React Native
About the Tutorial

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.

Audience

This tutorial is designed for JavaScript and React developers who aspire to learn mobile building skills. By following this course, you will expand your React and JavaScript knowledge, learn some concepts of functional programming, and prepare to enter the mobile world.

Since JavaScript world is moving forward, we will keep up with it and use EC6 syntax in this tutorial.

Prerequisites

To be able to follow this tutorial, you should be familiar with React and have solid JavaScript knowledge. Even if you do not have previous experience with React, you should be able to follow it. In this tutorial, we will explain some fundamental React concepts.

Copyright & Disclaimer

© Copyright 2017 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com.
# Table of Contents

About the Tutorial ........................................................................................................ i
Audience ....................................................................................................................... i
Prerequisites .................................................................................................................. i
Copyright & Disclaimer ............................................................................................... i
Table of Contents .......................................................................................................... ii

## CORE CONCEPTS ...................................................................................................... 1

1. React Native – Overview .......................................................................................... 2
2. React Native – Environment Setup .......................................................................... 3
3. Basic ReactNative App ............................................................................................. 15
4. React Native – State ............................................................................................... 18
   Difference between State and Props ................................................................. 18
5. React Native – Props ............................................................................................. 22
   Container Component ............................................................................................ 22
   Presentational Component .................................................................................... 23
6. React Native – Styling ............................................................................................. 27
   Container Component ............................................................................................ 27
   Presentational Component .................................................................................... 27
7. React Native – Flexbox ........................................................................................... 30
8. React Native – ListView ......................................................................................... 36
9. React Native – Text Input ....................................................................................... 40
10. React Native – ScrollView .................................................................................... 45
11. React Native – Images .......................................................................................... 48
    Adding Image ....................................................................................................... 48
    Screen Density ..................................................................................................... 49
    Network Images ................................................................................................... 50
12. React Native – HTTP ............................................................................................. 52
    Using Fetch ........................................................................................................... 52
13. React Native – Buttons ......................................................................................... 55
    Touchable Opacity ............................................................................................... 57
    Touchable Highlight ............................................................................................ 58
    Touchable Native Feedback ............................................................................... 59
    Touchable Without Feedback ........................................................................... 61
14. React Native – Animations ................................................................................... 62
    Animations Component ....................................................................................... 62
15. React Native – Debugging ..................................................................................... 64

© 2022 Tutorials Point (India) Private Limited
In App Developer Menu ................................................................................................................. 6466
16. React Native – Router .................................................................................................................. 6769
17. React Native – Running IOS ......................................................................................................... 7224
18. React Native – Running Android .................................................................................................. 7375

COMPONENTS AND APIs.............................................................................................................. 7476
19. React Native – View .................................................................................................................... 7572
   Use Cases...................................................................................................................................... 7572
20. React Native – WebView ............................................................................................................... 7779
   Using WebView .............................................................................................................................. 7779
21. React Native – Modal................................................................................................................... 7981
22. React Native – ActivityIndicator .............................................................................................. 8587
23. React Native – Picker .................................................................................................................. 8880
24. React Native – Status Bar ........................................................................................................... 9395
25. React Native – Switch ................................................................................................................ 9582
26. React Native – Text ..................................................................................................................... 98100
27. React Native – Alert .................................................................................................................. 102104
28. React Native – Geolocation ...................................................................................................... 106108
29. React Native – AsyncStorage .................................................................................................. 110112
Core Concepts
For better understanding of React Native concepts, we will borrow a few lines from the official documentation –

React Native lets you build mobile apps using only JavaScript. It uses the same design as React, letting you compose a rich mobile UI from declarative components. With React Native, you don't build a mobile web app, an HTML5 app, or a hybrid app; you build a real mobile app that's indistinguishable from an app built using Objective-C or Java. React Native uses the same fundamental UI building blocks as regular iOS and Android apps. You just put those building blocks together using JavaScript and React.

**React Native Features**

Following are the features of React Native:

- **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**

Follow are the advantages of React Native:

- **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**

Following are the limitations of React Native:

- **Native Components** – If you want to create native functionality which is not created yet, you will need to write some platform specific code.
2. 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.

<table>
<thead>
<tr>
<th>S.NO.</th>
<th>Software</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NodeJS and NPM</td>
<td>You can follow our NodeJS Environment Setup tutorial to install NodeJS.</td>
</tr>
</tbody>
</table>

**Step1: Install create-react-native-app**

After installing NodeJS and NPM successfully in your system you can proceed with installation of create-react-native-app (globally as shown below).

```bash
C:\Users\Tutorialspoint> npm install -g create-react-native-app
```

**Step2: Create project**

Browse through required folder and create a new react native project as shown below.

```bash
C:\Users\Tutorialspoint> cd Desktop
C:\Users\Tutorialspoint\Desktop>create-react-native-app MyReactNative
```

After executing the above command, a folder with specifies name is created with the following contents.
Step 3: NodeJS Python Jdk8

Make sure you have Python NodeJS and jdk8 installed in your system if not, install them. In addition to these it is recommended to install latest version of yarn to avoid certain issues.

Step 4: Install React Native CLI

You can install react native command line interface on npm, using the `install -g react-native-cli` command as shown below.

```
npm install -g react-native-cli
```
Step5: Start react native

To verify the installation browse through the project folder and try starting the project using the start command.

```
C:\Users\Tutorialspoint\Desktop>cd MyReactNative
C:\Users\Tutorialspoint\Desktop\MyReactNative>npm start
```

If everything went well you will get a QR code as shown below.
As instructed, one way to run react native apps on your android devise is to using expo. Install expo client in your android devise and scan the above obtained QR code.

**Step6: Eject the project**

If you want to run android emulator using android studio, come out of the current command line by pressing `ctrl+c`.

Then, execute `run eject` command as

```
npm run eject
```

This prompts you options to eject, select the first one using arrows and press enter.

Then, you should suggest the name of the app on home screen and project name of the Android studio and Xcode projects.

Though your project ejected successfully, you may get an error as:
Ignore this error and run react native for android using the following command-

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

But, before that you need to install android studio.

**Step 7: Installing Android Studio**

After downloading the installation file of it, double click on it and proceed with the installation.

**Step 8: Configuring AVD Manager**

To configure the AVD Manager click on the respective icon in the menu bar.
Step 9: Configuring AVD Manager

Choose a device definition, Nexus 5X is suggestable.

Click on the Next button you will see a System Image window. Select the **x86 Images** tab.
Then, select Marshmallow and click on next.

Finally, click on the Finish button to finish the AVD configuration.
After configuring your virtual device click on the play button under the Actions column to start your android emulator.

**Step 10: Running android**

Open command prompt, browse through your project folder and, execute the `react-native run-android` command.
Then, your app execution begins in another prompt you can see its status.

In your android emulator you can see the execution of the default app as:
Step 11: local.properties

Open the android folder in your project folder SampleReactNative/android (in this case). Create a file with named local.properties and add the following path in it.

```bash
sdk.dir = /C:\Users\Tutorialspoint\AppData\Local\Android\Sdk
```

here, replace Tutorialspoint with your user name.

Step 12: Hot Reloading

And to build application modify the App.js and the changes will be automatically updated on the android emulator.

If not, click on the android emulator press `ctrl+m` then, select Enable Hot Reloading option.
If you open the default app you can observe that the app.js file looks like

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

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Text>
        <Text>Changes you make will automatically reload.</Text>
        <Text>Shake your phone to open the developer menu.</Text>
      </View>
    );
  }
}

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

Output:
Hello world

To display a simple message saying "Welcome to Tutorialspoint" remove the CSS part and insert the message to be printed wrapped by the <text></text> tags inside <view></view> as shown below.

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

export default class App extends React.Component {
  render() {
    return (
      <View>
        <Text>Welcome to Tutorialspoint</Text>
      </View>
    );
  }
}```
React Native
The data inside React Components is managed by **state** and **props**. In this chapter, we will talk about **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.

### Using State

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

**App.js**

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

export default class App extends React.Component {
  state = {
    myState: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, used 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>
    );
  }
}
```

---

4. React Native – State
We can see in emulator text from the state as in the following screenshot.

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

**Home.js**

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

class Home extends Component {

    state = {
        myState: "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
    }

    deleteText = () => {
        this.setState({ myState: '' })
    }

    render() {
        return (
            <View>
                <Text>{this.state.myState}</Text>
            </View>
        )
    }
}
```
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' })

class Home extends Component {

    constructor(){
        super()
        this.updateState = this.updateState.bind(this)
    } 

    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 our next chapter.

We will also learn how to use 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)
    } 

    render() {
        return ( 
        <View>
            <Text onPress = {this.updateState}>
                {this.state.myState}
            </Text>
        </View>
        );
    }
}

export default Home;
```javascript
}

updateState() {
    //
}

render() {
    //
}
```
In our last chapter, we showed you how to use mutable state. In this chapter, we will show you how to combine the state and the 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.

App.js:

```javascript
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import PresentationalComponent from './PresentationalComponent'

export default class App extends React.Component {

  state = {
    myState: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit, used 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 style={styles.container}>
        <Text style={styles.title}>Hello, world!</Text>
      </View>
    );
  }
}
```
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.

PresentationalComponent.js

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

const PresentationalComponent = (props) => {
  return (
    <View>
      <Text onPress = {props.updateState}>
        {props.myState}
      </Text>
    </View>
  );
}
```

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.
6. React Native – Styling

There are a couple of ways to style your elements in React Native.

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.

**App.js**

```jsx
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import PresentationalComponent from './PresentationalComponent'

export default class App extends React.Component {
  state = {
    myState: 'This is my state'
  }
  render() {
    return (
      <View>
        <PresentationalComponent myState = {this.state.myState}/>
      </View>
    );
  }
}
```

**Presentational Component**

In the example given below, 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.
PresentationalComponent.js

```javascript
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.
To accommodate different screen sizes, React Native offers Flexbox support.

We will use the same code that we used in our React Native - Styling chapter. We will only change the PresentationalComponent.

**Layout**

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

The following table shows the possible options.

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

If the items needs to be aligned vertically and also centralized, consider the example given below.

**App.js**

```javascript
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'
  }
})

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

**App.js**

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

const App = (props) => {
    return (
        <View style={styles.container}>
            <View style={styles.redbox} />
            <View style={styles.bluebox} />
            <View style={styles.blackbox} />
        </View>
    )
}
```

export default App

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'
  }
})
In this chapter, we will show you how to create list in React Native. We will import `List` in our `Home` component and show it on screen.

**App.js**

```javascript
import React from 'react'
import List from './List.js'

const App = () => {
  return (<List />)
}
export default App
```

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

**List.js**

```javascript
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,
      }
    ]
  }

  render() {
    return <ListView list={this.state.names} />
  }
}
```
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,
  },
})
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.
React Native
In this chapter, we will show you how to work with **TextInput** elements in React Native. The Home component will import and render inputs.

### App.js

```javascript
import React from 'react';
import Inputs from './inputs.js'

const App = () => {
  return (<Inputs />)
}

export default App
```

### 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.

### inputs.js

```javascript
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.
In this chapter, we will show you how to work with the ScrollView element. We will again create ScrollViewExample.js and import it in Home.

**App.js**

```jsx
import React from 'react';
import ScrollViewExample from './scroll_view.js';

const App = () => {
  return (
    <ScrollViewExample />
  )
}
export default App
```

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

**ScrollView.js:**

```jsx
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},
    ]
  }
```

45
When we run the app, we will see the scrollable list of names.
11. React Native – Images

In this chapter, we will understand how to work with images in React Native.

**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.

**App.js**

```javascript
import React from 'react';
import ImagesExample from './ImagesExample.js'
const App = () => {
  return (<ImagesExample />)
}
export default App
```

Local image can be accessed using the following syntax.

**image_example.js:**

```javascript
import React, { Component } from 'react'
import { Image } from 'react-native'

const ImagesExample = () => {
  <Image source =
      {require('C:/Users/Tutorialspoint/Desktop/NativeReactSample/logo.png')} />
}
export default ImagesExample
```

Output:
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.

**App.js**

```javascript
import React from 'react';
import ImagesExample from './image_example.js'

const App = () => {
  return (<ImagesExample />
)
}
export default App
```

**image_example.js:**

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

const ImagesExample = () => (
  <Image source ={{uri:'https://pbs.twimg.com/profile_images/486929358120964097/gNLINY67_400x400.png'}}
    style = {{ width: 200, height: 200 }}
  />
)
export default ImagesExample
```

**Output:**
React Native
In this chapter, we will show you how to use `fetch` for handling network requests.

**App.js**

```javascript
import React from 'react';
import HttpExample from './http_example.js'

const App = () => {
  return (
    <HttpExample />
  )
}
export default App
```

**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.

**http_example.js**

```javascript
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);
        console.log(responseJson);
      })
  }
}
```
```javascript
this.setState({
  data: responseJson
});

.catch((error) => {
  console.error(error);
});

render() {
  return (
    <View>
      <Text>
        {this.state.data.body}
      </Text>
    </View>
  )
}

export default HttpExample
```
In this chapter, we will show you touchable components in react Native. We call them 'touchable' because they offer built in animations and we can use the `onPress` prop for handling touch event.

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

**App.js**

```jsx
import React, { Component } from 'react'
import { Button } from 'react-native'

const App = () => {
  const handlePress = () => false
  return (<Button
    onPress = {handlePress}
    title = "Red button!"
    color = "red"
  />
  )
}

export default App
```

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

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

App.js

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

const App = () => {
  return (
    <View style={styles.container}>
      <TouchableOpacity>
        <Text style={styles.text}>
          Button
        </Text>
      </TouchableOpacity>
    </View>
  )
}
export default App

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.

App.js

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

const App = (props) => {
  return (
    <View style={styles.container}>
      <TouchableHighlight>
        <Text style={styles.text}>Button</Text>
      </TouchableHighlight>
    </View>
  )
}
```
export default App

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.

### App.js

```javascript
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.

```jsx
<TouchableWithoutFeedback>
  <Text>
    Button
  </Text>
</TouchableWithoutFeedback>
```
In this chapter, we will show you how to use **LayoutAnimation** in React Native.

### Animations Component

We will set **myStyle** as a property of the state. This property is used for styling an element inside **PresentationalAnimationComponent**.

We will also create two functions: **expandElement** and **collapseElement**. These functions will update values from the state. The first one will use the **spring** preset animation while the second one will have the **linear** preset. We will pass these as props too. The **Expand** and the **Collapse** buttons call the **expandElement()** and **collapseElement()** functions.

In this example, we will dynamically change the width and the height of the box. Since the **Home** component will be the same, we will only change the **Animations** component.

### App.js

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

class Animations extends Component {
  componentWillMount = () => {
    this.animatedWidth = new Animated.Value(50)
    this.animatedHeight = new Animated.Value(100)
  }

  animatedBox = () => {
    Animated.timing(this.animatedWidth, {
      toValue: 200,
      duration: 1000
    }).start()
    Animated.timing(this.animatedHeight, {
      toValue: 500,
      duration: 500
    }).start()
  }

  render() {
    return null
  }
}
```
const animatedStyle = { width: this.animatedWidth, height: this.animatedHeight }

return (  
  <TouchableOpacity style = {styles.container} onPress = {this.animatedBox}>
    <Animated.View style = {[styles.box, animatedStyle]}/>
  </TouchableOpacity>
)

export default Animations

const styles = StyleSheet.create({
  container: {
    justifyContent: 'center',
    alignItems: 'center'
  },
  box: {
    backgroundColor: 'blue',
    width: 50,
    height: 100
  }
})
React native offers a couple of methods that help in debugging your code.

**In App Developer Menu**

You can open the developer menu on the IOS simulator by pressing `command + D`.

On Android emulator, you need to press `command + M`.
Reload

Stop Remote JS Debugging

Disable Live Reload

Start Systrace

Enable Hot Reloading

Show Inspector

Show Perf Monitor

Cancel

Reload
This is used for reloading simulator. You can use the shortcut, **command + R**
Debug JS Remotely
This is used to activate debugging inside the browser developer console.

Enable Live Reload
This is used for enabling live reloading whenever your code is saved. The debugger will open at localhost:8081/debugger-ui.

Start Systrace
This is used for starting the Android marker based profiling tool.

Show Inspector
This is used for opening inspector where you can find information about your components. You can use the shortcut, command + I.

Show Perf Monitor
This is used to track the performance of your app.
In this chapter, we will understand navigation in React Native.

Step 1 – Install Router
To begin with, we need to install the Router. We will use the React Native Router Flux in this chapter. You can run the following command in terminal, from the project folder.

```
npm i react-native-router-flux --save
```

Step 2
Since we want our router to handle the entire application, we will add it in `index.ios.js`. For Android, you can do the same in `index.android.js`.

**App.js**

```javascript
import React, { Component } from 'react';
import { AppRegistry, View } from 'react-native';
import Routes from './Routes.js'

class reactTutorialApp extends Component {
    render() {
        return (
            <Routes />
        )
    }
}
export default reactTutorialApp
AppRegistry.registerComponent('reactTutorialApp', () => reactTutorialApp)
```

Step 3 – Add Router
Now we will create the Routes component inside the components folder. It will return Router with several scenes. Each scene will need key, component and title. Router uses the key property to switch between scenes, component will be rendered on screen and the title will be shown in the navigation bar. We can also set the initial property to the scene that is to be rendered initially.

**Routes.js**

```javascript
import React from 'react'
```
import { Router, Scene } from 'react-native-router-flux'
import Home from './Home.js'
import About from './About.js'

const Routes = () => {
  <Router>
    <Scene key = "root">
      <Scene key = "home" component = {Home} title = "Home" initial = {true} />
      <Scene key = "about" component = {About} title = "About" />
    </Scene>
  </Router>
}
export default Routes

**Step 4 – Create components**

We already have the **Home** component from previous chapters; now, we need to add the **About** component. We will add the **goToAbout** and the **goToHome** functions to switch between scenes.

**Home.js**

```javascript
import React from 'react'
import { TouchableOpacity, Text } from 'react-native';
import { Actions } from 'react-native-router-flux';

const Home = () => {
  const goToAbout = () => {
    Actions.about()
  }
  return (
    <TouchableOpacity style = {{ margin: 128 }} onPress = {goToAbout}>
      <Text>This is HOME!</Text>
    </TouchableOpacity>
  )
}
export default Home
```

**About.js**
import React from 'react'
import { TouchableOpacity, Text } from 'react-native'
import { Actions } from 'react-native-router-flux'

const About = () => {
  const goToHome = () => {
    Actions.home()
  }
  return (
    <TouchableOpacity style={{ margin: 128 }} onPress={goToHome}>
      <Text>This is ABOUT</Text>
    </TouchableOpacity>
  )

export default About
The app will render the initial **Home** screen.

You can press the button to switch to the about screen. The Back arrow will appear; you can use it to get back to the previous screen.
If you want to test your app in the iOS simulator, all you need is to open the root folder of your app in terminal and run –

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

The above command will start the simulator and run the app.

We can also specify the device we want to use.

```
react-native run-ios --simulator "iPhone 5s"
```

After you open the app in simulator, you can press command + D on IOS to open the developers menu. You can check more about this in our debugging chapter.

You can also reload the IOS simulator by pressing command + R.
We can run the React Native app on Android platform by running the following code in the terminal.

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

Before you can run your app on Android device, you need to enable **USB Debugging** inside the **Developer Options**.

When **USB Debugging** is enabled, you can plug in your device and run the code snippet given above.

The Native Android emulator is slow. We recommend downloading [Genymotion](https://www.genymotion.com) for testing your app.

The developer menu can be accessed by pressing **command + M**.
Components and APIs
View is the most common element in React Native. You can consider it as an equivalent of the div element used in web development.

**Use Cases**

Let us now see a few common use cases.

- When you need to wrap your elements inside the container, you can use View as a container element.

- When you want to nest more elements inside the parent element, both parent and child can be View. It can have as many children as you want.

- When you want to style different elements, you can place them inside View since it supports style property, flexbox etc.

- View also supports synthetic touch events, which can be useful for different purposes.

We already used View in our previous chapters and we will use it in almost all subsequent chapters as well. The View can be assumed as a default element in React Native. In example given below, we will nest two Views and a text.

**App.js**

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

const App = () => {
  return (
    <View>
      <View>
        <Text>This is my text</Text>
      </View>
    </View>
  )
}

export default App
```

**Output:**
This is my text
In this chapter, we will learn how to use WebView. It is used when you want to render web page to your mobile app inline.

**Using WebView**

The **HomeContainer** will be a container component.

**App.js**

```javascript
import React, { Component } from 'react'
import WebViewExample from './web_view_example.js'

const App = () => {
    return (<WebViewExample/>
}
export default App;
```

Let us create a new file called **WebViewExample.js** inside the **src/components/home** folder.

**web_view_example.js**

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

const WebViewExample = () => {
    return (<View style={styles.container}>
        <WebView source={{ uri:'https://www.google.com/?gws_rd=cr,ssl&ei=SICcV9_EFqqk6ASA3ZaABA#q=tutorialspoint' }}
    />
}
```
The above program will generate the following output.
In this chapter, we will show you how to use the modal component in React Native.

Let us now create a new file: **ModalExample.js**

We will put logic inside **ModalExample**. We can update the initial state by running the `toggleModal`.

After updating the initial state by running the `toggleModal`, we will set the `visible` property to our modal. This prop will be updated when the state changes.

The `onRequestClose` is required for Android devices.

### App.js

```javascript
import React, { Component } from 'react'
import WebViewExample from './modal_example.js'

const Home = () => {
  return (<WebViewExample/>)
}

export default Home;
```

### modal_example.js

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

class ModalExample extends Component {
  state = {
    modalVisible: false,
  }
  toggleModal(visible) {
    this.setState({ modalVisible: visible });
  }
  render() {
```
return (
  <View style = {styles.container}>
    <Modal animationType = "slide" transparent = {false}
      visible = {this.state.modalVisible}
      onRequestClose = {() => { console.log("Modal has been closed.") }}
    >
      <View style = {styles.modal}>
        <Text style = {styles.text}>Modal is open!</Text>
        <TouchableHighlight onPress = {() => {
          this.toggleModal(!this.state.modalVisible)}}</TouchableHighlight>
        <Text style = {styles.text}>Close Modal</Text>
      </View>
    </Modal>
  </View>

  <TouchableHighlight onPress = {() => {this.toggleModal(true)}}>
    <Text style = {styles.text}>Open Modal</Text>
  </TouchableHighlight>
</View>

const styles = StyleSheet.create({
  container: {
    alignItems: 'center',
    backgroundColor: '#ede3f2',
    padding: 100
  },
  modal: {
    flex: 1,
    alignItems: 'center',
    backgroundColor: '#f7021a',
    padding: 100
  }
})
export default ModalExample
},
  text: {
    color: '#3f2949',
    marginTop: 10
  }
})
Our starting screen will look like this –
If we click the button, the modal will open.
React Native
In this chapter we will show you how to use the activity indicator in React Native.

**Step 1 – App**

App component will be used to import and show our **ActivityIndicator**.

**App.js**

```javascript
import React from 'react'
import ActivityIndicatorExample from './activity_indicator_example.js'

const Home = () => {
  return (<ActivityIndicatorExample />)
}

export default Home
```

**Step 2 – ActivityIndicatorExample**

Animating property is a Boolean which is used for showing the activity indicator. The latter closes six seconds after the component is mounted. This is done using the **closeActivityIndicator()** function.

**activity_indicator_example.js**

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

class ActivityIndicatorExample extends Component {
  state = { animating: true }

  closeActivityIndicator = () => setTimeout(() => this.setState({
    animating: false
  }), 60000)

  componentDidMount = () => this.closeActivityIndicator()

  render() {
```

85
const animating = this.state.animating

return (
    <View style = {styles.container}>
        <ActivityIndicator
            animating = {animating}
            color = '#bc2b78'
            size = "large"
            style = {styles.activityIndicator}/>
    </View>
)
}

export default ActivityIndicatorExample

const styles = StyleSheet.create ({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        marginTop: 70
    },
    activityIndicator: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        height: 80
    }
})
When we run the app, we will see the loader on screen. It will disappear after six seconds.
In this chapter, we will create simple Picker with two available options.

**Step 1 – Create File**
Here, the `App.js` folder will be used as a presentational component.

**App.js**

```javascript
import React from 'react'
import PickerExample from './PickerExample.js'

const App = () => {
  return (<PickerExample />)
}
export default App
```

**Step 2 – Logic**

`this.state.user` is used for picker control.
The `updateUser` function will be triggered when a user is picked.

**PickerExample.js**

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

class PickerExample extends Component {
  state = {user: ''}
  updateUser = (user) => {
    this.setState({ user: user })
  }
  render() {
    return (<View>
      <Picker selectedValue = {this.state.user} onValueChange = {this.updateUser}>
        23.
      </Picker>
    </View>)
  }
}
```
```jsx
<Picker>
  <Picker.Item label = "Steve" value = "steve" />
  <Picker.Item label = "Ellen" value = "ellen" />
  <Picker.Item label = "Maria" value = "maria" />
</Picker>
<Text style = {styles.text}> {this.state.user} </Text>
</View>
export default PickerExample

const styles = StyleSheet.create(
  {
    text: {
      fontSize: 30,
      alignSelf: 'center',
      color: 'red'
    }
  }
)
```

Output:
If you click on the name it prompts you all three options as:
And you can pick one of them and the output will be like.
React Native
In this chapter, we will show you how to control the status bar appearance in React Native. The Status bar is easy to use and all you need to do is set properties to change it.

The `hidden` property can be used to hide the status bar. In our example it is set to `false`. This is default value.

The `barStyle` can have three values – `dark-content`, `light-content` and `default`.

This component has several other properties that can be used. Some of them are Android or IOS specific. You can check it in official documentation.

**App.js**

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

const App = () => {
  return (
    <StatusBar barStyle = "dark-content" hidden = {false} backgroundColor = "#00BCD4" translucent = {true}/>
  )
}

export default App
```

If we run the app, status bar will be visible and content will have dark color.
React Native
In this chapter, we will explain the **Switch** component in a couple of steps.

**Step 1 – Create File**

We will use the **HomeContainer** component for logic, but we need to create the presentational component.

Let us now create a new file: **SwitchExample.js**.

**Step 2 – Logic**

We are passing value from the **state** and functions for toggling switch items to **SwitchExample** component. Toggle functions will be used for updating the state.

**App.js Example**

```javascript
import React, { Component } from 'react'
import { View } from 'react-native'
import SwitchExample from './switch_example.js'

export default class HomeContainer extends Component {
  constructor() {
    super();
    this.state = {
      switch1Value: false,
    }
  }

  toggleSwitch1 = (value) => {
    this.setState({switch1Value: value})
    console.log('Switch 1 is: ' + value)
  }

  render() {
    return ( 
```
```
Step 3 – Presentation

Switch component takes two props. The `onValueChange` prop will trigger our toggle functions after a user presses the switch. The `value` prop is bound to the state of the `HomeContainer` component.

`switch_example.js`

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

export default SwitchExample = (props) => {
  return (
    <View style = {styles.container}>
      <Switch
        onValueChange = {props.toggleSwitch1}
        value = {props.switch1Value}/>
    </View>
  );
}

const styles = StyleSheet.create ({
  container: {
    flex: 1,
    alignItems: 'center',
    marginTop: 100
  }
})
```

If we press the switch, the state will be updated. You can check values in the console.

**Output:**
In this chapter, we will talk about **Text** component in React Native.

This component can be nested and it can inherit properties from parent to child. This can be useful in many ways. We will show you example of capitalizing the first letter, styling words or parts of the text, etc.

**Step 1 – Create File**

The file we are going to create is

`text_example.js`

**Step 2 – App.js**

In this step, we will just create a simple container.

```
import React, { Component } from 'react'
import TextExample from './text_example.js'

const App = () => {
    return (
        <TextExample/>
    )
}
export default App
```

**Step 3 – Text**

In this step, we will use the inheritance pattern. `styles.text` will be applied to all `Text` components.

You can also notice how we set other styling properties to some parts of the text. It is important to know that all child elements have parent styles passed to them.

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

const TextExample = () => {
    ...
}
export default TextExample

const styles = StyleSheet.create({
  container: {
    alignItems: 'center',
    marginTop: 100,
  },
})
React Native

```javascript
{
  padding: 20,
  text: {
    color: '#41cdf4',
  },
  capitalizeLetter: {
    color: 'red',
    fontSize: 20
  },
  wordBold: {
    fontWeight: 'bold',
    color: 'black'
  },
  italicText: {
    color: '#37859b',
    fontStyle: 'italic'
  },
  textShadow: {
    textShadowColor: 'red',
    textShadowOffset: { width: 2, height: 2 },
    textShadowRadius: 5
  }
}
```
You will receive the following output –
In this chapter, we will understand how to create custom Alert component.

Step 1 – App.js

```javascript
import React from 'react'
import AlertExample from './alert_example.js'

const App = () => {
    return (<AlertExample />)
}

export default App
```

Step 2 – alert_example.js

We will create a button for triggering the showAlert function.

```javascript
import React from 'react'
import { Alert, Text, TouchableOpacity, StyleSheet } from 'react-native'

const AlertExample = () => {
    const showAlert = () => {
        Alert.alert(
            'You need to...' 
        )
    }
    return (
        <TouchableOpacity onPress = {showAlert} style = {styles.button}>
            <Text>Alert</Text>
        </TouchableOpacity>
    )
}

export default AlertExample
```
const styles = StyleSheet.create ({
  button: {
    backgroundColor: '#4ba37b',
    width: 100,
    borderRadius: 50,
    alignItems: 'center',
    marginTop: 100
  }
})
Output

When you click the button, you will see the following –
In this chapter, we will show you how to use **Geolocation**.

### Step 1 – App.js

```javascript
import React from 'react'
import GeolocationExample from './geolocation_example.js'

const App = () => {
  return (<GeolocationExample />)
}

export default App
```

### Step 2 – Geolocation

We will start by setting up the initial state for that will hold the initial and the last position.

Now, we need to get current position of the device when a component is mounted using the `navigator.geolocation.getCurrentPosition`. We will stringify the response so we can update the state.

`navigator.geolocation.watchPosition` is used for tracking the users’ position. We also clear the watchers in this step.\[vi\]

### AsyncStorageExample.js

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

class SwitchExample extends Component {
  state = {
    initialPosition: 'unknown',
    lastPosition: 'unknown',
  }
  watchID: ?number = null;

  componentDidMount = () => {
    navigator.geolocation.getCurrentPosition(
```

---

\[vi\] See the previous step for details on how to use `watchPosition`. \[vii\] For more information on `AsyncStorage`, refer to the previous chapter.
(position) => {
    const initialPosition = JSON.stringify(position);
    this.setState({ initialPosition });
},
(error) => alert(error.message),
{ enableHighAccuracy: true, timeout: 20000, maximumAge: 1000 }
);
this.watchID = navigator.geolocation.watchPosition((position) => {
    const lastPosition = JSON.stringify(position);
    this.setState({ lastPosition });
});
}
componentWillUnmount = () => {
    navigator.geolocation.clearWatch(this.watchID);
}
render() {
    return (
        <View style = {styles.container}>
            <Text style = {styles.boldText}>
            Initial position:
            </Text>
            
            <Text>
            {this.state.initialPosition}
            </Text>
            
            <Text style = {styles.boldText}>
            Current position:
            </Text>
            
            <Text>
            {this.state.lastPosition}
            </Text>
        </View>
    )
}
export default SwitchExample

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    marginTop: 50
  },
  boldText: {
    fontSize: 30,
    color: 'red',
  }
})
In this chapter, we will show you how to persist your data using **AsyncStorage**.

### Step 1 – Presentation

In this step, we will create the **App.js** file.

```jsx
import React from 'react'
import AsyncStorageExample from './async_storage_example.js'

const App = () => {
  return (<AsyncStorageExample />)
}

export default App
```

### Step 2 – Logic

**Name** from the initial state is empty string. We will update it from persistent storage when the component is mounted.

**setName** will take the text from our input field, save it using **AsyncStorage** and update the state.

```jsx
async_storage_example.js

import React, { Component } from 'react'
import { StatusBar } from 'react-native'
import { AsyncStorage, Text, View, TextInput, StyleSheet } from 'react-native'

class AsyncStorageExample extends Component {

  state = {
    'name': ''
  }

  componentDidMount = () => AsyncStorage.getItem('name').then((value) =>
    this.setState({ 'name': value })
  )

  setName = (value) => {
    AsyncStorage.setItem('name', value);
    this.setState({ 'name': value });
  }

} export default AsyncStorageExample
```
When we run the app, we can update the text by typing into the input field.