CST-391 Activity 5 Guide

Contents

[Part 1: React Music App Introduction 2](#_Toc127348515)

[One Parent Element in JSX 4](#_Toc127348516)

[Rending JSX content 4](#_Toc127348517)

[Custom Components 9](#_Toc127348518)

[Component Properties 10](#_Toc127348519)

[App Component 11](#_Toc127348520)

[Stopping Point #1 – Custom Components 12](#_Toc127348521)

[Mini App #1 - State Changer Demo 12](#_Toc127348522)

[Hooks History 14](#_Toc127348523)

[State Changer Demo Conclusion 16](#_Toc127348524)

[Part 2 Using State and Props in the Music Application 17](#_Toc127348525)

[More about Component State 18](#_Toc127348526)

[About the Map Function 19](#_Toc127348527)

[Using CSS FlexBox 21](#_Toc127348528)

[Stopping Point #2 – State and Props 22](#_Toc127348529)

# Part 1: React Music App Introduction

**Execution**

Execute this assignment according to the following guidelines:

1. Create new react app. Open the Terminal, navigate to the folder you wish to put the new application in, and type:

npx create-react-app music

1. Open the Visual Studio Code editor and select the new project folder.

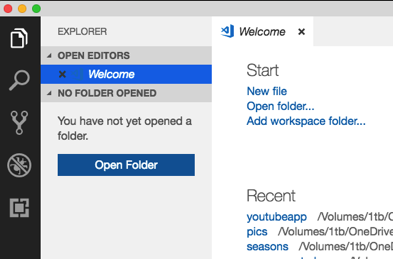
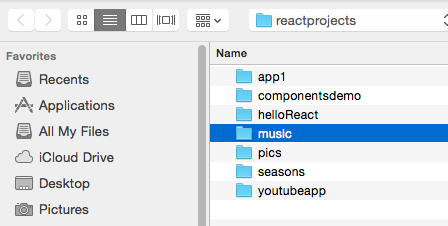
 

Figure 1 *Selecting the Project Folder*

1. Open the src folder and delete all of the default scripts and resources.

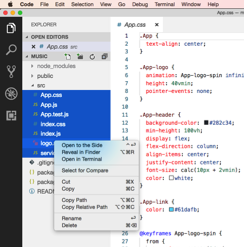


Figure 2 *Deleting the Default Project Contents*

1. Right-click on the **src** folder and choose **New File** to create new index.js file in src.

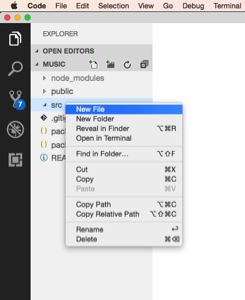


Figure 3 *Create a index.js File*

1. Create a React app as shown here.

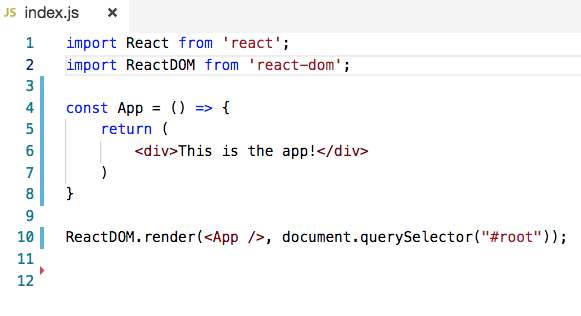


Figure 4 *Content of 'index.js'*

**Explanation**

**Imports**

The first two lines of the program import the React libraries necessary to run a React program. The first import statement will appear on nearly every file you create in a React application. The second import statement from react-dom is used only in this index file and applies only to web applications. React can be used in mobile phone applications that do not use the web DOM library.

**Const App**

The "App" function name is a commonly used starting point for many React applications. Its purpose is to display JSX code to the browser. JSX looks something like HTML but is really Javascript that is unique to React. For more information refer to "Introducing JSX," at <https://reactjs.org/docs/introducing-jsx.html>.

**ReactDOM.render**

The last statement of the program displays the entire application in a place on the webpage identified as #root. In the Visual Studio Code editor, open the public folder and find index.html. There will be a <div id="root" /> element on the page. This is where the entire React application will be rendered to the browser. It could be named something other than #root, but stick with the standard React naming convention. You will not need to open index.html again until we import a CSS library in a few pages.

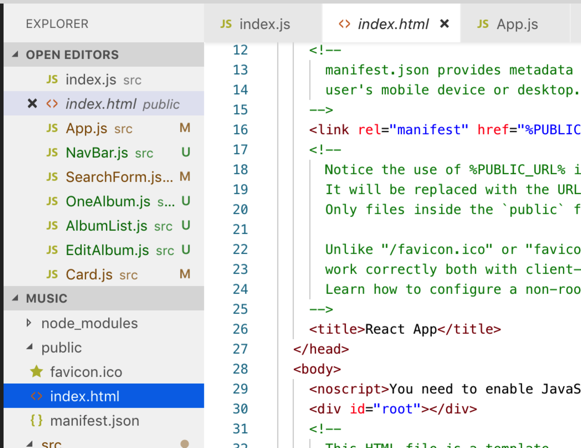


Figure 5 *index.html*

**Arrow Function Note**

Notice the strange notation for the App function using () =>. This is called an arrow function. It is a new way of writing the equivalent code that is shown below.

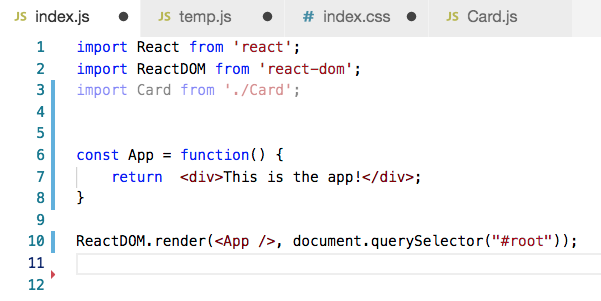


Figure 6 *index.js Updated with Arrow Function Syntax*

Refer to "Arrow Functions, The Basics," at <https://javascript.info/arrow-functions-basics>, to read more about arrow functions in JavaScript.

## One Parent Element in JSX

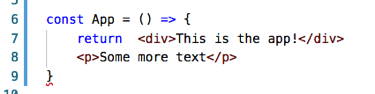


Figure 7 *Incorrect JSX*

Notice that the App content is returned within a <div> </div> section. JSX requires a single parent element. The following code shown in Figure 7 is not correct because it has two distinct parent elements.

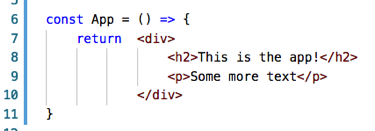


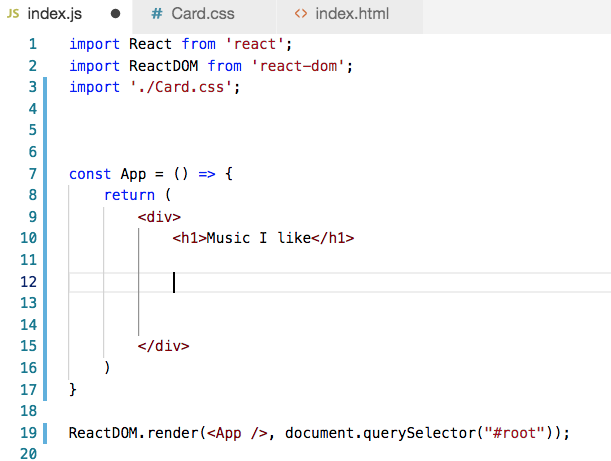
Figure 8 *Corrected JSX*

1. Launch the app in the npm web server from the console window by typing **npm start.**
2. You should see the app running in a new browser window at localhost:3000.

## Rending JSX content

Change the content of the app to include some album data.

1. Create a title and then leave some space.



Page title

Figure 9 *Updated 'index.js'*

1. Go to Bootstrap and copy the standard card HTML. Choose Bootstrap.com > Components > [Card](https://getbootstrap.com/docs/5.0/components/card/).

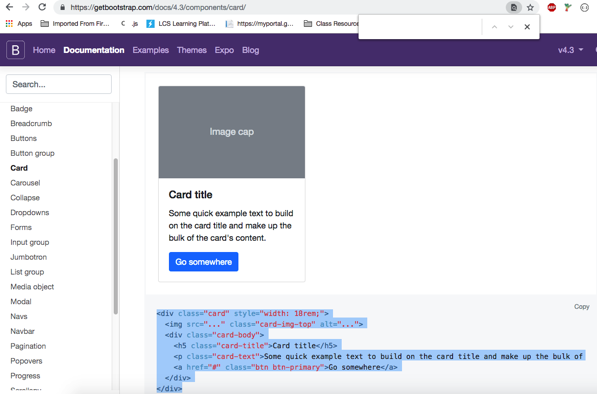


Figure 10 *Bootstrap Markup, Simple Card*

1. Paste the Bootstrap code into the App function.



HTML for Card from Bootstrap

Figure 11 *Updated App with Copied Bootstrap*

1. Fix the code to conform to JSX standards. The changes here are typical of an html to JSX conversion.
   1. Change all occurrences of class = to className=
   2. Terminate the <img tag with a /> instead of a simple > closing tag.



* 1. Change the style from



to this. Notice the double {{ }} curly brackets. A style is defined as a JSON object.



The updated code should look like this:



Figure 12 *Updated App with Bootstrap Markup Conforming to JSX*

Use [placeholder.com](https://placeholder.com/) to get a URL for the image source. Choose a 150 x150 placeholder and follow the instructions found on the website. Image placeholders are useful in web development because they allow developers to work on page layout before images are available.

1. Paste the placeholder link into the <img src="" > element as shown below.
2. Give the image an alt description attribute.



Image placeholder

Figure 12 *App Now has a Placeholder Image*

1. You should see something like this. The CSS is not yet included in the page, nor is Bootstrap included in the project, so it doesn't look very good yet. When we include Bootstrap, the Card markup we've added will take effect.

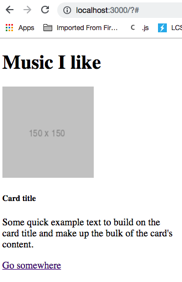


Figure 13 *Current View*

1. Copy the CSS link from [bootstrapcdn.com](https://www.bootstrapcdn.com/). This is the quickest way to import the bootstrap CSS library. To use a CDN (external) file like this, your computer must be connected to the internet to work properly. This lab is tested against version 5.x of the Bootstrap library, so version 5.x is recommended. There are also other ways to import CSS including using the npm install process as described in "Bootstrap 5," at <https://www.npmjs.com/package/bootstrap>.

**Note:** The screenshot shows version 4.x of Bootstrap. Update this link to be version 5.x of Bootstrap in all cases.

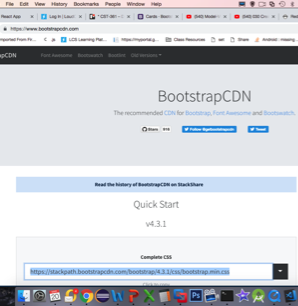


Figure 14 *Bootstrap CDN*

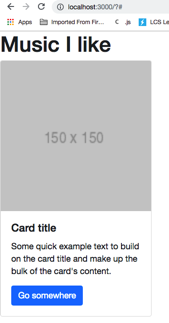
1. In the Index.html file, add a stylesheet link to the Bootstrap CDN link copied above.



Line #6 is new

Figure 14 *'index.html' with a Link to Include Bootstrap css.*

You should immediately notice a change in font, borders, and button colors on your page as Bootstrap markup can now be applied.



1. What if I want to show more than one card? The first idea is to (incorrectly) simply copy and paste the card <div> sections.



Figure 15 *Naïve Implementation with 3 Cards*

But this results in a very long list of code and still is not flexible. There is a better way called **custom components**.

## Custom Components

The solution to reducing the quantity of code on the page is to create each card as its own component.

1. Right-click on the src file and create a new file.
2. Name the new file **Card.js.**
3. Add the required React library import at the top of the page. Line #1 is required, while line #2 is not required even though it is shown here. Copy one of the Cards into Card.js as its JSX. Completed:

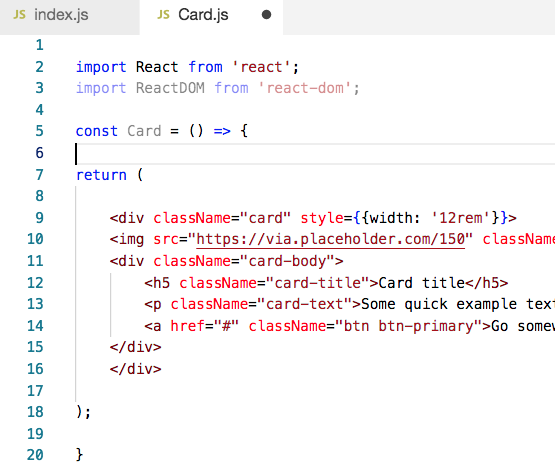


Figure 16 *Card.js*

1. Finish the page with **export default Card**; the export command is the other half of the import process. You should notice on most React components that the first few lines of text are importing methods and other components. Components share their code with this export command. Recall that we have seen examples of this in our Express server and in the Angular application.

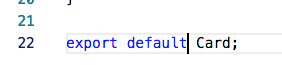


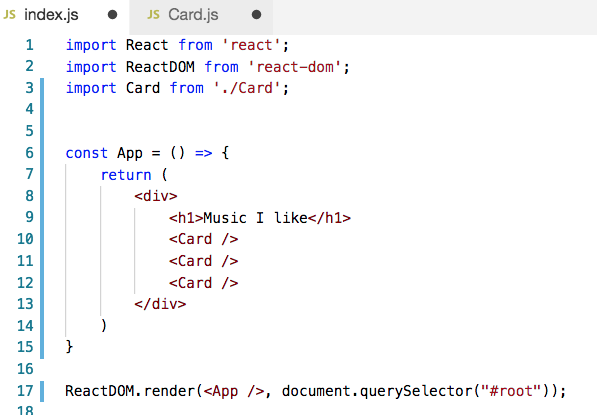
Figure 17 *Export the Card Method*

1. Return to the index.js page and import the Card component. The ./ prefix means that the Card.js file should be found in the same directory as index.js



Figure 18 *Import Card*

1. Replace all messy card codes with three <Card /> tags.



<Card/> x3

Figure 19 *Using the Card Component in index.js*

The resulting application should still look the same in the browser.

## Component Properties

The content of a component should be configurable. In React, each component has **properties** or **props**. In this example, the properties of a card should be the name on the button, the card title, the picture URL and the text of the card.

1. Go to Wikipedia for a music album and get some text for the album's description and an image. You can see in the example below text and an image link taken for the Beatles' Abbey Road Wikipedia page.
2. Modify the index.js file to include new properties for the first card.



Props include title, description, URL, and button. text.

Figure 20 *Custom attributes are added to Card. These become React props.*

Modify the contents of **Card**.js to include properties.

Add the (props) parameter in the function for Card (line 4)

Substitute {props.albumTitle} for the <h5> text, which is the card title.

Substitute {props.albumDescription} for the <p> text

Substitute {props.buttonText} inside the <button href tag.

(Change the anchor tag 'a' to 'button'.)

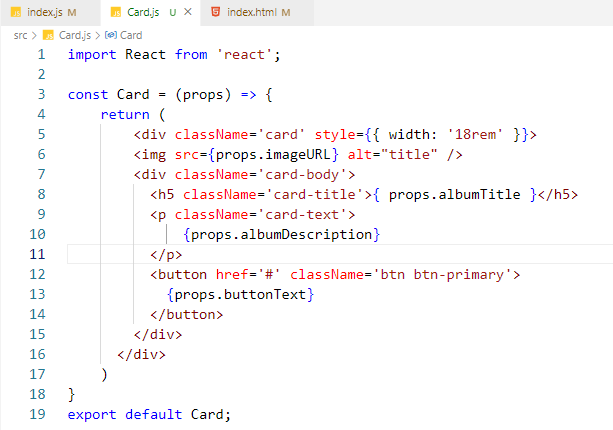


Figure 21 *Using Props in Card*

In the browser, you should see the first card now shows the properties that were passed to it. The following two cards shown here do not have any attributes for their props. Update the other two albums to have valid data.

## App Component

Just as we created a separate file for the Card component, we will now create a separate component for the **App** function. This will allow for two benefits:

The App code will be in its own script.

React applications are written in well-defined **components,** which will be explained in a few pages.

1. Inside the src folder create a new file called App.js
2. Copy most of the contents of index.js into the new App.js. The only code you do not copy is 'ReactDOM.render'. Replace this line with 'export default App'.

Index.js should import the new App component and call the render method to show App on the page, Index.js now looks like this:



Figure 22 *Updated index.js*

Many times during the development of a React App, you'll need to split components into new components as we have just done here.

You should be able to run the app as before.

## Stopping Point #1 – Custom Components

Save the project and summarize the progress you have made.

**Deliverables**

1. Take screenshots of the application you created. Be sure to show the various features that were illustrated in this lesson. Place the captured images in the Microsoft Word document provided titled "Activity Summary Page." Caption each picture to explain what is being shown.
2. Write a one-paragraph summary of the new features that have been added. Define new terminology that was used in the lesson.
3. Save this document to be turned in as directed by the instructor.

# Mini App #1 - State Changer Demo

It's time to start a new project to demonstrate new features. We will return to the music application in the next session.

In this application, we will demonstrate the use of state, props, and methods to update the state of an application. You should notice that a component can receive props values from its parent. The props values do not usually change once the component is rendered. The state values are set inside the component and do change on events such as keystrokes and mouse clicks.

**Execution**

Execute this assignment according to the following guidelines:

1. Create a new react app at the command prompt.
2. At the command prompt, type **create-react-app statechanger**.
3. Type **cd statechanger**.
4. Type **npm start**.
5. The new app should appear in the browser window.
6. Open Visual Studio Code and open the project folder for statechanger.
7. Delete all of the files in the src folder.
8. Add **index.js** to the src folder and add this code.

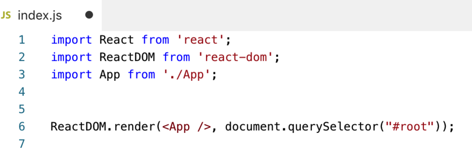


Figure 23 *index.js*

1. Create the file App.js in the src folder and add the following code.

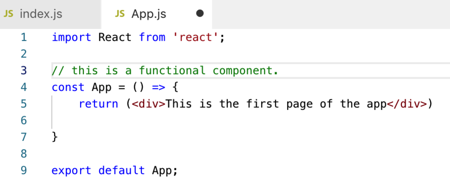


Figure 24 *App.js*

This is an example of a **functional component**. It consists of a single function that returns some JSX code. A React component should be the implementation of a single logical UI component.

**Explanation Pause**

Let's pause from the coding and consider these two examples of JavaScript functions.

**Functional Version of a Component**

const welcome = (props) => {  
 return <h1>Hello, {props.name}</h1>;  
}

**Welcome** is a function, which accepts props and returns a React element. In JavaScript, variables and functions are defined with **const**, **var** or the **let** keyword. For more details, read about the "Let Declaration," at <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>.

The parentheses around props are also optional so this is valid syntax.

const welcome = props => {  
 return <h1>Hello, {props.name}</h1>;  
}

The ES6 syntax of the arrow function is confusing, so here is the same component written with the ES5 syntax:

function Welcome(props) {  
 return <h1>Hello, {props.name}</h1>;  
}

Regardless of function syntax, the property object 'props' should be treated in the same way. An object, props contains properties specified by the parent component, information from the parent component for use by the child component. Logically, React component props are similar to function parameters in a conventional programming language. These properties should be treated as read-only by the child component. Because of this, 'props' cannot represent state changes in a component.

State changes are most often triggered by user actions (typing, mousing) and by the asynchronous arrival of API responses. To manage state, React components use hooks, named as such because they are hooks into the React state built into the library.

A React component using state hooks updates the user interface when the state changes. This state change then radiates out to the child components of the state-changing component.

## Hooks History

Hooks were introduced in React 16.8.0 (release date: Feb. 6, 2019). Before hooks, a React stateful component had to be a class. Functional components were always stateless child components. When the application state changes, functional components would be updated via props. As always, the state change radiates outward.

React hooks are called such because they allow a hook into React's state management system. React hooks essentially inverted state management in React because Hooks can only be used by functional components. This gave the previously stateless functional component access to React state, while denying the use of hooks to class components. Compared to state management from class components (via setState), hooks are better in every way. They are:

* Easier to learn
* Less error-prone
* Reusable as custom hooks

Our React activities have been updated to use hooks. This reflects how modern React is evolving and being used in the field.

However, old-school state controlling class components are fully supported by React and able to interact with functional components using hooks. You will see older React tutorials that use class components and setState. The official react tutorial is built on a stateful class component. It is recommended you review the "Optional – Tutorial: Intro to React," at <https://reactjs.org/tutorial/tutorial.html>.

Back to coding the State Changer Demo…

1. Create another component called **Counter.js** in the src folder and add the following code. Later in this document, we will review all the important features of this code.



Figure 25 *Counter.js*

1. Add css styling in a separate file called **Counter.css.** Add some CSS to make each element on the page look nicer. Notice that the css file is imported into Counter.js in Figure 25.

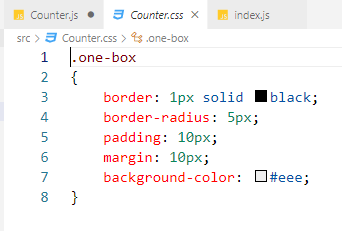


Figure 26 *Counter.css*

1. Add <Counter /> components to the App.js file.

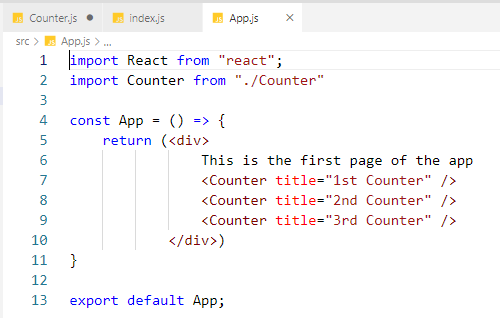


Figure 27 *Using Counter*

You should now see the application render in the browser. Notice that the props and the state both are used to display messages.

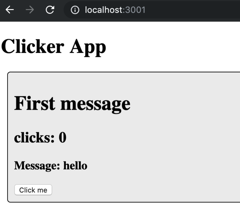


Figure 28 *Initial Browser View*

**Counter.js Explained**

Counter.js illustrates the use of two hooks with useState:

1. The useState hook takes a parameter that, on the first call, sets the initial state. This initial state can be any data type. The return value of useState is an array with two elements:
   1. The current state (same data type as the initializing parameter).
   2. A method to mutate (change) the current state.
   3. JavaScript array destructuring is used to create convenient local variables. You can name these local variables anything you want; however, a useful convention is to give the current state variable a meaningful name and prefix 'set' in front of the method that changes the state.
   4. Read about JavaScript array destructuring in the "Destructuring Assignment," at <https://javascript.info/destructuring-assignment>.
2. The first example of the useState Hook in Counter.js maintains the click count state:

const [clicks, setClicks] = useState(0);

1. In this example, the initial state of the click counter (0) is passed to the hook 'useState'. It returns the current state (clicks == 0) and a method to modify that state. Examine how 'setClicks' is used.
2. The next, independent, example of useState in Counter.js maintains the keystrokes in the message input control. Each keystroke results in a state change that remembers the key. This code also illustrates the creation of a controlled component, a common React pattern for data entry components.

const [message, setMessage] = useState(props.title);

1. Notice that you can create as many hooks as you need to manage the state in a component. If the states vary independently (as they do in Counter.js), each state should be handled by its own hook.
2. Updating the state in React sets off a chain of events that results in the component *and its* *child components* being rendered again. In Counter.js, for example, calling setClicks(clicks + 1) triggers a render of the component each time it is called. It is the same for calling setMessage. UI updates in React are always the result of state changes.

## State Changer Demo Conclusion

**Props**

You have seen the use of props. Props are defined by the parent and can be used to affect the looks of a child. The parent sets the props dynamically.

**State**

A class component may have a state. The state is usually changed by user action and asynchronous API results. Understanding that state change is how the UI updates in React is central to understanding React. A state change in React is a pulse that updates the React UI.

**Event Handers**

A class component may have methods. Methods respond to onClick and onChange events.

**Deliverables**

1. Take screenshots of the application you created. Be sure to show the various features that were illustrated in this lesson. Place the captured images in a Microsoft Word document. Caption each picture to explain what is being shown.
2. ZIP file.
   1. Delete the npm\_modules folder to remove the 40,000 or so files it contains. This folder can easily be recreated with the npm install command.
   2. Zip the project folder and include it as an attachment to the assignment.
3. Save these deliverables to be turned in as directed by the instructor.

# Part 2 Using State and Props in the Music Application

Let's return to the music application and apply what we just learned about props, state, and event handlers.

Here is the updated **App.js** file that applies the new lessons. We will then examine the important changes to the code.

Figure 29 *Updated App.js*

import React, { useState } from 'react';

import Card from './Card';

import './App.css';

const App = () => {

const [albumList, setAlbumList] = useState([

{

artistId: 0,

artist: 'The Beatles',

title: 'Yellow Submarine',

description:

'Yellow Submarine is the tenth studio album by English rock band the Beatles, released on 13 January 1969 in the United States and on 17 January 1969 in the United Kingdom.',

year: 1969,

image:

'https://upload.wikimedia.org/wikipedia/en/thumb/a/ac/TheBeatles-YellowSubmarinealbumcover.jpg/220px-TheBeatles-YellowSubmarinealbumcover.jpg',

},

{

artistId: 1,

artist: 'The Beatles',

title: 'Abbey Road',

description:

'Yellow Submarine is the tenth studio album by English rock band the Beatles, released on 13 January 1969 in the United States and on 17 January 1969 in the United Kingdom.',

year: 1969,

image:

'https://upload.wikimedia.org/wikipedia/en/thumb/4/42/Beatles\_-\_Abbey\_Road.jpg/220px-Beatles\_-\_Abbey\_Road.jpg',

},

{

artistId: 2,

artist: 'The Beatles',

title: 'Let It Be',

description:

"Let It Be is the twelfth and final studio album by the English rock band the Beatles. It was released on 8 May 1970, almost a month after the group's break-up.",

year: 1970,

image:

'https://upload.wikimedia.org/wikipedia/en/thumb/2/25/LetItBe.jpg/220px-LetItBe.jpg',

},

]);

const renderedList = () => {

return albumList.map((album) => {

return (

<Card

albumTitle={album.title}

albumDescription={album.description}

buttonText='OK'

imgURL={album.image}

/>

);

});

};

return <div className='container'>{renderedList()}</div>;

};

export default App;

Since App.js is presented in a text box, you should be able to select this code and replace the current code in App.js. Since the font is very small, viewing the code in a programming editor will be easier.

The first advantage is the ability of a component to have a **state** property. In the new App.js, the current state property is 'albumList'. We are not updating this state yet, so setAlbumList is unused. However, we are putting in place the basic wiring that will allow us to set the albumList from the results of an asynchronous API call.

## More about Component State

State is a JavaScript object that holds values used inside an app. We already have a props value for the component, so why make another value?

**State vs. Props**

In a React component, props are variables passed to it by its parent component. State, on the other hand, is still variable but directly initialized and managed by the component. Props are static. State is dynamic. The state values can be initialized by props.

For example, a parent component might include a child component:

**<ChildComponent />**

The parent can pass a **prop** by using this syntax:

**<ChildComponent color=green />**

State can be set to a default status as done in the current version of App.js, as well as set dynamically from outside sources such as databases, files, user input or web services. In the following steps, we are going to move the data of each card three times:

1. First, we are going give the component a state variable (complete in the example).
2. Second, we will move the data from the state variable to an external text file. (Activity 5)
3. Third, we will rely on a REST service from another program to provide our app with album data. (Activity 6)

In our version of the application, the only state variable we need to track right now is the **array of the albums** we want to show on the page. Our state variable is called **albumList**. As stated, setAlbumList is ready to modify the current state but is unused in the current App.js.

The use of albumList:

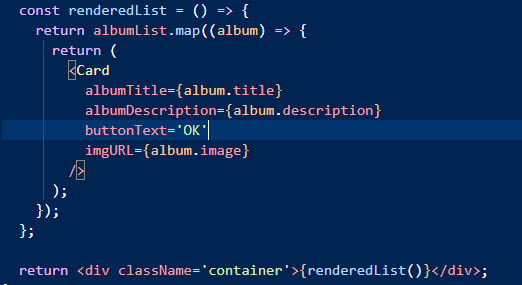


Figure 30 *RenderedList in App.js*

The function renderedList() is evoked inside the return statement of App.js. This function iterates through each element of albumList. The map function is a transformation function (more on the map function follows). In this use, it transforms the entries in albumList to JSX, which contains a Card component for each entry in albumList. Each one of the Card props (albumTitle, albumDescription, buttonText, and imgURL) is set according to a single album from the albumList.

## About the Map Function

Line 33 relies on the **map** function to generate a list of JSX controls. The map function may be new to you so let's do three exercises to show how the map function works.

1. In the browser console window, type the following function definition. You can ignore the "undefined" return value.

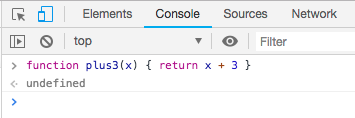


Figure 31 *Define Plus3*

1. Create an array of numbers as follows. The console will output the value of the array when you press enter.

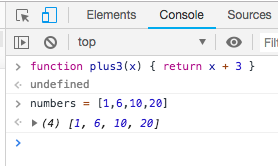


Figure 32 *Create an Array*

1. Run the map function on the numbers array using the plus3 function as the parameter.

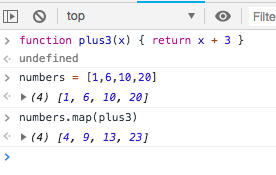
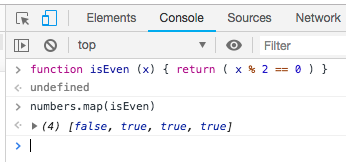


Figure 33 *'map'*

Notice that the map function returns a new array. Every member of the array is +3 more than the original array. The value of the original numbers array should not be affected by the map function.

1. For another example, make a new function called isEven. It will return a boolean value. True if the number is divisible by 2. False if it is not divisible by 2.
2. Run the function against the number array and see the results.

In the next example, we will generate some HTML code using each element in the numbers array.

Figure 34 *isEven*

1. Create a function that will surround the number with a list item tag. <li>
2. Run the function on the numbers array.

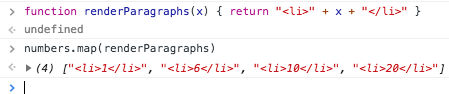


Figure 35 *Mapping to Html Elements*

Now go back to the previous example in the music app and look again at the map function being used. While slightly more complex than the map examples we just did, the code uses the same principles. Notice above how the renderedList function returns a list of <Card /> elements.

## Using CSS FlexBox

Right now, each card is stacked on top of the previous, which creates a long narrow column of cardsGo to <https://getbootstrap.com/docs/5.0/layout/grid/> to read about grid and layout settings in Bootstrap.

1. Notice how the class 'container' is called out in JSX. In JSX, 'class' is a keyword and so it cannot be used in this context.



Figure 36 *Applying css with a className*

1. Add the following code in new file called 'App.css'.

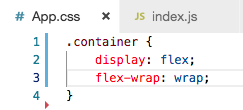


Figure 37 *App,css*

1. Import the css file into App.js.



Figure 38 *App.js*

The cards should now render on the page horizontally.

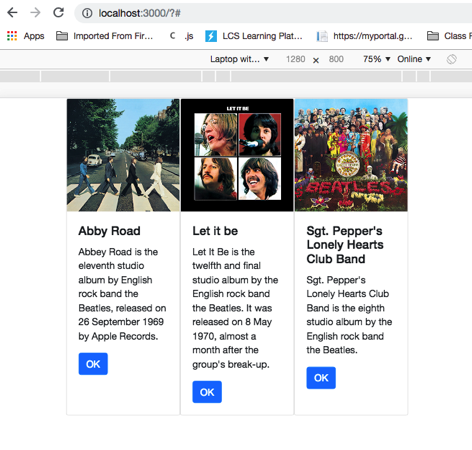


Figure 39 *Browser View*

1. Add some padding and margins to spread the cards.

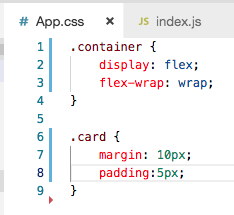


Figure 39 *Updated App.css*

## Stopping Point #2 – State and Props

Save the project and summarize the progress you have made. Submit all deliverables as directed by the instructor.

**Deliverables**

1. Take screenshots of the application you created. Be sure to show the various features that were illustrated in this lesson. Place the captured images in the Microsoft Word document provided titled "Activity Summary Page." Caption each picture to explain what is being shown.
2. Write a one-paragraph summary of the new features that have been added. Define new terminology that was used in the lesson.
3. Two ZIP files. Submit the mini app in one zip file. Submit the current progress of the music app in a second zip file.
   1. Delete the npm\_modules folder to remove the 40,000 or so files it contains. This folder can easily be recreated with the npm install command.
   2. Zip the two project folders and include them as attachments to the assignment.