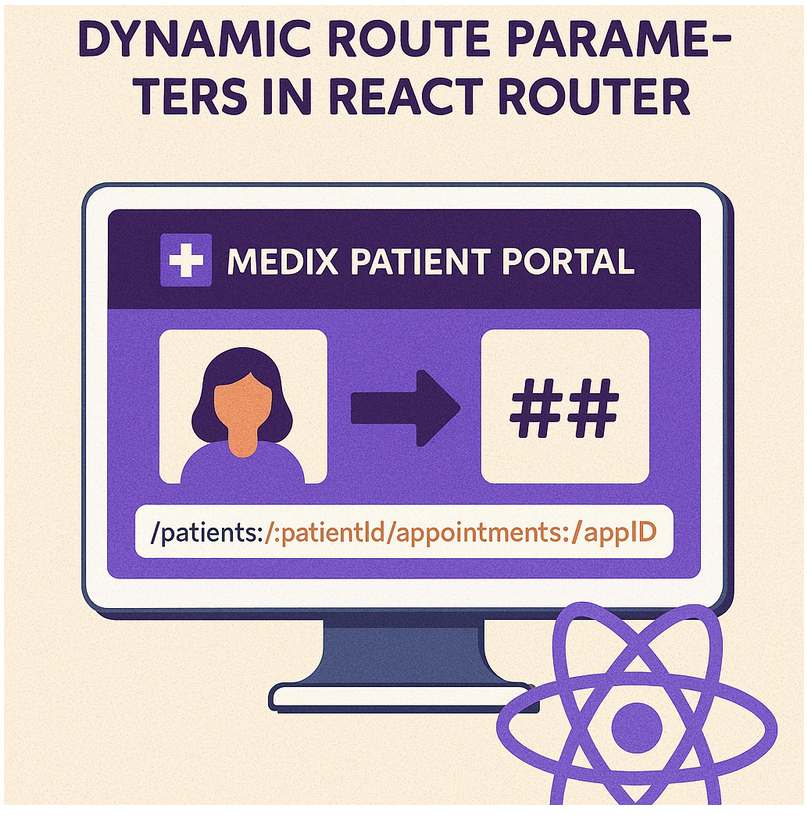
Routing in React: Type-Safe Route Parameters with React Router & TypeScript

1. Problem Statement

Case Study: Medix Patient Portal

Medix is building a patient portal:

- Each patient, doctor, and appointment has a unique ID in the URL (e.g., `/patients/:patientId/appointments/:appointmentId`).
- The dashboard must display details for the correct patient/appointment and prevent type errors (e.g., using a string where a number is expected).
- Developers want to catch mistakes at compile-time, not after deployment (e.g., a route handler expecting a `number` but getting `undefined`).
- The codebase must remain maintainable as routes grow more complex.



The challenge:
How do you define, use, and enforce type-safe dynamic route parameters in React Router, so your navigation and data fetching are both robust and error-free?

2. Learning Objectives

By the end of this tutorial, you will:

- Define dynamic routes in React Router with parameters (e.g., `/patients/:patientId`).
- Use TypeScript interfaces and generics to type route params.
- Extract and validate params in components using `useParams`.
- Prevent common runtime errors (e.g., missing/undefined params, wrong types).
- Explore advanced patterns for type-safe navigation and route config.

3. Concept Introduction with Analogy

Analogy: The Hospital Reception Desk

- **Routes** are like appointment slips: Each slip has fields (patient ID, appointment ID) that must be filled out correctly.
- **TypeScript** is the receptionist: It checks every slip for missing or mistyped info before sending it to the doctor.
- **React Router** is the hospital's navigation system: It makes sure each patient/doctor/appointment page receives the right information, in the right format, every time.

4. Technical Deep Dive

A. Defining Dynamic Routes with Parameters

- In React Router, you define parameters in the path using `:paramName`:

```
<Route path="/patients/:patientId/appointments/:appointmentId" element={<AppointmentDetails />} />
```

- These parameters are parsed from the URL and made available to your component.

B. Extracting and Typing Route Parameters with `useParams`

- The `useParams` hook returns an object with the params from the URL.
- In TypeScript, you can specify the expected param types using a generic

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

interface AppointmentParams {
  patientId: string;
  appointmentId: string;
}

const AppointmentDetails: React.FC = () => {
  const { patientId, appointmentId } = useParams<AppointmentParams>();
  // patientId and appointmentId are now strongly typed as string
  return (
    <div>
      <h1>Patient: {patientId}</h1>
      <h2>Appointment: {appointmentId}</h2>
    </div>
  );
};
```

- If you expect a number, convert and validate:

```
const { appointmentId } = useParams<AppointmentParams>();
const numericId = Number(appointmentId);
if (isNaN(numericId)) return <div>Invalid appointment ID</div>;
```

C. Passing and Navigating with Parameters

- Use the `Link` or `useNavigate` to create URLs with params:

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

<Link to={`/patients/${patientId}/appointments/${appointmentId}`}>View Appointment</Link>
```

For programmatic navigation:

```tsx
import { useNavigate } from 'react-router-dom';
const navigate = useNavigate();
navigate(`/patients/${patientId}/appointments/${appointmentId}`);
```

D. Type-Safe Route Config and Advanced Patterns

- Libraries like [react-router-typesafe-routes] and [react-router-typed-object] provide helpers for even stricter typing, including:
    - Compile-time errors for missing/wrong params.
    - Type-safe navigation helpers.
- You can also infer types from route path strings using TypeScript conditional types and template literals.

Common Pitfalls & Best Practices (React Router Params)

Pitfall	Best Practice
Accessing params without typing	Always use `useParams<YourParams>()`
Expecting query params in `useParams`	Only path params are returned by `useParams`
Forgetting to validate param types	Convert and check (e.g., `parseInt`, `isNaN`)
Using params before they're loaded	Handle undefined cases for async routes

5. Step-by-Step Data Modeling & Code Walkthrough

A. Define Routes with Parameters

```tsx
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import AppointmentDetails from './AppointmentDetails';

const App = () => (
  <BrowserRouter>
    <Routes>
      <Route path="/patients/:patientId/appointments/:appointmentId" element={<AppointmentDetails />} />
    </Routes>
  </BrowserRouter>
);
```

B. Extract and Type Params in the Component

```tsx
// AppointmentDetails.tsx
import { useParams } from 'react-router-dom';

interface AppointmentParams {
 patientId: string;
 appointmentId: string;
}

const AppointmentDetails: React.FC = () => {
 const { patientId, appointmentId } = useParams<AppointmentParams>();

 // Validate and use parameters
 if (!patientId || !appointmentId) {
   return <div>Missing or invalid parameters</div>;
 }
```

```
  // Optionally convert to number if needed
  const apptId = Number(appointmentId);
  if (isNaN(apptId)) {
    return <div>Invalid appointment ID</div>;
  }

  return (
    <div>
      <h1>Patient: {patientId}</h1>
      <h2>Appointment: {apptId}</h2>
      {/* Fetch and display appointment details */}
    </div>
  );
};

export default AppointmentDetails;
```

C. Navigating with Typed Params

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

const PatientRow: React.FC<{ patientId: string }> = ({ patientId }) => (
  <Link to={`/patients/${patientId}/appointments/123`}>View Appointment 123</Link>
);
```

D. Advanced: Type-Safe Navigation with Helper Libraries

- With [react-router-typesafe-routes]

```
import { route, useTypedParams } from "react-router-typesafe-routes";
const routes = route({ patient: route({ path: "patients/:patientId" }) });
// In component:
const { patientId } = useTypedParams(routes.patient);
```

- With [react-router-typed-object]

```
const params = ROUTES["/patients/:patientId/appointments/:appointmentId"].path.useParams();
// params.patientId and params.appointmentId are typed as string
```

6. Interactive Challenge / Mini-Project

Your Turn!

1. Define a route `/doctors/:doctorId/patients/:patientId` and a `DoctorPatientDetails` component.
2. Use a typed interface for params and extract them in the component.
3. Validate that both IDs are present and numeric; display an error if not.
4. Add a link from a doctor list to a specific doctor/patient page, passing the IDs as parameters.

7. Common Pitfalls & Best Practices

Common Pitfalls & Best Practices (Routing & Params)

Pitfall	Best Practice
Not typing params	Always use `useParams<YourParams>()`

Pitfall	Best Practice
Not validating param types	Check and convert as needed
Expecting query params in `useParams`	Use `useSearchParams` for query strings
Hardcoding route strings	Use helper libraries for type-safe navigation

8. Optional: Programmer's Workflow Checklist

- Define route params in the path using `:param`.
- Create a TypeScript interface for param types.
- Always use `useParams<YourParams>()` in components.
- Validate and convert params as needed (e.g., parseInt).
- Use `Link` or `useNavigate` for navigation, passing params as strings.
- Consider helper libraries for large/complex route configs.