

# Project 5 - Fetch

2024/6/26

## 20 Points Possible

Attempt 1



2024/6/26

**NEXT UP: Review Feedback**

Attempt 1 Score:

N/A



Add Comment

### Unlimited Attempts Allowed


2024/6/15 to 2024/6/30

#### Details

**Points:** 20 points**Deliverables:** Upload [Name]BookstoreFetch.war to <https://cs5244.cs.vt.edu:8443/ArchiveUpload/> (<https://cs5244.cs.vt.edu:8443/ArchiveUpload/>)**Resources:**

- **CorsFilter.java** (<https://canvas.vt.edu/courses/192345/files/34010112/download>)   
([https://canvas.vt.edu/courses/192345/files/34010112/download?download\\_frd=1](https://canvas.vt.edu/courses/192345/files/34010112/download?download_frd=1))

## Overview

In this project, you will combine Project 3 (React) with Project 4 (Rest). The result will be a home page and all the category pages that dynamically update depending on the selected category. To do this, we are going to use the "Axios" library ([Axios API](https://axios-http.com/docs/intro)  <https://axios-http.com/docs/intro>) to retrieve data from the database using the REST APIs created in Project 4.

## Setup your Server Project

Duplicate your server project [Name]BookstoreRest on your machine, and name the new project [Name]BookstoreFetch.

Open your new project in IntelliJ and find the settings.gradle file. Update the project name to end with **Fetch** instead of **Rest**. Then click the elephant to reload Gradle. Open your Tomcat run configuration, and go to the Deployment tab. Delete the war file asset that no longer works. Press the Fix button, and change the application context to **/[Name]BookstoreFetch**. Run the Tomcat configuration. The REST API should work as in Project 4. Test some complex APIs. For example: [http://localhost:8080/\[Name\]BookstoreFetch/api/categories/name/Classics/suggested-books?limit=2](http://localhost:8080/[Name]BookstoreFetch/api/categories/name/Classics/suggested-books?limit=2).



## Setup your Client Project

Duplicate your client project **[name]-bookstore-react-client** on your machine, and name the new project **[name]-bookstore-fetch-client**. Open the new project in VS Code or IntelliJ. Edit the package.json file so that the name property is **[name]-bookstore-fetch-client**. Edit the vite.config.ts file so that the base property is **/[Name]BookstoreFetch**. Finally, run the client project in the terminal with "npm ci". Ensure that it works like Project 3 running at [http://localhost:5173/\[Name\]BookstoreFetch](http://localhost:5173/[Name]BookstoreFetch).

## Use Axios to Fetch the Categories

In this project, we will use some important features of React as well as the axios library. There are many tools you can use to make HTTP requests in React applications, including the browser's built-in **Fetch API** ([https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API/Using\\_Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)). However, we are going to use the **Axios** library. Axios is a lightweight package that provides a simple syntax for making network calls, working with JSON data, and handling errors. Go to your client folder on the terminal and install Axios by running **npm install axios**.

React provides a hook called **useEffect** to perform side effects, such as API calls that are not directly triggered by a user event or action. In this project, we will make an Axios GET request to the REST API we created in Project 4, using the **useEffect** hook to fetch the categories from our database, and then passing the categories down to the components that need to use them.

Multiple components need to use the categories, such as **HeaderDropdown**, **HomeCategoryList**, and **CategoryNav**. In Project 3, we used a function **getCategories** in the data.ts file to import it from db.json whenever we needed it. Importing from a local file multiple times is fine. However, downloading the category data from the database using network access multiple times is not efficient. Therefore, we will download the category data only once and share the downloaded copy among multiple components. There are different ways of sharing data among components. For this project, we will use **props** to share data among components. This approach involves passing data down through components using props, similar to how we pass data to other functions using arguments in a traditional programming language. Since props require us to pass data down through the component hierarchy, we will fetch the category data from the database in the **App** component, which is the root component at the top of our hierarchy.

We will call the data to be shared **categories**. This will be the *state* of the App component that we are going to share with other components. React has a hook **useState** that is used to initialize and update the state data of a component. The following code creates a state called **categories**, initializes it with an empty array, and defines a mutator (setter) method **setCategories**.

```
const [categories, setCategories] = useState([]);
```

Next, we can use some utility code to use the correct url for accessing our server from the client no matter whether the code is running on your laptop or on the class server. In utils.ts add the following:

```
export const apiUrl =  
  `${location.protocol}//${location.hostname}:` +
```

```
`${location.port === '5173' ? '8080' : location.port}` +  
`${import.meta.env.BASE_URL}/api`
```

Now we can use **axios** and **useEffect** hook to download the categories data from the database.

```
import { apiUrl } from "../utils";  
...  
useEffect(() => {  
  axios.get(`${apiUrl}/categories`)  
    .then((result) => setCategories(result.data))  
    .catch(console.error);  
}, []);
```

The above code fragment downloads the category list from the database and updates the component state (**categories**) using the data. Since we need the category list to be downloaded only once, we use an empty array ( `[]` ) as a dependency array in **useEffect**. Note that **axios**, **useEffect**, and **useState** should be imported before we use them.

This is the easiest way to use the **get** method of **axios** to obtain data. You simply pass in the URL of the REST API you want. The **get** method returns a **Promise** object. A promise must resolve before it can be used. That's because promises are asynchronous – it takes time for them to resolve into an object. The **then** method is used to capture the object that the promise resolves to, and then you can manipulate that object. A promise resolves to a **Response** object, because you are getting a response back from the network. The first **then** method above takes the body of the response object and parses it into a data object. If everything works as expected, the **then** promise will resolve into a data object. If the promise is not fulfilled (an error happened) the **catch** block will be executed.

Another way of dealing with a promise is to use **async** and **await**. The **async** keyword is used when declaring a function and it denotes that the function is asynchronous. The **await** keyword is used when calling an asynchronous function. A function that returns a promise (like **fetch**) is asynchronous. You can get the same effect of the code above by using **async** and **await**. Once we are done fetching the categories from our REST API, we can share the data with other components using **props**.

The **getCategories(): CategoryItem[]** array we defined in **data.ts** is now downloaded in the **App** component, and it should be passed to the **HeaderDropdown** component. If you have categories on your welcome page, it should also be passed to the **CategoryBookList** component. Add the following interface in your **types.tsx**, and delete the **categoryList** array from the same file.

```
export interface CategoryProps {  
  categories: CategoryItem[];  
}
```

Second, modify the App component's JSX as:

```
return (  
  <Router basename={"[Name]BookstoreFetch"}>  
    <AppHeader categories={categories} />  
    <Routes>  
      <Route path="/" element={<WelcomePage categories={categories} />} />  
      <Route path="/categories/:categoryName" element={<CategoryPage categories={categories} />} />  
      <Route path="*" element={<div>Page Not Found</div>} />  
    </Routes>  
  )
```

When you see the expression **categories = {categories}** in the code above, we are passing the downloaded category information to the components. The **categories** on the left of the equals sign refers to the property of the **props** argument in the subcomponent (in this case the **AppHeader**, **Home**, and **CategoryBookList** components). All of these components will need a props argument, which has type **CategoryProps**. Go to the three components and modify them to accept the data, remove the categoryList from the import section, and import **CategoryProps**. For example, the AppHeader will be:

```
import { CategoryProps } from "../types";
...

function AppHeader(props: CategoryProps) {
  return (
    ...
    <HeaderDropdown categories={props.categories} />
    ...
  );
}
```

When you see the expression **categories = {props.categories}** in the AppHeader code above, the **categories** on the left of the equals sign refers to the property of the **props** argument in the subcomponent (in this case **HeaderDropdown**), which has type **CategoryProps**. The **{props.categories}** on the right side of the equals sign refers to the property of the props argument in the current component (in this case **AppHeader**).

Unlike regular JavaScript functions, a React function component is only allowed to have one input argument, which is the props. For simplicity in this description, we'll always name the argument **props**. But feel free to be more descriptive. For example, you may want to have categoryProps, bookProps, etc.) The props argument is an object with properties that can be accessed by the component.

Modify the rest of the components that need access to the categories accordingly.

Make sure your server project is running, and remember that you need to be connected with Virginia Tech's Pulse Secure app to access the database on the **cs5244.cs.vt.edu** server.

Now run your client project using **npm run dev**.

Your front-end should look the same as the previous bookstore site (Project 3). However, there will be an error that prevents the front-end from downloading the data. Go to "Developer tools" and open the console, you should see an error that says something similar to: **Access to fetch at 'http://localhost:8080/[name]BookstoreReactFetch/api/categories/' from origin 'http://localhost:5173' has been blocked by CORS policy.**

CORS stands for "Cross-Origin Resource Sharing". It is a policy that allows the server to decide which URLs can connect to it. By default, only clients coming from the same host **and the same port** can connect to the server. Our client and server share the same host (localhost), but they have a different port. Our client runs on port 5173 and our server runs on port 8080. To fix this problem, we have to modify the server so that it allows port 3000 to connect.



[Add a CORS filter to the server project](#)

Go into IntelliJ IDEA and stop Tomcat. Go to the "api" package in the java directory and create a Java class called "CorsFilter". Paste the code from the **CorsFilter.java** resource we gave you (above). Notice that it includes the line:

```
"Access-Control-Allow-Origin", "http://localhost:5173");
```

That tells the server that it should allow clients with that hostname and port to connect.

On the client side, add the following line in the **package.json** file

```
"proxy": "http://localhost:8080",
```

The server project does not have hot reloading like our client project does, so we have to restart Tomcat. Once we do, we can go back to the page displaying our React App (which does have hot reloading) and we should be able to see the header dropdown, and when we hover over it, we should see all the categories. Clicking on any one of the categories should take us to the category page, just like in Project 3.

## Use Axios to Fetch the Books

It's time to fetch the books. If your app is similar to AnotherBookstore, you get the book list in the **CategoryBookList** component and pass each book as a **prop** to the **CategoryBookListItem** component. The *CategoryBookList* component iterates over the books, and invokes the *CategoryBookListItem* component, passing the current book as a prop. This means, in principle, that you only need to fetch the books into the book list in the **CategoryBookList** component and everything should work. Make sure you have images for each book in your "src/assets/book-images" folder (if you are using the same folder structure as the AnotherBookstore). **Note that the CategoryBookList should now have a state, which is the book list array.**

We will use the **useEffect** hook and **Axios** to download the books from the database just the same way we did for downloading the category list. This function then updates the state (bookList array) with the downloaded data. The only slightly complex thing here is the URL path being passed into axios. If we need to fetch all "romance" books in our database the REST API will be:

[http://localhost:8080/\[name\]BookstoreReactFetch/api/categories/name/romance/books](http://localhost:8080/[name]BookstoreReactFetch/api/categories/name/romance/books) ➡  
(<http://localhost:8080/DrMBookstoreReactRest/api/categories/name/classics/books>)

The category name *romance* is based on the user's selection on the **Categories** button on the header component (AppHeader in AnotherBookstore), i.e., it is a parameter(variable) that can change. React has a hook called **useParam** that helps us to capture the user's selection and use it in a component. Here is the code snippet that will give us the complete URL:

```
const {categoryName} = useParams ();
...
axios.get(`http://localhost:8080/[Name]BookstoreFetch/api/categories/name/${categoryName}/books/`);
...
?
```



Notice the backticks (``...`) at the beginning and the end of that string. This is called a **template literal** [↗\(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template\\_literals\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals). You can think of the template literal used here as a string that can nest an expression inside of it.

The `${categoryName}` inside the template literal is an expression that evaluates to a string, and that string becomes part of the remaining string. This code successfully returns the books for the category name parameter inside the route. Clicking through the categories in the menu or the navbar will eventually display the books for that category, where the selected category's name is the category name that will be read as a parameter. For the navigation to work, we have to modify the routing in the App component.

Run your project and check if the URL on your browser address bar changes when you click on a specific category. Note that you have to configure the navigation in the application to display the appropriate category page to be displayed when the category is selected.

Go to the **Developer tools console** and make sure the app is not downloading the book information infinitely. Remember the purpose of the dependency array in `useEffect`.

## Style and Interface Requirements

Remember that the requirements of the project are cumulative, so all relevant requirements from Projects 1–4 still apply.

### Fix Past Problems

For Project 5, please fix any problems that an instructor or grader has mentioned in their comments on previous projects (specifically, Project 3, your React project). If you are still working on issues when your submission is due, let us know in Canvas when you submit Project 5. In general, we try not to take off points twice for the same problem if you did not have time to fix it. For example, if an instructor or grader took off points for something in Project 2, then you did not have time to fix it when you submitted Project 3, so we should not take off points for the same problem. However, for Project 5, you *will* have time to fix these problems, so we will take off points again.

### Implement a transition

- Implement a transition somewhere in your project. If the transition is not obvious (for example, sliding dropdown menus), please let us know what it is in Canvas when you submit Project 5
- If you implement a hamburger menu similar to the one in the hamburger-demo, please add or change something about the menu that distinguishes it from the one in the demo. For example, maybe the hamburger icon changes to an "X", or maybe the sub-menu comes out from the side.

### Buttons

- The CTA button (call-to-action) on the welcome page should take the user to your default category page. Your default category will typically be your first one (with ID 1001). This also means that your CTA button should be named appropriately. For example, it should be called "Shop Now" or "Shop for Books" or something like that. It should not be called "Sign up Now" or "Create an Account" or "Buy this Book Now".

- The cursor should always change to a pointer (hand) when you hover over any button (including icon buttons) or link. The CTA, primary, and secondary buttons should all have a style change (for example, a darker background or bold text) when the user hovers over them.
- Note that when you hover over an element and its style changes, the position of surrounding elements should not change. For example, if you hover over a button and the text changes to bold, the button should be large enough so that the size of the button does not change. If you intend the size of the button to change, there should be enough space around the button so that elements around it do not move.

### Padding in icon buttons and cart-count

- Make sure your icon buttons have padding around the icon. The icon should not touch the edge of the button. There should be at least a few pixels of padding around it.
- The same thing applies to the cart-count, whether you put your cart-count in a little circle or inside of your shopping cart. Make sure the count is not too close to the edge of the shape that surrounds it.
- If you put your cart count inside (or beside) your cart, make sure it is centered vertically or horizontally (as appropriate).

### Book Boxes

- We've seen a few book boxes (in Project 3) that do not wrap until the boxes get very thin. Make sure you set a min-width on your book box that keeps both the book image and the text (book info) easily readable.
  - The book image should not shrink. It should always keep its original height and width (and therefore it will always maintain its correct aspect ratio).
  - The text of the book title should not wrap so much that there is only one word on a line.
- Book boxes should wrap within the space of the browser width that we will be looking at, which is from about 1000 pixels to 1400 pixels. When we expand or shrink the browser between those two widths, we should see the book wrapping occur. For example, we should not have to shrink the browser to 600 pixels before we see evidence of wrapping.

### No Book Boxes on the Welcome Page

- Do not put book boxes on your welcome page. You can put book images on your welcome page, and even include the title, but do not put the same kind of book boxes that you have on your category page on your welcome page. Users should never be able to mistake your welcome page for your category page.
- Do not put add-to-cart buttons (or read-now buttons) on your welcome page. The user's attention should go to the call-to-action button, and the add-to-cart buttons are distractions from this goal.

### Miscellaneous

- Make sure that your header and footer extend the width of the page (up to the max width of the body). For example, do not use a footer that remains at the bottom left (or button right) of the page.



<http://cs5244.cs.vt.edu:8080/JinhengBookstoreFetch/>

Assignment Project Quiz Exam Essay Help  
WeChat: cestbon\_6888  
Email: accoder\_overseas@163.com

