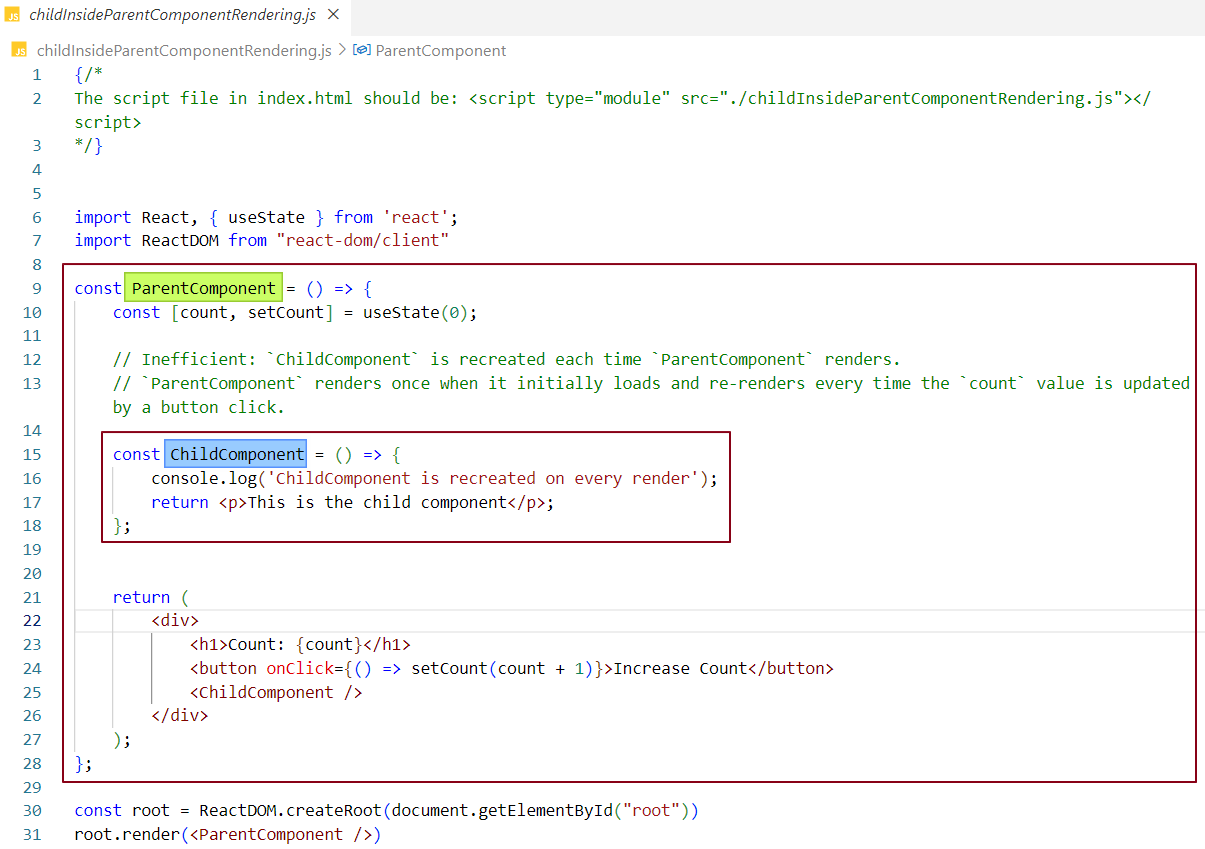
Chapter 7: Routing

React Guidelines

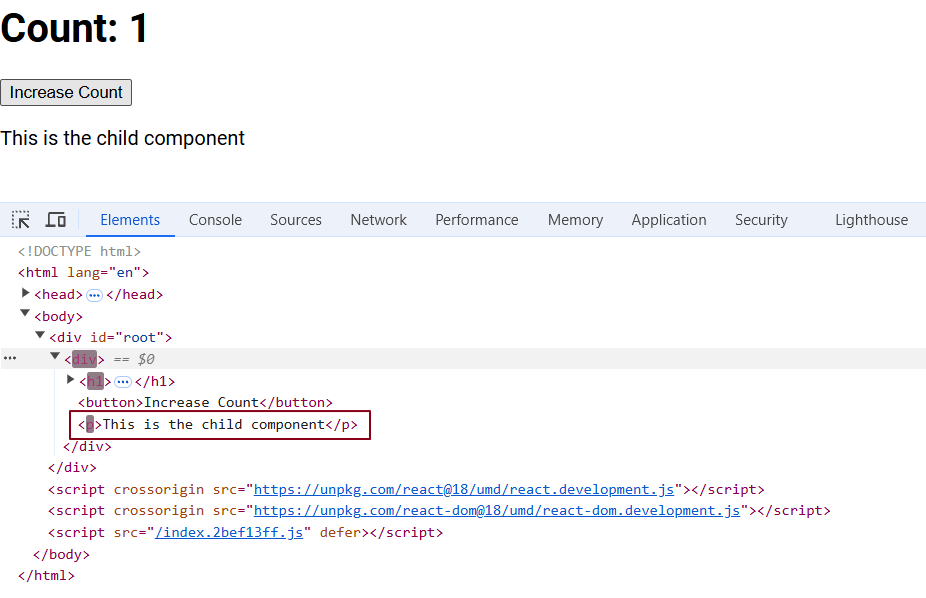
**1. Never define a component within another component-**

Defining a component inside another component causes the child component to be recreated every time the parent renders or re-renders. This can lead to memory inefficiency and degrade performance.

Example of Inefficient Nested Component Declaration -



Here, we’ve created a ChildComponent inside the ParentComponent, which means that each time ParentComponent renders or re-renders, ChildComponent is recreated—a significant performance drawback. In this example, every time the count value is updated by clicking the increment button, ParentComponent re-renders. Notice the reconciliation in action in the figure below, where the highlighted flash in the <p> tag of ChildComponent indicates it is also re-rendering. This shows that ChildComponent is recreated with each click on the increment button.

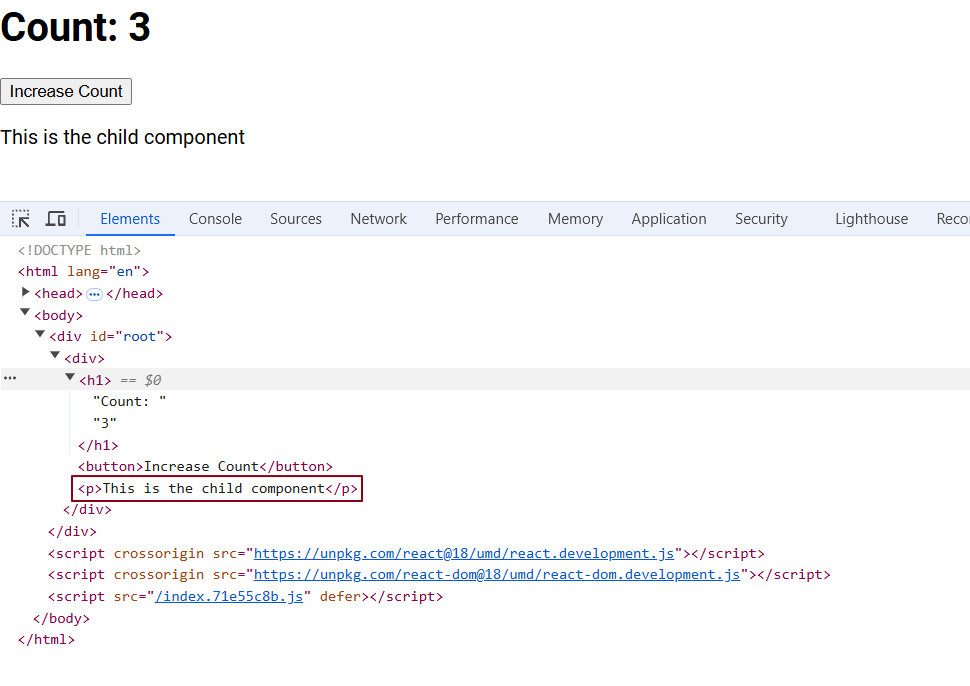


How can we fix this?

By defining ChildComponent outside of ParentComponent.



Here, we’ve created ChildComponent outside of ParentComponent, so ChildComponent is only created once. During each re-render of ParentComponent, ChildComponent is not recreated, as reflected by its absence from the virtual DOM changes below.



**2. Never create useState variables inside an if-else block.**

Because if you define useState variables inside an if-else block and the block does not execute, react will not be able to determine whether the state variable exists or not. This inconsistency makes it difficult for React to trace the variable, leading to potential issues. React requires a predictable structure to understand the behavior of your component.

**3. Never create useState variables inside a for loop.**

Because doing so will create a new state variable for each iteration of the loop. If only one state variable is needed, multiple variables will be unnecessarily created, leading to confusion and potential errors.

**4. Never declare useState variables outside of a function component**

State variables defined outside a component are not linked to the component’s lifecycle, meaning they won’t update or respond as intended. To work correctly, useState variables must be declared within the function component itself.

Routing

Routing is the process of defining how the web application handles different URL paths and what content to show for each path.

Routing in web applications is like a map that tells the application where to go and what to show when you enter a specific URL.

What is Client-side routing?

**Definition:** In client-side routing, the browser handles all routing logic. When you navigate to a different route, the application updates the URL and changes the content without requiring a full page reload. This is typically implemented using JavaScript frameworks like React, Angular, or Vue.js.

**Benefits:**

**Speed:** Faster transitions because only parts of the page are updated.

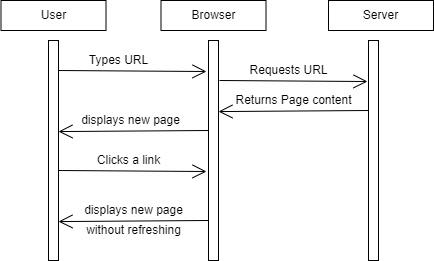
**Single-Page Applications (SPAs):** Suitable for SPAs where the application loads once and dynamically changes content.

**Dynamic Routing:** Allows for easy manipulation of routes in response to user actions or events.

**Drawbacks:**

**SEO Limitations:** Search engines may have difficulty indexing pages that rely heavily on JavaScript for rendering content.

**Initial Load Time:** The initial loading can be slower because the entire framework and application logic need to be loaded at once.



What is Server-side routing?

**Definition:** In server-side routing, the server is responsible for handling all routing. Each time a user requests a new route, the server generates and sends a new HTML page corresponding to that route.

**Benefits:**

**SEO-Friendly:** Each route corresponds to a distinct HTML page, making it easier for search engines to crawl and index content.

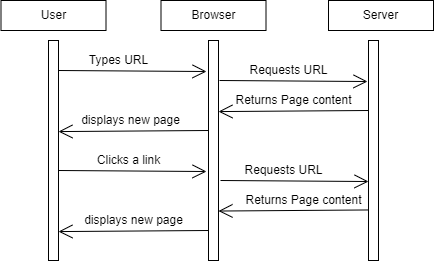
**Simplicity:** Easier to implement for traditional websites with static pages or simple routing needs.

**Quick Initial Load:** The initial page load can be faster since the server sends fully-rendered HTML pages.

**Drawbacks:**

**Slower Transitions:** Page transitions can be slower because each navigation requires a full page reload.

**Less Interactivity:** The user experience may feel less fluid compared to client-side routing, as each interaction can lead to a more jarring transition.



When to Use Client-Side Routing?

Best for SPAs and dynamic applications, offers faster transitions but may struggle with SEO.

When to Use Server-Side Routing?

Better for SEO and traditional websites, but can be slower and less interactive.

What is a SPA?

A type of web application that loads a single HTML page and updates content dynamically without refreshing the entire page. SPAs improve user experience by allowing fast, dynamic updates without reloading the entire page, making web applications feel more like desktop applications.

**Key Characteristics:**

1. **Dynamic Updates:** Content changes occur without full page reloads, thanks to JavaScript and client-side routing.
2. **Smooth User Experience:** Users experience faster and more responsive interactions since only parts of the page update.
3. **Faster Initial Load:** The first time you load a SPA, it may take longer because it downloads all necessary JavaScript and assets. However, once loaded, navigating is quicker because only data is exchanged with the server.
4. **Client-Side Routing:** SPAs use libraries (like React Router or Vue Router) to handle navigation, simulating traditional page changes without actually loading new HTML pages.
5. **API-Centric:** SPAs often rely on backend APIs to fetch and send data, usually in JSON format. This separates the front end from the back end.
6. **State Management:** To keep track of data and application state, SPAs often use libraries (like Redux for React or Vuex for Vue).

How can I achieve routing in react?

The react-router-dom library is used to implement routing in React applications. You can install it using the command: npm install react-router-dom

Features that react-router-dom provides:

**1. createBrowserRouter Function:** This function allows you to create a router for your application. It takes an array of route configuration objects, where each object contains specific settings.

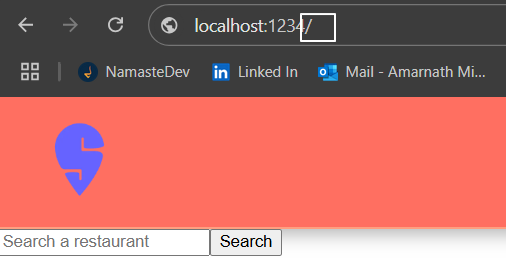
**2. Route Configuration:** Each configuration object should include a path and an element. The path defines the URL for the route, and the element specifies the component to be rendered when the user navigates to that path. The createBrowserRouter function sets up the router based on these configurations.

**3. RouterProvider:** Simply creating a router isn't enough; we must provide it to your application. This is done using the RouterProvider component, which we import from react-router-dom.

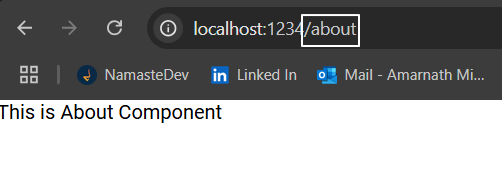
**4. Rendering Routes:** Instead of rendering a component directly in our render function, we will now render the RouterProvider component, passing the route configuration array as a prop. This setup enables our application to render the appropriate component based on the current route.



Now if we navigate to the base URL [**http://localhost:1234/**](http://localhost:1234/) we get AppLayout component page

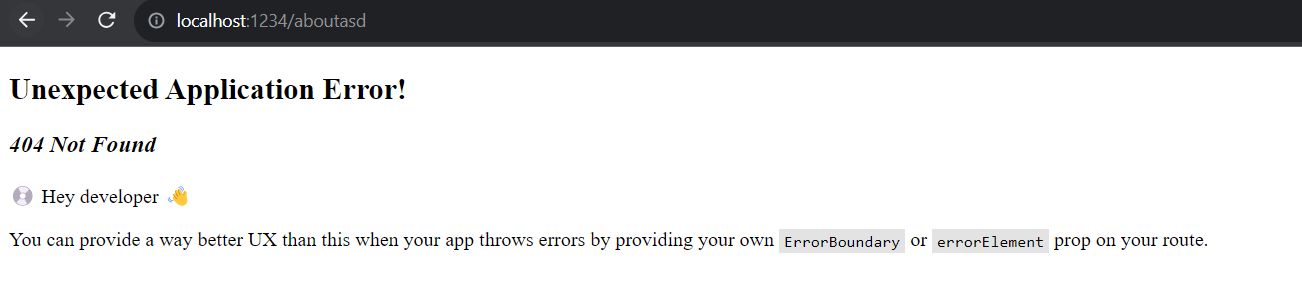


If we navigate to about page URL **http://localhost:1234/about** we get about component page.

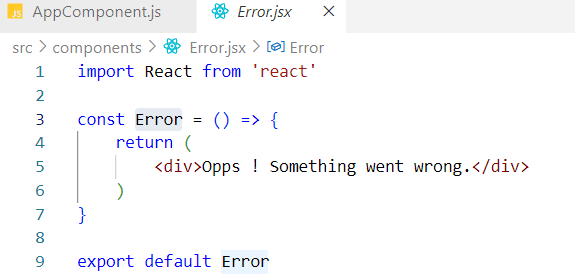


What happens when user navigates to a URL that does not exist?

react-router-dom is a powerful library that also manages errors when navigating to a route that does not exist. In such cases, it provides a default 404 error page to inform users that the requested route is unavailable.

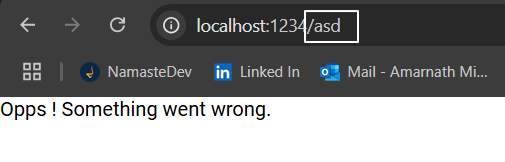


However, we can display a custom error page by using the errorElement attribute in the route configuration object. This attribute allows us to specify an error component that will be rendered when a user navigates to a URL that does not match any defined routes.

Let's create an Error component and assign it to the errorElement attribute in the route configuration object.

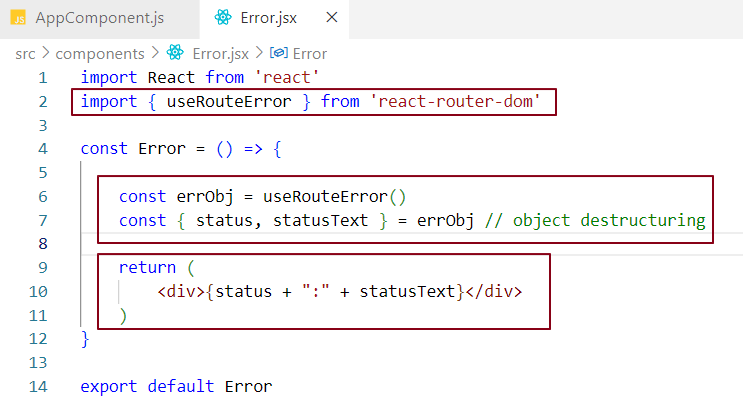


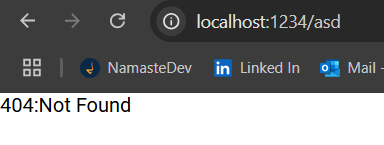
When an invalid URL is accessed, the specified error component will be displayed on the UI.



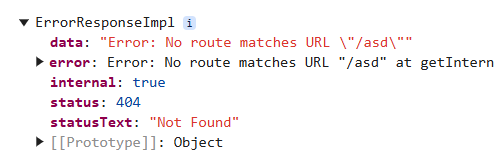
How can we display additional information about the error when a user navigates to an invalid URL?

react-router-dom provides a hook called useRouteError, which returns an error object that indicates the type of error encountered.





If we log the **errObj** to the console



So far, we have been manually typing the path in the URL to navigate to the desired component page. However, in a real-world application, clicking on links or hyperlinks should direct users to the corresponding 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 enable users to move between different routes or pages within a single-page application (SPA). It serves as a replacement for traditional anchor (<a>) tags and offers additional functionality specifically designed for routing.

Why do we use Link component?

Using anchor tags for navigation typically leads to server-side routing. When you use an anchor (<a>) tag with an href attribute, clicking the link triggers a request to the server for the specified URL. This results in a full page reload, which can lead to a suboptimal user experience, as users must wait for the new page to load and render.

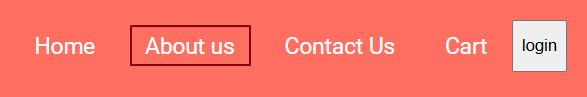
To achieve client-side routing and provide a smoother user experience, we use the Link component from react-router-dom in React applications. By enclosing navigation elements within a Link tag, we can navigate between different routes without reloading the entire page. This allows for faster transitions and a more responsive interface.

Why do we configure Link component for routing?

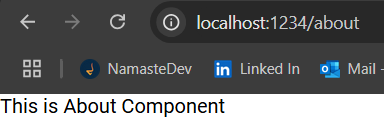
In the Link component, we define a prop called **to**, which specifies the path for the link. This **to** prop determines where the application will navigate when the link is clicked.



Now, when we click on the "About Us" link, the /about path specified in the **to** prop will be matched with the route configuration object defined in the appRouter. Once a match is found, the corresponding component is rendered, which is exactly what's happening in our case.

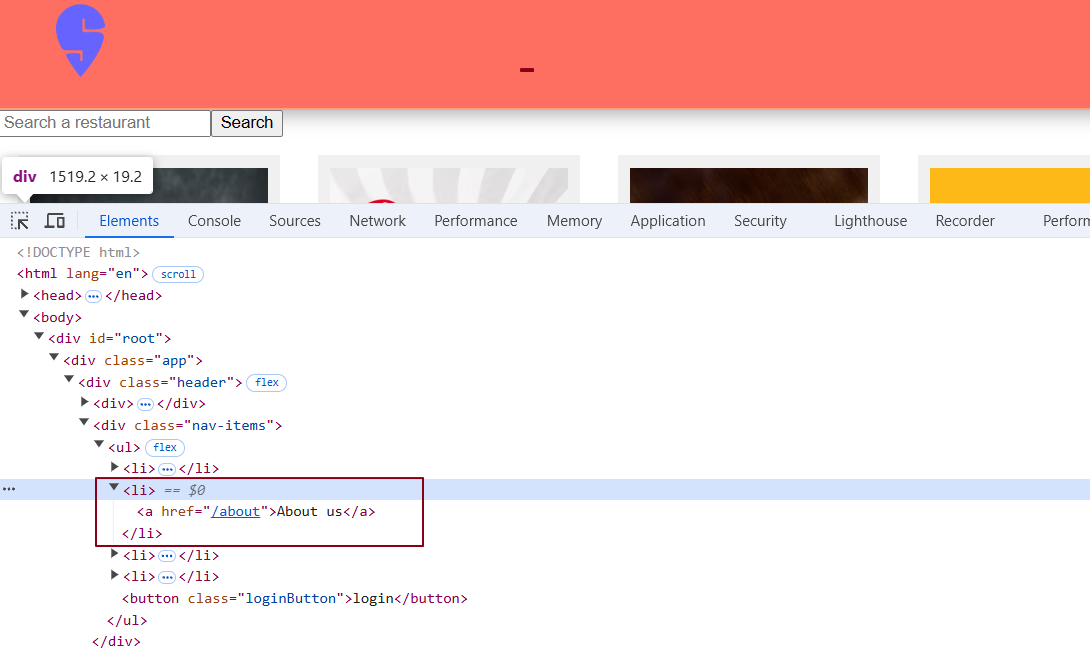


When the "About Us" link is clicked, the About component is rendered in the UI.



When we inspect the page, we notice that the Link component is rendered as an anchor (<a>) tag rather than a Link tag. This occurs because the Link component is converted into an anchor tag by the react-router-dom library, allowing the browser to recognize it.

The Link component essentially acts as a wrapper around the anchor tag. When we create a link to the home route, React Router keeps track of it, enabling us to navigate without refreshing the page. The browser does not understand the Link component directly; instead, it understands standard anchor tags. React Router handles the conversion of the Link into an anchor tag and manages the navigation functionality seamlessly.

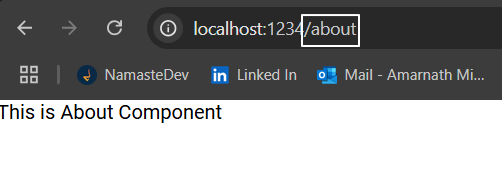


**What is Nested Routing?**

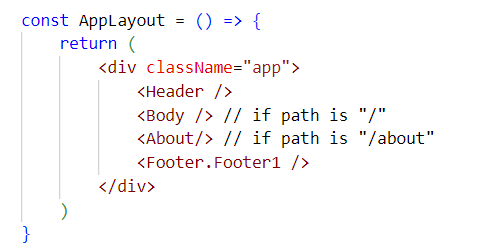
Nested routing allows us to place multiple child components within a parent component, enabling routing for child components without reloading the entire page. This means that when a user navigates to a different section of our application, only the content within the specified child component updates, while the parent component (like the header and footer) remains unchanged.

**Why Use Nested Routing?**

Using nested routing is beneficial because it provides a consistent user experience. For instance, if we navigate to the About page, we want the header and footer to be visible alongside the main content. However, with the current setup, if we go to the About component, the header and footer do not load, which can disrupt the navigation experience.



In the screenshot below, when the path is "/", we want the "Body" component to be displayed within the header and footer components. Similarly, when the path is "/about", we want the "About" component to be rendered inside the header and footer components as well.

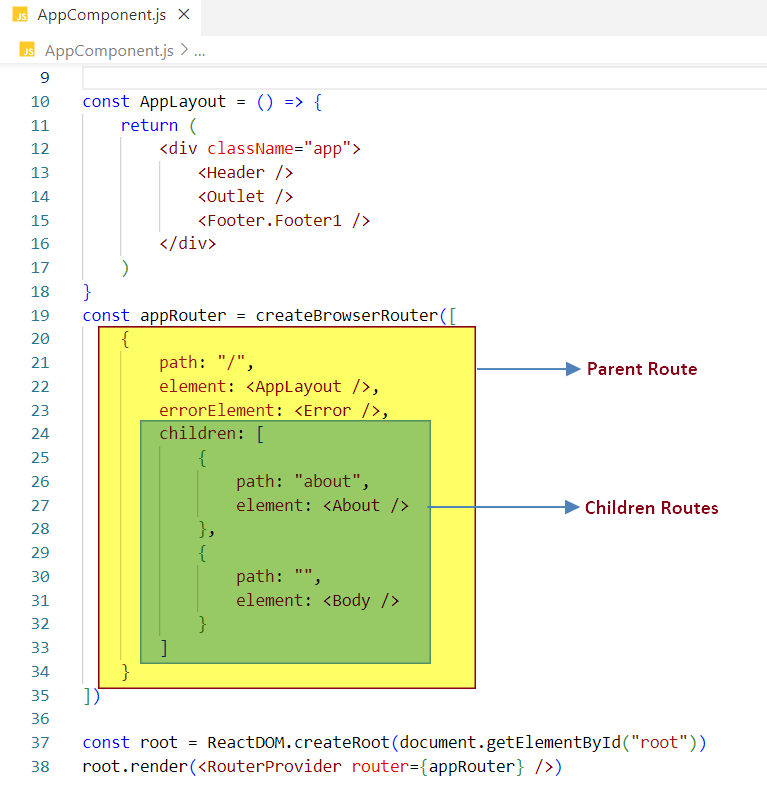


We want our header and footer components to remain fixed while displaying different components in the content area during page navigation.

**How Do We Achieve Nested Routing in React?**

To implement nested routing in React using the react-router-dom library, we define our routes in a way that includes a parent route with child routes. This allows us to keep shared components (like the header and footer) rendered while swapping out the main content based on the active route.

To implement nested routing in our app setup, we need to make the About and Body components children of the AppLayout component. Additionally, we will include an <Outlet /> in the layout, which will serve as the area where the child components are displayed during navigation.



### Route Structure

**Parent Route**:

* The parent route is defined by the path / in the routing configuration.
* In this case, the parent route is associated with the AppLayout component.
* This parent route(http://localhost:1234**/**) renders shared components: Header, Footer, and an <Outlet /> for displaying child components.

**Child Routes**: There are two child routes defined under the parent route:

* + - **Child Route 1**:
      * Path: /about
      * Component: About
      * This route is used when the user navigates to http://localhost:1234**/about**.
    - **Child Route 2**:
      * Path: <no path>
      * Component: Body
      * This route is used for the default path (http://localhost:1234**/**) and is rendered inside the <Outlet /> when no other specific route matches.

Routing Flow

1. **Initial Load** (http://localhost:1234):
   * The URL matches the parent route /.
   * The AppLayout component is rendered, which includes the Header, Footer, and an <Outlet />.
   * The <Outlet /> will render the Body component because the child route with path / is matched.
2. **Navigating to About** (http://localhost:1234/about):

The URL now matches the child route /about.

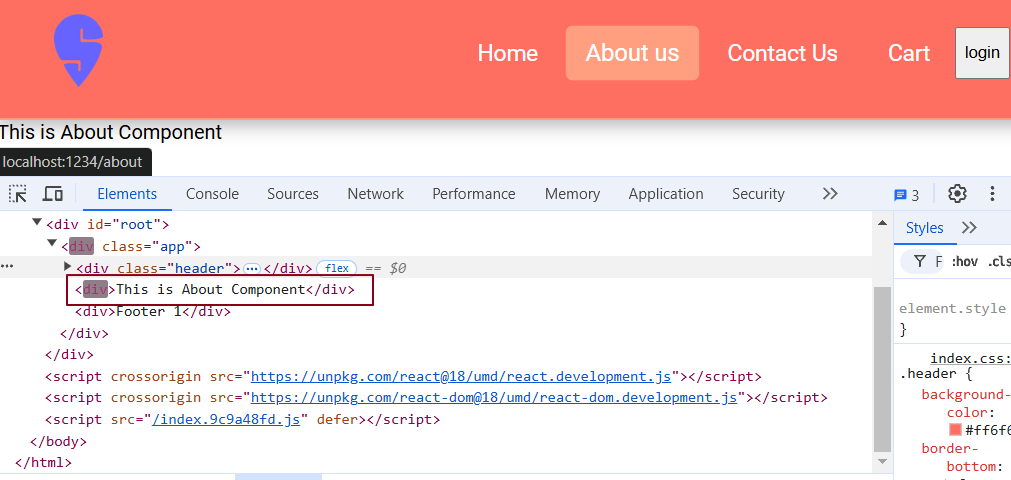
* + The AppLayout remains rendered, but the <Outlet /> now displays the About component instead of the Body component.

What is Outlet?

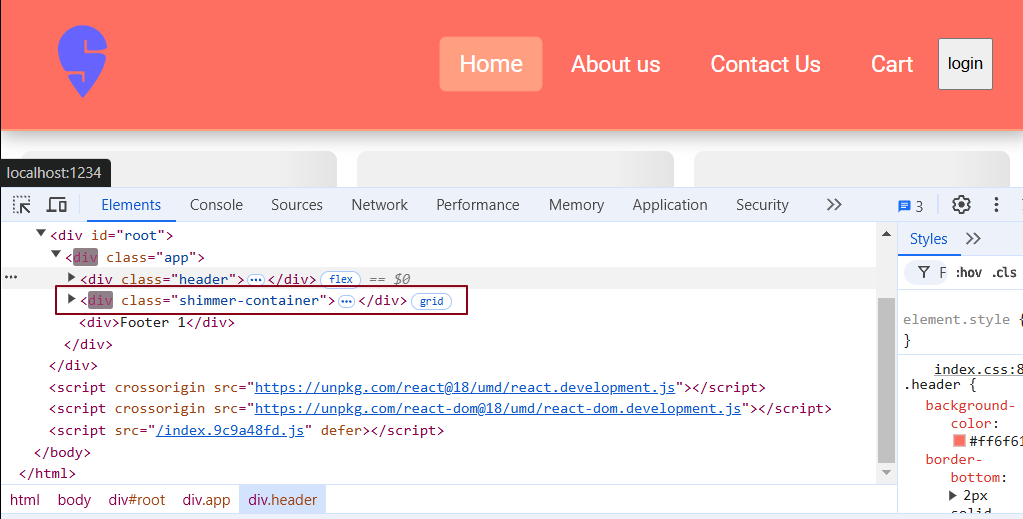
**Outlet** is a component provided by react-router-dom that serves as a placeholder for rendering child components within a layout component during page navigation. It allows you to insert child components defined in the route configuration objects into the layout.

When you navigate from one page to another, react triggers reconciliation, which updates only the part of the DOM that has changed. In this case, it updates the content inside the **Outlet** component, while the rest of the layout (like the header and footer) remains intact.

The moment when we click on About us link the outlet renders About component within the AppLayout component.



When we click on "Home," the outlet renders the Body component within 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 fixed, and we navigate to these specific links directly. However, in a real-world scenario, we might be fetching a list of restaurant cards from an API and displaying them in the UI.

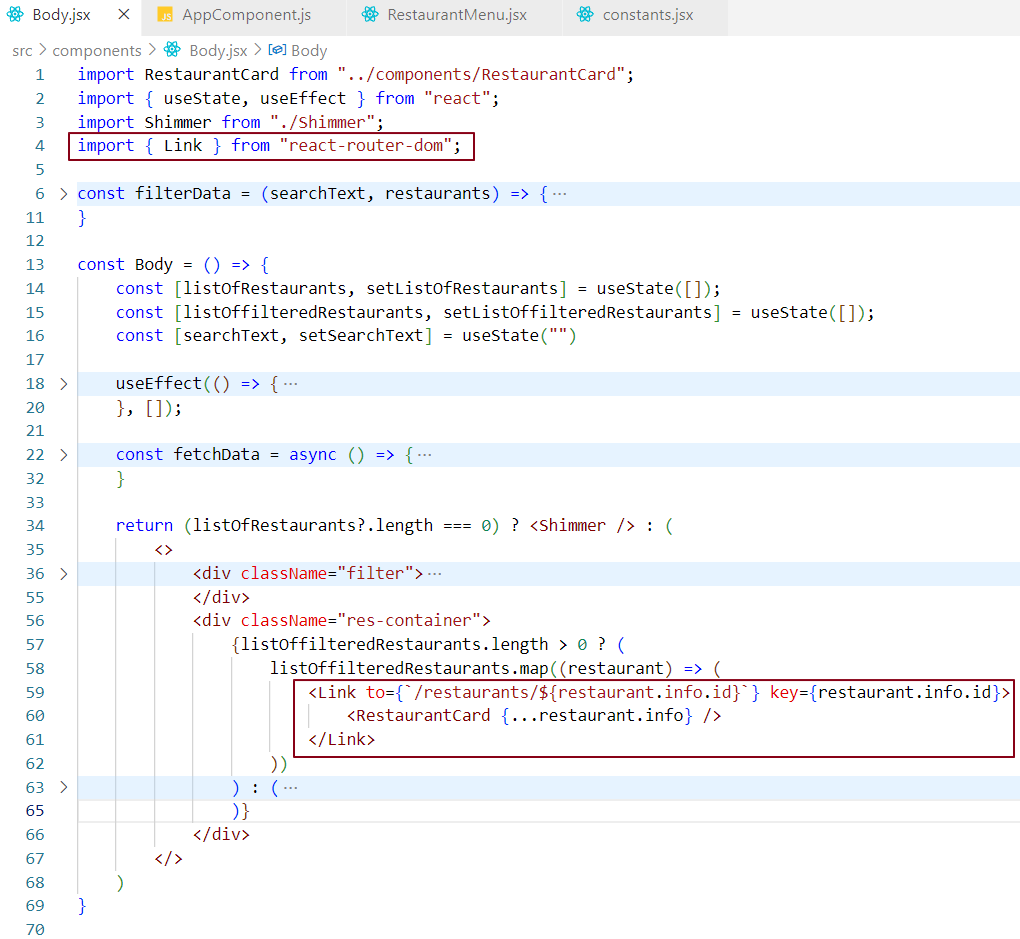
When we click on any one of these restaurant cards, we should be navigated to a specific component that shows details for that restaurant. Since we don’t know how many restaurant cards will be displayed, dynamic routing becomes essential. Dynamic routing allows us to create routes that can change based on user interactions, such as clicking on a restaurant card, thus enabling us to navigate to the respective card's specific component.

What is Dynamic routing?

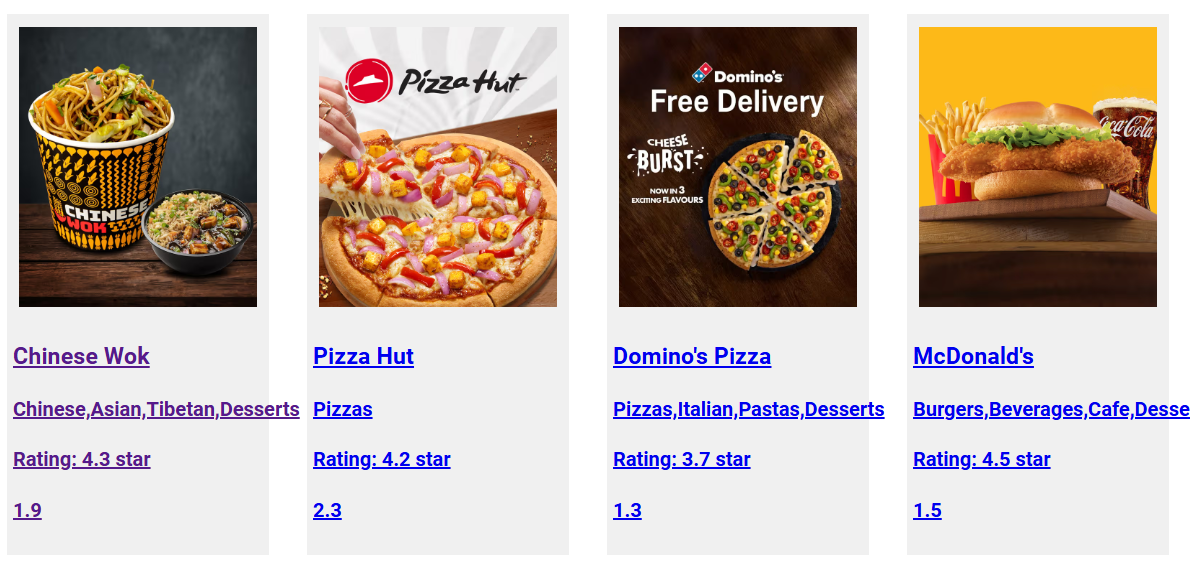
Dynamic routing is the process of setting up routes while the application is running, instead of fixing them before it starts. This means that routes can change based on user actions or data. With dynamic routing, we can create flexible navigation options that adapt to what the user does or the information that comes from outside sources, like an API.

Let's integrate dynamic routing into our code.

So far, we’ve been displaying a list of restaurants in the Body component, but these restaurant cards are not clickable. Let’s wrap them in a Link component to make them clickable.

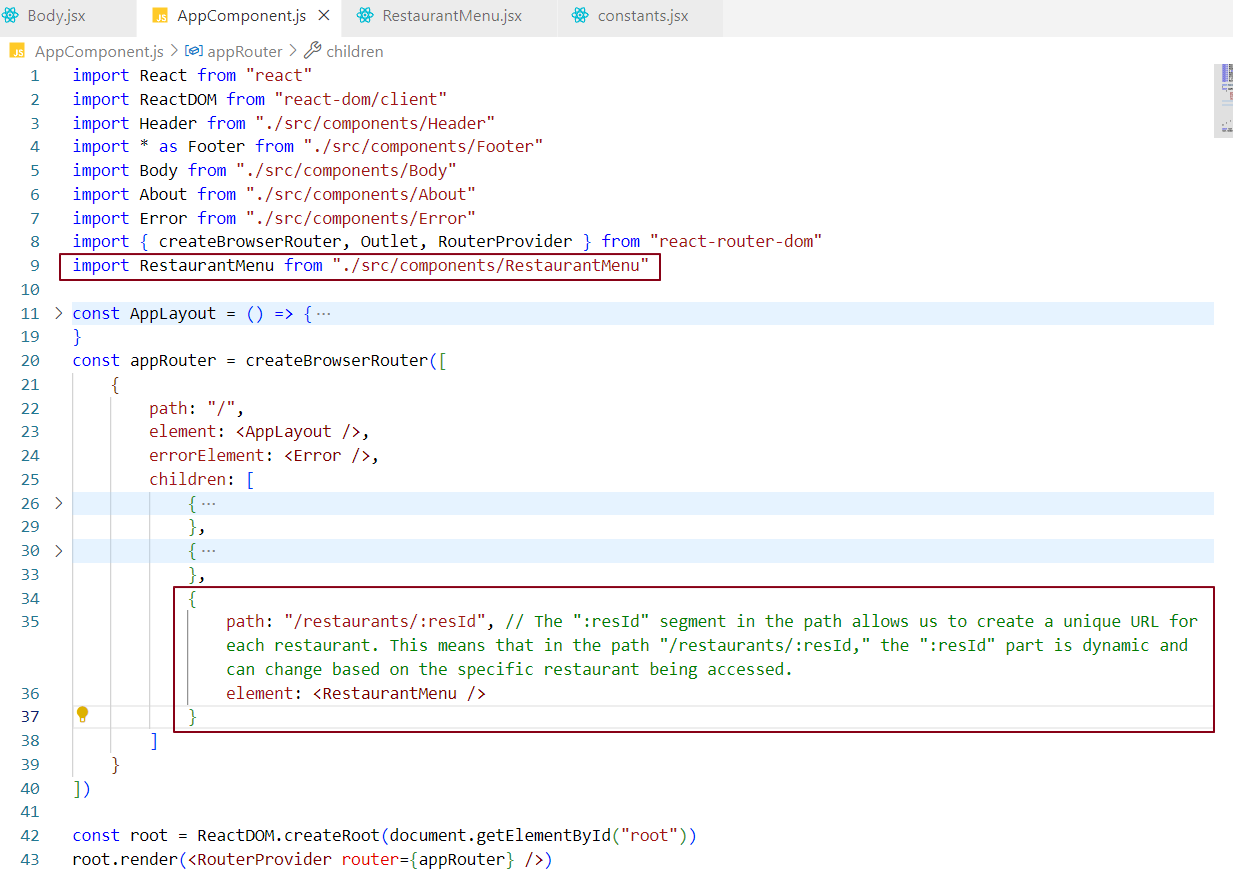


Note- We’ve moved the key from the restaurant card component to the Link component. The key should be assigned to the parent Link instead of the individual restaurant card.



As we can see, these restaurant cards are now clickable.

If we observe the code, we are passing the path /restaurants/${restaurant.info.id} to the Link component for each restaurant card. This path corresponds to the route configuration object defined in the appRouter.

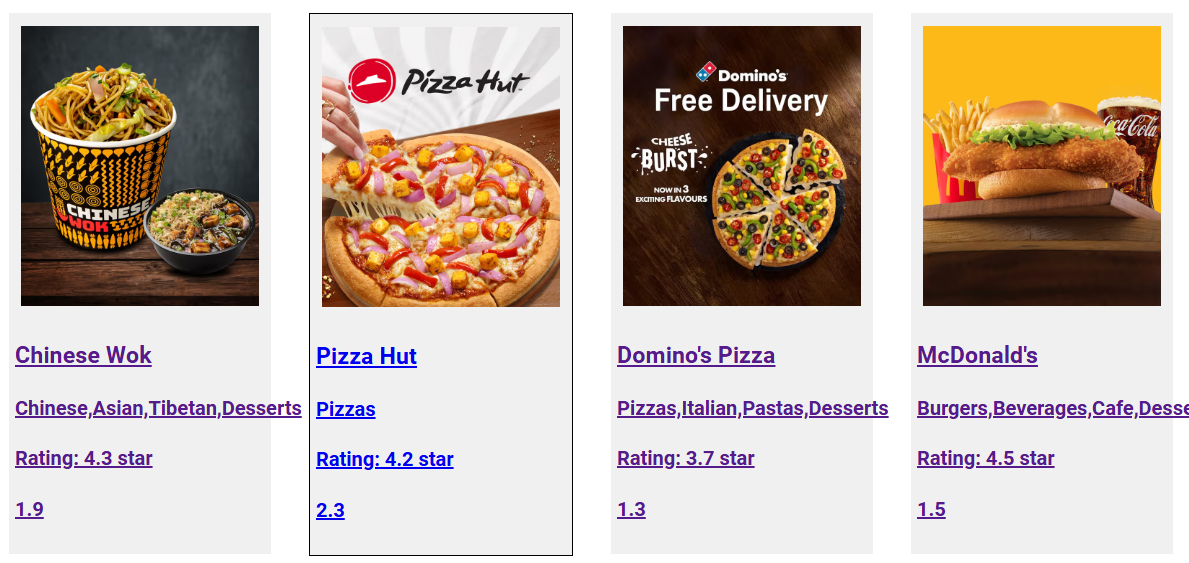


This route configuration is dynamic because we don’t know the value of resId at compile time. The resId is determined at runtime when the restaurant card is clicked. When a card is clicked, resId will hold the ID of that card, which can be accessed using the useParams hook provided by React Router DOM, as discussed later.

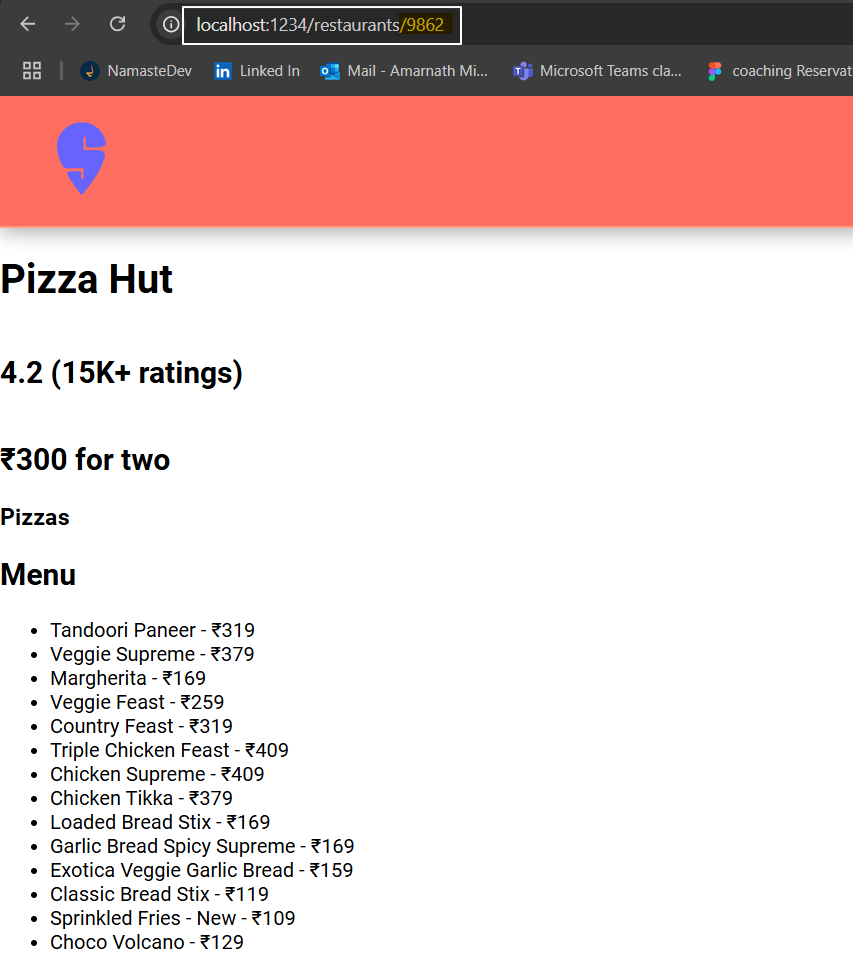
Now, let's create the Restaurant Menu component that will display the menu specific to a restaurant when the restaurant card is clicked.



In this component, we are fetching the resId to use in the API call, which retrieves the restaurant menu data. A shimmer effect is shown while the data is being fetched. Once the data is available, we display the restaurant menu along with the restaurant details.



When I click on the card, such as the Pizza Hut card, the card ID (e.g., 9862) is added to the URL. This ID is then read by the Restaurant Menu component, which uses it to make API calls and display the relevant data in the UI.

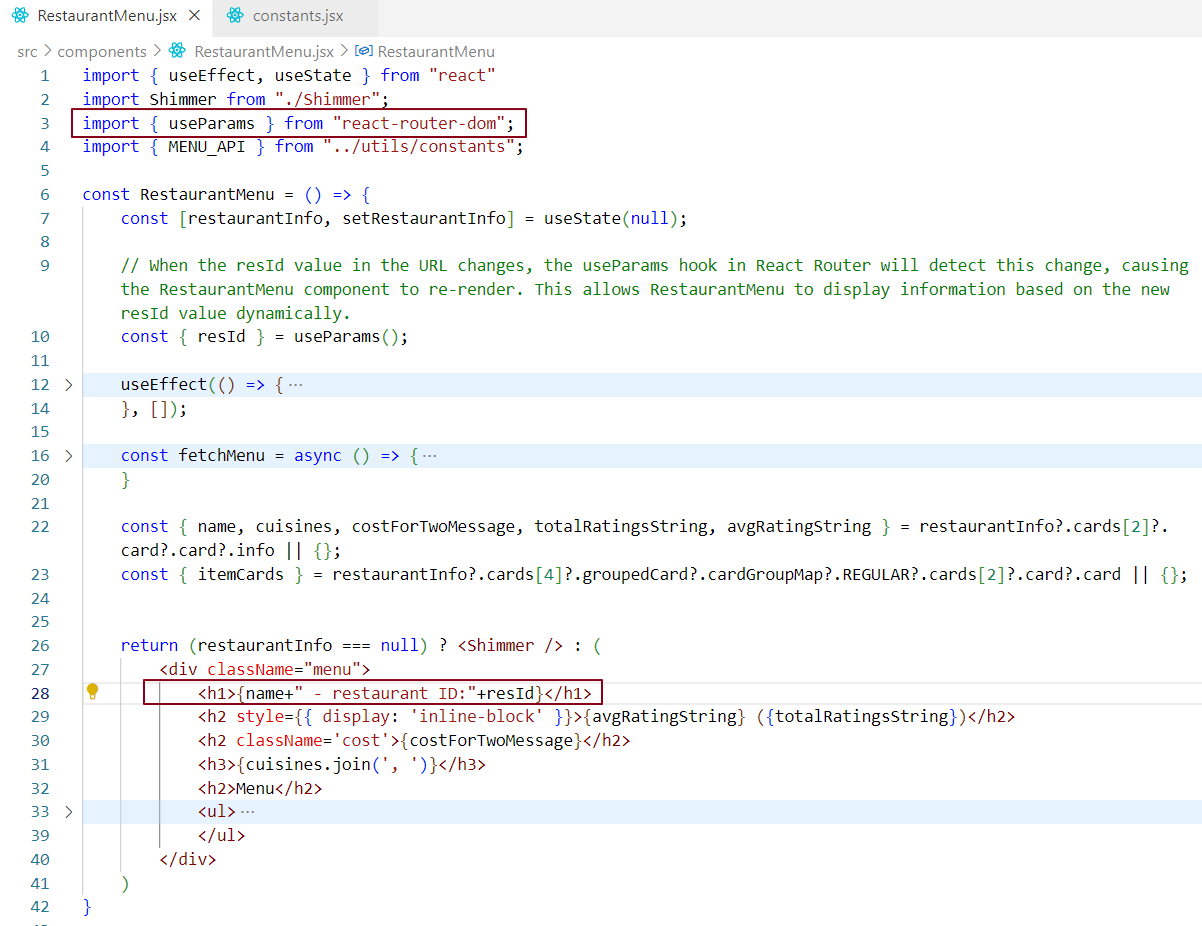


How is the ID being read from the Restaurant Menu component?

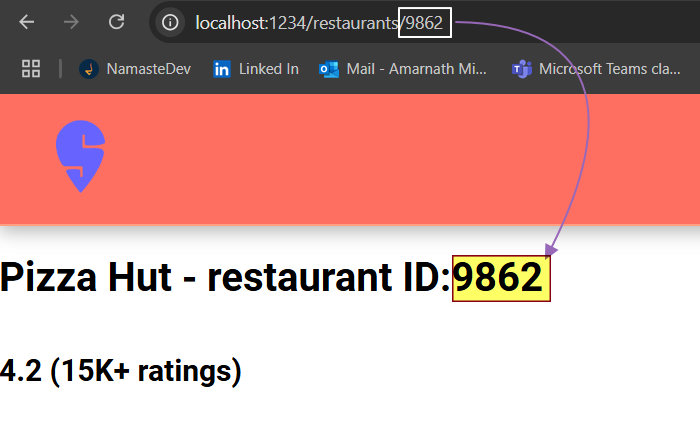
The Restaurant Menu component utilizes the useParams hook provided by React Router DOM to retrieve the ID.

What is useParam hook in react?

The useParams hook is provided by the react-router-dom library. It allows us to get parameters directly from the URL. This means we can access and use data from the URL within the component we are currently viewing.



When the RestaurantMenu component loads, it reads the resId value from the URL using useParams. This trigger fetching the menu for the restaurant associated with that resId. Once the data is fetched, the component re-renders and displays the restaurant’s information, including the resId, in the UI.



Why is a CDN a great option for hosting images?

* CDNs are fast and deliver images quickly.
* CDNs offer 100% uptime.
* CDNs optimize images before sending them to the browser and cache them for quicker access.

Large applications like Swiggy and Zomato utilize CDNs to host images due to these advantages.

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

If we run console.log (useState ()), we will see an array in the console, typically in the format [undefined, function]. The first item in this array represents the current state, which is undefined initially, and the second item is the setState function used to update the state.