Introduction to Client-Side Routing

(https://github.com/learn-co-curriculum/react-hooks-react-router-intro-v6) (https://github.com/learn-co-curriculum/react-hooks-react-router-intro-v6/issues/new)

Learning Goals

- Explain how client-side routing works and how it differs from server-side routing.
- Use the Location and History APIs in JavaScript.

Introduction

We have learned about building components, changing state, passing props, and even interacting with APIs. We have one last major feature to talk about when it comes to making **single-page applications**: how can we separate out our components onto different "pages", each with their own unique URL? This is where **client-side** routing comes in.

For the majority of applications that aren't single-page applications, **routing** describes the following process:

- A user clicks on a link
- That link has its own distinct URL (`/contact`)
- The browser makes a **GET request** to the server for the content at that URL
- The server sends a **response** with a new HTML document for the `/contact` page

With **client-side routing**, the process will look different for one very important reason: the goal of **client-side routing** is to handle all the routing logic with JavaScript, without making any additional GET requests for some new HTML document. Remember, we have a **single-page application**, so there is by definition only one HTML document for our entire application!

Lets say that our **client-side** app is going to have these routes:

- `/movies`
- `/about`
- `/login`

Our React server's only job is to render the HTML, which will look similar to this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Movie Maker 3000</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  </head>
  <body>
    <div id="root"></div>
    <script src="./bundle.js" charset="utf-8"></script>
  </body>
</html>
```

With **client-side** routing, the server is not responsible for handling the routing, fetching and displaying of the data in the browser. These things are the responsibility of the **client-side code** instead.

In the example above, we'll want separate pages for `/movies` , `/login` , and `/about` . With client-side routing, you might get all the data necessary to render all three pages on the first page load. Then, when a user clicks around your site, the client-side router swaps the 'movies page' component with the 'about page' component and renders faster than it would if you were requesting each separate page from a server.

Client-side routing brings with it some great benefits. The major one is *speed*. Since we are only making one request to the server, we don't have to wait for a round trip server call for each page change. We have everything stored on the client side already, so we just notify our client-side code to display the info as we need it.

Single-Page Applications (SPAs)

Create React App was designed to build single-page applications. This means we won't require multiple pages to be loaded from the server, just the original **GET** request with our initial HTML, CSS and JS files. This requires us to figure out how to make the experience of client-side routing work to our advantage.

There are a few things we need to take into consideration:

- We want to make sure that we have a URL that displays what the user is doing at that moment. So if they are viewing a bio page it might look like `https://worlds-best-app.com/bio` instead of `https://worlds-best-app.com` .
- We want a user to be able to use the browser's back and forward buttons, as well as the browser history, with ease.
- We want a user to be able to input a URL into the address bar and navigate to the view they need to see.

This is easy with server-side routing, since this is the way the web has worked since its inception; with client-side routing, we'll need a few tricks to emulate this behavior.

Limits of Client-Side Routing

So this all sounds great, but what are the limitations?

- **Loading of CSS & Javascript**: Since we are now loading all of our CSS and Javascript on the initial **GET** request it can take a while to load our first page. This can be important as the first page load can take a long time if you have a huge application.
- **Analytics**: Analytic tools normally track page views, but an SPA doesn't have pages in the traditional sense, so this makes it harder for analytical tools to track page views. We will need to add extra scripts to handle this limitation.
- Single-page applications with client-side routing are harder to design than traditional multi-page applications.

React Router

The most popular client-side routing library to use with React is **React Router (https://reactrouter.com/en/main)** .

React Router has a lot of great features, but at its core, the two key things it enables are:

- Conditional rendering of components based on the URL: when the URL is `/movies` , the `<Movie>` component is displayed
- Programmatic navigation using JavaScript: when a link to the Movie page is clicked, the URL changes to `/movies` and the content is updated without making a request for a new HTML document

All of the features of React Router build on top of features that are already built into JavaScript via different web APIs, primarily the **Location (https://developer.mozilla.org/en-US/docs/Web/API/Location)** and **History (https://developer.mozilla.org/en-US/docs/Web/API/History_API)** APIs. Below, we'll briefly explain the purpose of each. You won't be interacting with these APIs directly going forward (that's the job of React Router), but knowing how they work will help you better understand how React Router does its job.

The Location API

You can access the location in the URL bar from any website by typing this in the console in the browser's dev tools:

```
window.location;
```

This will return a `Location` object with all kinds of useful information, including the `pathname`. For example, the Location object for `http://localhost:3000/movies` has the following properties:

- `origin`: `"http://localhost:3000"`
- `pathname`: `"/movies"`
- `protocol`: `"http:"`

If we were designing client-side routing ourselves *without* React Router, the `pathname` in particular would be useful for associating a component with a "page" in our application:

```
function App() {
  let currentPage;
  if (window.location.pathname === "/movies") {
    currentPage = <Movies />;
  } else if (window.location.pathname === "/about") {
    currentPage = <About />;
  } else {
    currentPage = <h2>404 not found</h2>;
  }

  return (
    <div>
      <h1>Movie Maker 3000</h1>
      {currentPage}
    </div>
    );
  }
```

React Router has a much more elegant way of handling this routing logic, as we'll see in coming lessons. But at a basic level, having some conditional rendering based on the URL is key to any client-side routing solution.

The History API

Whenever we load a new page in the browser, that information is saved in browser history. Go to the JavaScript console in the browser and type:

```
window.history;
```

This should return the following code:

```
{ length: 32, state: null, scrollRestoration: "auto" };
```

The length is how many locations you have visited in this window session.

If you type the following code, it will take you to the last location in your browser history:

```
window.history.back();
```

Go ahead and try it out!

That is the JavaScript equivalent of using the back button in the browser toolbar. You can also move forward using `window.history.forward()` .

With JavaScript's History API, we also have the ability to use the `window.history.pushState()` method to programmatically navigate to a new page. This method takes in three parameters:

- `state` : This is a plain JavaScript object that is associated with the new history entry we are going to create with the **pushState()** function.
- `title` : This is currently ignored by most browsers and it is safe to just pass an empty string here.
- `url` : This is the URL for the new history entry. The browser will not attempt to load this URL after it calls pushState.

Try programmatically navigating to a new URL in your browser by running this code in the console:

```
const newState = {
  goal: "Learn about pushState()",
};

window.history.pushState(newState, "new state", "new-state");
```

You should notice that your browser has now changed to show `new-state` at the end of your URL address. Go ahead and type:

```
window.history.state;
```

It should return:

```
{
  goal: "Learn about pushState()";
}
```

If you now use the `window.history.back()` function you will not go back to the previous page, but your URL address will return to the original URL address. If you use `window.history.forward()` you will move back to our new URL that ends in **new-state**.

In a React application, we could use the History API to change the default behavior of an `<a>` tag to use `pushState` instead of making a GET request to the provided URL:

```
function NavBar() {
  function navigate(e) {
    // don't make a GET request
    e.preventDefault();
    // use pushState to navigate using the href attribute of the <a> tag
    window.history.pushState(null, "", e.target.href);
  }

  return (
    <nav>
```

```
        <a href="/movies" onClick={navigate}>
          Movies
        </a>
        <a href="/about" onClick={navigate}>
          About
        </a>
        <a href="/login" onClick={navigate}>
          Login
        </a>
      </nav>
    );
  }
```

React Router provides similar functionality with its own custom version of the History API that lets it listen for changes to the URL and re-render your application with the correct component based on the new URL.

Conclusion

Client-side routing is an important feature to make our single-page applications feel like real websites by changing what the user sees on the page based on the URL. It also comes with the benefit of speed since the browser doesn't have to re-load the entire HTML document and all the JavaScript/CSS files associated with it for each request.

Thanks to browser features like the Location and History APIs, JavaScript has all the tools needed to enable client-side routing. In the coming lessons, we'll see how to use React Router to make it easy to take advantage of these features in our React applications.

Resources

- **React Router (https://reactrouter.com/en/main)**
- **Manipulating Browser History (https://developer.mozilla.org/en-US/docs/Web/API/History_API)**