Data Fetching in React

[(https://github.com/learn-co-curriculum/react-hooks-data-fetching)](https://github.com/learn-co-curriculum/react-hooks-data-fetching) [(https://github.com/learn-co-curriculum/react-hooks-data-fetching/issues/new)](https://github.com/learn-co-curriculum/react-hooks-data-fetching/issues/new)

Learning Goals

- Use `fetch` within components
- Decide whether to use `fetch` from an event handler or `useEffect` callback

Problem Statement

We've seen that React components come with some neat-o bells and whistles. They can be nested within each other. They can pass information and logic between them with props and they can keep track of their own information in state.

So far though, we've been restricted to displaying information organized by the React app itself. In this lesson, we're going to go a step further and incorporate remote data into our React apps. Using `fetch` requests to APIs, we can build dynamic, responsive apps around data that is provided to us remotely.

Fetching Data in React Components

For a minute, consider how a site like **[Instagram](https://www.instagram.com/)** (https://www.instagram.com/) works. If you've got an account on Instagram, when you visit the site, you'll see an endless scroll of photos from people you follow. Everyone sees the same *Instagram* website, but the images displayed are unique to the user.

Similarly, consider **[AirBnb](https://airbnb.com/)** (https://airbnb.com/). When you click to look at a listing's information, the page layout is always the same. The data, the images, the reviews... this information changes.

Both of these websites are built with React. When you go to one of these sites, React doesn't have the specific listing or image content. If you're on a slow connection (or **[want to mimic one using the Chrome Dev Tools](https://developer.chrome.com/docs/devtools/network/reference#throttling)** (https://developer.chrome.com/docs/devtools/network/reference#throttling)), you can see what is happening more clearly.

React shows up first and renders *something*. Sometimes it is just the background or the skeleton of a website, or maybe navigation and CSS. On Instagram, a photo 'card' might appear but without an image or username attached.

React is *updating the DOM* based on the JSX being returned by its components *first*. Once the DOM has been updated, remote data is then requested. When that data has been received, React runs through an update of the necessary components and fills in the info it received. Text content will appear, user information, etc... This first set of data is likely just a JSON object specific to the user or content requested. This object might contain image URLs, so right after the component update, images will be able to load.

So, since the data is being requested *after* React has updated the DOM, is there a *side effect* that might be useful here?

Well, yes there is! Whenever we want to fetch data in our components without making a user trigger that request by clicking a button or submitting a form, the `useEffect` hook gives us a great place to do that. By putting a `fetch()` within `useEffect` , when the data is received, we can use `setState` to store the received data. This causes an update with that remote data stored in state. A very simple implementation of the App component with `fetch` might look like this:

```
import React, { useState, useEffect } from "react";

function App() {
  const [peopleInSpace, setPeopleInSpace] = useState([]);

  useEffect(() => {
    fetch("http://api.open-notify.org/astros.json")
      .then((response) => response.json())
      .then((data) => {
        setPeopleInSpace(data.people);
      });
  }, []);
  // use an empty dependencies array, so we only run the fetch request ONCE

  return <div>{peopleInSpace.map((person) => person.name)}</div>;
}

export default App;
```

React includes a **similar example of this pattern (https://reactjs.org/docs/faq-ajax.html#example-using-ajax-results-to-set-local-state)** in their FAQs, since it's a pretty common problem for single page applications.

In the code above, we start with an empty array in state for `peopleInSpace` . When the component is first rendered to the DOM, it will just display an empty `<div>` , since we have no elements in the `peopleInSpace` array.

After the `App` component has been rendered to the DOM, the `useEffect` callback runs, and `fetch` initiates a network request to an API. Once data is returned from the API, we update React's internal state with that data, which tells React to re-render our component. After the component re-renders, new content will appear in the `<div>` based on the `peopleInSpace` array.

Placing `fetch` in a `useEffect` with an empty dependencies array is ideal for data that you need immediately when a user visits your website or uses your app. Since `useEffect` is also commonly used to initialize intervals, it is ideal to set up any repeating fetch requests here as well.

We can also add a loading indicator using this technique. Since our component will render once *before* `useEffect` runs our `fetch` request, we can set up another state variable to add a loading indicator, like this:

```
function App() {
  const [peopleInSpace, setPeopleInSpace] = useState([]);
  const [isLoaded, setIsLoaded] = useState(false);

  useEffect(() => {
    fetch("http://api.open-notify.org/astros.json")
      .then((response) => response.json())
      .then((data) => {
        setPeopleInSpace(data.people);
        setIsLoaded(true);
      });
  }, []);

  // if the data hasn't been loaded, show a loading indicator
  if (!isLoaded) return <h3>Loading...</h3>;
  return <div>{peopleInSpace.map((person) => person.name)}</div>;
}
```

Fetching Data With Events

We aren't limited to sending fetch requests with `useEffect`. We can also tie them into events:

```
function handleClick() {
  fetch("your API url")
    .then((res) => res.json())
    .then((json) => setData(json));
}

return <button onClick={handleClick}>Click to Fetch!</button>;
```

This lets us send requests on demand. Submitting form data would be handled this way, using a POST request instead of GET.

A slightly more complicated example would be the infinite scroll of sites like Instagram. An event listener tied to changes in the scroll position of a page could fire off a `handleScroll` method that requests data before a user reaches the bottom of a page.

Using State with POST Requests

One of the beautiful features of state is that we can organize it however we need. If we were building a form to submit to a server, we can structure state to work nicely with what the server is expecting in a POST request.

Say we were building a user sign up form. When we send the data, our server is expecting two values within the body of the POST, `username` and `password`.

Setting up a React controlled form, we can structure our state in the same way:

```
function Form() {
  const [formData, setFormData] = useState({
    username: "",
    password: "",
  });

  //since the id values are the same as the keys in formData, we can write an abstract setFormData here
  function handleChange(event) {
    setFormData({
      ...formData,
      [event.target.id]: event.target.value,
    });
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        id="username"
        value={formData.username}
        onChange={handleChange}
      />
      <input
        type="text"
        id="password"
        value={formData.password}
        onChange={handleChange}
      />
    </form>
  );
}
```

Then, when setting up the fetch request, we can just pass the entire state within the body, as there are no other values:

```
  function handleSubmit(event) {
    event.preventDefault();
    fetch("the server URL", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(formData),
    });
  }
```

Notice how we're not bothering to worry about `event.target` when posting the data. Since the form is controlled, state contains the most up-to-date form data, and it is already in the right format!

Conclusion

When you need to *get* data from an API when your component is first rendered, using `useEffect` with an empty dependencies array, like this, is a good approach:

```
useEffect(() => {
  fetch("/api")
    .then((r) => r.json())
    .then(setData);
}, []);
```

Aside from that, there are no hard and fast rules for how to include fetch requests, and a lot of structure will depend on the data you're working with. As a general practice for writing simpler component code, include `fetch` calls in the same component as your top level state.

Resources

- **fetch (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)**
- **React `fetch` with `useEffect` Example (https://reactjs.org/docs/faq-ajax.html#example-using-ajax-results-to-set-local-state)**