React Router

Continue with our One List app

- Let us add the ability to show a details page for each todo in our list
- The detail page should show the ID, text, complete status, and perhaps the created/updated timestamps
- The detail page should allow us to *DELETE* the todo item

We now have an app that supports different views of data

- We will distinguish what we are looking at by varying the URL
- The home page of "`/`" will always show the todo items list
- A url like "`/items/42`" will show the details of item 42

So our app has to change behavior based on what URL the user is displaying

- We need a way to differentiate our user interface based on URL
- We need a way to let the user navigate around

Enter React Router

React Router

- Transforms our application into a *Single Page App* (SPA)
- Even though our page will *respond* to many URLs, it is still one page (index.html)
- React Router makes it *seem* like we support many URLs

Add it to an existing app

```
npm install react-router react-router-dom
npm install --save-dev @types/react-router @types/react-router-dom
```

Let's add it to our One List App!

Adding the packages is not enough

- We need to update our `main.tsx` to add *React Router* support.

Step 1 - Import the `BrowserRouter`

- The purpose of the `Router` is to allow our application to handle different URLs.
- There are several different kinds of "Routers" but the most common, and the one we will use is `BrowserRouter`

```
import { BrowserRouter as Router } from 'react-router-dom'
```

- What is `{ BrowserRouter as Router }`?
- Imports the `BrowserRouter` component but calls it `Router` in **OUR** code.

Step 2 - Wrap our App

```
ReactDOM.render(
  <React.StrictMode>
    <Router>
      <App />
    </Router>
  </React.StrictMode>,
  document.getElementById('root')
)
```

Notice we have **surrounded** our App in a Router. This allows us to use react router throughout our app!

We are now ready to use React Router!

Time to get a bit more organized!

Our main App **has too much in it.**

- Let us refactor the `ul` and the `form` out into its own component, moving all the state stuff with it

Whew, now we have a distinct component for the `TodoList`

- But how does this help us?
- Well, we can now tell our app to only render this if the URL is `/`

```
<main>
  <Switch>
    <Route path="/">
      <TodoList />
    </Route>
  </Switch>
</main>
```

Try it out

- Visit the site at `/`
- Visit the site at `/should-not-work`
  - Wait, this still works! Why?

Route matching rules

- The path="/" means "If the path **starts with** /"
- If we want it to be **exact** we have to tell it

```
<main>
  <Switch>
    <Route exact path="/">
      <TodoList />
    </Route>
  </Switch>
</main>
```

- Now try it!

When we visit a URL that is not a match we still get our header and footer!

- This is because the only part of the page that swaps out is what is inside of our `Switch`
- Let us add a "not found!" with `path="*"`

```
<main>
  <Switch>
    <Route exact path="/">
      <TodoList />
    </Route>
    <Route path="*">
      <p>Ooops, that URL is unknown</p>
    </Route>
  </Switch>
</main>
```

Order is important

`Switch` will find the *first* match and stop

Add a page to show the details of a page

- Put a "Route" to `/items/42`

```
<Switch>
  <Route exact path="/">
    <TodoList />
  </Route>
  <Route path="/items/42">This would be the details of item 42!</Route>
  <Route path="*">
    <p>Ooops, that URL is unknown</p>
  </Route>
</Switch>
```

But how do we handle multiple pages??

- Certainly, we do not want to write out many Route entries!?
- We can put a "parameter" in the path=

```
<Switch>
  <Route exact path="/">
    <TodoList />
  </Route>
  <Route path="/items/:id">
    <p>This would be the details of item 42!</p>
  </Route>
  <Route path="*">
    <p>Ooops, that URL is unknown</p>
  </Route>
</Switch>
```

Now, rather than putting the JSX right in here, make a component

```
function TodoItemPage() {
  return <p>This would be the details of item 42!</p>
}

<Route path="/items/:id">
  <TodoItemPage />
</Route>
```

Yeah, but how do we know which ID we want to show?

- React Router hooks!
- `useParams()`
- Add a TypeScript type inside <> to declare what our params are

```
function TodoItemPage() {
  const params = useParams<{id: string}>()

  return <p>This would be the details of item {params.id}!</p>
}
```

The `params` is similar to `props`. However, the values come from our `Route`

```
<Route path="/items/:id">
                      V
                      |
                      |
                      +-----><--------><--------><---------+
                                                           |
function TodoItemPage() {                                  v
  const params = useParams<{id: string}>()                 |
                                                           v
  return <p>This would be the details of item {params.id}!</p>
}
```

See that we can put in any item ID and see it on the page!

Ok, so now let us load some data for that specific item!

Make a state

```
const [todoItem, setTodoItem] = useState({
  id: undefined,
  text: '',
  complete: false,
  created_at: undefined,
  updated_at: undefined,
})
```

In comes useEffect again

```
useEffect(
  function () {
    async function loadItems() {
      const response = await axios.get(
        `https://one-list-api.herokuapp.com/items/${params.id}?access_token=cohort42`
      )

      if (response.status === 200) {
        setTodoItem(response.data)
      }
    }

    loadItems()
  },
  [params.id]
)
```

Render something

```
return (
  <div>
    <p className={todoItem.complete ? 'completed' : ''}>{todoItem.text}</p>
    <p>Created: {todoItem.created_at}</p>
    <p>Updated: {todoItem.updated_at}</p>
    <button>Delete</button>
  </div>
)
```

```
function TodoItemPage() {
  const params = useParams<{ id: string }>()
  const [todoItem, setTodoItem] = useState({
    id: undefined,
    text: '',
    complete: false,
    created_at: undefined,
    updated_at: undefined,
  })

  useEffect(
    function () {
      async function loadItems() {
        const response = await axios.get(
          `https://one-list-api.herokuapp.com/items/${params.id}?access_token=cohort42`
        )

        if (response.status === 200) {
          setTodoItem(response.data)
        }
      }

      loadItems()
    },
    [params.id]
  )

  return (
    <div>
      <p className={todoItem.complete ? 'completed' : ''}>{todoItem.text}</p>
      <p>Created: {todoItem.created_at}</p>
      <p>Updated: {todoItem.updated_at}</p>
      <button>Delete</button>
    </div>
  )
}
```

Make the button work!

```
async function deleteTodoItem() {
  await axios.delete(
    `https://one-list-api.herokuapp.com/items/${params.id}?access_token=cohort42`
  )

  // Need to redirect back to the main page!
}
```

Add a handler

```
<button onClick={deleteTodoItem}>Delete</button>
```

Handle the redirect

- Another hook: `useHistory`
- This allows us to manipulate the history/browser location

```
const history = useHistory()

async function deleteTodoItem() {
  const response = await axios.delete(
    `https://one-list-api.herokuapp.com/items/${params.id}?access_token=cohort42`
  )

  if (response.status === 204) {
    // Send the user back to the homepage
    history.push('/')
  }
}
```

Ok, but how do we make these pages linkable?

Instead of

```
<a href=""></a>
```

we can use

```
<Link to="" />
```

We can generate those links dynamically:

```
<Link to={`/items/${id}`}
```

```
return (
  <li className={complete ? 'completed' : ''} onClick={toggleCompleteStatus}>
    {text}
    <Link to={`/items/${id}`}>Show</Link>
  </li>
)
```

Update the TodoPage to have a link "home."

```
return (
  <div>
    <p>
      <Link to="/">Home</Link>
    </p>
    <p className={todoItem.complete ? 'completed' : ''}>{todoItem.text}</p>
    <p>Created: {todoItem.created_at}</p>
    <p>Updated: {todoItem.updated_at}</p>
    <button onClick={deleteTodoItem}>Delete</button>
  </div>
)
```

Navigate around the app

- Some UI/UX aspects we could improve
- However, we have a working app!