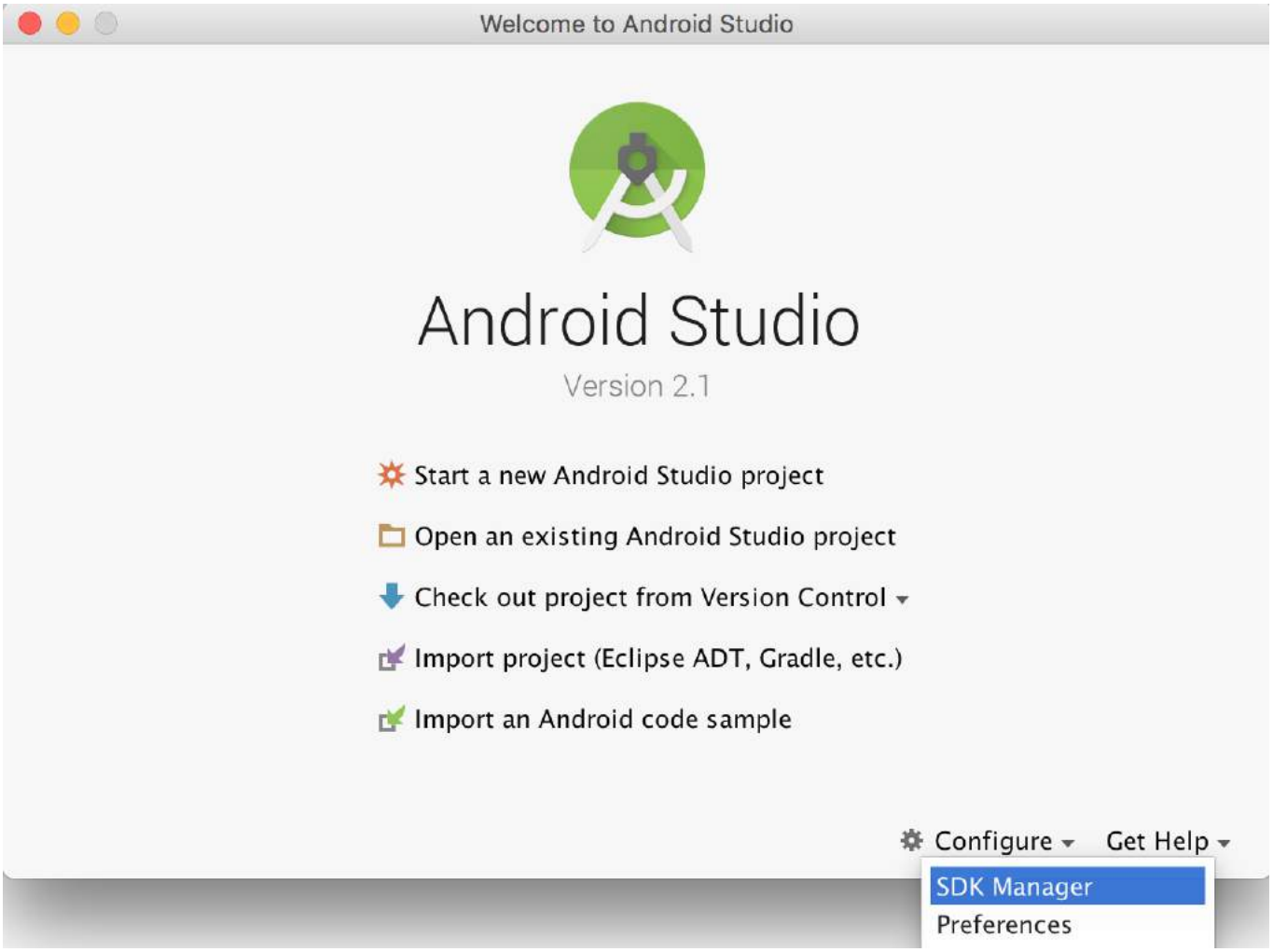
  Choose a Custom installation



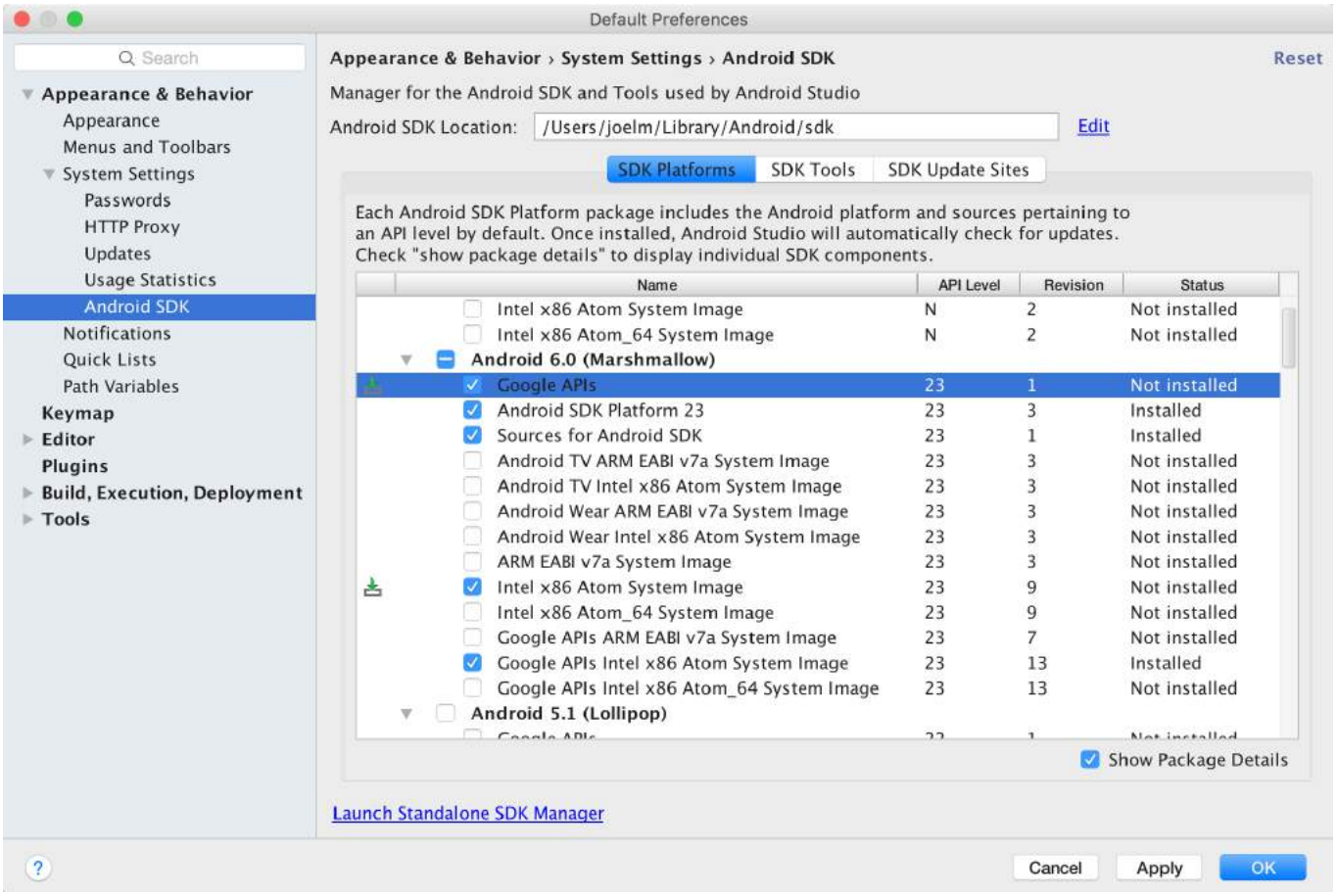
Choose both Performance and Android Virtual Device



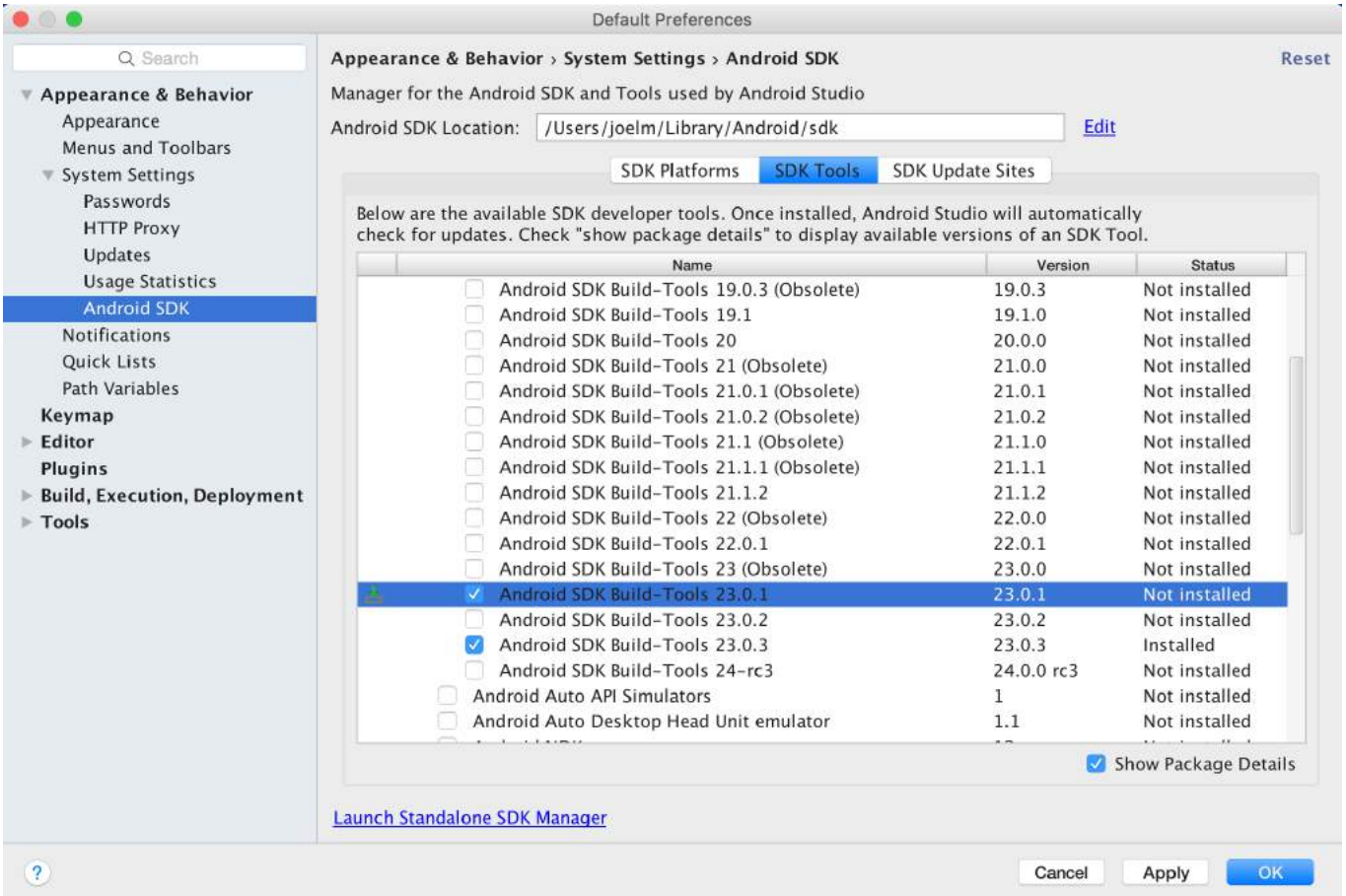
After installation, choose Configure -> SDK Manager from the Android Studio welcome window.



In the SDK Platforms window, choose Show Package Details and under Android 6.0 (Marshmallow), make sure that Google APIs, Intel x86 Atom System Image, Intel x86 Atom_64 System Image, and Google APIs Intel x86 Atom_64 System Image are checked.



In the SDK Tools window, choose Show Package Details and under Android SDK Build Tools, make sure that Android SDK Build-Tools 23.0.1 is selected.



- Environment Variable `ANDROID_HOME`

  Ensure the ANDROID_HOME environment variable points to your existing Android SDK. To do that, add this to your ~/.bashrc, ~/.bash_profile (or whatever your shell uses) and re-open your terminal:

  If you installed the SDK without Android Studio, then it may be something like: /usr/local/opt/android-sdk

  ```
  export ANDROID_HOME=~/Library/Android/sdk
  ```

Dependencies for Mac

You will need Xcode for iOS and Android Studio for android, node.js, the React Native command line tools, and Watchman.

We recommend installing node and watchman via Homebrew.

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

> Watchman is a tool by Facebook for watching changes in the filesystem. It is highly recommended you install it for better performance. It is optional.

Node comes with npm, which lets you install the React Native command line interface.

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

If you get a permission error, try with sudo:

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

For iOS the easiest way to install Xcode is via the Mac App Store. And for android download and install Android Studio.

If you plan to make changes in Java code, we recommend Gradle Daemon which speeds up the build.

Testing your React Native Installation

Use the React Native command line tools to generate a new React Native project called "AwesomeProject", then run react-native run-ios inside the newly created folder.

```
react-native init AwesomeProject
cd AwesomeProject
react-native run-ios
```

You should see your new app running in the iOS Simulator shortly. react-native run-ios is just one way to run your app - you can also run it directly from within Xcode or Nuclide.

Modifying your app

Now that you have successfully run the app, let's modify it.

- Open index.ios.js or index.android.js in your text editor of choice and edit some lines.

- Hit Command⌘ + R in your iOS Simulator to reload the app and see your change! That's it!

Congratulations! You've successfully run and modified your first React Native app.

source: Getting Started - React-Native

Section 1.2: Setup for Linux (Ubuntu)

1) Setup Node.JS

Start the terminal and run the following commands to install nodeJS:

```
curl -sL https://deb.nodesource.com/setup_5.x | sudo -E bash -

sudo apt-get install nodejs
```

If node command is unavailable

```
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

Alternatives NodeJS instalations:

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
sudo apt-get install -y nodejs
```

or

```
curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
sudo apt-get install -y nodejs
```

check if you have the current version

```
node -v
```

Run the npm to install the react-native

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

2) Setup Java

```
sudo apt-get install lib32stdc++6 lib32z1 openjdk-7-jdk
```

3) Setup Android Studio:

Android SDK or Android Studio

```
http://developer.android.com/sdk/index.html
```

Android SDK e ENV

```
export ANDROID_HOME=/YOUR/LOCAL/ANDROID/SDK
export PATH=$PATH:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools
```

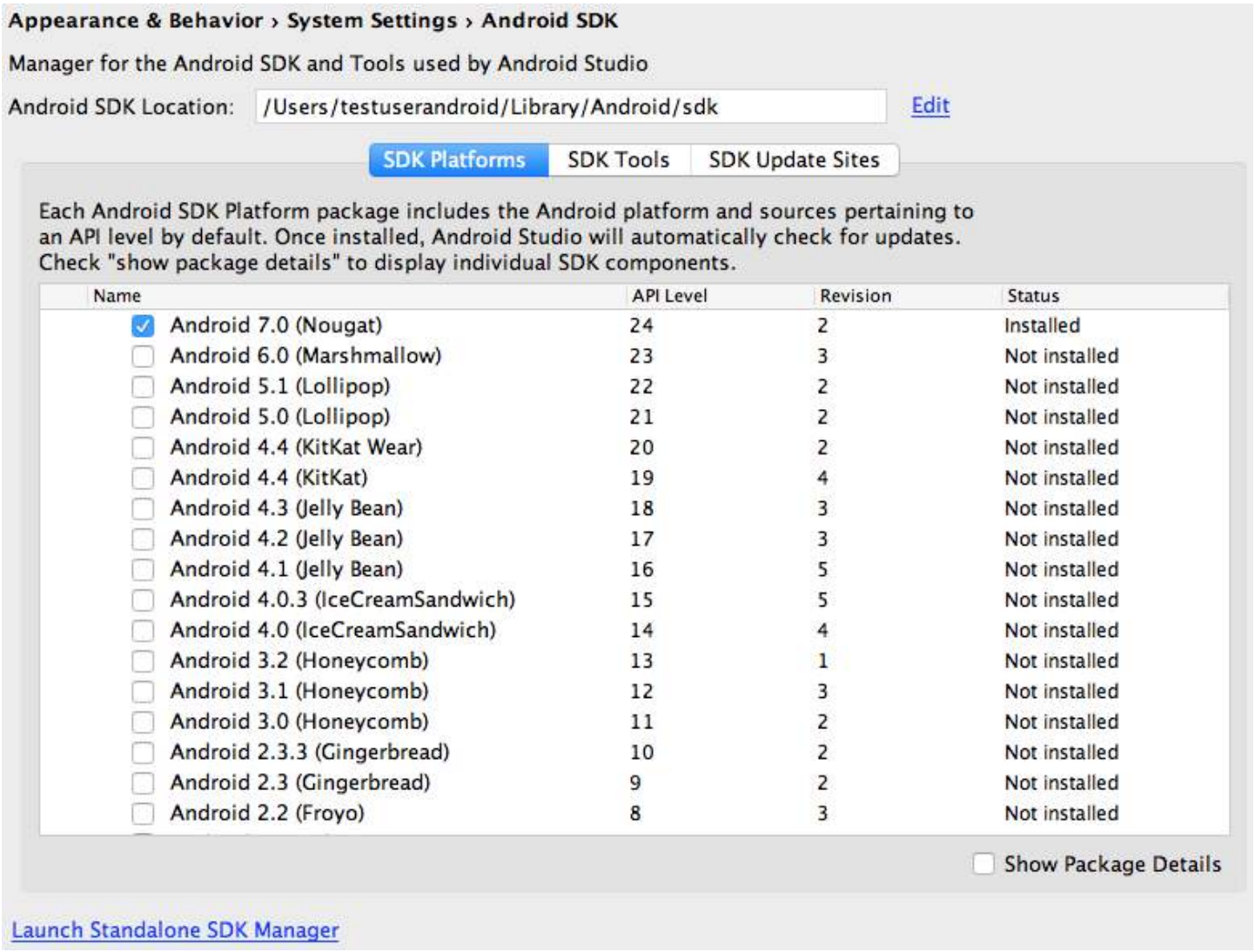
4) Setup emulator:

On the terminal run the command

```
android
```

Select "SDK Platforms" from within the SDK Manager and you should see a blue checkmark next to "Android 7.0

(Nougat)". In case it is not, click on the checkbox and then "Apply".



5) Start a project

Example app init

```
react-native init ReactNativeDemo && cd ReactNativeDemo
```

Obs: Always check if the version on `android/app/build.gradle` is the same as the Build Tools downloaded on your android SDK

```
android {
    compileSdkVersion XX
    buildToolsVersion "XX.X.X"
...
```

6) Run the project

Open Android AVD to set up a virtual android. Execute the command line:

```
android avd
```

Follow the instructions to create a virtual device and start it

Open another terminal and run the command lines:

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

```
react-native start
```

Section 1.3: Setup for Windows

Note: You cannot develop react-native apps for iOS on Windows, only react-native android apps.

The official setup docs for react-native on windows can be found here. If you need more details there is a granular guide here.

Tools/Environment

- Windows 10
- command line tool (eg Powershell or windows command line)
- Chocolatey (steps to setup via PowerShell)
- The JDK (version 8)
- Android Studio
- An Intel machine with Virtualization technology enabled for HAXM (optional, only needed if you want to use an emulator)

1) Setup your machine for react native development

Start the command line as an administrator run the following commands:

```
choco install nodejs.install
choco install python2
```

Restart command line as an administrator so you can run npm

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

After running the last command copy the directory that react-native was installed in. You will need this for Step 4. I tried this on two computers in one case it was: `C:\Program Files (x86)\Nodist\v-x64\6.2.2`. In the other it was: `C:\Users\admin\AppData\Roaming\npm`

2) Set your Environment Variables

A Step by Step guide with images can be found here for this section.

Open the Environment Variables window by navigating to:

[Right click] "Start" menu -> System -> Advanced System Settings -> Environment Variables

In the bottom section find the "Path" System Variable and add the location that react-native was installed to in step 1.

If you haven't added an ANDROID_HOME environment variable you will have to do that here too. While still in the "Environment Variables" window, add a new System Variable with the name "ANDROID_HOME" and value as the path to your android sdk.

Then restart the command line as an admin so you can run react-native commands in it.

3) Create your project In command line, navigate to the folder you want to place your project and run the following command:

```
react-native init ProjectName
```

4) Run your project Start an emulator from android studio Navigate to the root directory of your project in command line and run it:

```
cd ProjectName
react-native run-android
```

You may run into dependency issues. For example, there may be an error that you do not have the correct build tools version. To fix this you will have to open the sdk manager in Android Studio and download the build tools from there.

Congrats!

To refresh the ui you can press the `r` key twice while in the emulator and running the app. To see developer options you can press `ctrl + m`.

Chapter 2: Hello World

Section 2.1: Editing index.ios.js or index.android.js

Open `index.ios.js` or `index.android.js` and delete everything between the **`<View> </View>`**. After that, write **`<Text>`** `Hello World!` **`</Text>`** and run the emulator.

You should see `Hello World!` written on the screen!

Congrats! You've successfully written your first Hello World!

Section 2.2: Hello world!

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

class HelloWorldApp extends Component {
  render() {
    return (
      <Text>Hello world!</Text>
    );
  }
}

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

Chapter 3: Props

Props, or properties, are data that is passed to child components in a React application. React components render UI elements based on their props and their internal state. The props that a component takes (and uses) defines how it can be controlled from the outside.

Section 3.1: PropTypes

The `prop-types` package allows you to add runtime type checking to your component that ensures the types of the props passed to the component are correct. For instance, if you don't pass a `name` or `isYummy` prop to the component below it will throw an error in development mode. In production mode the prop type checks are not done. Defining `propTypes` can make your component more readable and maintainable.

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

import styles from './styles.js';

class Recipe extends Component {
  static propTypes = {
    name: PropTypes.string.isRequired,
    isYummy: PropTypes.bool.isRequired
  }
  render() {
    return (
      <View style={styles.container}>
        <Text>{this.props.name}</Text>
        {this.props.isYummy ? <Text>THIS RECIPE IS YUMMY</Text> : null}
      </View>
    )
  }
}

AppRegistry.registerComponent('Recipe', () => Recipe);


// Using the component
<Recipe name="Pancakes" isYummy={true} />
```

Multiple PropTypes

You can also have multiple `propTypes` for one props. For example, the name props I'm taking can also be an object, I can write it as.

```
static propTypes = {
  name: PropTypes.oneOfType([
      PropTypes.string,
      PropTypes.object
  ])
}
```

Children Props

There is also a special props called `children`, which is **not** passed in like

```
<Recipe children={something}/>
```

Instead, you should do this

```
<Recipe>
  <Text>Hello React Native</Text>
</Recipe>
```

then you can do this in Recipe's render:

```
return (
  <View style={styles.container}>
    {this.props.children}
    {this.props.isYummy ? <Text>THIS RECIPE IS YUMMY</Text> : null}
  </View>
)
```

You will have a **<Text>** component in your `Recipe` saying `Hello React Native`, pretty cool hum?

And the propType of children is

```
children: PropTypes.node
```

Section 3.2: What are props?

Props are used to transfer data from parent to child component. Props are read only. Child component can only get the props passed from parent using **this.props.keyName**. Using props one can make his component reusable.

Section 3.3: Use of props

Once setup is completed. Copy the code below to `index.android.js` or to `index.ios.js` file to use the props.

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

class Greeting extends Component {
  render() {
    return (
      <Text>Hello {this.props.name}!</Text>
    );
  }
}

class LotsOfGreetings extends Component {
  render() {
    return (
      <View style={{alignItems: 'center'}}>
        <Greeting name='Rexxar' />
        <Greeting name='Jaina' />
        <Greeting name='Valeera' />
      </View>
    );
  }
}

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

Using props one can make his component generic. For example, you have a Button component. You can pass different props to that component, so that one can place that button anywhere in his view.

source: Props-React Native

Section 3.4: Default Props

defaultProps allows you to set default prop values for your component. In the below example if you do not pass the name props, it will display John otherwise it will display the passed value

```
class Example extends Component {
  render() {
    return (
      <View>
        <Text>{this.props.name}</Text>
      </View>
    )
  }
}


Example.defaultProps = {
  name: 'John'
}
```

Chapter 4: Multiple props rendering

Section 4.1: render multiple variables

For rendering multiple props or variables we can use ``.

```
render() {
  let firstName = 'test';
  let lastName = 'name';
  return (
    <View style={styles.container}>
      <Text>{`${firstName} ${lastName}` } </Text>
    </View>
  );
}
```

Output: test name

Chapter 5: Modal

Prop	details
animationType	it's an enum of ('**none**', '**slide**', '**fade**') and it controls modal animation.
visible	its a bool that controls modal visiblity.
onShow	it allows passing a function that will be called once the modal has been shown.
transparent	bool to set transparency.
onRequestClose (**android**)	it always defining a method that will be called when user tabs back button
onOrientationChange (**IOS**)	it always defining a method that will be called when orientation changes
supportedOrientations (**IOS**)	enum('portrait', 'portrait-upside-down', 'landscape', 'landscape-left', 'landscape-right')

Modal component is a simple way to present content above an enclosing view.

Section 5.1: Modal Basic Example

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

const styles = StyleSheet.create({
  mainContainer: {
    marginTop: 22,
  },
  modalContainer: {
    marginTop: 22,
  },
});

class Example extends Component {
  constructor() {
    super();
    this.state = {
      visibility: false,
    };
  }


  setModalVisibility(visible) {
    this.setState({
      visibility: visible,
    });
  }

  render() {
    return (
      <View style={styles.mainContainer}>
        <Modal
          animationType={'slide'}
          transparent={false}
          visible={this.state.visibility}
        >
          <View style={styles.modalContainer}>
            <View>
              <Text>I'm a simple Modal</Text>
              <Button
```

```
                color="#000"
                onPress={() => this.setModalVisibility(!this.state.visibility)}
                title="Hide Modal"
              />
            </View>
          </View>
        </Modal>

        <Button
          color="#000"
          onPress={() => this.setModalVisibility(true)}
          title="Show Modal"
        />
      </View>
    );
  }
}

export default Example;
```

Section 5.2: Transparent Modal Example

See this example here.

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

export default class App extends Component {
  state = {
    modalVisible: false,
  };

  _handleButtonPress = () => {
    this.setModalVisible(true);
  };

  setModalVisible = (visible) => {
    this.setState({modalVisible: visible});
  }

  render() {
    var modalBackgroundStyle = {
      backgroundColor: 'rgba(0, 0, 0, 0.5)'
    };
    var innerContainerTransparentStyle = {backgroundColor: '#fff', padding: 20};
    return (
      <View style={styles.container}>
      <Modal
          animationType='fade'
          transparent={true}
          visible={this.state.modalVisible}
          onRequestClose={() => this.setModalVisible(false)}
          >
          <View style={[styles.container, modalBackgroundStyle]}>
            <View style={innerContainerTransparentStyle}>
              <Text>This is a modal</Text>
              <Button title='close'
                onPress={this.setModalVisible.bind(this, false)}/>
            </View>
          </View>
```

```
        </Modal>
        <Button
          title="Press me"
          onPress={this._handleButtonPress}
        />

      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
    paddingTop: Constants.statusBarHeight,
    backgroundColor: '#ecf0f1',
  }
});
```

Chapter 6: State

Section 6.1: setState

To change view in your application you can use `setState` - this will re-render your component and any of its child components. setState performs a shallow merge between the new and previous state, and triggers a re-render of the component.

`setState` takes either a key-value object or a function that returns a key-value object

Key-Value Object

```
this.setState({myKey: 'myValue'});
```

Function

Using a function is useful for updating a value based off the existing state or props.

```
this.setState((previousState, currentProps) => {
    return {
        myInteger: previousState.myInteger+1
    }
})
```

You can also pass an optional callback to `setState` that will be fired when the component has re-rendered with the new state.

```
this.setState({myKey: 'myValue'}, () => {
    // Component has re-rendered... do something amazing!
));
```

Full Example

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

export default class MyParentComponent extends Component {
  constructor(props) {
    super(props);

    this.state = {
      myInteger: 0
    }

  }
  getRandomInteger() {
    const randomInt = Math.floor(Math.random()*100);

    this.setState({
      myInteger: randomInt
    });

  }
  incrementInteger() {

    this.setState((previousState, currentProps) => {
      return {
        myInteger: previousState.myInteger+1
      }
```

```
    });

  }
  render() {

    return <View style={styles.container}>

      <Text>Parent Component Integer: {this.state.myInteger}</Text>

      <MyChildComponent myInteger={this.state.myInteger} />

      <Button label="Get Random Integer" onPress={this.getRandomInteger.bind(this)} />
      <Button label="Increment Integer" onPress={this.incrementInteger.bind(this)} />

    </View>

  }
}

export default class MyChildComponent extends Component {
  constructor(props) {
    super(props);
  }
  render() {

    // this will get updated when "MyParentComponent" state changes
    return <View>
      <Text>Child Component Integer: {this.props.myInteger}</Text>
    </View>

  }
}

export default class Button extends Component {
  constructor(props) {
    super(props);
  }
  render() {

    return <TouchableOpacity onPress={this.props.onPress}>
        <View style={styles.button}>
          <Text style={styles.buttonText}>{this.props.label}</Text>
        </View>
      </TouchableOpacity>

  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  button: {
    backgroundColor: '#444',
    padding: 10,
    marginTop: 10
  },
  buttonText: {
    color: '#fff'
```

```
  }
});

AppRegistry.registerComponent('MyApp', () => MyParentComponent);
```

Section 6.2: Initialize State

You should initialize state inside the constructor function of your component like this:

```
export default class MyComponent extends Component {
  constructor(props) {
    super(props);

    this.state = {
      myInteger: 0
    }
  }
  render() {
    return  (
      <View>
        <Text>Integer: {this.state.myInteger}</Text>
      </View>
    )
  }
}
```

Using setState one can update the view.

Chapter 7: Routing

Routing or navigation allows applications to between different screens. Its vital to a mobile app as it provides context to user about where they are, decouple user actions between screens and move between them, provide a state machine like model of the whole app.

Section 7.1: Navigator component

Navigator works for both IOS and android.

```
import React, { Component } from 'react';
import { Text, Navigator, TouchableHighlight } from 'react-native';

export default class NavAllDay extends Component {
  render() {
    return (
      <Navigator
        initialRoute={{ title: 'Awesome Scene', index: 0 }}
        renderScene={(route, navigator) =>
          <Text>Hello {route.title}!</Text>
        }
        style={{padding: 100}}
      />
    );
  }
}
```

Routes to `Navigator` are provided as objects. You also provide a `renderScene` function that renders the scene for each route object. `initialRoute` is used to specify the first route.

Chapter 8: Styling

Styles are defined within a JSON object with similar styling attribute names like in CSS. Such an object can either be put inline in the style prop of a component or it can be passed to the function `StyleSheet.create(StyleObject)` and be stored in a variable for shorter inline access by using a selector name for it similar to a class in CSS.

Section 8.1: Conditional Styling

```
<View style={[(this.props.isTrue) ? styles.bgcolorBlack : styles.bgColorWhite]}>
```

If the value of `isTrue` is **`true`** then it will have black background color otherwise white.

Section 8.2: Styling using inline styles

Each React Native component can take a `style` prop. You can pass it a JavaScript object with CSS-style style properties:

```
<Text style={{color:'red'}}>Red text</Text>
```

This can be inefficient as it has to recreate the object each time the component is rendered. Using a stylesheet is preferred.

Section 8.3: Styling using a stylesheet

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

const styles = StyleSheet.create({
    red: {
        color: 'red'
    },
    big: {
        fontSize: 30
    }
});

class Example extends Component {
    render() {
        return (
            <View>
                <Text style={styles.red}>Red</Text>
                <Text style={styles.big}>Big</Text>
            </View>
        );
    }
}
```

`StyleSheet.create()` returns an object where the values are numbers. React Native knows to convert these numeric IDs into the correct style object.

Section 8.4: Adding multiple styles

You can pass an array to the `style` prop to apply multiple styles. When there is a conflict, the last one in the list takes precedence.

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

const styles = StyleSheet.create({
    red: {
        color: 'red'
    },
    greenUnderline: {
        color: 'green',
        textDecoration: 'underline'
    },
    big: {
        fontSize: 30
    }
});

class Example extends Component {
    render() {
        return (
            <View>
                <Text style={[styles.red, styles.big]}>Big red</Text>
                <Text style={[styles.red, styles.greenUnderline]}>Green underline</Text>
                <Text style={[styles.greenUnderline, styles.red]}>Red underline</Text>
                <Text style={[styles.greenUnderline, styles.red, styles.big]}>Big red
underline</Text>
                <Text style={[styles.big, {color:'yellow'}]}>Big yellow</Text>
            </View>
        );
    }
}
```

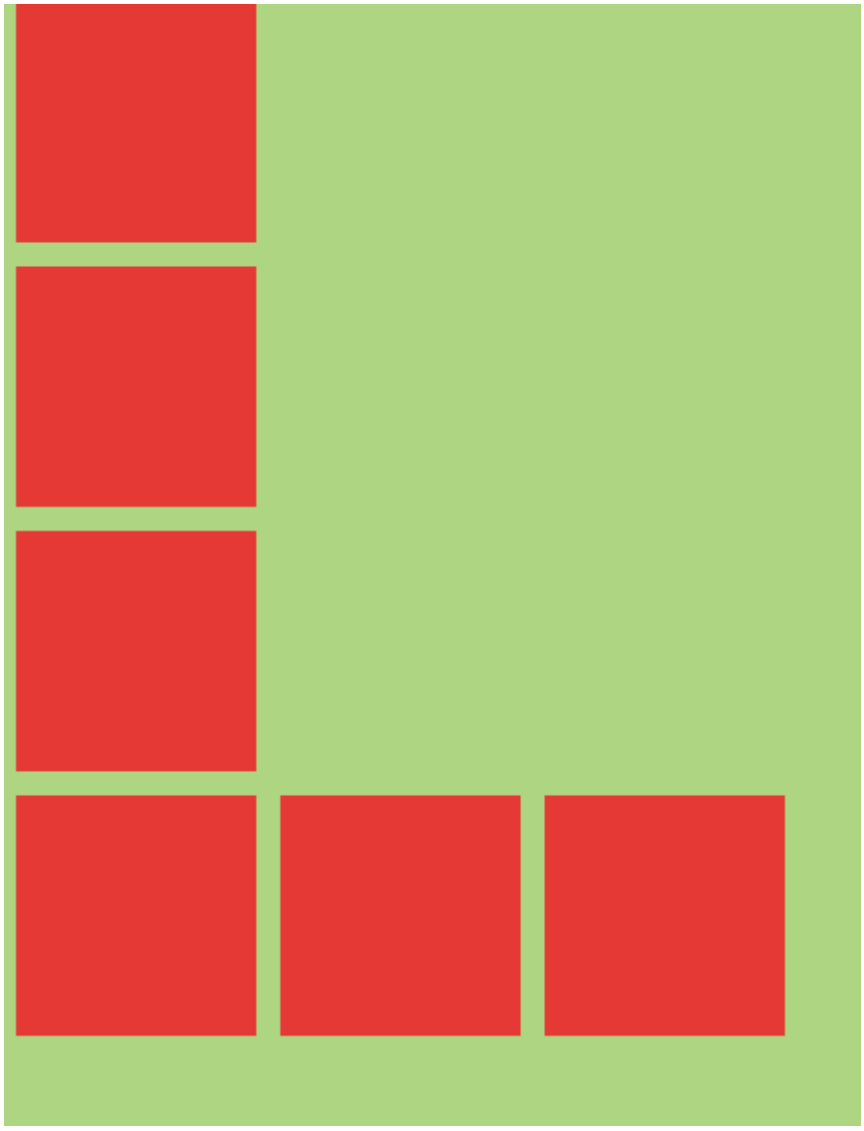
Chapter 9: Layout

Section 9.1: Flexbox

Flexbox is a layout mode providing for the arrangement of elements on a page such that the elements behave predictably when the page layout must accommodate different screen sizes and different display devices. By default flexbox arranges children in a column. But you can change it to row using `flexDirection: 'row'`.

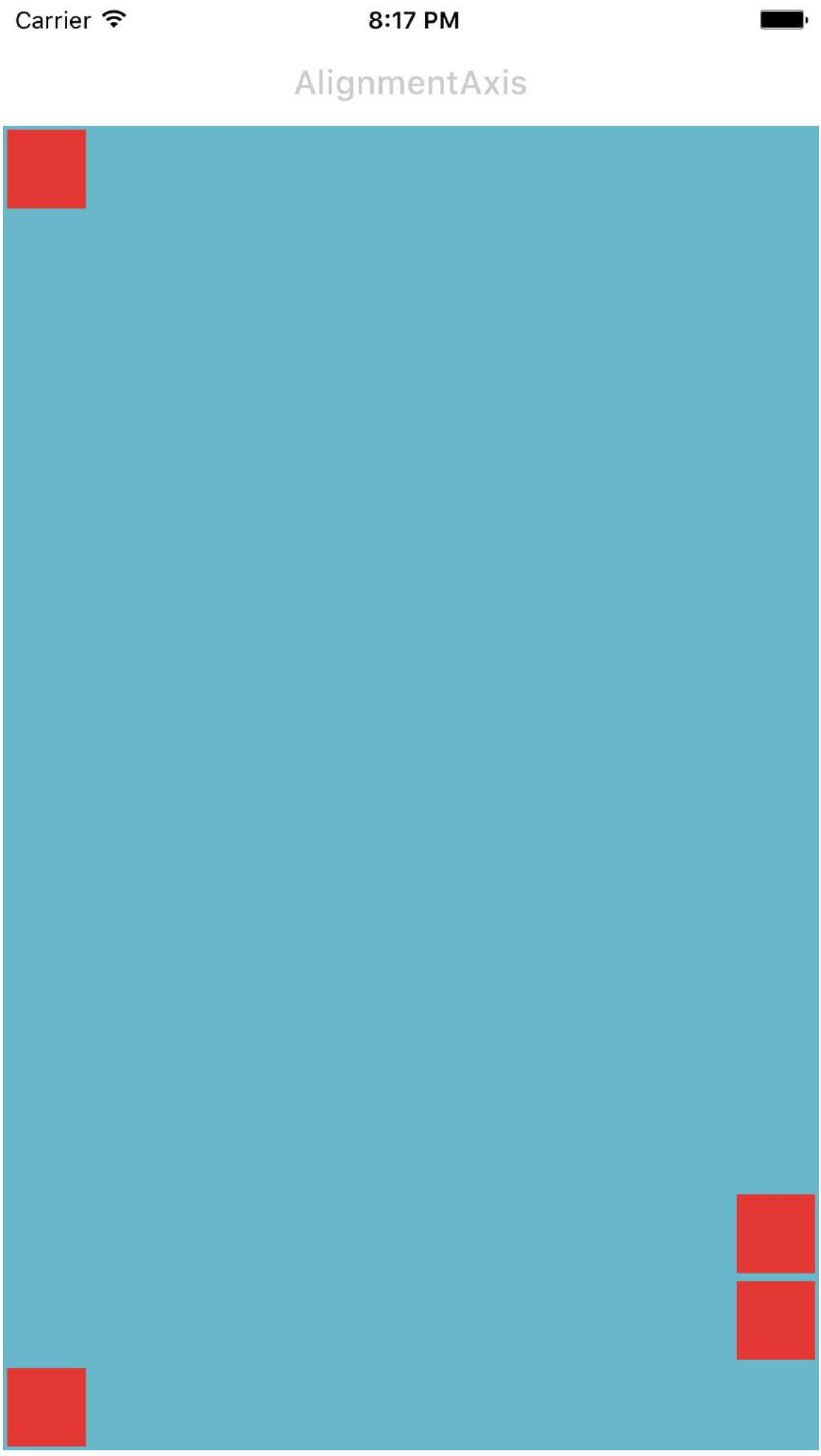
flexDirection

```
const Direction = (props)=>{
  return (
    <View style={styles.container}>
      <Box/>
      <Box/>
      <Box/>
      <View style={{flexDirection:'row'}}>
        <Box/>
        <Box/>
        <Box/>
      </View>
    </View>
  )
}

const styles = StyleSheet.create({
  container: {
    flex:1,
    backgroundColor: '#AED581',
  }
});
```



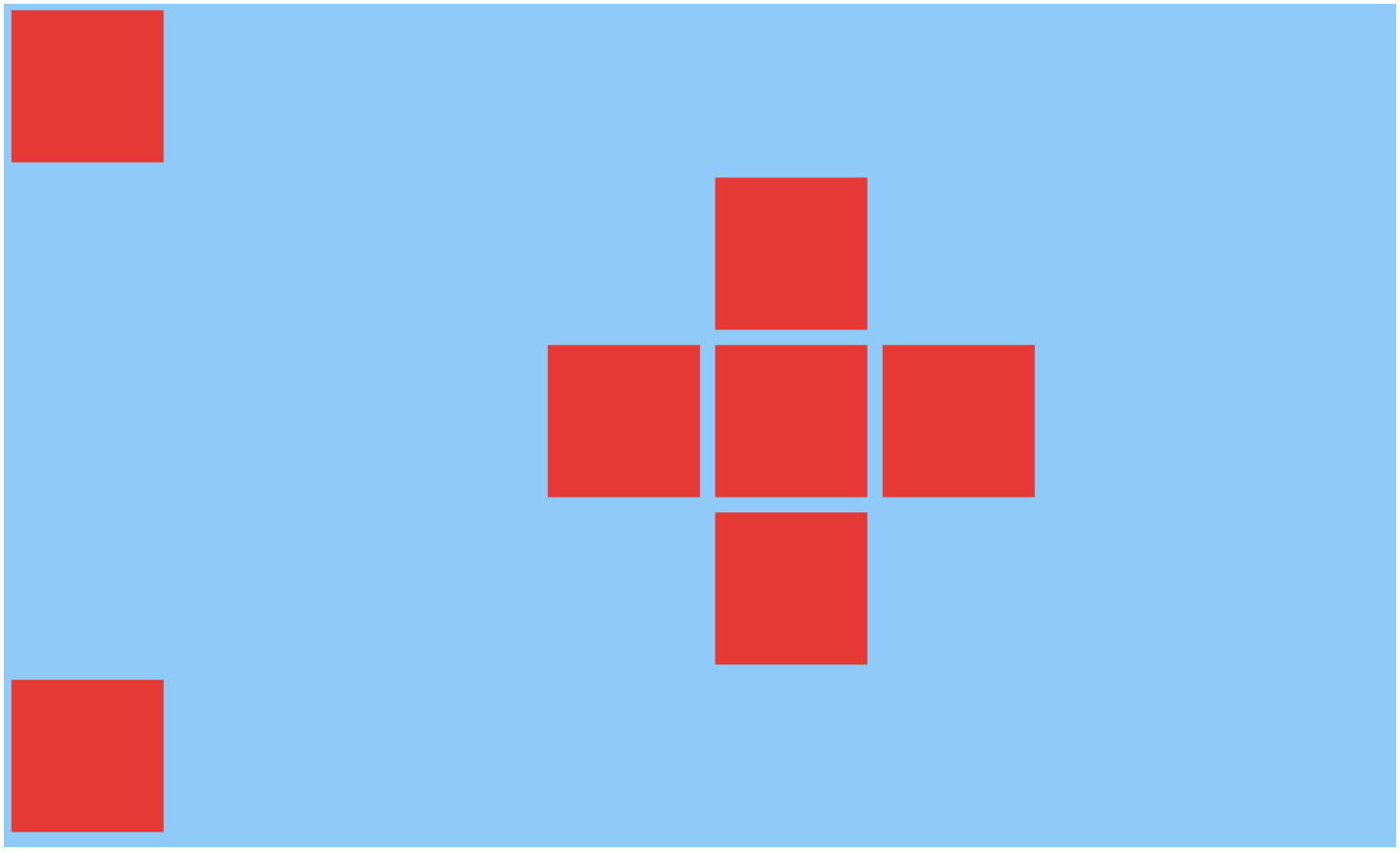
Alignment axis

```
const AlignmentAxis = (props)=>{
  return (
    <View style={styles.container}>
      <Box />
      <View style={{flex:1, alignItems:'flex-end', justifyContent:'flex-end'}}>
        <Box />
        <Box />
      </View>
      <Box />
    </View>
  )
}

const styles = StyleSheet.create({
  container: {
    flex:1,
    backgroundColor: `#69B8CC`,
  },
  text:{
    color: 'white',
    textAlign:'center'
  }
});
```


Carrier
8:17 PM
AlignmentAxis

Alignment

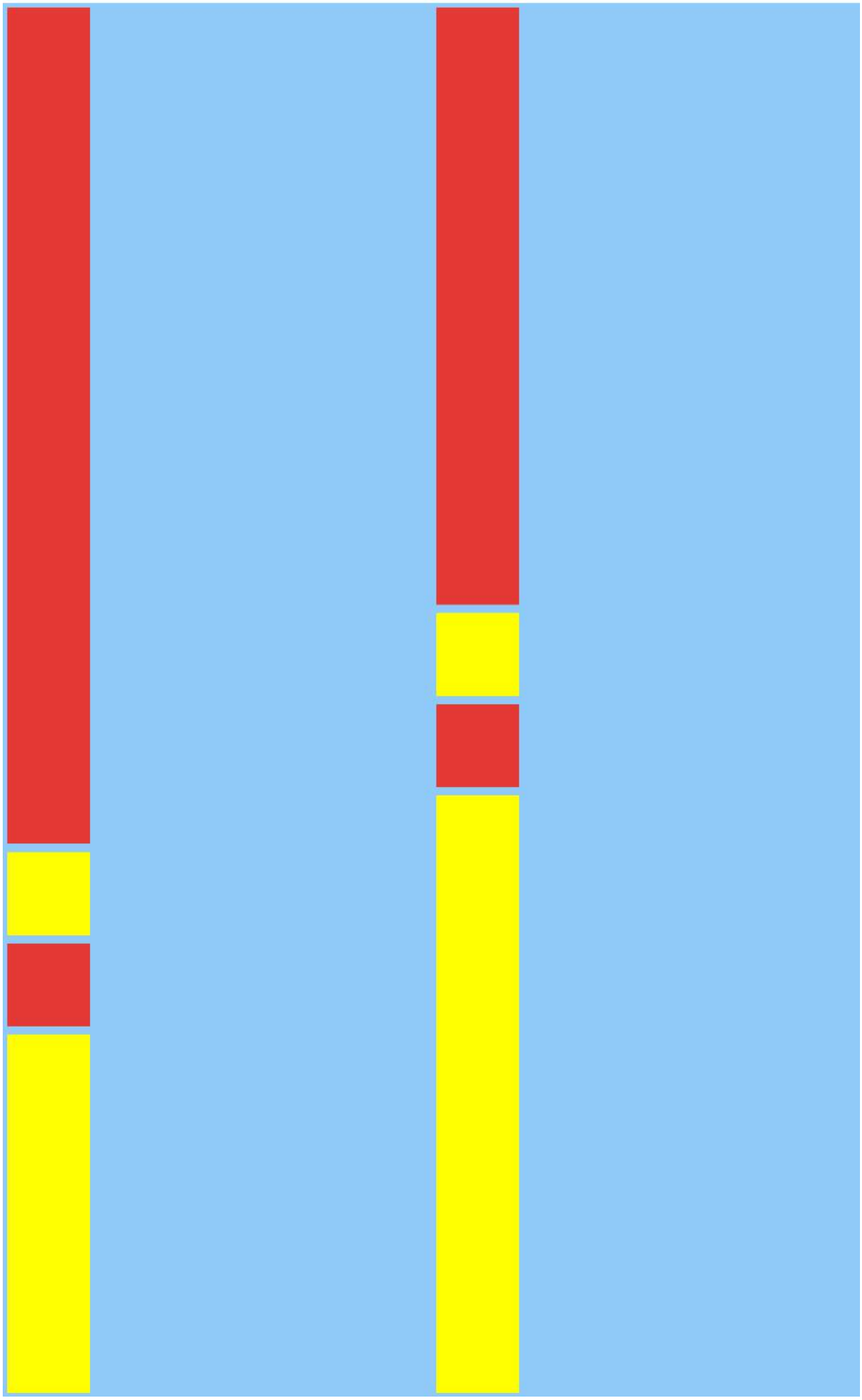
```
const Alignment = (props)=>{
  return (
    <View style={styles.container}>
      <Box/>
      <View style={{alignItems:'center'}}>
        <Box/>
        <View style={{flexDirection:'row'}}>
          <Box/>
          <Box/>
          <Box/>
        </View>
        <Box/>
      </View>
      <Box/>
    </View>
  )
}

const styles = StyleSheet.create({
  container: {
    flex:1,
    backgroundColor: `#69B8CC`,
  },
  text:{
    color: 'white',
    textAlign:'center'
  }
});
```



Flex size

```
const FlexSize = (props)=>{
  return (
```

```
    <View style={styles.container}>
      <View style={{flex:0.1}}>
        <Box style={{flex:0.7}}/>
        <Box style={{backgroundColor: 'yellow'}}/>
        <Box/>
        <Box style={{flex:0.3, backgroundColor: 'yellow'}}/>
      </View>
      <View style={{flex:0.1}}>
        <Box style={{flex:1}}/>
        <Box style={{backgroundColor: 'yellow'}}/>
        <Box/>
        <Box style={{flex:1, backgroundColor: 'yellow'}}/>
      </View>
    </View>
  )
}

const styles = StyleSheet.create({
  container: {
    flex:1,
    flexDirection:'row',
    backgroundColor: colors[1],
  },
});
```



More about Facebook's flexbox implementation here.

Chapter 10: Components

Section 10.1: Basic Component

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

class Example extends Component {
  render () {
    return (
      <View>
        <Text> I'm a basic Component </Text>
      </View>
    )
  }
}

AppRegistry.registerComponent('Example', () => Example)
```

Section 10.2: Stateful Component

These components will have changing States.

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

class Example extends Component {
  constructor (props) {
    super(props)
    this.state = {
      name: "Sriraman"
    }
  }
  render () {
    return (
      <View>
        <Text> Hi, {this.state.name}</Text>
      </View>
    )
  }
}

AppRegistry.registerComponent('Example', () => Example)
```

Section 10.3: Stateless Component

As the name implies, Stateless Components do not have any local state. They are also known as **Dumb Components**. Without any local state, these components do not need lifecycle methods or much of the boilerplate that comes with a stateful component.

Class syntax is not required, you can simply do `const name = ({props}) => ( ... )`. Generally stateless components are more concise as a result.

Beneath is an example of two stateless components `App` and `Title`, with a demonstration of passing props between components:

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

const Title = ({Message}) => (
  <Text>{Message}</Text>
)

const App = () => (
  <View>
    <Title title='Example Stateless Component' />
  </View>
)

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

This is the recommended pattern for components, when possible. As in the future optimisations can be made for these components, reducing memory allocations and unnecessary checks.

Chapter 11: ListView

Section 11.1: Simple Example

ListView - A core component designed for efficient display of vertically scrolling lists of changing data. The minimal API is to create a ListView.DataSource, populate it with a simple array of data blobs, and instantiate a ListView component with that data source and a renderRow callback which takes a blob from the data array and returns a renderable component.

Minimal example:

```
getInitialState: function() {
  var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
  return {
    dataSource: ds.cloneWithRows(['row 1', 'row 2']),
  };
},

render: function() {
  return (
    <ListView
      dataSource={this.state.dataSource}
      renderRow={(rowData) => <Text>{rowData}</Text>}
    />
  );
},
```

ListView also supports more advanced features, including sections with sticky section headers, header and footer support, callbacks on reaching the end of the available data (onEndReached) and on the set of rows that are visible in the device viewport change (onChangeVisibleRows), and several performance optimizations.

There are a few performance operations designed to make ListView scroll smoothly while dynamically loading potentially very large (or conceptually infinite) data sets:

- Only re-render changed rows - the rowHasChanged function provided to the data source tells the ListView if it needs to re-render a row because the source data has changed - see ListViewDataSource for more details.
- Rate-limited row rendering - By default, only one row is rendered per event-loop (customizable with the pageSize prop). This breaks up the work into smaller chunks to reduce the chance of dropping frames while rendering rows.

Chapter 12: RefreshControl with ListView

Section 12.1: Refresh Control with ListView Full Example

RefreshControl is used inside a ScrollView or ListView to add pull to refresh functionality. at this example we will use it with ListView

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


class RefreshControlExample extends Component {
  constructor () {
    super()
    this.state = {
      refreshing: false,
      dataSource: new ListView.DataSource({
        rowHasChanged: (row1, row2) => row1 !== row2 }),
      cars : [
        {name:'Datsun',color:'White'},
        {name:'Camry',color:'Green'}
      ]
    }
  }

   componentWillMount(){
     this.setState({ dataSource:
       this.state.dataSource.cloneWithRows(this.state.cars) })
   }

  render() {
    return (
      <View style={{flex:1}}>
        <ListView
          refreshControl={this._refreshControl()}
          dataSource={this.state.dataSource}
          renderRow={(car) => this._renderListView(car)}>
        </ListView>
      </View>
    )
  }

  _renderListView(car){
    return(
      <View style={styles.listView}>
        <Text>{car.name}</Text>
        <Text>{car.color}</Text>
      </View>
    )
  }

  _refreshControl(){
    return (
      <RefreshControl
        refreshing={this.state.refreshing}
        onRefresh={()=>this._refreshListView()} />
    )
  }
```

```
  _refreshListView(){
    //Start Rendering Spinner
    this.setState({refreshing:true})
    this.state.cars.push(
      {name:'Fusion',color:'Black'},
      {name:'Yaris',color:'Blue'}
    )
    //Updating the dataSource with new data
    this.setState({ dataSource:
        this.state.dataSource.cloneWithRows(this.state.cars) })
    this.setState({refreshing:false}) //Stop Rendering Spinner
  }

}

const styles = StyleSheet.create({

  listView: {
    flex: 1,
    backgroundColor:'#fff',
    marginTop:10,
    marginRight:10,
    marginLeft:10,
    padding:10,
    borderWidth:.5,
    borderColor:'#dddddd',
    height:70
  }

})

module.exports = RefreshControlExample
```

Section 12.2: Refresh Control

```
  _refreshControl(){
    return (
      <RefreshControl
        refreshing={this.state.refreshing}
        onRefresh={()=>this._refreshListView()} />
    )
  }
```

refreshing: is the state of the spinner (true, false).

onRefresh: this function will invoke when refresh the ListView/ScrollView.

Section 12.3: onRefresh function Example

```
  _refreshListView(){
    //Start Rendering Spinner
    this.setState({refreshing:true})
    this.state.cars.push(
      {name:'Fusion',color:'Black'},
      {name:'Yaris',color:'Blue'}
    )
    //Updating the dataSource with new data
    this.setState({ dataSource:
        this.state.dataSource.cloneWithRows(this.state.cars) })
    this.setState({refreshing:false}) //Stop Rendering Spinner
```

```
  }
```

here we are updating the array and after that we will update the dataSource. we can use fetch to request something from server and use async/await.

Chapter 13: WebView

Webview can be used to load external webpages or html content. This component is there by default.

Section 13.1: Simple component using webview

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

class MyWeb extends Component {
  render() {
    return (
      <WebView
        source={{uri: 'https://github.com/facebook/react-native'}}
        style={{marginTop: 20}}
       />
    );
  }
}
```

Chapter 14: Command Line Instructions

Section 14.1: Check version installed

```
$ react-native -v
```

Example Output

```
react-native-cli: 0.2.0
react-native: n/a - not inside a React Native project directory //Output from  different folder
react-native: react-native: 0.30.0 // Output from the react native project directory
```

Section 14.2: Initialize and getting started with React Native project

To initialize

```
react-native init MyAwesomeProject
```

To initialize with a specific version of React Native

```
react-native init --version="0.36.0" MyAwesomeProject
```

To Run for Android

```
cd MyAwesomeProject
react-native run-android
```

To Run for iOS

```
cd MyAwesomeProject
react-native run-ios
```

Section 14.3: Upgrade existing project to latest RN version

In the app folder find `package.json` and modify the following line to include the latest version, save the file and close.

```
"react-native": "0.32.0"
```

In terminal:

```
$ npm install
```

Followed by

```
$ react-native upgrade
```

Section 14.4: Add android project for your app

If you either have apps generated with pre-android support or just did that on purpose, you can always add android project to your app.

```
$ react-native android
```

This will generate `android` folder and `index.android.js` inside your app.

Section 14.5: Logging

Android

```
$ react-native log-android
```

iOS

```
$ react-native log-ios
```

Section 14.6: Start React Native Packager

```
$ react-native start
```

On latest version of React Native, no need to run the packager. It will run automatically.

By default this starts the server at port 8081. To specify which port the server is on

```
$ react-native start --port PORTNUMBER
```

Chapter 15: HTTP Requests

Section 15.1: Using Promises with the fetch API and Redux

Redux is the most common state management library used with React-Native. The following example demonstrates how to use the fetch API and dispatch changes to your applications state reducer using redux-thunk.

```
export const fetchRecipes = (action) => {
  return (dispatch, getState) => {
    fetch('/recipes', {
        method: 'POST',
        headers: {
          'Accept': 'application/json',
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({
          recipeName,
          instructions,
          ingredients
        })
    })
    .then((res) => {
      // If response was successful parse the json and dispatch an update
      if (res.ok) {
        res.json().then((recipe) => {
          dispatch({
            type: 'UPDATE_RECIPE',
            recipe
          });
        });
      } else {
        // response wasn't successful so dispatch an error
        res.json().then((err) => {
          dispatch({
            type: 'ERROR_RECIPE',
            message: err.reason,
            status: err.status
          });
        });
      }
    })
    .catch((err) => {
      // Runs if there is a general JavaScript error.
      dispatch(error('There was a problem with the request.'));
    });
  };
};
```

Section 15.2: HTTP with the fetch API

It should be noted that Fetch *does not support progress callbacks*. See: https://github.com/github/fetch/issues/89.

The alternative is to use XMLHttpRequest https://developer.mozilla.org/en-US/docs/Web/Events/progress.

```
fetch('https://mywebsite.com/mydata.json').then(json => console.log(json));

fetch('/login', {
  method: 'POST',
```

```
  body: form,
  mode: 'cors',
  cache: 'default',
}).then(session => onLogin(session), failure => console.error(failure));
```

More details about fetch can be found at MDN

Section 15.3: Networking with XMLHttpRequest

```
var request = new XMLHttpRequest();
request.onreadystatechange = (e) => {
  if (request.readyState !== 4) {
    return;
  }

  if (request.status === 200) {
    console.log('success', request.responseText);
  } else {
    console.warn('error');
  }
};

request.open('GET', 'https://mywebsite.com/endpoint/');
request.send();
```

Section 15.4: WebSockets

```
var ws = new WebSocket('ws://host.com/path');

ws.onopen = () => {
  // connection opened

  ws.send('something'); // send a message
};

ws.onmessage = (e) => {
  // a message was received
  console.log(e.data);
};

ws.onerror = (e) => {
  // an error occurred
  console.log(e.message);
};

ws.onclose = (e) => {
  // connection closed
  console.log(e.code, e.reason);
};
```

Section 15.5: Http with axios

Configure

For web request you can also use library axios.

It's easy to configure. For this purpose you can create file axios.js for example:

```
import * as axios from 'axios';

var instance = axios.create();
instance.defaults.baseURL = serverURL;
instance.defaults.timeout = 20000;]
//...
//and other options

export { instance as default };
```

and then use it in any file you want.

Requests

To avoid using pattern 'Swiss knife' for every service on your backend you can create separate file with methods for this within folder for integration functionality:

```
import axios from '../axios';
import {
    errorHandling
} from '../common';

const UserService = {
        getCallToAction() {
        return axios.get('api/user/dosomething').then(response => response.data)
            .catch(errorHandling);
    },
}
export default UserService;
```

Testing

There is a special lib for testing axios: axios-mock-adapter.

With this lib you can set to axios any responce you want for testing it. Also you can configure some special errors for your axois'es methods. You can add it to your axios.js file created in prevous step:

```
import MockAdapter from 'axios-mock-adapter';

var mock = new MockAdapter(instance);
mock.onAny().reply(500);
```

for example.

Redux Store

Sometimes you need to add to headers authorize token, that you probably store in your redux store.

In this case you'll need another file, interceptors.js with this function:

```
export function getAuthToken(storeContainer) {
    return config => {
        let store = storeContainer.getState();
        config.headers['Authorization'] = store.user.accessToken;
        return config;
    };
}
```

Next in constructor of your root component you can add this:

```
axios.interceptors.request.use(getAuthToken(this.state.store));
```

and then all your requests will be followed with your authorization token.

As you can see axios is very simple, configurable and useful library for applications based on react-native.

Section 15.6: Web Socket with Socket.io

Install *socket.io-client*

```
npm i socket.io-client --save
```

Import module

```
import SocketIOClient from 'socket.io-client/dist/socket.io.js'
```

Initialize in your constructor

```
constructor(props){
    super(props);
    this.socket = SocketIOClient('http://server:3000');
  }
```

Now in order to use your socket connection properly, you should bind your functions in constructor too. Let's assume that we have to build a simple application, which will send a ping to a server via socket after every 5 seconds (consider this as ping), and then the application will get a reply from the server. To do so, let's first create these two functions:

```
_sendPing(){
    //emit a dong message to socket server
    socket.emit('ding');
}

_getReply(data){
    //get reply from socket server, log it to console
    console.log('Reply from server:' + data);
}
```

Now, we need to bind these two functions in our constructor:

```
constructor(props){
    super(props);
    this.socket = SocketIOClient('http://server:3000');

    //bind the functions
    this._sendPing = this._sendPing.bind(this);
    this._getReply = this._getReply.bind(this);
}
```

After that, we also need to link _getReply function with the socket in order to receive the message from the socket server. To do this we need to attach our _getReply function with socket object. Add the following line to our constructor:

```
this.socket.on('dong', this._getReply);
```

Now, whenever socket server emits with the 'dong' your application will able to receive it.

Chapter 16: Platform Module

Section 16.1: Find the OS Type/Version

The first step is to import Platform from the 'react-native' package like so:

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

After you've done that, you can go ahead and access the OS type through `Platform.OS` allowing you to use it in conditional statements like

```
const styles = StyleSheet.create({
  height: (Platform.OS === 'ios') ? 200 : 100,
})
```

If you want to detect the Android version, you can use `Platform.Version` like so:

```
if (Platform.Version === 21) {
  console.log('Running on Lollipop!');
}
```

For iOS, Platform.Version is returning a String, for complex condition don't forget to parse it.

```
if (parseInt(Platform.Version, 10) >= 9) {
    console.log('Running version higher than 8');
}
```

If the platform specific logic is complex, one can render two different files based on platform. Ex:

- `MyTask.android.js`
- `MyTask.ios.js`

and require it using

```
const MyTask = require('./MyTask')
```

Chapter 17: Images

Section 17.1: Image Module

You're going to have to import `Image` from the `react-native` package like so then use it:

```
import { Image } from 'react';

<Image source={{uri: 'https://image-souce.com/awesomeImage'}} />
```

You can also use a local image with a slightly different syntax but same logic like so:

```
import { Image } from 'react';

<Image source={require('./img/myCoolImage.png')} />
```

Note: You should give height, width to the image otherwise it won't show.

Section 17.2: Image Example

```
class ImageExample extends Component {
  render() {
    return (
      <View>
        <Image style={{width: 30, height: 30}}
          source={{uri: 'http://facebook.github.io/react/img/logo_og.png'}}
        />
      </View>
    );
  }
}
```

Section 17.3: Conditional Image Source

```
<Image style={[this.props.imageStyle]}
       source={this.props.imagePath
       ? this.props.imagePath
       : require('../theme/images/resource.png')}
   />
```

If the path is available in `imagePath` then it will be assigned to source else the default image path will be assigned.

Section 17.4: Using variable for image path

```
let imagePath = require("../../assets/list.png");

<Image style={{height: 50, width: 50}} source={imagePath} />
```

From external resource:

```
<Image style={{height: 50, width: 50}} source={{uri: userData.image}} />
```

Section 17.5: To fit an Image

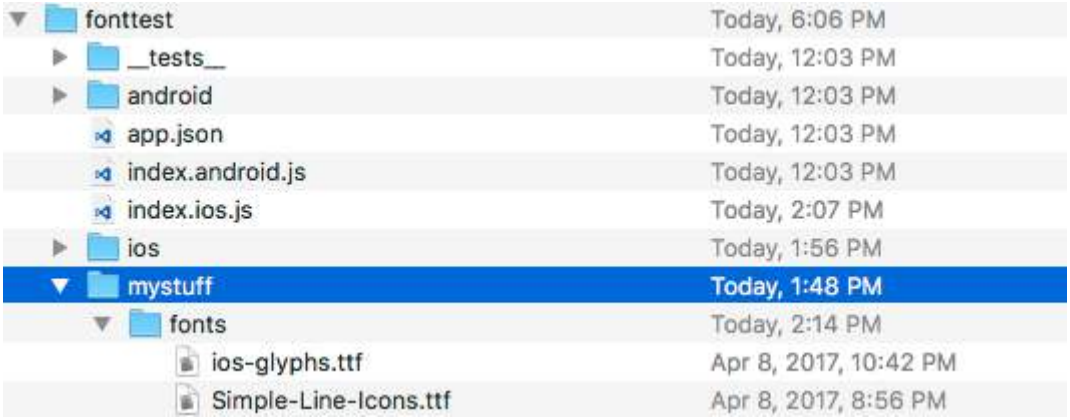
```
<Image
    resizeMode="contain"
    style={{height: 100, width: 100}}
    source={require('../assets/image.png')} />
```

Try also **cover**, **stretch**, **repeat** and **center** parameters.

Chapter 18: Custom Fonts

Section 18.1: Custom fonts for both Android and IOS

- Create a folder in your project folder, and add your fonts to it. Example:
    - Example: Here we added a folder in root called "mystuff", then "fonts", and inside it we placed our fonts:



- Add the below code in `package.json`.

  { ... "rnpm": { "assets": ["path/to/fontfolder"] }, ... }
    - For the example above, our package.json would now have a path of "mystuff/fonts":

```
"rnpm": {
  "assets": [
    "mystuff/fonts"
  ]
}
```

- Run `react-native link` command.
- Using custom fonts on project below code

  <Text style={{ fontFamily: 'FONT-NAME' }}> My Text </Text>

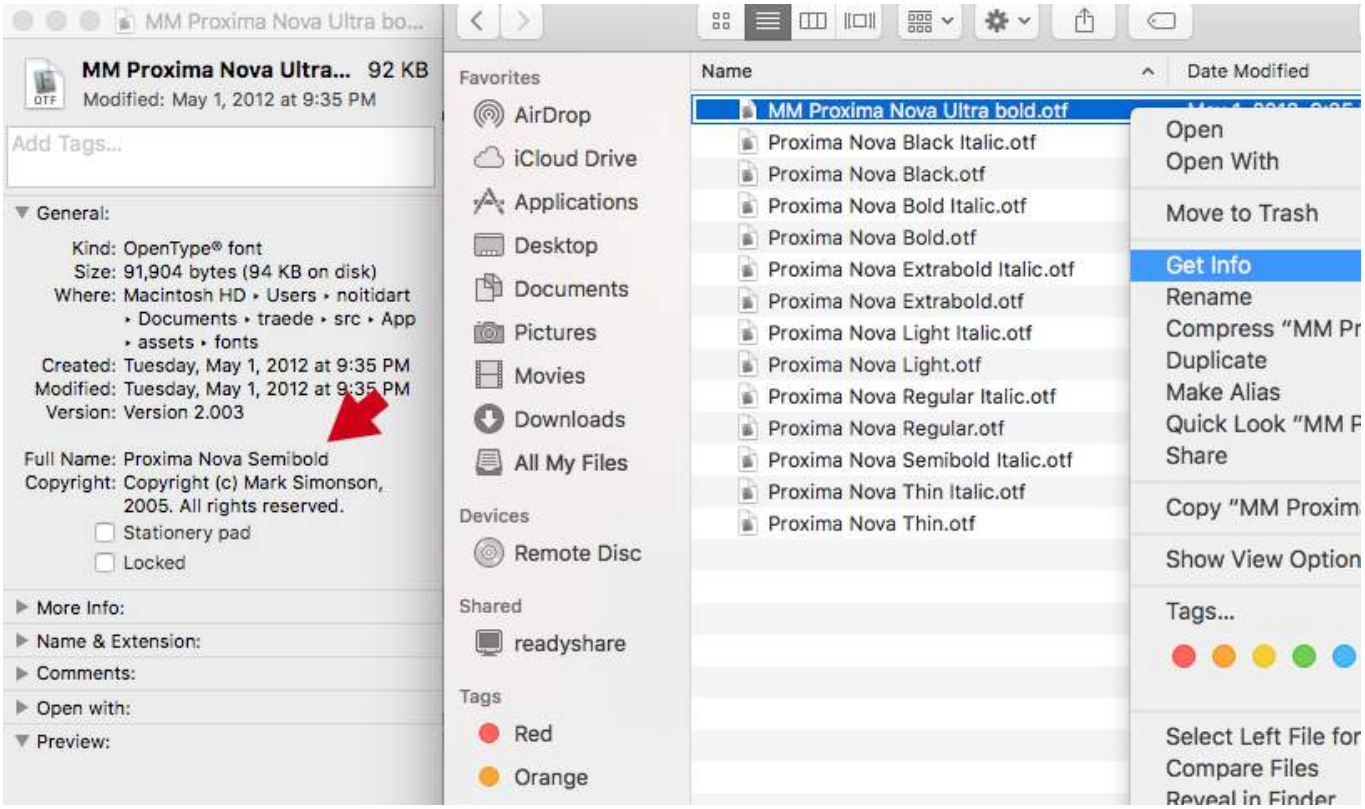
  Where `FONT-NAME` is the prefix platform specific.

  Android

  FONT-NAME is the words before the extension in file. Example: Your font's file name is `Roboto-Regular.ttf`, so you would set `fontFamily: Roboto-Regular`.

  iOS

  FONT-NAME is "Full Name" found after right clicking, on the font file, then clicking on "Get Info". (Source: https://stackoverflow.com/a/16788493/2529614), in the screenshot below, the file name is `MM Proxima Nova Ultra bold.otf`, however "Full Name" is "Proxima Nova Semibold", thus you would set `fontFamily: Proxima Nova Semibold`. Screenshot -



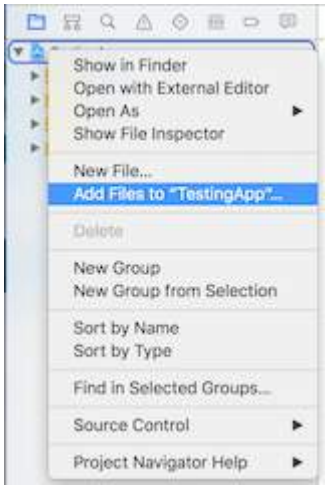
- Run `react-native run-ios` or `react-native run-android` again (this will recompile with the resources)

Section 18.2: Steps to use custom fonts in React Native (Android)

1. Paste your fonts file inside `android/app/src/main/assets/fonts/font_name.ttf`
2. Recompile the Android app by running `react-native run-android`
3. Now, You can use `fontFamily: 'font_name'` in your React Native Styles

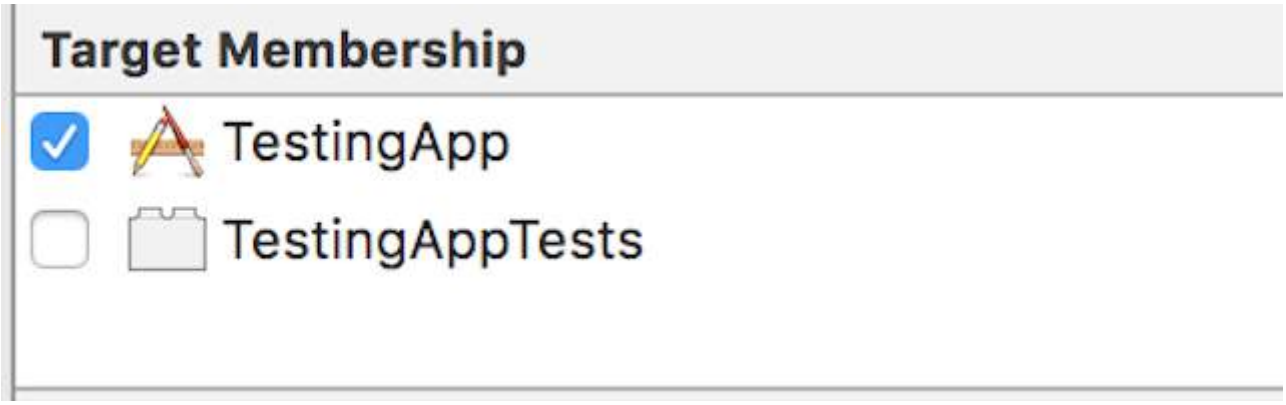
Section 18.3: Steps to use custom fonts in React Native (iOS)

1. Include the font in your Xcode project.



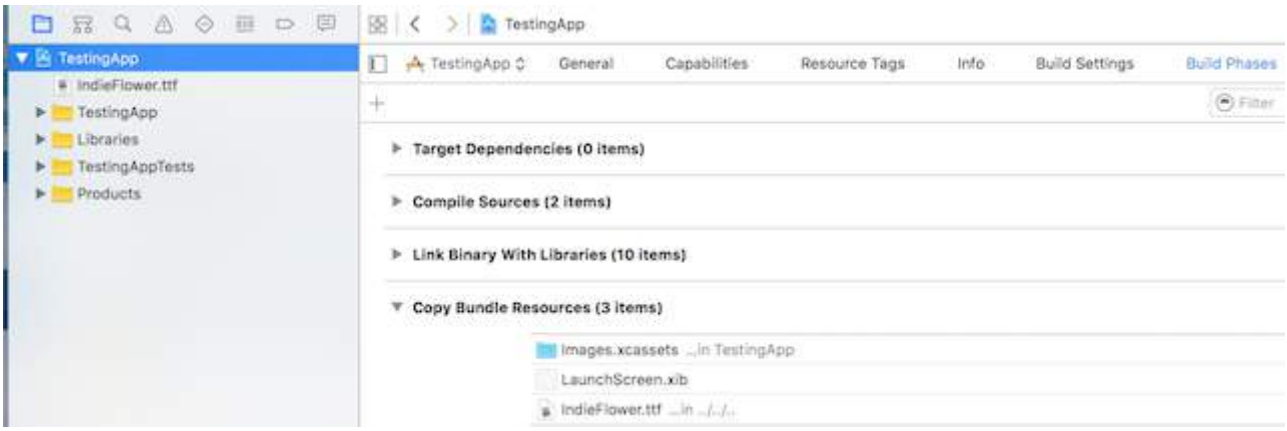
2. Make sure that they are included in the Target Membership column

Click on the font from the navigator, and check if the font included.



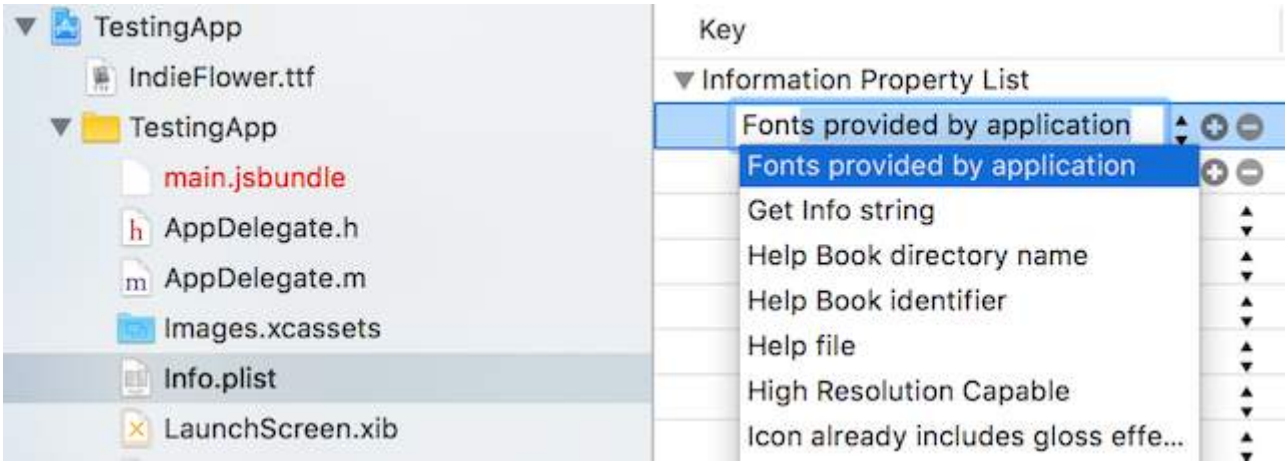
3. Check if the font included as Resource in your bundle

click on your Xcode project file, select "Build Phases, select "Copy Bundle Resources". Check if your font is added.



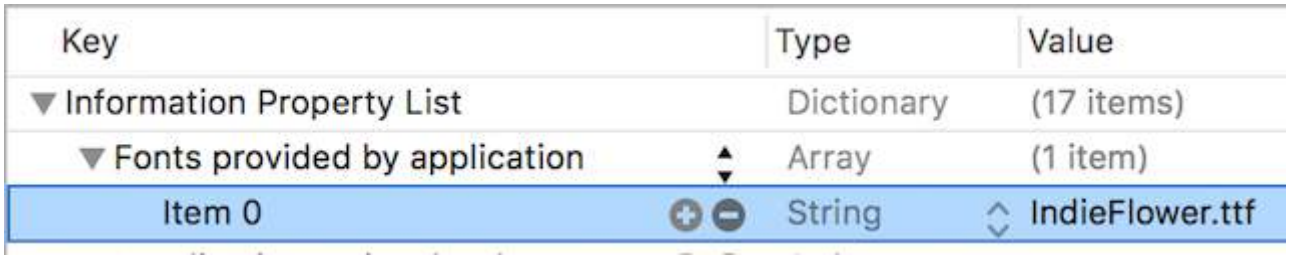
4. Include the font in Application Plist (Info.plist)

from the application main folder open Info.plist, click on "Information Property List", and then click the plus sign (+). from drop down list choose "Fonts provided by application".



5. Add Font name in Fonts provided by application

expand Fonts Provided by Application and add the Font Name exactly to value column



6. Use it in the Application

```
<Text style={{fontFamily:'IndieFlower'}}>
  Welcome to React Native!
</Text>
```

Chapter 19: Animation API

Section 19.1: Animate an Image

```
class AnimatedImage extends Component {
    constructor(props){
        super(props)
        this.state = {
            logoMarginTop: new Animated.Value(200)
        }
    }
    componentDidMount(){
        Animated.timing(
            this.state.logoMarginTop,
            { toValue: 100 }
        ).start()
    }
    render () {
      return (
        <View>
           <Animated.Image source={require('../images/Logo.png')} style={[baseStyles.logo, {
              marginTop: this.state.logoMarginTop
           }]} />
        </View>
      )
    }
}
```

This example is animating the image position by changing the margin.

Chapter 20: Android - Hardware Back Button

Section 20.1: Detect Hardware back button presses in Android

```
BackAndroid.addEventListener('hardwareBackPress', function() {
    if (!this.onMainScreen()) {
        this.goBack();
        return true;
    }
    return false;
});
```

Note: `this.onMainScreen()` and `this.goBack()` are not built in functions, you also need to implement those. (https://github.com/immidi/react-native/commit/ed7e0fb31d842c63e8b8dc77ce795fac86e0f712)

Section 20.2: Example of BackAndroid along with Navigator

This is an example on how to use React Native's `BackAndroid` along with the `Navigator`.

`componentWillMount` registers an event listener to handle the taps on the back button. It checks if there is another view in the history stack, and if there is one, it goes back -otherwise it keeps the default behaviour.

More information on the `BackAndroid` docs and the `Navigator` docs.

```
import React, { Component } from 'react'; // eslint-disable-line no-unused-vars

import {
  BackAndroid,
  Navigator,
} from 'react-native';

import SceneContainer from './Navigation/SceneContainer';
import RouteMapper from './Navigation/RouteMapper';

export default class AppContainer extends Component {

  constructor(props) {
    super(props);

    this.navigator;
  }

  componentWillMount() {
    BackAndroid.addEventListener('hardwareBackPress', () => {
      if (this.navigator && this.navigator.getCurrentRoutes().length > 1) {
        this.navigator.pop();
        return true;
      }
      return false;
    });
  }

  renderScene(route, navigator) {
    this.navigator = navigator;

    return (
```

```
      <SceneContainer
        title={route.title}
        route={route}
        navigator={navigator}
        onBack={() => {
          if (route.index > 0) {
            navigator.pop();
          }
        }}
        {...this.props} />
    );
  }

  render() {
    return (
      <Navigator
        initialRoute={<View />}
        renderScene={this.renderScene.bind(this)}
        navigationBar={
          <Navigator.NavigationBar
            style={{backgroundColor: 'gray'}}
            routeMapper={RouteMapper} />
        } />
    );
  }
};
```

Section 20.3: Hardware back button handling using BackHandler and Navigation Properties (without using deprecated BackAndroid & deprecated Navigator)

This example will show you back navigation which is expected generally in most of the flows. You will have to add following code to every screen depending on expected behavior. There are 2 cases:

1. If there are more than 1 screen on stack, device back button will show previous screen.
2. If there is only 1 screen on stack, device back button will exit app.

Case 1: Show previous screen

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

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

componentWillMount() {
    BackHandler.addEventListener('hardwareBackPress', this.handleBackButtonClick);
}

componentWillUnmount() {
    BackHandler.removeEventListener('hardwareBackPress', this.handleBackButtonClick);
}

handleBackButtonClick() {
    this.props.navigation.goBack(null);
    return true;
}
```

Important: Don't forget to bind method in constructor and to remove listener in componentWillUnmount.

Case 2: Exit App

In this case, no need to handle anything on that screen where you want to exit app.

Important: This should be only screen on stack.

Section 20.4: Example of Hardware back button detection using BackHandler

Since BackAndroid is deprecated. Use BackHandler instead of BackAndroid.

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

{...}
  ComponentWillMount(){
    BackHandler.addEventListener('hardwareBackPress',()=>{
      if (!this.onMainScreen()) {
        this.goBack();
        return true;
      }
      return false;
    });
  }
```

Chapter 21: Run an app on device (Android Version)

Section 21.1: Running an app on Android Device

1. `adb devices`
    - Is your phone displaying? If not, enable developer mode on your phone, and connect it by USB.
2. `adb reverse tcp:8081 tcp:8081` :
    - In order to link correctly your phone and that React-Native recognize him during build. (**NOTE:`Android Version 5` or above.**)
3. `react-native run-android` :
    - To run the app on your phone.
4. `react-native start` :
    - In order to start a local server for development (mandatory). This server is automatically started if you use the last version of React-native.

Chapter 22: Native Modules

Section 22.1: Create your Native Module (IOS)

Introduction

from http://facebook.github.io/react-native/docs/native-modules-ios.html

> Sometimes an app needs access to platform API, and React Native doesn't have a corresponding module yet. Maybe you want to reuse some existing Objective-C, Swift or C++ code without having to reimplement it in JavaScript, or write some high performance, multi-threaded code such as for image processing, a database, or any number of advanced extensions.

A Native Module is simply an Objective-C Class that implements the `RCTBridgeModule` protocol.

Example

In your Xcode project create a new file and select **Cocoa Touch Class**, in the creation wizard choose a name for your Class (*e.g. NativeModule*), make it a **Subclass of**: `NSObject` and choose `Objective-C` for the language.

This will create two files `NativeModuleEx.h` and `NativeModuleEx.m`

You will need to import `RCTBridgeModule.h` to your `NativeModuleEx.h` file as it follows:

```
#import <Foundation/Foundation.h>
#import "RCTBridgeModule.h"

@interface NativeModuleEx : NSObject <RCTBridgeModule>

@end
```

In your `NativeModuleEx.m` add the following code:

```
#import "NativeModuleEx.h"

@implementation NativeModuleEx

RCT_EXPORT_MODULE();

RCT_EXPORT_METHOD(testModule:(NSString *)string )
{
  NSLog(@"The string '%@' comes from JavaScript! ", string);
}

@end
```

`RCT_EXPORT_MODULE()` will make your module accessible in JavaScript, you can pass it an optional argument to specify its name. If no name is provided it will match the Objective-C class name.

`RCT_EXPORT_METHOD()` will expose your method to JavaScript, only the methods you export using this macro will be accessible in JavaScript.

Finally, in your JavaScript you can call your method as it follows:

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

var NativeModuleEx = NativeModules.NativeModuleEx;

NativeModuleEx.testModule('Some String !');
```

Chapter 23: Linking Native API

Linking API enables you to both send and receive links between applications. For example, opening the Phone app with number dialed in or opening the Google Maps and starting a navigation to a chosen destination. You can also utilise Linking to make your app able to respond to links opening it from other applications.

To use `Linking` you need to first import it from `react-native`

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

Section 23.1: Outgoing Links

To open a link call openURL.

```
Linking.openURL(url)
.catch(err => console.error('An error occurred ', err))
```

The preferred method is to check if any installed app can handle a given URL beforehand.

```
Linking.canOpenURL(url)
.then(supported => {
  if (!supported) {
    console.log('Unsupported URL: ' + url)
  } else {
    return Linking.openURL(url)
  }
}).catch(err => console.error('An error occurred ', err))
```

URI Schemes

Target App	Example	Reference
Web Browser	`https://stackoverflow.com`	
Phone	`tel:1-408-555-5555`	Apple
Mail	`mailto:email@example.com`	Apple
SMS	`sms:1-408-555-1212`	Apple
Apple Maps	`http://maps.apple.com/?ll=37.484847,-122.148386`	Apple
Google Maps	`geo:37.7749,-122.4194`	Google
iTunes	See iTunes Link Maker	Apple
Facebook	`fb://profile`	Stack Overflow
YouTube	`http://www.youtube.com/v/oHg5SJYRHA0`	Apple
Facetime	`facetime://user@example.com`	Apple
iOS Calendar	`calshow:514300000` [1]	iPhoneDevWiki

[1] Opens the calendar at the stated number of seconds since 1. 1. 2001 (UTC?). For some reason this API is undocumented by Apple.

Section 23.2: Incomming Links

You can detect when your app is launched from an external URL.

```
componentDidMount() {
  const url = Linking.getInitialURL()
  .then((url) => {
    if (url) {
      console.log('Initial url is: ' + url)
    }
  }).catch(err => console.error('An error occurred ', err))
```

```
}
```

To enable this on iOS Link `RCTLinking` to your project.

To enable this on Android, follow these steps.

Chapter 24: ESLint in React Native

This is the topic for ESLint rules explanation for react-native.

Section 24.1: How to start

It's highly recommended to use ESLint in your project on react-native. ESLint is a tool for code validation using specific rules provided by community.

For react-native you can use rulesets for javascript, react and react-native.

Common ESLint rules with motivation and explanations for javascript you can find here: https://github.com/eslint/eslint/tree/master/docs/rules . You can simply add ready ruleset from ESLint developers by adding in your .eslintr.json to 'extends' node 'eslint:recommended'. ("extends": ["eslint:recommended"]) More about ESLint configuring you can read here: http://eslint.org/docs/developer-guide/development-environment . It's recommended to read full doc about this extremely useful tool.

Next, full docs about rules for ES Lint react plugin you can find here: https://github.com/yannickcr/eslint-plugin-react/tree/master/docs/rules . Important note: not all rules from react are relative to react-native. For example: react/display-name and react/no-unknown-property for example. Another rules are 'must have' for every project on react-native, such as react/jsx-no-bind and react/jsx-key.

Be very careful with choosing your own ruleset.

And finaly, there is a plugin explicidly for react-native: https://github.com/intellicode/eslint-plugin-react-native Note: If you split your styles in separate file, rule react-native/no-inline-styles will not work.

For correct working of this tool in react-native env you might need to set value or 'env' in your config to this:

```
"env": {
"browser": true,
"es6": true,
"amd": true
},
```

ESLint is a key tool for development of high quality product.

Chapter 25: Integration with Firebase for Authentication

```
//Replace firebase values with your app API values
import firebase from 'firebase';

componentWillMount() {

    firebase.initializeApp({
        apiKey: "yourAPIKey",
        authDomain: "authDomainNAme",
        databaseURL: "yourDomainBaseURL",
        projectId: "yourProjectID",
        storageBucket: "storageBUcketValue",
        messagingSenderId: "senderIdValue"
    });

    firebase.auth().signInWithEmailAndPassword(email, password)
    .then(this.onLoginSuccess)
})
}
```

Section 25.1: Authentication In React Native Using Firebase

Replace firebase values with your app api values:

```
import firebase from 'firebase';
componentWillMount() {
firebase.initializeApp({
  apiKey: "yourAPIKey",
  authDomain: "authDomainNAme",
  databaseURL: "yourDomainBaseURL",
  projectId: "yourProjectID",
  storageBucket: "storageBUcketValue",
  messagingSenderId: "senderIdValue"
});
    firebase.auth().signInWithEmailAndPassword(email, password)
  .then(this.onLoginSuccess)
  .catch(() => {
    firebase.auth().createUserWithEmailAndPassword(email, password)
      .then(this.onLoginSuccess)
      .catch(this.onLoginFail)
  })
}
```

Section 25.2: React Native - ListView with Firebase

This is what I do when I'm working with Firebase and I want to use ListView.

Use a parent component to retrieve the data from Firebase (Posts.js):

Posts.js

```
import PostsList from './PostsList';

class Posts extends Component{
```

```
    constructor(props) {
        super(props);
        this.state = {
            posts: []
        }
    }

    componentWillMount() {
        firebase.database().ref('Posts/').on('value', function(data) {
            this.setState({ posts: data.val() });
        });
    }

    render() {
        return <PostsList posts={this.state.posts}/>
    }
}
```

PostsList.js

```
class PostsList extends Component {
    constructor(props) {
        super(props);
        this.state = {
            dataSource: new ListView.DataSource({
                rowHasChanged: (row1, row2) => row1 !== row2
            }),
        }
    }

    getDataSource(posts: Array<any>): ListView.DataSource {
        if(!posts) return;
        return this.state.dataSource.cloneWithRows(posts);
    }

    componentDidMount() {
        this.setState({dataSource: this.getDataSource(this.props.posts)});
    }

    componentWillReceiveProps(props) {
        this.setState({dataSource: this.getDataSource(props.posts)});
    }

    renderRow = (post) => {
        return (
            <View>
                <Text>{post.title}</Text>
                <Text>{post.content}</Text>
            </View>
        );
    }

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

```
}
```

I want to point out that in `Posts.js`, I'm not importing `firebase` because you only need to import it once, in the main component of your project (where you have the navigator) and use it anywhere.

This is the solution someone suggested in a question I asked when I was struggling with ListView. I thought it would be nice to share it.

Source: [http://stackoverflow.com/questions/38414289/react-native-listview-not-rendering-data-from-firebase][1]

Chapter 26: Navigator Best Practices

Section 26.1: Navigator

`Navigator` is React Native's default navigator. A `Navigator` component manages a *stack* of route objects, and provides methods for managing that stack.

```
<Navigator
 ref={(navigator) => { this.navigator = navigator }}
  initialRoute={{ id: 'route1', title: 'Route 1' }}
  renderScene={this.renderScene.bind(this)}
  configureScene={(route) => Navigator.SceneConfigs.FloatFromRight}
  style={{ flex: 1 }}
  navigationBar={
    // see "Managing the Navigation Bar" below
    <Navigator.NavigationBar routeMapper={this.routeMapper} />
  }
/>
```

Managing the Route Stack

First of all, notice the `initialRoute` prop. A route is simply a javascript object, and can take whatever shape you want, and have whatever values you want. It's the primary way you'll pass values and methods between components in your navigation stack.

The `Navigator` knows what to render based on the value returned from its `renderScene` prop.

```
renderScene(route, navigator) {
  if (route.id === 'route1') {
    return <ExampleScene navigator={navigator} title={route.title} />; // see below
  } else if (route.id === 'route2') {
    return <ExampleScene navigator={navigator} title={route.title} />; // see below
  }
}
```

Let's imagine an implementation of `ExampleScene` in this example:

```
function ExampleScene(props) {

  function forward() {
    // this route object will passed along to our `renderScene` function we defined above.
    props.navigator.push({ id: 'route2', title: 'Route 2' });
  }

  function back() {
    // `pop` simply pops one route object off the `Navigator`'s stack
    props.navigator.pop();
  }

  return (
    <View>
      <Text>{props.title}</Text>
      <TouchableOpacity onPress={forward}>
        <Text>Go forward!</Text>
      </TouchableOpacity>
      <TouchableOpacity onPress={back}>
        <Text>Go Back!</Text>
      </TouchableOpacity>
```

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

Configuring the Navigator

You can configure the `Navigator`'s transitions with the `configureScene` prop. This is a function that's passed the `route` object, and needs to return a configuration object. These are the available configuration objects:

- Navigator.SceneConfigs.PushFromRight (default)
- Navigator.SceneConfigs.FloatFromRight
- Navigator.SceneConfigs.FloatFromLeft
- Navigator.SceneConfigs.FloatFromBottom
- Navigator.SceneConfigs.FloatFromBottomAndroid
- Navigator.SceneConfigs.FadeAndroid
- Navigator.SceneConfigs.HorizontalSwipeJump
- Navigator.SceneConfigs.HorizontalSwipeJumpFromRight
- Navigator.SceneConfigs.VerticalUpSwipeJump
- Navigator.SceneConfigs.VerticalDownSwipeJump

You can return one of these objects without modification, or you can modify the configuration object to customize the navigation transitions. For example, to modify the edge hit width to more closely emulate the iOS `UINavigationController`'s `interactivePopGestureRecognizer`:

```
configureScene={(route) => {
  return {
    ...Navigator.SceneConfigs.FloatFromRight,
    gestures: {
      pop: {
        ...Navigator.SceneConfigs.FloatFromRight.gestures.pop,
        edgeHitWidth: Dimensions.get('window').width / 2,
      },
    },
  };
}}
```

Managing the NavigationBar

The `Navigator` component comes with a `navigationBar` prop, which can theoretically take any properly configured React component. But the most common implementation uses the default `Navigator.NavigationBar`. This takes a `routeMapper` prop that you can use to configure the appearance of the navigation bar based on the route.

A `routeMapper` is a regular javascript object with three functions: `Title`, `RightButton`, and `LeftButton`. For example:

```
const routeMapper = {

  LeftButton(route, navigator, index, navState) {
    if (index === 0) {
      return null;
    }

    return (
      <TouchableOpacity
        onPress={() => navigator.pop()}
        style={styles.navBarLeftButton}
      >
```

```
        <Text>Back</Text>
      </TouchableOpacity>
    );
  },

  RightButton(route, navigator, index, navState) {
    return (
      <TouchableOpacity
        onPress={route.handleRightButtonClick}
        style={styles.navBarRightButton}
      >
        <Text>Next</Text>
      </TouchableOpacity>
   );
  },

  Title(route, navigator, index, navState) {
    return (
      <Text>
        {route.title}
      </Text>
    );
  },
};
```

See more

For more detailed documentation of each prop, see the the official React Native Documentation for `Navigator`, and the React Native guide on Using Navigators.

Section 26.2: Use react-navigation for navigation in react native apps

With the help of react-navigation, you can add navigation to your app really easy.

Install react-navigation

```
npm install --save react-navigation
```

Example:

```
import { Button, View, Text, AppRegistry } from 'react-native';
import { StackNavigator } from 'react-navigation';

const App = StackNavigator({
  FirstPage: {screen: FirstPage},
  SecondPage: {screen: SecondPage},
});

class FirstPage extends React.Component {
  static navigationOptions = {
    title: 'Welcome',
  };
  render() {
    const { navigate } = this.props.navigation;

    return (
      <Button
        title='Go to Second Page'
        onPress={() =>
```

```
          navigate('SecondPage', { name: 'Awesomepankaj' })
        }
      />
    );
  }
}

class SecondPage extends React.Component {
  static navigationOptions = ({navigation}) => ({
    title: navigation.state.params.name,
  });

  render() {
    const { goBack } = this.props.navigation;
    return (
      <View>
        <Text>Welcome to Second Page</Text>
        <Button
          title="Go back to First Page"
          onPress={() => goBack()}
        />
      </View>
    );
  }
}
```

Section 26.3: react-native Navigation with react-native-router-flux

Install by using `npm install --save react-native-router-flux`

In react-native-router-flux, each route is called a **<Scene>**

> `<Scene key="home" component={LogIn} title="Home" initial />`

`key` A unique string that can be used to refer to the particular scene.

`component` Which component to show, here it's

`title` make a NavBar and give it a title 'Home'

`initial` Is this the first screen of the App

> Example:

```
import React from 'react';
import { Scene, Router } from 'react-native-router-flux';
import LogIn from './components/LogIn';
import SecondPage from './components/SecondPage';

const RouterComponent = () => {
  return (
    <Router>
      <Scene key="login" component={LogIn} title="Login Form" initial />
      <Scene key="secondPage" component={SecondPage} title="Home" />
    </Router>
```

```
  );
};

export default RouterComponent;
```

Import this file in the main App.js(index file) and render it. For more information can visit this link.

Chapter 27: Navigator with buttons injected from pages

Section 27.1: Introduction

Instead of bloating your main js file that contains your navigator with buttons. It's cleaner to just inject buttons on-demand in any page that you need.

```
//In the page "Home", I want to have the right nav button to show
//a settings modal that resides in "Home" component.

componentWillMount() {
  this.props.route.navbarTitle = "Home";

  this.props.route.rightNavButton = {
    text: "Settings",
    onPress: this._ShowSettingsModal.bind(this)
  };
}
```

Section 27.2: Full commented example

```
'use strict';

import React, {Component} from 'react';
import ReactNative from 'react-native';

const {
  AppRegistry,
  StyleSheet,
  Text,
  View,
  Navigator,
  Alert,
  TouchableHighlight
} = ReactNative;


//This is the app container that contains the navigator stuff
class AppContainer extends Component {

    renderScene(route, navigator) {
        switch(route.name) {
            case "Home":
      //You must pass route as a prop for this trick to work properly
            return <Home route={route} navigator={navigator} {...route.passProps}  />
            default:
            return (
        <Text route={route}
        style={styles.container}>
            Your route name is probably incorrect {JSON.stringify(route)}
            </Text>
      );
        }
    }

  render() {
    return (
```

```
      <Navigator
        navigationBar={
          <Navigator.NavigationBar
            style={ styles.navbar }
            routeMapper={ NavigationBarRouteMapper } />
        }

        initialRoute={{ name: 'Home' }}
        renderScene={ this.renderScene }

      />
    );
  }
}


//Nothing fancy here, except for checking for injected buttons.
//Notice how we are checking if there are injected buttons inside the route object.
//Also, we are showing a "Back" button when the page is not at index-0 (e.g. not home)
var NavigationBarRouteMapper = {
  LeftButton(route, navigator, index, navState) {
    if(route.leftNavButton) {
      return (
        <TouchableHighlight
        style={styles.leftNavButton}
        underlayColor="transparent"
        onPress={route.leftNavButton.onPress}>
          <Text style={styles.navbarButtonText}>{route.leftNavButton.text}</Text>
        </TouchableHighlight>
      );
    }
    else if(route.enableBackButton) {
      return (
        <TouchableHighlight
        style={styles.leftNavButton}
        underlayColor="transparent"
        onPress={() => navigator.pop() }>
          <Text style={styles.navbarButtonText}>Back</Text>
        </TouchableHighlight>
      );
    }
  },
  RightButton(route, navigator, index, navState) {
    if(route.rightNavButton) {
      return (
        <TouchableHighlight
        style={styles.rightNavButton}
        underlayColor="transparent"
        onPress={route.rightNavButton.onPress}>
          <Text style={styles.navbarButtonText}>{route.rightNavButton.text}</Text>
        </TouchableHighlight>
      );
    }
  },
  Title(route, navigator, index, navState) {
    //You can inject the title aswell.  If you don't we'll use the route name.
    return (<Text style={styles.navbarTitle}>{route.navbarTitle || route.name}</Text>);
  }
};

//This is considered a sub-page that navigator is showing
class Home extends Component {
```

```
  //This trick depends on that componentWillMount fires before the navbar is created
  componentWillMount() {
        this.props.route.navbarTitle = "Home";

        this.props.route.rightNavButton = {
            text: "Button",
            onPress: this._doSomething.bind(this)
        };
    }

  //This method will be invoked by pressing the injected button.
  _doSomething() {
      Alert.alert(
      'Awesome, eh?',
      null,
      [
        {text: 'Indeed'},
      ]
    )
  }

  render() {
    return (
      <View style={styles.container}>
            <Text>You are home</Text>
      </View>
    );
  }
}

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
    marginTop: 66
  },
  navbar: {
    backgroundColor: '#ffffff',
  },
  navbarTitle: {
    marginVertical: 10,
    fontSize: 17
  },
  leftNavButton: {
    marginVertical: 10,
    paddingLeft: 8,
 },
  rightNavButton: {
    marginVertical: 10,
    paddingRight: 8,
  },
  navbarButtonText: {
    fontSize: 17,
    color: "#007AFF"
  }
});

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

Chapter 28: Create a shareable APK for android

Steps to create an APK (signed and unsigned) which you can install on a device using CLI and share as well:

Problem statement: I've built my app, I can run it on my local emulator (and also on my android device by changing debug server). But, I want to build an apk that I can send to someone without access to development server and I want them to be able to test application.

Section 28.1: Create a key to sign the APK

```
keytool -genkey -v -keystore my-app-key.keystore -alias my-app-alias -keyalg RSA -keysize 2048 -
validity 10000
```

Use a password when prompted

Section 28.2: Once the key is generated, use it to generate the installable build:

```
react-native bundle --platform android --dev false --entry-file index.android.js \
--bundle-output android/app/src/main/assets/index.android.bundle \
--assets-dest android/app/src/main/res/
```

Section 28.3: Generate the build using gradle

```
cd android && ./gradlew assembleRelease
```

Section 28.4: Upload or share the generated APK

Upload the APK to your phone. The -r flag will replace the existing app (if it exists)

```
adb install -r ./app/build/outputs/apk/app-release-unsigned.apk
```

The shareable signed APK is located at:

```
./app/build/outputs/apk/app-release.apk
```

Chapter 29: PushNotification

We can add Push Notification to react native app by using the npm module **react-native-push-notification** by **zo0r**. This enables for a cross platform development.

Installation

npm install --save react-native-push-notification

react-native link

Section 29.1: Push Notification Simple Setup

Create new project PushNotification

```
react-native init PushNotification
```

Put following in index.android.js

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

import {
  AppRegistry,
  StyleSheet,
  Text,
  View,
  Button
} from 'react-native';

import PushNotification from 'react-native-push-notification';

export default class App extends Component {

    constructor(props){
        super(props);

        this.NewNotification = this.NewNotification.bind(this);
      }

    componentDidMount(){

        PushNotification.configure({

            // (required) Called when a remote or local notification is opened or received
            onNotification: function(notification) {
                console.log( 'NOTIFICATION:', notification );
            },

            // Should the initial notification be popped automatically
            // default: true
            popInitialNotification: true,

            /**
              * (optional) default: true
              * - Specified if permissions (ios) and token (android and ios) will requested or not,
              * - if not, you must call PushNotificationsHandler.requestPermissions() later
              */
            requestPermissions: true,
```

```
        });

    }

      NewNotification(){

          let date = new Date(Date.now() + (this.state.seconds * 1000));

          //Fix for IOS
        if(Platform.OS == "ios"){
            date = date.toISOString();
        }

        PushNotification.localNotificationSchedule({
            message: "My Notification Message", // (required)
            date: date,// (optional) for setting delay
            largeIcon:""// set this blank for removing large icon
            //smallIcon: "ic_notification", // (optional) default: "ic_notification" with fallback
for "ic_launcher"
        });
    }

      render() {

        return (
            <View style={styles.container}>
                <Text style={styles.welcome}>
                  Push Notification
                </Text>
                <View style={styles.Button} >
                <Button
                  onPress={()=>{this.NewNotification()}}
                  title="Show Notification"
                  style={styles.Button}
                  color="#841584"
                  accessibilityLabel="Show Notification"
                />
                </View>
            </View>
        );
      }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  Button:{
    margin: 10,
  }
});

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

Section 29.2: Navigating to scene from Notification

Here's a simple example to demonstrate that how can we jump/open a specific screen based on the notification. For example, when a user clicks on the notification, the app should open and directly jump to notifications page instead of home page.

```
'use strict';

import React, { Component } from 'react';
import {
    StyleSheet,
    Text,
    View,
    Navigator,
    TouchableOpacity,
    AsyncStorage,
    BackAndroid,
    Platform,
} from 'react-native';
import PushNotification from 'react-native-push-notification';

let initialRoute = { id: 'loginview' }

export default class MainClass extends Component
{
    constructor(props)
    {
        super(props);

        this.handleNotification = this.handleNotification.bind(this);
    }

    handleNotification(notification)
    {
        console.log('handleNotification');
        var notificationId = ''
        //your logic to get relevant information from the notification

    //here you navigate to a scene in your app based on the notification info
        this.navigator.push({ id: Constants.ITEM_VIEW_ID, item: item });
    }

    componentDidMount()
    {
        var that = this;

        PushNotification.configure({

            // (optional) Called when Token is generated (iOS and Android)
            onRegister: function(token) {
                console.log( 'TOKEN:', token );
            },

            // (required) Called when a remote or local notification is opened or received
            onNotification(notification) {
                console.log('onNotification')
                console.log( notification );

                that.handleNotification(notification);
            },
```

```
            // ANDROID ONLY: (optional) GCM Sender ID.
            senderID: "Vizido",

            // IOS ONLY (optional): default: all - Permissions to register.
            permissions: {
                alert: true,
                badge: true,
                sound: true
            },

            // Should the initial notification be popped automatically
            // default: true
            popInitialNotification: true,

            /**
              * (optional) default: true
              * - Specified if permissions (ios) and token (android and ios) will requested or not,
              * - if not, you must call PushNotificationsHandler.requestPermissions() later
              */
            requestPermissions: true,
        });
    }

    render()
    {

        return (
            <Navigator
                ref={(nav) => this.navigator = nav }
                initialRoute={initialRoute}
                renderScene={this.renderScene.bind(this)}
                configureScene={(route) =>
                    {
                        if (route.sceneConfig)
                        {
                            return route.sceneConfig;
                        }
                        return Navigator.SceneConfigs.FadeAndroid;
                    }
                }
            />
        );
    }

    renderScene(route, navigator)
    {

        switch (route.id)
        {
            // do your routing here
            case 'mainview':
                return ( <MainView navigator={navigator} /> );

            default:
                return ( <MainView navigator={navigator} /> );
        }
    }
}
```

Chapter 30: Render Best Practises

Topic for important notes about specific Component.render method behavoir.

Section 30.1: Functions in JSX

For better performance it's important to avoid using of array (lambda) function in JSX.

As explained at https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/jsx-no-bind.md :

> A bind call or arrow function in a JSX prop will create a brand new function on every single render. This is bad for performance, as it will result in the garbage collector being invoked way more than is necessary. It may also cause unnecessary re-renders if a brand new function is passed as a prop to a component that uses reference equality check on the prop to determine if it should update.

So if have jsx code block like this:

```
<TextInput
 onChangeValue={  value => this.handleValueChanging(value) }
/>
```

or

```
<button onClick={ this.handleClick.bind(this) }></button>
```

you can make it better:

```
<TextInput
  onChangeValue={  this.handleValueChanging }
/>
```

and

```
<button onClick={ this.handleClick }></button>
```

For correct context within handleValueChanging function you can apply it in constructor of component:

```
constructor(){
  this.handleValueChanging = this.handleValueChanging.bind(this)
}
```

more in binding a function passed to a component

Or you can use solutions like this: https://github.com/andreypopp/autobind-decorator and simply add @autobind decorator to each methos that you want bind to:

```
@autobind
handleValueChanging(newValue)
{
    //processing event
}
```

Chapter 31: Debugging

Section 31.1: Start Remote JS Debugging in Android

You can start the remote debugging from Developer menu. After selecting the enable remote debugging it will open Google Chrome, So that you can log the output into your console. You can also write debugger syntax into your js code.

Section 31.2: Using console.log()

You can print log message in the terminal using `console.log()`. To do so, open a new terminal and run following command for Android:

```
react-native log-android
```

or following command if you are using iOS:

```
react-native log-ios
```

You will now start to see all the log message in this terminal

Chapter 32: Unit Testing

Unit testing is a low level testing practice where smallest units or components of the code are tested.

Section 32.1: Unit Test In React Native Using Jest

Starting from react-native version 0.38, a Jest setup is included by default when running react-native init. The following configuration should be automatically added to your package.json file:

```
    "scripts": {
    "start": "node node_modules/react-native/local-cli/cli.js start",
    "test": "jest"
    },
    "jest": {
     "preset": "react-native"
    }
```

You can run `run npm test or jest` to test in react native. For code example: Link

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Abdulaziz Alkharashi	Chapters 18 and 12
Aditya Singh	Chapter 28
Ahmed Al Haddad	Chapter 27
Ahmed Ali	Chapter 5
Alex Belets	Chapters 9, 15, 30 and 24
Alireza Valizade	Chapter 15
Andres C. Viesca	Chapter 22
Ankit Sinha	Chapters 26, 25 and 32
AntonB	Chapter 15
Cássio Santos	Chapter 20
CallMeNorm	Chapter 3
Chris Pena	Chapters 3 and 15
corasan	Chapter 25
Daniel Schmidt	Chapter 15
David	Chapter 6
Dmitry Petukhov	Chapters 1, 14 and 15
Dr. Nitpick	Chapter 1
epsilondelta	Chapter 14
fson	Chapter 3
Gabriel Diez	Chapter 16
Idan	Chapters 3 and 14
Jagadish Upadhyay	Chapters 9, 3, 14, 15, 16, 6, 17 and 31
Jigar Shah	Chapters 17, 8 and 4
Kaleb Portillo	Chapters 1 and 11
Liron Yahdav	Chapter 5
Lucas Oliveira	Chapter 1
Lwin Kyaw Myat	Chapters 18 and 21
manosim	Chapters 1, 14 and 20
Mayeul	Chapter 21
Michael Hancock	Chapter 10
Michael Helvey	Chapter 26
Michael S	Chapter 20
mostafiz rahman	Chapter 31
Mozak	Chapter 14
Noitidart	Chapter 18
Pankaj Thakur	Chapter 26
Pascal Le Merrer	Chapter 20
respectTheCode	Chapter 15
Scimonster	Chapters 1 and 8
Serdar Değirmenci	Chapter 17
shaN	Chapters 15 and 29
Sriraman	Chapters 14, 18, 19, 20 and 10
stereodenis	Chapter 2
sudo bangbang	Chapters 9, 7, 32 and 13
Tejashwi Kalp Taru	Chapters 15 and 29
Tim Rijavec	Chapters 14 and 6
Tushar Khatiwada	Chapter 1
Viktor Seč	Chapter 23
Virat18	Chapter 20

xhg	Chapter 1
Yevhen Dubinin	Chapters 1 and 3
Zakaria Ridouh	Chapters 16, 2 and 17
zhenjie ruan	Chapter 3

You may also like