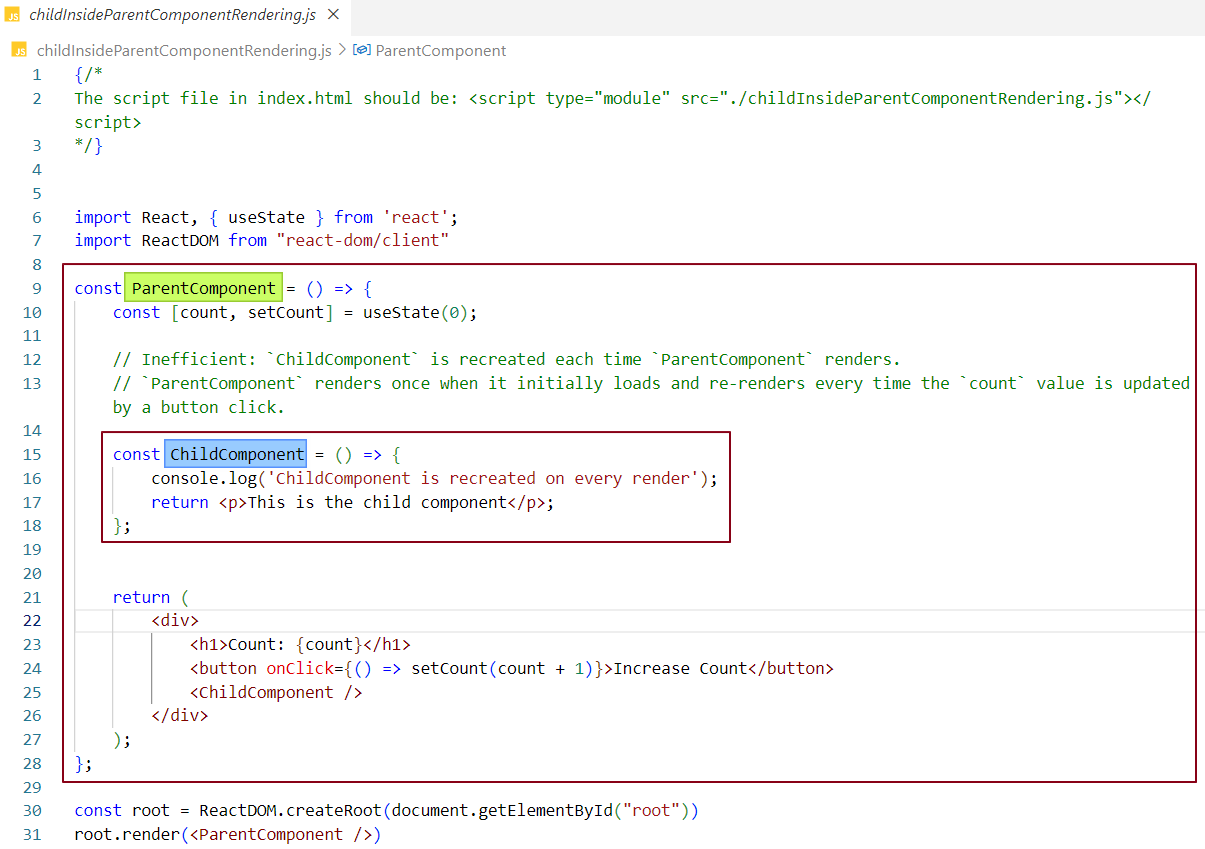
Chapter 7: Routing

React Guidelines

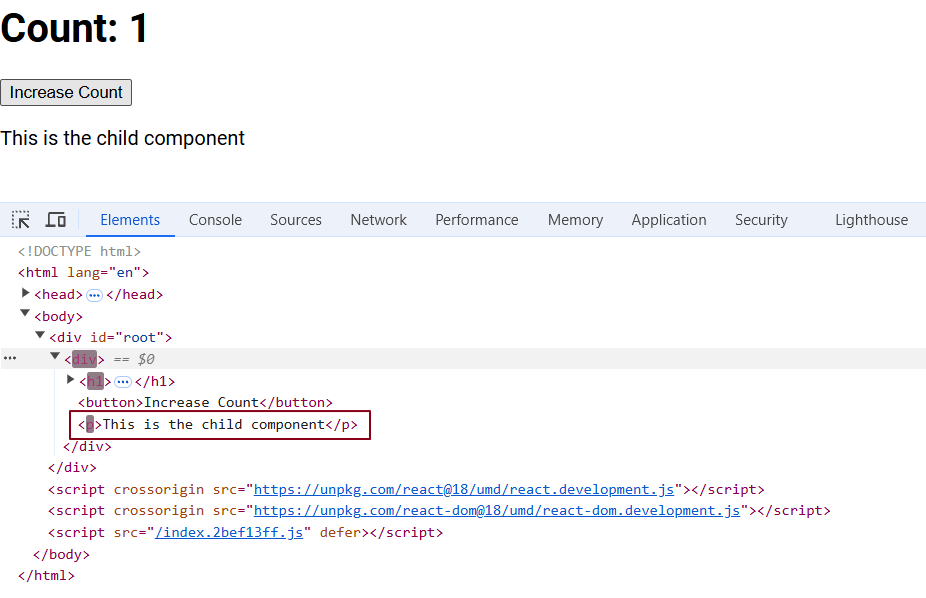
### **🚫 Don't define a component inside another component**

* If you define a **child component inside a parent**, it will be **created again every time** the parent updates.
* This is **bad for performance** and can use more memory.



In the image, ChildComponent is written **inside** ParentComponent.

* When you click the button to increase the count, ParentComponent re-renders.
* Because ChildComponent is **inside** it, it also **gets recreated** every time.
* That’s why you see a message in the console and the <p> flashing—it shows it's re-rendering too.

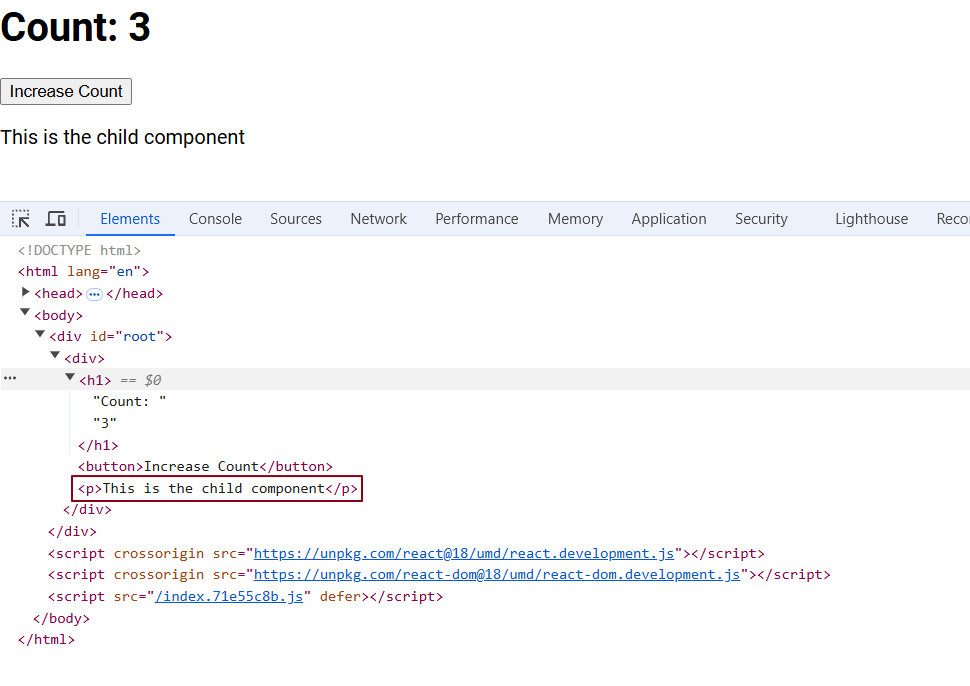


**✅ Better Way:**

Move ChildComponent **outside** of ParentComponent so it's not recreated on every render. This makes your app **faster** and avoids **unnecessary work.**



* In this example, we placed ChildComponent outside of ParentComponent.
* This means ChildComponent is created only once when the app loads.
* When you click the button and ParentComponent re-renders, ChildComponent does not get recreated.
* That’s why you only see the log for ChildComponent **once** when the app first loads — after that, it doesn't re-render or log again during parent updates.



**🚫 Never create useState variables inside an if-else block in React. Always call hooks like useState unconditionally.**

In React, **hooks** like useState must always be called in the same order on every render. This is because React relies on the **order of hook calls** to correctly track state, effects, and other hooks.

If you place useState inside an if-else block, it can be conditionally executed based on the condition. This breaks the rule because in some renders, useState may be called, and in others, it might not. React won't know if the state variable exists during the render when the condition isn't met because the hook wasn't called, leading to errors when trying to access it.

In the example above:

* If Math.random() is less than or equal to 0.5, useState will not be called, so message will be undefined, and React won’t know if it exists.
* This results in a **reference error** when trying to access message because React couldn't track it properly.

### **Fix**:

To avoid this issue, **always call** useState **unconditionally,** ensuring it's called in every render.



This way, react can always track the state properly, regardless of the condition.

### 🚫 **Never create useState inside a for loop**.

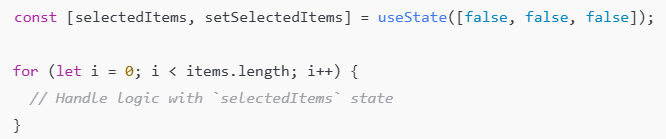
If you place useState inside a loop, it will create a new state variable each time the loop executes, even though you only need **one** state variable. This breaks React's hook rules and can lead to confusion and errors.

 In this example:

* useState gets called multiple times—once for every iteration of the loop.
* **+** but this approach creates unnecessary variables every time the loop runs.

### **Correct Approach:**

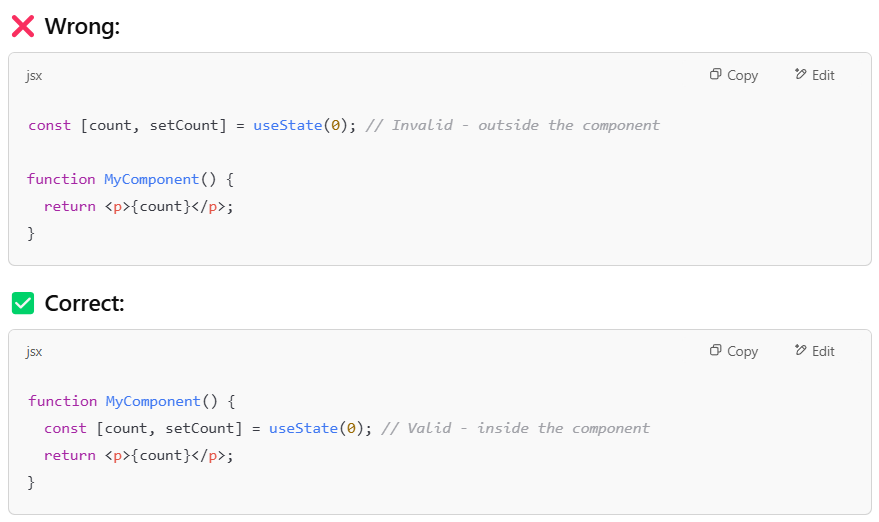
Always define the useState **outside** of the loop to ensure that **only one state variable** is created.



This way, react can manage **one state variable** efficiently, as intended.

🚫 **Never declare useState outside a function component.**

Because such variables won’t be tied to the component’s lifecycle. They won’t trigger re-renders or update the UI.  
To work properly, useState must be declared **inside** the function component.



Routing

Routing is the process of telling a web application what to show when a user visits a specific URL.

Think of it like a map for your app. If the user goes to /home, it shows the Home page. If they go to /profile, it shows the Profile page.

In modern web apps especially single-page applications (SPAs), routing lets users move between pages without reloading the entire site. This makes the app feel faster and more seamless.

### **What is Client-Side Routing?**

**Client-side routing** is when the browser handles **navigating between pages** of a website without refreshing the entire page.

Here’s how it works:

* When you click a link or type a new URL**, JavaScript in the browser** takes over.
* **It updates the URL** in the address bar, but it doesn’t reload the entire page.
* **Only the content** that needs to change is updated, so the page feels fast and smooth.

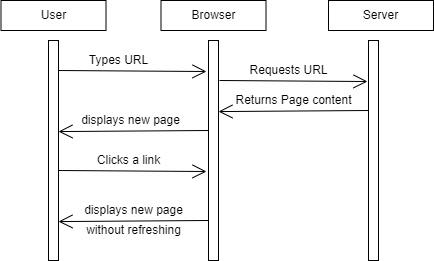
This is especially common in **Single Page Applications (SPAs)** where the app loads once, and navigation happens dynamically—meaning the page content changes without full reloads.

### **✅ Benefits**

* **Faster Navigation:** Only the content that changes is updated—not the whole page.
* **Smooth User Experience:** Feels like using an app rather than a website.
* **Great for SPAs:** The app loads once, and routing handles switching views without page reloads.
* **Flexible:** Routes can change based on user interactions (like clicking a button).

**❌ Drawbacks**

* **SEO Issues:** Search engines may have trouble reading dynamic JavaScript pages because they rely on static HTML to index content, and JavaScript-generated content may not be visible to crawlers, which are bots used by search engines to scan and index web pages.
* **Slower First Load:** Since the entire app loads upfront, the initial load can feel heavy.
* **Requires JavaScript:** If JavaScript fails, routing won't work.



### **What is Server-Side Routing?**

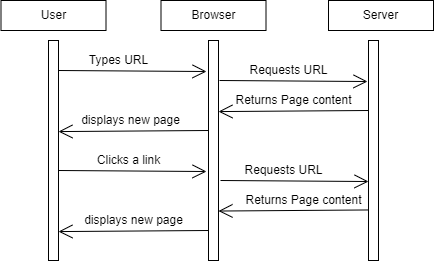
### In **server-side routing**, the server handles all the routing. When a user requests a new route (e.g., a page or a URL), the server sends back a completely new HTML page corresponding to that route.

**Benefits:**

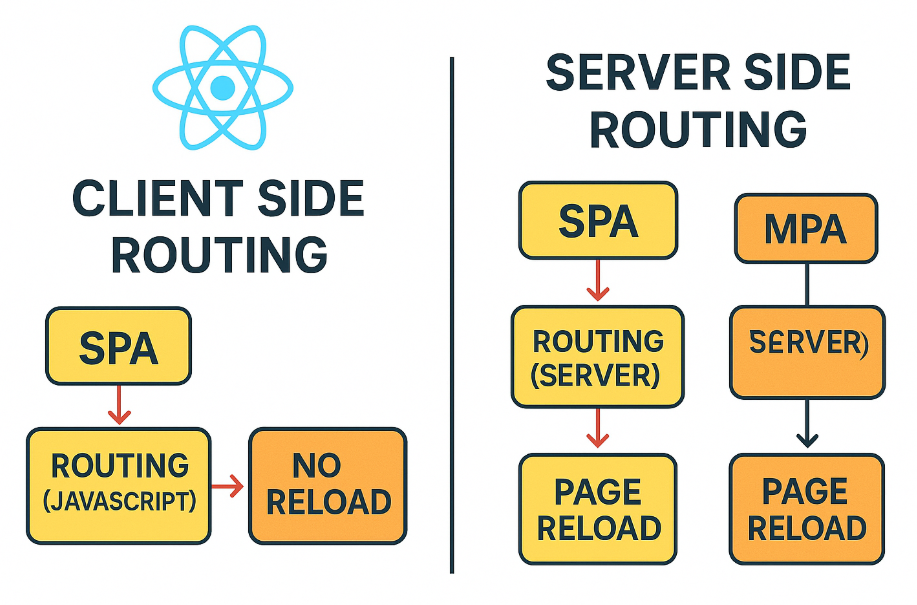
* **SEO-Friendly:**  
  Since each route corresponds to a distinct HTML page, search engines can easily crawl and index the content, which improves search rankings.
* **Simplicity:**  
  It's easier to implement for traditional websites or simpler applications where each page is static or has minimal routing needs.
* **Quick Initial Load:**  
  The server sends fully-rendered HTML pages, so the first page load can be faster because the browser doesn't need to load all the JavaScript to render the page.

**Drawbacks:**

* **Slower Transitions:**  
  Page transitions may be slower because each navigation requires the browser to reload the entire page, which can take time.
* **Less Interactivity:**  
  The user experience can feel less fluid compared to client-side routing, where content can be updated without reloading the page, making transitions feel jarring or less dynamic.



**Client-Side Routing vs. Server-Side Routing**



### **Client-Side Routing (SPA) -**Routing is handled in the browser using JavaScript, enabling smooth transitions without page reloads.

### **Server-Side Routing in SPA-** Each route change triggers a server request and full page reload, even in a single-page app.

### **Server-Side Routing (MPA) -** Every user action loads a new HTML page from the server, making it a traditional multi-page experience.

### **When to Use Client-Side Routing?**

* **Best for:** Single Page Applications (SPAs) and dynamic applications.
* **Advantages:** Faster transitions and smoother user experience.
* **Disadvantages:** May struggle with SEO as content is loaded dynamically.

### **When to Use Server-Side Routing?**

* **Best for:** Traditional websites and applications where SEO is a priority.
* **Advantages:** Better SEO, as each page is fully rendered by the server.
* **Disadvantages:** Can result in slower page transitions and less interactivity.

### **What is a SPA (Single Page Application)?**

A **Single Page Application (SPA)** is a type of web app that loads a single HTML page and dynamically updates the content **without refreshing the entire page**. This makes the app feel fast and smooth, more like a desktop application.

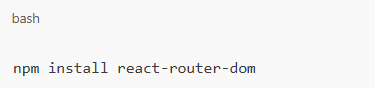
**Key Characteristics of a SPA:**

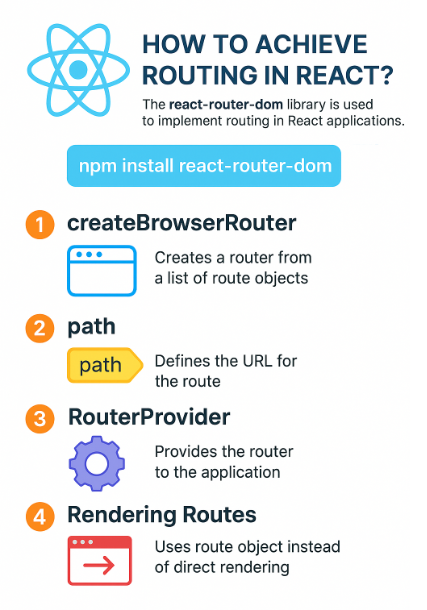
* **Dynamic Updates:** Only parts of the page update when needed—no full reloads. JavaScript and client-side routing handle it.
* **Smooth User Experience:** Fast and responsive navigation between views, giving a seamless feel to users.
* **Faster After First Load:** Initial load may be heavier (due to all JavaScript and assets), but after that, navigation is fast because only data is exchanged.
* **Client-Side Routing:** Uses tools like **React Router** or **Vue Router** to switch views by updating the URL without reloading the page.
* **API-Centric:** Communicates with the backend using APIs (usually via **JSON**), keeping the front end and back end separate.
* **State Management:** Uses libraries like **Redux** (React) to manage and track app data across components.

**In short:**  
A SPA makes your web app fast, dynamic, and smooth—like a native app—by loading content on the fly without full page reloads.

### **How to Achieve Routing in React?**

To add routing in a React app, we use the react-router-dom library.  
📦 Install it using:





**Key Features of** react-router-dom**:**

**createBrowserRouter Function**

* Used to create the router instance for your React app.
* Accepts an array of route objects, where each object defines:
  + path: the URL pattern (e.g., /, /about)
  + element: the component to render when the path matches.

**Route Configuration**

Each route object inside createBrowserRouter() defines which component to show for a specific URL.



**RouterProvider Component**

* Used to inject the router into your React application.
* Wrap your root component using <RouterProvider router={appRouter} /> to enable routing.

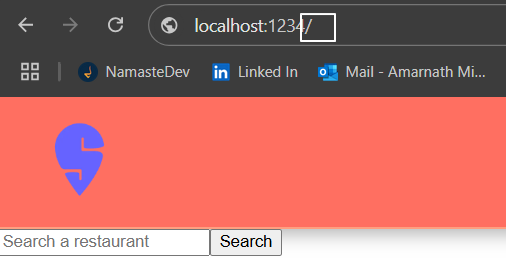
**Rendering the Routes**

Instead of rendering a specific component directly via ReactDOM.render(), now you render the RouterProvider:

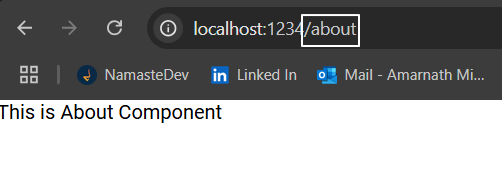




Now, when we navigate to the base URL **http://localhost:1234/**, the AppLayout component is displayed.

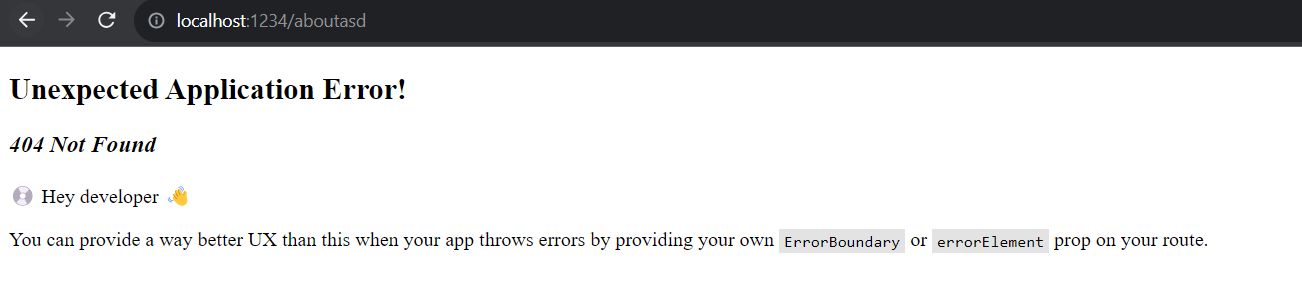


When we navigate to the About page URL **http://localhost:1234/about**, the About component is rendered.

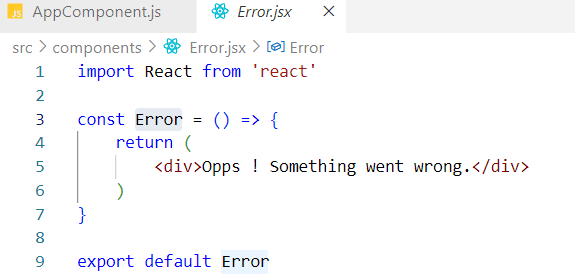


What happens when a user navigates to a URL that does not exist?

react-router-dom is a powerful library that also handles errors when users navigate to a route that doesn't exist. In such cases, it can display a custom 404 error page to inform users that the requested route is unavailable.

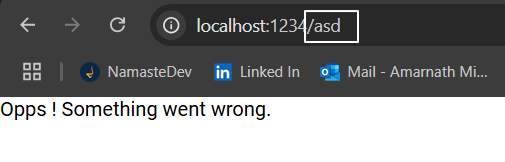


We can show a custom error page by using the errorElement property in the route setup. This lets us show an error component when a user goes to a URL that doesn’t match any of the routes we have defined. Let’s make an Error component and add it to the errorElement in our route settings.



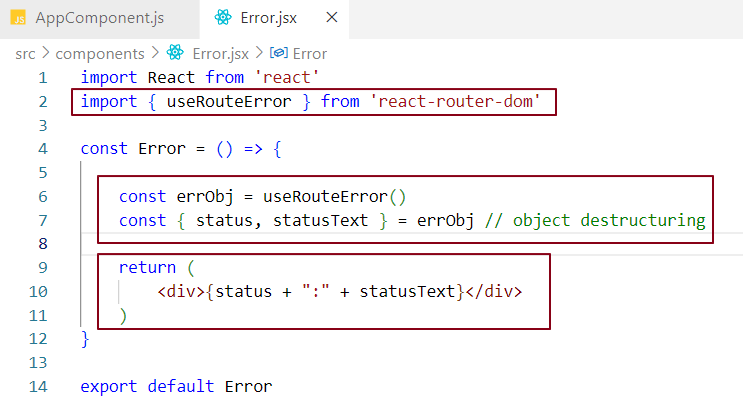


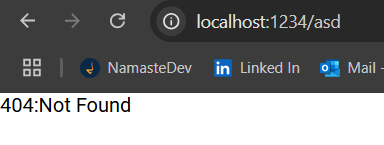
When a user visits an invalid URL, the error component we set will be shown on the screen.



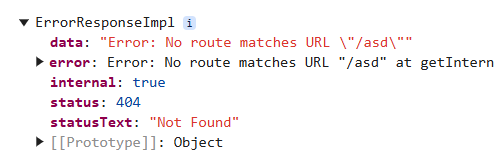
How can we display additional information about the error when a user navigates to an invalid URL?

react-router-dom provides a hook called useRouteError, which returns an error object describing the type of error that occurred during routing.





If we log the errObj to the console...



So far, we've been manually typing the URL to navigate to the desired component. However, in a real-world application, clicking on links or hyperlinks should take users to the correct URL. Let's implement this functionality using the Link component.

What is a Link component?

In the react-router-dom library, the Link component is used to create navigation links that allow users to move between different routes or pages within a single-page application (SPA). The Link component replaces the traditional <a> tag by enabling navigation without page reloads, making it ideal for single-page applications (SPAs) with seamless routing.

### Why Do We Use the Link Component in React?

In React applications, we often need to navigate between different views or routes. The **Link** component from **react-router-dom** is used for handling navigation in **Single Page Applications (SPA)**, offering several benefits over traditional anchor (<a>) tags.

#### **1. Anchor Tags Trigger Full Page Reload**

* When using an anchor tag (<a>) with an href attribute for navigation, clicking the link makes the browser send a request to the server for the specified URL.
* This causes a **full page reload**, meaning the browser reloads the entire page from the server, including all resources (HTML, CSS, JavaScript, etc.).
* **Drawback**: Full page reloads result in **slower navigation**, increased load times, and a less smooth user experience because the page must re-render and reset its state.

#### **2. Client-Side Routing with Link**

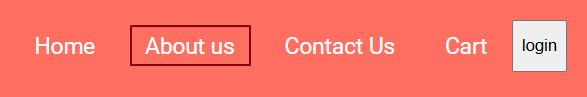
* React applications use **client-side routing** to enable fast and smooth navigation. The **Link** component from **react-router-dom** is the core tool for this.
* When you use the **Link** component, react handles the URL changes internally. Instead of reloading the entire page, react updates the content dynamically, without fetching the page from the server.
* **Key Benefit**: This results in **faster transitions** and a **more responsive interface** because only the components related to the new route are re-rendered, and no page reload occurs.

How do we configure Link component for routing?

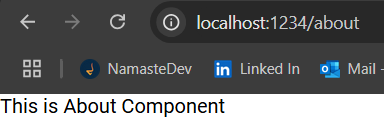
In the Link component from react-router-dom, we use the **"**to**"** prop to specify the target route. This to prop defines the path the application should navigate to when the link is clicked, enabling client-side navigation without a full page reload.



Now, when the user clicks on the **"About Us"** link, the /about path defined in the **"**to**"** prop is matched against the route configuration in the appRouter.  
If a match is found, React Router renders the corresponding component associated with that path.  
This is exactly what happens in our case clicking the link triggers the route match and loads the appropriate view **without reloading the page.**



When the **"About Us"** link is clicked, the /about route is matched, and the **About** component is rendered dynamically in the UI without reloading the page, thanks to React Router’s client-side routing.

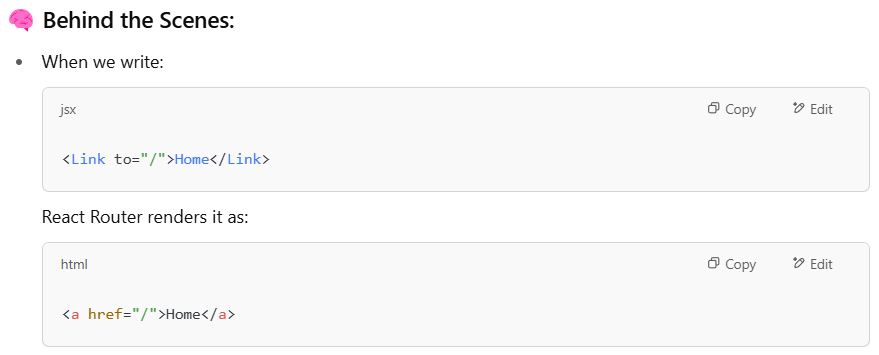


### 🔗 How the Link Component Works in React Router ?

When we inspect the page in the browser, we observe that the Link component from react-router-dom is rendered as a standard HTML <a> (anchor) tag—not as a custom Link tag. This behavior is intentional and necessary for compatibility with the browser.

**✅ Why It Happens:**

* The browser only understands native HTML elements like <a>, not React components like Link.
* To work within these constraints, the Link component **internally renders** an <a> tag and attaches **client-side navigation logic** to it.
* React Router intercepts the click event on this anchor tag, prevents the default full-page reload, and handles the navigation through JavaScript.



But instead of letting the browser reload the page, it:

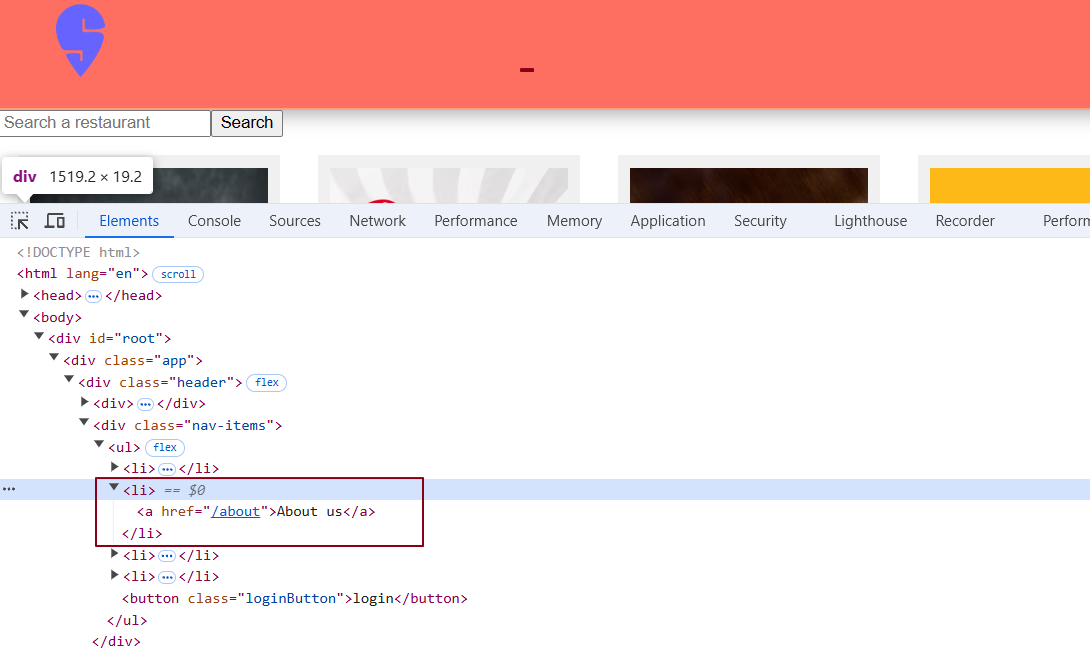
* Prevents the default browser behavior.
* Updates the URL.
* Loads the new route's component **without reloading** the entire page.

**Notes -**  
**Client-side navigation logic** refers to React Router using JavaScript to update the URL and render components without triggering a full page reload.

**Intercepting the anchor tag** means preventing its default behavior (page reload) and handling navigation via JavaScript.

### **If anchor (**<a>**) tags cause full-page reloads, and React Router's** Link **component eventually renders as an** <a> **tag, doesn't that mean it still triggers a page reload (server-side routing)?**

### **No, it doesn’t.** Although the Link component renders as an <a> tag in the DOM, **React Router intercepts the click event** using JavaScript before the browser can perform a default full-page reload. This interception allows React Router to update the URL and render the new component **without refreshing the page**, enabling fast **client-side navigation**.



### **What is Nested Routing?**

**Nested routing** allows child components to be placed inside a parent component. This means that when navigating to different sections of the app, only the child component updates, while the parent component (like the header or footer) stays the same, preventing a full page reload.

**Why Use Nested Routing?**

**Nested routing** helps maintain a consistent user experience by ensuring that parts of the UI, like the header and footer, stay visible while only the main content changes.

Without nested routing, when you navigate to a new page (like the About page), the whole page, including parts like the header and footer, gets reloaded. This means the entire layout, including the static parts (header, footer), will refresh even though they haven't changed.

This can make the navigation feel slower and less smooth because you’re reloading the entire layout instead of just updating the content inside the main area.

With nested routing, only the main content area (like the About page content) updates, while the static parts (like the header and footer) stay unchanged, leading to a faster and more seamless user experience.

### Without Nested Routing (Flat Routing)

**Problem:** Every route (like / and /about) defines its **own layout,** so Header and Footer get **re-rendered or duplicated**, and there's **no shared structure**.



**Behavior:**

* Header and Footer are copied manually in every route.
* Not DRY (Don't Repeat Yourself).
* Updating layout (e.g., adding a sidebar) needs changes in multiple places.
* Navigating between routes like / ➝ /about re-renders the entire layout (Header/Footer are recreated).
* State inside Header/Footer (like a dropdown open/close) will be lost on navigation.

### With Nested Routing

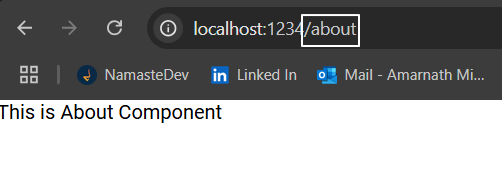
**Benefit:** Layout (Header, Footer) is defined **once**, and only **inner content** (like Body, About) is rendered dynamically using <Outlet />.



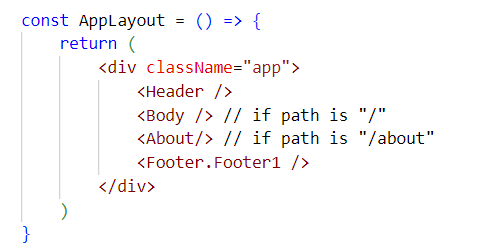
**Behavior:**

* Header and Footer are defined once in AppLayout.
* Only the part inside <Outlet /> changes with navigation.
* Cleaner, DRY, easier to maintain and scale.
* Navigating to /about preserves layout - Header/Footer are not re-rendered.
* Any state in Header or Footer is retained.

With our current setup, navigating to the **About** component does **not render the Header and Footer,** which can disrupt the overall navigation experience and layout consistency.



In the screenshot below, when the path is "/", the **Body** component should be displayed between the **Header** and **Footer**. Likewise, when the path is "/about", the **About** component should also be rendered within the same shared layout.



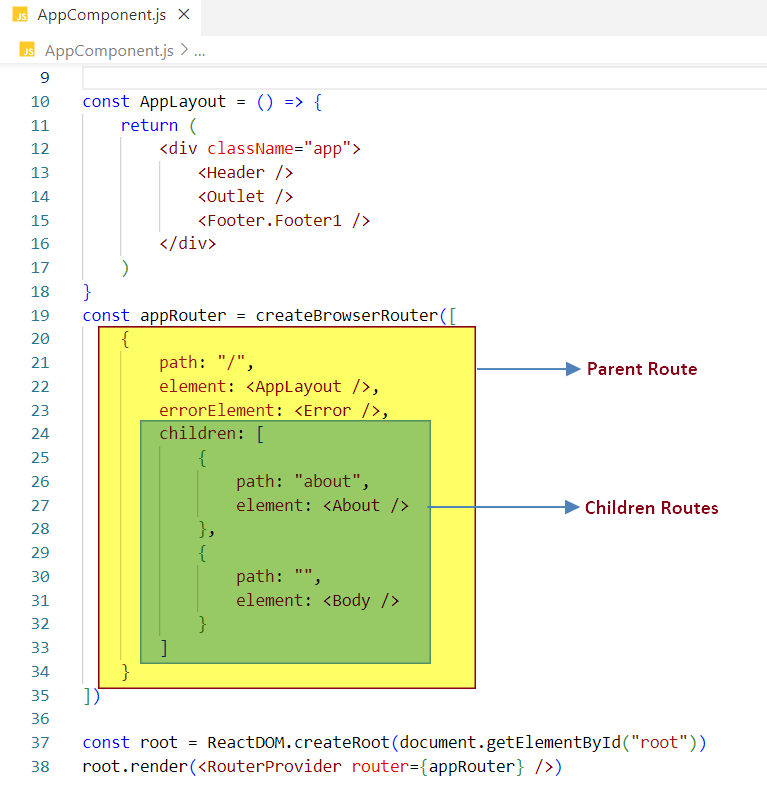
We want the **Header** and **Footer** components to remain consistent across all pages, while the content between them updates based on the current route.

How Do We Achieve Nested Routing in React?

To implement nested routing in React using the react-router-dom library, we define a **parent route** that contains **child routes.** This setup allows shared components like the header and footer to remain persistent across different pages, while only the main content area updates during navigation.

### **Implementation in Our App**

In our setup, we nest the About and Body components as children of the AppLayout component. The <Outlet /> inside AppLayout acts as a placeholder where the active child route is rendered based on the current path.



### 🔗 Route Structure

#### **Parent Route**

* **Path:** /
* **Component:** AppLayout
* **Purpose:** Renders shared layout components (Header, Footer) and an <Outlet /> for injecting child routes.
* **URL:** http://localhost:1234/

#### **Child Routes**

1. **About Page**
   * **Path:** about
   * **Component:** About
   * **URL:** http://localhost:1234/about
2. **Home (Default Page)**
   * **Path:** "" (empty string)
   * **Component:** Body
   * **URL:** http://localhost:1234/

### 🔁 Routing Flow

#### 1. Initial Load (http://localhost:1234)

* Matches the parent route /
* Renders the AppLayout with Header, Footer, and an <Outlet />
* The <Outlet /> displays the Body component since it matches the default (empty) child route.

#### 2. Navigating to About (http://localhost:1234/about)

* Still matches the parent route /
* AppLayout continues to render (with Header and Footer)
* <Outlet /> now renders the About component based on the child route /about

### **What is** Outlet **in** react-router-dom**?**

Outlet is a component provided by react-router-dom that acts as a placeholder for rendering child components inside a layout component. It is particularly useful for rendering nested routes while preserving the overall layout during page navigation.

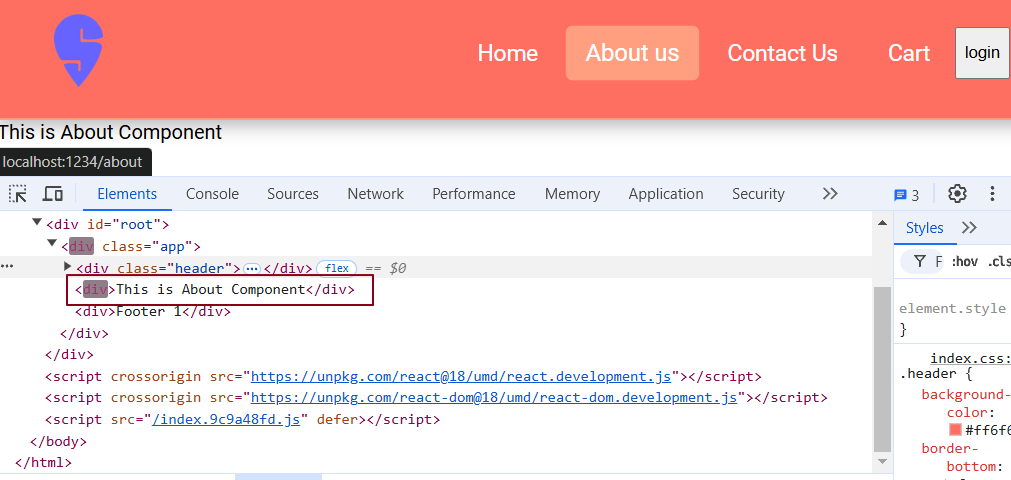
**Key Features of Outlet:**

* **Placeholder for Child Routes:** Outlet serves as a placeholder where child components are rendered based on the current route. It is typically placed inside a layout component (e.g., AppLayout).
* **Dynamic Content Rendering:** When you navigate to different routes, react triggers a process called **reconciliation**, updating only the parts of the DOM that have changed. This means the content inside the Outlet is updated, while the rest of the layout (like headers or footers) remains unchanged.
* **Nested Routes:** Outlet allows you to structure your routes hierarchically, enabling the rendering of nested routes within a shared layout.

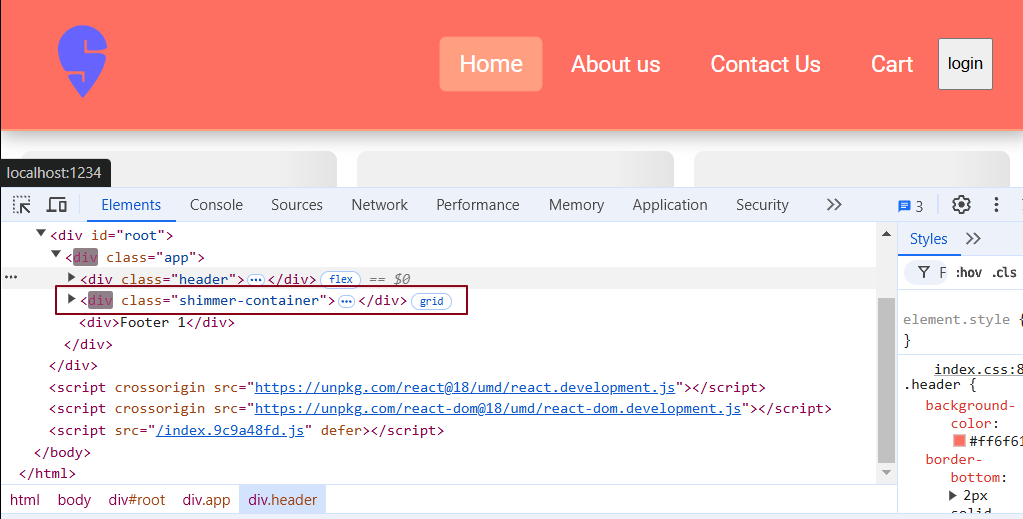
**How Outlet Works:**

When you navigate to a different page, react determines which component to render inside the Outlet based on the active route.

For example, when you click on a link (e.g., "About Us"), React will render the About component inside the Outlet of the AppLayout, keeping the rest of the layout intact.



When we click on "Home," the Outlet renders the Body component inside the AppLayout component.



Note: The highlighted text in the browser’s developer tools indicates that reconciliation is taking place.

So far, we have configured routing for the header navigation links, which are static routes. This means the routes are predefined, and we navigate directly to these specific links. However, in a real-world application, we might fetch a list of restaurant cards from an API and display them in the UI.

When a user clicks on any restaurant card, we should navigate to a component that displays the details for that specific restaurant. Since the number of restaurant cards is dynamic and can change, dynamic routing becomes essential. Dynamic routing allows us to create routes that change based on user interactions. For example, by clicking on a restaurant card, we can dynamically navigate to the corresponding component to display the details of the selected restaurant.

## 📘 What is Dynamic Routing?

**Dynamic routing** is the process of setting up routes **at runtime**, based on **API data, user actions**, or **other dynamic inputs**,instead of predefining them when the app starts.

**🔑 Key Characteristics**

* Routes are generated on the fly.
* They change based on API responses or user behavior.
* Allows flexible navigation paths like /restaurant/101 or /product/xyz.

**💡 Why Use Dynamic Routing?**

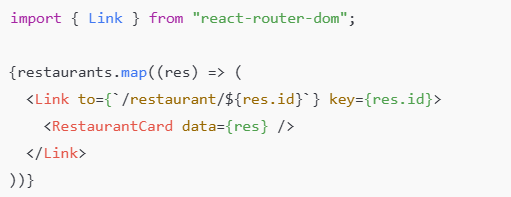
* Scales easily: No need to manually define hundreds of routes.
* Adapts dynamically: Routes adjust to what the user does or what data the API provides.
* Ideal for real-world apps: Like restaurants, products, blogs, or user profiles.

🔁 How It Works with API Data?

Suppose your app fetches restaurant data from an API



You use this data to dynamically generate links

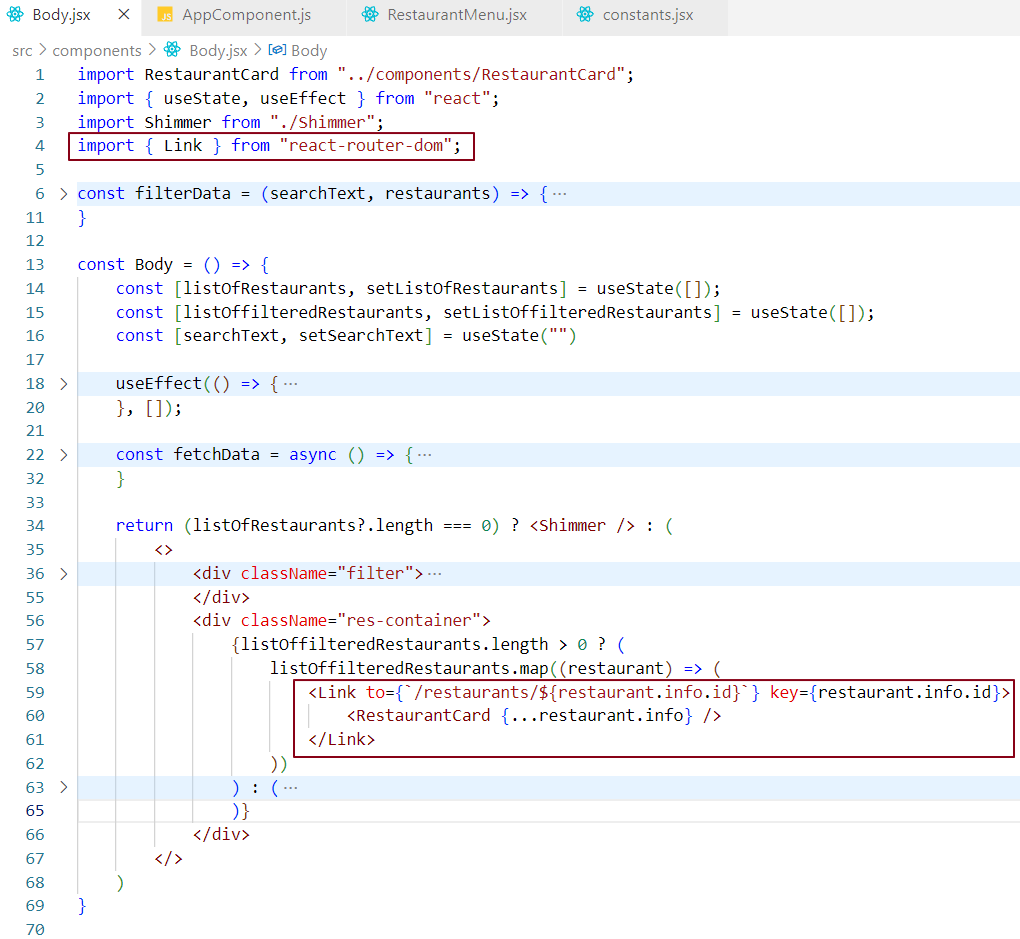


Each link takes the user to a route like /restaurant/101.

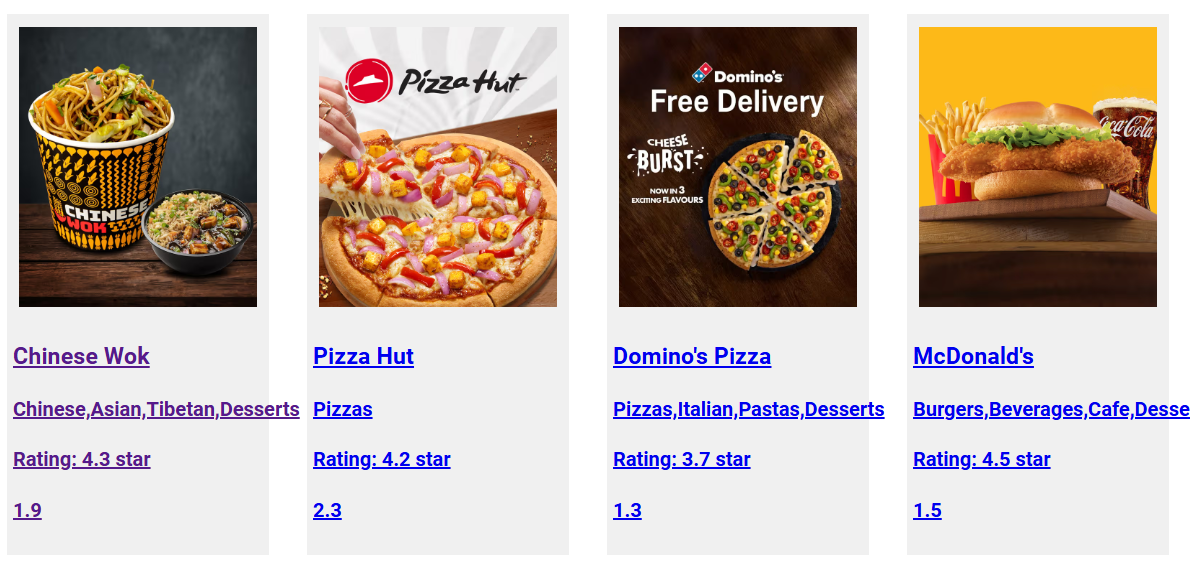
You handle these dynamic routes in your route config:



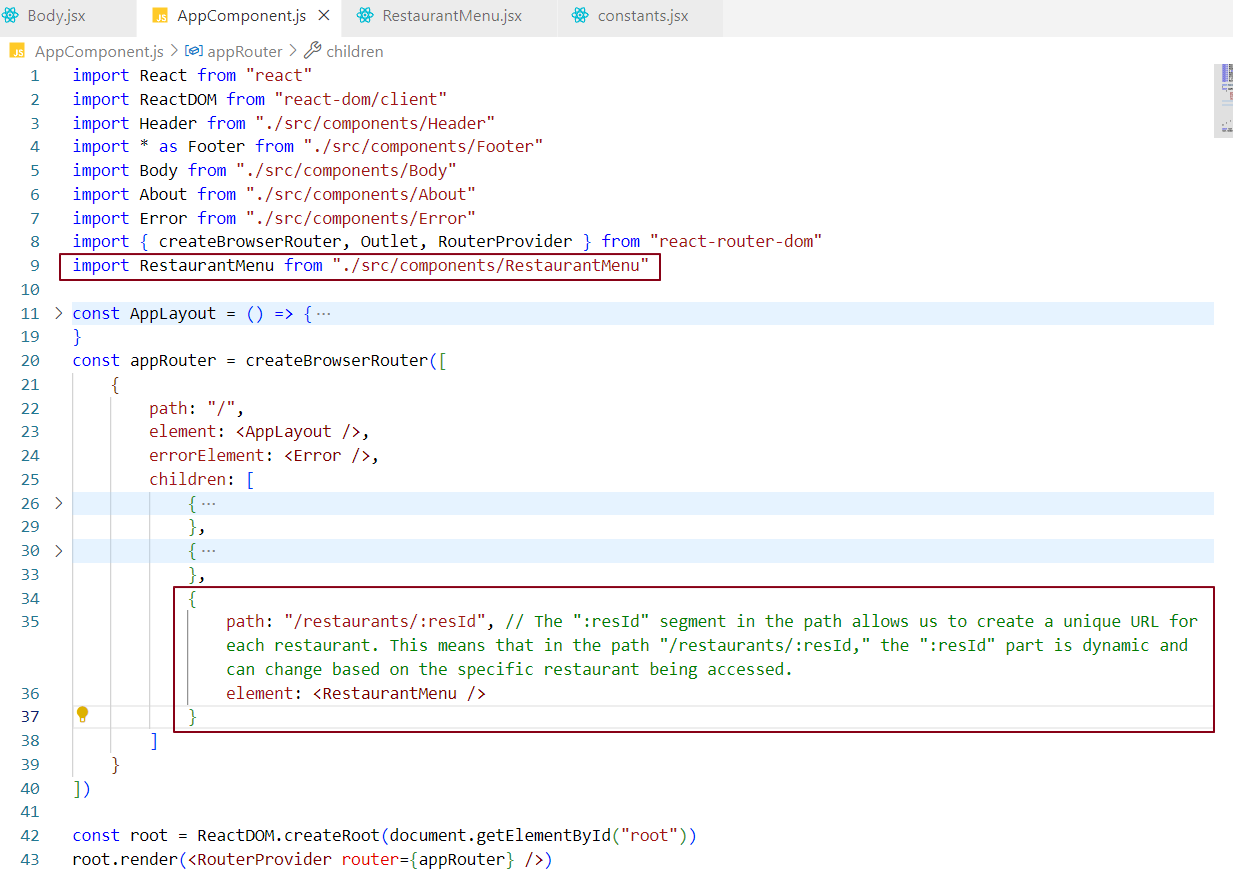
Now, let’s integrate dynamic routing into our code.  
Up until now, we've been displaying a list of restaurants in the Body component, but the restaurant cards are not clickable. To enable navigation, we’ll wrap each restaurant card in a Link component, allowing users to click and view details for a specific restaurant.



**Note:** We've moved the key from the RestaurantCard component to the Link component.  
The key should always be assigned to the outermost (parent) element in a list — in this case, the Link — to ensure optimal rendering performance and avoid React warnings.



As we can see, the restaurant cards are now clickable.  
If we look at the code, we’re passing the path /restaurants/${restaurant.info.id} to each Link component.  
This dynamic path matches the route configuration defined in the appRouter, allowing each card to navigate to its respective restaurant details page.

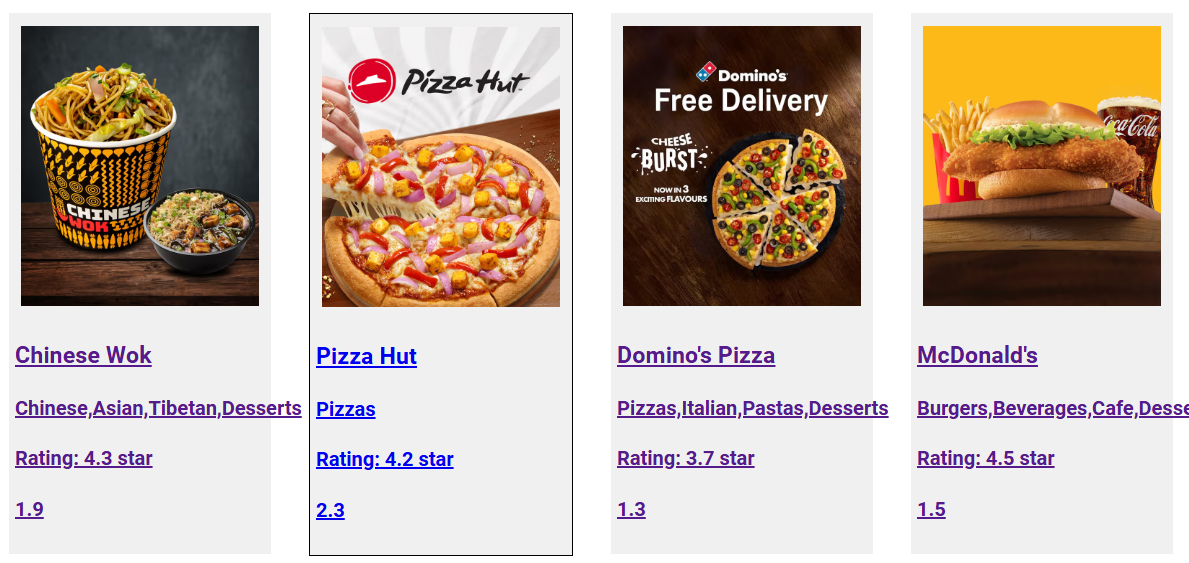


This route configuration is considered dynamic because we don’t know the value of resId at compile time.  
The resId is determined at runtime, when a user clicks on a restaurant card. At that point, resId holds the unique ID of the selected restaurant, which can be accessed using the useParams hook provided by React Router DOM, as we’ll see later.

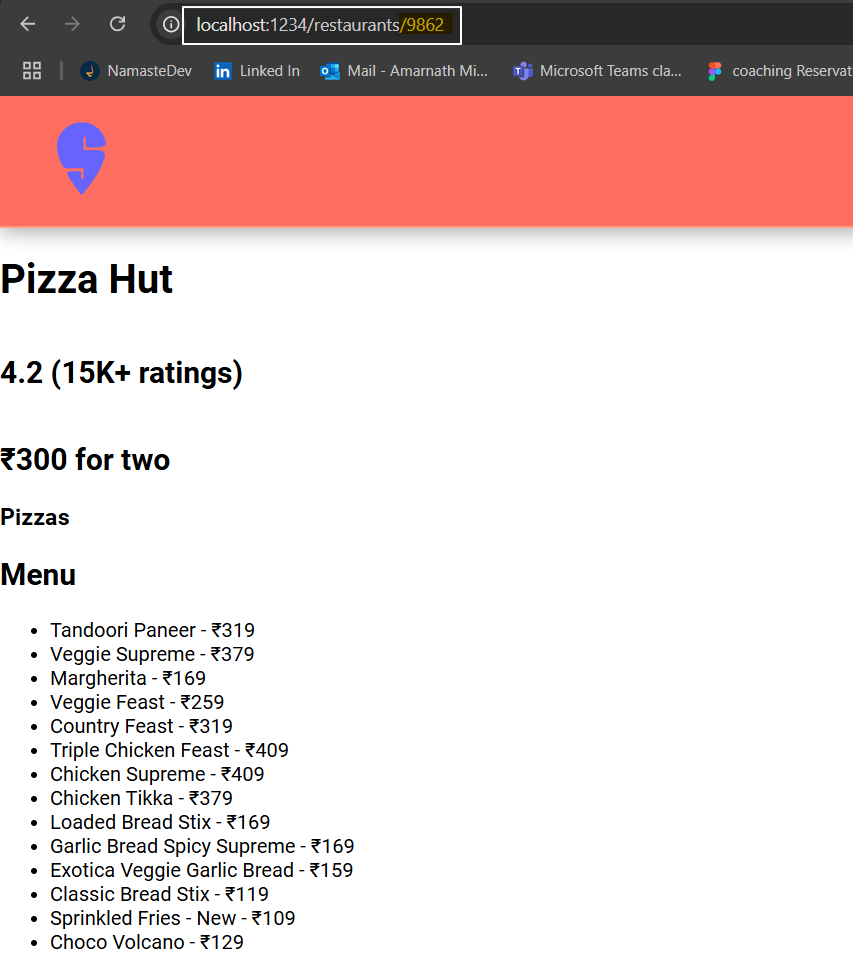
Now, let’s create the RestaurantMenu component, which will display the menu specific to the selected restaurant.



In this component, we fetch the resId using the useParams hook, which will be used in the API call to retrieve the restaurant's menu data. While the data is being fetched, we display a shimmer effect as a loading indicator. Once the data is successfully loaded, we show the restaurant menu along with the restaurant's details.



When I click on a restaurant card, such as the **Pizza Hut** card, the restaurant’s unique ID (e.g., 9862) is added to the URL. The **Restaurant Menu** component captures this ID using the useParams hook from React Router. Using this ID, the component makes an **API call** to fetch the restaurant's menu data. Once the data is fetched, it’s displayed in the UI along with the restaurant’s details.



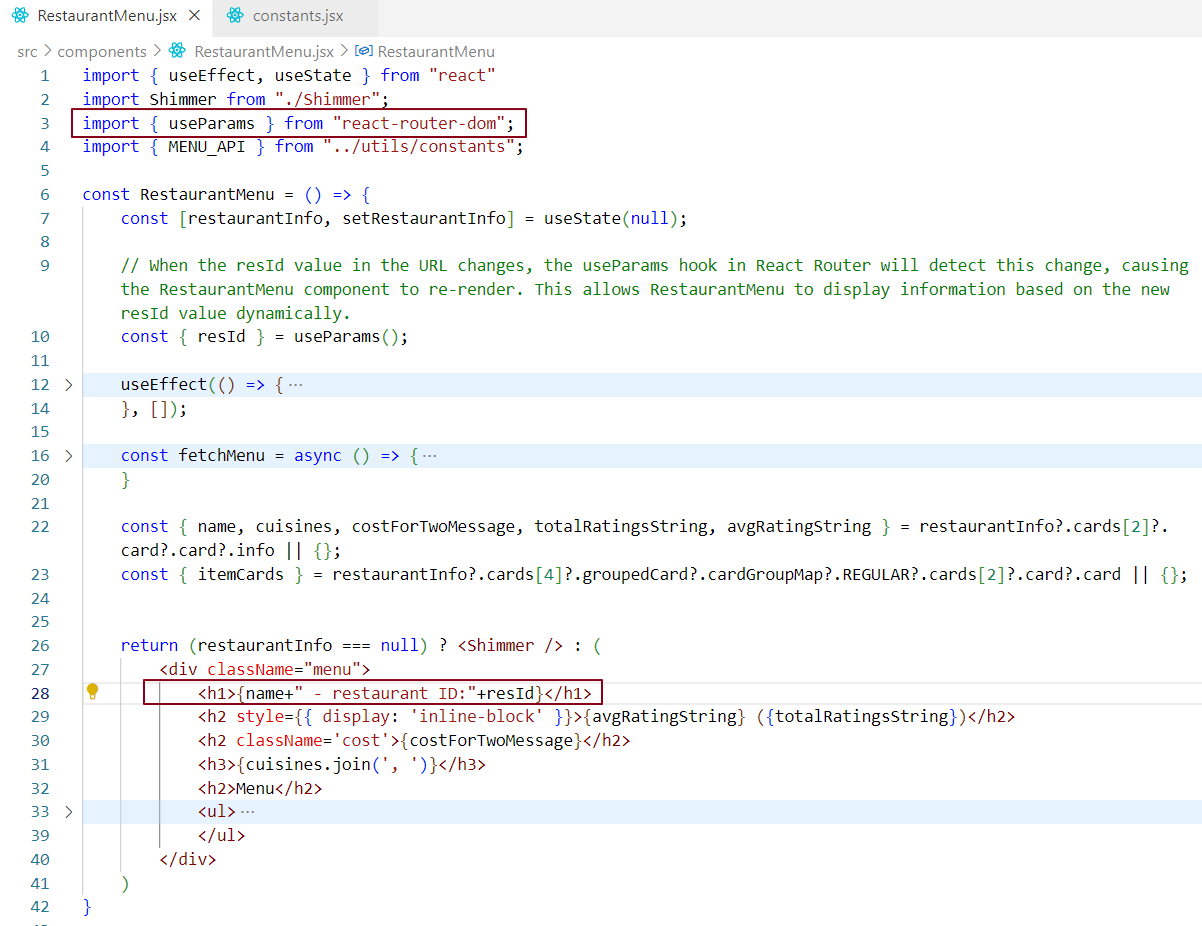
### How is the ID being read from the **Restaurant Menu** component?

The **Restaurant Menu** component uses the useParams hook provided by **React Router DOM** to retrieve the restaurant's unique ID from the URL.  
This ID is then used to make the API call to fetch the relevant data for that specific restaurant.

What is useParam hook in react?

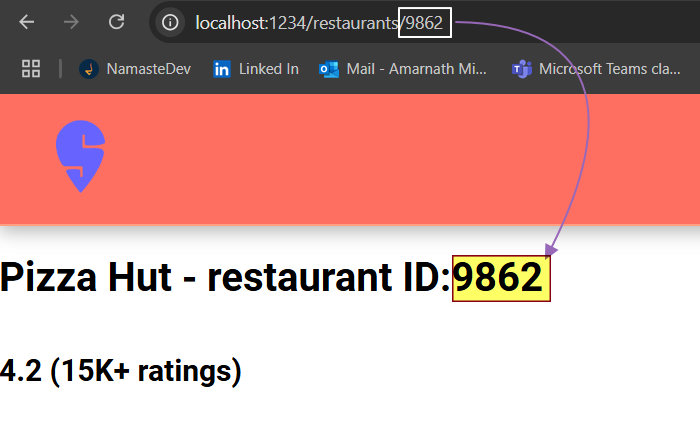
The useParams hook is provided by the react-router-dom library in React.  
It allows you to **access the dynamic parameters** from the current URL.  
These parameters are typically defined in your route path using a colon syntax (e.g., /user/:id).

**Example:**



When the RestaurantMenu component loads, it uses the useParams hook from react-router-dom to extract the resId value from the URL.  
This resId is then used to **fetch the menu data** for the corresponding restaurant (e.g., from an API or a backend service).

Once the data is successfully fetched and the state is updated using the fetched data, react triggers a re-render of the component. This re-render updates the UI to display the restaurant’s information including the resId.



Why is a CDN a great option for hosting images?

* CDNs are fast - They deliver images from edge servers located geographically closer to the user, reducing latency.
* CDNs provide high availability - While not always *100% uptime*, they are built with redundancy and load balancing, which ensures very high reliability.
* CDNs optimize and cache images - They compress and resize images based on the device/browser, and cache them to reduce repeated server requests.
* Scalability - CDNs can handle large volumes of traffic efficiently without affecting performance.
* Reduced load on origin server - Since images are served from edge servers, the main server doesn't get overwhelmed.

That's why large-scale apps like Swiggy and Zomato use CDNs for hosting images—to ensure fast, reliable, and optimized delivery to users.

### ✅ What will be logged if we run console.log(useState())?

If you call console.log(useState()) inside a **React component,** you will see:



* The first element (undefined) is the **initial state value,** because you didn't pass any argument to useState().
* The second element is the **state updater function** (commonly called setState), which you use to update the state.



### **Important Note:**

This works only **inside functional components or custom hooks**.  
If you try to run useState**() outside of a component,** you'll get an error:

