React Router V6 Notes

```
Basic Setup
In app.js ->
function app {
    return (
        <BrowserRouter>
            <Routes>
                <Route path="/" element={<SignUp />} />
                <Route path="login" element={<Login />} />
                <Route path="*" element={<NotFound />} />
            </ Routes>
        </ BrowserRouter>
    )
}
```

Route Params

```
<Route path="/page/:id" element={<Component />} />
// The ':id' makes this a param
// Multiple params
<Route path="/page/:id/:name/:mood" element={<Component />} />
```

useParams Hook

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

function VanDetail() {
    const params = useParams();
    // Assume we visited /page/2 (Assume we have a "/page/:id" route)
    console.log(params.id) // logs "2" to the console
}

// Fetching data for a specific id
function VanDetail() {
    const params = useParams();
    const [vanData, setVanData] = useState(null);

    useEffect(() => {
        fetch(`backend/users/${params.id}`)
            .then(res => res.json())
            .then(data => setData(data))
    }, [params.id]);
    // We put this in a useEffect to avoid infinite re-renders
    // We have params.id as a dependency but we don't need it (I think)
}
```

Nested Routes

```
<Route path="/host" element={<Dashboard />}>
    <Route path="/host/income" element={<Income />} />
    <Route path="/host/reviews" element={<Reviews />} />
</Route>
```

Outlet

```
import { Outlet } from "react-router-dom"
// Suppse we have the same routes as above, then, in Dashboard.jsx

export default function Dashboard() {
    // Whatever you want
    return (
        <>
            <nav>Add a navbar maybe</nav>
            <Outlet />
            <footer>Add a footer maybe</footer>
        </>
    )
}
What this will do is - it will display the dashboard content on all the routes
// YOU NEED AN OUTLET IN THE PARENT COMPONENT/ROUTE
// WHICH WILL THEN MATCH THE CHILD COMPONENT/ROUTE
Example - on the /host/income route, we have the dashboard and the income elements
displayed
```

Note - Nesting is only useful if you have some shared UI between those routes (eg - a navbar, a header, a footer, etc.)

Better Nested Routes

```
<Route path="host" element={<HostLayout />}>
    <Route index element={<Dashboard />} />
    <Route path="income" element={<Income />} />
    <Route path="reviews" element={<Reviews />} />
</Route>
In this case, the HostLayout will be displayed on every route
while the dashboard component is the index route
```

Absolute vs Relative Paths

```
Routes starting with '/' are absolute
Routes that do not begin with '/' are relative to the parent route (this is usually
used in nested routes)

<Route path="/" />
<Route path="/login" />
<Route path= "/login/admin" />
// These are absolute paths

<Route path="/">
    <Route path="login" />
</Route>
// Here, the login route is relative to the "/" absolute route
```

NavLinks

Navlinks are a solution such that if we are on a specific link, then we can pass isActive and isPending properties to determine className and style properties of the link, just use NavLink instead of Link to create this effect

```
<NavLink
    to="/host"
    className={({isActive}) => isActive ? "active-link" : null}
>
    Host
</NavLink>
There is also an isPending and isTransitioning property which can be used
Instead of classname, we can directly use style and use the same isActive,
isPending and isTransitioning properties.
```

A PROBLEM - Navlinks match all active routes, so we would have the active-link stlye applied to all the routes such as / and /host in the above example

What we want is to have the matching to end at a particular point if a more rested route matches the URL. Solution? the "end" prop

```
<NavLink
    to="/host"
    end
    className={({isActive}) => isActive ? "active-link" : null}>
    Host
</NavLink>
```

```
Now, if a more rested route matches, the /host route will not match, therefore we
will get the desired styling
```

Relative to path vs Relative to route

```
Lets say you have the following routes
<Route to="host">
    <Route to="something" />
    <Route to="something/:id" />
</Route>

// In host/something/id =>
<Link to="..">Something</Link> // This will take you to /host instead of
/host/something as the link is relative to the route heirarchy

// To get the desired functionality, we need
<Link
    to=".."
    relative="path">
    Something
</Link>
// We have made the link relative to the path instead of the route so we get what we
want
```

Hook – useOutletContext

```
If you want to pass state (context) to an Outlet
// In the parent component =>
<Outlet context={ [data, setData] } />

// In the child component =>
import { useOutletContext } from "react-router-dom"

export default function ChildComponent() {
    const [data, setData] = useOutletContext();

    return (
        <h1>{data}</h1>
    )
}
```

Search/Query Params

```
If you want to filter, sort, paginate, etc. then you might want to use query params
which appear like
www.example.com/?type=something&filter=anotherthing
// Here type and filter are the query parameters
```

Hook – useSearchParams

```
// Assume we have the url - www.example.com/?type=happy&filter=none
Const [searchParam, setSearchParam] = useSearchParams()
Const typeFilter = searchParams.get("type")
console.log(typeFilter) // logs "happy" to the console
```

Filtering using search params

```
const colourFilter = searchParams.get("type")
// Suppose you have an array called elements and you want to filter based on colour
const displayed = colourFilter ? elements.filter((element) => element.colour ==
colourFilter) : elements
const displayedMapped = displayed.map((element) => <h1>{element.colour}</h1>)
```

Buttons to filter

```
<Link to="?type=happy">Happy</Link>

// Clear Filters
<Link to="."> Clear Filters </Link>

// Better method - Using the setSearchParams setter function
// setSearchParams is extremely flexible so we can do the following and they are all
equivalent
<button onClick={() => setSearchParams("?type=happy")}></button>
<button onClick={() => setSearchParams("type=happy")}></button> // Without question
mark

// Best Practice =>
<button onClick={() => setSearchParams({ type: "happy" })}></button>

// Note, we can use buttons and links to achieve the same effect
```

State in Links

```
<Link to="/" state={ { key: value, key2: value2} } /> // See below for how to get
this state
```

Hook - useLocation

```
// If you passed in state in a link, you can get that state using the useLocation
hook
const location = useLocation();
console.log(location) // logs {pathname: "xyz/xyz", search: "", hash: "", state:
{key: value, key2: value2}, key: "xyz"}
// This can be useful for LOTS of reasons
```

Custom 404 Page

```
// We need a 'catch-all' route if a non-existent route is visited
<Routes>
    <Route path="/home" element={<Home />} />
    <Route path="/about" element={<About />} />
    <Route path="*" element={<h1>Page not found!</h1>} />
</Routes>
// NOTE - React router is smart enough that it will first check all the routes with
the URL and only if no route matches will it
// go to the catch-all route, but it is best to put it at the bottom anyway
```

Loaders

```
// In general -> Better to use loader instead of useEffects to fetch data
// React can delay the rendering of the page until the loader is done
// First get data, then go to the route is the main idea
// No requirement of a loading or error state management
// First, we have to "subscribe" to this feature
// Replace
<Routes>
    <Route path="/home" element={<Home />} />
    <Route path="/about" element={<About />} />
    <Route path="*" element={<h1>Page not found!</h1>} />
</Routes>

// With (everything between <Routes> <Routes />)
const router = createBrowserRouter(createRouteFromElements(
    <Route path="/home" element={<Home />} loader={homeLoader} />
    <Route path="/about" element={<About />} />
    <Route path="*" element={<h1>Page not found!</h1>} />
))
// Put a loader in whichever route you are fetching data

function App() {
    return (
        <RouterProvider router={router} />
    )
}

// Remember to import RouterProvider, createBrowserRouter, createRouteFromElements
```

Loader Setup

```
// In the component where you are fetching data =>
export function loader() {
    return "data here"
}

export default function Component() {
    return <h1>Hi!</h1>
}

// In App.jsx (or wherever you have your routes)
import Component, { loader as componentLoader } from "./components/Component"

const router = createBrowserRouter(createRouteFromElements(
    <Route path="/home" element={<Component />} loader={componentLoader} />
))

function App() {
    return (
        <RouterProvider router={router} />
    )
}
```

Hook - useLoaderData

```
// In Component.jsx =>

export function loader() {
    return "data wohoo!"
}

export default function Component() {
    const data = useLoaderData()
    console.log(data) // Logs "data wohoo!" to the console
    // (Assuming everything is correct in App.jsx)
}
```

Error handling with loaders and routes

```
// If your Component.jsx throws for any reason (failed to get data, etc.)
// Then you can pass an errorElement prop to the route
<Route path="/home"
    element={<Component />}
    loader={componentLoader}
    errorElement={<h1>There was an error</h1>} />
// Now if Component throws, we will see an h1 with the message there was an err
```

A problem - We want to give actual information about the error Solution - useRouteError hook

Hook - useRouteError (Only available with data routers (loaders))

```
// Instead of errorElement={<h1>Error occurred</h1>}
// Use an Error component
// In Error.jsx =>
import { useRouteError } from "react-router-dom"

export default function Error() {
    const error = useRouteError()
    console.log(error) // logs {message: "message", statusText: "text", status: 
number}
    return (
        <h1>{error.message}</h1>
    )
}
// NOTE - We could have the errorElement prop in a parent route and it would still
work in the child routes (play around with this lol)
```

Protected Routes

Central idea - User should be logged in to access a specific page/route

```
// Solution - Wrap the route you want to protect in an AuthRequired layout route
// which either renders the Outlet or redirects the user to the home/login page

// In AuthRequired.jsx =>
export default function AuthRequired() {
    const isLoggedIn = true; // fake auth for now
    if (!isLoggedIn) {
        return <Navgate to="/login" />
    }
    return <Outlet />
}
// NOTE - This approach is without data loaders' functionalities
```

Navigate Component

```
// Enables us to navigate a user from one route to another
<Navigate to="wherever" />
```

Approach using loaders

```
export function loader() {
    const isLoggedIn = false; // fake auth for now again
    if (!isLoggedIn) {
        return redirect("/login") // we can also throw redirect("/login")
    } // Note, redirect is not the same as Navigate as we are not in a component
here
    // Fetch your data now and do whatever you want
}
```

Disadvantage of using authentication in loaders

We have to include authentication in EVERY loader function
Why? because loaders run parallelly and not first in the parent, then in the child and so on
This creates the requirement to have authentication in every loader so that sensitive info/data is not exposed if the child route gets some data that should not be accessible

Hook - useNavigate

```
// Remember, this is different from redirect as hooks can be used at the top level
of components
// It will not be useful outside of the component
const navigate = useNavigate()
navigate("/dashboard", { replace: true }) // What does replace do?
// Im not quite sure :D
```

Forms and Actions

Forms in react are... hard to manage, but we actually have access to a Form element in react router (If we are using loaders and data routes)

```
// Replace <form></form> with <Form method="post"></Form>
// In Component.jsx =>
export async function action(obj) {
    // Do something, You have access to an object which has the following structure
    // {request: Request {}, params: {}}
    const formData = await obj.request.formData();
    const email = formData.get("email"); // name property of the input (IMPORTANT)
    const password = formData.get("password");
    console.log(email, password) // Works as expected
}

// In App.jsx =>
import { action as loginAction } from "./components/Login"
<Route
    path="/login"
    element={<Login />}
    loader={/* loader here */}
    action={loginAction}
/>
```

Hook - useActionData

```
// When using data routes, we can useActionData like we would useLoaderData
const data = useActionData(); // gives us the whatever was returned by the action
function :)
```

Hook - useNavigation

```
// Consists of many properties, one of which is state, has "idle" and "submitting"
states
// We do not need to track the state of the application because of this hook :D
const navigation = useNavigation()
const status = navigation.state()
```

Not adding in detail usage since it is pretty intuitive and documentation is good

Suspense

Sometimes, using data routes leads to a poor UI even though it improves the developer experience This is why we may opt to use Suspense

Await, defer

```
import { defer, Await } from "react-router-dom"

export async function loader() {
    const promise = getData(); // Not awaited, so we have a promise
    return defer({data: dataPromise});
}

export default function Component() {
    const loaderData = useLoaderData();
    console.log(loaderData); // Logs {promiseData: Promise {}} to the console

    return (
        <Await resolve={loaderData.weather}>
            {(loadedWeather) => {
                const iconUrl =

`http://openweathermap.org/img/wn/${loadedWeather.weather[0].icon}@2x.png`
                return (
                    <>
                        <h3>{loadedWeather.main.temp}ºF</h3>
                        <img src={iconUrl} />
                    </>
                )
            }}
        </Await>
    )
}
// The above example has been taken from scrimba.com
```

Suspense component

```
// The above will give us an error... for now
import React, { Suspense } from "react"
// We can simple wrap whatever we are awaiting and get the desired result
<React.Suspense /* or just Suspense since we imported it */ fallback={<h1>Loading...
</h1>} /* fallback is just the loading state */>
    <Await resolve={loaderData.weather}>
        {(loadedWeather) => {
            const iconUrl =

`http://openweathermap.org/img/wn/${loadedWeather.weather[0].icon}@2x.png`
            return (
                <>
                    <h3>{loadedWeather.main.temp}ºF</h3>
                    <img src={iconUrl} />
                </>
            )
        }}
    </Await>
</React.Suspense>
```