**React Native: Basics**

React Native is a JavaScript framework for building native mobile apps. It uses the React framework and offers large amount of inbuilt components and APIs.

React Native Features

* **React** − This is a Framework for building web and mobile apps using JavaScript.
* **Native** − You can use native components controlled by JavaScript.
* **Platforms** − React Native supports IOS and Android platform.

React Native Advantages

* **JavaScript** − You can use the existing JavaScript knowledge to build native mobile apps.
* **Code sharing** − You can share most of your code on different platforms.
* **Community** − The community around React and React Native is large, and you will be able to find any answer you need.

React Native Limitations

* **Native Components** − If you want to create native functionality which is not created yet, you will need to write some platform specific code.

# **React Native - Environment Setup**

There are a couple of things you need to install to set up the environment for React Native. We will use OSX as our building platform.

|  |  |  |
| --- | --- | --- |
| **S.NO.** | **Software** | **Description** |
| 1 | NodeJS and NPM | You can follow our **NodeJS Environment Setup**tutorial to install NodeJS. |

## Step 1 - Install Homebrew

Open your terminal and run the following code to install Homebrew −

/usr/bin/ruby -e "$(curl -fsSL

https://raw.githubusercontent.com/Homebrew/install/master/install)"

## Step 2 - Install Watchman

Run the following code to install Watchman.

brew install watchman

## Step 3 - Install React Native

Now, run the following code to install React Native.

npm install -g react-native-cli

## Step 4 Android - Install Android Studio

You can install Android studio by following this link.

https://developer.android.com/studio/install

## Step 4 IOS - Install XCode

For IOS development you will need XCode.

https://itunes.apple.com/us/app/xcode/id497799835?mt=12

## Step 5 - Create First App

We will initialize our first app by running the code given below in the terminal from the folder where we want to create the app (in our case Desktop).

react-native init reactTutorialApp

## Step 6 - Run React Native Packager

First, we need to open the app folder in terminal.

cd reactTutorialApp

Now, we can run the packager.

react-native start

You should keep this terminal window running while developing your app.

## Step 7 - Run the App on IOS simulator

This step will open your app in the IOS simulator. Run the following command in another terminal.

react-native run-ios

# **Native - State**

## Difference between State and Props

The **state** is mutable while **props** are immutable. This means that **state** can be updated in the future while props cannot be updated.

### **Using State**

This is our root component. We are just importing **Home** which will be used in most of the chapters.

**NOTE** − This file won't change during the course of this tutorial, so we will leave it out in the future.

**index.ios.js**

import React, { Component } from 'react';

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

import Home from './src/components/home/Home.js'

class reactTutorialApp extends Component {

render() {

return (

<View>

<Home />

</View>

);

}

}

export default reactTutorialApp

AppRegistry.registerComponent('reactTutorialApp', () ⇒ reactTutorialApp);

Initial state is defined inside the **Home** class by using the **state = {}** syntax.

We will bind **myText** in a view using the **{this.state.myText}** syntax.

**src/components/home/Home.js**

import React, { Component } from 'react';

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

class Home extends Component {

state = {

myState: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit,

sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris

nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in

reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui

officia deserunt mollit anim id est laborum.'

}

render() {

return (

<View>

<Text>

{this.state.myState}

</Text>

</View>

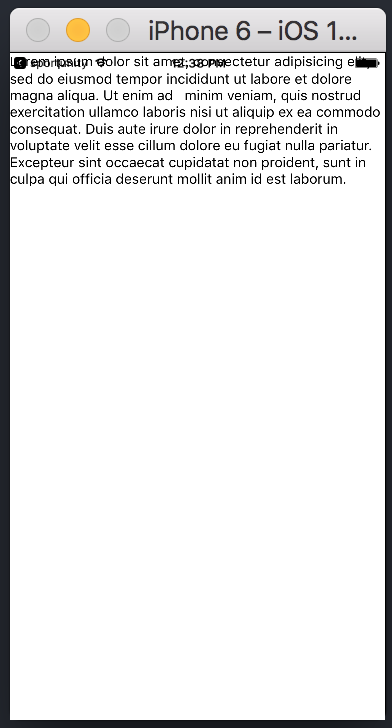
);

}

}

export default Home;

We can see in emulator text from the state as in the following screenshot.



### **Updating State**

Since state is mutable, we can update it by creating the **deleteState** function and call it using the **onPress = {this.deleteText}** event.

**src/components/home/Home.js**

import React, { Component } from 'react'

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

class Home extends Component {

state = {

myState: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed

do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi

ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit

in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia

deserunt mollit anim id est laborum.'

}

updateState = () ⇒ this.setState({ myState: 'The state is updated' })

render() {

return (

<View>

<Text onPress = {this.updateState}>

{this.state.myState}

</Text>

</View>

);

}

}

export default Home;

**NOTES** − In all chapters, we will use the class syntax for stateful (container) components and function syntax for stateless (presentational) components. We will learn more about components in the next chapter.

In our second example, we are using the arrow function syntax for **updateState**. You should keep in mind that this syntax uses the lexical scope, and **this** keyword will be bound to the environment object (Class). This will sometimes lead to unexpected behavior.

The other way to define methods is to use the EC5 functions but in that case we will need to bind **this** manually in the constructor. Consider the following example to understand this.

class Home extends Component {

constructor(){

super()

this.updateState = this.updateState.bind(this)

}

updateState(){

//

}

render(){

//

}

}

# **Native - Props**

Presentational components should get all data by passing **props**. Only container components should have **state**.

## Container Component

We will now understand what a container component is and also how it works.

### **Theory**

Now we will update our container component. This component will handle the state and pass the props to the presentational component.

Container component is only used for handling state. All functionality related to view(styling etc.) will be handled in the presentational component.

### **Example**

If we want to use example from the last chapter we need to remove the **Text**element from the render function since this element is used for presenting text to the users. This should be inside the presentational component.

Let us review the code in the example given below. We will import the **PresentationalComponent** and pass it to the render function.

After we import the **PresentationalComponent** and pass it to the render function, we need to pass the props. We will pass the props by adding **myText = {this.state.myText}** and **deleteText = {this.deleteText}** to **<PresentationalComponent>**. Now, we will be able to access this inside the presentational component.

**src/components/home/Home.js**

import React, { Component } from 'react'

import { View } from 'react-native'

import PresentationalComponent from './PresentationalComponent'

class Home extends Component {

state = {

myState: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit,

sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi

ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in

voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur

sint occaecat cupidatat non proident, sunt in culpa qui officia

deserunt mollit anim id est laborum.'

}

updateState = () ⇒ {

this.setState({ myState: 'The state is updated' })

}

render() {

return (

<View>

<PresentationalComponent myState = {this.state.myState} updateState =

{this.updateState}/>

</View>

)

}

}

export default Home

## Presentational Component

We will now understand what a presentational component is and also how it works.

### **Theory**

Presentational components should be used only for presenting view to the users. These components do not have state. They receive all data and functions as props.

The best practice is to use as much presentational components as possible.

### **Example**

As we mentioned in our previous chapter, we are using the EC6 function syntax for presentational components.

Our component will receive props, return view elements, present text using **{props.myText}** and call the **{props.deleteText}** function when a user clicks on the text.

**src/components/home/PresentationalComponent.js**

import React, { Component } from 'react'

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

const PresentationalComponent = (props) ⇒ {

return (

<View>

<Text onPress = {props.updateState}>

{props.myState}

</Text>

</View>

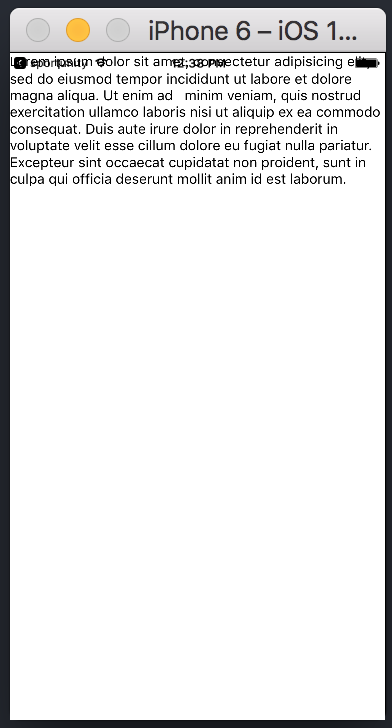
)

}

export default PresentationalComponent

Now, we have the same functionality as in our **State** chapter. The only difference is that we refactored our code to the container and the presentational component.

You can run the app and see the text as in the following screenshot.



If you click on text, it will be removed from the screen.

# **Native - Styling**

You can use the **style** property to add the styles inline. However, this is not the best practice because it can be hard to read the code.

In this chapter, we will use the **Stylesheet** for styling.

Container Component

In this section, we will simplify our container component from our previous chapter.

**src/components/home/Home.js**

import React, { Component } from 'react'

import { View } from 'react-native'

import PresentationalComponent from './PresentationalComponent'

class Home extends Component {

state = {

myState: 'This is my state'

}

render() {

return (

<View>

<PresentationalComponent myState = {this.state.myState}/>

</View>

)

}

}

export default Home

Presentational Component

In the following example, we will import the **StyleSheet**. At the bottom of the file, we will create our stylesheet and assign it to the **styles** constant. Note that our styles are in **camelCase** and we do not use **px** or % for styling.

To apply styles to our text, we need to add **style = {styles.myText}**property to the **Text** element.

**src/components/home/PresentationalComponent.js**

import React, { Component } from 'react'

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

const PresentationalComponent = (props) ⇒ {

return (

<View>

<Text style = {styles.myState}>

{props.myState}

</Text>

</View>

)

}

export default PresentationalComponent

const styles = StyleSheet.create ({

myState: {

marginTop: 20,

textAlign: 'center',

color: 'blue',

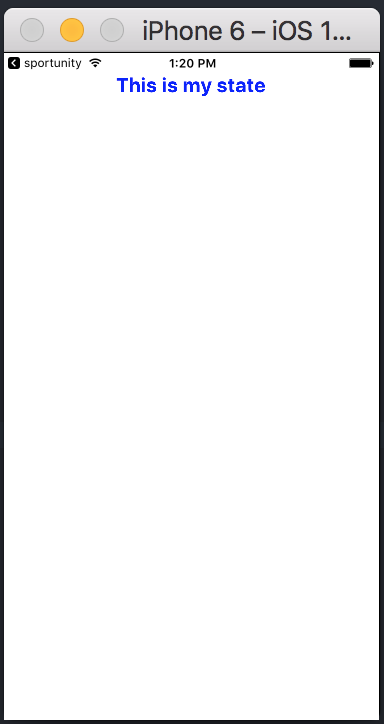
fontWeight: 'bold',

fontSize: 20

}

})

When we run the app, we will receive the following output.



# **Native - Flexbox**

Layout

To achieve the desired layout, flexbox offers three main properties − **flexDirection justifyContent** and **alignItems**.

The following table shows the possible options.

|  |  |  |
| --- | --- | --- |
| **Property** | **Values** | **Description** |
| flexDirection | 'column', 'row' | Used to specify if elements will be aligned vertically or horizontally. |
| justifyContent | 'center', 'flex-start', 'flex-end', 'space-around', 'space-between' | Used to determine how should elements be distributed inside the container. |
| alignItems | 'center', 'flex-start', 'flex-end', 'stretched' | Used to determine how should elements be distributed inside the container along the secondary axis (opposite of flexDirection) |

If you want to align the items vertically and centralize them, then you can use the following code.

**src/components/home/Home.js**

import React, { Component } from 'react'

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

const Home = (props) ⇒ {

return (

<View style = {styles.container}>

<View style = {styles.redbox} />

<View style = {styles.bluebox} />

<View style = {styles.blackbox} />

</View>

)

}

export default Home

const styles = StyleSheet.create ({

container: {

flexDirection: 'column',

justifyContent: 'center',

alignItems: 'center',

backgroundColor: 'grey',

height: 600

},

redbox: {

width: 100,

height: 100,

backgroundColor: 'red'

},

bluebox: {

width: 100,

height: 100,

backgroundColor: 'blue'

},

blackbox: {

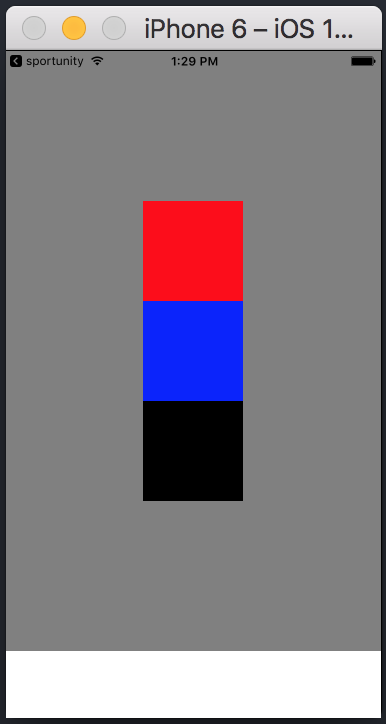
width: 100,

height: 100,

backgroundColor: 'black'

},

})



If the items need to be moved to the right side and spaces need to be added between them, then we can use the following code.

**src/components/home/Home.js**

import React, { Component } from 'react'

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

const Home = (props) ⇒ {

return (

<View style = {styles.container}>

<View style = {styles.redbox} />

<View style = {styles.bluebox} />

<View style = {styles.blackbox} />

</View>

)

}

export default Home

const styles = StyleSheet.create ({

container: {

flexDirection: 'column',

justifyContent: 'space-between',

alignItems: 'flex-end',

backgroundColor: 'grey',

height: 600

},

redbox: {

width: 100,

height: 100,

backgroundColor: 'red'

},

bluebox: {

width: 100,

height: 100,

backgroundColor: 'blue'

},

blackbox: {

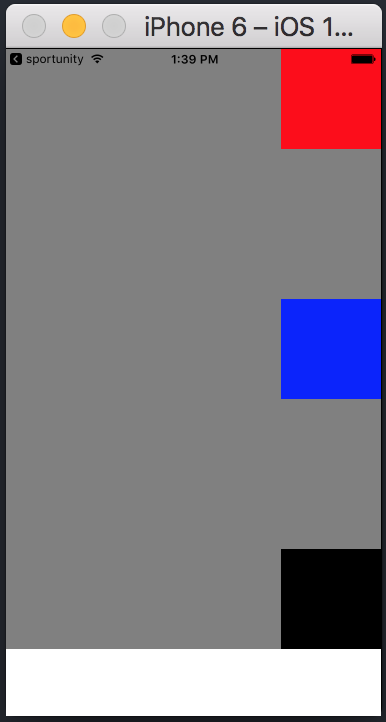
width: 100,

height: 100,

backgroundColor: 'black'

},

})



The following example shows how you can style black and yellow boxes using flexbox. Each container uses different properties.

**src/components/home/Home.js**

import React, { Component } from 'react'

import { View, Image, StyleSheet } from 'react-native'

const Home = () ⇒ {

return (

<View>

<View style = {style.container1}>

<View style = {style.blackImg}/>

<View style = {style.yellowImg}/>

</View>

<View style = {style.container2}>

<View style = {style.blackImg}/>

<View style = {style.yellowImg}/>

</View>

<View style = {style.container3}>

<View style = {style.blackImg}/>

<View style = {style.yellowImg}/>

</View>

<View style = {style.container4}>

<View style = {style.blackImg}/>

<View style = {style.yellowImg}/>

</View>

<View style = {style.container5}>

<View style = {style.blackImg}/>

<View style = {style.yellowImg}/>

</View>

)

}

export default Home

const style = StyleSheet.create ({

container1: {

borderBottomWidth: 1,

borderBottomColor: '#f4c842'

},

container2: {

flexDirection: 'row',

justifyContent: 'space-around',

alignItems: 'center',

borderBottomWidth: 1,

borderBottomColor: '#f4c842'

},

container3: {

flexDirection: 'row',

justifyContent: 'flex-end',

borderBottomWidth: 1,

borderBottomColor: '#f4c842'

},

container4: {

alignItems: 'center',

borderBottomWidth: 1,

borderBottomColor: '#f4c842'

},

container5: {

flexDirection: 'row',

justifyContent: 'center',

alignItems: 'center',

},

blackImg: {

backgroundColor: 'black',

height: 90,

width: 90

},

yellowImg: {

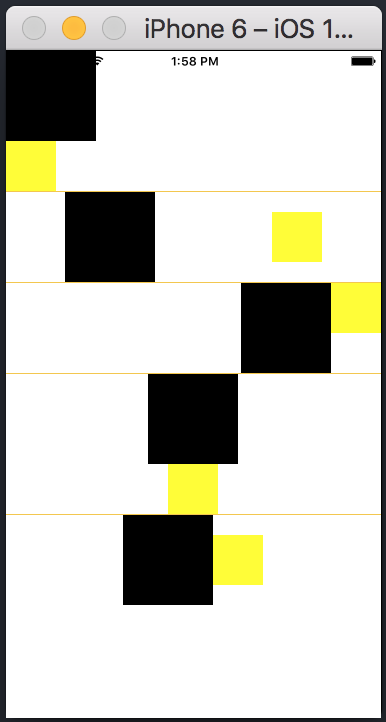
backgroundColor: 'yellow',

height: 50,

width: 50

}

})



# **Native - ListView**

**src/components/home/Home.js**

import React from 'react'

import List from './List.js'

const Home = () => {

return (

<List />

)

}

export default Home

To create a list, we will use the **map()** method. This will iterate over an array of items, and render each one.

**src/components/home/List.js**

import React, { Component } from 'react'

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

class List extends Component {

state = {

names: [

{

id: 0,

name: 'Ben',

},

{

id: 1,

name: 'Susan',

},

{

id: 2,

name: 'Robert',

},

{

id: 3,

name: 'Mary',

}

]

}

alertItemName = (item) => {

alert(item.name)

}

render() {

return (

<View>

{

this.state.names.map((item, index) => (

<TouchableOpacity

key = {item.id}

style = {styles.container}

onPress = {() => this.alertItemName(item)}>

<Text style = {styles.text}>

{item.name}

</Text>

</TouchableOpacity>

))

}

</View>

)

}

}

export default List

const styles = StyleSheet.create ({

container: {

padding: 10,

marginTop: 3,

backgroundColor: '#d9f9b1',

alignItems: 'center',

},

text: {

color: '#4f603c'

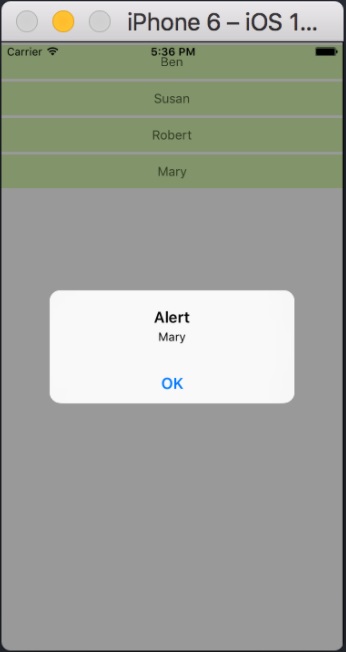
}

})

When we run the app, we will see the list of names.



You can click on each item in the list to trigger an alert with the name.



# **Native - Text Input**

The Home component will import and render inputs.

**src/components/home/Home.js**

import React from 'react';

import Inputs from './Inputs.js'

const Home = () ⇒ {

return (

<Inputs />

)

}

export default Home

Inputs

We will define the initial state. After defining the initial state, we will create the **handleEmail** and the **handlePassword** functions. These functions are used for updating state.

The **login()** function will just alert the current value of the state.

We will also add some other properties to text inputs to disable auto capitalisation, remove the bottom border on Android devices and set a placeholder.

**src/components/home/Inputs.js**

import React, { Component } from 'react'

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

class Inputs extends Component {

state = {

email: '',

password: ''

}

handleEmail = (text) ⇒ {

this.setState({ email: text })

}

handlePassword = (text) ⇒ {

this.setState({ password: text })

}

login = (email, pass) ⇒ {

alert('email: ' + email + ' password: ' + pass)

}

render(){

return (

<View style = {styles.container}>

<TextInput style = {styles.input}

underlineColorAndroid = "transparent"

placeholder = "Email"

placeholderTextColor = "#9a73ef"

autoCapitalize = "none"

onChangeText = {this.handleEmail}/>

<TextInput style = {styles.input}

underlineColorAndroid = "transparent"

placeholder = "Password"

placeholderTextColor = "#9a73ef"

autoCapitalize = "none"

onChangeText = {this.handlePassword}/>

<TouchableOpacity

style = {styles.submitButton}

onPress = {

() ⇒ this.login(this.state.email, this.state.password)

}>

<Text style = {styles.submitButtonText}> Submit </Text>

</TouchableOpacity>

</View>

)

}

}

export default Inputs

const styles = StyleSheet.create({

container: {

paddingTop: 23

},

input: {

margin: 15,

height: 40,

borderColor: '#7a42f4',

borderWidth: 1

},

submitButton: {

backgroundColor: '#7a42f4',

padding: 10,

margin: 15,

height: 40,

},

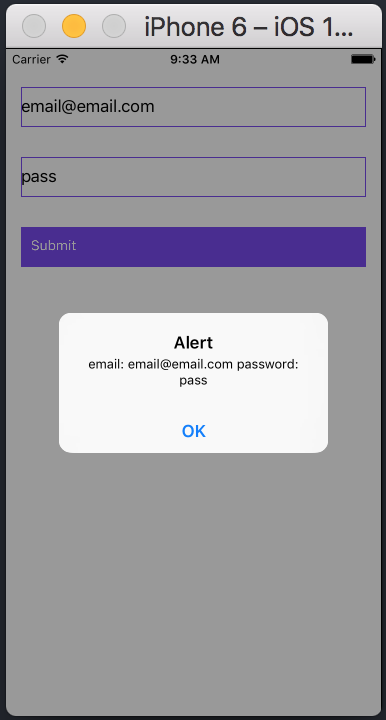
submitButtonText:{

color: 'white'

}

})

Whenever we type in one of the input fields, the state will be updated. When we click on the **Submit** button, text from inputs will be shown inside the dialog box.



# **Native - ScrollView**

We will again create **ScrollViewExample.js** and import it in **Home**.

**src/components/home/Home.js**

import React from 'react'

import ScrollViewExample from './ScrollViewExample.js'

const Home = () ⇒ {

return (

<ScrollViewExample />

)

}

export default Home

Scrollview will render a list of names. We will create it in state.

**src/components/home/ScrollViewExample.js**

import React, { Component } from 'react';

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

class ScrollViewExample extends Component {

state = {

names: [

{'name': 'Ben', 'id': 1},

{'name': 'Susan', 'id': 2},

{'name': 'Robert', 'id': 3},

{'name': 'Mary', 'id': 4},

{'name': 'Daniel', 'id': 5},

{'name': 'Laura', 'id': 6},

{'name': 'John', 'id': 7},

{'name': 'Debra', 'id': 8},

{'name': 'Aron', 'id': 9},

{'name': 'Ann', 'id': 10},

{'name': 'Steve', 'id': 11},

{'name': 'Olivia', 'id': 12}

]

}

render() {

return (

<View>

<ScrollView>

{

this.state.names.map((item, index) ⇒ (

<View key = {item.id} style = {styles.item}>

<Text>{item.name}</Text>

</View>

))

}

</ScrollView>

</View>

)

}

}

export default ScrollViewExample

const styles = StyleSheet.create ({

item: {

flexDirection: 'row',

justifyContent: 'space-between',

alignItems: 'center',

padding: 30,

margin: 2,

borderColor: '#2a4944',

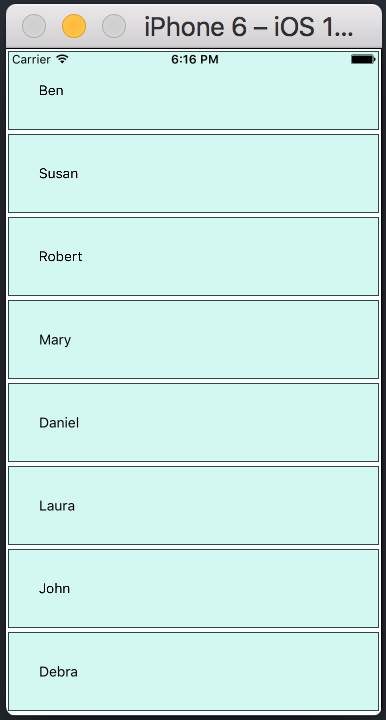
borderWidth: 1,

backgroundColor: '#d2f7f1'

}

})

When we run the app, we will see the scrollable list of names.



# **Native - Images**

Adding Image

Let us create a new folder **img** inside the **src** folder. We will add our image (**myImage.png**) inside this folder.

We will show images on the home screen.

**src/components/home/Home.js**

import React from 'react';

import ImagesExample from './ImagesExample.js'

const Home = () ⇒ {

return (

<ImagesExample />

)

}

export default Home

Local image can be accessed using the following syntax.

**src/components/home/ImagesExample.js**

import React, { Component } from 'react'

import { Image } from 'react-native'

const Home = () ⇒ (

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

)

export default Home

Screen Density

React Native offers a way to optimize images for different devices using **@2x, @3x** suffix. The app will load only the image necessary for particular screen density.

The following will be the names of the image inside the **img** folder.

my-image@2x.jpg

my-image@3x.jpg

Network Images

When using network images, instead of **require**, we need the **source**property. It is recommended to define the **width** and the **height** for network images.

**src/components/home/Home.js**

import React, { Component } from 'react'

import { View, Image } from 'react-native'

const Home = () ⇒ (

<Image

source = {{ uri:

'https://pbs.twimg.com/profile\_images/486929358120964097/gNLINY67\_400x400.png' }}

style = {{ width: 200, height: 200 }}

/>

)

export default Home

It will show the following output −

# **Native - HTTP**

**src/components/home/Home.js**

import React from 'react';

import HttpExample from './HttpExample.js'

const Home = () ⇒ {

return (

<HttpExample />

)

}

export default Home

Using Fetch

We will use the **componentDidMount** lifecycle method to load the data from server as soon as the component is mounted. This function will send GET request to the server, return JSON data, log output to console and update our state.

**src/components/home/HttpExample.js**

import React, { Component } from 'react'

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

class HttpExample extends Component {

state = {

data: ''

}

componentDidMount = () ⇒ {

fetch('https://jsonplaceholder.typicode.com/posts/1', {

method: 'GET'

})

.then((response) ⇒ response.json())

.then((responseJson) ⇒ {

console.log(responseJson);

this.setState({

data: responseJson

})

})

.catch((error) ⇒ {

console.error(error);

});

}

render() {

return (

<View>

<Text>

{this.state.data.body}

</Text>

</View>

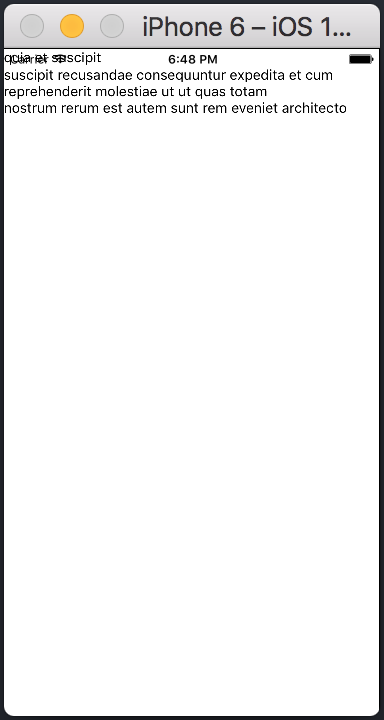
)

}

}

export default HttpExample

If the request is successful, the data will be displayed on screen.



We can also check the console to see the response.



# **Native - Buttons**

Facebook offers the **Button** component, which can be used as a generic button. Consider the following example to understand the same.

**src/components/home/Home.js**

import React, { Component } from 'react'

import { Button } from 'react-native'

const Home = () ⇒ {

const handlePress = () ⇒ false

return (

<Button

onPress = {handlePress}

title = "Red button!"

color = "red"

/>

)

}

export default Home

If the default **Button** component does not suit your needs, you can use one of the following components instead.

## Touchable Opacity

This element will change the opacity of an element when touched.

**src/components/home/Home.js**

import React from 'react'

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

const Home = () ⇒ {

return (

<View style = {styles.container}>

<TouchableOpacity>

<Text style = {styles.text}>

Button

</Text>

</TouchableOpacity>

</View>

)

}

export default Home

const styles = StyleSheet.create ({

container: {

alignItems: 'center',

},

text: {

borderWidth: 1,

padding: 25,

borderColor: 'black',

backgroundColor: 'red'

}

})

## Touchable Highlight

When a user presses the element, it will get darker and the underlying color will show through.

**src/components/home/Home.js**

import React from 'react'

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

const Home = (props) ⇒ {

return (

<View style = {styles.container}>

<TouchableHighlight>

<Text style = {styles.text}>

Button

</Text>

</TouchableHighlight>

</View>

)

}

export default Home

const styles = StyleSheet.create ({

container: {

alignItems: 'center',

},

text: {

borderWidth: 1,

padding: 25,

borderColor: 'black',

backgroundColor: 'red'

}

})

## Touchable Native Feedback

This will simulate ink animation when the element is pressed.

**src/components/home/Home.js**

import React from 'react'

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

const Home = (props) ⇒ {

return (

<View style = {styles.container}>

<TouchableNativeFeedback>

<Text style = {styles.text}>

Button

</Text>

</TouchableNativeFeedback>

</View>

)

}

export default Home

const styles = StyleSheet.create ({

container: {

alignItems: 'center',

},

text: {

borderWidth: 1,

padding: 25,

borderColor: 'black',

backgroundColor: 'red'

}

})

## Touchable Without Feedback

This should be used when you want to handle the touch event without any animation Usually, this component is not used much.

<TouchableWithoutFeedback>

<Text>

Button

</Text>

</TouchableWithoutFeedback>