Route Modules in React Router v7 are files that define everything needed for a specific route, including the behavior, UI, data loading, actions, and more. These modules serve as the foundation for React Router's framework features, supporting automatic code-splitting, data fetching before rendering (via loader functions), handling data mutations (via action functions), and defining the UI component to render when the route matches.

Each route is connected to a route module file that contains the route's logic and UI, making route management more structured and powerful. For example, a route module file may export loader functions for fetching data, action functions for handling form submissions, and a default component that renders the UI for that route. Routes and their nested structures are typically configured in a central routes file, where paths are mapped to these route module files.

This architecture enables advanced features like nesting routes, layouts, and revalidation while keeping everything related to a route self-contained in a single file. React Router v7 also supports optional file-based routing and framework features like Split Route Modules, which optimize loading and performance.

In summary, a Route Module is the core unit that defines a route's complete behavior and presentation in React Router v7, combining routing configuration, data management, and UI in one place

Route modules refer to a file-based convention where each route is represented by a single module (file) that exports everything needed for that route—such as:

- The UI component (`default export` or named `Component`)
- Loaders (`loader`) for data fetching
- Actions (`action`) for form submissions or mutations
- Error boundaries (`ErrorBoundary`)
- Meta tags (`meta`)
- Headers (`headers`)

Why Route Modules?

- Colocation: Keeps route logic (UI, data, actions) in one file.
- Simplifies data flow: No need for separate hooks or context for data fetching.
- Framework-like DX: Inspired by modern meta-frameworks (Next.js, Remix, SvelteKit).

- Built-in conventions: Reduces boilerplate and configuration.

Route modules in React Router v7 enable automatic code-splitting by allowing each route to be treated as a separate module or entry point in the build process. When a user navigates to a specific route, only the code for that route's module (and any required dependencies) is loaded, rather than loading the entire application's code at once. This reduces the initial JavaScript bundle size and improves page load performance.

The mechanism works because route modules are referenced directly in the route configuration. The bundler (like Vite) recognizes these references and creates separate bundles for each route module. When a user requests a route, only the corresponding bundle is downloaded. For example, if a user visits "/about", only the code for `about.tsx` is loaded, and not the code for other routes like `contact.tsx`. Also, server-only code (such as loader and action functions) is excluded from the client-side bundle, further reducing the size.

React Router v7.2 introduced an advanced feature called Split Route Modules, which can further split parts of a single route module (like the loader and the UI component) into individual chunks, ensuring that only the necessary code for the current action is loaded at the right time. This means components, loaders, and even actions can be split and downloaded independently, optimizing both navigation and data fetching workflows

In **React Router v7 (Framework Mode)**, each route is a file.
A route can export special functions (like `loader`, `action`) and components.
These exports tell React Router **how to fetch data, handle actions, and render UI**.

---

2. Component (default)

Every route must export a **default component** → this is what gets rendered.

```
// routes/home.tsx
export default function Home() {
  return <h1>Welcome Home 🏠</h1>;
}
```

---

3. Props passed to the Component

React Router passes some props automatically, like `params`.

```
// routes/user.$id.tsx
export default function User({ params }: { params: { id: string } }) {
  return <h2>User ID: {params.id}</h2>;
}
```

- URL `/user/101` → shows `User ID: 101`

---

4. Using props

Props can also come from **loader** or **action** return values.

```
import { useLoaderData } from "react-router";

// routes/about.tsx
export function loader() {
  return { company: "Flexibility Cloud 🚀" };
}

export default function About() {
  const data = useLoaderData<typeof loader>();
  return <p>Company: {data.company}</p>;
}
```

---

5. middleware

Middleware = runs **before route handlers**.
Example: Logging every request.

```
// routes/_middleware.ts
export async function middleware({ request }: { request: Request }) {
```

```
  console.log("Incoming request:", request.url);
  return null; // continue
}
```

---

6. clientMiddleware

Runs only on **client-side navigation**, not initial server load.

```
// routes/_middleware.client.ts
export function clientMiddleware({ request }: { request: Request }) {
  console.log("Client navigation to:", request.url);
  return null;
}
```

---

7. loader

`loader` = fetches data **before rendering**.

```
// routes/products.tsx
export async function loader() {
  const res = await fetch("https://fakestoreapi.com/products");
  return res.json();
}

export default function Products() {
  const products = useLoaderData<typeof loader>();
  return (
    <ul>
      {products.map((p: any) => (
        <li key={p.id}>{p.title}</li>
      ))}
    </ul>
  );
}
```

---

8. clientLoader

Runs only in the browser, not during SSR.

```
// routes/time.tsx
export async function clientLoader() {
  return { time: new Date().toLocaleTimeString() };
}

export default function Time() {
  const data = useLoaderData<typeof clientLoader>();
  return <p>Local Time: {data.time}</p>;
}
```

---

9. action

Handles **form submissions / mutations** (server-side).

```
// routes/contact.tsx
import { Form, useActionData } from "react-router";

export async function action({ request }: { request: Request }) {
  const formData = await request.formData();
  return { message: `Thanks, ${formData.get("name")}!` };
}

export default function Contact() {
  const data = useActionData<typeof action>();
  return (
    <div>
      <Form method="post">
        <input name="name" placeholder="Your name" />
        <button type="submit">Submit</button>
      </Form>
      {data?.message && <p>{data.message}</p>}
    </div>
  );
}
```

---

10. clientAction

Runs action only on the **client**.

```
// routes/feedback.tsx
export async function clientAction({ request }: { request: Request }) {
  const data = await request.formData();
  alert(`Feedback received: ${data.get("msg")}`);
  return null;
}
```

---

11. ErrorBoundary

Handles route-specific errors.

```
// routes/error-demo.tsx
export function loader() {
  throw new Error("Something went wrong!");
}

export function ErrorBoundary({ error }: { error: Error }) {
  return <p style={{ color: "red" }}>⚠️ {error.message}</p>;
}

export default function ErrorDemo() {
  return <h2>This will not render if error happens</h2>;
}
```

---

12. HydrateFallback

Shown while SSR data is loading/hydrating.

```
export function HydrateFallback() {
  return <p>Loading...</p>;
}
```

---

13. headers

Set custom HTTP headers for a route.

```
export function headers() {
  return { "Cache-Control": "max-age=3600" };
}
```

---

14. handle

Custom route metadata you can use anywhere.

```
export const handle = { requiresAuth: true };

export default function Dashboard() {
  return <h2>Dashboard (Protected)</h2>;
}
```

---

15. links

Add `<link>` tags dynamically (e.g., CSS).

```
export function links() {
  return [{ rel: "stylesheet", href: "/styles/dashboard.css" }];
}
```

---

16. meta

Set `<meta>` tags for SEO.

```
export function meta() {
  return [
    { title: "My App - Home" },
    { name: "description", content: "Welcome to my app!" },
  ];
}
```

---

17. shouldRevalidate

Controls when the loader refetches.

```
export function shouldRevalidate({ currentUrl, nextUrl }: any) {
  return currentUrl.pathname !== nextUrl.pathname; // only if route changes
}
```
