

Week 08: Client-side routing, React forms

Petr Wehrenberg

Outline

Routing

- Introduction to client-side routing
- Routes definition
- Links and NavLink
- Code splitting and lazy loading

Forms

- Controlled and uncontrolled inputs
- Basic form definition
- Form validation
- Form array fields

What is routing, and why do we need it?

UX problem

#1 Hey, look at these excellent products!

- <https://eshop.com>
- <https://eshop.com/products>

#2 Click and find all completed tasks!

- <https://todo.com>
- <https://todo.com/tasks?filter=status~eq~completed>

#3 Situation: The user clicks in the browser to go back, and nothing happens.

DX problem

How can we handle more pages in our app?

```
const App: FC = () => {
  const [currentPage, setCurrentPage] = useState("homepage");

  return (
    <div>
      <Navbar onClick={(page) => setCurrentPage(page)} />
      {currentPage === "homepage" && <Homepage />}
      {currentPage === "preferences" && <Preferences />}
      {currentPage === "products" && <Products />}
      {/* another pages */}
    </div>
  );
};
```

Very naive approach. Do not use this!

Client-side routing introduction

We know our goal and the naive approach, but how can we do that?

We need ...

- fake routing in the browser
- all features of the old-style routing (search params, dynamic routes...)
- simple API

Solution: React router DOM v6

Alternative: Tanstack Router

Routers in React router DOM

- Browser router
- Hash router
- Memory router

Please, use only the Browser router in your projects and applications.

React router DOM – Route Object

Routes definition with RouteObject

```
import { createBrowserRouter, RouterProvider } from "react-router-dom";

const router = createBrowserRouter([
  {
    path: "/preferences",
    Component: PreferencesPage,
  },
  {
    path: "/products",
    Component: ProductListPage,
  },
]);

const App: FC = () => {
  return <RouterProvider router={router} />;
};
```


React router DOM – Route Element

Also, possibility to define routes with elements:

```
import { BrowserRouter, Routes, Route } from "react-router-dom";

const App: FC = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/preferences" Component={PreferencesPage} />
        <Route path="/products" Component={ProductListPage} />
      </Routes>
    </BrowserRouter>
  );
};
```

Nested routes

- sometimes, we need to specify routes like `/preferences/users/managers`
- we want to write the same part on multiple pages only once
- we would like to see some nice structure

```
const router = createBrowserRouter([\n  { path: "/preferences/users", Component: UsersPreferences },\n  { path: "/preferences/users/managers", Component: ManagersPreferences },\n  { path: "/preferences/products", Component: ProductsPreferences },\n]);
```

Not like that.

Nested routes – Route Object

```
const router = createBrowserRouter([
  {
    path: "/preferences",
    Component: PreferencesLayout,
    children: [
      { path: "users", Component: UsersPreferences },
      { path: "products", Component: ProductsPreferences },
    ],
  },
]);
```

```
const PreferencesLayout: FC = () => (
  <div>
    <PreferencesNavbar />
    <Outlet /> { /* imported from React router dom */ }
  </div>
);
```

Nested routes – Route Elements

```
<BrowserRouter>
  <Routes>
    <Route path="/preferences" Component={PreferencesLayout}>
      <Route path="users" Component={UsersPreferences} />
      <Route path="products" Component={ProductsPreferences} />
    </Route>
    <Route path="/products" Component={ProductListPage} />
  </Routes>
</BrowserRouter>
```

```
const PreferencesLayout: FC = () => (
  <div>
    <PreferencesNavbar />
    <Outlet /> { /* imported from React router dom */ }
  </div>
);
```

Dynamic routes

We need a dynamic part of the URL for each product's detail page.

```
const router = createBrowserRouter([
  { path: "/preferences/1", element: <ProductDetail id={1} /> },
  { path: "/preferences/2", element: <ProductDetail id={2} /> },
  { path: "/preferences/3", element: <ProductDetail id={3} /> },
  /* ..... */
]);
```

Not like that.

Dynamic routes

- be careful, you need to parse String to number
- check always if the parameter exists

```
const router = createBrowserRouter([
  {
    path: "/products/:productId",
    Component: ProductDetail,
  },
]);
```

```
const ProductDetail: FC = () => {
  const { productId } = useParams(); /* imported from React router dom */
  const { product } = useProductDetail(productId);

  return; /* html part with product detail */
};
```

Links in React router DOM

At the beginning, we mentioned that client-side routing is fake. So when we need to use links in our SPA, and we can not use `<a>`, what is the solution?

Luckily, the `react-router-dom` includes two types of links:

- `Link` is just a simple link to some location
- `NavLink` is more suitable for navigation, like Menu or Navbar

Link

Links move us to the specified location.

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

const SomeComponent: FC = () => {
  return (
    <div>
      {/* Absolute path */}
      <Link to="/preferences">Go to preferences</Link>

      {/* Relative path */}
      <Link to="../products" relative="path">
        Go to products
      </Link>
    </div>
  );
};
```


NavLink

The `isActive` state is helpful when we are building some more intuitive Navbars

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

const Navbar: FC = () => {
  return (
    <nav>
      <NavLink
        to="/preferences"
        className={({ isActive }) =>
          clsx("navlink", isActive && "navlink--active")
        }
      >
        Preferences
      </NavLink>
    </nav>
  );
};
```

Search parameters

Back to the first problem-solving. What if I create a link to my app and specify some filters? After opening that link, the user will immediately see the desired state of the application without additional clicking.

Naive solution:

```
const router = createBrowserRouter([
  { path: "/tasks/completed", Component: CompletedTasks },
  { path: "/tasks/in-progress", Component: InProgressTasks },
]);
```

We see that this solution will not work with all possible filters.

Search parameters

We can use search parameters:

```
const Menu: FC = () => {
  return (
    <Link to={{ path: "/products", search: "filter=status~eq~completed" }}>
      Completed tasks
    </Link>
  </nav>
);
};
```

```
const Tasks: FC = () => {
  let [searchParams, setSearchParams] = useSearchParams();
  const filter = searchParams.get("filter");

  return; /* render products */
};
```

Code splitting

We have simple routing:

```
import PreferencesPage from "./pages/PreferencesPage";
import ProductsPage from "./pages/ProductsPage";

const router = createBrowserRouter([
  {
    path: "/preferences",
    Component: PreferencesPage,
  },
  {
    path: "/products",
    Component: ProductsPage,
  },
]);
```

Code splitting

The final bundle has only one file. The file `index-*.js` is our application.

```
> routing-demo@0.0.0 build
> tsc && vite build

vite v5.2.8 building for production...

✓ 42 modules transformed.
dist/index.html          0.46 kB | gzip: 0.30 kB
dist/assets/index-l8X_Fozx.css 27.31 kB | gzip: 5.00 kB
dist/assets/index-DpfbmpuW.js 204.85 kB | gzip: 66.39 kB
✓ built in 814ms
```

Code splitting

What if we lazy load the `PreferencesPage` ?

```
import { lazy } from "react";
import ProductsPage from "./pages/ProductsPage";

const PreferencesPage = lazy(() => import("./pages/PreferencesPage"));

const router = createBrowserRouter([
  {
    path: "/preferences",
    Component: PreferencesPage,
  },
  {
    path: "/products",
    Component: ProductsPage,
  },
]);
```

Code splitting

Now our application is split into `PreferencesPage-*.js` and `index-*.js`

```
> routing-demo@0.0.0 build
> tsc && vite build

vite v5.2.8 building for production...
transforming (1) index.html

✓ 43 modules transformed.
dist/index.html                0.46 kB | gzip: 0.30 kB
dist/assets/index-l8X_Fozx.css 27.31 kB | gzip: 5.00 kB
dist/assets/PreferencesPage-Dy_e6Zun.js 0.60 kB | gzip: 0.36 kB
dist/assets/index-2YkGLfz0.js 205.43 kB | gzip: 66.89 kB
✓ built in 856ms
```

The PreferencesPage is now fetched from the server only if the user visits the route.

Suspense

But of course, we do not want to have a white page when the part of our page is not available yet.

```
import { Suspense } from "react";

const App = () => {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <AppLayout />
    </Suspense>
  );
};
```

The `Suspense` will render some fallback in that case.

Routing checkpoint

- Definition of the routes with Route Elements or Route Objects
- Dynamic routes with `useParams` hook to access the parameters
- Nested routes with `Outlet` and layouts
- Link and NavLink
- Search parameters with `useSearchParams` hook
- Code splitting and lazy loading

Forms

Why are forms important

The way, how user can somehow edit, create and delete data on server is throw forms. Because lot of application aims to somehow manage data, implementing and designing forms on frontend is important.

HTML Form

A pure HTML form is underestimated. It can do a lot of work. One of the main disadvantages is interactiveness. We want to inform users about the form's invalid data before sending it to the server.

Also, extensive and complicated forms are challenging to write in pure HTML. We need a lot of JS scripting to solve some complex parts. There, React and other UI libs and frameworks come into play.

```
<form action="/products" method="POST">
  <input name="title" />
  <input name="description" />
  <button>Submit</button>
</form>
```

Controlled and uncontrolled inputs

Two types of inputs are fundamental concepts of React. It can help us a lot or be a foot gun.

Controlled

```
const ProductForm: FC = () => {
  const [value, setValue] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log("Submitted form with: ", value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input value={value} onChange={(e) => setValues(e.target.value)} />
      <button>Submit</button>
    </form>
  );
};
```

Uncontrolled

```
const ProductForm: FC = () => {
  const inputRef = useRef<HTMLInputElement>();

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log("Submitted form with: ", inputRef.current?.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input ref={inputRef} />
      <button>Submit</button>
    </form>
  );
};
```

React hook form

We need a library...

- that can abstract the work with the form
- that helps us with easy data validation
- typescript friendly
- that has performance and can be used on large forms

There are many libraries, but we recommend using the React hook form.

React hook form - Input registration

```
import { useForm } from "react-hook-form";

const ProductForm: FC = () => {
  const { register, handleSubmit } = useForm();

  const submitHandler = (values) => {
    console.log("Submitted form with: ", inputRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit(submitHandler)}>
      <input {...register("title")} />
      <button>Submit</button>
    </form>
  );
};
```

React hook form with Typescript

By defining the form type, we can get handy suggestions in the development process, and of course, if we change the attribute or delete it, the build fails.

```
type CreateProduct = {
  title: string;
  description: string;
};

const ProductForm: FC = () => {
  const { register, handleSubmit } = useForm<CreateProduct>();

  const submitHandler: SubmitHandler<CreateProduct> = (values) => {
    console.log("Submitted form with: ", values.title, values.description);
  };

  return; /* form */
};
```

React hook form with validation

The form validation can be achieved with any validation library you want. You need to write the resolver for your library. For well-known libs like Zod, the resolvers are already written.

```
const productSchema = z.object({ title: z.string().min(3) });
type CreateProduct = z.infer<typeof productSchema>;

const ProductForm: FC = () => {
  const { register, handleSubmit, formState } = useForm<CreateProduct>({
    resolver: zodResolver(productSchema),
  });

  return (
    <form onSubmit={handleSubmit(submitHandler)}>
      <input {...register("title")} />
      {formState.errors.title && <p>{formState.errors.title.message}</p>}
      <button>Submit</button>
    </form>
  );
};
```

React hook form – Controller

In some cases, when the input is complicated or the library has limited API, we can use the Controller for communication in a controlled way.

```
const ProductForm: FC = () => {
  const { control, handleSubmit } = useForm<CreateProduct>();

  return (
    <form onSubmit={handleSubmit(submitHandler)}>
      <Controller
        control={control}
        name="deliveryDate"
        render={({ field }) => (
          <ReactDatePicker onChange={field.onChange} selected={field.value} />
        )}
      />
      <button>Submit</button>
    </form>
  );
};
```

React hook form - useFieldArray

Sometimes, the form has some dynamic parts. For example, a button which adds another user with a set of form fields. These array-like parts are not easy to implement, but there is a hook in reacting hook form, which does all the job for us.

```
type User = {
  name: string;
  email: string;
};

type UserListForm = {
  users: User[];
};

const Form: FC = () => {
  const { control, handleSubmit } = useForm<UserListForm>();

  const { append, remove, fields } = useFieldArray({ control, name: "users" });

  return; /* form */
};
```

React hook form - useFieldArray

```
const Form: FC = () => {
  const { control, handleSubmit } = useForm<UserListForm>();
  const { append, remove, fields } = useFieldArray({ control, name: "users" });
  const handleAddNew = () => append({ name: "", email: "" });

  return (
    <form onSubmit={handleSubmit(submitHandler)}>
      {fields.map((field, index) => (
        <div key={field.id}>
          <input {...register(`users.${index}.name`)} />
          <input {...register(`users.${index}.email`)} />
          <RemoveBtn type="button" onClick={() => remove(index)} />
        </div>
      ))}

      <AddBtn type="button" onClick={handleAddNew} />
      <button>Submit</button>
    </form>
  );
};
```

Forms – checkpoint

- we have controlled and uncontrolled inputs
- React hook form with `useForm` hook
- typescript support and data validation
- Controller and its usage
- complicated forms with `useFieldArray`