Getting started guide for React Native

Erica Ahlqvist and Anna Furugård

January 29, 2020

Contents

1 Introduction

This guide will cover how to get started with React Native. You need to have React Native installed to be able to follow the guide. It is recommended to start installing Expo CLI and Node.js. You can find a full guide on how to get started by follow this link

This guide will mainly focus on:

- Simple Layout of components/widgets
- Interaction with listeners/callback functions
- Navigation between screens

1.1 Simple Layout

When working with React Native, it is necessary to know about the different views. Views help you organise your containers into the right place and make sure that they stay inline with one another.

1.1.1 View

The first one is Simple View, one of the fundamental assets when creating a React Native application. With Simple View you create a container of your choice. The container can easily be changed, for example different color, size and alignment, from the stylesheets. You can also apply a style called flex on it, but more on that later.

```
<View
   style={{ YOUR STYLE HERE }}>
   // You can have more than one view in a view!
        <View style={{ YOUR STYLE HERE }} />
        <View style={{ YOUR STYLE HERE }} />
</View>
```

Listing 1: View

1.1.2 Scrollview

When working with applications you often want to add a ScrollView to ensure that the user can see everything provided on the screen in a smooth way. ScrollView is a pretty straight forward method and as the name says it allows the user to scroll in the view. When adding a ScrollView the height can be infinite even if the container itself has a fixed height which is useful when working with applications and the device has a limited screen, i.e. a phone or tablet.

An important thing to remember when working with ScrollViews is that it must have a bounded height! If you do not have a bounded height it will simply not work, this is because the container contains unbounded-height children. This can easily be fixed by adding a height to the container, or that all parents have a set height.

```
7  import { ScrollView } from 'react-native';
8
9  <ScrollView style={{ YOUR STYLE HERE }}>
10        // You can add text here
11       </ScrollView>
```

Listing 2: ScrollView

1.1.3 KeyboardAvoidingView

When the device has a limited screen as well as the keyboard on the screen you may come upon the issue when the keyboard hides the content, which is not user friendly. To ensure that the content move away from the keyboard you can use the KeyboardAvoidingView, as the name says this view avoids the keyboard.

This is especially useful if you create an application and the device has a virtual keyboard and the screen is rather small so that there is a risk that the keyboard will hide the content. To ensure that the view in fact will move for the keyboard you add "enabled" in the style, however this is by default set to true.

```
12 import { KeyboardAvoidingView } from 'react-native';
13
14 <KeyboardAvoidingView style={{ YOUR STYLE HERE }} enabled>
15        // You can add your content here
16       </KeyboardAvoidingView>
```

Listing 3: KeyboardAviodingView

1.1.4 SafeAreaView

Next up is the SafeAreaView, which main purpose it to only render the content in the safe area boundaries of the device. The paddding of the SafeAreaView reflects the limitations of the screen, i.e. the size of the actual screen. SafeAreaView is used to wrap the whole content, as well as adding a flex : 1 style, other than this you can of course use a background color and other styles to it.

Worth noting here is that it is yet only applicable to iOS 11 devices or later.

```
17 import { SafeAreaView } from 'react-native';
18
19 <SafeAreaView style={{ flex : 1 }}>
20        // You can add your content here
21       </SafeAreaView>
```

Listing 4: SafeAreaView

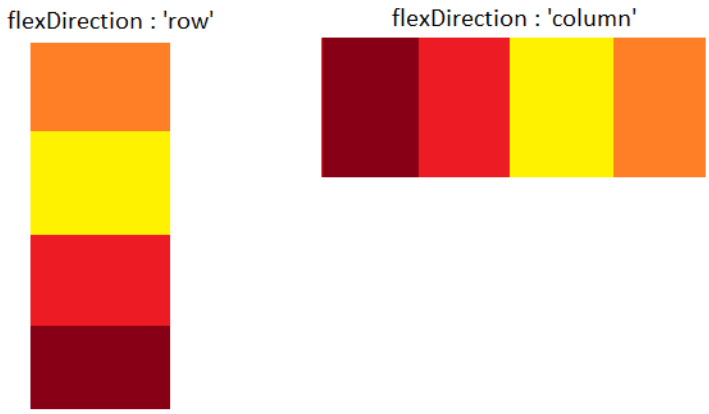
1.1.5 FlexBox

FlexBox is used on a component to specify the layout of its children. FlexBox is a CSS layout mode, but is works in the same way for React Native with a few modifications.

FlexDirection wallows the children of a container to be placed in rows or columns.

```
22 import { View } from 'react-native';
23
24 <View style={{ flex : 1, flexDirection : 'row' }}>
25        // When adding containers here they will appear in a
      row
26       </View>
27
28 <View style={{ flex : 1, flexDirection : 'column' }}>
29        // When adding containers here they will appear in a
      column
30       </View>
```

Listing 5: flexDirection

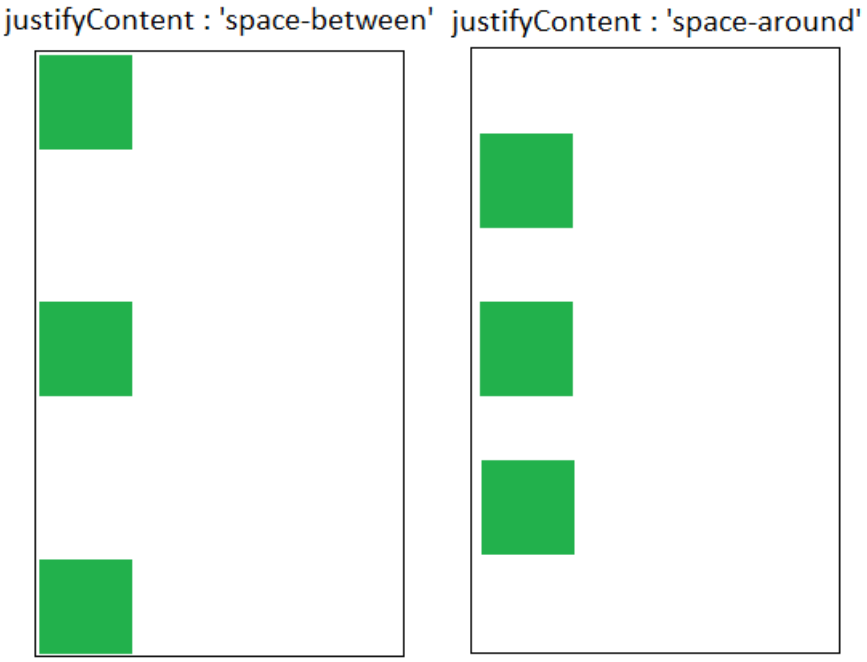


By default the content will be placed in the left-top corner of the screen. To change this you can use justifyContent, which describes how the children should be aligned within the container. When using justifyContent you can use *flex-start, flex-end, center, space-between, space-around and space-evenly.*

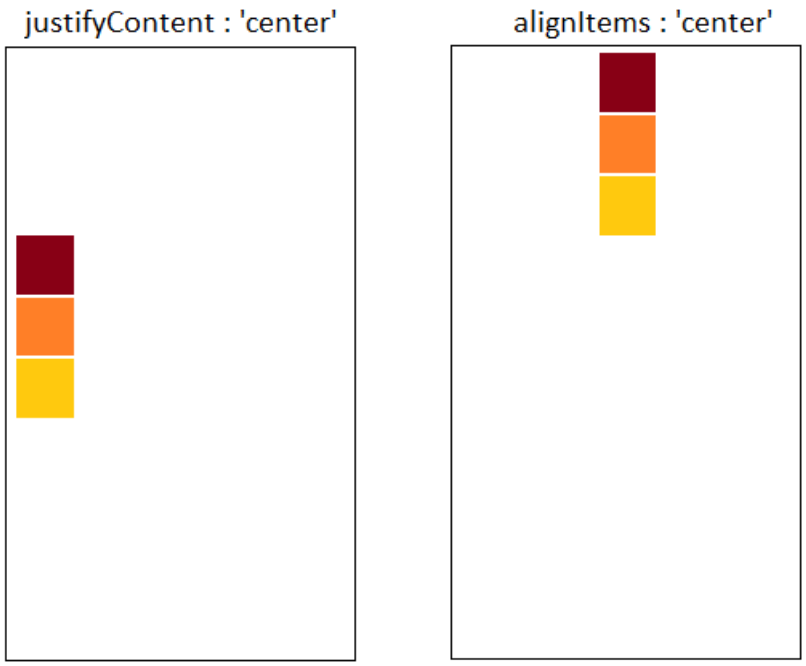
Flex-start and *flex-end* aligns the children to the start/end of the container's main axis. *Space-between* inserts an even space between the children across the main axis, *space-around* works similar but instead distribute the space around the children.

```
31 import { View } from 'react-native';
32
33 <View style={{ flex : 1, flexDirection : 'row',
      justifyContent :'center' }}>
34         // When adding containers here they will appear in a
35         // row
36         // These will also be centered to the container
37       </View>
38
39 <View style={{ flex : 1, flexDirection : 'column',
      justifyContent : 'space-between' }}>
40         // When adding containers here they will appear in a
41         // column
42         // These will also have space between them
43       </View>
```

Listing 6: justifyContent



As discussed the justifyContent will place the children according to the main axis, if you instead want to apply the children to the cross axis you can use alignItems. However these can be used together to create the wanted result.




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

<View style={{ flex : 1, flexDirection : 'row',
   justifyContent :'center', alignItems : 'stretch' }}>
       // When adding containers here they will appear in a
       // row
       // These will also be centered to the container
       // It will also be stretched to fill the cross axis
</View>
```

Listing 7: alignItems

There is a lot more to learn about FlexBox and we recommend you to check this link, it will help you understand the concept of FlexBox in an interactive way.

1.2 Callback functions

A callback function is a function that is accessible by another function. A callback function is invoked when the first function (from where the callback function is accessible by) is finished executing.

When using React Native, you will be writing in JavaScript. This leads to **callback functions** will be written with the same syntax as in JavaScript.

```

function doWork(jobDescription, callback) {
    alert('Started working with ${jobDescription}.');
    callback();
}

function workIsDone(){
    alert('Finished with work!');
}

doWork('Dishwashing', workIsDone);
```

Listing 8: Callback function

This would result in, if you called *doWork('Dishwashing', workIsDone);*:

```

Started working with Dishwashing.
Finished with work!
```

Listing 9: Callback function

1.3 Listeners

Event listeners makes it possible for the web page/application respond when an event is occurring, an example of event is when pressing a button.

The function **addEventListener()** is a built in JavaScript function which will, when the event is occurring, call for another argument.

```

class exampleButton extends React.Component{


    constructor(props){
        super(props);
        this.state = {resultText: ''}

        //Binding the function to the class.
        this.handlePress = this.handlePress.bind(this);
    }

    //The function that is executed when the button is
   pressed. The function set the state.
```

```
82      handlePress(){
83          var tempText = "You have changed the text";
84          this.setState({resultText: tempText});
85      }
86
87      render(){
88          return(
89              //An "onPress"-event which will occur when the
    button is pressed.
90              <Button onPress = {this.handlePress}/>
91              <Text>{resultText}</Text>
92          )
93      }
94  }
```

Listing 10: Listeners

In order for the event to occur in React Native, the function need to be linked to the class, see line 78. This is necessary for displaying variables when setting the state.

In this example the **handlePress** function is attached to a button (with a "onPress"-event), but it can be attached other components as well.

1.4 Navigating between different screens

When working with different screens in React Native you need to install *react-native-gesture-handler* as well as *react-native-stack*. The easiest way to do this is by using *yarn add* in the Command-line interface. You can also use *npm install* to install the components.

```
96  yarn add react-native-gesture-handler
97
98  yarn add react-native-stack
```

Listing 11: Yarn add

With these installed you can start working on the application. Firstly you have to create a *navigator* to declare the different screen components.

```
99  import {createAppContainer} from 'react-navigation';
100 import {createStackNavigator} from 'react-navigation-stack';
101 //remember to add the new components!
102
103 const myNavigator = createStackNavigator({
104     Page1: {screen: Page1Screen},
```

```
    Home: {screen: HomeScreen},
});

const App = createAppContainer(myNavigator);

export default App;
```

Listing 12: Navigator

After creating the navigator it's time to link it to the screens.

```
class Page1Screen extends React.Component {
// A components class per page!
  static navigationOptions = {
    title: 'This is page 1',
  };
  render() {
    const {navigate} = this.props.navigation;
    return (
      // YOUR TEXT HERE
    );
  }
}
```

Listing 13: Link the screens

You have now have created multiple screens for your application!