CHAPTER 8

Incorporating Data

Data is the lifeblood of our applications. It flows like water, and it nourishes our components with value. The user interface components we've composed are vessels for data. We fill our applications with data from the internet. We collect, create, and send new data to the internet. The value of our applications is not the components themselves—it's the data that flows through those components.

When we talk about data, it may sound a little like we're talking about water or food. *The cloud* is the abundantly endless source from which we send and receive data. It's the internet. It's the networks, services, systems, and databases where we manipulate and store zettabytes of data. The cloud *hydrates* our clients with the latest and freshest data from the source. We work with this data locally and even store it locally. But when our local data becomes out of sync with the source, it loses its freshness and is said to be *stale*.

These are the challenges we face as developers working with data. We need to keep our applications hydrated with fresh data from the cloud. In this chapter, we're going to take a look at various techniques for loading and working with data from the source.

Requesting Data

In the movie Star Wars, the droid C-3P0 is a protocol droid. His specialty, of course, is communication. He speaks over six million languages. Surely, C-3P0 knows how to send an HTTP request, because the Hyper Text Transfer Protocol is one of the most popular ways to transmit data to and from the internet.

HTTP provides the backbone for our internet communication. Every time we load *http://www.google.com* into our browser, we're asking Google to send us a search form. The files necessary for us to search are transmitted to the browser over HTTP.

When we interact with Google by searching for "cat photos," we're asking Google to find us cat photos. Google responds with data, and images are transferred to our browser over HTTP.

In JavaScript, the most popular way to make an HTTP request is to use fetch. If we wanted to ask GitHub for information about Moon Highway, we could do so by sending a fetch request:

```
fetch(`https://api.github.com/users/moonhighway`)
  .then(response => response.json())
  .then(console.log)
  .catch(console.error);
```

The `fetch` function returns a promise. Here, we're making an asynchronous request to a specific URL: *https://api.github.com/users/moonhighway*. It takes time for that request to traverse the internet and respond with information. When it does, that information is passed to a callback using the `.then(callback)` method. The GitHub API will respond with JSON data, but that data is contained in the body of the HTTP response, so we call `response.json()` to obtain that data and parse it as JSON. Once obtained, we log that data to the console. If anything goes wrong, we'll pass the error to the `console.error` method.

GitHub will respond to this request with a JSON object:

```
{
  "login": "MoonHighway",
  "id": 5952087,
  "node_id": "MDEyOk9yZ2FuaXphdGlvbjU5NTIwODc=",
  "avatar_url": "https://avatars0.githubusercontent.com/u/5952087?v=4",
  "bio": "Web Development classroom training materials.",

  ...

}
```

On GitHub, basic information about user accounts is made available by their API. Go ahead, try searching for yourself: *https://api.github.com/users/<YOUR_GITHUB_USER_NAME>*.

Another way of working with promises is to use `async/await`. Since `fetch` returns a promise, we can `await` a fetch request inside of an `async` function:

```
async function requestGithubUser(githubLogin) {
  try {
    const response = await fetch(
      `https://api.github.com/users/${githubLogin}`
    );
    const userData = await response.json();
    console.log(userData);
  } catch (error) {
    console.error(error);
    }
  }
```

This code achieves the exact same results as the previous fetch request that was made by chaining `.then` functions on to the request. When we `await` a promise, the next line of code will not be executed until the promise has resolved. This format gives us a nice way to work with promises in code. We'll be using both approaches for the remainder of this chapter.

Sending Data with a Request

A lot of requests require us to upload data with the request. For instance, we need to collect information about a user in order to create an account, or we may need new information about a user to update their account.

Typically, we use a POST request when we're creating data and a PUT request when we're modifying it. The second argument of the `fetch` function allows us to pass an object of options that `fetch` can use when creating our HTTP request:

```
fetch("/create/user", {
  method: "POST",
  body: JSON.stringify({ username, password, bio })
});
```

This fetch is using the POST method to create a new user. The `username`, `password`, and user's `bio` are being passed as string content in the `body` of the request.

Uploading Files with fetch

Uploading files requires a different type of HTTP request: a `multipart-formdata` request. This type of request tells the server that a file or multiple files are located in the body of the request. To make this request in JavaScript, all we have to do is pass a `FormData` object in the body of our request:

```
const formData = new FormData();
formData.append("username", "moontahoe");
formData.append("fullname", "Alex Banks");
forData.append("avatar", imgFile);

fetch("/create/user", {
  method: "POST",
  body: formData
});
```

This time, when we create a user, we're passing the `username`, `fullname`, and `avatar` image along with the request as a `formData` object. Although these values are hardcoded here, we could easily collect them from a form.

Authorized Requests

Sometimes, we need to be authorized to make requests. Authorization is typically required to obtain personal or sensitive data. Additionally, authorization is almost always required for users to take action on the server with POST, PUT, or DELETE requests.

Users typically identify themselves with each request by adding a unique token to the request that a service can use to identify the user. This token is usually added as the `Authorization` header. On GitHub, you can see your personal account information if you send a token along with your request:

```
fetch(`https://api.github.com/users/${login}`, {
  method: "GET",
  headers: {
    Authorization: `Bearer ${token}`
  }
});
```

Tokens are typically obtained when a user signs into a service by providing their username and password. Tokens can also be obtained from third parties like GitHub or Facebook using with an open standard protocol called OAuth.

GitHub allows you to generate a Personal User token. You can generate one by logging in to GitHub and navigating to: Settings > Developer Settings > Personal Access Tokens. From here, you can create tokens with specific read/write rules and then use those tokens to obtain personal information from the GitHub API. If you generate a Personal Access Token and send it along with the fetch request, GitHub will provide additional private information about your account.

Fetching data from within a React component requires us to orchestrate the `useState` and `useEffect` hooks. The `useState` hook is used to store the response in state, and the `useEffect` hook is used to make the fetch request. For example, if we wanted to display information about a GitHub user in a component, we could use the following code:

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

function GitHubUser({ login }) {
  const [data, setData] = useState();

  useEffect(() => {
    if (!login) return;
    fetch(`https://api.github.com/users/${login}`)
      .then(response => response.json())
      .then(setData)
      .catch(console.error);
  }, [login]);
  if (data)
    return <pre>{JSON.stringify(data, null, 2)}</pre>;

  return null;
}

export default function App() {
  return <GitHubUser login="moonhighway" />;
}
```

In this code, our `App` renders a `GitHubUser` component and displays JSON data about `moonhighway`. On the first render, `GitHubUser` sets up a state variable for `data` using the `useState` hook. Then, because `data` is initially `null`, the component returns `null`. Returning `null` from a component tells React to render nothing. It doesn't cause an error; we'll just see a black screen.

After the component is rendered, the `useEffect` hook is invoked. This is where we make the fetch request. When we get a response, we obtain and parse the data in that response as JSON. Now we can pass that JSON object to the `setData` function, which causes our component to render once again, but this time it will have data. This `useEffect` hook will not be invoked again unless the value for `login` changes. When it does, we'll need to request more information about a different user from GitHub.

When there is `data`, we're rendering it as a JSON string in a `pre` element. The `JSON.stringify` method takes three arguments: the JSON data to convert to a string, a replacer function that can be used to replace properties of the JSON object, and the number of spaces to use when formatting the data. In this case, we sent `null` as the replacer because we don't want to replace anything. The `2` represents the number of spaces to be used when formatting the code. This will indent the JSON string two spaces. Using the `pre` element honors whitespace, so readable JSON is what is finally rendered.

Saving Data Locally

We can save data locally to the browser using the Web Storage API. Data can be saved by either using the `window.localStorage` or `window.sessionStorage` objects. The `sessionStorage` API only saves data for the user's session. Closing the tabs or restarting the browser will clear any data saved to `sessionStorage`. On the other hand, `localStorage` will save data indefinitely until you remove it.

JSON data should be saved in browser storage as a string. This means converting an object into a JSON string before saving it and parsing that string into JSON while loading it. Some function to handle saving and loading JSON data to the browser could look like:

```
const loadJSON = key =>
  key && JSON.parse(localStorage.getItem(key));
```

```
const saveJSON = (key, data) =>
  localStorage.setItem(key, JSON.stringify(data));
```

The `loadJSON` function loads an item from `localStorage` using the `key`. The `localStorage.getItem` function is used to load the data. If the item is there, it's then parsed into JSON before being returned. If it's not there, the `loadJSON` function will return `null`.

The `saveJSON` function will save some data to `localStorage` using a unique `key` identifier. The `localStorage.setItem` function can be used to save data to the browser. Before saving the data, we'll need to convert it to a JSON string.

Loading data from web storage, saving data to web storage, stringifying data, and parsing JSON strings are all synchronous tasks. Both the `loadJSON` and `saveJSON` functions are synchronous. So be careful—calling these functions too often with too much data can lead to performance issues. It's typically a good idea to throttle or debounce these functions for the sake of performance.

We could save the user's data that we received from our GitHub request. Then the next time that same user is requested, we could use the data saved to `localStorage` instead of sending another request to GitHub. We'll add the following code to the `GitHubUser` component:

```
const [data, setData] = useState(loadJSON(`user:${login}`));
useEffect(() => {
  if (!data) return;
  if (data.login === login) return;
  const { name, avatar_url, location } = data;
  saveJSON(`user:${login}`, {
    name,
    login,
    avatar_url,
    location
  });
}, [data]);
```

The `loadJSON` function is synchronous, so we can use it when we invoke `useState` to set the initial value for data. If there was user data saved to the browser under `user:moonhighway`, we'll initially set the data using that value. Otherwise, `data` will initially be `null`.

When `data` changes here after it has been loaded from GitHub, we'll invoke `saveJSON` to save only those user details that we need: `name`, `login`, `avatar_url`, and `location`. No need to save the rest of the user object when we're not using it. We also skip saving the `data` when that object is empty, `!data`. Also, if the current login and `data.login` are equal to each other, then we already have saved data for that user. We'll skip the step of saving that data again.

Here's a look at the entire `GitHubUser` component that uses `localStorage` to save data in the browser:

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

const loadJSON = key =>
  key && JSON.parse(localStorage.getItem(key));
const saveJSON = (key, data) =>
  localStorage.setItem(key, JSON.stringify(data));

function GitHubUser({ login }) {
  const [data, setData] = useState(
    loadJSON(`user:${login}`)
  );

  useEffect(() => {
    if (!data) return;
    if (data.login === login) return;
    const { name, avatar_url, location } = data;
    saveJSON(`user:${login}`, {
      name,
      login,
      avatar_url,
      location
    });
  }, [data]);

  useEffect(() => {
    if (!login) return;
    if (data && data.login === login) return;
    fetch(`https://api.github.com/users/${login}`)
      .then(response => response.json())
      .then(setData)
      .catch(console.error);
  }, [login]);

  if (data)
    return <pre>{JSON.stringify(data, null, 2)}</pre>;

  return null;
}
```

Notice the `GitHubUser` component now has two `useEffect` hooks. The first hook is used to save the data to the browser. It's invoked whenever the value for `data` changes. The second hook is used to request more data from GitHub. The fetch request is not sent when there's already data saved locally for that user. This is handled by the second `if` statement in the second `useEffect` hook: `if (data && data.login === login) return;`. If there is `data` and the `login` for that data matches the `login` property, then there's no need to send an additional request to GitHub. We'll just use the local data.

The first time we run the application, if the `login` is set to `moonhighway`, the following object will be rendered to the page:

```
{
  "login": "MoonHighway",
  "id": 5952087,
  "node_id": "MDEyOk9yZ2FuaXphdGlvbjU5NTIwODc=",
  "avatar_url": "https://avatars0.githubusercontent.com/u/5952087?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/MoonHighway",
  "html_url": "https://github.com/MoonHighway",

  ...

}
```

This is the response from GitHub. We can tell because this object contains a lot of extra information about the user that we don't need. The first time we run this page we'll see this lengthy response. But the second time we run the page, the response is much shorter:

```
{
  "name": "Moon Highway",
  "login": "moonhighway",
  "avatar_url": "https://avatars0.githubusercontent.com/u/5952087?v=4",
  "location": "Tahoe City, CA"
}
```

This time, the data we saved locally for `moonhighway` is being rendered to the browser. Since we only needed four fields of data, we only saved four fields of data. We'll always see this smaller offline object until we clear the storage:

```
localStorage.clear();
```

Both `sessionStorage` and `localStorage` are essential weapons for web developers. We can work with this local data when we're offline, and they allow us to increase the performance of our applications by sending fewer network requests. However, we must know when to use them. Implementing offline storage adds complexity to our applications, and it can make them tough to work with in development. Additionally, we don't need to work with web storage to cache data. If we're simply looking for a performance bump, we could try letting HTTP handle caching. Our browser will automatically cache content if we add `Cache-Control: max-age=<EXP_DATE>` to our headers. The `EXP_DATE` defines the expiration date for the content.

Handling Promise States

HTTP requests and promises both have three states: pending, success (fulfilled), and fail (rejected). A request is *pending* when we make the request and are waiting for a response. That response can only go one of two ways: success or fail. If a response is successful, it means we've successfully connected to the server and have received data. In the world of promises, a successful response means that the promise has been *resolved*. If something goes wrong during this process, we can say the HTTP request has failed or the promise has been *rejected*. In both cases, we'll receive an `error` explaining what happened.

We really need to handle all three of these states when we make HTTP requests. We can modify the GitHub user component to render more than just a successful response. We can add a "loading…" message when the request is pending, or we can render the `error` details if something goes wrong:

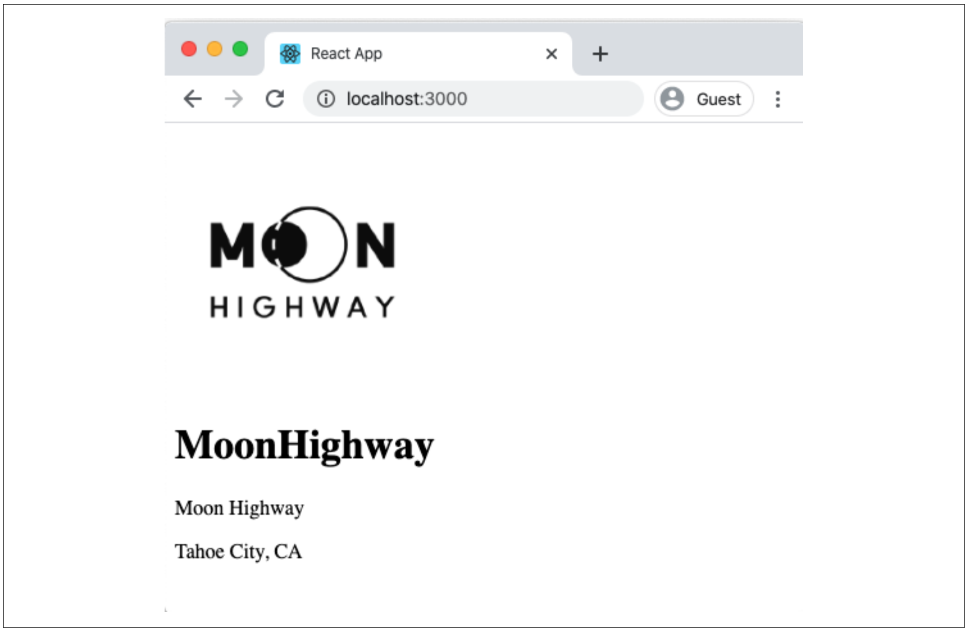
```
function GitHubUser({ login }) {
  const [data, setData] = useState();
  const [error, setError] = useState();
  const [loading, setLoading] = useState(false);

  useEffect(() => {
    if (!login) return;
    setLoading(true);
    fetch(`https://api.github.com/users/${login}`)
      .then(data => data.json())
      .then(setData)
      .then(() => setLoading(false))
      .catch(setError);
  }, [login]);

  if (loading) return <h1>loading...</h1>;
  if (error)
    return <pre>{JSON.stringify(error, null, 2)}</pre>;
  if (!data) return null;

  return (
    <div className="githubUser">
      <img
        src={data.avatar_url}
        alt={data.login}
        style={{ width: 200 }}
      />
      <div>
        <h1>{data.login}</h1>
        {data.name && <p>{data.name}</p>}
        {data.location && <p>{data.location}</p>}
      </div>
    </div>
  );
}
```

When this request is successful, Moon Highway's information is rendered for the user to see on the screen, as shown in Figure 8-1.



MoonHighway

Moon Highway

Tahoe City, CA

Figure 8-1. Sample output

If something goes wrong, we're simply displaying the `error` object as a JSON string. In production, we would do more with the error. Maybe we would track it, log it, or try to make another request. While in development, it's OK to render error details, which gives the developer instant feedback.

Finally, while the request is pending, we simply display a "loading…" message using an `h1`.

Sometimes an HTTP request can succeed with an error. This happens when the request was successful—successfully connected to a server and received a response—but the response body contains an error. Sometimes servers pass additional errors as successful responses.

Handling all three of these states bloats our code a little bit, but it's essential to do so on every request. Requests take time and a lot could go wrong. Because all requests—and promises—have these three states, it makes it possible to handle all HTTP requests with a reusable hook, or a component, or even a React feature called Suspense. We'll cover each of these approaches, but first, we must introduce the concept of render props.