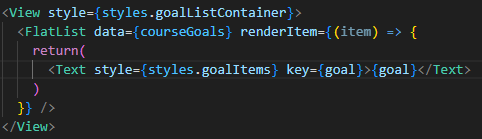
|  |  |
| --- | --- |
| **Activity No. 3 More Components and Styling** | |
| **Course Code:** | **Program:** Computer Engineering |
| **Course Title:** | **Date Performed:** |
| **Section:** | **Date Submitted:** |
| **Name:** | **Instructor:** |
| **1. Objective(s)** | |
| This activity aims to finalize the simple React Native application developed in the previous activity by allowing students to utilize more components and implement further styling techniques. | |
| **2. Intended Learning Outcomes (ILOs)** | |
| After this module, the student should be able to:   * Utilize React Native components to make scrollviews and optimized lists; and * Decompose the app component into multiple custom components. | |
| **3. Discussion** | |
| This laboratory does not come with a separate discussion. | |
| **4. Materials and Equipment** | |
| To properly perform this activity, the student must have:   * Node.js LTS * Expo App * Android Emulator/Simulator (or any iOS/Android mobile phone) | |
| **5. Procedure** | |
| *Sample Goal List Application from the Previous Activity*  ***Before we begin with this section***, include a screenshot of your goal list application after you’ve finished the application so far in the previous activity. | |

# Making Scrollable Content with ScrollView



1. In your goal list application, keep adding goals until the screen can no longer hold the elements. Similar to the image above.
2. Import *ScrollView* from React Native and change the view component containing your list of goals into

*<ScrollView></ScrollView>*. Note that style must be retained. Save and show the output.

1. What can you say about how the ScrollView looks now with your goals?
2. Now, create another view component such that your ScrollView becomes encapsulated by the <View></View> component. The style from the ScrollView component must be applied to this new View component instead. Did anything change? What do you think is the benefit of having the ScrollView component contained by the View component?
3. Create a new style object called *goalListContainer* and assign it to the View component containing your ScrollView. Manipulate it by using props like height and flex. Observe what happens to the scrollable content; add screenshots for the tests you perform.

# Optimizing Lists with FlatList

A common problem with using ScrollView is that even if items are not yet viewable, they are still rendered. ScrollView renders all its items regardless of the number. This becomes a performance issue when you have tens of thousands of items in the item – and they are rendered even if they’re not viewable. So, for situations like that, we do not want to use ScrollView. We will use another component called *FlatList* to solve this problem.

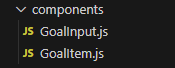
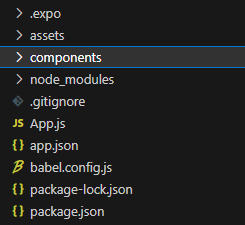
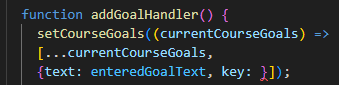
1. Import FlatList and change ScrollView in your function App to FlatList.
2. Remove the mapping of the course goals that was done previously. This task will now be passed to FlatList. Your code must look similar to below.
3. In order to pass this task to FlatList, we must use *data* prop and pass our list.
4. Remove the text component and turn FlatList into a self-closing component.
5. Add another required prop called *requiredItem*. This prop requires a function to tell it how to render the items in the given data. In this case, we will add the code we removed from earlier as the content of an arrow function.

In the example below, the arrow function passed to the requiredItem prop is the item function, which returns the Text component from earlier. Your code must look similar to the example below.

1. The *item* passed here is actually an object generated by FlatList internally that is wrapped around the individual data items. We will rename it to *itemData* to fully reflect this function. This object has its own properties, some of which we will get to see. We can track the index using itemData.index, but the prop we need here is *itemData.item* which gives us the individual items in the data.

So, change the goal to *itemData.item* and remove the key prop from the view, since we are no longer mapping it manually. Show the output.

1. Since we are still dealing with lists, keys are still important and nowhere in our code a key exists. To solve this, we will turn our data values from primitive values like strings into objects that have a key property.



Return to your addGoalHandler() function and change *enteredGoalText* by wrapping it into an object and add a text property and a key property. Your code should look similar to below.

Once this has been added, you must define your key value. For the purpose of this procedure, we will use

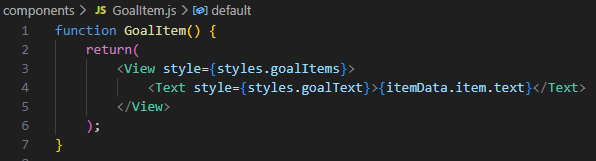
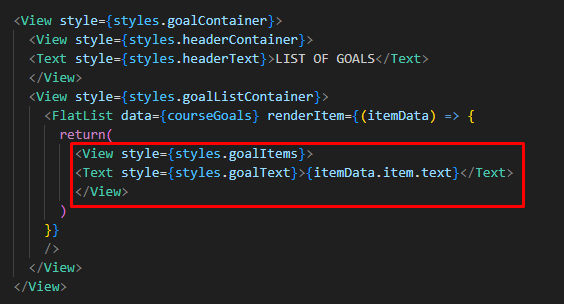
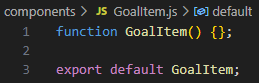
*Math.random().toString()*. FlatList will now automatically look for the keys associated with the values.

1. Now, since we’re dealing with objects with keys, we have to ensure that we are able to access the text property of the items. Access the text property for the *itemData.item* below.
2. Reload the app and add multiple items in your list with the same value. What can you observe?

# Splitting Components into Smaller Components

Notice that as we develop this basic application further, our code keeps getting larger and our components also keep getting larger. The same happens with applications developed in React Native.

1. Create a new folder called *components*. This is where we will store our components.
2. Create two files for the components, this will contain the actual JSX code and functionality related to our components. Name the files *GoalInput.js* and *GoalItem.js*.
3. In the GoalItem.js, create a function goal item and export as default.

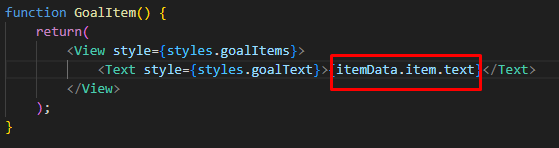


1. Within your GoalItem component, you want it to return the JSX code responsible for outputting a single goal item. Not the list of items, but a single item.

In our App.js, this is the section of the code responsible for that.

Cut it, and return it as your item in GoalItem component.

1. All the styles being referenced also have to be brought over. Create a stylesheet object in GoalItem.js and include all the associated styles. Cut them from your App.js. *Hint: It is good practice to keep your styles close to your JSX code.*
2. Clean up your code by removing unused imports. Import your GoalItem.js.
3. Now, down in your JSX code, where the code for the return item used to be, return a self-closing GoalItem component instead. Show your code so far.
4. Is this easier than the previous method performed? What benefits do you see to applying this technique?



Note: Running the code as it is right now will return an error. Finish the procedure first before trying to run the code.

# Utilizing Props

If you did not heed the advice from the note and ran your code anyway, you would have probably gotten an error. This error can be solved now with the use of *props*.

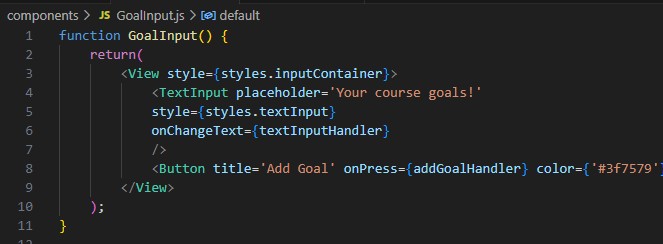
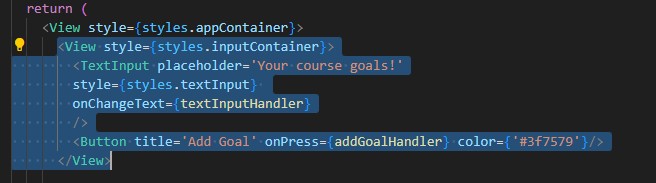
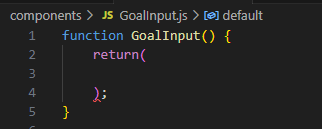
Looking at the snippet above, because we cut the code from App.js to our custom component; there is no way yet for the component to know what this itemData.item.text is, unlike beforehand. To solve this, we’ll use props:

1. Add *props* as a keyword to your GoalItem() function and inside the Text component’s display, where itemData.item.text is, replace with *props.text*.
2. Go back to your App.js, we will set a text prop to the GoalItem component being returned within the FlatList, this must be *text={itemData.item.text}*. Save, run and show your output. Try adding a goal. Is your output similar to the image below? Why do you think that is?
3. Solve by importing View and Text components. Save the code and try inputting a goal in the app. What output did you get and why do you think so?

# Working on the “Goal Input” Component

We will basically apply the same steps as we did earlier for the GoalItem component but this time for our other custom component, GoalInput.

1. Create the functional component for GoalInput. Include *export default GoalInput;* at the end of this function as well.

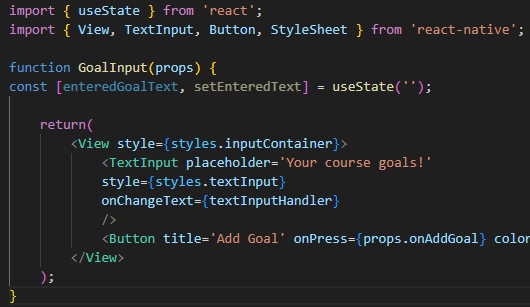
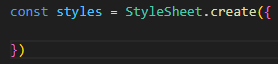


1. Identify from the App.js the JSX responsible for the text input of our goal. In this figure, it is the highlighted portion, we are including the view component for this because that is the container of our textInput component as well.

Your GoalInput.js should look similar to the figure below.

1. Addressing the same issues as encountered earlier, make sure to import the necessary components from react native by adding the following line of code at the top of your custom component.

Additionally, you want to include all the necessary styles to be close to your JSX file.



At this point, you may already realize some things will go wrong with the finished component file. What are the problems you expect and why?

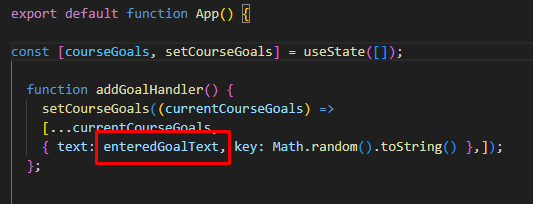
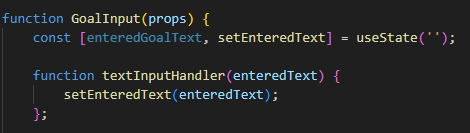
1. Similar to our earlier solution, one problem can be solved instantly through the utilization of *props*. Include a prop to your functional component like in the figure.

This will be used to talk to the parent component and activate upon press. Modify *onPress={addGoalHandler}* so that it utilizes the prop like in the figure below.

1. In your App component, input the custom component GoalInput.js. Then use the component in the same place where your previous code was in the JSX. Show your code screenshot here.
2. Now, add the prop that we created in GoalInput component. Remember that *onPress* we want the button to handle the textinput and add it to the list. This means that, we just have to replicate the passing of a function done previously. What function should be passed here?
3. A new issue arises now: how do we handle the user input? Remember that *const [enteredGoalText, setEnteredText] = useState('');* is the useState we’re using to handle the user input. Cut this from your App component and wrap it in your GoalInput function component. Your code should now be similar to the figure below.

Do not forget to import the useState.

You must also obtain the function meant to update the enteredGoalText.



These configurations make it so that the textInputHandler in your JSX file of GoalInput works.

1. However, enteredGoalText is also still needed as the data for the list to be displayed in your app component. But we moved the useState for enteredGoalText to a different component. So, we need to find a way to make this work.

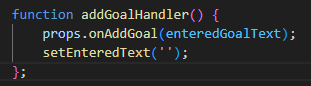
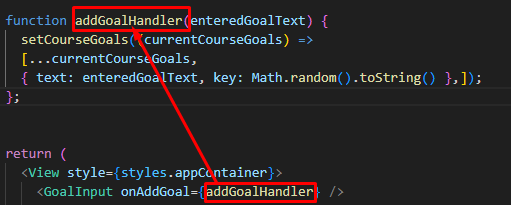
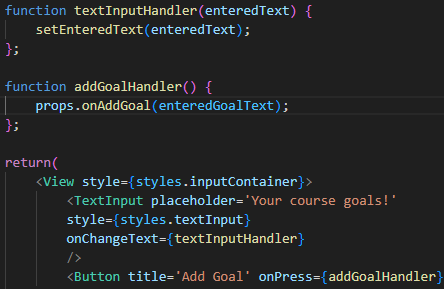
A simple solution is to add *enteredGoalText* as the parameter for addGoalHandler(), refer to the image below.

But now, no text is provided yet.

1. In your GoalInput.js, create a new function called *addGoalHandler*. Yes, the same name! Name clashing will not occur here because we’re in different components. In this new function, execute props.onAddGoal inside instead.

Next, pass the new addGoalHandler to the onPress={} prop as a function object.

Once onPress prop has the function as an object, we can pass enteredGoalText as the parameter of the prop. Your code should look similar to this figure.



To clarify:

In your app component, you have defined appGoalHandler which takes an enterGoalText as a parameter. This same function is passed in your GoalInput component. This means that, if onAddGoal passes a value to addGoalHandler function, it becomes the value for enteredGoalText.

In your GoalInput component, you made a function with the same name (which can even be named anything else). This function will execute the prop to pass enteredGoalText to the function in the app component. This is standard React.

Lastly, we can add a value prop to contain enteredGoalText then use setEnteredText(‘’) as an empty text to reset the text input field within the addGoalHandler.

***Your application should be working properly. If it isn’t you must troubleshoot and document your process.***

|  |
| --- |
|  |
| **6. Output** |
|  |
| **7. Supplementary Activity** |
| For this activity, you must demonstrate your ability to further break down your application into custom components.  - Show a screenshot of your application so far.  **ILO1: Utilize React Native components to make scrollviews and optimized lists.**   * Limit the size of your goal list’s container. Demonstrate the difference between scrollview and flatlist. Screenshots must be provided.   **ILO2: Decompose app component into multiple custom components.**   * Duplicate the activity by creating your own custom components to reflect how you further customized the application as required in the first activity. |
| **8. Conclusion** |
|  |
| **9. Assessment Rubric** |
|  |
| **10. Additional Screenshots** |
|  |