p3-react-router

3. Routing in React

> - **React for Dummies**, I.T.T. "Blaise Pascal" @ Cesena, IT
> - Written by: Nicholas Magi - nicholas.magi24[@]gmail.com

Table of contents

3.1 React makes S.P.A.

It's important to keep in mind that we are working with a **library**, and not a **framework**. Let's briefly recall the key difference: a library consists in a set of functions used by the developer to solve a particular problem encountered during the development process, while a framework gives a strict structure the programmer must follow to develop an application. The gap between the two concepts becomes sharper in the case of **routing**.

> ⓘ **Info**
>
> **Navigating through pages is not natively supported by React**
> and, in general, **having multiple pages neither**.

React allows to develop a **S**ingle **P**age **A**pplication (S.P.A.), an application displayed in a single page that contains all the components you developed. Thus we need to rely on a third-part system to implement this almost fundamental feature.

3.2 `react-router`: installation

Luckily for us there's a very popular library that implements routing: some time ago it was called `react-router-dom`, now developers shortened its name to `react-router`.

Inside your React application folder, open up a new terminal and run

```
npm i react-router
```

This will install the library and add it to the project dependencies. The official documentation (from which the information contained in this page comes from) is available at https://reactrouter.com/home.

3.3 How it works

> ✓ **From the official guide**
>
> "*[...] React Router is a simple and declarative library for routing. Its only job will be matching the URL to a set of components, providing access to URL data, and navigating around the app. [...]*"

It can be configured and used as a **Framework based on React**, but we won't consider this option (*it's probably a better alternative considering `Next.js` in this case*).

We split the navigation concept in two:

1. Routing — declaring how all the possible routes our application has looks like;
2. Navigating — creating the actual links.

This documentation will explain how to design a simple application made of few pages. Let's say we have `HomePage`, `AboutUs` and `Shop` and we want to be able to easily navigate through them via a `NavBar` component.

3.3.1 Routing

3.3.1.1 `BrowserRouter` placement

To accomplish step 1. cited on the previous paragraph, we need to locate a point to place a **Router** — which works as context provider for all the routes.

The most suitable location we can think of is a "high" place in the **UI Component Tree**, such as `App()` or, even better, `main.tsx`. Let's stick with the last one.

The default code of `main.tsx` (provided by the Vite template) looks something like this:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import App from './App.jsx'
import './index.css'

createRoot(document.getElementById('root')).render(
        <StrictMode>
                <App />
        </StrictMode>,
)
```

First of all, you need to import `BrowserRouter` in your code.

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import { BrowserRouter } from 'react-router'
import App from './App.jsx'
import './index.css'

createRoot(document.getElementById('root')).render(
        <StrictMode>
                <App />
        </StrictMode>,
)
```

Then we put the `BrowserRouter` component between `StrictMode` and `App`, like so:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import { BrowserRouter } from 'react-router'
import App from './App.jsx'
import './index.css'

createRoot(document.getElementById('root')).render(
        <StrictMode>
                <BrowserRouter>
                        <App />
                </BrowserRouter>
        </StrictMode>,
)
```

In this way we accomplish our goal: every node in the UI Tree descending from `BrowserRouter` can receive information from it. `BrowserRouter` plays as **Context provider** for the application, and that means we are free to choose a place in our code to design the possible routes we can navigate through.

We could define our Routes replacing the default `App` component, or keep it and work inside it. We'll stick with the last option.

3.3.1.2 `Routes` definition

Let's say we have erased all the template code of `App` provided at the project configuration stage of our React app. `App.jsx` is just a component declaration:

```
export default function App() {
        return <></>
}
```

As always, we need to import the stuff we need at the moment. We want to use `Routes` and `Route` and, like we did before, we'll write:

```
import { Routes, Route } from 'react-router'

export default function App() {
        return <></>
}
```

Now the crucial part: we need to list all the routes, each represented by a `Route` component. The list is nested inside a wrapper component, `Routes`, as follows:

```
import { Routes, Route } from 'react-router'
import HomePage from '../pages/HomePage.jsx'
import AboutUs from '../pages/AboutUs.jsx'
```

```
import Shop from '../pages/Shop.jsx'
import NotFound from '../pages/NotFound.jsx'

export default function App() {
        return <Routes>
                <Route index element={<HomePage/>}/>
                <Route path='/about-us' element={<AboutUs/>}/>
                <Route path='/shop' element={<Shop/>}/>
                <Route path='/*' element={<NotFound/>}/>
        </Routes>
}
```

Few things worth of mention:

1. Each `Route` has a particular set of attributes. The basic ones are `path` and `element`. The second defines the element (either a component or simple JSX) the `Route` leads to, while the first assigns it an **unique** string (that will be the one to appear in the address bar of our browser).
2. We put the attribute `index` to the `Route` that corresponds to the `HomePage`. In this case we can omit the `path` attribute, since the root path will always lead to whatever specified in the `element` one.
3. Observe the last `Route`: the wildcard `*` used in the `path` attribute matches everything not matched by the previous routes. It can be used, as shown in the example, to render a `NotFound` page — in order to notify the user the page he was looking for is not available.

And for the first part we're almost done! We successfully designed our routes.

3.3.1.3 Nested Routes

This, however, is a toy example. We may want to have more complex routes structure. For instance, consider the `Shop` page. It's pretty easy to imagine and common to see that it will have a grid of available products, and each product will provide a link to its own details page. We want to design an unique detail page, which we'll call `ItemPage`, that changes dynamically its content — depending on the product the user chose to view.

The routing structure needs to be changed:

```
import { Routes, Route } from 'react-router'
import HomePage from '../pages/HomePage.jsx'
import AboutUs from '../pages/AboutUs.jsx'
import Shop from '../pages/Shop.jsx'
import NotFound from '../pages/NotFound.jsx'
import ItemPage from '../pages/ItemPage.jsx'

export default function App() {
        return <Routes>
```

```
                <Route index element={<HomePage/>}/>
                <Route path='/about-us' element={<AboutUs/>}/>
                <Route path='shop'>
                        <Route index element={<Shop/>}/>
                        <Route path=':itemId' element={<ItemPage/>}/>
                </Route>
                <Route path='/*' element={<NotFound/>}/>
        </Routes>
}
```

Now we added a nesting level to `Shop`'s route, which can be interpreted as:

- the index page of `/shop` path is the general page `Shop`;
- the specific product details are rendered by putting the product id after the relative index path (e.g. `/shop/1`, `/shop/4325` ecc..); each id will render the `ItemPage` page.

We need a way to tell `ItemPage` that the user wants to display the item with id 1, 4325 ecc...

The solution to this problem lives in the page itself:

```
import Layout from "../components/Layout";
import { useParams } from "react-router";

export default function ItemPage() {
        const { itemId } = useParams()
        /* Or, equivantely (without "object destructuring")
        const urlParams = useParams()
        */
        return <Layout>
                <section>
                        <h3 className="display-5 fw-bold">Item #{itemId}
</h3>
                        {/* or, equivantely (without "object
destructuring")
                                <h3 className="display-5 fw-bold">
                                        Item #{urlParams.itemId}
                                </h3>
                        */}
                        <h1 className="display-1">Item Page</h1>
                </section>
        </Layout>
}
```

`useParams` is a custom hook supplied by the `react-router` library, which can be used to read the dynamic parameters written in the URL bar. This can go inside a `fetch()` request to retrieve data from an external server and display it inside our page.

3.3.2 Navigating

Until now we are only able to navigate our application by manually changing the URL. An average user would not want that (neither he could be able to), so we need to provide an user-friendly way to simplify the navigation.

`react-router` comes with 3 possibilities (actually you can consider 2 of them pretty similar):

1. **NavLink**
2. **Link**
3. `useNavigate()`

The official documentation provides enough information about them, so here's the link: https://reactrouter.com/start/library/navigating.