Tutorial

Introduction

Check out the completed version of the app here.

React Router is a fully-featured client and server-side routing library for React, a JavaScript library for building user interfaces. React Router runs anywhere React runs; on the web, on the server with node.js, and on React Native.

If you're just getting started with React generally, we recommend you follow the excellent Getting Started guide in the official docs. There is plenty of information there to get you up and running. React Router is compatible with React >= 16.8.

We'll keep this tutorial quick and to the point. By the end you'll know the APIs you deal with day-to-day with React Router. After that, you can dig into some of the other docs to get a deeper understanding.

While building a little bookkeeping app we'll cover:

- Configuring Routes

- Navigating with Link
- Creating Links with active styling
- Using Nested Routes for Layout
- Navigating programmatically
- Using URL params for data loading
- Using URL Search params
- Creating your own behaviors through composition
- Server Rendering

Installation

Recommended: StackBlitz

To do this tutorial you'll need a working React app. We recommend skipping bundlers and using this demo on StackBlitz to code along in your browser:

Open in StackBlitz

As you edit files, the tutorial will update live.

Using a bundler

Feel free to use your bundler of choice like Create React App or Vite.

```
# create react app
npx create-react-app router-tutorial

# vite
npm init vite@latest router-tutorial --template react
```

Then install React Router dependencies:

```
cd router-tutorial
npm install react-router-dom@6
```

Then edit your App.js to be pretty boring:

src/App.js

```
export default function App() {
  return (
    <div>
      <h1>Bookkeeper!</h1>
    </div>
  );
}
```

Actually, that "!" doesn't look boring at all. This is pretty exciting. We sat on React Router v6 beta for over a year as we shifted gears with our business after a global pandemic. THIS IS THE MOST EXCITING THING WE'VE DONE IN A WHILE!

Finally, go make sure `index.js` or `main.jsx` (depending on the bundler you used) is actually boring:

src/main.jsx

```
import { render } from "react-dom";
import App from "./App";
```

```
const rootElement = document.getElementById("root");
render(<App />, rootElement);
```

Finally, start your app:

```
# probably this
npm start

# or this
npm run dev
```

Connect the URL

First things first, we want to connect your app to the browser's URL: import `BrowserRouter` and render it around your whole app.

```
src/main.jsx

import { render } from "react-dom";
import { BrowserRouter } from "react-router-dom";
import App from "./App";

const rootElement = document.getElementById("root");
render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  rootElement
);
```

Nothing changes in your app, but now we're ready to start messing with the URL.

Add Some Links

Open up `src/App.js`, import `Link` and add some global navigation. Side note: don't take the styling too seriously in this tutorial, we're just using inline styles for convenience, style your apps however you want.

src/App.js

```
import { Link } from "react-router-dom";

export default function App() {
  return (
    <div>
      <h1>Bookkeeper</h1>
      <nav
        style={{
          borderBottom: "solid 1px",
          paddingBottom: "1rem"
        }}
      >
        <Link to="/invoices">Invoices</Link> |{" "}
        <Link to="/expenses">Expenses</Link>
      </nav>
    </div>
  );
}
```

Go ahead and click the links and the back/forward button (if you're using StackBlitz, you'll need to click the "Open in New Window" button in the inline-browser's toolbar). React Router is now controlling the URL!

We don't have any routes that render when the URL changes yet, but Link is changing the URL without causing a full page reload.

Add Some Routes

Add a couple new files:

- `src/routes/invoices.jsx`
- `src/routes/expenses.jsx`

(The location of the files doesn't matter, but when you decide you'd like an automatic backend API, server rendering, code splitting bundler and more for this app, naming your files like this way makes it easy to port this app to our other project, Remix 😉)

Now fill 'em up with some code:

src/routes/expenses.jsx

```jsx
export default function Expenses() {
  return (
    <main style={{ padding: "1rem 0" }}>
      <h2>Expenses</h2>
    </main>
  );
}
```

src/routes/invoices.jsx

```jsx
export default function Invoices() {
  return (
    <main style={{ padding: "1rem 0" }}>
      <h2>Invoices</h2>
    </main>
  );
}
```

Finally, let's teach React Router how to render our app at different URLs by creating our first "Route Config" inside of `main.jsx` or `index.js`.

src/main.jsx

```jsx
import { render } from "react-dom";
import {
  BrowserRouter,
  Routes,
  Route
} from "react-router-dom";
import App from "./App";
import Expenses from "./routes/expenses";
import Invoices from "./routes/invoices";

const rootElement = document.getElementById("root");
render(
  <BrowserRouter>
    <Routes>
      <Route path="/" element={<App />} />
      <Route path="expenses" element={<Expenses />} />
      <Route path="invoices" element={<Invoices />} />
    </Routes>
  </BrowserRouter>,
  rootElement
);
```

Notice at `"/"` it renders `<App>`. At `"/invoices"` it render `<Invoices>`. Nice work!

Remember if you're using StackBlitz to click the "Open in New Window" button in the inline browser's toolbar to be able to click the back/forward buttons in your browser.

Nested Routes

You may have noticed when clicking the links that the layout in `App` disappears. Repeating shared layouts is a pain in the neck. We've learned that most UI is a series of nested layouts that almost always map to segments of the URL so this idea is baked right in to React Router.

Let's get some automatic, persistent layout handling by doing just two things:

1. Nest the routes inside of the App route
2. Render an Outlet

First let's nest the routes. Right now the expenses and invoices routes are siblings to the app, we want to make them *children* of the app route:

src/main.jsx

```
import { render } from "react-dom";
import {
```

```jsx
  BrowserRouter,
  Routes,
  Route
} from "react-router-dom";
import App from "./App";
import Expenses from "./routes/expenses";
import Invoices from "./routes/invoices";

const rootElement = document.getElementById("root");
render(
  <BrowserRouter>
    <Routes>
      <Route path="/" element={<App />}>
        <Route path="expenses" element={<Expenses />} />
        <Route path="invoices" element={<Invoices />} />
      </Route>
    </Routes>
  </BrowserRouter>,
  rootElement
);
```

When routes have children it does two things:

1. It nests the URLs (`"/" + "expenses"` and `"/" + "invoices"`)
2. It will nest the UI components for shared layout when the child route matches:

However, before (2) will work we need to render an `Outlet` in the `App.jsx` "parent" route.

```
src/App.jsx
```

```
import { Outlet, Link } from "react-router-dom";

export default function App() {
  return (
    <div>
      <h1>Bookkeeper</h1>
      <nav
        style={{
          borderBottom: "solid 1px",
          paddingBottom: "1rem"
        }}
      >
        <Link to="/invoices">Invoices</Link> |{" "}
        <Link to="/expenses">Expenses</Link>
      </nav>
      <Outlet />
    </div>
  );
}
```

Now click around again. The parent route (`App.js`) persists while the `<Outlet>` swaps between the two child routes (`<Invoices>` and `<Expenses>`)!

As we'll see later, this works at *any level* of the route hierarchy and is incredibly powerful.

Listing the Invoices

Normally you'd be fetching data from a server somewhere, but for this tutorial let's hard code some fake stuff so we can focus on routing.

Make a file at `src/data.js` and copy/paste this in there:

src/data.js

```js
let invoices = [
  {
    name: "Santa Monica",
    number: 1995,
    amount: "$10,800",
    due: "12/05/1995"
  },
  {
    name: "Stankonia",
    number: 2000,
    amount: "$8,000",
    due: "10/31/2000"
  },
  {
    name: "Ocean Avenue",
    number: 2003,
    amount: "$9,500",
    due: "07/22/2003"
  },
  {
    name: "Tubthumper",
    number: 1997,
    amount: "$14,000",
    due: "09/01/1997"
  },
  {
    name: "Wide Open Spaces",
    number: 1998,
    amount: "$4,600",
    due: "01/27/2998"
  }
];

export function getInvoices() {
  return invoices;
}
```

Now we can use it in the invoices route. Let's also add a bit of styling to get a sidebar nav layout going on. Feel free to copy/paste all of this, but take special note of the `<Link>` elements `to` prop:

src/routes/invoices.jsx

```
import { Link } from "react-router-dom";
import { getInvoices } from "../data";

export default function Invoices() {
  let invoices = getInvoices();
  return (
    <div style={{ display: "flex" }}>
      <nav
        style={{
          borderRight: "solid 1px",
          padding: "1rem"
        }}
      >
        {invoices.map(invoice => (
          <Link
```

```
            style={{ display: "block", margin: "1rem 0" }}
            to={`/invoices/${invoice.number}`}
            key={invoice.number}
          >
            {invoice.name}
          </Link>
        ))}
      </nav>
    </div>
  );
}
```

Cool! Now click an invoice link and see what happens.

😨😨😨

Adding a "No Match" Route

That didn't go as you might have expected. If you click those links the page goes blank! That's because none of the routes we've defined match a URL like the ones we're linking to: `"/invoices/123"`.

Before we move on, it's good practice to always handle this "no match" case. Go back to your route config and add this:

src/main.jsx

```
<Routes>
  <Route path="/" element={<App />}>
    <Route path="expenses" element={<Expenses />} />
    <Route path="invoices" element={<Invoices />} />
    <Route
      path="*"
      element={
        <main style={{ padding: "1rem" }}>
          <p>There's nothing here!</p>
        </main>
      }
    />
  </Route>
</Routes>
```

The `"*"` has special meaning here. It will match only when no other routes do.

Reading URL Params

Alright, back to the individual invoice URLs. Let's add a route for a specific invoice. We just visited some URLs like `"/invoices/1998"` and `"/invoices/2005"`, let's make a new component at `src/routes/invoice.jsx` to render at those URLs:

```
src/routes/invoice.jsx

export default function Invoice() {
  return <h2>Invoice #???</h2>;
}
```

We'd like to render the invoice number instead of `"???"`. Normally in React you'd pass this as a prop: `<Invoice invoiceId="123" />`, but you don't control that information because it comes from the URL.

Let's define a route that will match these kinds of URLs and enable us to get the invoice number from it.

Create a new `<Route>` *inside* of the "invoices" route like this:

src/main.jsx

```
<Routes>
  <Route path="/" element={<App />}>
    <Route path="expenses" element={<Expenses />} />
    <Route path="invoices" element={<Invoices />}>
      <Route path=":invoiceId" element={<Invoice />} />
    </Route>
    <Route
      path="*"
      element={
        <main style={{ padding: "1rem" }}>
          <p>There's nothing here!</p>
        </main>
      }
    />
  </Route>
</Routes>
```

A couple things to note:

- We just created a route that matches urls like "/invoices/2005" and "/invoices/1998". The `:invoiceId` part of the path is a "URL param", meaning it can match any value as long as the pattern is the same.

- The `<Route>` adds a second layer of route nesting when it matches: `<App><Invoices><Invoice /></Invoices></App>`. Because the `<Route>` is nested the UI will be nested too.

Alright, now go click a link to an invoice, note that the URL changes but the new invoice component doesn't show up yet. Do you know why?

That's right! We need to add an outlet to the parent layout route (we're really proud of you).

src/routes/invoices.jsx

```jsx
import { Link, Outlet } from "react-router-dom";
import { getInvoices } from "../data";

export default function Invoices() {
  let invoices = getInvoices();
  return (
    <div style={{ display: "flex" }}>
      <nav
        style={{
          borderRight: "solid 1px",
          padding: "1rem"
        }}
      >
        {invoices.map(invoice => (
          <Link
            style={{ display: "block", margin: "1rem 0" }}
            to={`/invoices/${invoice.number}`}
            key={invoice.number}
          >
            {invoice.name}
          </Link>
        ))}
      </nav>
      <Outlet />
    </div>
  );
}
```

Okay, let's close the circle here. Open up the invoice component again and let's get the `:invoiceId` param from the URL:

src/routes/invoice.jsx

```
import { useParams } from "react-router-dom";

export default function Invoice() {
  let params = useParams();
  return <h2>Invoice: {params.invoiceId}</h2>;
}
```

Note that the key of the param on the `params` object is the same as the dynamic segment in the route path:

```
:invoiceId -> params.invoiceId
```

Let's use that information to build up a more interesting invoice page. Open up `src/data.js` and add a new function to lookup invoices by their number:

src/data.js

```
// ...

export function getInvoices() {
  return invoices;
}

export function getInvoice(number) {
  return invoices.find(
    invoice => invoice.number === number
  );
}
```

And now back in `invoice.jsx` we use the param to look up an invoice and display more information:

routes/invoice.jsx

```
import { useParams } from "react-router-dom";
import { getInvoice } from "../data";

export default function Invoice() {
  let params = useParams();
  let invoice = getInvoice(parseInt(params.invoiceId, 10));
  return (
    <main style={{ padding: "1rem" }}>
      <h2>Total Due: {invoice.amount}</h2>
      <p>
        {invoice.name}: {invoice.number}
      </p>
      <p>Due Date: {invoice.due}</p>
    </main>
  );
}
```

Note that we used `parseInt` around the param. It's very common for your data lookups to use a `number` type, but URL params are always `string`.

Index Routes

Index routes are possibly the most difficult concept in React Router for people to understand. So if you've struggled before, we hope this can clarify it for you.

Right now you're probably looking at one of the invoices. Click on the "Invoices" link in the global nav of your app. Notice that the main content area goes blank! We can fix this with an "index" route.

src/main.jsx

```jsx
<Routes>
  <Route path="/" element={<App />}>
    <Route path="expenses" element={<Expenses />} />
    <Route path="invoices" element={<Invoices />}>
      <Route
        index
        element={
          <main style={{ padding: "1rem" }}>
            <p>Select an invoice</p>
          </main>
        }
      />
      <Route path=":invoiceId" element={<Invoice />} />
    </Route>
    <Route
      path="*"
      element={
        <main style={{ padding: "1rem" }}>
          <p>There's nothing here!</p>
        </main>
      }
    />
  </Route>
</Routes>
```

Sweet! Now the index route fills the empty space!

Notice it has the `index` prop instead of a `path`. That's because the index route shares the path of the parent. That's the whole point--it doesn't have a path.

Maybe you're still scratching your head. There are a few ways we try to answer the question "what is an index route?". Hopefully one of these sticks for you:

- Index routes render in the parent routes outlet at the parent route's path.
- Index routes match when a parent route matches but none of the other children match.
- Index routes are the default child route for a parent route.
- Index routes render when the user hasn't clicked one of the items in a navigation list yet.

Active Links

It's very common, especially in navigation lists, to display the link as the active link the user is looking at. Let's add this treatment to our invoices list by swapping out `Link` for `NavLink`.

src/routes/invoices.jsx

```jsx
import { NavLink, Outlet } from "react-router-dom";
import { getInvoices } from "../data";

export default function Invoices() {
  let invoices = getInvoices();
  return (
    <div style={{ display: "flex" }}>
      <nav
        style={{
          borderRight: "solid 1px",
          padding: "1rem"
        }}
      >
        {invoices.map(invoice => (
          <NavLink
            style={({ isActive }) => {
              return {
                display: "block",
                margin: "1rem 0",
                color: isActive ? "red" : ""
              };
            }}
            to={`/invoices/${invoice.number}`}
            key={invoice.number}
          >
            {invoice.name}
          </NavLink>
        ))}
      </nav>
      <Outlet />
    </div>
  );
}
```

We did three things there:

1. We swapped out `Link` for `NavLink`.
2. We changed the `style` from a simple object to a function that returns an object.
3. We changed the color of our link by looking at the `isActive` value that `NavLink` passed to our styling function.

You can do the same thing with `className` on `NavLink`:

```
// normal string
<NavLink className="red" />

// function
<NavLink className={({ isActive }) => isActive ? "red" : "blue"} />
```

Search Params

Search params are like URL params but they sit in a different position in the URL. Instead of being in the normal URL segments separated by `/`, they are at the end after a `?`. You've seen them across the web like `"/login?success=1"` or `"/shoes?brand=nike&sort=asc&sortby=price"`.

React Router makes it easy to read and manipulate the search params with `useSearchParams`. It works a lot like `React.useState()` but stores and sets the state in the URL search params instead of in memory.

Let's see it in action by adding a little filter on the invoices nav list.

routes/invoices.jsx

```jsx
import {
  NavLink,
  Outlet,
  useSearchParams
} from "react-router-dom";
import { getInvoices } from "../data";

export default function Invoices() {
  let invoices = getInvoices();
  let [searchParams, setSearchParams] = useSearchParams();

  return (
    <div style={{ display: "flex" }}>
      <nav
        style={{
          borderRight: "solid 1px",
          padding: "1rem"
        }}
      >
        <input
          value={searchParams.get("filter") || ""}
          onChange={event => {
            let filter = event.target.value;
            if (filter) {
              setSearchParams({ filter });
            } else {
              setSearchParams({});
            }
          }}
        />
        {invoices
          .filter(invoice => {
            let filter = searchParams.get("filter");
            if (!filter) return true;
            let name = invoice.name.toLowerCase();
            return name.startsWith(filter.toLowerCase());
          })
          .map(invoice => (
            <NavLink
              style={({ isActive }) => ({
                display: "block",
                margin: "1rem 0",
                color: isActive ? "red" : ""
              })}
              to={`/invoices/${invoice.number}`}
              key={invoice.number}
            >
              {invoice.name}
            </NavLink>
          ))}
      </nav>
      <Outlet />
```

```
    </div>
  );
}
```

Check this out, as the user types:

- `setSearchParams()` is putting the `?filter=...` search params in the URL and rerendering the router.
- `useSearchParams` is now returning a `URLSearchParams` with `"filter"` as one of its values.
- We set the value of the input to whatever is in the filter search param (it's just like `useState` but in the URLSearchParams instead!)
- We filter our list of invoices based on the filter search param.

Custom Behavior

If you filter the list and then click a link, you'll notice that the list is no longer filtered and the search param is cleared from the `<input>` and the URL. You might want this, you might not! Maybe you want to keep the list filtered and keep the param in the URL.

We can persist the query string when we click a link by adding it to the link's href. We'll do that by composing `NavLink` and `useLocation` from React Router into our own `QueryNavLink` (maybe there's a better name, but that's what we're going with today).

```
import { useLocation, NavLink } from "react-router-dom";

function QueryNavLink({ to, ...props }) {
  let location = useLocation();
  return <NavLink to={to + location.search} {...props} />;
}
```

You can put that code anywhere you want in your app and then replace your `NavLink` in `src/routes/invoices.jsx` with `QueryNavLink` and you're done.

Like `useSearchParams`, `useLocation` returns a location that tells us information about the URL. A location looks something like this:

```
{
  pathname: "/invoices",
  search: "?filter=sa",
  hash: "",
  state: null,
  key: "ae4cz2j"
}
```

With that information, the task in `QueryNavLink` is pretty simple: add the `location.search` onto the `to` prop. You might be thinking, "Geez, seems like this should be a built-in component of React Router or something?". Well, let's look at another example.

What if you had links like this on an ecommerce site.

```
<Link to="/shoes?brand=nike">Nike</Link>
<Link to="/shoes?brand=vans">Vans</Link>
```

And then you wanted to style them as "active" when the url search params match the brand? You could make a component that does exactly that pretty quickly with stuff you've learned in this tutorial:

```
function BrandLink({ brand, ...props }) {
  let [params] = useSearchParams();
  let isActive = params.getAll("brand").includes(brand);
  return (
    <Link
      style={{ color: isActive ? "red" : "" }}
      to={`/shoes?brand=${brand}`}
      {...props}
    />
  );
}
```

That's going to be active for `"/shoes?brand=nike"` as well as `"/shoes?brand=nike&brand=vans"`. Maybe you want it to be active when there's only one brand selected:

```
let brands = params.getAll("brand");
let isActive =
  brands.includes(brand) && brands.length === 1;
// ...
```

Or maybe you want the links to be *additive* (clicking Nike and then Vans adds both brands to the search params) instead of replacing the brand:

```
function BrandLink({ brand, ...props }) {
  let [params] = useSearchParams();
  let isActive = params.getAll("brand").includes(brand);
  if (!isActive) {
    params.append("brand", brand);
  }
  return (
    <Link
      style={{ color: isActive ? "red" : "" }}
      to={`/shoes?${params.toString()}`}
      {...props}
    />
  );
}
```

Or maybe you want it to add the brand if it's not there already and remove it if it's clicked again!

```
function BrandLink({ brand, ...props }) {
  let [params] = useSearchParams();
  let isActive = params.getAll("brand").includes(brand);
  if (!isActive) {
    params.append("brand", brand);
  } else {
    params = new URLSearchParams(
      Array.from(params).filter(
        ([key, value]) => key !== "brand" || value !== brand
      )
    );
  }
  return (
    <Link
      style={{ color: isActive ? "red" : "" }}
      to={`/shoes?${params.toString()}`}
      {...props}
    />
  );
}
```

As you can see, even in this fairly simple example there are a lot of valid behaviors you might want. React Router doesn't try to solve every use-case we've ever heard of directly. Instead, we give you the components and hooks to compose whatever behavior you need.

Navigating Programmatically

Okay, back to our app. Hang in there, you're almost done!

Most of the time the URL changes is in response to the user clicking a link. But sometimes you, the programmer, want to change the URL. A very common use case is after a data update like creating or deleting a record.

Let's add a button that marks the invoice as paid and then navigates to the index route.

First you can copy and paste this function that deletes an invoice from our fake data store:

```
src/data.js

export function deleteInvoice(number) {
  invoices = invoices.filter(
    invoice => invoice.number !== number
  );
}
```

Now let's add the delete button, call our new function, and navigate to the index route:

src/routes/invoice.jsx

```jsx
import { useParams, useNavigate } from "react-router-dom";
import { getInvoice, deleteInvoice } from "../data";

export default function Invoice() {
  let navigate = useNavigate();
  let params = useParams();
  let invoice = getInvoice(parseInt(params.invoiceId, 10));

  return (
    <main style={{ padding: "1rem" }}>
      <h2>Total Due: {invoice.amount}</h2>
      <p>
        {invoice.name}: {invoice.number}
      </p>
      <p>Due Date: {invoice.due}</p>
      <p>
        <button
          onClick={() => {
            deleteInvoice(invoice.number);
            navigate("/invoices");
          }}
        >
          Delete
        </button>
      </p>
    </main>
  );
}
```

Getting Help

Congrats! You're all done with this tutorial. We hope it helped you get your bearings with React Router.

If you're having trouble, check out the Resources page to get help. Good luck!