
Welcome to Lecture 29

Agenda

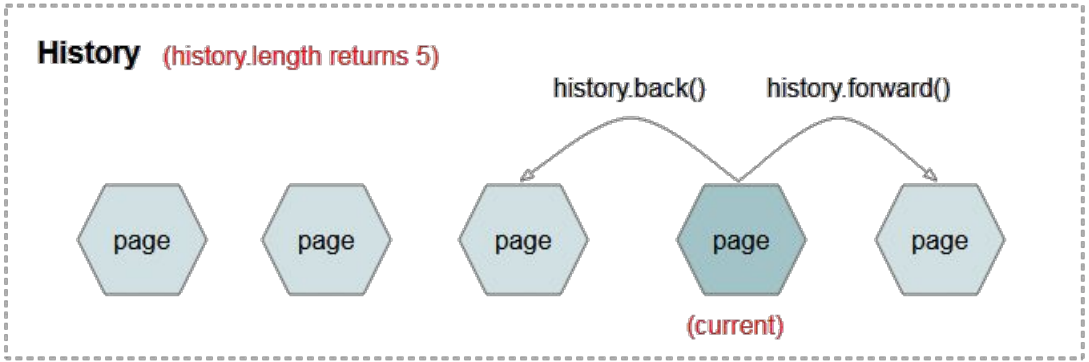
Session Objectives

- Reset
- Review
- React Router
    - Foundations: Browser’s History
    - Revisit: SPA & MPA
    - Understand the “What: of React Router V6

The "Why" - From MPA to SPA (a recap)

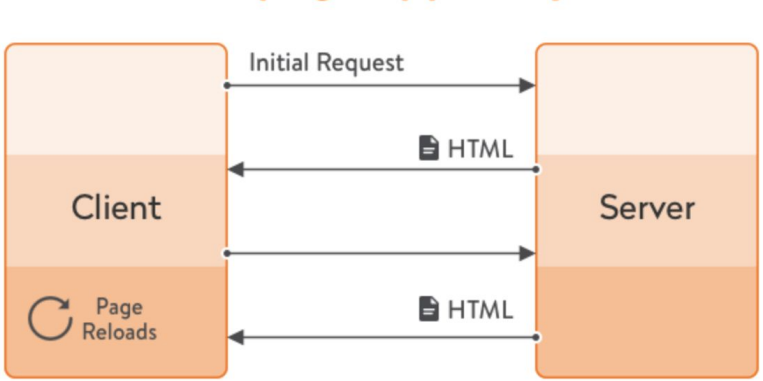
The Browser APIs That Power Modern Routing

- React Router is a library built on top of standard features that exist in all modern web browsers.
- The History API: The core engine browser APIs providing JavaScript with access to the browser's session history. Crucially, its pushState() method allows us to change the URL in the address bar without making a new request to the server
- The URL Object: Provides a standardized way to parse, construct, and read URLs, allowing the router to easily get information like the pathname, search parameters, and hash



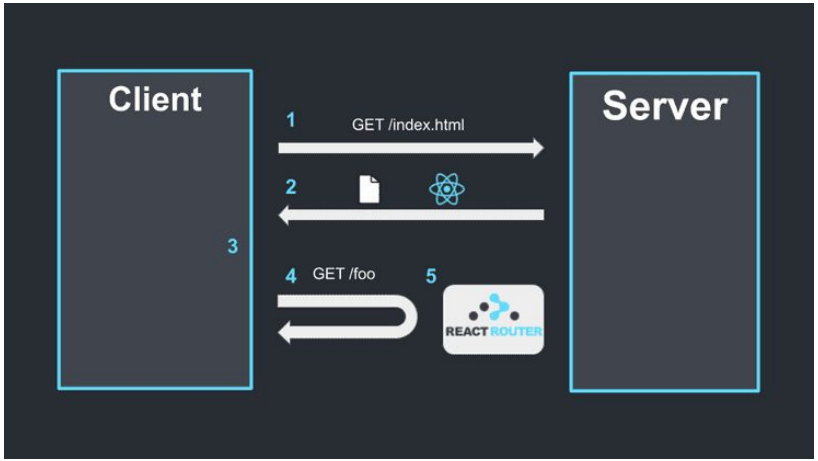
Traditional Web Model: Multi-Page Applications

- Recall: In a traditional MPA, every distinct URL corresponds to a separate HTML document on a server.
- The Process
  - User clicks an <a href="/about"> link
  - The browser sends a full HTTP GET request to the server for the /about.html document
  - The server responds with the new document
  - The browser discards the old page entirely, including all JavaScript state, and renders the new one
- **The Drawback**: This model results in a full-page reload, which leads to a "white flash," loss of application state (like form inputs or cart data), and increased server load



The Solution: Client-Side Routing in SPAs

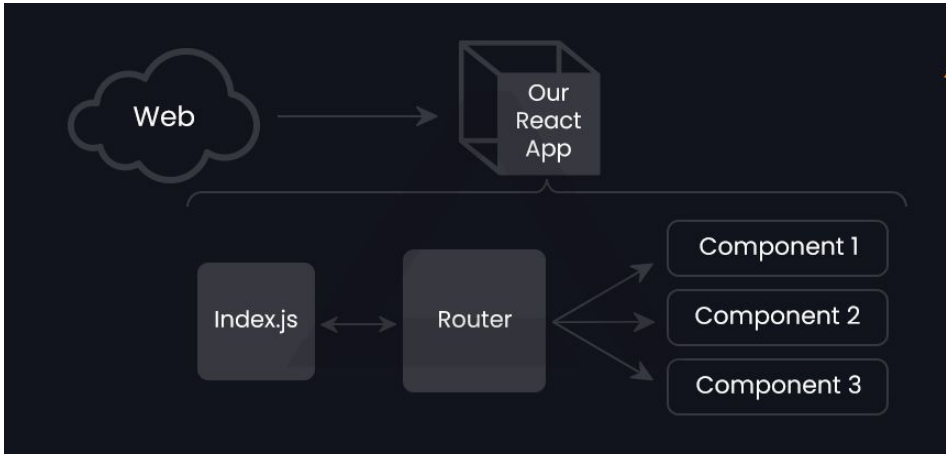
- In an SPA, the server sends only a single HTML document (index.html) initially. All subsequent "page" changes are handled by JavaScript on the client-side.
- The Process
  - User clicks a navigation element
  - A client-side routing library intercepts this event
  - The library uses the History API (history.pushState()) to update the URL in the address bar without a page reload
  - The library's internal state updates to reflect the new URL
  - The library determines which view or component corresponds to the new URL and renders it, updating the DOM efficiently without discarding the application's state



The "What" - Core React Router v6 Concepts

The Solution for React: React Router

- While the concepts of client-side routing are universal, React applications need a way to declaratively map URL paths to React components
- React Router is the de-facto standard routing library for React
- It provides a collection of components and hooks that integrate seamlessly with the React component model
- It allows us to build a descriptive routing structure directly within our JSX.



Core Components for Declarative Routing

- **BrowserRouter**: A context provider that should wrap your root application component (<App />).
    - Connects your component tree to the browser's URL and makes routing features available throughout your app
- **Routes**: A container component that acts as a switchboard. It examines the current URL and renders the UI of the first child <Route> that matches the URL path
- **Route**: Defines a mapping between a URL path and a React component. It uses two key props
    - **path**: A string that defines the URL segment to match (e.g., "/", "/products")
    - **element**: The React element to render when the path matches (e.g., <HomePage />)
- **Link**: A component used to create navigation links. It renders an <a> tag but overrides its default behavior, using the History API to change the URL without triggering a server request. It uses the to prop instead of href

Dynamic Segments and the useParams Hook

- **Concept**: Dynamic segments are variable parts of a URL path used to display specific resources (e.g., a product with a specific ID).
- **Syntax**: A dynamic segment is defined in a route's path prop with a colon prefix (e.g., :productId).
- **useParams Hook**: Returns an object of key/value pairs from the current URL's dynamic segments.

```
// In your routing setup:
<Route path="/products/:productId" element={<ProductPage />} />

// In your ProductPage.jsx component:
import { useParams } from 'react-router-dom';

function ProductPage() {
  const { productId } = useParams(); // returns { productId: 'value-from-url' }
  // Now you can use productId to fetch data for that specific product.
  return <div>Displaying product with ID: {productId}</div>;
}
```

Layout Routes, Nested Routes, and <Outlet />

- **Concept**: A Layout Route is a parent <Route> without a path that renders a component responsible for a shared UI structure (e.g., a Navbar, Sidebar, and Footer)
- **Concept**: Nested Routes are <Route> components defined as children of a layout route. Their paths are relative to the parent
- **The <Outlet /> Component**: A special component used within a layout route's element. It acts as a placeholder, marking the spot where the matching nested child route's element should be rendered

```
// In your routing setup:
<Route path="/" element={<MainLayout />}>
  <Route index element={<HomePage />} /> {/* Renders at "/" */}
  <Route path="about" element={<AboutPage />} /> {/* Renders
at "/about" */}
</Route>

// In your MainLayout.jsx component:
import { Outlet } from 'react-router-dom';

function MainLayout() {
  return (
    <div>
      <Navbar />
      <main>
        <Outlet /> {/* Child routes will render here */}
      </main>
      <Footer />
    </div>
  );
}
```

Imperative Navigation and Not Found Pages

- **useNavigate Hook**: A hook that provides a function to change the URL from within your component's logic, such as after a form submission or a successful login
  - Example
    - const navigate = useNavigate();
      navigate('/dashboard');
- **"Not Found" Route**: A special route with path="*" acts as a catch-all.
  - The <Routes> component will render this route if no other path in the list matches the current URL
  - This is used to display a custom "404 Page Not Found" component

Demo!

That’s it for today.
Questions?

Have a
good one!