React Router

Prior Knowledge

- Understand the parent-child relationship of html elements.
- Understand that React is a Single Page Application (SPA).
- Understand the lifecycle of `useEffect` hooks.
- Understand how html documents are served on different routes using server-side rendering.

Learning objectives

- Understand how React can emulate different pages using conditional rendering.
- Understand how the React Router library implements this functionality.
- Understand how to plan React applications with several routes.

Single page applications vs multi-page applications

React applications are generally **single page applications** (SPAs) - as all the JavaScript is sent (though not necessarily all the data) required to construct the whole website, instead of individual HTML pages. Essentially, it does not matter what the path of the URL is on an SPA, as everything is sent on the home path anyway. There are advantages to this, but there are two major drawbacks to losing this routing behaviour:

- Different areas of our application are not individually addressable (preventing, for example, sharing links)
- Losing use of the browser's history (back and forward navigation)

React Router is a library that allows **client-side routing**, as opposed to **server-side routing**:

Server-side routing

- The server has views for every single route of our app.
- User navigates to `/about` , the browser sends a GET request to `/about` and our server responds with the corresponding view.

Client-side routing

- In React applications, the server provides a single HTML file (with an empty div) and a bundle of JavaScript. The rendering of the application happens client-side.
- Because all the views of our application are already in the browser, we don't need to make GET requests to get different views.
- We can use a router, a library that catches the changes in the URL and renders different components accordingly.
- HTTP requests still happen in the background, but not for displaying views, just for getting or sending data to servers.
- The page never reloads, components come and go from the DOM, giving the illusion of navigation.

Using React Router

React router can be installed via npm

```
npm i react-router-dom
```

BrowserRouter

React router can be used in the browser or native applications. As we are making web apps we will use the `BrowserRouter` component. This component must be wrapped around our entire App. We can then make use of other components from React router to render our components on specific paths.

```
// src/main.jsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import { BrowserRouter } from 'react-router-dom';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
);
```

Routes and Route

A `Routes` component takes a number of `Route` components as children. It will render the `Route` that matches the current path.

A Route component takes a prop of `path` which will be compared to the current url.

```
// src/App.jsx
import { Routes, Route } from 'react-router-dom';

const App = () => {
  return (
    <div className="App">
      <h1>My App </h1>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/animals" element={<Animals />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </div>
  );
};

// src/components/Home.jsx
const Home = () => {
  return (
    <div>
      <h2>Home</h2>
    </div>
  );
};

// src/components/About.jsx
const About = () => {
  return (
    <div>
      <h2>About</h2>
    </div>
  );
};

// src/components/Animals.jsx
const Animals = () => {
  return (
    <div>
      <h2>Animals</h2>
    </div>
  );
};
```

Links

Though typing in a new URL will take you to the correct place if you have routing set up, you will still be reloading the entire application, which defeats one of the purposes of React and single page applications. **Anchor tags** (`<a />`) are normally used for linking between pages, but the default behaviour of these is also to load a new page.

React router provides a useful `<Link>` component that removes this behaviour for us. It takes a `to` property that will set the url. It could be useful in a `NavBar` component, for example:

```
<nav>
  <Link to="/">Home</Link>
  <Link to="/about">About</Link>
  <Link to="/animals">Animals</Link>
</nav>
```

nb Just as with `a` tags, Links can take more than just text as children. You can wrap a Link component around any number of children.

```
<ul>
  <li>
    <Link to="/animals/bear">
      <h2>Bears</h2>
      <img src="http://example.com/bear.jpg" alt="bear in the forest" />
      <p>Tagline about bears</p>
    </Link>
  </li>
</ul>
```

Parametric endpoints - useParams

Just as when writing servers we can declare routes to include parametric endpoints. The syntax for a parametric endpoint is the same as in express, we use a `':'` in our path, followed by the parameters name. e.g. `path="/animals/:species_name"`

React router will parse the current url and extract the values of any params for us. These are accessible on the (match.params) prop from our Route components. Passing these props around can be overly complicated and do we are taking advantage of React hooks we can use the provided `useParams hook` to access the values of our params.

```
// src/App.jsx
import { Routes, Route } from 'react-router-dom';

const App = () => {
  return (
    <div className="App">
      <h1>My App </h1>
      <Routes>
        <Route path="/animals" element={<Animals />} />
        <Route path="/animals/:species_name" element={<SingleAnimal />} />
      </Routes>
    </div>
  );
};

// src/components/Animals.jsx
const Animals = () => {
  return (
    <h2>Animals</h2>
  );
};

// src/components/SingleAnimal.jsx
import { useParams } from 'react-router-dom';

const SingleAnimal = () => {
  const { species_name } = useParams(); // useParams returns an object containing the url params
  return (
    <div>
      <h2>Rendering {species_name}</h2>
    </div>
  );
};
```

A common pattern is to use the value from the params to fetch data from an api. In the above example we could use the `{species_name}` to fetch the rest of the data from an api. If the `{species_name}` is changed, by the user navigating to a new animal's page for example, the component will be re-rendered with the new `{species_name}`. (If your interested in how to implement this kind of functionality it is covered in the context api and custom hooks sections of these notes.) We should bear this in mind when writing our side effects and make sure that our dependencies are correct. In this case, if the `{species_name}` changes, re-run the effect and fetch new data.

```
// src/components/SingleAnimal.jsx
import { useState, useEffect } from 'react';
import { useParams } from 'react-router-dom';
import { fetchAnimalBySpecies } from '../utils/api';

const SingleAnimal = () => {
  const [animal, setAnimal] = useState({});
  const { species_name } = useParams();

  useEffect(() => {
    fetchAnimalBySpecies(species_name).then((animalData) => {
      setAnimal(animalData);
    });
  }, [species_name]);

  return (
    <div>
      <h2>Rendering info about {species_name}</h2>
      {/* here data rendered here... */}
    </div>
  );
};

// ./utils/api.js
import axios from 'axios';

const myApi = axios.create({
  baseURL: 'https://myExampleServer.com/api',
});

export const fetchAnimalBySpecies = (species_name) => {
  return myApi.get(`/species/${species_name}`).then((res) => {
    return res.data.topic;
  });
};
```

The functionality of making an api call is not "component specific" as it isn't tied to any one component and its implementation. It is a good idea to separate our concerns and define the logic of making these calls in their own module (a different file) so that they can be reused, mocked out or refactored easily.

These are some of the core features of React Router and the docs offer much more with excellent examples and guides on how to implement common routing techniques such as nested routing, styling active links, redirects and much more. See the docs for more info.

Queries - useSearchParams

Another common feature is to add queries to the end of our endpoints to provide additional functionality. Common use cases are search strings or optional filtering options. React router will also parse any queries and make them available via the useSearchParams hook.

Using the above example if the client made a request to `/animals?sort_by=average_weight` we can access the queries, called `{sort_by:'average'}` in this case.

nb queries are optional and adding to an endpoint doesn't change the path so our route is still `<Route path="/animals" element={<Animals />} />`.

```
// src/components/Animals.jsx
import { useSearchParams } from 'react-router-dom';

const Topics = () => {
  const [searchParams, setSearchParams] = useSearchParams();
  console.log(searchParams); // URLSearchParams {}

  const sortbyQuery = searchParams.get("sort_by"); // 'average_weight'

  return (
    <section>
      <h2>Topics</h2>
    </section>
  );
};
```

The value of `searchParams` is an instance of the browsers `URLSearchParams` constructor. This is an object containing methods such as `.get` which will return the value of a passed query key. If the query does not exist then this method will return `null`.

Updating Queries

We often want to provide links or buttons to change the queries programmatically. Links can be hard coded to contain the queries as part of the string. e.g.

```
<Link to="/animals?sort_by=average_weight">Animals by weight</Link>
```

A more complicated use case is to add or change existing queries on a page that we are already on. This is common when adding filtering options, toggles etc as we want to preserve any existing queries whilst updating one. The hook also returns a `setSearchParams()` function to update the queries programmatically if hard coding the string isn't sufficient.

```
// src/components/Animals.jsx
import { useEffect } from 'react';
import { useSearchParams } from 'react-router-dom';

const Animals = () => {
  const [searchParams, setSearchParams] = useSearchParams();
  const sortbyQuery = searchParams.get("sort_by") // 'average_weight'
  const orderQuery = searchParams.get('order'); // 'asc'

  const setSortOrder = (direction) => {
    // copy existing queries to avoid mutation
    const newParams = new URLSearchParams(searchParams);
    // set the order query
    newParams.set('order', direction);
    setSearchParams(newParams);
  };

  useEffect(() => {
    // fetch new data based on the queries
  }, [sortbyQuery, orderQuery])

  return (
    <section>
      <h2>Topics</h2>
      <button onClick={() => setSortOrder('asc')}>Ascending</button>
      <button onClick={() => setSortOrder('desc')}>Ascending</button>
    </section>
  );
};
```

Navigating Programmatically

`react-router-dom` gives access to a `useNavigate` hook which returns a function that lets you navigate programmatically, for example after a form is submitted.

The navigate function has two different argument types:

- A string, which represents the "to" value (the same as with `<Link to="/path">`)
- A number, which represents how many steps forwards (positive numbers) or backwards (negative numbers) you'd like to go in the browser's history (e.g. `navigate(-1)` is the same as clicking the back button)

Note: This is not a replacement for `<Link>` components!

e.g.

```
import { useNavigate } from 'react-router-dom';

function MyComponent() {
  const navigate = useNavigate();

  function handleSubmit(e) {
    e.preventDefault();
    doSomething()
      .then(() => {
        navigate('/somewhere/else');
      })
      .catch((err) => {
        // handle error
      });
  }

  return (
    <form onSubmit={handleSubmit}>
      {/* labels, inputs and fun things... */}
    </form>
  );
}
```