React Router 6.5+ (Data Router) vs Classic React Router

Key Differences

1. Data Loading Pattern

```
sequenceDiagram
    participant Classic
    participant DataRouter

    Note over Classic: Classic React Router
    Classic->>Classic: 1. Component mounts
    Classic->>Classic: 2. useEffect runs
    Classic->>Classic: 3. State updates
    Classic->>Classic: 4. Component re-renders

    Note over DataRouter: React Router 6.5+ (Data Router)
    DataRouter->>DataRouter: 1. Route matches
    DataRouter->>DataRouter: 2. Loader runs
    DataRouter->>DataRouter: 3. Data is ready
    DataRouter->>DataRouter: 4. Component renders
```

2. Code Comparison

Classic React Router:

```
// Route Definition
<Route path="/users" element={<Users />} />;

// Component
function Users() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function fetchUsers() {
      try {
        const response = await fetch("/api/users");
        const data = await response.json();
        setUsers(data);
      } catch (err) {
        setError(err);
      } finally {
        setLoading(false);
      }
```

```
    }
    fetchUsers();
  }, []);

  if (loading) return <LoadingSpinner />;
  if (error) return <ErrorMessage error={error} />;

  return <UserList users={users} />;
}
```

React Router 6.5+ (Data Router):

```
// Route Definition
const router = createBrowserRouter([
  {
    path: "/users",
    element: <Users />,
    loader: async () => {
      const response = await fetch("/api/users");
      if (!response.ok) throw new Error("Failed to fetch users");
      return response.json();
    },
    errorElement: <ErrorMessage />,
  },
]);

// Component
function Users() {
  const users = useLoaderData();
  return <UserList users={users} />;
}
```

Major Improvements in 6.5+

1. Data Management

- **Before**: Data fetching was component-level, leading to:
  - Multiple loading states
  - Inconsistent error handling
  - Race conditions
  - Waterfall requests
- **After**: Data fetching is route-level, providing:
  - Parallel data loading
  - Automatic error boundaries
  - Request deduplication
  - Data caching

2. Form Handling

```
sequenceDiagram
    participant Classic
    participant DataRouter

    Note over Classic: Classic Approach
    Classic->>Classic: 1. Form submit
    Classic->>Classic: 2. Prevent default
    Classic->>Classic: 3. Manual fetch
    Classic->>Classic: 4. Handle response
    Classic->>Classic: 5. Update state
    Classic->>Classic: 6. Navigate manually

    Note over DataRouter: Data Router Approach
    DataRouter->>DataRouter: 1. Form submit
    DataRouter->>DataRouter: 2. Action runs
    DataRouter->>DataRouter: 3. Automatic navigation
    DataRouter->>DataRouter: 4. Data revalidation
```

3. Error Handling

- **Before**: Manual error handling in each component
- **After**: Route-level error boundaries with `errorElement`

4. Loading States

- **Before**: Manual loading state management
- **After**: Built-in loading states with `useNavigation`

Migration Guide

1. Router Setup

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

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
      </Routes>
    </BrowserRouter>
  );
}

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

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
    loader: homeLoader,
  },
]);

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

2. Data Loading

```
// Before
function UserProfile() {
  const [user, setUser] = useState(null);
  useEffect(() => {
    fetchUser().then(setUser);
  }, []);
  // ...
}

// After
function UserProfile() {
  const user = useLoaderData();
  // ...
}

export async function loader() {
  return fetchUser();
}
```

3. Form Handling

```
// Before
function CreateUser() {
  const navigate = useNavigate();

  async function handleSubmit(e) {
    e.preventDefault();
    await createUser(new FormData(e.target));
    navigate("/users");
  }

  return <form onSubmit={handleSubmit}>...</form>;
}
```

```
// After
function CreateUser() {
  return <Form method="post">...</Form>;
}

export async function action({ request }) {
  await createUser(await request.formData());
  return redirect("/users");
}
```

Best Practices

1. **Route Organization**
    - Group related routes
    - Use nested routes for shared layouts
    - Keep loaders close to their routes
2. **Data Loading**
    - Use parallel data loading when possible
    - Implement proper error boundaries
    - Cache data appropriately
3. **Form Handling**
    - Use `<Form>` instead of `<form>`
    - Implement proper validation
    - Handle optimistic updates
4. **Error Handling**
    - Define error boundaries at appropriate levels
    - Provide meaningful error messages
    - Implement fallback UI

Common Patterns

1. **Protected Routes**

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Root />,
    loader: rootLoader,
    children: [
      {
        path: "protected",
        element: <Protected />,
        loader: protectedLoader,
        // Check auth in loader
```

```
      },
    ],
  },
]);
```

2. **Data Mutations**

```
function UserForm() {
  return (
    <Form method="post">
      <input name="name" />
      <button type="submit">Save</button>
    </Form>
  );
}

export async function action({ request }) {
  const formData = await request.formData();
  await updateUser(formData);
  return redirect("/users");
}
```

3. **Real-time Updates**

```
function LiveData() {
  const data = useLoaderData();
  const [liveData, setLiveData] = useState(data);

  useEffect(() => {
    const ws = new WebSocket("ws://...");
    ws.onmessage = (e) => setLiveData(JSON.parse(e.data));
    return () => ws.close();
  }, []);

  return <DataDisplay data={liveData} />;
}
```