Contents

[REACT NATIVE 2](#_Toc168242104)

[REACT NATIVE – UNDER THE HOOD 3](#_Toc168242105)

[COMPILATION 3](#_Toc168242106)

[SETTING UP REACT NATIVE APP – USING EXPO 3](#_Toc168242107)

[CREATIING REACT NATIVE PROJECT 3](#_Toc168242108)

[REACT NATIVE -CORE CONCEPTS 4](#_Toc168242109)

[STYLING 4](#_Toc168242110)

[LAYOUT & FLEXBOX 4](#_Toc168242111)

[EVENT HANDLING 13](#_Toc168242112)

[MAKING CONTENT SCROLLABLE WITH SCROLLVIEW 14](#_Toc168242113)

[OPTIMIZING LIST WITH FLASTLIST 15](#_Toc168242114)

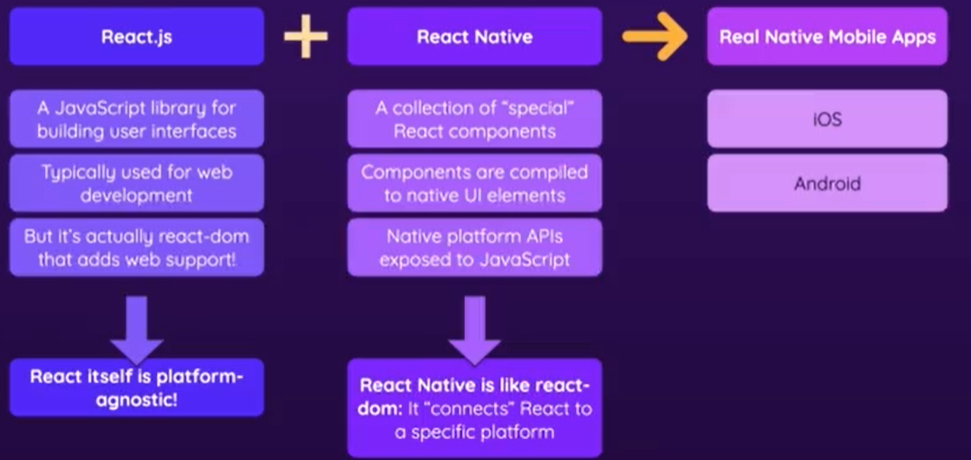
[HANDLING TAPS USING PRESSABLE COMPONENT 18](#_Toc168242115)

[CREATING CUSTOM COMPONENTS 18](#_Toc168242116)

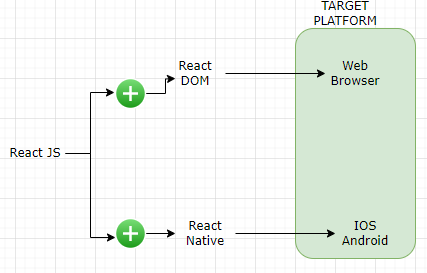
[STYLING FOR ANDROID AND IOS 19](#_Toc168242117)

[NAVIGATION 19](#_Toc168242118)

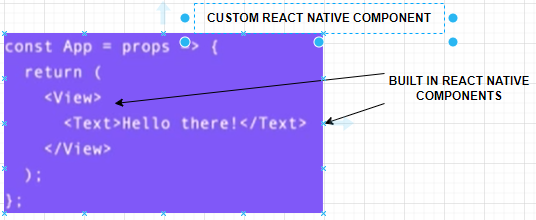
# REACT NATIVE



* React Native helps in building React Native based mobile apps for iOS and Android, which can distribute through the app stores and apps that can be used by any user with an iOS or Android device.



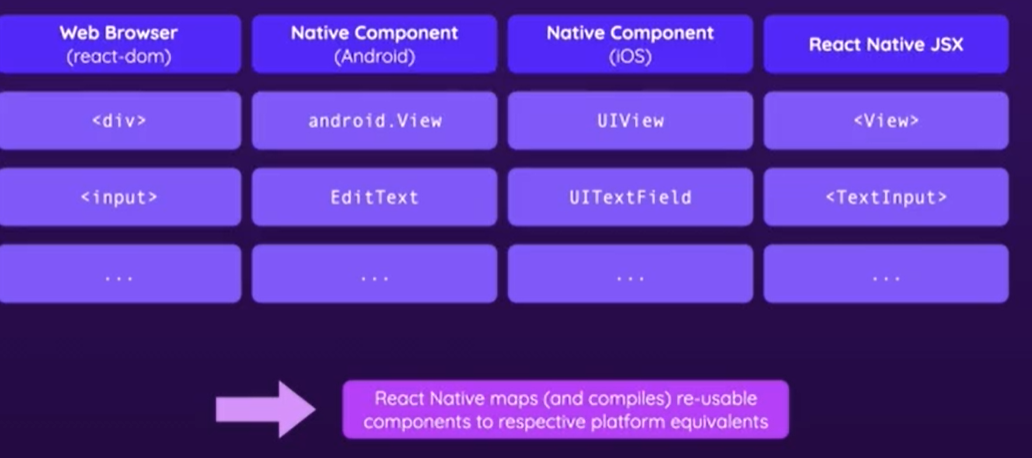
* Now - React.js is as library does not care about the underlying platform. React just gives us tools for managing state, for building virtual component trees. It is the React DOM library that adds the actual web support.
* Similarly React Native is basically an alternative to React DOM. It gives us
  + Collection of special built-in react components, which we can use in JSX code. Those components are then compiled to native UI elements for the iOS and Android platforms



* + It also exposes certain native platform APIs like using the device camera

## REACT NATIVE – UNDER THE HOOD

### COMPILATION



* the JSX elements (with the in-built react native components) are compiled to native elements for the respective platforms.
* The JSX elements are compiled to their native equivalence. But the JavaScript cod e(that we write outside of JSX) is the logic is not compiled but instead it's running on a JavaScript thread that's hosted by React Native
* in the native app that was built.
* So React Native basically spins up a simple JavaScript process as part of our native app that's being built and it manages this process for us.
* It allows this process to talk to the underlying native platform (Android /IOS)

## SETTING UP REACT NATIVE APP – USING EXPO

* Expo is a set of tools and services for building, deploying, and managing native iOS, Android, and web applications using JavaScript and React.
* It provides a development environment and a suite of tools to simplify the process of developing and deploying mobile apps.

### CREATIING REACT NATIVE PROJECT

|  |  |
| --- | --- |
| CREATE EXPO DRIVEN REACT NATIVE PROJECT | npx create-expo-app@latest <project\_name> |
| FOR A BLANK & BASIC PROJECT | npx create-expo-app –template blank <project\_name> |

## REACT NATIVE -CORE CONCEPTS

|  |  |
| --- | --- |
| CORE COMPONENTS LIBRARY | <https://reactnative.dev/docs/components-and-apis> |

* We use core component to build our react native custom components.

### STYLING

These is no CSS in react native. The way styling can be applied.

Example

|  |  |
| --- | --- |
| Inline style using props (style) on the core components | Using StyleSheet Object  (*The Stylesheet object helps in reusability of styles )* |
| export default function App() {    return (      <View style={styles.container}>        <Text style={{color:'blue', borderWidth:2, borderColor:'red',padding:16}}>Open up App.js to start working on your app Hello Work!</Text>      </View>    );  }  **USING THE STYLESHEET OBJECT**  export default function App() {  return (  <View style={styles.container}>  <Text style={styles.textStyle}> Hello Work!</Text>  <Text style={styles.textStyle}>Dummy text data</Text>  </View>  );  }  const styles = StyleSheet.create({  container: {  flex: 1,  backgroundColor: '#fff',  alignItems: 'center',  justifyContent: 'center',  },  textStyle:{  color:'blue',  borderWidth:2,  borderColor:'red',  padding:16  }  }); | import { StatusBar } from 'expo-status-bar';  import { StyleSheet, Text, View } from 'react-native';  export default function App() {  return (  <View style={styles.container}>  <Text>Open up App.js to start working on your app Hello Work!</Text>  <StatusBar style="auto" />  </View>  );  }  const styles = StyleSheet.create({  container: {  flex: 1,  backgroundColor: '#fff',  alignItems: 'center',  justifyContent: 'center',  },  }); |

### LAYOUT & FLEXBOX

#### OVERVIEW OF FLEXBOX

* Flexbox, short for "Flexible Box Layout," is a CSS layout module that provides a flexible way to layout and align elements within a container.
* It is designed to simplify the process of building responsive and flexible web layouts.
* With Flexbox, we can define the layout and behavior of elements in a container along a single axis (row or column) or both axes simultaneously.
* It offers a set of properties that allow us to control how the elements are sized, aligned, and spaced within the container.
* FlexBox Cheat Sheet : <https://reactnative.dev/docs/flexbox>
* **By default - every view in React Native, even if we assign no special styles, uses Flexbox.**

#### PRINCIPLES OF FLEXBOX

##### FLEXBOX AXIS

* In Flexbox - we have two important axis
  + MAIN AXIS
    1. The main axis depends on the flex direction.
    2. For flex direction row - the main axis is from left to right.
    3. For a row reverse, it would be right to left.
    4. For column, it would be top to bottom
    5. For column reverse it would be bottom to top.
  + CROSS AXIS -

1. **It is simply the opposite of the main axis**.
2. For a row where the main axis is from left to right 🡪 the cross axis would be from top to bottom.
3. If the main axis is from right to left 🡪 The cross axis would be from bottom to top.

|  |  |
| --- | --- |
|  |  |

Note :

1. When we specify “display:flex” to the container- it will align all the child elements on a row from left to right. Hence by default left to right is the main axis & top 🡪 bottom will be cross axis.

|  |  |
| --- | --- |
| To align items in main axes | justifyContent |
| To align items in cross axes | alignItems |

#### FLEXBOX CONTAINERS

* The parent element that contains the flex items.
* It is defined by setting the CSS property "display" to "flex" or "inline-flex".

###### DIFFERENT PROPERTIES ON FLEX CONTAINERS

The different properties on flex container will help in aligning the child within the container

1. FLEX DIRECTION

|  |  |
| --- | --- |
| HTML | <section class="**container**">  <div></div>  <div></div>  <div></div>  <div></div>  </section> |
| **CSS** | **UI** |
| .container {      display: flex;  }  OR  .container {      display: flex;  flex-direction: row;  } | * **By default, the flex container is going to line up all the elements in a row from left to right.** |
| .container {      display: flex;  flex-direction: row-reverse;  } |  |
| .container {      display: flex;  flex-direction: column;  } |  |
| .container {      display: flex;  flex-direction: column-reverse;  } |  |

1. JUSTIFY CONTENT : ALIGN CHILD ELEMENTS IN MAIN AXIS

* Justify-content property will always work on main axis (which is set by flex direction). Note that the justify-content property only affects the alignment of flex items along the main axis. To align items along the cross axis (vertically by default), you would use the align-items property
* justify-content has six different values(work quite similar to text-align)

|  |  |
| --- | --- |
| CSS | UI |
| **flex-start (default):**  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  justify-content: flex-start;  } | Flex items are aligned to the start of the container. |
| **flex-end**:  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  justify-content: flex-end;  } | Flex items are aligned to the end of the container. |
| **center**:  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  justify-content:center;  } | Flex items are centered within the container. |
| **space-between**  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  justify-content:space-between;  } | Flex items are evenly distributed along the main axis. The first item is positioned at the start of the container, the last item at the end, and the remaining space is divided equally between the items |
| **space-around**  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  justify-content:space-around;  } | Flex items are evenly distributed along the main axis, with equal space around them. The space is divided between the items and also before the first item and after the last item. |
| **space-evenly**  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  justify-content:space-evenly;  } | Flex items are evenly distributed along the main axis, with equal space around them. The space is divided equally between the items, including before the first item and after the last item. |

1. ALIGN- ITEM: ALIGN ITEMS IN CROSS AXIS

* The align-items property is used to align flex items along the cross axis (vertically by default).
* It determines how flex items are positioned within their flex container.

|  |  |
| --- | --- |
| CSS | UI |
| stretch (default):  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  height: 300px;  **align-items: stretch;**  } | **Flex items are stretched to fill the container along the cross axis.** |
| flex-start: **Flex items are aligned to the start of the container along the cross axis.**  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  height: 300px;  **align-items: flex-start;**  } |  |
| flex-end: **Flex items are aligned to the end of the container along the cross axis.**  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  height: 300px;  **align-items: flex-end;**  } |  |
| center: **Flex items are centered within the container along the cross axis.**  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  height: 300px;  **align-items: center;**  } |  |
| baseline: **Flex items are aligned based on their baselines (the line that text sits on).**  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  height: 300px;  align-items: baseline;  }  /\*css for last div in the container\*/  .container div:last-child {  font-size: 50px;  } |  |

1. FLEX-WRAP : ALIGN CHILD ELEMENTS IN MAIN AXIS

* The flex-wrap property is used to control whether flex items should wrap or not when they exceed the width of their flex container.

|  |  |
| --- | --- |
| CSS | UI |
| **nowrap** (default  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  width: 300px;  flex-wrap: no-wrap;  } | Flex items will be forced to fit within a single line. This means that if the flex items exceed the width of the container, they will shrink to fit. |
| **wrap**:  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  width: 300px;  flex-wrap: wrap;  } | Flex items will wrap onto multiple lines when necessary. This means that if the flex items exceed the width of the container, they will move to the next line. |
| **wrap-reverse**:  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  width: 300px;  flex-wrap: wrap-reverse;  } | Flex items will wrap onto multiple lines, but in reverse order. This means that if the flex items exceed the width of the container, they will move to the next line, but in reverse order. |

1. ALIGN-CONTENT : ALIGN CHILD ELEMENTS IN MAIN AXIS

* the align-content property is used to control how multiple lines of flex items are aligned along the cross axis when there is extra space in the flex container.
* It is very similar to justify-content. But it's important to differentiate between these two is **justify-content** defines spacing along the main axis, even if we have just one row. But **align-conten**t defines the spacing along the cross axis only if we have more than one row .

|  |  |
| --- | --- |
| CSS | UI |
| **flex-start**:  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  width: 300px;  align-items: flex-start;  align-content: flex-start;  flex-wrap: wrap;  height: 300px;  } | Lines of flex items are packed at the start of the container along the cross axis. |
| **flex-end**:  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  width: 300px;  align-items: flex-start;  align-content: flex-end;  flex-wrap: wrap;  height: 300px;  } | Lines of flex items are packed at the end of the container along the cross axis. |
| **center**:  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  width: 300px;  align-items: flex-start;  align-content: center;  flex-wrap: wrap;  height: 300px;  } | Lines of flex items are centered within the container along the cross axis. |
| **space-between**: Lines of flex items are evenly distributed along the cross axis. The first line is positioned at the start of the container, the last line at the end, and the remaining space is divided equally between the lines.  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  width: 300px;  align-items: flex-start;  align-content:space-between;  flex-wrap: wrap;  height: 300px;  } |  |
| **space-around**  .container {  border: 1px solid #fff;  display: flex;  flex-direction: row;  width: 300px;  align-items: flex-start;  align-content:space-around;  flex-wrap: wrap;  height: 300px;  } | Lines of flex items are evenly distributed along the cross axis, with equal space around them. The space is divided between the lines and also before the first line and after the last line. |
| **stretch (default**): Lines of flex items are stretched to fill the container along the cross axis. |  |

* **Flex Items**:
  + The child elements within the flex container.
  + They are laid out and positioned based on the flexbox rules.
* **Main Axis**:
  + The primary axis along which the flex items are laid out. It can be either horizontal (row) or vertical (column) depending on the flex container's "flex-direction" property.
* **Cross Axis**:
  + The perpendicular axis to the main axis. It is used for aligning flex items when they do not take up the full available space.
* **Flex Properties**:
  + Flexbox provides a set of properties such as "flex-grow", "flex-shrink", and "flex-basis" to control the sizing, growth, and shrinking behavior of flex items.
* **Alignment**:
  + Flexbox offers properties like "justify-content" and "align-items" to control the alignment of flex items along the main and cross axes.

EXAMPLE

Lets start with a simple example with following code

|  |  |
| --- | --- |
| import React from 'react';  import {Text, View} from 'react-native';  export default function App() {      return (          <View style={{padding: 50}}>              <View                  style={{                      backgroundColor: 'red',                      width: 100,                      height: 100,                      justifyContent: 'center',                      alignItems: 'center',                  }}>                  <Text>1</Text>              </View>              <View                  style={{                      backgroundColor: 'blue',                      width: 100,                      height: 100,                      justifyContent: 'center',                      alignItems: 'center',                  }}>                  <Text>2</Text>              </View>              <View                  style={{                      backgroundColor: 'green',                      width: 100,                      height: 100,                      justifyContent: 'center',                      alignItems: 'center',                  }}>                  <Text>3</Text>              </View>          </View>      );  } | * **By default – Flexbox organizes its children in columns** |
|  | |
|  | |
|  | |

### EVENT HANDLING

* The application will have text input to input the goal
* Once the “Add Goal” is pressed. The Goal will be added to the list below

|  |  |
| --- | --- |
| INITIAL STATE | FINAL STATE |
|  |  |
| import { StyleSheet, Text, View, Button, TextInput } from 'react-native';  import React, { useState } from 'react';  export default function App() {    const [goal, setGoal] = React.useState('');    const [goalList, setGoalList] = React.useState([]);    function getGoalInputHandler(enteredText) {      setGoal(enteredText);    }    function addGoalHandler() {      setGoalList([goal, ...goalList]);      setGoal('');    }    return (      <View style={styles.appContainer}>        <View style={styles.inputContainer}>          <TextInput            style={styles.textInput}            placeholder="Your Course Goal"            onChangeText={getGoalInputHandler}            value={goal}          />          <Button title="Add Goal" onPress={addGoalHandler} />        </View>        <View>          {goalList.map((goal, index) => (            <Text key={index}>{goal}</Text>          ))}        </View>      </View>    );  }  const styles = StyleSheet.create({    appContainer: {      paddingTop: 50,      paddingHorizontal: 16,    },    inputContainer: {      flexDirection: 'row',      justifyContent: 'space-between',      paddingBottom: 24,    },    textInput: {      borderColor: '#cccccc',      borderWidth: 1,      width: '70%',      marginRight: 8,      padding: 8,    },  }); | |

### MAKING CONTENT SCROLLABLE WITH SCROLLVIEW

* The ScrollView component is used to create a scrollable view that allows users to scroll through a larger content area. It provides a way to handle scrolling and display content that exceeds the available screen space.
* Different props available to tweak the behavior of Scroll View : <https://reactnative.dev/docs/view#props>

|  |
| --- |
| import { StyleSheet, Text, View, Button, TextInput, ScrollView} from 'react-native';  import React, { useState } from 'react';  export default function App() {    const [goal, setGoal] = React.useState('');    const [goalList, setGoalList] = React.useState([]);    function getGoalInputHandler(enteredText) {      setGoal(enteredText);    }    function addGoalHandler() {      setGoalList([goal, ...goalList]);      setGoal('');    }    return (      <View style={styles.appContainer}>        <View style={styles.inputContainer}>          <TextInput            style={styles.textInput}            placeholder="Your Course Goal"            onChangeText={getGoalInputHandler}            value={goal}          />          <Button title="Add Goal" onPress={addGoalHandler} />        </View>        <View style={styles.goalContainer}>        <ScrollView>          {goalList.map((goal, index) => (            <Text key={index} style={styles.goalContainer}>{goal}</Text>          ))}          </ScrollView>        </View>      </View>    );  }  const styles = StyleSheet.create({    appContainer: {      paddingTop: 50,      paddingHorizontal: 16,    },    inputContainer: {      flexDirection: 'row',      justifyContent: 'space-between',      paddingBottom: 24,    },    textInput: {      borderColor: '#cccccc',      borderWidth: 1,      width: '70%',      marginRight: 8,      padding: 8,    },    goalContainer:{      marginVertical:20    }  }); |

### OPTIMIZING LIST WITH FLASTLIST

* In React Native, the FlatList component is used to efficiently render large lists of data.
* It provides an optimized way to render and interact with long lists, as it only renders the items that are currently visible on the screen.

EXAMPLE

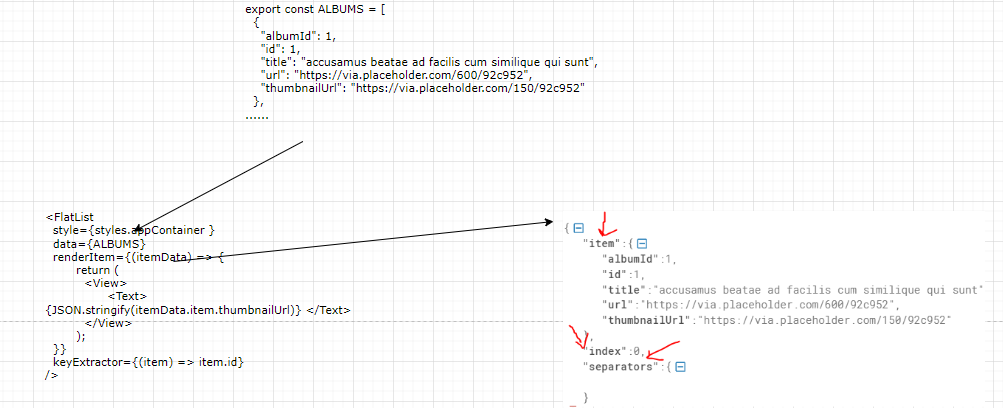
|  |
| --- |
| import React from 'react';  import { FlatList, Text, View } from 'react-native';    const MyFlatList = () => {  const data = [  { id: '1', title: 'Item 1' },  { id: '2', title: 'Item 2' },  { id: '3', title: 'Item 3' },  // Add more data items here  ];    const renderItem = ({ item }) => (  <View style={{ padding: 10 }}>  <Text>{item.title}</Text>  </View>  );    return (  <FlatList  data={data}  renderItem={renderItem}  keyExtractor={(item) => item.id}  />  );  }    export default MyFlatList; |

* In this example, we have a functional component called "MyFlatList" that renders a FlatList component.
* The data array contains the list of items to be rendered. Each item in the array is an object with an id and title property.
* The renderItem function is a callback function that takes an item from the data array and returns the UI representation of that item. In this case, it renders a View component with some text content.

THE FLATLIST COMPONENT TAKES THREE PROPS:

* **data**: The array of data to be rendered.
* **renderItem**:
  + This props let the Flatlist know – How to render the individual item .
  + The callback function receives each item in the data array and render it.
  + The item received by the callback function is an object , automatically created by Flatlist. The object has 3 properties

1. “item“ 🡪 This holds the actual data items. Hence to access the actual data : itemData.item.thumbnailUrl
2. “index” property- which can be leverged in keyExtractor props of FlatList



* **keyExtractor**:
  + A function that returns a unique key for each item. In this example, we use the item's id as the key.
  + The keyExtractor prop wants a function as a value, for example, an inline arrow function, which will automatically receive two parameter values,**item and index**
  + These two values will be provided by FlatList when it calls this function for every list item that's being rendered.

By providing these props, the FlatList component efficiently renders the list of items. It only renders the items that are currently visible on the screen, recycling the components as the user scrolls.  
EXAMPLE 2:

|  |
| --- |
| import { StyleSheet, Text, View, Button, TextInput,FlatList} from 'react-native';  import { ALBUMS } from './data/data';  export default function App() {    return (      <View style={styles.appContainer}>  <View>          <FlatList            style={styles.appContainer }            data={ALBUMS}            renderItem={(itemData) => {              return (                <View>                  <Text>{JSON.stringify(itemData.item.thumbnailUrl)} </Text>                </View>              );            }}            keyExtractor={(item) => item.id}          />        </View>      </View>    );  }  const styles = StyleSheet.create({    appContainer: {      paddingTop: 50,      paddingHorizontal: 16,    }  }); |

### HANDLING TAPS USING PRESSABLE COMPONENT

* Now we want delete the “Goal” by tapping on them . To make the item Tappable we need to wrap the itesm between Pressable component.
* In my "GoalItem," I wrap my view
* with "Pressable," and with that,
* this item becomes pressable.
* And whenever we press any item inside of "Pressable,"
* so in this case, this view with this text,
* this "onPress" prop will trigger the function
* that we provide to it.
* So we should provide a function to "onPress"
* and since I plan on deleting items in the future,
* I will define this function in App.js,
* because here, I'm managing my "courseGoals" state,
* and here, I will have to delete items in the future.
* So I will add a "deleteGoalHandler" function here in App.js.
* And for the moment, I'll just "console.log('DELETE')" here
* in this function.
* We will of course replace this with proper code later.
* Now, "deleteGoalHandler" should be provided
* to the goal item,
* In the infer, I'll add a prop here,
* and since it's my component,
* it's my choice how this prop should be named,
* and I'll name it "onDeleteItem"
* and pass my "deleteGoalHanlder" function to that.
* A pointer to this "deleteGoalHandler" function
* is passed as a value to "onDeleteItem"
* on my "GoalItem" component.
* And now this "onDeleteItem" prop can be used
* in that component.
* And here, I can for example,
* simply forward it.
* Like that.
* We might tweak it later but for the moment,
* we can just do it like this.
* Now, if I save this,
* If I tap on one of these items, I get no visual feedback,
* but if I go back, you see the that "DELETE" was locked
* to the terminal.
* So indeed, this already works.
* 'Course, some visual feedback would be nice
* and we will add it in a second,
* but for the moment, it's great that this works.

## CREATING CUSTOM COMPONENTS

|  |  |
| --- | --- |
| CUSTOM COMPONENT (Button Component) | USING CUSTOM COMPONENT |
| import {Text, View} from 'react-native';  function PrimaryButton({children}){    return (      <View>        <Text>{children}</Text>      </View>  )  }  export default PrimaryButton; | <PrimaryButton>Reset</PrimaryButton> |

## STYLING FOR ANDROID AND IOS

## NAVIGATION

* Documentation Link : <https://reactnavigation.org/docs/getting-started/>

|  |  |
| --- | --- |
| INSTALL REACT NAVIGATION PACKAGES | npm install @react-navigation/native |