Chapter 6 – Exploring the World

Before we explore external frameworks or advanced topics, it’s important to understand the fundamental design patterns commonly used in the software industry. These patterns serve as best practices for structuring code, solving recurring problems, and improving the maintainability and scalability of applications.

Monolithic Architecture:

Monolithic architecture refers to an approach where an application is developed as a single, unified unit, with all components - such as the user interface, business logic, data access, authentication, and APIs - integrated together in one codebase. In this architecture, all parts of the application are tightly coupled and interdependent.

While it is easier to develop and deploy for smaller projects, as everything exists in one place, it can become challenging to scale or modify as the system grows. A change in one component often requires modifications to others, making it harder to maintain and test. Additionally, since all components are packaged together, updating or scaling one feature often involves scaling the entire application, which can lead to inefficiencies.

**Pros**

1. Simple to develop - Ideal for small teams and projects, as everything can be built in a single codebase.
2. Easier to test - Because the application is unified, it is easier to run tests across the entire system.
3. Easier to deploy - You deploy the entire application as one package, simplifying the deployment process for small or less‑complex systems.
4. Better performance at small scale - For small applications, monolithic systems can sometimes perform better due to fewer service‑to‑service communication delays.

**Cons**

1. Difficult to Scale

* Explanation: In monolithic architecture, the entire application runs as a single, unified system. This means that if one part of the system (like the database or a certain feature) starts to experience heavy loads, you can't just scale that one part by adding resources to it. You have to scale the entire system together.
* Impact: For example, if just one feature of your application (like user login) is getting a lot of traffic, you can't allocate extra resources to just that feature. Instead, you would need to add more servers or resources for the whole application, even though the other parts of the app might not need them. This leads to inefficient use of resources and higher costs.

2. Tightly Coupled

* Explanation: Monolithic applications are tightly coupled, meaning that all the components of the system are dependent on each other. If you make a change to one part of the system, it can affect other parts of the application because they’re all interconnected.
* Impact: If you want to update or fix one module (say the payment processing system), you might have to touch other parts of the codebase that are related, even if you don’t intend to change them. This increases the risk of bugs and requires more careful testing. Making updates becomes more complicated and riskier because a small change could potentially break other functionalities of the application.

3. Slower Development for Large Apps

* Explanation: As monolithic applications grow larger and more complex; the development process slows down. This happens because many developers are working on the same codebase, which increases the chances of conflicts and requires greater coordination.
* Impact: Imagine multiple teams working on different features, all in the same codebase. If one team makes changes, it can impact another team’s work. Developers might have to wait for each other to finish before making their changes, which creates bottlenecks. Also, as the codebase grows, it becomes harder to understand and navigate, slowing down the development process. The application becomes harder to maintain over time as the complexity grows.

4. Limited Flexibility

* Explanation: In a monolithic architecture, all components are bundled together in one system, which limits the flexibility to adopt new technologies for specific components. If you want to use a new technology or framework for just one part of the application, you can't do it easily without affecting the whole system.
* Impact: Let’s say you want to implement a new, faster database for your application’s analytics module. In a monolithic system, since everything is tightly integrated, you can't just change the database for one part of the system without considering how it will affect the rest of the application. This makes it hard to introduce new technologies, slow down innovation, and can cause the app to become outdated over time, as it’s harder to evolve the technology stack.

Microservices Architecture:

Microservices architecture is an approach where an application is built as a collection of small, loosely-coupled, independent services. Each service is responsible for a specific business function and can be developed, deployed, and scaled independently. These services communicate with each other through well-defined APIs, ensuring clear boundaries between them. By adhering to the Single Responsibility Principle, each service focuses on a distinct task, promoting separation of concerns.

For example, services like the UI service, backend API service, authentication service, and notification service can all be deployed and scaled independently. While they interact with one another, each service maintains its own responsibility and autonomy, allowing for greater flexibility and ease of maintenance.

**Pros:**

1. Independent scaling - You can scale specific services rather than the entire application, making it more resource-efficient.

2. Fault isolation – If one service fails, it doesn’t necessarily bring down the entire system. The other services can continue to run.

3. Technology flexibility - Each service can be built using the best tools or languages suited for its job, allowing greater flexibility.

4. Faster development - Different teams can work on different services in parallel without interfering with each other.

5. Easier to maintain - Since services are smaller and focused on specific tasks, they’re easier to understand and manage.

**Cons:**

1. Increased complexity – Managing multiple services introduces complexities in terms of service coordination, deployment, and communication.

2. Difficult to test – Testing a distributed system can be more challenging because you need to ensure that all services work correctly together.

3. Network latency – Service-to-service communication adds some overhead, which can slow down the overall system.

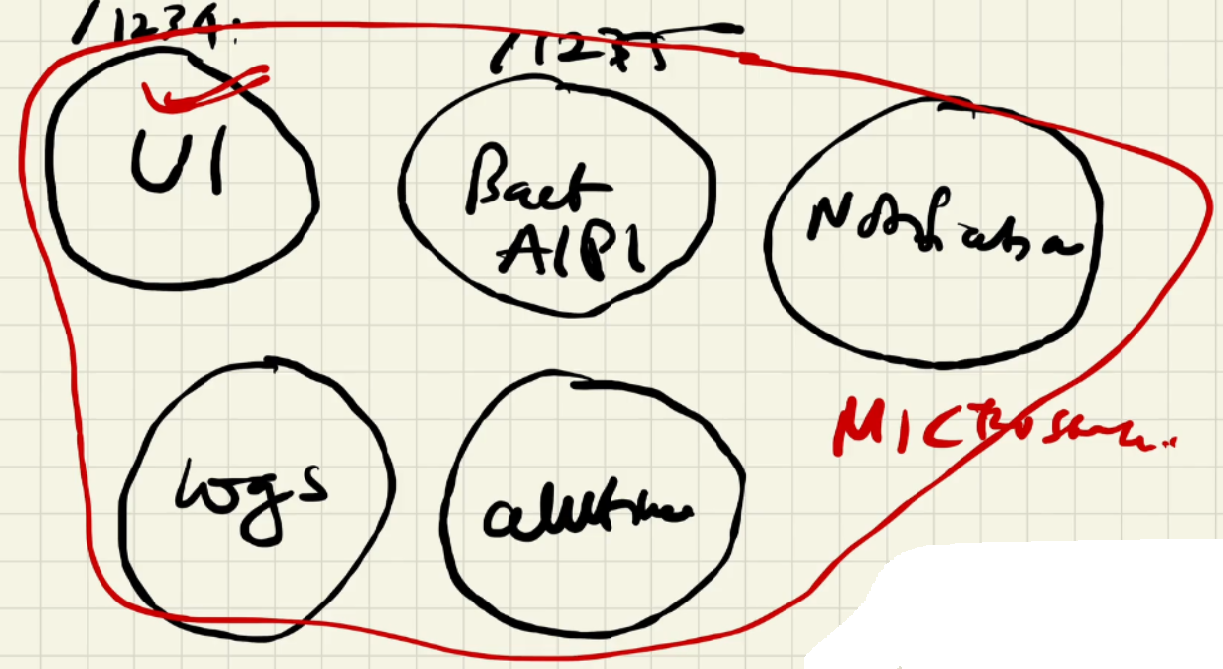
4. Distributed data management – Each service often has its own database, making data management more complex.

5. Higher operational costs – You need robust infrastructure and tools to manage, monitor, and secure multiple services running independently.

How do these services interact with each other?

* In our setup, the UI microservice is built using React, which handles the user interface and interacts with other services for data and functionality.
* These services communicate with each other using various protocols like HTTP/HTTPS or gRPC. For example, the UI microservice might need to request data from the backend microservice. The backend microservice processes this request and may need to access the database to retrieve the necessary data.
* Each microservice runs on its own specific port, allowing for independent deployment and scaling. These individual services are mapped to a single domain name, which aggregates all the services and provides a unified access point for the entire application. This enables users or external services to interact with the application as a whole, without having to manage or know the details of each individual service's port or location.
* By keeping the services independent but connected under a single access point, we gain flexibility, scalability, and ease of maintenance.

In summary, monolithic architecture is simple and easier for smaller applications but becomes harder to maintain as the application grows. Microservices architecture, while more scalable and flexible, is complex to manage, making it suitable for larger, evolving applications.



It's time to step into the real world! We're going to **fetch real-time data from an external API** that our application doesn't yet know. This process typically involves making a request to the API, retrieving data in a format like JSON, and then using that data in your application.

**Note:** React is known for its speed because it has highly efficient rendering cycles.

How to Make an API Call ?

**In JavaScript**: API calls are made using the fetch() method to retrieve data from a server.

**In React**: API calls are typically made within the useEffect() hook to handle side effects like fetching data when the component mounts.

Methods for Making API Calls in React

There are two common approaches to making API calls in React:

Blocking API Call (Approach 1)

Flow: Load the website ➔ Wait for the API call to complete (e.g., fetch()) ➔ Once the data is available, render the page.

The page becomes available after a combined time of 500 ms (300 ms for the API call + 200 ms for rendering).

Explanation: In this approach, the main screen is **blocked** until the API call is finished. The application waits for the data to be fetched before rendering any content. Once the data is available, the page is rendered with the data.

Non-Blocking API Call (Approach 2)

Flow: Load the website ➔ Render the initial page (100 ms) ➔ Make the API call (300 ms) ➔ Update the UI with data once the call completes.

The initial page becomes visible in 100 ms, allowing the user to see content faster. Once the API call finishes, the UI is updated with the fetched data.

Explanation: In this approach, the page is rendered **immediately** with initial content, and the API call is made asynchronously in the background. The UI is updated dynamically when the data becomes available, allowing the user to see something while waiting for the full content to load.

**Note -** **Non-Blocking API Call (Approach 2)** is better as it allows the page to load quickly with initial content, improving user experience by dynamically updating the UI once the data is fetched.

Blocking vs. Non-Blocking API Calls:

Blocking API Call: The main screen **remains blocked** and cannot be interacted with until the API call is completed. The application waits for the data to be fetched before rendering the content. This ensures that the page displays complete information, but it can delay the user’s experience.

Non-Blocking API Call: The application renders the initial page quickly while making the API call in the background. The user sees the content almost immediately, and the data is dynamically injected into the page once it's available. This reduces the perceived load time and improves user experience.

### Best Place to Call an API in React

We can't call an API just anywhere in a React component. If we place an API call directly in the component, it will be triggered every time the component re-renders due to state or prop changes, leading to multiple API calls and reduced performance.  
To ensure the API is called only once on the initial load, we use the useEffect() hook, one of React's most essential hooks. By configuring useEffect() to run only on the initial render, we avoid unnecessary API calls and optimize performance.

What is the useEffect Hook?

useEffect is a special function (hook) that lets us perform actions like fetching data, updating the page title, or setting up timers when a component load. These actions are called side effects because they happen outside the main rendering process.

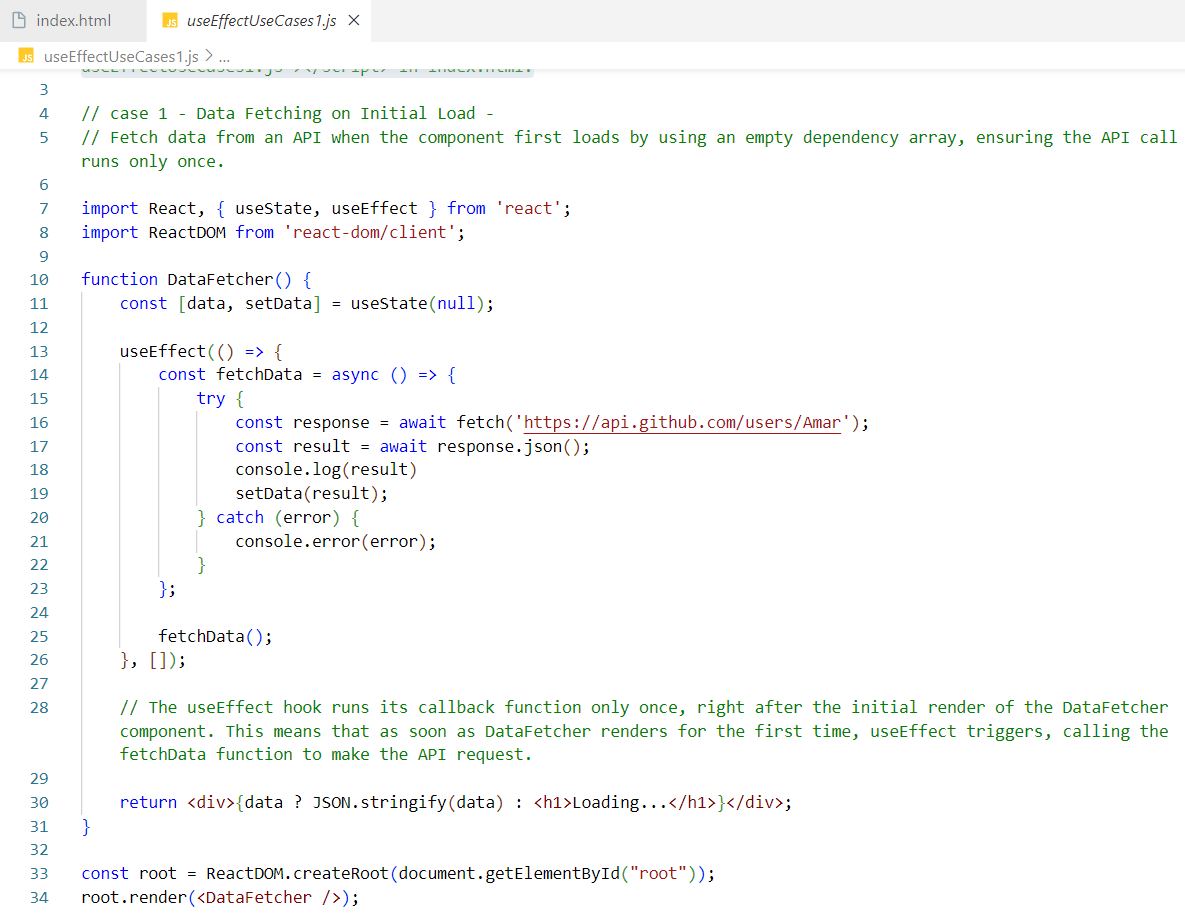
* The useEffect hook takes two parameters: a callback function and an optional dependency array.
* The callback function is not called immediately; instead, it runs when useEffect decides it should.
* By default, useEffect calls its callback every time the component renders or re-renders.
* We can control this behavior by providing a dependency array. If the dependency array is empty, useEffect calls its callback only once—after the component’s initial render. When the component re-renders, useEffect will not call its callback again.
* If we add a state variable to the dependency array, useEffect will call its callback after the component’s initial render and again each time the state variable changes.
* To make an API call, we place the call inside the useEffect callback function and provide an empty dependency array, so the API is called just once.
* The dependency array is optional. If we omit it, useEffect will run its callback after every render and re-render of the component, as it doesn't depend on any specific variables.

**When does our component render and re-render?**

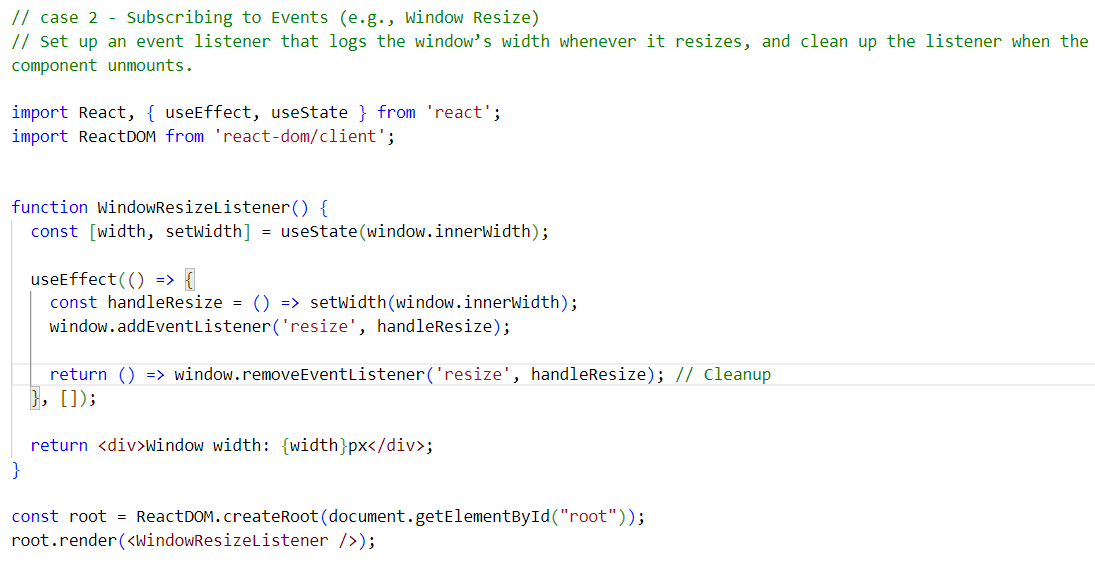
When our page loads for the first time, the component renders. The component re-renders whenever there is a change in its state or props.

What are the use cases of the useEffect Hook?

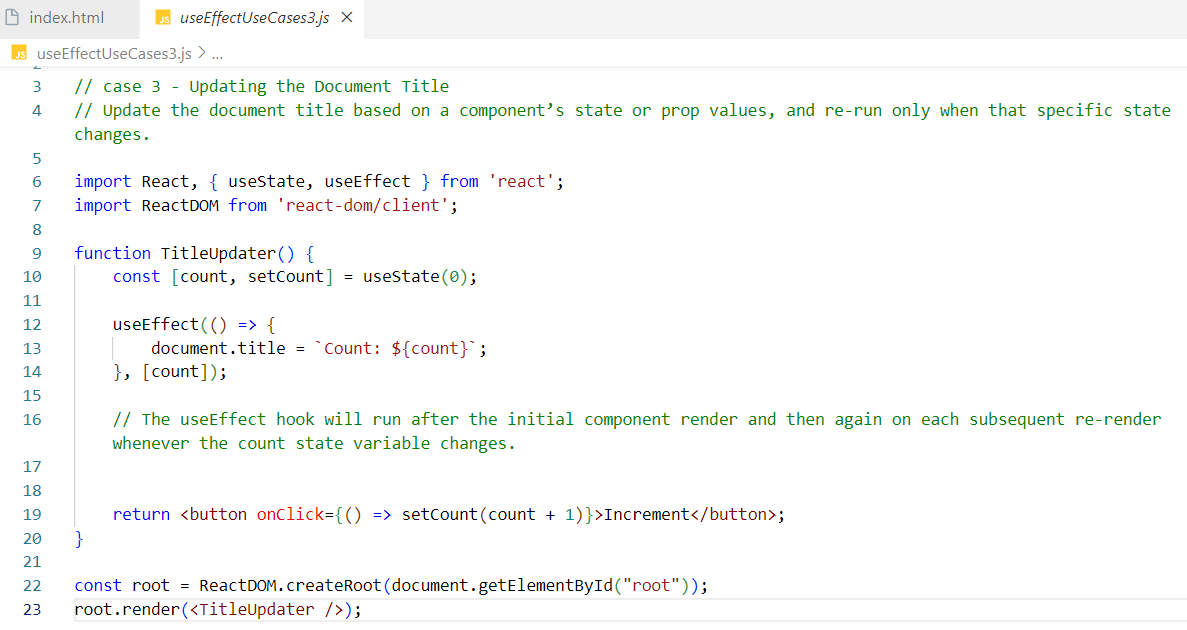
**Data Fetching:** Use useEffect to fetch data from an API when the component mounts. This is often done with an empty dependency array to ensure the data is fetched only once.



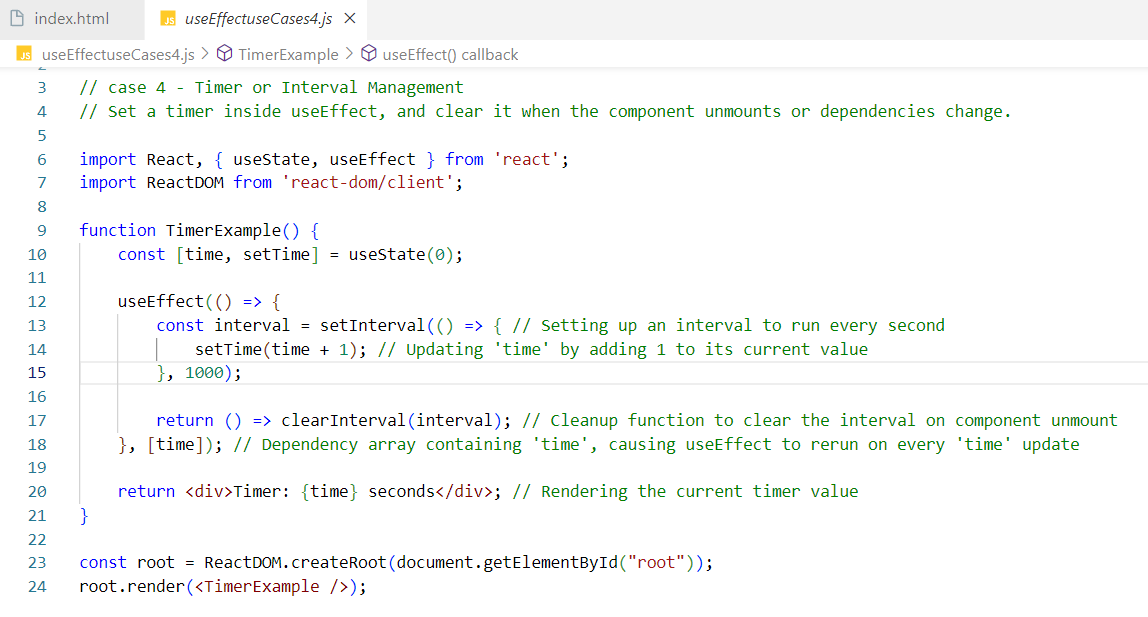
**Subscribing to Events:** Use useEffect to set up event listeners or WebSocket connections when the component loads. Make sure to clean them up when the component is removed from the page.



**Updating the Document Title -** Use useEffect to change document.title whenever the component’s state or props change.



**Timers and Intervals -** Use useEffect to start a timer or interval and make sure you clear it in the Cleanup function so it stops when the component unmounts (or when the effect re‑runs).



**Local Storage -** Use useEffect to save updated data to localStorage whenever the state changes, and retrieve the stored value when the component initially mounts.

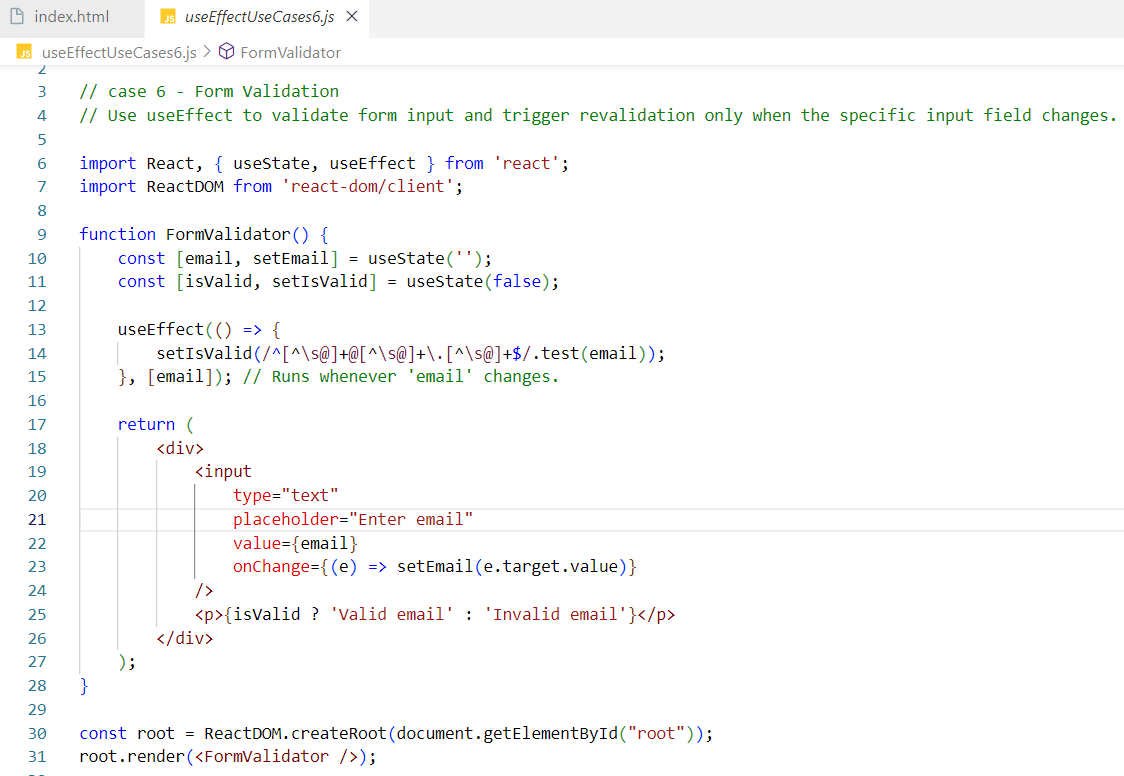


Explanation

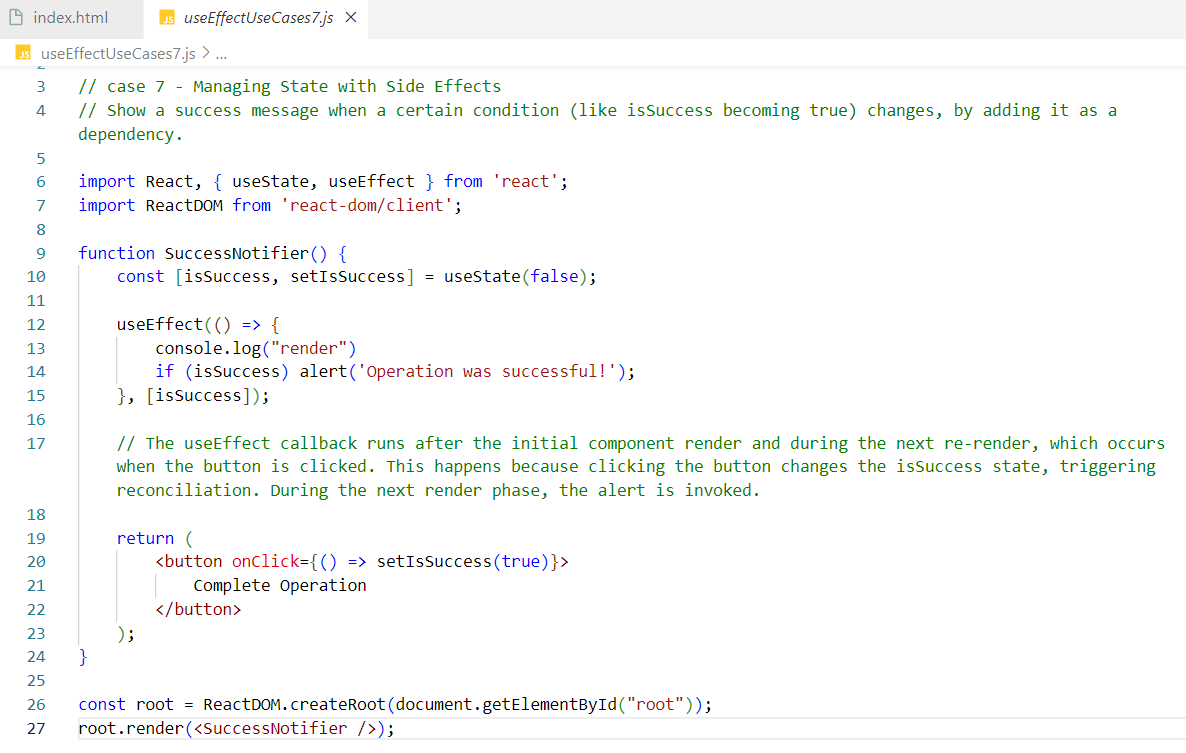
* State Initialization: useState retrieves the saved value from localStorage (if available) or sets it to an empty string.
* Updating LocalStorage: useEffect updates localStorage whenever the state (value) changes.

This ensures that the input field persists its value across page reloads.

**Form Validation -** Use useEffect to validate form inputs and update the component state based on the validation outcome.

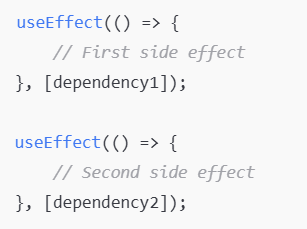


Managing State with Side Effects - Use useEffect to trigger actions in response to state changes, like resetting forms or showing success messages.

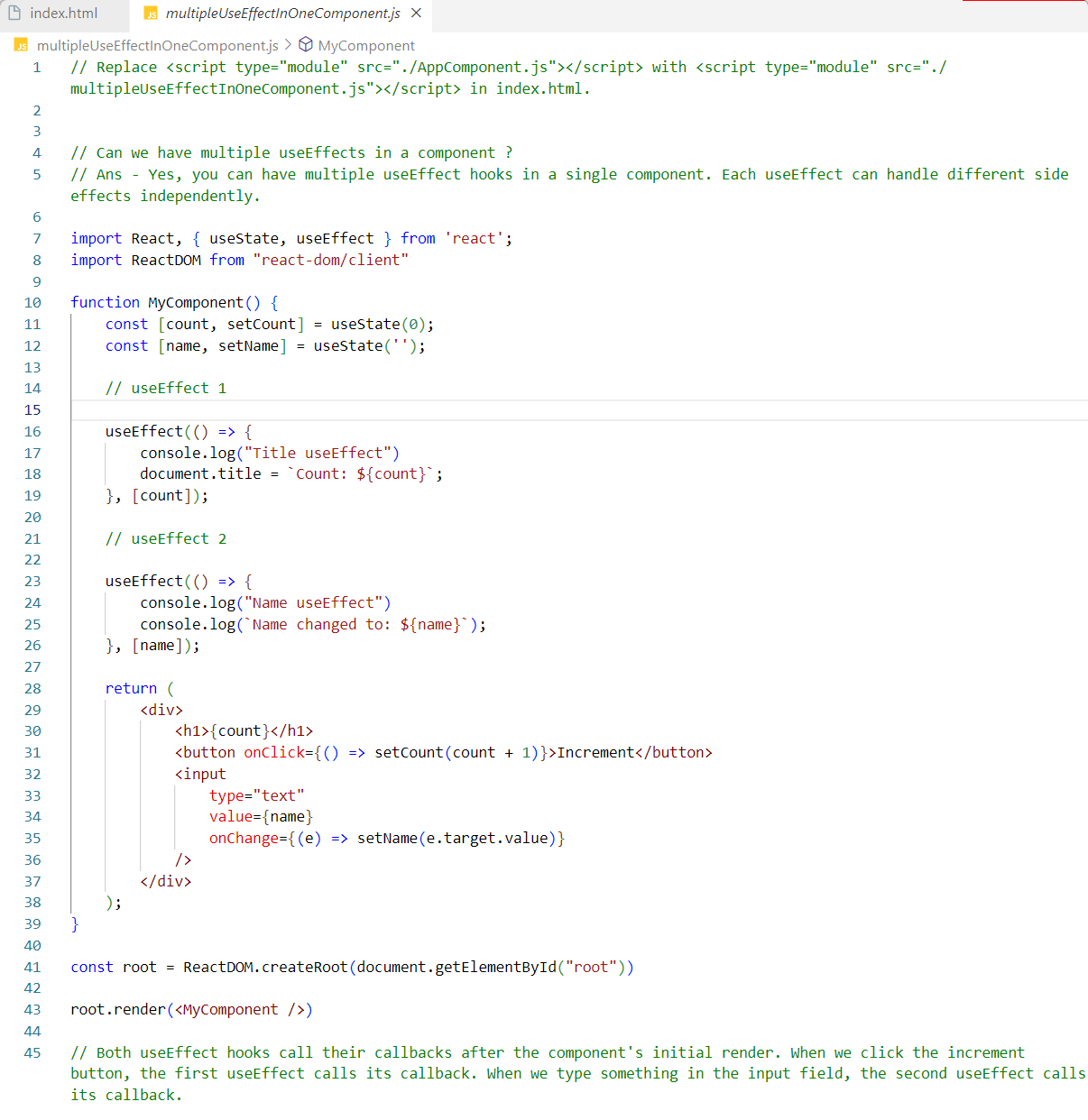


Can we use multiple useEffect in a component?

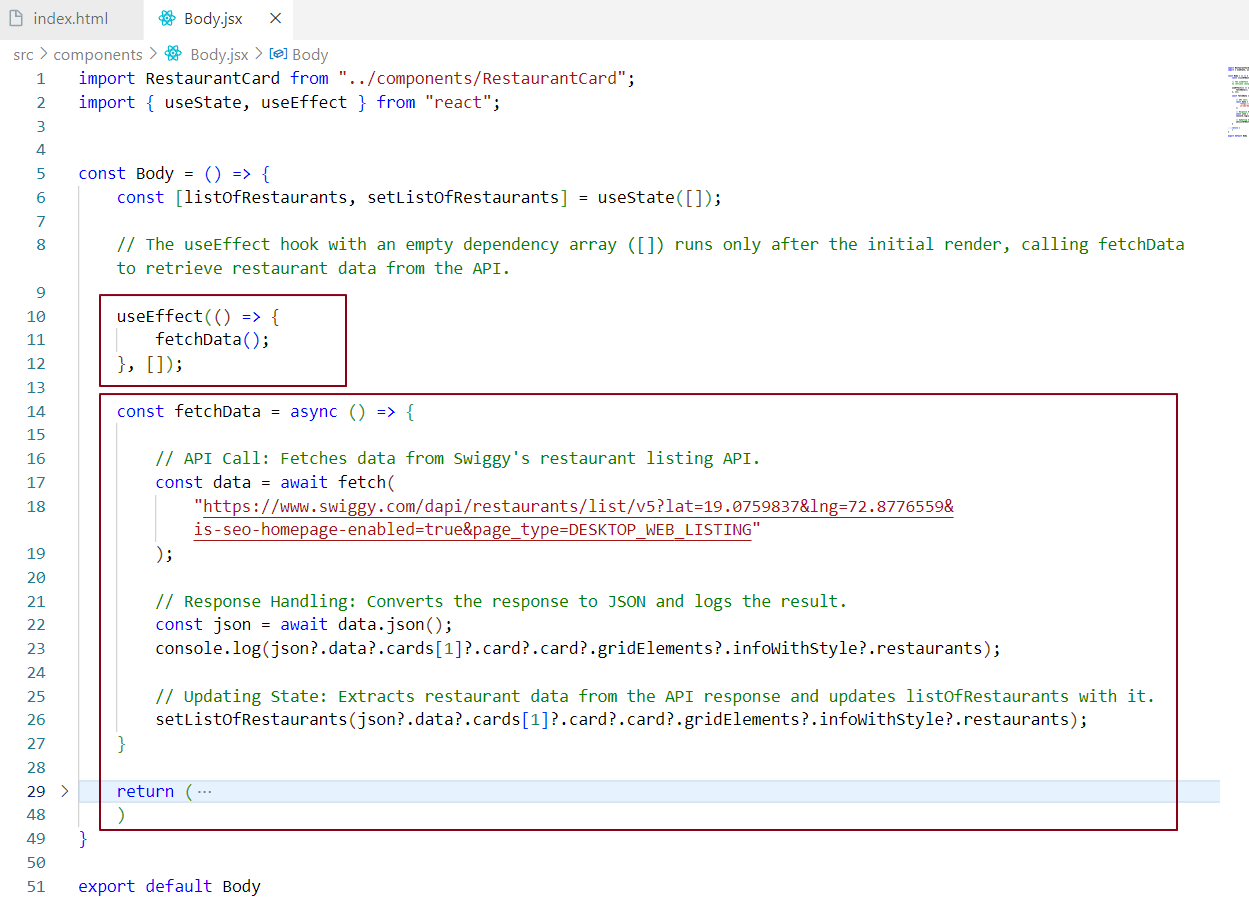
Yes, we can use multiple useEffect hooks in a component. Each useEffect can handle different side effects or actions based on specific dependencies, making the code more organized and easier to manage.

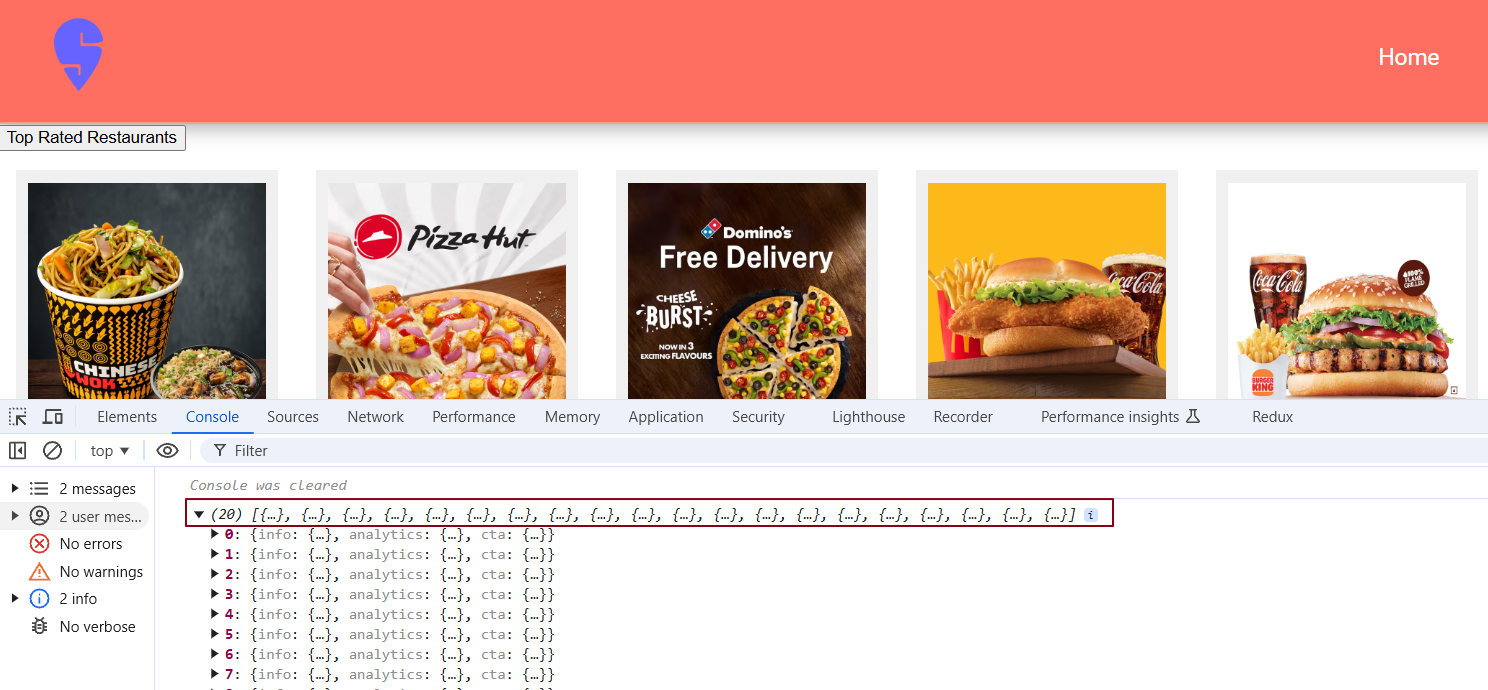


Each useEffect is independent, so they will run based on their respective dependencies.



Let’s place the Swiggy API call inside a useEffect hook.





Potential Issue: A CORS error might occur when calling the Swiggy API because the browser can block requests from localhost due to Cross-Origin Resource Sharing (CORS) policy restrictions.

🧠 Easy CORS Explanation (Playground Analogy)

Imagine this: You're a kid playing in your school's playground (your website's origin). You want to borrow a football from another school's playground (another domain/origin).

But the guard (the browser) stops you and says:

❌ "You're not allowed to take things from another school unless that school’s principal (the server) says it's okay."

That "okay" is called the CORS header (Access-Control-Allow-Origin) that must be set by the server.

**In Tech Terms:**

* Your website runs on localhost (like your school).
* You try to fetch data from swiggy.com (another school).
* The browser blocks it unless Swiggy's server explicitly says:  
  "Yes, I allow localhost to access me."  
  (via CORS headers)

💡 What is a Website's Origin?

A website’s origin is basically where a website is hosted, and it's made up of 3 key parts:

* Protocol: The method used for communication between a client and server (e.g., HTTP or HTTPS).
* Domain: The unique address of a website, which helps locate it on the internet (e.g., example.com or localhost).
* Port: A **port** is like a door on a server, where each port listens for specific types of incoming requests and allows the server to respond through that door.

## 🌐 Why Do We Need the Concept of "Origin"?

The **origin** (defined as: protocol + domain + port) helps browsers decide **whether to allow or block communication** between different websites. It plays a vital role in maintaining **web security**.

**🔐 1. Security**

The browser uses origin to protect **sensitive user data**, such as:

* Cookies (e.g., login tokens)
* Local and session storage
* API responses

Without origin checks, **malicious websites** could easily access or steal this data.

**🔄 2. Same-Origin Policy (SOP)**

This is a **security rule enforced by all modern browsers**.

* ✅ **Same origin** → Request is allowed  
  (Same protocol, domain, and port)
* ❌ **Different origin** → Request is blocked by default

This prevents one website (e.g., https://hacker.com) from reading data from another (e.g., https://bank.com), even if both are open in the same browser.

**🌍 3. What If Cross-Origin Access Is Needed?**

If your site needs to access a **different origin** (like calling a third-party API), the **target server must explicitly allow it** by sending a **CORS header**: Access-Control-Allow-Origin: https://your-website.com

This tells the browser: "It’s safe to share this resource with that specific origin."

**🚀** Simple Example:

* **Your Website:** https://mywebsite.com  
  → Origin: https + mywebsite.com
* **External API:** https://api.swiggy.com  
  → Origin: https + api.swiggy.com

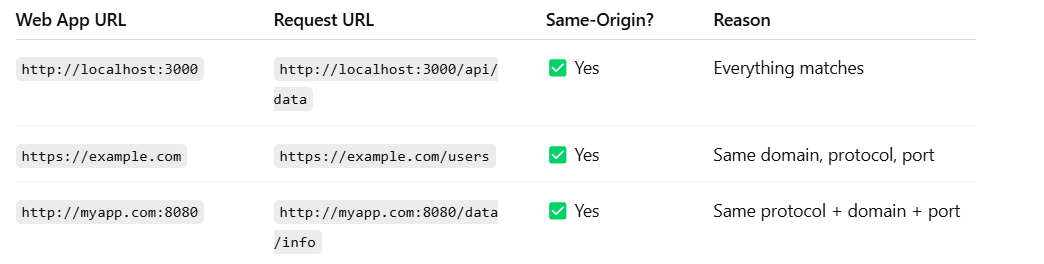
Since the domains are different, the browser sees this as a **cross-origin request** and blocks it **unless** api.swiggy.com allows it using **CORS headers**.

🔐 What is Same-Origin?

A request is considered Same-Origin when protocol, domain, and port are all exactly the same as the origin from which the page was loaded.

**✅ Rules:**

* Protocols must match (e.g., http vs https)
* Domains must match (e.g., example.com vs api.example.com)
* Ports must match (e.g., :3000 vs :5000)

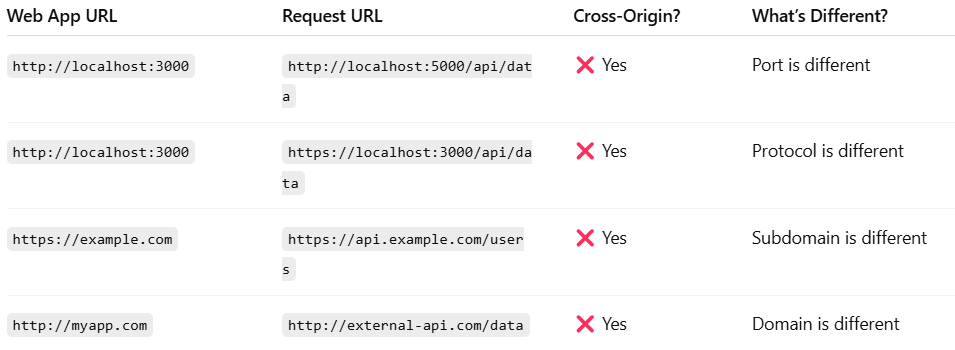


🚫 What is Cross-Origin?

A request is considered **Cross-Origin** when **any** of the protocol, domain, or port is different from the origin the page was loaded from.

**❌ Rules:**

* ❌ Different domain = Cross-Origin
* ❌ Different port = Cross-Origin
* ❌ Different protocol = Cross-Origin



What is CORS and Why Do We Get CORS Errors?

CORS (Cross-Origin Resource Sharing) is a security feature implemented by browsers that blocks requests made from one origin (e.g., localhost) to another (e.g., swiggy.com) unless the server explicitly allows it through special headers. These headers tell the browser which domains are permitted to access the server's resources. If the server doesn’t include the proper headers, the browser will block the request, resulting in a CORS error.

### 🌐 **Cross-Origin Request with CORS Headers**

### ****1. Case 1: Cross-Origin Request Allowed (CORS Headers Provided)****

When the server explicitly includes the correct CORS headers, the browser allows the cross-origin request.

* **Web App URL:** http://localhost:3000
* **Request URL:** https://api.example.com/data (Different domain — Cross-Origin)

#### **Server Code (Node.js with Express - CORS Allowed):**

#### 

#### **Response:**

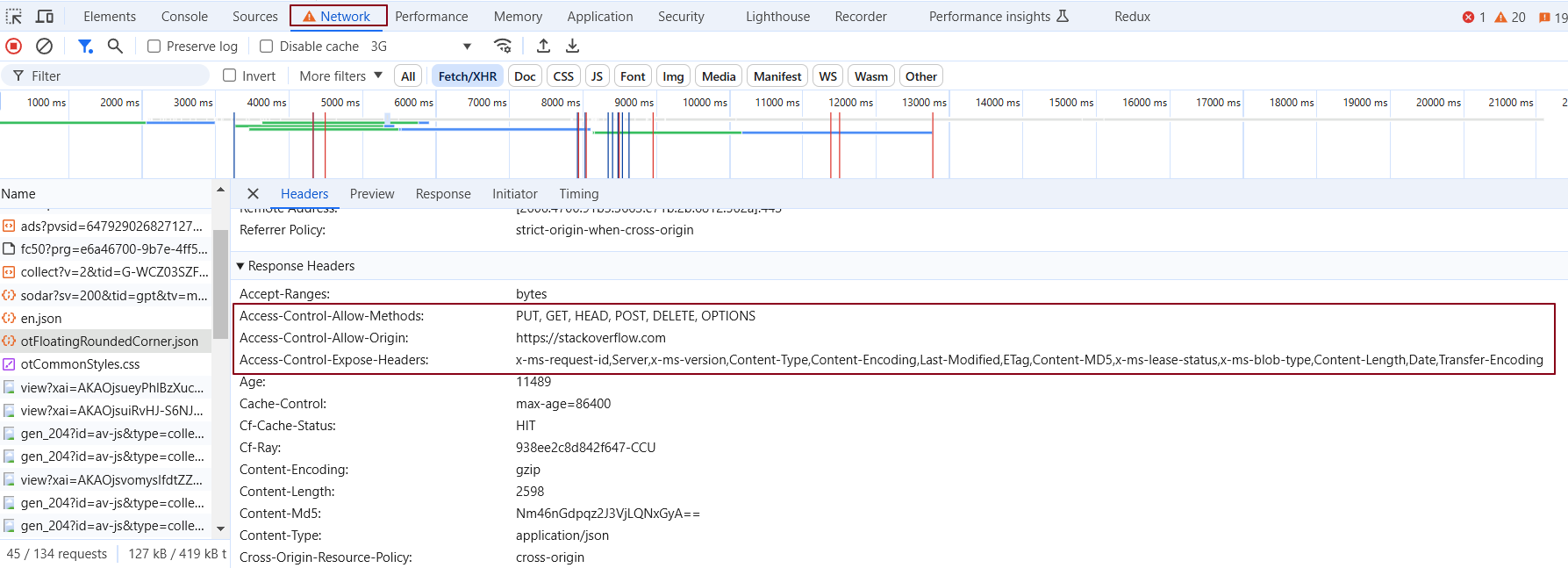
When the browser sends a request to https://api.example.com/data from http://localhost:3000, the server responds with the following headers:

Access-Control-Allow-Origin: http://localhost:3000

Access-Control-Allow-Methods: GET, POST

Access-Control-Allow-Headers: Content-Type

Since the server includes the **CORS headers**, the browser allows the request to go through, and the response is sent back to the client without any issues.



#### **Explanation:**

* The server allows cross-origin requests from http://localhost:3000 by setting the Access-Control-Allow-Origin header.
* Other headers, like Access-Control-Allow-Methods and Access-Control-Allow-Headers, define which methods and headers are allowed in the request.
* **Result:** The browser successfully processes the cross-origin request without any CORS error.

**2. Case 2: Cross-Origin Request Blocked (CORS Headers Missing)**

If the server **does not include the required CORS headers,** the browser will block the request for security reasons, resulting in a CORS error.

* **Web App URL:** http://localhost:3000
* **Request URL:** https://api.example.com/data (Cross-Origin)

#### **Server Code (Node.js without CORS Headers - CORS Blocked) :**



#### **Response:**

When the browser sends a request to https://api.example.com/data from http://localhost:3000, the server does **not** include any CORS headers. The browser will block the request and display a **CORS error** in the console:

Access to XMLHttpRequest at 'https://api.example.com/data' from origin 'http://localhost:3000' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.

#### **Explanation:**

* Since the server does **not** send the necessary CORS headers, the browser detects the request as cross-origin and blocks it.
* **Result:** The browser blocks the request, and the client does not receive the data, triggering a CORS error.

### Solutions for CORS Errors:

When you encounter **CORS (Cross Origin Resource Sharing)** errors, it means that your browser is blocking requests from one domain to another due to security reasons. Here are two common ways to handle or bypass these errors during development.

**1.** **Using a CORS Browser Extension**

* **What it does**: It temporarily modifies the browser’s behavior to **allow cross-origin requests**.
* **How it works**: A CORS extension can be installed in your browser to disable or adjust the security checks for CORS headers. This is typically used for **development purposes**.
* **Limitations**: This solution is quick but is **not recommended for production** as it bypasses security measures. It's just a development workaround.

**Example**: Extensions like "CORS Unblock" or "Allow CORS" can be used in browsers like Chrome or Firefox to unblock the requests during local development.

### ****2. Using a Proxy Server to Bypass CORS****

### **What it does:**A **proxy server** acts as a middleman between your frontend (browser) and the external API. Instead of the browser calling the external API directly (which may block the request due to **CORS**), it sends the request to the proxy server. The proxy server then:

* Calls the actual API.
* Forwards the response back to your frontend.

Since the browser is only communicating with the proxy (not the external domain), it **bypasses CORS restrictions.**

* ⚙️ **How it works:**

**👉 Without Proxy:**  
 Frontend → ❌ External API (Blocked by CORS)

**👉 With Proxy:**  
 Frontend → ✅ Proxy Server → ✅ External API  
Then: External API → 🔄 Proxy Server → ✅ Frontend

Example

In this code

* https://corsproxy.io/ is the proxy server.
* The actual API is passed as a query parameter to the proxy.
* The proxy forwards the request to Swiggy’s API and returns the response.

⚠️ Limitations of Using a Proxy Server to Bypass CORS

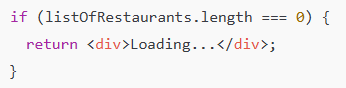
1. Performance Overhead: Requests are routed through an extra server (proxy), increasing response time.
2. Security Risks: Sensitive data is sent through a third-party server, which can compromise security.
3. **Rate Limits or Restrictions:** Public proxies often **restrict the number of requests** you can make, causing delays and slower response times.
4. Cost for Private Proxies: Using a private proxy for better control and security can incur additional infrastructure and maintenance costs.
5. **Downtime or Unavailability:** Public proxy services can be **unreliable**, may **go offline** unexpectedly, or could be **blocked** by the target API, preventing access to the service.
6. Not Meant for Production: Public proxies are typically not secure and are unsuitable for production environments.

**Now, let's return to our food ordering app.**

When fetching restaurant list data from the API, there is usually a brief delay before the data loads. To improve the user experience during this wait, consider adding a visual loading indicator to inform users that the data is being retrieved.

Loading Spinner

We could implement a loading spinner to appear while the data is being fetched. This provides immediate feedback to users, indicating that the application is actively working.



Shimmer UI

For an even smoother experience, consider adding a Shimmer UI effect. This involves displaying animated placeholder elements while the data loads, offering a more polished look. It helps set user expectations by showing an animated skeleton screen until the actual data is ready.

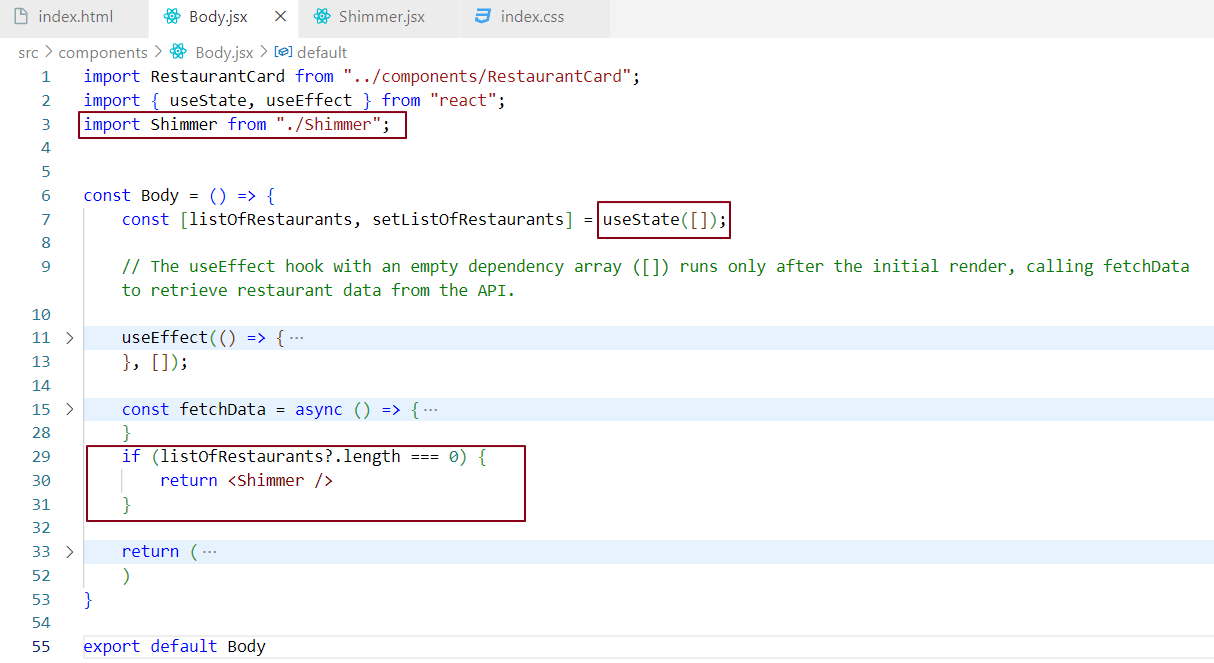
What is Shimmer UI?

Shimmer UI is a representation of a placeholder page with a shimmering effect. This effect is a visual technique used in UI design, where an animated light passes over an object or text, creating the impression that it’s shimmering or gleaming. In the past, we would show a loading spinner on the screen until data was fetched from an API and displayed in the UI. Shimmer UI provides a much better user experience by offering a more engaging and dynamic interface.

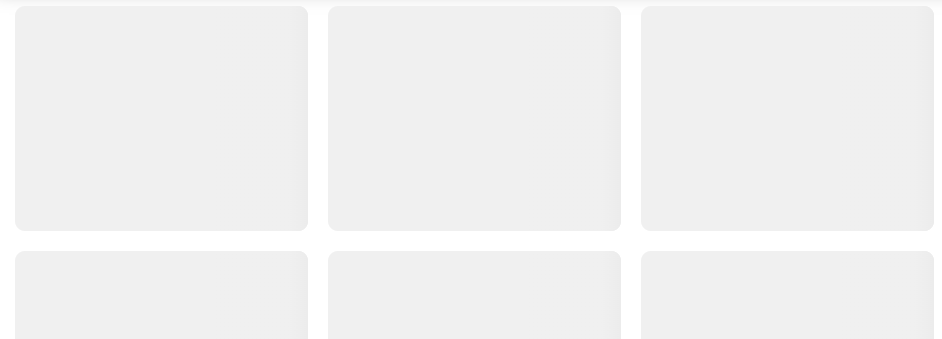
In the figure below, the end user can anticipate the appearance of cards that will load within the Shimmer UI. Once the data is fetched from the API, the placeholders are replaced with actual card content.



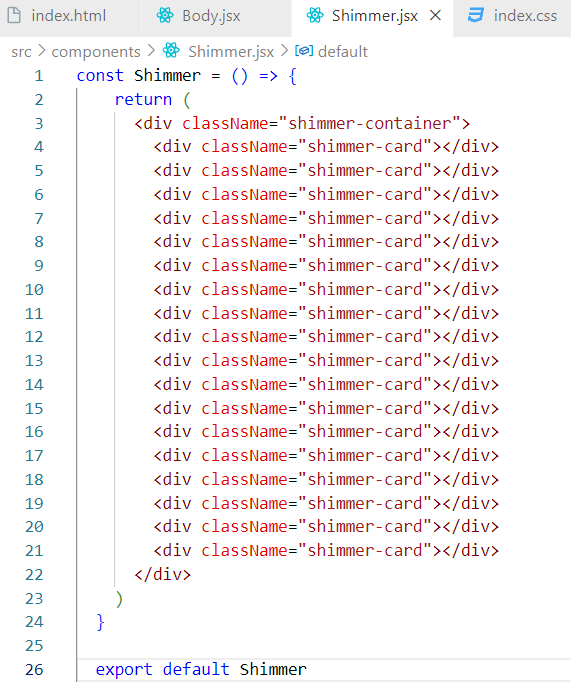
Let's integrate the Shimmer UI into our application.



The listOfRestaurants array will remain empty until the data is fetched from the API. In the meantime, we will display the Shimmer UI to indicate that content is loading.



Shimmer UI Component -



Ideally, we could use the map() function to refactor this code, but let's leave it as it is for now.

Shimmer UI Design -



Let's use conditional rendering to display the shimmer effect when the resList data is unavailable, and show the restaurant card data in the UI once the resList data has been fetched from the API.

Conditional Rendering in React

Conditional rendering refers to the practice of rendering components based on a specific condition.

**Syntax:** return condition? <Component1 />: <Component2 />;

If the condition is true, Component1 is rendered in the UI; otherwise, Component2 is rendered instead.



In the example above, we display a shimmer effect in the UI when the restaurant list data is unavailable. As we know, useEffect is called after a component renders or re-renders. In our case, useEffect is executed only once because we provided an empty dependency array.

During the initial render, the restaurants state variable is initialized as an empty array []. According to our conditional rendering logic, this causes the shimmer effect to be shown in the UI.

Once the component has finished its initial render, useEffect is triggered. The callback function inside useEffect calls the API to fetch the restaurant data. While the data is being fetched, the shimmer effect remains visible in the UI.

When the data is received, setRestaurants updates the restaurants state variable with the fetched data. As a result, the Body component re-renders. This time, since the restaurants state contains data, the list of restaurants is displayed in the UI instead of the shimmer effect.

Q) Why do we need State variable?

To better understand this, let’s introduce a feature in our app: a dynamic 'Login/Logout' button inside the Header component.



### Step-by-Step Explanation:

**Step 1:** We create a variable, btnName, with an initial value of "Login" and use it as the button text.

**Step 2:** We want the button text to change to "Logout" upon clicking. However, despite updating btnName and seeing the change logged in the console, the button text in the UI remains unchanged.

**Why doesn’t it change?**

The issue is that btnName is a regular variable, not a reactive one. In React, updating regular variables does not cause the component to re-render, so the UI doesn’t reflect changes to btnName.

## **✅ Solution: Using useState()**

To ensure the UI updates whenever btnName changes, we need to use a **state variable.** React’s useState() hook allows us to create btnName as a state variable, which automatically triggers a re-render whenever it is updated.

Using useState() for btnName makes the button reactive, allowing it to display the current state - either **"Login"** or **"Logout"** — based on user interaction.

When creating a dynamic **Login/Logout** button in React, using a regular variable won't trigger the necessary UI updates. Here’s how we resolved this using useState().

### Steps to Fix the Issue

**Step1 - Create a State Variable with** useState

* **What:** Instead of using a regular variable, define btnName as a state variable using the useState hook, initializing it with "Login".
* **Syntax:** const [btnName, setBtnName] = useState("Login");
* **Why:** State variables in React are reactive. When updated, they automatically trigger a re-render of the component, ensuring the UI reflects the latest value.

**Step 2 - Update State on Button Click**

* **What:** Add an onClick event to the button that calls setBtnName("Logout") to update the state.
* **Syntax:** <button onClick={() => setBtnName("Logout")}>{btnName}</button>
* **Why:** Calling setBtnName updates the state, which triggers a re-render and updates the button text in the UI.

**Step 3 - Automatic UI Update**

* **What:** Allow React to manage the re-render automatically.
* **Why:** With useState, the UI updates automatically to reflect the current state value ("Login" or "Logout"), without needing any manual DOM manipulation or re-rendering logic.

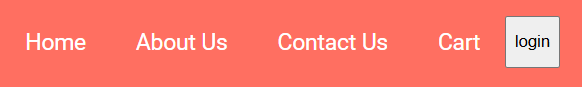


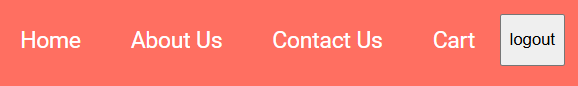
We need **state variables** in React because only they trigger component re-renders, ensuring the UI reflects updated values.

Implementing a Login/Logout Toggle Button in React –

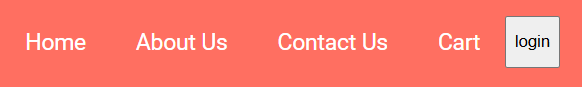


The code toggles btnName between **"Login"** and **"Logout"** using a ternary operator. It checks the current value: if btnName is **"Login"**, it sets it to **"Logout"**, and vice versa. This state update automatically triggers a re-render, updating the UI to reflect the new state.



When the user clicks the **Login** button, the button text changes from **"Login"** to **"Logout"**.

When the user clicks the **Logout** button, the button text changes from **"Logout"** to **"Login"**.



We have successfully implemented the button toggle functionality.

### If btnName is declared with const, how is it still updated when setBtnName() is called?

### It looks like the const value is being updated — but that's not actually what happens. In React, when you call setBtnName(), you're not modifying the const btnName directly. Instead:

1. React saves the new state value internally.
2. It then **re-renders the entire component**, meaning it calls the component function again from the top.
3. During this new render, the line:



is executed again — and React gives btnName the **new value** from its internal state (e.g., "Logout").

So even though you’re using const, you're always working with a **new variable** on each render, not changing the original one. That’s why using const with useState is completely safe and even encouraged in React.

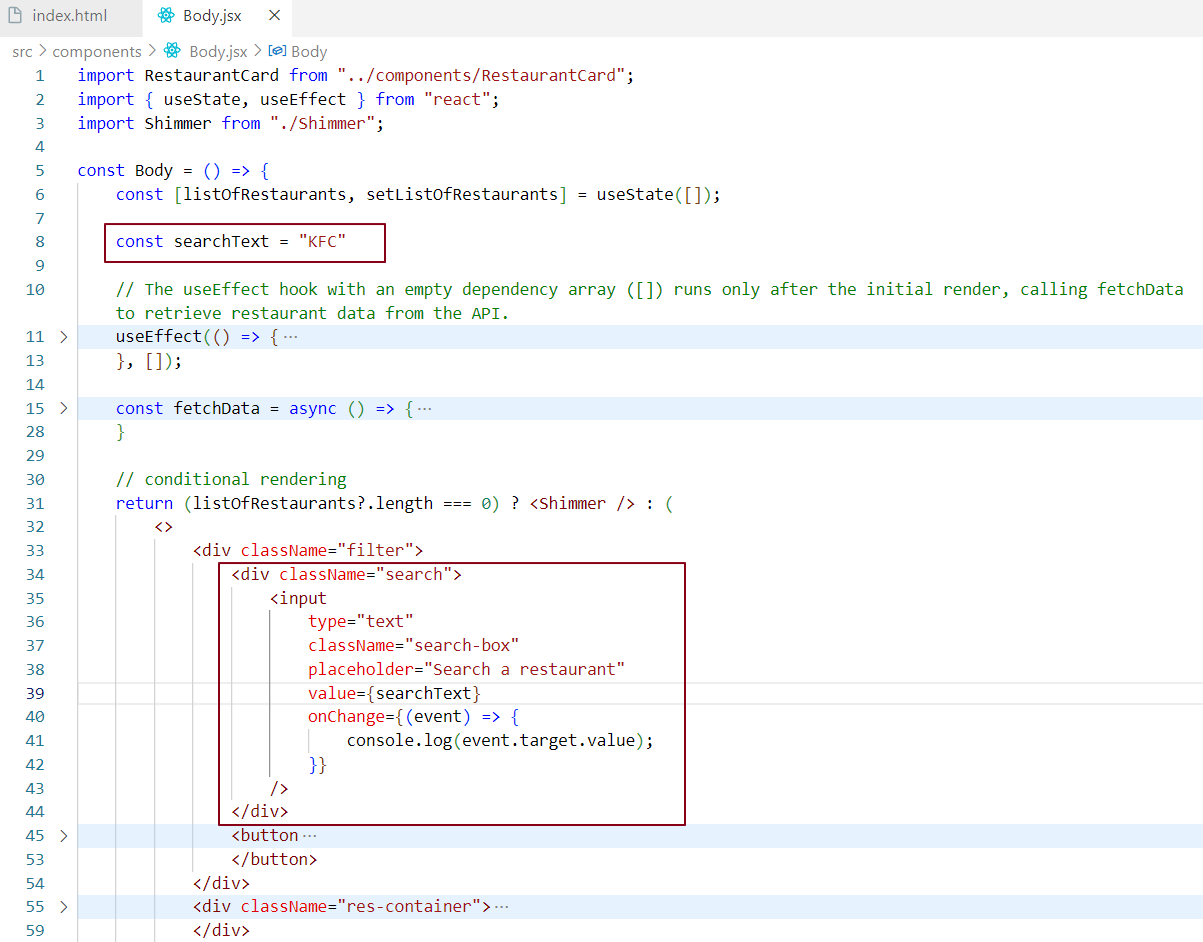
### ✅ Does React create a new const variable on every re-render?

**Yes, it does.**  
Each time a React component re-renders (usually after a state update), the **entire component function runs again from the top.** During this new run:

* All const variables (like btnName) are **re-declared.**
* React gives btnName the **latest state value** from its internal memory.
* So, you're not updating the original const; you're working with a **fresh one** every time.

This is safe because React's useState() remembers the value between renders and gives it back to you each time the component runs.

Implementing Search Functionality

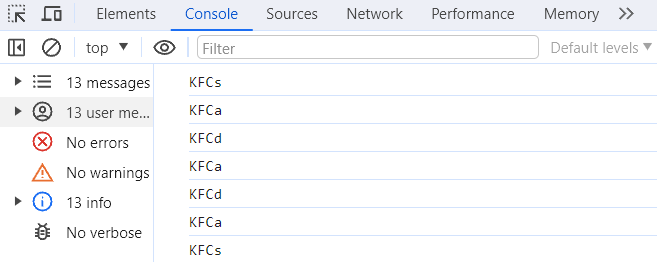


We added a search bar and created a local JavaScript variable called searchText inside the Body component. We set this variable as the value of the search bar, which is why "KFC" appears in the input box.

However, when we try to edit the value, nothing changes in the UI. Why does this happen?

* This issue arises due to React's one-way data flow. In our case, data flows from the local variable to the input field. We're simply displaying the value of searchText in the UI. To make the UI interactive, we need to read the data from the textbox and update the variable — enabling two-way communication.
* Another reason is how React re-renders components. When a change event (like typing) occurs, the onChange handler is triggered, causing React to quickly re-render the Body component. During each render, the value "KFC" is reassigned to searchText, which updates the input field again. As a result, no matter how many times we type in the input, react resets it to "KFC" on every render.

For now, the code logs the updated value to the console whenever the input changes.



The input field is effectively read-only because it's bound to a constant (searchText = "KFC") that doesn't update, so React resets it on every re-render.

This is not the behavior we want — we expect to see the updated text instead of a hard-coded value.

To achieve this functionality, regular local JavaScript variables are not sufficient. We need to maintain the variable's state within the component, which is why React state variables are used.

In our code, if the local variable searchText is modified by a function, react won’t be aware of the change. Even if the variable is used in multiple places, react doesn’t know *why* it was updated — and as a result, it won’t display the updated value in the UI. This happens because regular variables are stateless - they don’t retain their value across renders, and React doesn't track them.

**Note:**  
If we want our variables to stay in sync with the UI, we must use state variables. These allow React to track changes by maintaining their state. This is what enables two-way data binding in React — keeping the UI and data consistent at all times.

Understanding Data Binding in React

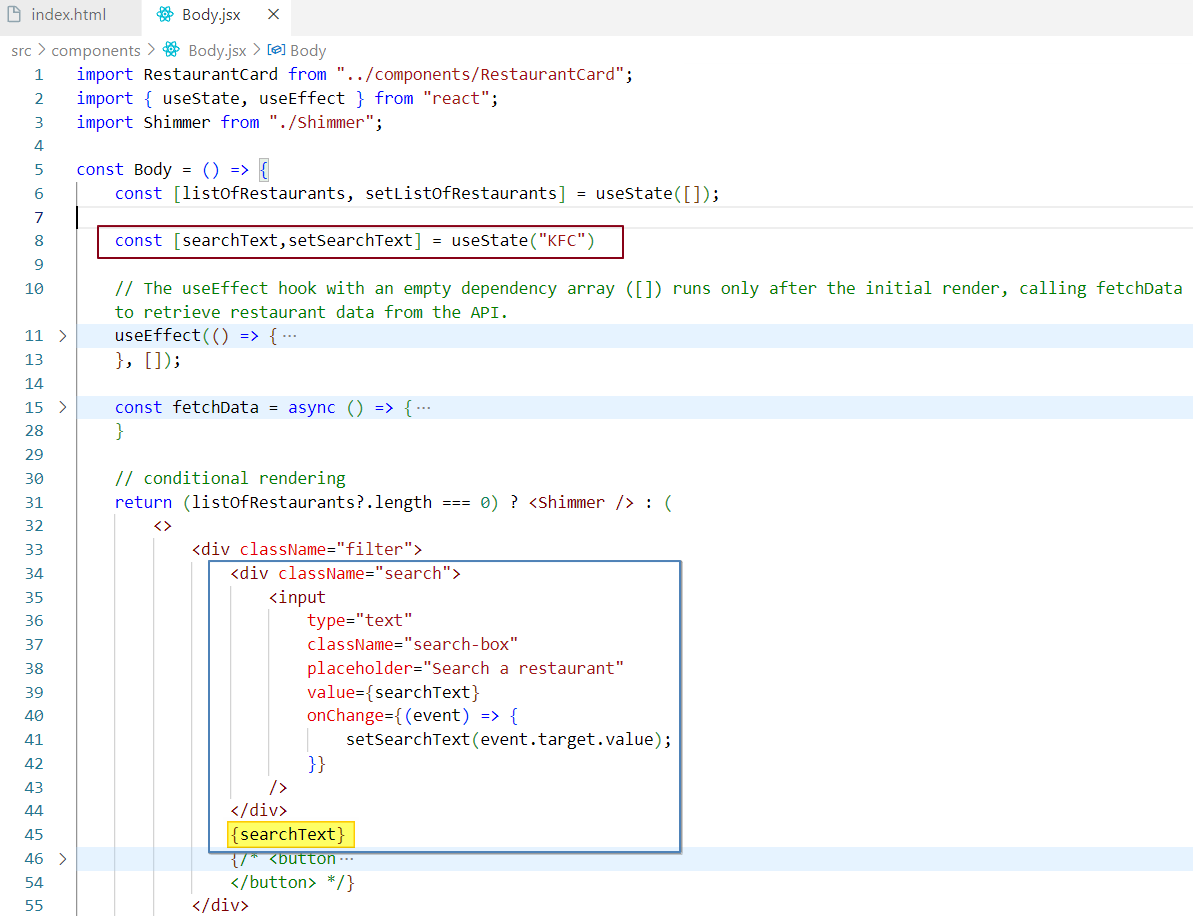
In React, there are two types of data binding:

1. One-Way Data Binding: Data flows in a single direction—from the component (state or props) to the UI.
2. Two-Way Data Binding: Data flows in both directions—changes in the UI also update the component's state.

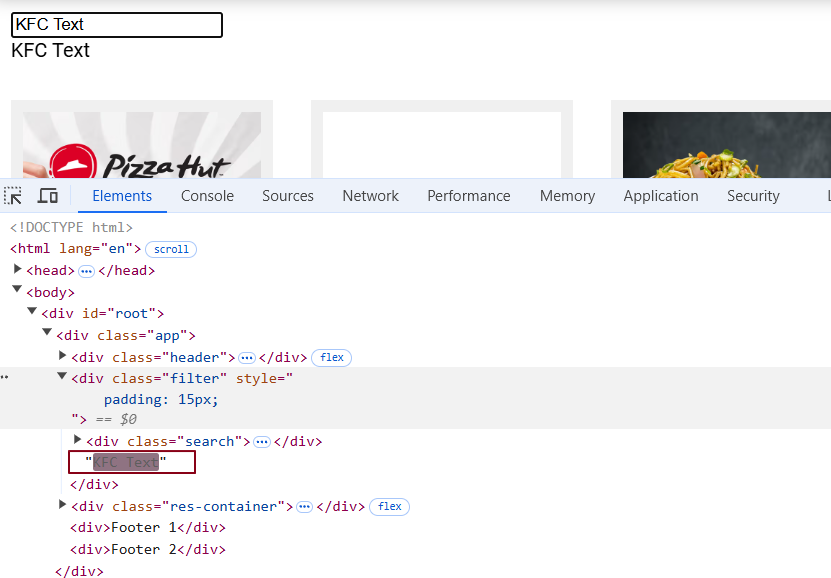
In our code example, we placed a local JavaScript variable into the textbox, which writes the variable’s value to the input field. This demonstrates React’s one-way data binding. However, our goal is to allow editing of the textbox value, meaning the variable should also update when the input changes.

In other words, we need to both read from and write to the variable simultaneously. This concept is known as two-way data binding.

So far, we've seen one-way data binding in action. Now, to enable two-way binding, we’ll use the useState hook - which allows React to track and update the state as the input changes.



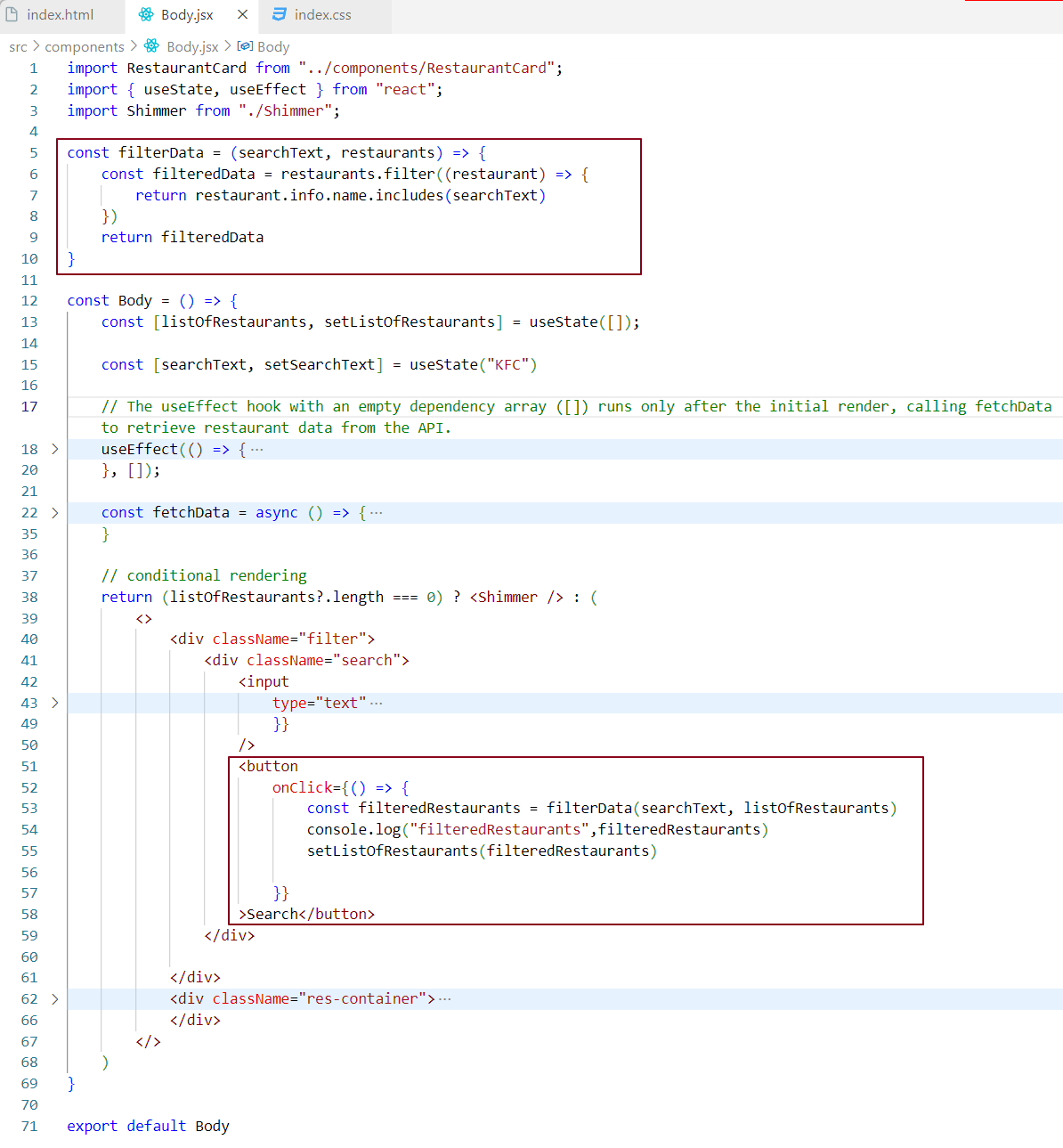
I placed the React state variable searchText right below the search input field. The setSearchText function, provided by the useState hook, updates the value of searchText. Since it's a state variable, react tracks its changes and re-renders the component whenever it gets updated.



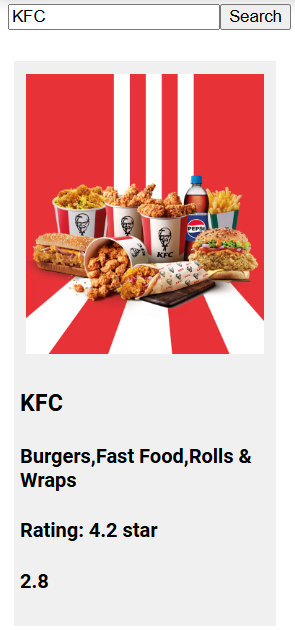
Whenever searchText is updated or changed, react re-renders the entire Body component but only updates the necessary parts of the DOM. This efficient update process is managed through React’s reconciliation mechanism and its underlying diffing algorithm.

Note: When we say React "re-renders" the component, it means React discards the current instance of the Body component and creates a new one — very quickly. So, when you type into the search box, searchText gets updated, and the UI reflects those changes immediately. This is a clear example of two-way data binding in action.

Next, let’s add a button beside the search bar and implement the filter logic to respond to user input.

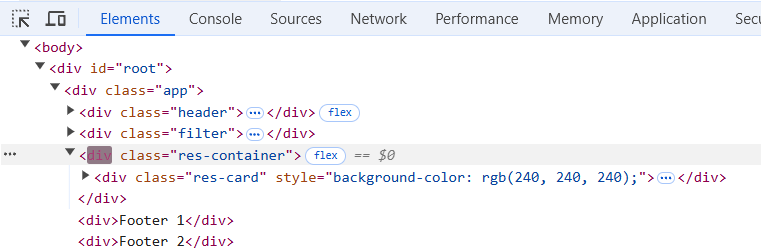


**When the search button is clicked, the app filters the list of restaurants based on the text entered in the input field.** The filtered results are stored in the listOfRestaurants state variable, which triggers a re-render to display the updated restaurant list in the UI.



**When the search button is clicked, the state variable** listOfRestaurants **is updated, triggering reconciliation.**

This invokes React's diffing algorithm, which compares the new virtual DOM with the previous one. Once it identifies the changes, it updates only the modified portion in the actual DOM.  
You can visually observe this in the figure below - the highlighted <div> represents the updated, filtered restaurant that is rendered in the UI.



Here, we have filtered the restaurants based on their names. By entering "KFC," we expect our filter logic to work, and it does, successfully listing this card from the restaurant list and displaying it in the UI.

**However, there's a limitation with this approach: the search functionality works only once.**If you try to search for a restaurant a second time, it won’t function as expected.

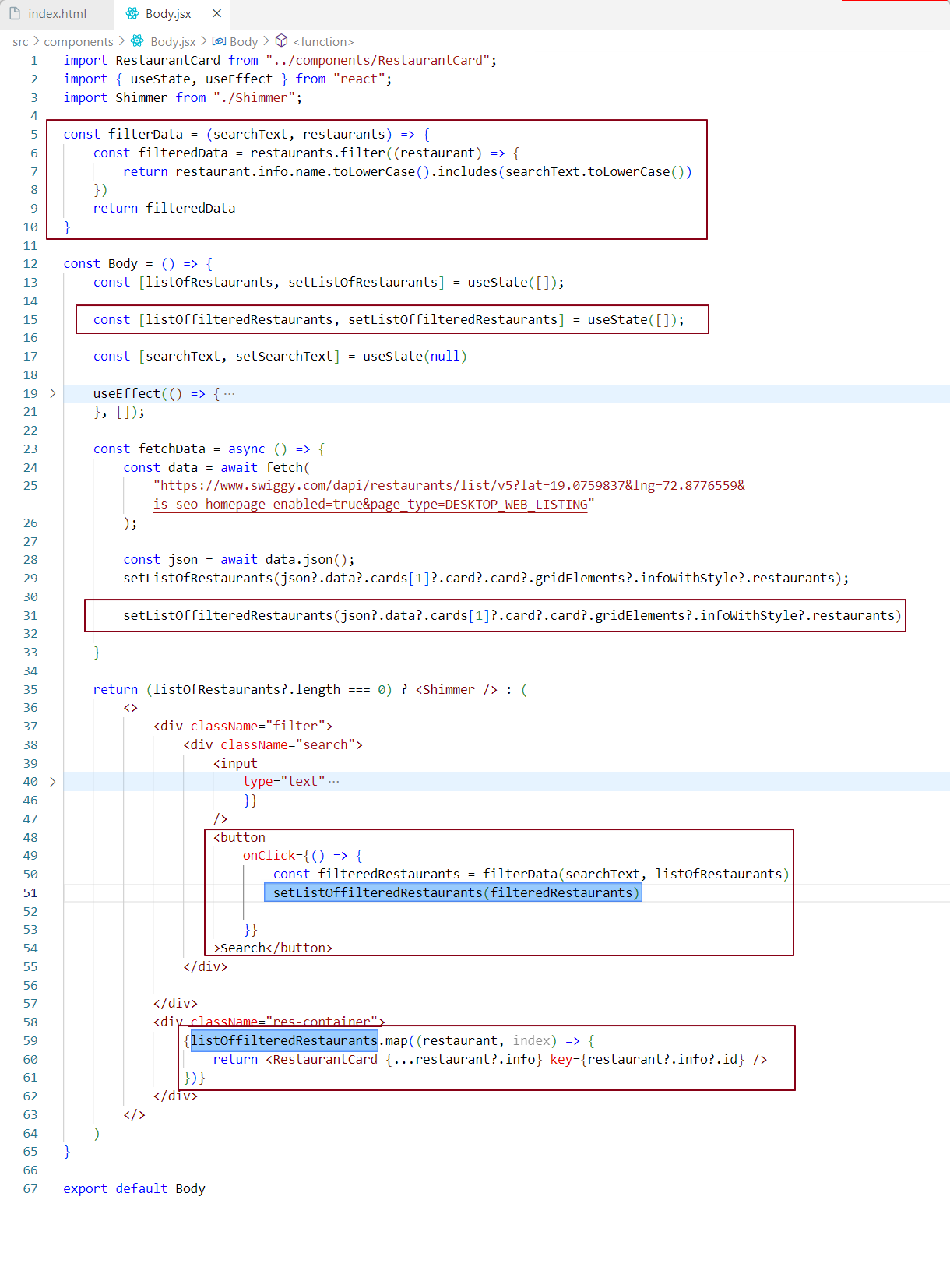
Why Is the Search Functionality Not Behaving as Expected?

When we type text into the search box and click the search button, the filterData function is invoked. This function returns a list of filtered restaurant data, which is then used to update the listOfRestaurants state variable using setListOfRestaurants(filteredData). As a result, the Body component re-renders and displays only the filtered restaurant cards in the UI.

At this point, the listOfRestaurants state variable contains only the filtered data. Therefore, during the next search, the input is compared against this already-filtered list. If the searched restaurant isn't found within this smaller list, the result is a blank screen, indicating no matches.

To fix this issue, the search should always be performed on the complete list of all restaurants, not just the filtered results. To do that, we need two separate state variables: one to store the full list of restaurants and another to store the filtered list. This allows us to preserve the original dataset and apply filtering as needed, without losing the complete source of data.

As a solution, we will define two state variables inside the Body component—one for the full restaurant list and one for the filtered results.



We’ve created two state variables: listOfRestaurants and listOfFilteredRestaurants.

* listOfRestaurants holds the full list of all restaurants.
* listOfFilteredRestaurants stores only the filtered list of restaurants, which is used for display when a search is applied.

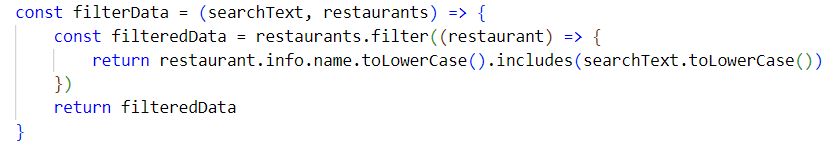
When the API is first called, both state variables are populated with real-time data, initially starting as empty.  
Previously, we applied the filter logic directly to listOfRestaurants, which handled both storing and rendering data in the UI. Now, we apply the filter logic specifically to listOfRestaurants but update only listOfFilteredRestaurants for display.

This way, listOfRestaurants always contains the full list of restaurants, while listOfFilteredRestaurants changes based on the filter and is used to display the filtered results in the UI.

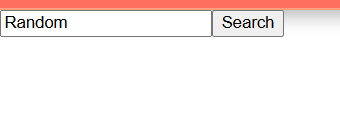
In this setup, listOfRestaurants acts as the main data store, and listOfFilteredRestaurants serves as the data provider for the filtered view.

**Updated Filter Logic:**

The search is now case-insensitive, so it doesn’t matter if you use uppercase or lowercase letters. Earlier, it was case-sensitive, which could miss results.



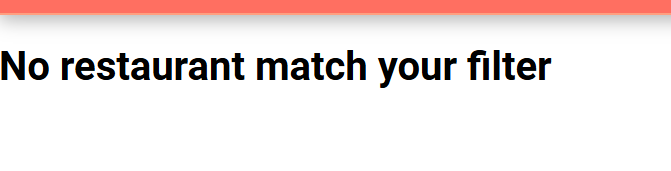
The search now works well. But there’s a small problem — if no results are found, the page just looks blank instead of showing a message.



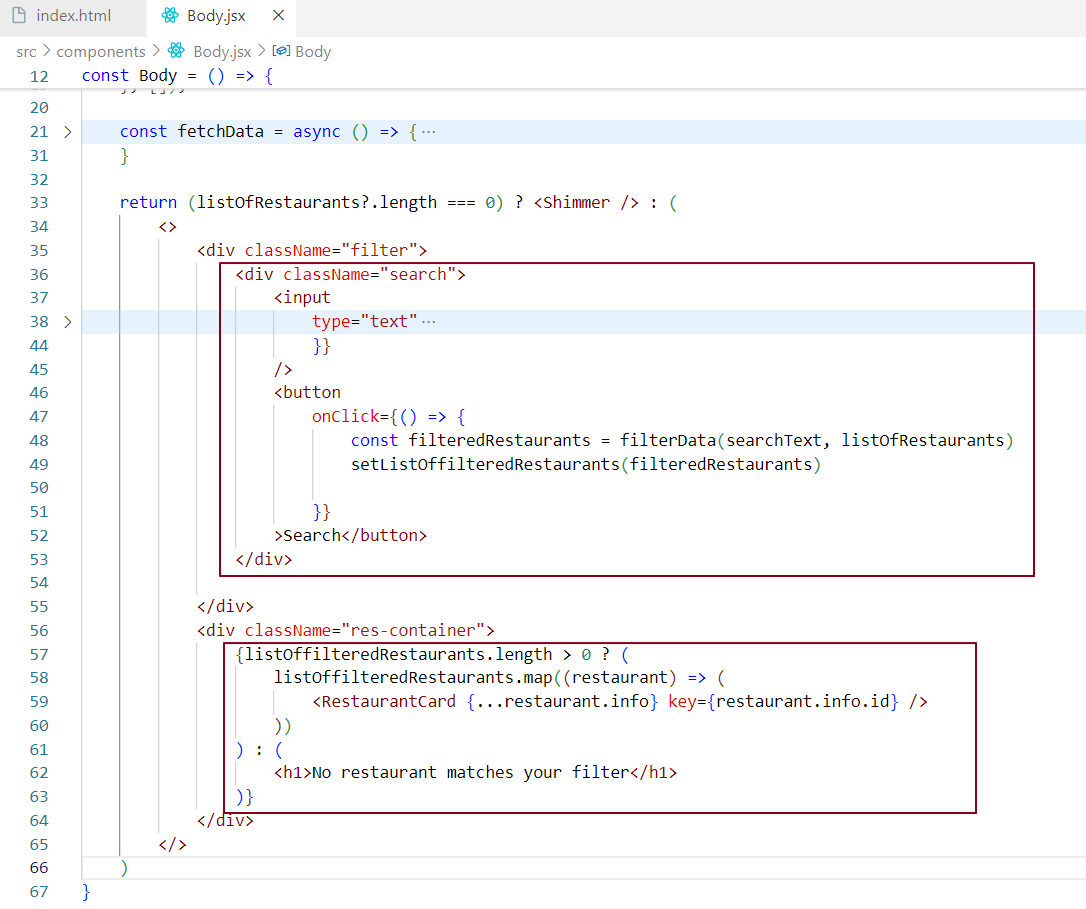
**Let’s fix this issue:**  
Instead of showing a blank space when no matching restaurants are found, we can display a friendly message like "No restaurants match your filter." This improves the user experience and makes the app more informative.

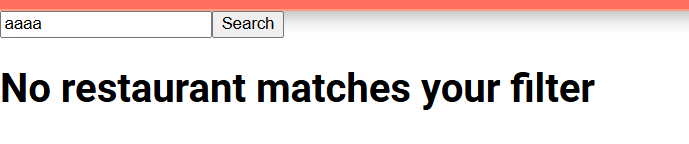


Now, when an invalid search is entered, we correctly display a message saying, "No restaurant matches your filter."  
However, this also causes a new issue: the search bar disappears from the UI.

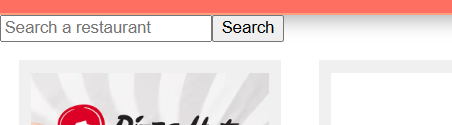


Let’s fix this issue by moving the conditional logic for listOfFilteredRestaurants inside the return statement and using conditional rendering. This approach ensures that the search bar remains visible, even when no restaurants match the search filter.



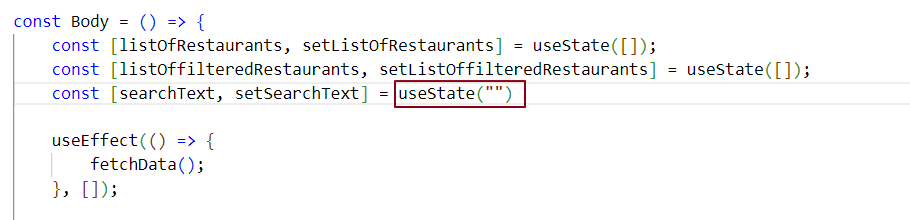


When the page first loads, searchText is initialized as null. If we click the search button without entering any text, this can lead to an error.





This error occurs because restaurant?.info?.name can be null, and calling toLowerCase() on null results in an error.  
To fix this, we should initialize searchText with an empty string ("") instead of null, ensuring that string methods like toLowerCase() can be safely called.



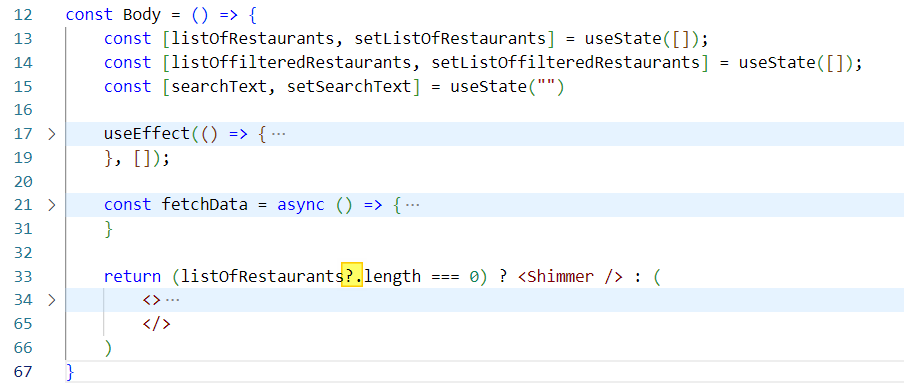
✅ This will resolve the issue. Now, let’s understand an important concept in React called ***Early Return***.

**What is Early Return in React? Why Use It?**

Early return in React means stopping the component from rendering if the required data isn't available yet.  
It's used to prevent rendering errors and improve performance by avoiding rendering logic when the necessary data hasn’t been loaded or initialized.

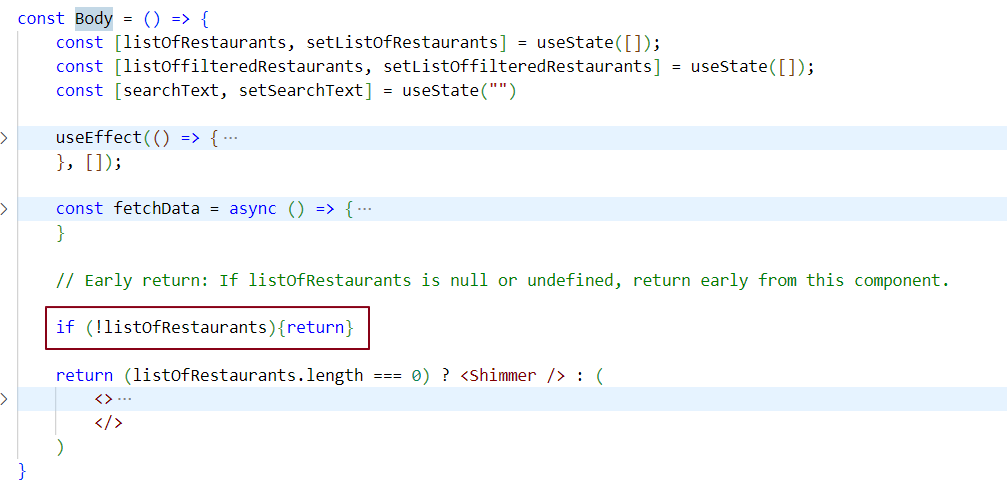
**Example: Early Return with Optional Chaining**

Optional chaining (listOfRestaurants?.length) safely checks if listOfRestaurants exists before accessing ***.length***, preventing runtime errors and enabling an early return to show *<Shimmer />* only when needed.



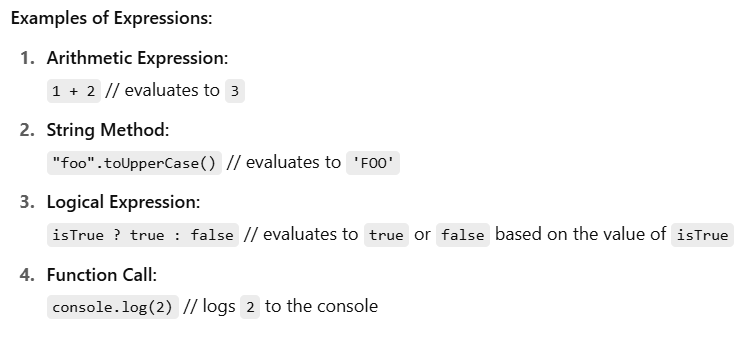
**Example of Early Return with Conditions**

We use early return (if (!listOfRestaurants) return;) to stop rendering when data is missing, preventing errors and unnecessary UI updates.



### **JavaScript Expressions vs. Statements**

**JavaScript Expression:**  
An expression is any valid unit of code that resolves to a value. It can include values, variables, operators, and functions that produce a result.



**JavaScript Statement:**  
A statement performs an action but does not necessarily return a value. Statements control the flow of execution in a program.

### 

### **Using Expressions and Statements in JSX**

#### **Using Expressions in JSX**

In JSX, you can embed any **JavaScript expression** by wrapping it in curly braces {}.

For example:

#### **Using Statements in JSX**

You **cannot use JavaScript statements** (like if, for, or let) directly inside JSX.

JSX only allows **expressions that return a value.**

#### **Recommended Approach**

Perform logic and control flow **outside JSX** or within functions that return expressions.



### **Important Points**

* **DOM Updates:** The actual updates to the DOM are handled by the React DOM library.
* **Diffing Algorithm:** React's core library implements the diffing algorithm to efficiently compare and update the DOM.
* **State Changes:** When a state variable changes, react re-renders only the components that depend on that state.
* **Prop Changes:** When props change, react re-renders the components that receive those props.