React Fetch CRUD Codealong (CodeGrade)

(https://github.com/learn-co-curriculum/react-hooks-fetch-crud-code-along) (https://github.com/learn-co-curriculum/react-hooks-fetch-crud-code-along/issues/new)

Learning Goals

- Write `fetch` requests for `GET` , `POST` , `PATCH` , and `DELETE`
- Initiate `fetch` requests with the `useEffect` hook
- Initiate `fetch` requests from user events
- Update state and trigger a re-render after receiving a response to the `fetch` request
- Perform CRUD actions on arrays in state

Introduction

Up to this point, we've seen how to use `fetch` in a React application for some common single-page application patterns, such as:

- Requesting data from a server when our application first loads
- Persisting data to a server when a user submits a form

In both of those cases, our workflow in React follows a similar pattern:

- When X event occurs (*our application loads*, *a user submits a form*)
- Make Y fetch request (*GET*, *POST*)
- Update Z state (*add all items to state*, *add a single item to state*)

In this codealong lesson, we'll get more practice following this pattern to build out all four CRUD actions to work with both our **server-side** data (the database; in our case, the `db.json` file) as well as our **client-side** data (our React state). We'll be revisiting the shopping list application from the previous module, this time using `json-server` to create a RESTful API which we can interact with from React.

Instructions

To get started, let's install our dependencies:

```
$ npm install
```

Then, to run `json-server`, we'll be using the `server` script in the `package.json` file:

```
$ npm run server
```

This will run `json-server` on http://localhost:4000 (http://localhost:4000). Before moving ahead, open http://localhost:4000/items (http://localhost:4000/items) in the browser and familiarize yourself with the data. What are the important keys on each object?

Leave `json-server` running. Open a new terminal, and run React with:

```
$ npm start
```

View the application in the browser at http://localhost:3000 (http://localhost:3000). We don't have any data to display yet, but eventually, we'll want to display the list of items from `json-server` in our application and be able to perform CRUD actions on them.

Take some time now to familiarize yourself with the components in the `src/components` folder. Which components are stateful and why? What does our component hierarchy look like?

Once you've familiarized yourself with the starter code, let's start building out some features!

Displaying Items

Our first goal will be to display a list of items from the server when the application first loads. Let's see how this goal fits into this common pattern for working with server-side data in React:

- When X event occurs (*our application loads*)
- Make Y fetch request (*GET /items*)
- Update Z state (*add all items to state*)

With that structure in mind, our first step is to **identify which component triggers this event**. In this case, the event isn't triggered by a user interacting with a specific DOM element. We want to initiate the fetch request without making our users click a button or anything like that.

So the event we're looking for is a **side-effect** of a component being rendered. Which component? Well, we can make that determination by looking at which state we're trying to update. In our case, it's the `items` state which is held in the `ShoppingList` component.

We can call the `useEffect` hook in the `ShoppingList` component to initiate our `fetch` request. Let's start by using `console.log` to ensure that our syntax is correct, and that we're fetching data from the server:

```
// src/components/ShoppingList.js

// import useEffect
import React, { useEffect, useState } from "react";
// ...rest of imports

function ShoppingList() {
  const [selectedCategory, setSelectedCategory] = useState("All");
  const [items, setItems] = useState([]);

  // Add useEffect hook
  useEffect(() => {
    fetch("http://localhost:4000/items")
      .then((r) => r.json())
      .then((items) => console.log(items));
  }, []);

  // ...rest of component
}
```

Check your console in the browser — you should see an **array of objects** representing each item in our shopping list.

Now all that's left to do is to update state, so that React will re-render our components and use the new data to display our shopping list. Our goal here is to replace our current `items` state, which is an empty array, with the new array from the server:

```
// src/components/ShoppingList.js

function ShoppingList() {
  const [selectedCategory, setSelectedCategory] = useState("All");
  const [items, setItems] = useState([]);

    // Update state by passing the array of items to setItems
  useEffect(() => {
    fetch("http://localhost:4000/items")
      .then((r) => r.json())
      .then((items) => setItems(items));
  }, []);

  // ...rest of component
}
```

Check your work in the browser and make sure you see the list of items. Which component is responsible for rendering each item from the list of items in state?

To recap:

- When X event occurs
  - Use the `useEffect` hook to trigger a side-effect in the `ShoppingList` component after the component first renders
- Make Y fetch request
  - Make a `GET` request to `/items` to retrieve a list of items
- Update Z state
  - Replace our current list of items with the new list

Creating Items

Our next goal will be to add a new item to our database on the server when a user submits the form. Once again, let's plan out our steps:

- When X event occurs (*a user submits the form*)

- Make Y fetch request (*POST /items with the new item data*)
- Update Z state (*add a new item to state*)

To tackle the first step, we'll need to **identify which component triggers the event**. In this case, the form in question is in the `ItemForm` component. Let's start by handling the form `submit` event in this component and access the data from the form inputs, which are saved in state:

```
// src/components/ItemForm.js

function ItemForm() {
  const [name, setName] = useState("");
  const [category, setCategory] = useState("Produce");

  // Add function to handle submissions
  function handleSubmit(e) {
    e.preventDefault();
    console.log("name:", name);
    console.log("category:", category);
  }

  return (
    // Set up the form to call handleSubmit when the form is submitted
    <form className="NewItem" onSubmit={handleSubmit}>
      {/** ...form inputs here */}
    </form>
  );
}
```

One step down, two to go! Next, we need to determine what data needs to be sent to the server with our `fetch` request. Our goal is to create a new item, and it should have the same structure as other items on the server. So we'll need to send an object that looks like this:

```
{
  "name": "Yogurt",
  "category": "Dairy",
  "isInCart": false
}
```

Let's create this item in our `handleSubmit` function using the data from the form state:

```
// src/components/ItemForm.js

function handleSubmit(e) {
  e.preventDefault();
  const itemData = {
    name: name,
    category: category,
    isInCart: false,
  };
  console.log(itemData);
}
```

Check your work in the browser again and make sure you are able to log an item to the console that has the right key/value pairs. Now, on to the `fetch`!

```
function handleSubmit(e) {
  e.preventDefault();
  const itemData = {
    name: name,
    category: category,
    isInCart: false,
  };
  fetch("http://localhost:4000/items", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(itemData),
  })
    .then((r) => r.json())
    .then((newItem) => console.log(newItem));
}
```

Recall that to make a `POST` request, we must provide additional options along with the URL when calling `fetch` : the `method` (HTTP verb), the `headers` (specifying that we are sending a JSON string in the request), and the `body` (the stringified object we are sending). If you need a refresher on this syntax, check out the **[MDN article on Using Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#uploading_json_data) (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#uploading_json_data)** .

Try submitting the form once more. You should now see a new item logged to the console that includes an `id` attribute from the server. You can also verify the object was persisted by refreshing the page in the browser and seeing the new item at the bottom of the shopping list.

However, our goal isn't to make our users refresh the page to see their newly created item — we want it to show up as soon as it's been persisted. So we have one more step left: **updating state**.

For this final step, we need to consider:

- Which component owns the state that we're trying to update?
- How can we get the data from the `ItemForm` component to the component that owns state?
- How do we correctly update state?

For the first question, we're trying to update state in the `ShoppingList` component. Our goal is to display the new item in the list alongside the other items, and this is the component that is responsible for that part of our application. Since the `ShoppingList` component is a **parent** component to the `ItemForm` component, we'll need to **pass a callback function as a prop** so that the `ItemForm` component can send the new item up to the `ShoppingList` .

Let's add a `handleAddItem` function to `ShoppingList` , and pass a reference to that function as a prop called `onAddItem` to the `ItemForm` :

```
// src/components/ShoppingList.js

function ShoppingList() {
  const [selectedCategory, setSelectedCategory] = useState("All");
  const [items, setItems] = useState([]);

  useEffect(() => {
    fetch("http://localhost:4000/items")
      .then((r) => r.json())
      .then((items) => setItems(items));
  }, []);

  // add this function!
  function handleAddItem(newItem) {
    console.log("In ShoppingList:", newItem);
  }

  function handleCategoryChange(category) {
    setSelectedCategory(category);
  }

  const itemsToDisplay = items.filter((item) => {
    if (selectedCategory === "All") return true;

    return item.category === selectedCategory;
  });

  return (
    <div className="ShoppingList">
      {/* add the onAddItem prop! */}
      <ItemForm onAddItem={handleAddItem} />
      <Filter
        category={selectedCategory}
        onCategoryChange={handleCategoryChange}
      />
      <ul className="Items">
        {itemsToDisplay.map((item) => (
          <Item key={item.id} item={item} />
        ))}
      </ul>
    </div>
  );
}
```

Then, we can use this prop in the `ItemForm` to send the new item **up** to the `ShoppingList` when we receive a response from the server:

```
// src/components/ItemForm.js

// destructure the onAddItem prop
function ItemForm({ onAddItem }) {
  const [name, setName] = useState("");
  const [category, setCategory] = useState("Produce");

  function handleSubmit(e) {
    e.preventDefault();
    const itemData = {
      name: name,
      category: category,
      isInCart: false,
    };
    fetch("http://localhost:4000/items", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(itemData),
    })
      .then((r) => r.json())
      // call the onAddItem prop with the newItem
      .then((newItem) => onAddItem(newItem));
  }

  // ...rest of component
}
```

Check your work by submitting the form once more. You should now see the new item logged to the console, this time from the `ShoppingList` component. We're getting close! For the last step, we need to call `setState` with a new array that has our new item at the end. Recall from our lessons on working with arrays in state that we can use the spread operator to perform this action:

```
// src/components/ShoppingList.js

function handleAddItem(newItem) {
  setItems([...items, newItem]);
}
```

Now each time a user submits the form, a new item will be added to our database and will also be added to our client-side state, so that the user will immediately see their item in the list.

Let's recap our steps here:

- When X event occurs
  - When a user submits the `ItemForm` , handle the form submit event and access data from the form using state
- Make Y fetch request
  - Make a `POST` request to `/items` , passing the form data in the **body** of the request, and access the newly created item in the response
- Update Z state
  - Send the item from the fetch response to the `ShoppingList` component, and set state by creating a new array with our current items from state, plus the new item at the end

Phew! This is a good time to **take a break** before proceeding — we've got a few more steps to cover. Once you're ready and recharged, we'll dive back in.

Updating Items

For our update action, we'd like to give our users the ability to keep track of which items from their shopping list they've added to their cart. Once more, we can outline the basic steps for this action like so:

- When X event occurs (*a user clicks the Add to Cart button*)
- Make Y fetch request (*PATCH /items*)
- Update Z state (*update the* `isInCart` *status for the item*)

From here, we'll again need to **identify which component triggers the event**. Can you find where the "Add to Cart" button lives in our code? Yep! It's in the `Item` component. We'll start by adding an event handler for clicks on the button:

```
// src/components/Item.js

function Item({ item }) {
  // Add function to handle button click
  function handleAddToCartClick() {
    console.log("clicked item:", item);
  }

  return (
    <li className={item.isInCart ? "in-cart" : ""}>
      <span>{item.name}</span>
      <span className="category">{item.category}</span>
      {/* add the onClick listener */}
      <button
        className={item.isInCart ? "remove" : "add"}
        onClick={handleAddToCartClick}
      >
        {item.isInCart ? "Remove From" : "Add to"} Cart
      </button>
      <button className="remove">Delete</button>
    </li>
    );
  }
```

Check your work by clicking this button for different items — you should see each item logged to the console. We can access the `item` variable in the `handleAddToCartClick` function thanks to JavaScript's scope rules.

Next, let's write out our `PATCH` request:

```
// src/components/Item.js

function handleAddToCartClick() {
  // add fetch request
  fetch(`http://localhost:4000/items/${item.id}`, {
    method: "PATCH",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      isInCart: !item.isInCart,
    }),
  })
    .then((r) => r.json())
    .then((updatedItem) => console.log(updatedItem));
}
```

Just like with our `POST` request, we need to specify the `method` , `headers` , and `body` options for our `PATCH` request as well. We also need to include the item's ID in the URL so that our server knows which item we're trying to update.

Since our goal is to let users add or remove items from their cart, we need to toggle the `isInCart` property of the item on the server (and eventually client-side as well). So in the body of the request, we send an object with the key we are updating, along with the new value.

Check your work by clicking the "Add to Cart" button and see if you receive an updated item in the response from the server. Then, refresh the page to verify that the change was persisted server-side.

Two steps done! Once more, our last step is to update state. Let's think this through. Right now, what state determines whether or not the item is in our cart? Where does that state live? Do we need to add new state?

A reasonable thought here is that we might need to add new state to the `Item` component to keep track of whether or not the item is in the cart. While we could theoretically make this approach work, it would be an anti-pattern: we'd be **duplicating state**, which makes our components harder to work with and more prone to bugs.

We already have state in our `ShoppingList` component that tells us which items are in the cart. So instead of creating new state, our goal is to call `setItems` in the `ShoppingList` component with a new list of items, where the `isInCart` state of our updated item matches its state on the server.

Just like with our `ItemForm` deliverable, let's start by creating a callback function in the `ShoppingList` component and passing it as a prop to the `Item` component:

```
// src/components/ShoppingList.js

function ShoppingList() {
  const [selectedCategory, setSelectedCategory] = useState("All");
  const [items, setItems] = useState([]);

  useEffect(() => {
    fetch("http://localhost:4000/items")
      .then((r) => r.json())
      .then((items) => setItems(items));
  }, []);

  // add this callback function
  function handleUpdateItem(updatedItem) {
    console.log("In ShoppingCart:", updatedItem);
  }

  // ...rest of component
```

```
    return (
      <div className="ShoppingList">
        <ItemForm onAddItem={handleAddItem} />
        <Filter
          category={selectedCategory}
          onCategoryChange={handleCategoryChange}
        />
        <ul className="Items">
          {/* pass it as a prop to Item */}
          {itemsToDisplay.map((item) => (
            <Item key={item.id} item={item} onUpdateItem={handleUpdateItem} />
          ))}
        </ul>
      </div>
    );
  }
```

In the `Item` component, we can destructure the `onUpdateItem` prop and call it when we have the updated item response from the server:

```
// src/components/Item.js

// Destructure the onUpdateItem prop
function Item({ item, onUpdateItem }) {
  function handleAddToCartClick() {
    // Call onUpdateItem, passing the data returned from the fetch request
    fetch(`http://localhost:4000/items/${item.id}`, {
      method: "PATCH",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        isInCart: !item.isInCart,
      }),
    })
      .then((r) => r.json())
      .then((updatedItem) => onUpdateItem(updatedItem));
  }
  // ... rest of component
}
```

Check your work by clicking the button once more. You should now see the updated item logged to the console, this time from the `ShoppingList` component.

As a last step, we need to call `setState` with a new array that replaces one item with the new updated item from the server. Recall from our lessons on working with arrays in state that we can use `.map` to help create this new array:

```
// src/components/ShoppingList.js

function handleUpdateItem(updatedItem) {
  const updatedItems = items.map((item) => {
    if (item.id === updatedItem.id) {
      return updatedItem;
    } else {
      return item;
    }
  });
  setItems(updatedItems);
}
```

Clicking the button should now toggle the `isInCart` property of any item in the list on the server as well as in our React state! To recap:

- When X event occurs
  - When a user clicks the Add to Cart button, handle the button click
- Make Y fetch request
  - Make a `PATCH` request to `/items/:id`, using the clicked item's data for the ID and body of the request, and access the updated item in the response

- Update Z state
    - Send the item from the fetch response to the `ShoppingList` component, and set state by creating a new array which contains the updated item in place of the old item

Deleting Items

Last one! For our delete action, we'd like to give our users the ability to remove items from their shopping list:

- When X event occurs (*a user clicks the DELETE button*)
- Make Y fetch request (*DELETE /items*)
- Update Z state (*remove the item from the list*)

From here, we'll again need to **identify which component triggers the event**. Our delete button is in the `Item` component, so we'll start by adding an event handler for clicks on the button:

```
// src/components/Item.js

function Item({ item, onUpdateItem }) {
  // ...rest of component

  function handleDeleteClick() {
    console.log(item);
  }

  return (
    <li className={item.isInCart ? "in-cart" : ""}>
      {/* ... rest of JSX */}

      {/* ... add onClick */}
      <button className="remove" onClick={handleDeleteClick}>
        Delete
      </button>
    </li>
    );
  }
```

This step should feel similar to our approach for the update action. Next, let's write out our `DELETE` request:

```
// src/components/Item.js

function handleDeleteClick() {
  fetch(`http://localhost:4000/items/${item.id}`, {
    method: "DELETE",
  })
    .then((r) => r.json())
    .then(() => console.log("deleted!"));
}
```

Note that for a `DELETE` request, we must include the ID of the item we're deleting in the URL. We only need the `method` option — no `body` or `headers` are needed since we don't have any additional data to send besides the ID.

You can verify that the item was successfully deleted by clicking the button, checking that the console message of `"deleted!"` appears, and refreshing the page to check that the item is no longer on the list.

Our last step is to update state. Once again, the state that determines which items are being displayed is the `items` state in the `ShoppingList` component, so we need to call `setItems` in that component with a new list of items that **does not contain our deleted item**.

We'll pass a callback down from `ShoppingList` to `Item`, just like we did for the update action:

```
// src/components/ShoppingList.js

function ShoppingList() {
  const [selectedCategory, setSelectedCategory] = useState("All");
  const [items, setItems] = useState([]);

  useEffect(() => {
    fetch("http://localhost:4000/items")
      .then((r) => r.json())
      .then((items) => setItems(items));
  }, []);

  // add this callback function
  function handleDeleteItem(deletedItem) {
    console.log("In ShoppingCart:", deletedItem);
  }

  // ...rest of component

  return (
    <div className="ShoppingList">
      <ItemForm onAddItem={handleAddItem} />
      <Filter
        category={selectedCategory}
        onCategoryChange={handleCategoryChange}
      />
      <ul className="Items">
        {/* pass it as a prop to Item */}
        {itemsToDisplay.map((item) => (
          <Item
            key={item.id}
            item={item}
            onUpdateItem={handleUpdateItem}
            onDeleteItem={handleDeleteItem}
          />
        ))}
      </ul>
    </div>
    );
}
```

Call the `onDeleteItem` prop in the `Item` component once the item has been deleted from the server, and pass up the item that was clicked:

```
// src/components/Item.js

// Deconstruct the onDeleteItem prop
function Item({ item, onUpdateItem, onDeleteItem }) {
  function handleDeleteClick() {
    // Call onDeleteItem, passing the deleted item
    fetch(`http://localhost:4000/items/${item.id}`, {
      method: "DELETE",
    })
      .then((r) => r.json())
      .then(() => onDeleteItem(item));
  }
  // ... rest of component
}
```

As a last step, we need to call `setState` with a new array that removes the deleted item from the list. Recall from our lessons on working with arrays in state that we can use `.filter` to help create this new array:

```
// src/components/ShoppingList.js
function handleDeleteItem(deletedItem) {
  const updatedItems = items.filter((item) => item.id !== deletedItem.id);
  setItems(updatedItems);
}
```

Clicking the delete button should now delete the item in the list on the server as well as in our React state! To recap:

- When X event occurs
  - When a user clicks the Delete button, handle the button click

- Make Y fetch request
  - Make a `DELETE` request to `/items/:id`, using the clicked item's data for the ID
- Update Z state
  - Send the clicked item to the `ShoppingList` component, and set state by creating a new array in which the deleted item has been filtered out

Conclusion

Synchronizing state between a client-side application and a server-side application is a challenging problem! Thankfully, the general steps to accomplish this in React are the same regardless of what kind of action we are performing:

- When X event occurs
- Make Y fetch request
- Update Z state

Keep these steps in mind any time you're working on a feature involving synchronizing client and server state. Once you have this general framework, you can apply the other things you've learned about React to each step, like handling events, updating state, and passing data between components.

Resources

- [MDN: Using Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#uploading_json_data) (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch#uploading_json_data)

This tool needs to be loaded in a new browser window

Load React Fetch CRUD Codealong (CodeGrade) in a new window