

Activity No. 2 Exploring Core Components and Component Styling

Course Code:	Program: Computer Engineering
Course Title:	Date Performed:
Section:	Date Submitted:
Name:	Instructor:

1. Objective(s)

This activity aims to create a basic react native application to enable students to learn how to use core components and style them.

2. Intended Learning Outcomes (ILOs)

After this module, the student should be able to:

- Utilize React Native components and build user interfaces;
- Style React Native applications; and
- Implement additional interactivity and manage states.

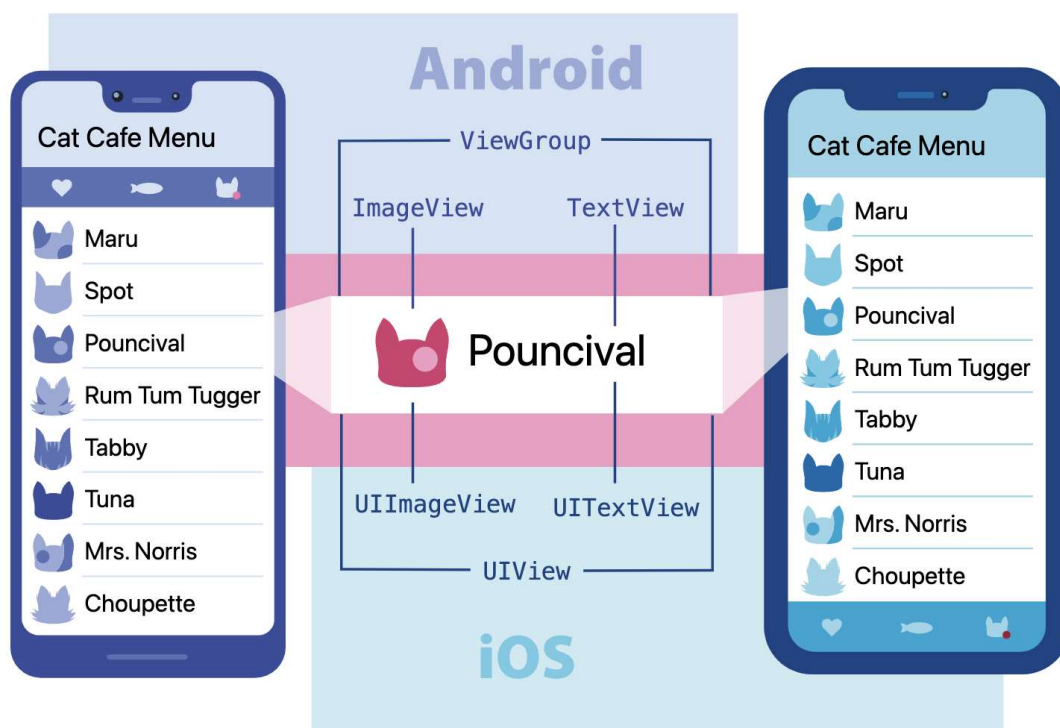
3. Discussion

Core Components and Native Components

React Native is an open source framework for building Android and iOS applications using React and the app platform's native capabilities. With React Native, you use JavaScript to access your platform's APIs as well as to describe the appearance and behavior of your UI using React components: bundles of reusable, nestable code.

Views and mobile development

In Android and iOS development, a view is the basic building block of a UI: a small rectangular element on the screen which can be used to display text, images, or respond to user input. Even the smallest visual elements of an app, like a line of text or a button, are kinds of views. Some kinds of views can contain other views. It's views all the way down!



Native Components

In Android development, you write views in Kotlin or Java; in iOS development, you use Swift or Objective-C. With React Native, you can invoke these views with JavaScript using React components. At runtime, React Native creates the corresponding Android and iOS views for those components. Because React Native components are backed by the same views as Android and iOS, React Native apps look, feel, and perform like any other apps. We call these platform-backed components Native Components.

React Native comes with a set of essential, ready-to-use Native Components you can use to start building your app today. These are React Native's Core Components.

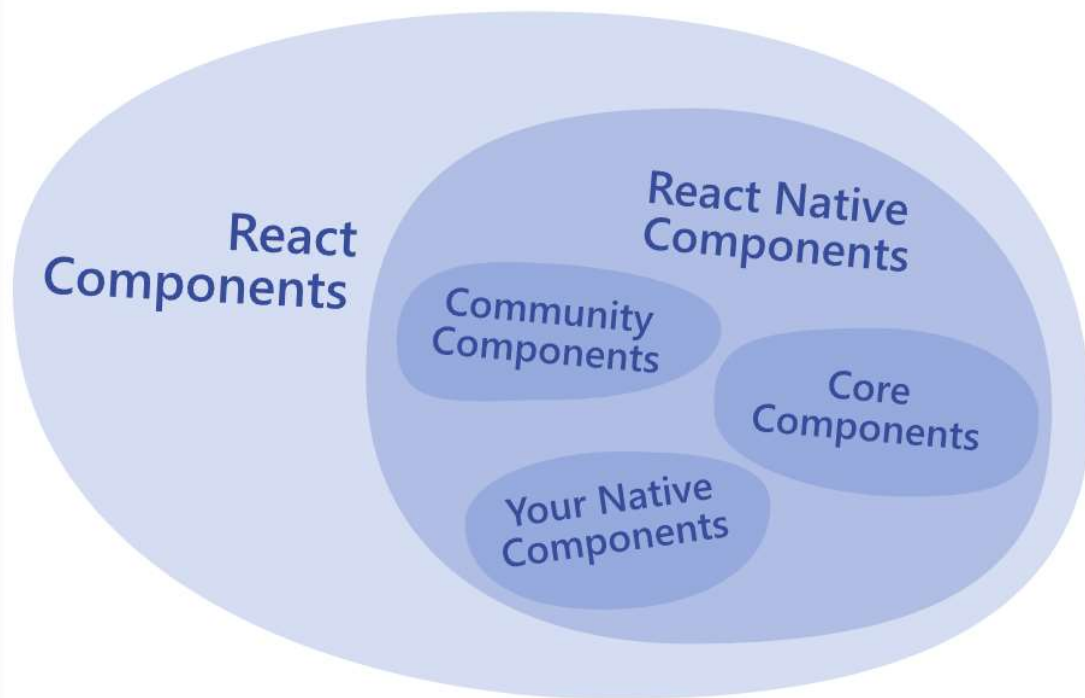
React Native also lets you build your own Native Components for Android and iOS to suit your app's unique needs. We also have a thriving ecosystem of these community-contributed components. Check out Native Directory to find what the community has been creating.

Core Components

React Native has many Core Components for everything from controls to activity indicators. You can find them all documented in the API section. You will mostly work with the following Core Components:

REACT NATIVE UI COMPONENT	ANDROID VIEW	iOS VIEW	WEB ANALOG	DESCRIPTION
<code><View></code>	<code><ViewGroup></code>	<code><UIView></code>	A non scrolling <code><div></code>	A container that supports layout with flexbox, style, some touch handling, and accessibility controls
<code><Text></code>	<code><TextView></code>	<code><UITextView></code>	<code><p></code>	Displays, styles and nests strings of text and even handles touch events.
<code><Image></code>	<code><ImageView></code>	<code><UIImageView></code>	<code></code>	Displays different types of images.
<code><ScrollView></code>	<code><ScrollView></code>	<code><UIScrollView></code>	<code><div></code>	A generic scrolling container that can contain multiple components and views.
<code><TextInput></code>	<code><EditText></code>	<code><UITextField></code>	<code><input type="text"></code>	Allows the user to enter text.

Because React Native uses the same API structure as React components, you'll need to understand React component APIs to get started. The next section makes for a quick introduction or refresher on the topic. However, if you're already familiar with React, feel free to skip ahead.



Discussion content taken from official React Native documentation. View more here <https://reactnative.dev/docs/intro-react-native-components> and read more.

4. Materials and Equipment

To properly perform this activity, the student must have:

- Node.js LTS 18.17.1
- Visual Studio Code
- Android Emulator

5. Procedure

Before starting with the procedures, ensure that you have the project from the previous activity. If you have not, you may recreate the project and install dependencies such as `npm install`. You may then check by running the commands `a` or `i` to run on your selected *Android* or *iOS* devices, respectfully. Note: You may also choose to preview the app on the web by pressing `w`. This will require you to install dependencies but is not recommended for the class.

By exploring the previously created project, your `app.js` may look similar to the figure. The figure below shows what is call a *React* component, specifically a **functional React component**.

```
export default function App() {
  return (
    <View style={styles.container}>
      <Text>Team Name</Text>
      <Text>Roman Richard</Text>
      <StatusBar style="auto" />
    </View>
  );
}
```

Aside from this, you can find a styles constant – the StyleSheet object – below.

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

At the top, there are imports to import one component from an Expo related package, and other components from React Native.

```
JS App.js > ...
1 import { StatusBar } from 'expo-status-bar';
2 import { StyleSheet, Text, View } from 'react-native';
```

Text and View are two of some built-in components that React Native exposes to you to use in your JSX code. within the functional component, you **cannot** use h2 tags or any other HTML elements that work in a browser.

Setting Up Your Android Simulator

Option 1: Using Android Studio Simulator.

1. Start Expo by typing expo start on the command line.
2. Open Android Studio.
3. Open your project in Android Studio.
4. Open the 'AVD manager' from the Android Studio toolbar.
5. Run an Android emulator.
6. Click 'run on Android device/emulator' in the Expo client.

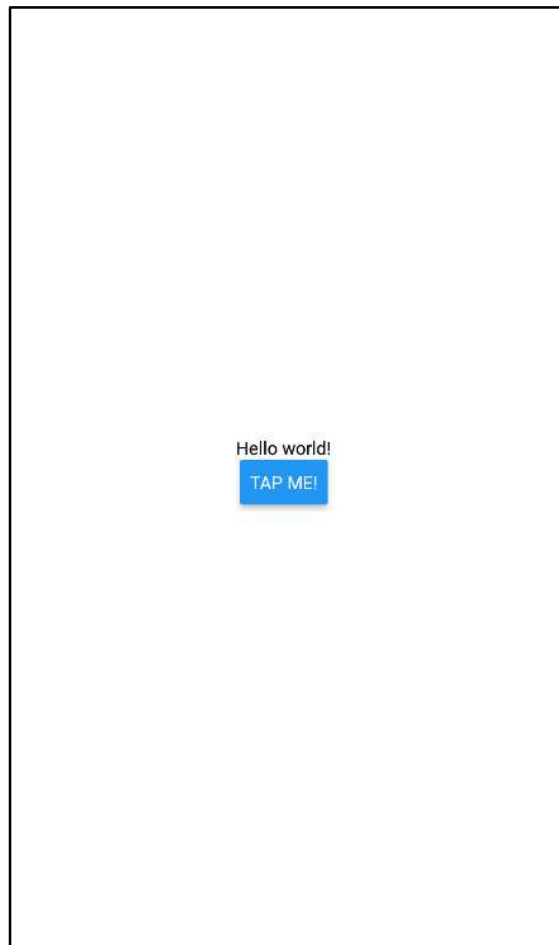
Option 2: Using any Android Emulator. This will not include any steps; all it entails are the same things you've done on your actual device. Copy and paste the socket from your terminal to the expo go app to run your application demo.

Checking Out Simple Core Components

1. In your *app.js* delete the import for the *expo-status-bar* and delete *StatusBar* under *View*.
2. Save the *app.js* and see if there are any changes to the output.
3. Delete the `<Text></Text>` component tags in your code. Your code should look similar to below:

```
1  import { StyleSheet, Text, View, Button } from 'react-native';
2
3  export default function App() {
4    return (
5      <View style={styles.container}>
6        Team Name
7        Roman Richard
8      </View>
9    );
10 }
```

4. Save *app.js*. Screenshot your output. What does this mean? Make sure that for this step, you are previewing the app on your phone and not web deployment.
5. Preview it on the web. What output do you get?
6. Add a nested *View* in your first *View* component and add a button by adding `<Button title = "Tap me!" />`.
7. Your output must be similar to this figure below. (Disregard the kind of simulator used in the figure)



Styling React Native Apps

1. Create a new Text component with the name of your team members.
2. For each Text component, add a margin of 16. Show the output.
3. Next, add a border: try a border through a provided shortcut like 'blue' and through hex code. Show the output for both.
4. Next, add a padding with the same value as the margin. Show the output.
5. Add a 'color' style to the Button component with the value "green". Show the output.

We will further explore additional styling methods as we go further. However, feel free to check the React Native documentation if you wish to explore more. If you only followed the steps above, your app should look similar to the figure below.



6. Create another View component with a dummy text nested alongside the component we added with a red border. Copy the style from the old Text in the View component to the new one. Your output must be similar below.

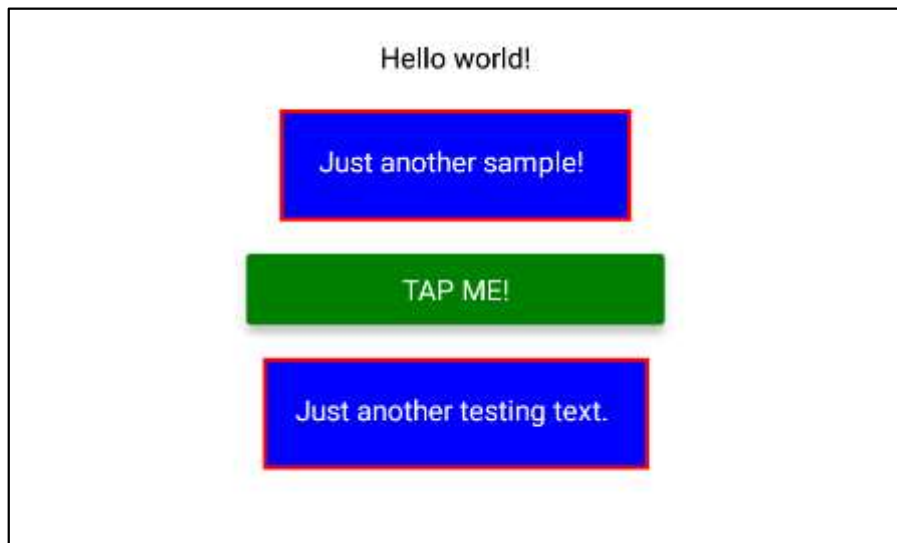


7. However, this is inefficient to have to do repeatedly. Instead, we will create an object under our StyleSheet. Your code must now be similar to the figures below. Did your output change? Why/Why not?

```
export default function App() {
  return (
    <View style={styles.container}>
      <Text>Hello world!</Text>
      <View>
        <Text style={styles.textStyle}>
          <Button title="Tap me!" color="
        </View>
      <View>
        <Text style={styles.textStyle}>
        </View>
      </View>
    );
  );
}
```

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  textStyle: {
    margin: 16,
    borderWidth: 2,
    borderColor: 'red',
    padding: 16,
  }
});
```

8. Add a new style to the *textStyle* StyleSheet component. Background color should be 'blue' and the color should be 'white'. Save the file and observe the app, what changed? Was this method more convenient? Your output must be similar to the figure shown.



9. Create a new style property called *Styles2*, this will not use the imported React Native StyleSheet nor its built-in methods. The figure below shows the steps taken and the corresponding output. Show your output.


```
const styles2 = {
};
```

Hello world!
Just another sample!

TAP ME!

Just another testing text.

```
const styles2 = {
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  textStyle: {
    margin: 16,
    borderWidth: 2,
    borderColor: 'red',
    backgroundColor: 'blue',
    color: 'white',
    padding: 16,
  },
};
```

```
export default function App() {
  return (
    <View style={styles.container}>
      <Text>Hello world!</Text>
      <View>
        <Text style={styles2.textStyle}>
          <Button title="Tap me!" color="red">TAP ME!</Button>
        </View>
        <View>
          <Text style={styles2.textStyle}>Just another testing text.</Text>
        </View>
      </View>
    );
}
```

Hello world!

Just another sample!

TAP ME!

Just another testing text.

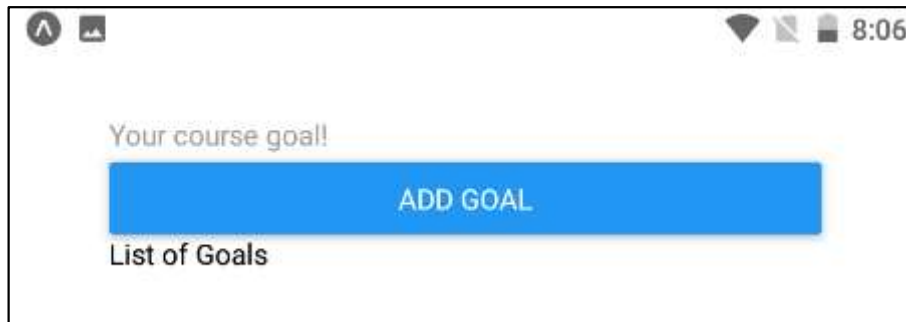
FOR THE NEXT SECTION: We will start creating a basic app that will **help us manage our goals**. To do this, we will first clean up our JSX. Before continuing, your code should look similar to what is shown below:

```
JS App.js > ...
1  import { StyleSheet, Text, View, Button } from 'react-native';
2
3  export default function App() {
4    return (
5      <View>
6        <View></View>
7        <View></View>
8      </View>
9    );
10 }
11
12 const styles = StyleSheet.create({
13
14 });
```

Exploring Layouts and Flexbox

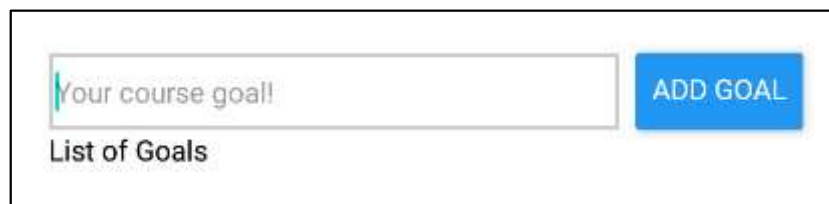
To create our goal manager app, perform the following:

1. In the first nested View, add a *TextInput* component (don't forget to import!) with a *placeholder* containing the string 'Your Course Goal'.
2. Next, add a *Button* below it with the *title* 'Add Goal'.
3. In the second nested View, add a *Text* component that says "List of Goals". This will be a goal later on.
4. Show the output.
5. Create a styles object called *appContainer* meant to contain the entire app with a padding of 50 pixels. Your output must be similar to the output below.



Now, we will use FlexBox.

1. Our goal is to make the button and the textinput components be next to each other. To do this, we are going to create a new styles object for the view that holds both components – we will call it *inputContainer*. Add *flexDirection* set to row and *justifyContent* set to space-between. Assign it to that View component.
2. Show your output from #1.
3. Next, we want to distinguish the textInput component. Create a new object under styles called *textInput*, add a border with a width of 2 and a color of #cccccc. Assign to the TextInput then show the output.
4. Next, add a width of 80%. Make sure it is a string. Show the output.
5. Add a marginRight with a value 8 and a padding value 3. Show the output.
6. Your output must be similar to the figure below.



Improving the Layout

1. Change paddings in *appContainer* to *paddingTop: 50*, *paddingHorizontal: 16* and adding *textInput width to 70% and padding to 13*. Show the output so far.
2. Notice the weird placement of the text on your button now. Try to add a style to prop to fix this. What happened?
3. Add *alignItems: center* to the *inputContainer*. Did this fix the problem? What does this do?
4. To further fix the layout, add a *paddingBottom* to the *inputContainer* with a value of 24, create a 1 pixel border on the bottom with the same color as the textInput's border. Show the result.
5. The textInput area has to take 1/4 of the overall available height. To do this, a flex property has to be added. Add *flex 1* to the *inputContainer*.
6. We are also going to add a new object under styles for the second nested view to support this. Create a new object called *goalsContainer* and give it a flex value of 3.
7. Show the output so far. Is that the expected output?

- Working on the inner containers isn't enough. The outer container must also be adjusted. The appContainer must be forced to take all the available height. Add flex: 1. Show the output.
- Change the flex in goalsContainer to 4. Change paddingBottom to marginBottom in the inputContainer. Show the output.
- Change the flex in goalsContainer to 5. Show the output. This will be your final layout *for now* as we proceed with the next steps.

Handling Events

Now, we have a textbox for user input and a button that should place this text into the goal list given when the button is tapped. In order to make these happen, we have to handle events.

1. Make two functions and add them before the return statement in your *export default function App()*:

```
function goalInputHandler() {}  
function addGoalHandler() {};
```

It must now look similar to the figure below.

```
export default function App() {  
    function goalInputHandler() {};  
  
    function addGoalHandler() {};  
  
    return (  

```

2. First, we will connect the `textInput` component that should take in the texts given by the user through the use of the `onChangeText={}` prop built-in with React Native. The `onChangeText` prop wants a function as a value; a pointer at a function. So, we'll pass the `goalInputHandler`. But not as a function call.

This now means that the `goalInputHandler` function receives the entered value automatically. The entered value becomes the parameter to the function, so we add a parameter to this function called *enteredText*.

To check, add `console.log(enteredText);` to the body of the function. Save and then type “Testing” to see whether the connection works properly.

Your output should be similar to the figure shown below.

3. However, this saved text should be handled by the button, not the text box. So, now that we've confirmed that the value per keystroke is passed, remove the contents of `goalInputHandler`'s function body.

In React Native, we have to use the `onPress={}` prop to connect our function passed as input to the prop. It should now appear as `onPress={addGoalHandler}`; and similar to earlier, it must not be in the form of a function call.

4. Now that it's connected, we want to get the `enteredText` in the `textInput` to be handled by our button. We can do this through `states`. At the top of your imports, include:

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

To use the state, first call `useState` and set as an empty string initially.

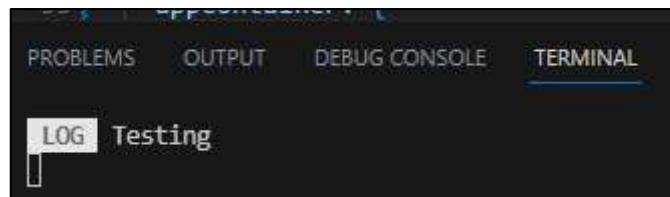
```
const [] = useState("");
```

The idea is we want to include within the `[]` the user input state or `enteredGoalText` state which can be updated by the `setEnteredGoalText` function. This is standard React syntax.

5. Going back to `goalInputHandler`, we want to call the update function from the `useState` call above and pass the `enteredText` as its parameter. Your code should look similar to the figure below.

```
export default function App() {  
  const [enteredGoalText, setEnteredGoalText] = useState('');  
  
  function goalInputHandler(enteredText) {  
    setEnteredGoalText(enteredText);  
  };  
  
  function addGoalHandler() {}  
}
```

6. We now want to test if we have access to this text from the `useState`. Add `console.log(enteredGoalText);` in your `addGoalHandler`'s function body. Save and type down "Testing" and your output should be similar to the figure below.



By the end of step 6, your code should be similar to the figure below.

```

JS App.js > ...
1  import { useState } from 'react';
2  import { StyleSheet, Text, View, Button, TextInput } from 'react-native';
3
4  export default function App() {
5    const [enteredGoalText, setEnteredGoalText] = useState('');
6
7    function goalInputHandler(enteredText) {
8      setEnteredGoalText(enteredText);
9    };
10
11   function addGoalHandler() {
12     console.log(enteredGoalText);
13   };

```

Managing the List of Course Goals

Now, we want to be able to update our list of goals that found in the secondary view component.

1. Typically, when data changes dynamically we use state. Such as in this case, with the data for the list of goals changing, we want to use `useState` as well. Add another `useState` below the first one created in the earlier steps. However, this second state will be initialized with an empty array instead.

This state should be named *courseGoals* that can be changed by the *setCourseGoals* function.

2. In the body of your `addGoalHandler`, instead of using the console, use the `setCourseGoals`. We want to be able to take your existing goals and append (add to the end) a new one.

We do this by passing an array to the `setCourseGoals` function.

```
setCourseGoals([]);
```

Then we use the *spread* operator to spread existing course goals into the new array, keeping all the existing goals.

```
setCourseGoals([...courseGoals]);
```

And then adding the new goal.

```
setCourseGoals([...courseGoals, enteredGoalText]);
```

HOWEVER, this is also not the best way to update a state if the new state depends on the previous state. Instead of the aforementioned code, we will pass a function to the state updating function that will be automatically called by react.

We will use an arrow function instead.

```
setCourseGoals((currentCourseGoals) => [...currentCourseGoals, enteredGoalText,]);
```

3. At this point, your code should look similar to the figure below.

```
JS App.js > App
1  import { useState } from 'react';
2  import { StyleSheet, Text, View, Button, TextInput } from 'react-native';
3
4  export default function App() {
5    const [enteredGoalText, setEnteredGoalText] = useState('');
6    const [courseGoals, setCourseGoals] = useState([]);
7
8    function goalInputHandler(enteredText) {
9      setEnteredGoalText(enteredText);
10   };
11
12   function addGoalHandler() {
13     setCourseGoals((currentCourseGoals) =>[
14       ...currentCourseGoals,
15       enteredGoalText,
16     ]);
17   };
18
19   return (
```

4. The next goal is to output the list of goals. Since this data is dynamic, we can now refer to *courseGoals* enclosed in curly brackets then call the *map* method on it.

`{currentGoals.map()}`

We will pass here another arrow function, which takes individual *courseGoals* for the parameter and as a return, it returns a JSX element.

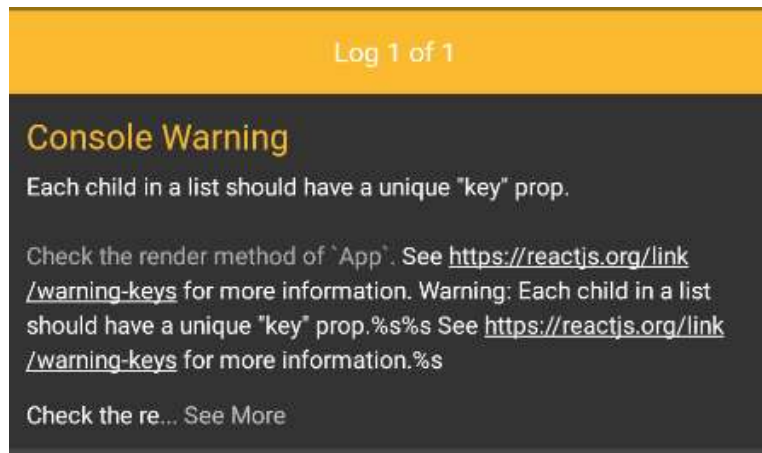
`{currentGoals.map((goal) => <Text>{goal}</Text>)}`

At this point, your code should look like the figure below.

```
19   return (
20     <View style={styles.appContainer}>
21       <View style={styles.inputContainer}>
22         <TextInput
23           placeholder='Your course goal!'
24           style={styles.textInput}
25           onChangeText={goalInputHandler}
26         />
27         <Button title='Add Goal' onPress={addGoalHandler} />
28       </View>
29       <View style={styles.goalsContainer}>
30         {courseGoals.map((goal) => <Text>{goal}</Text>)}
31       </View>
32     </View>
33   );
34 }
```

5. Save and run your code. Input in the textInput area as a goal “Learn React Native”. Press on “Add Goal” button. What happens? Was it added? Did anything else happen? Show output/screenshots.

You should receive a warning. Not because of React Native, but due to the nature of React in general. The error states that when outputting a list of data every item should receive a key prop.



To make it quick, this is meant for React to manage the list in a more efficient way. Always add the key prop when dealing with individual items being outputted as part of a list.

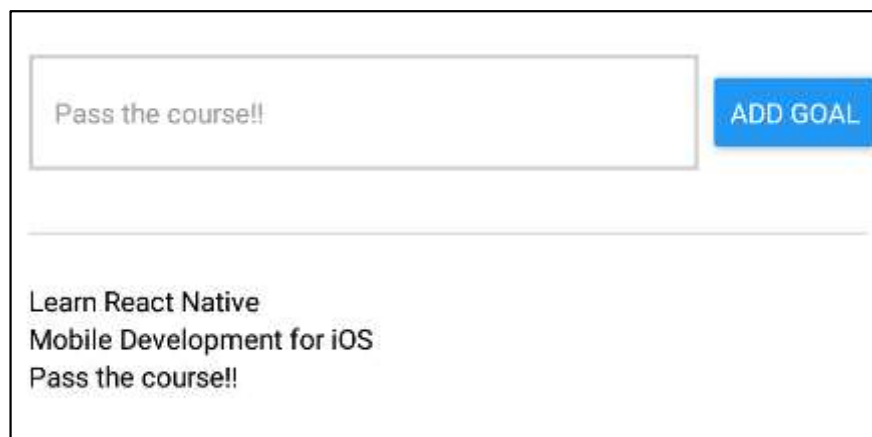
6. The value passed to the key **can be anything** but it must be unique. It has to uniquely identify the concrete value that's being outputted. In this case, each string is *assumed to be* unique so we'll use the goal string itself.

```
<Text key={goal}>{goal}</Text>
```

Temporarily, it will work. Test it with different goals and same goals. Such as:

“Learn React Native” / “Mobile Development for iOS” / “Pass the Course” (*Unless you do not want to learn anything and not pass the course.*)

The app in your simulator should look similar to the figure below.



Make sure to save your progress! The application will be continued/finished in the next activity.

6. Output

7. Supplementary Activity

Now that you have experienced how to use React Native in the development of a basic application with the guides provided in this material, it is time for you to experiment on your own.

ILO1: Utilize React Native components and build user interfaces and ILO2: Style React Native applications.

- Using the base application created in the procedure of this activity, stylize the application further. You may change fonts, add images, and even further try out different ways to use flexbox. However, one thing must remain: *the application must still retain the same function*; by the end of this activity, it must still be a goal list application for this course. You're only changing the style.
- Hint: You get points for making *aesthetic* choices. Please avoid bright, obnoxious and uncoordinated color coordinations. You will be graded accordingly.

ILO3: Implement additional interactivity and manage states.







- The application lacks in providing ample user experience. To change this, make it so that every time the “add goal” button is pressed, the previous inputs are also cleared reverting back to the placeholder.
- Sample output for ILO3 in figures below.

The figure displays three sequential screenshots of a mobile application interface, illustrating the state of the app after each 'ADD GOAL' button press. Each screenshot features a text input field with the placeholder text 'Your course goal!' and a blue 'ADD GOAL' button to its right. Below the input field is a horizontal line, and further down is a list of previously added goals.

- Top Screenshot:** The input field contains the placeholder text. The list below the line is empty.
- Middle Screenshot:** The input field contains the placeholder text. The list below the line contains one item: 'Learn React Native'.
- Bottom Screenshot:** The input field contains the placeholder text. The list below the line contains two items: 'Learn React Native' and 'Mobile App Development'.

8. Conclusion

9. Assessment Rubric

T.I.P. SO 7											 
Criteria	Ratings										Pts
 T.I.P. SO 7.1 Acquire and apply new knowledge from outside sources threshold: 4.2 pts	6 pts [Excellent] Educational interests and pursuits exist and flourish outside classroom requirements, knowledge and/or experiences are pursued independently and applies knowledge learned into practice		5 pts [Good] Educational interests and pursuits exist and flourish outside classroom requirements, knowledge and/or experiences are pursued independently		4 pts [Satisfactory] Look beyond classroom requirements, showing interest in pursuing knowledge independently		3 pts [Unsatisfactory] Begins to look beyond classroom requirements, showing interest in pursuing knowledge independently		2 pts [Poor] Relies on classroom instruction only	1 pts [Very Poor] No initiative or interest in acquiring new knowledge	6 pts
 T.I.P. SO 7.2 Learn independently threshold: 4.2 pts	6 pts [Excellent] Completes an assigned task independently and practices continuous improvement		5 pts [Good] Completes an assigned task without supervision or guidance		4 pts [Satisfactory] Requires minimal guidance to complete an assigned task		3 pts [Unsatisfactory] Requires detailed or step-by-step instructions to complete a task		2 pts [Poor] Shows little interest to complete a task independently	1 pts [Very Poor] No interest to complete a task independently	6 pts
 T.I.P. SO 7.3 Critical thinking in the broadest context of technological change threshold: 4.2 pts	6 pts [Excellent] Synthesizes and integrates information from a variety of sources; formulates a clear and precise perspective; draws appropriate conclusions		5 pts [Good] Evaluate information from a variety of sources; formulates a clear and precise perspective.		4 pts [Satisfactory] Analyze information from a variety of sources; formulates a clear and precise perspective.		3 pts [Unsatisfactory] Apply the gathered information to formulate the problem		2 pts [Poor] Gather and summarized the information from a variety of sources but failed to formulate the problem	1 pts [Very Poor] Gather information from a variety of sources	6 pts
 T.I.P. SO 7.4 Creativity and adaptability to new and emerging technologies threshold: 4.2 pts	6 pts [Excellent] Ideas are combined in original and creative ways in line with the new and emerging technology trends to solve a problem or address an issue.		5 pts [Good] Ideas are creative and adapt the new knowledge to solve a problem or address an issue		4 pts [Satisfactory] Ideas are creative in solving a problem, or address an issue		3 pts [Unsatisfactory] Shows some creative ways to solve the problem		2 pts [Poor] Shows initiative and attempt to develop creative ideas to solve the problem	1 pts [Very Poor] Ideas are copied or restated from the sources consulted	6 pts
Total Points: 24											

10. Additional Screenshots