CST-391 Activity 6 Guide

Contents

[Part 3 External Data Source 2](#_Toc126841975)

[Stopping Point #3 – External Data Sources 22](#_Toc126841976)

[Mini App #2 - Routing Application Demo 22](#_Toc126841977)

[Conclusion – Routing App Demo 32](#_Toc126841978)

[Part 4: Navigation Routing 32](#_Toc126841979)

[Stopping Point #4 - Navigation Routing 40](#_Toc126841980)

# Part 3 External Data Source

In this section of the activity, we will continue to work on the music application.

**External Data Source - JSON File**

Now let's get the album list data out of the state initialization and into its text file. This is the second stage of three stages in how we manage the album data. In the third stage, we will transition from using a text file to using the Express Music API you developed in Activity 1.

1. From the App.js file, cut the album (pasted to a different file in the next step) array and replace it with an empty array. See line 8. We chose an empty array as the initialization value instead of 0, null, or an empty string because we expect the data type of albumlist to eventually become an **array**.

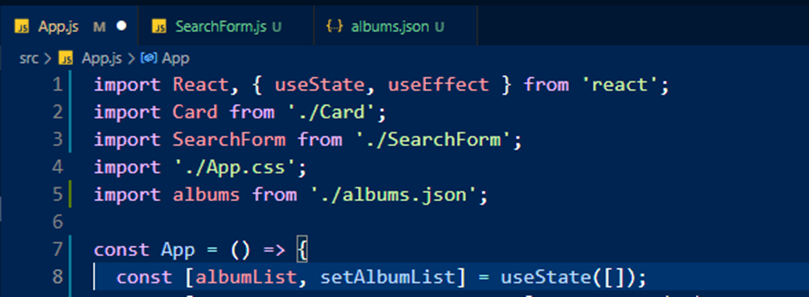
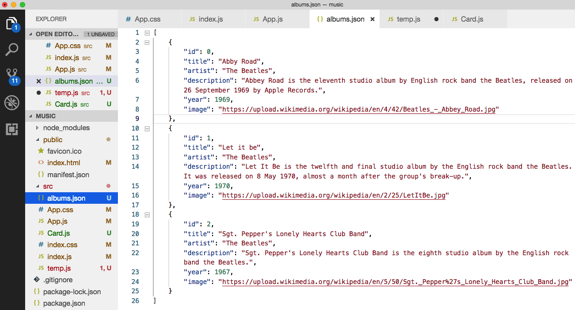


Figure 1 *App.js*

Now the program is much shorter.

1. Create a new file inside the src folder. Name the file **albums.json**

Paste the array of JSON objects into the albums.json file and save it.

Figure 2 *albums.js*

1. In the App.js file, import the JSON file as shown on line 5. This figure is also referred to in the next section.

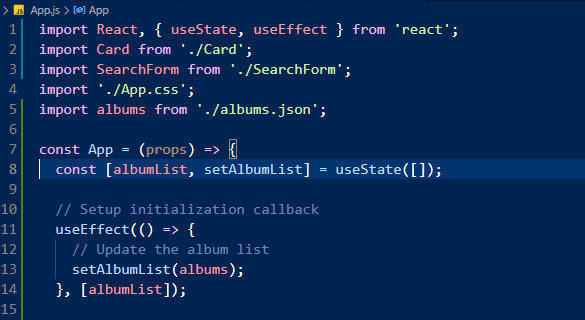


Figure 3 *App.js Including 'albums'*

The callback function registered by useEffect on line 11 above uses the special method called setAlbumList. Once the state variable has been initially defined, setAlbumList **is the only method you can use to update the albumList state variable**. Recall that the call to useState can take any data type as the initial state. Since we expect our data to be in an array, we set the initial state variable to an empty array. The return value of useState is an array with two elements: the current value of the state and the sole method used to update that value. The code shown on line 8 destructures that array giving those elements the names albumList and setAlbumList. These names follow React naming conventions.

The call to useEffect provides a 'side effect' callback function. 'Effect' callback functions are invoked during the life cycle when React deems it safe to call functions that will generate side effects (state changes). This is where a call to a web service API is typically placed. For now, we use the locally defined albums list built earlier to set a new albumList state. It will update and render the list. The following sections will help you understand useEffect by giving it context in the React Hook Lifecycle.

**React Hooks Lifecycle**

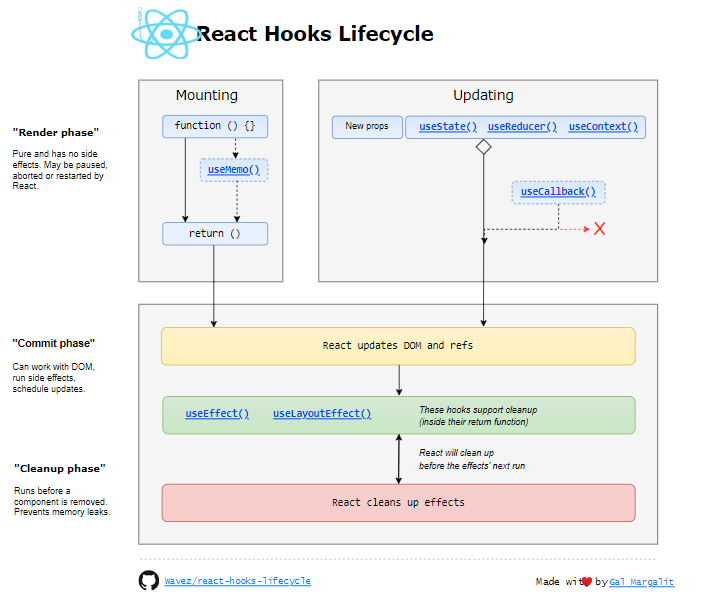


Figure 4 *React Hooks Lifecycle Diagram* ([Margalit](https://wavez.github.io/react-hooks-lifecycle/), from GitHub)

This diagram shows that useEffect() and useLayoutEffect() are the hooks an application programmer needs to know about to manage React's lifecycle. More, useLayoutEffect() is an edge-use case so we are going to concentrate on useEffect(). You can do a deep dive by reviewing the "Hooks API Reference," at <https://reactjs.org/docs/hooks-reference.html>.

useEffect() is cleverly simple. From the React documentation:

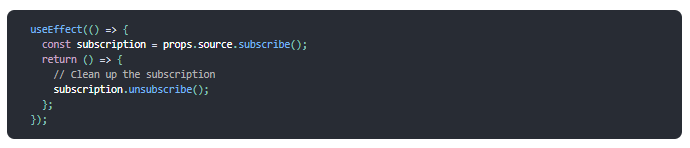


Figure 5 *'useEffect' Example*

With added comments:

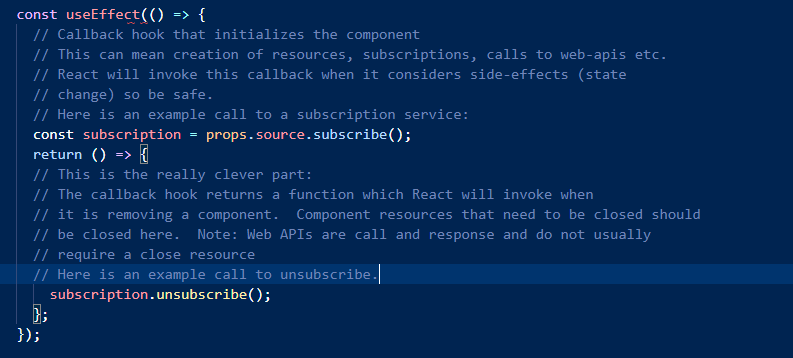


Figure 6 *'useEffect' with Comments*

One more variation, a second parameter to useEffect:

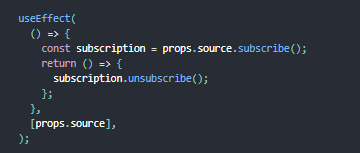


Figure 7 '*useEffect' with Second Parameter*

The second parameter creates a conditional firing of 'useEffect', invoking only when props.source changes. We use [albumList] for our application. If you edit the external file, the changes will show up in the browser. Add this code to your App component:

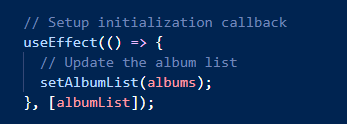


Figure 8 '*useEffect' for our Application*

**Key Values in a List**

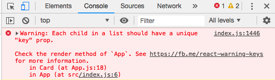
Notice that there is a warning in the browser console. React needs to have a unique number or string associated with each element in a list. It says we should look at line 18 in App.js to fix the problem.

Figure 9 *Error Seen in Console*

1. Open App.js and add the following change on line 18 to give each Card component a unique key value.

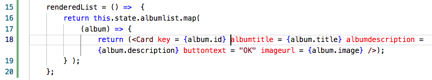


Figure 10 *Cards with Keys*

Fortunately, the JSON source file had an ID property for each element. Many data sources come from databases where it is common to have an ID number for each row in a table.

**Searching for Music**

Let's add a new component to the application that will allow us to search for some data.

1. Start by creating a new file called SearchForm.js and add the basic starting code.

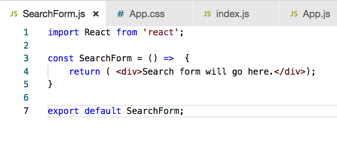


Figure 11 *SearchForm.js*

1. Add the <SearchForm /> tag to the App.js file.

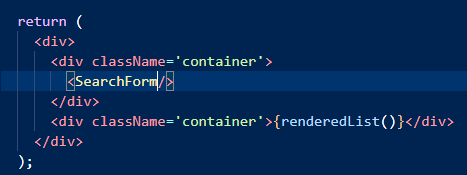


Figure 12 *Include the SearchForm*

You should see the placeholder text appear on the browser

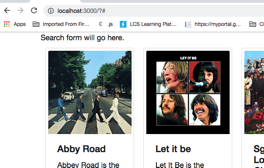


Figure 13 *In the Browser*

**Input Form Formatted by Bootstrap**

1. Go to Bootstrap's Component page. This is the source of our starter code. No need to copy the code from the blurry screenshot. Its purpose is to credit the source. I'll include the modified code in the next step.

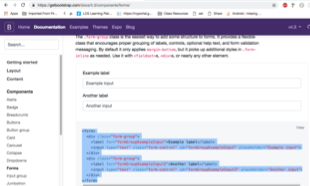


Figure 14 *Bootstrap Code*

1. Replace the current Search Form code with the code that follows. This makes several changes, which we will review in the following section.

Figure 15 *Updated SearchForm*

import React, { useState } from 'react';

const SearchForm = (props) => {

const [inputText, setInputText] = useState("");

const handleChangeInput = (event) => {

setInputText(event.target.value);

console.log(inputText);

};

const handleFormSubmit = (event) => {

event.preventDefault();

props.onSubmit(inputText);

};

return (

<div>

<form onSubmit={handleFormSubmit}>

<div className='form-group'>

<label htmlFor='search-term'>Search for</label>

<input

type='text'

className='form-control'

placeholder='Enter search term here'

onChange={handleChangeInput}

/>

</div>

</form>

</div>

);

};

export default SearchForm;

1. Compare the tags in the form returned by this component. They are modified to conform to JSX standards. All tags are properly closed, and 'class' is changed to 'className'. Other changes have been made to work with the music application. Learn the pattern so you too can copy code from Bootstrap.
2. A call to 'useState(""), creates a state variable "inputText" that will hold the value of the text input. The hook also returns setInputText, the only method for changing inputText:



1. A property for the <input> control called onChange.

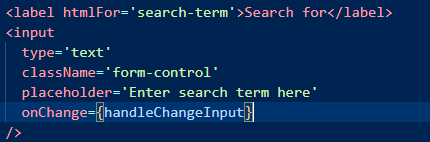


Figure 16 *onChange*

onChange is a reserved keyword in JavaScript to handle events. There is an extensive list of event handlers in Javascript. We will cover this later.

1. Now add a method in the class definition to handle the event.

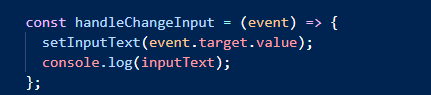


Figure 17 *handleChangeInput*

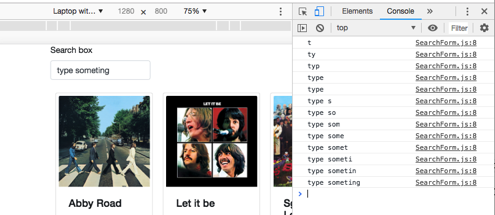


Figure 18 *Browser Console*

1. Finally, add an onSubmit event to the <form> element and create a handler method to display the state of the component.

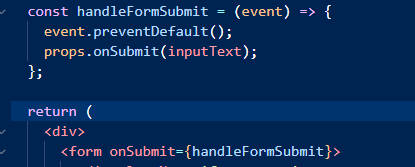


Figure 19 *onSubmit*

The default handler for form submission is for the browser to call a server with updated data. We are not talking to the server here so we're preventing the default handler. Via props, we call onSubmit. Since the function onSubmit is a member of the props object, we know that it must be specified by the parent component. In this way, the parent knows when the search component is submitted (the user hits the 'enter' key). We see that change next.

**New Search Handler**

The job of updateSearchResults is to receive a phrase and change the search Phrase state variable. The next question is, "how does the data from the SearchForm get to this handler?" The answer is through callbacks. A callback is a process of passing the name of a parent's method as a prop to a child. When the child calls the method, the code on the parent is executed.

1. Return to the App.js file and add a new state variable called **searchPhrase** and a new method called u**pdateSearchResults.**

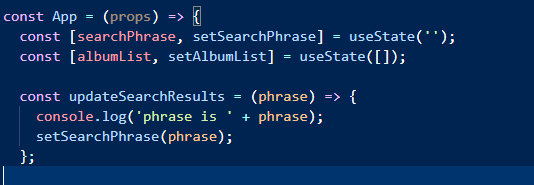


Figure 20 *updateSearchResult*

The next step is to pass updateSearchResults to the SearchForm component:



You should see the console log has some output when you run the app even though nothing changes on the album arrangement. This indicates that the communication path is complete.

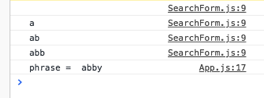


Figure 21 *Browser Console*

The searchPhrase variable will be used to filter the list of albums that are being displayed.

1. Update the renderedList method to return only those albums that include a substring match in the description.

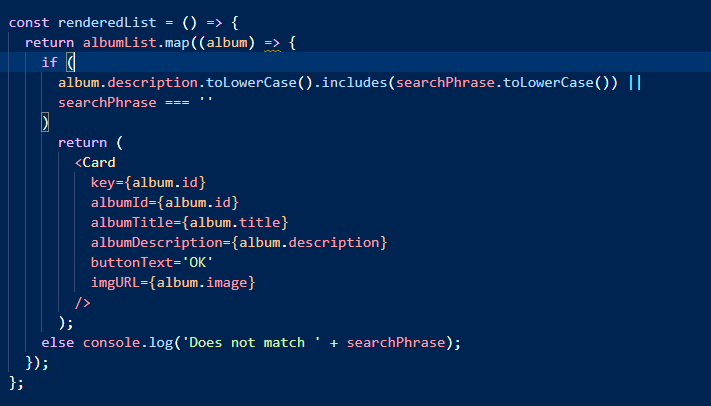


Figure 22 *Updated renderedList*

You should now see a filtered list of albums. In the console, there is a message for each album that does not match the search term. Callbacks are one of the more confusing parts of React so we will try to visually trace the flow of data on the next page.

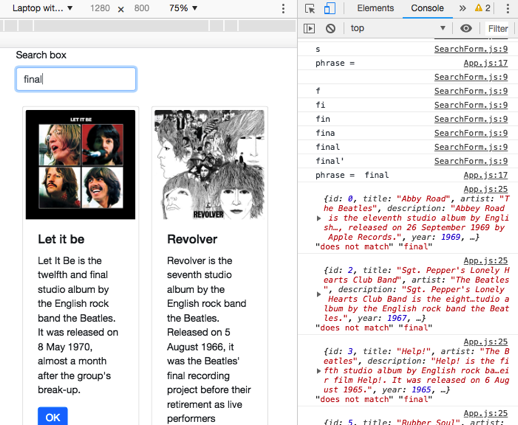


Figure 23 *Search Results*

**About HTML 5 Events**

Some HTML 5 events are listed here for your reference. Here script indicates a JavaScript function to be executed against that event.

|  |  |
| --- | --- |
| **Event Name** | **Description** |
| onblur | Triggers when the window loses focus |
| onchange | Triggers when an element changes |
| onclick | Triggers on a mouse click |
| oncontextmenu | Triggers when a context menu is triggered |
| ondblclick | Triggers on a mouse double-click |
| ondrag | Triggers when an element is dragged |
| ondragend | Triggers at the end of a drag operation |
| ondragenter | Triggers when an element has been dragged to a valid drop target |
| ondragleave | Triggers when an element is being dragged over a valid drop target |
| ondragover | Triggers at the start of a drag operation |
| ondragstart | Triggers at the start of a drag operation |
| ondrop | Triggers when dragged element is being dropped |
| onerror | Triggers when an error occurs |
| onfocus | Triggers when the window gets focus |
| onformchange | Triggers when a form changes |
| onforminput | Triggers when a form gets user input |
| onhaschange | Triggers when the document has changed |
| oninput | Triggers when an element gets user input |
| oninvalid | Triggers when an element is invalid |
| onkeydown | Triggers when a key is pressed |
| onkeypress | Triggers when a key is pressed and released |
| onkeyup | Triggers when a key is released |
| onload | Triggers when the document loads |
| onloadeddata | Triggers when media data is loaded |
| onmousedown | Triggers when a mouse button is pressed |
| onmousemove | Triggers when the mouse pointer moves |
| onmouseout | Triggers when the mouse pointer moves out of an element |
| onmouseover | Triggers when the mouse pointer moves over an element |
| onmouseup | Triggers when a mouse button is released |
| onmousewheel | Triggers when the mouse wheel is being rotated |
| onpagehide | Triggers when the window is hidden |
| onpageshow | Triggers when the window becomes visible |
| onpopstate | Triggers when the window's history changes |
| onprogress | Triggers when the browser is fetching the media data |
| onredo | Triggers when the document performs a redo |
| onresize | Triggers when the window is resized |
| onscroll | Triggers when an element's scrollbar is being scrolled |
| onselect | Triggers when an element is selected |
| onstorage | Triggers when a document loads |
| onsubmit | Triggers when a form is submitted |
| onsuspend | Triggers when the browser that has been fetching media data is stopped before the entire media file is fetched |
| onundo | Triggers when a document performs an undo |
| onunload | Triggers when the user leaves the document |

**Component Hierarchy**

Consider the design of the music application. It currently has three components: **App**, **SearchForm**, and **Card**. App is considered the parent component since the Cards and SearchForm are placed within the App page.

Figure 24 *Current App Hierarchy*

Later, we may want to create another component whose job it is to organize the Card components. If we do this, then another layer will be added to the hierarchy as pictured below.

Search Form

AlbumContainer

Card

Card

Card

Card

Card

Card

Figure 25 *AlbumContainer*

Figure 26 *Updated App Hierarchy*

Still later in the process, we will want to display the song titles on each music album. The hierarchy can grow very large for complex applications. Figure 27 *Card*

Search Form

AlbumContainer

Card

Song

Song

Song

Card

Song

Song

Song

Figure 28 *Component Hierarchy for a Single Page Music Application*

**Passing Values Upward**

The next phase of the application is to process the input from the SearchForm and filter out albums based on the user's input. We will need to pass the *inputtext* state value from SearchForm up to App and display only the albums that contain the key words provided.

**Props vs State**

**Event handlers and callbacks** pass state variables up

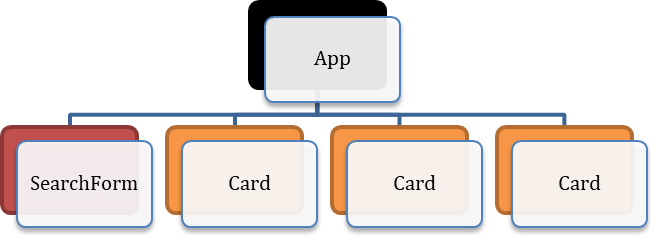
**Props** pass data down the hierarchy

**App**

**SearchForm**

At this point in the lessons, it is a good idea to distinguish between **props** and **state**. The purpose of props is to pass information from the parent to the child component. The child component displays data based on those props. In the reverse process, the child will pass data up the hierarchy through a process called **callback functions**, which we will do in the next few pages.

FFigure 29 *Information Flow*



Sends the search word "abbey"

Filters the albumslist array to show only matching albums

Figure 30 *SearchForm Information Flow*

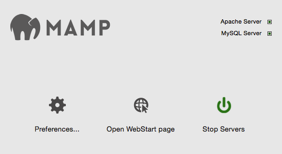
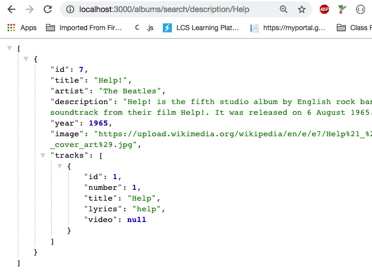
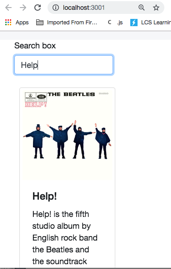
**State Rules**

1. Changing the state values are done through special methods returned by useState().
2. Updating state values will cause the component to be automatically re-rendered.

**External Data Source – REST Service**

Now it is time to move the JSON text file out of the application and served from an external REST service. Here is a diagram that shows the flow of data that we will use. Essentially there are three servers in use:

* Server 1 handles the React application.
* Server 2 is the REST server running express.
* Server 3 is the MySQL server, which is part of MAMP.



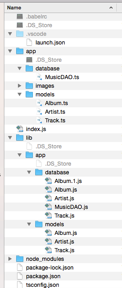
#4 Here is a JSON formatted string

Step 1 – Please give me all albums with "help " in the description.

#2 – SELECT \* FROM ALBUMS WHERE DESC = %Help%

#3 Here are some rows of data. It is your job to format them.

Figure 31 *Completed App Data Flow*

1. Ensure the express application is working.

Here is the folder structure of the rest service music app that was developed earlier in this course.

Start the application and run some queries in the browser's URL.

Figure 32 *Rest Service Music App* *Folder Structure*

Here is a test of the Express app:

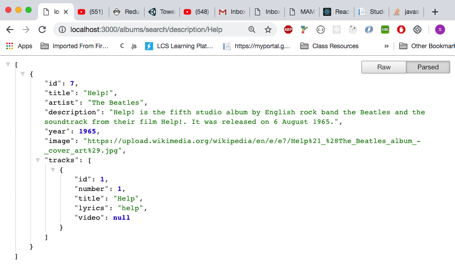


Figure 33 *Express Server Response*

Notice that the rest service is providing JSON formatted data as a result of a search query indicated in the browser's URL.

Confirm that you have data in your MySQL tables. You should have several albums and tracks installed from previous assignments in this course.

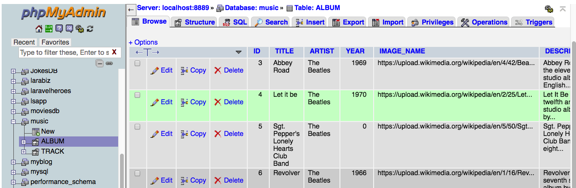


Figure 34 *Data Tables*

**Axios**

[Axios](https://github.com/axios/axios) is a popular library whose job it is to fetch data from JSON formatted rest services. There are other choices for fetching data including these names.

**Data Fetching** (Learn more about each in the topic Resource)

* **Apollo:** Easy to set up and use GraphQL client.
* **Axios:** Promise-based HTTP client for the browser and node.js.
* **Relay Modern** - A JavaScript framework for building data-driven React applications.
* **Request:** Simplified HTTP request client.
* **Superagent:** A lightweight "isomorphic" library for AJAX requests.

You may find tutorials online that use the **fetch** function. Fetch is built into many modern browsers but not all.

Table 1*. Axios Compared with Fetch*

|  |  |
| --- | --- |
| Axios | Fetch |
| Required library installation. Works with older browser clients. | Native to modern browsers. Will fail on older browsers. |
| One-step process to fetch data. | Two-step process to fetch data and transpose it to JSON format. |
| Uses try { } catch { } to correctly notify fetch failure. | Does not trigger exceptions even if data fetch fails. |

1. Install the Axios library in the terminal.

Stop the current web server session with control-C and type **npm install Axios.**

1. Create a new file named dataSource.js and place the following code inside.

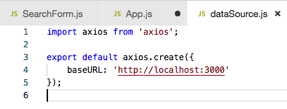


Figure 35 *dataSource.js*

1. App.js is modified:

Text

Description automatically generated

Figure 36 *Updated App.js*

* 1. The static dataSource is no longer needed, line 5
  2. useEffect, line 20, now calls loadAlbums to get album information from the server.
     1. The second parameter [refresh] can be used to signal a need to call useEffect again to refresh the data. It is not wired to change yet, but it is important. Otherwise, useEffect will be called endlessly.

**About Async and Await**

1. Async/Await is used to fetch data from slow external sources. "Slow" means anything longer than instantaneous.
2. Async/await is a new JavaScript ES6 way to write asynchronous code. Asynchronous means that a function can be started and code execution continues down the page even if the function has not completed its task. This is necessary to prevent an application from freezing up while waiting for something like data fetching to finish.
3. Previous options for asynchronous code are callbacks and promises. Many tutorials will show an older style of coding that you should be aware of.
4. Async/Await is non-blocking, which means that other processes will continue to run while the data fetch is being done. The application will not freeze the execution thread.
5. Async/Await makes asynchronous code look and behave a little more like synchronous code, which makes coding more readable.

**Promises vs Async**

Before async and await, JavaScript executed data fetch routines using a "then" statement. Here is an example of the old and the new syntax.



Figure 37 *Original Promise Syntax*

Assuming a function getJSON that returns a promise, and that promise resolves with some JSON object, we just want to call it and log that JSON, then return a string "done".

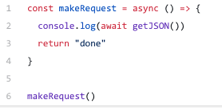


Figure 38 *Updated with Asynch and Await*

There are a few differences here:

1. Our function has the keyword async before it. The await keyword can only be used inside functions defined with async.
2. Await getJSON() means that the console.log call will wait until getJSON() promise resolves and print it value.
3. We can't use **Await** in the top level of our code since that is not inside an **Async** function.

**Callback Hell**

Before Async/Await, a multi-step data-fetch exercise could get nested very deep very quickly as the following example shows. Such a situation has come to be known as "callback hell" for a good reason. It is very confusing to read and error prone.

const verifyUser = function(username, password, callback){

dataBase.verifyUser(username, password, (error, userInfo) => {

if (error) {

callback(error)

}else{

dataBase.getRoles(username, (error, roles) => {

if (error){

callback(error)

}else {

dataBase.logAccess(username, (error) => {

if (error){

callback(error);

}else{

callback(null, userInfo, roles);

}

})

}

})

}

})

};

Figure 39 *Callback Hell*

**Run the App with a Rest Service**

Run the new version of the app in a separate terminal window.

**npm run start**

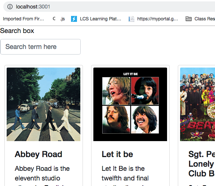


Figure 40 *Browser View*

The app should open in the browser, probably running on port 3001.

**Search Function**

Run the app and perform a search. You should see some debugging data being printed in the console. Albums are fetched and then filtered locally.

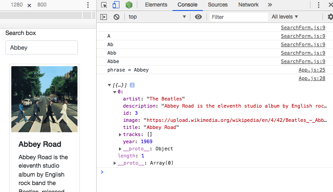


Figure 41 *Browser Console*

## Stopping Point #3 – External Data Sources

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 provided Microsoft Word document 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 documentation to be turned in as directed by the instructor.

# Mini App #2 - Routing Application Demo

In this application, we will demonstrate the use of a routing library in React. We will return to the music application in a few pages.

**What is a route?**

A web application route is a connection between the browser's URL and the contents of the web page. For example, two of the routes we will create in this tutorial are localhost\about and localhost\contact. Both of these addresses will show different pages.

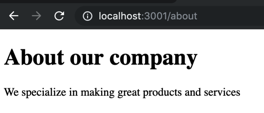
 

Figure 42 *App Preview*

**Start a New Application**

1. Start a new project
   1. Open the Terminal.
   2. Navigate to the folder you wish to put the application.
   3. Type npx **create-react-app router.**
2. Add a new library called react-router-dom.
   1. Type **cd router.**
   2. Type **npm install react-router-dom –save.**
   3. Type **npm start.**
3. Open the project folder using Visual Studio Code.
4. Delete all of the files from the src folder.
5. Create index.js and add the following code.

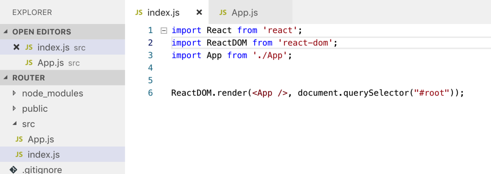


Figure 43 *index.js*

1. Create App.js and add the following code.



Figure 44 *App.js*

**Four Components**

Next we will create four basic components that will represent three different pages of our website.

1. Create a "Contact Us" page with some standard message about your company.

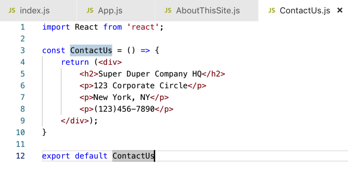


Figure 45 *Contact Us*

1. Create an "About Us" page with some more information.



Figure 46 *About This Site*

1. Create a LoginPage component with the following code. The login for our application just requires a button click to complete the login. Nevertheless, there is a lot going on in this code. Some of this will be understood only when you have studied related components in this application:

Text

Description automatically generated

Figure 47 *Login*

* 1. Starting on Line 5, handleLogin calls props.onClick. Since 'props' is involved, you know that 'props.onClick' is a method that was passed in by the parent component of LoginPage. This technique is standard in React: Since both user actions and API calls employ callback methods to signal state changes, the component that owns the state should also own the callback method. Given the parameters used on line 8, you know that 'from' and 'navigate' must be important to state management. We will see that when we review the parent of LoginPage.
  2. On lines 12 and 13, hooks are used (remember that React hooks always begin with 'use') to get navigate and location objects. React derives these objects from the internal router state. This is a great example of how hooks simplify application code.
     1. Navigate: A ready-to-use method allowing the code to use routing to navigate to the next application page. It is not used directly. It is passed to its parent on line 8. The parent cannot call the hook useNavigate(). In that context, it returns the error: "useNavigate() may be used only in the context of a <Router> component."

Question: Why is this?

(Think about it first.)

Answer: React hooks (the 'use' methods) are derived from the internal React application state. Providing the navigate() method requires a router state in the overall React application state. Be careful: This is missing if the component is not activated via a route. This happens if the component is included via a tag: <LoginPage /> in this case. In our application, LoginPage is only activated via a route.

* 1. On line 15, location is used to get location state. We are interested in where the 'from' path. Routed from the main menu, from.pathname is undefined. This is tamed to '/'. Learn about the null propagation operator to understand line 16 completely. In the next step, we will learn of the second possible 'from', the PrivateRoute component.

1. PrivateRoute is used to protect routes so that unauthorized users (not logged in) are prevented from reaching protected (private) routes. If authorized, props.children (line 11) contains the user requested route and navigation continues. When the user is not authorized, navigation to /login occurs (line 13). The requested route is saved in the state property so that after successful login, the user is routed to the originally requested component. Create a PrivateRoute component with the following code:

Text

Description automatically generated

Figure 48 *PrivateRoute*

1. Next, we will look at App.js, which will tie the child components that we have examined together. The first part of App.js:

Text

Description automatically generated

Figure 49 *App.js Part 1*

* 1. The first part of this file contains the usual import statements for components that we are going to use. Some of these haven't been written yet, so comment out those.

Of interest is the call to useState which sets the current logged-in state to false.

handleLogin is the method we saw in use by the LoginPage component. It calls setIsLoggedIn to set the logged-in state to true. It then uses navigate to go to the component the user originally requested.

1. The next part of App.js is entirely the JSX markup that makes this component:

Text

Description automatically generated

Figure 50 *App.js Part 2*

The code has been surrounded by an empty element <></>. Currently, this doesn't serve a purpose, but during development it allows you to place test components in the JSX code. Remember that we must return one top-level element the empty element is an easy way to accomplish this.

Otherwise, most of the markup mostly defines the routing for our application. At the top level is <BrowserRouter>. BrowserRouter uses HTML5 history API to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components.

The code shown uses the React Router v6, the most current version of the React router. When you are studying this, watch out for Router v5 tutorials. The v5 routing code looks very similar to v6, but v6 had many breaking changes. Refer to the "v6 Tutorial," at <https://reactrouter.com/en/6.8.0/start/overview>.

Of interest is the route to LoginPage on line 47. It defines an onClick prop that is set to the handleLogin method defined on line 17 of this component.

The other point of interest is PrivateRoute used on lines 33 and 41. The two components protected by PrivateRoute are AboutThisSite and ContactUs. Any area of the application that needs to be forbidden unless the user is authenticated can be nested inside a PrivateRoute. The component nested inside is accessed by props.children in PrivateRoute.

1. The NavBar component was copied from an example on the Bootstrap site. The HTML needs to be tightened up for JSX. All components need end tags and class (a JavaScript keyword) needs to become className.

Text

Description automatically generated

Figure 51 *NavBar*

1. The other components in this application are simple components that allow us to test routing, navigation, and private routes. See the code below:

Text

Description automatically generated

Figure 52 *AboutThisSite.js*

Text

Description automatically generated

Figure 53 *ContactUs.js*

Text

Description automatically generated

Figure 54 *User.js*

## Conclusion – Routing App Demo

The library React router provides new functionality to the React program, allowing you to specify the actions of the app based on the URL in the browser.

**Deliverables**

1. Take screenshots of the application being run. Place the images in a Microsoft Word document and caption each image. You should demonstrate each page in the application.
2. Write a paragraph explaining the new terminology and new features that you just demonstrated.
3. Delete the folder node\_modules from your project to remove tens of thousands of files! This folder can be recreated with npm install.
4. Create a zip file of the project folder and submit it as an attachment.
5. Save these deliverables to be turned in as directed by the instructor.

# Part 4: Navigation Routing

Let's return to the music application. Since we are adding application routing, add the routing library to your application:

**npm install react-router-dom**

**Refactor renderedList Method**

There is room for improvement in the design of the App component of the music application. Currently, there is a helper function called renderedList which generates a list of <Card /> components as shown here.

One of the best practices in a React app is to reduce a web page to a collection of components. The renderedList method is a red flag that should be shouting, "This is a good place to create a separate component!"

We are going to add a new container component called AlbumList whose job is to manage a collection or list of Card elements. We are also going to create a SearchAlbum component that will contain the SearchForm and AlbumList components. Finally, we will modify App.js to use SearchAlbum. App.js is still the stateful component in the refactored application, but much of the details are moved into child components. As we modify the code, we will temporarily refer to objects that are not yet defined. Work through this tutorial and all shall be defined!

1. Create the AlbumList.js file with the following code.

Text

Description automatically generated

Figure 55 *AlbumList.js*

1. Next, create a SearchAlbum.js file with the following code. Notice that this component is the direct parent of AlbumList:

Text

Description automatically generated

Figure 56 *SearchAlbum.js*

1. The modifications made to App.js will take advantage of the new components.

renderedList is no longer a function but is simplified to a filtered list in App.js:

Text

Description automatically generated

Figure 57 *renderedList*

This simplification of App.js paves the way to add additional functionality:

1. Application routing
2. The editing of single albums

We will add these now.

**Application Routing**

We are going to apply the knowledge we just learned about routing to the music application. App.js will define our routes. Modify App.js to return the following:

Text

Description automatically generated

Figure 58 *App.js*

This code refers to variables and components that we have not defined yet. We will start from the top and define those missing pieces in the following sections by line.

*Line 65*

1. Import the following for the Router components we are using:



BrowserRouter: The wrapper component for router components.

Routes: The immediate parent component of an individual Route.

Route: Defines an individual Route in the application.

*Line 66*

1. The use of the NavBar component defined in NavBar.js. This code is derived from a Bootstrap example:

Text

Description automatically generated

Figure 59 *NavBar*

*Line 75*

'updateSingleAlbum' is passed as a prop to SearchAlbum. In response to a click, it allows the editing of an album. It is defined inside of App.js:

Text

Description automatically generated

Figure 60 *'updateSingleAlbum'*

This code depends on the setCurrentlySelectedAlbumId hook defined here:



It uses navigate on the path /show/:albumId. This path will route to OneAlbum, the final component we need to define.

*Line 79*

Line 79 refers to the NewAlbum component. For activity 6, this component is a stub. We will develop it in Activity 7:

Text

Description automatically generated

Figure 61 *NewAlbum.js*

*Line 83*

As stated, this line refers to the OneAlbum component. Here is its code:

Text

Description automatically generated

Figure 62 *OneAlbum*

*App.js Complete*

We have made several edits to App.js. App.js remains the manager of state in our application, but some of its functionality has been moved to other components and new behaviors have been added to our application via new components. For reference, the completed App.js file in three screenshots:

Text

Description automatically generated

Figure 63 *App.js Complete Part 1 of 3*

Text

Description automatically generated

Figure 64 *App.js Complete Part 2 of 3*

Text

Description automatically generated

Figure 65 *App.js Complete Part 3 of 3*

## Stopping Point #4 - Navigation Routing

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 provided Microsoft Word document 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 one zip file of the mini application and another zip file of the current state of the music application.
   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.