Chapter 6 – Exploring the World

Before we explore external frameworks or advanced topics, it's important to understand some fundamental design patterns commonly followed 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:

**Definition:** Monolithic architecture is an approach where all components of an application (UI, Business Logic, Data Access, Authentication, API) are combined into a single, tightly-coupled unit or package. This means that all parts of the application are interdependent.

**Pros**

**1. Simple to develop** – Ideal for small teams and projects, as you can develop everything in one codebase.

**2. Easier to test** – Since 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, which simplifies the deployment process for small or less complex systems.

**4. Performance** – 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:

**Definition:**

Microservices architecture is an approach in which an application is composed of small, loosely-coupled, independent services. Each service is responsible for a specific function and can be developed, deployed, and scaled separately. These services communicate with one another via well-defined APIs. They adhere to the Single Responsibility Principle, ensuring separation of concerns. For example, the UI service, backend API service, authentication service, and notification service are all independently deployable. While they interact with each other, they maintain clear boundaries and separation of responsibilities.

**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.

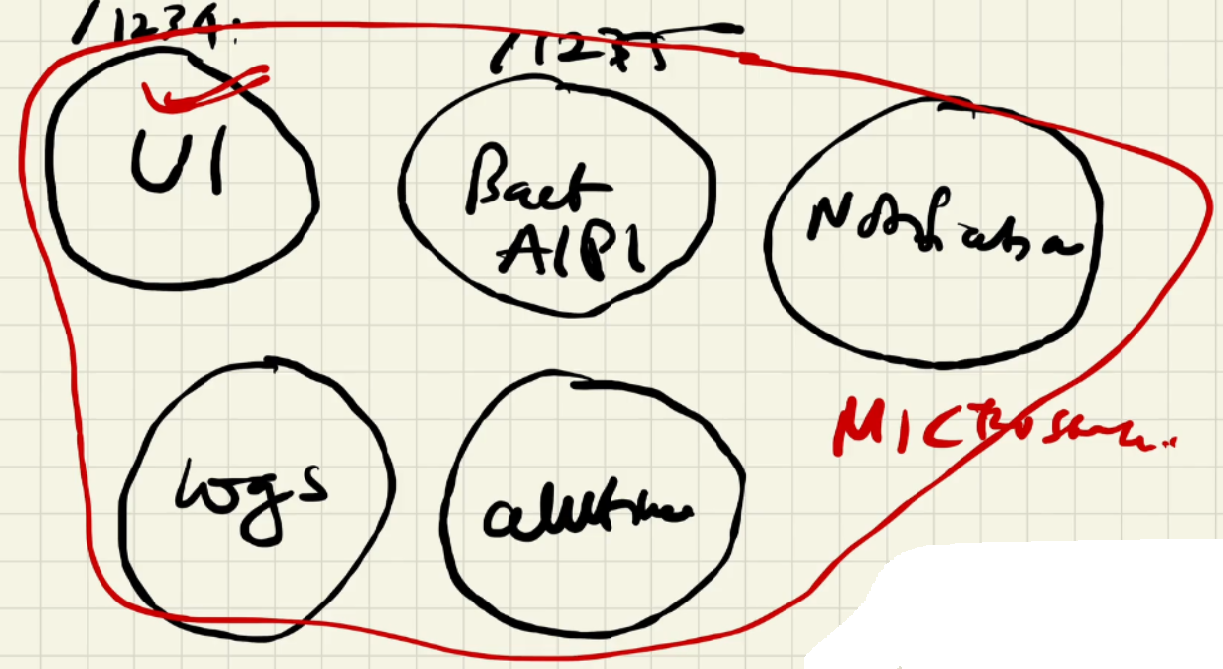
How do these services interact with each other?

In our setup, the UI microservice is written in React, which handles the user interface.

These services interact with each other through various communication channels. For instance, the UI microservice might need data from the backend microservice, which in turn might need to access the database.

Each microservice runs on its specific port. This means that different services can be deployed independently, with each one assigned to a different port. All these ports are then mapped to a domain name, providing a unified access point for the entire application.

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**: We can make an API call using the fetch () method.

**In React**: We typically make API calls within the useEffect () hook (discussed in detail later).

Methods for Making API Calls in React

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

1. **Approach 1:** Load the website➔ Make the API call (300 ms) ➔Render the page (200 ms)
2. **Approach 2:** Load the website➔Render the initial page (100 ms) ➔Make the API call (300 ms) ➔Update the UI with data.

In Approach 1, the page becomes available after a combined 500 ms (300 ms for the API call + 200 ms for rendering).

In Approach 2, the initial page is available in 100 ms, which is faster, improving the user experience by minimizing load time.

Approach 2 is generally preferred for better user experience, as users see content faster.

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 useEffect Hook?

useEffect is a special function (hook) that lets us do things 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 behaviour 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.

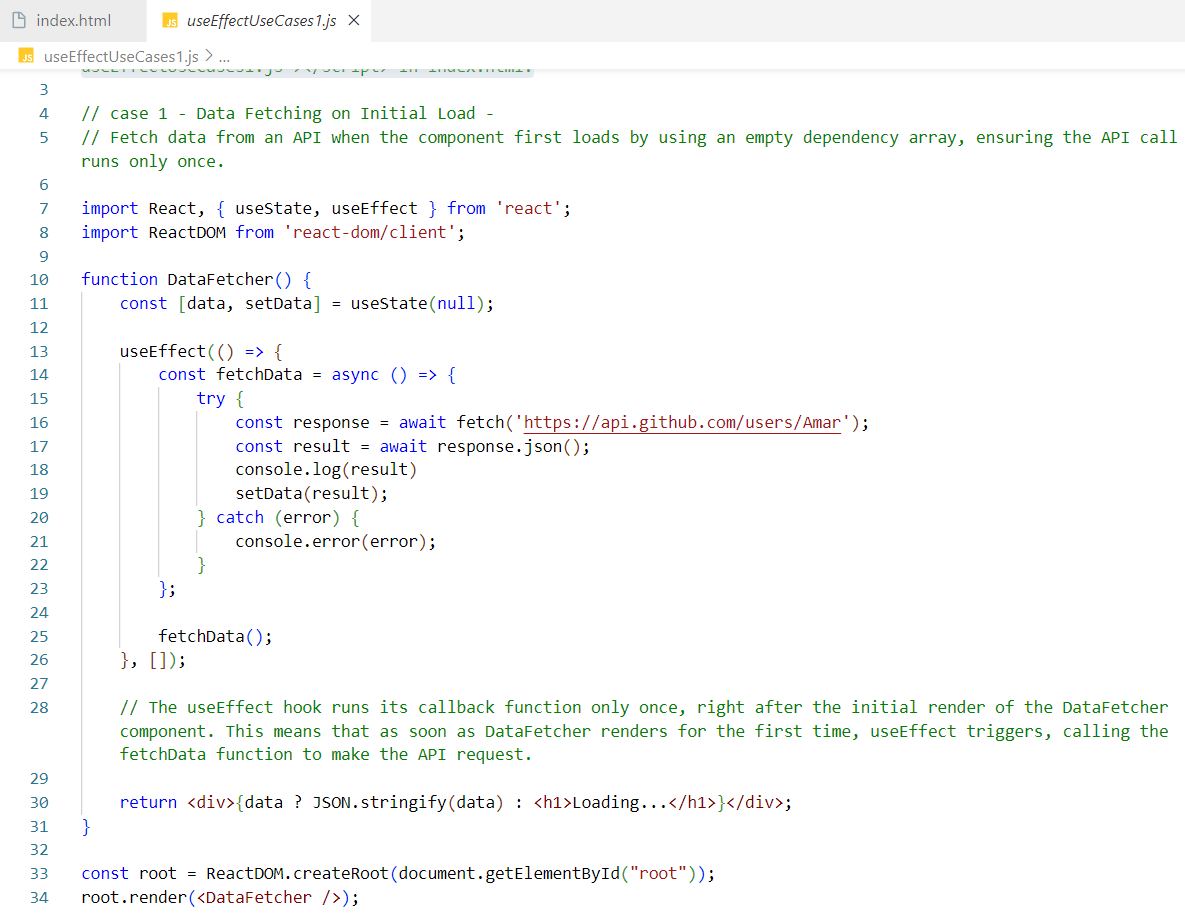
\* The dependency array is optional. If we omit it, useEffect will run its callback after every render and re-render of the component, as it doesn't depend on any specific variables.

When does our component renders and rerenders?

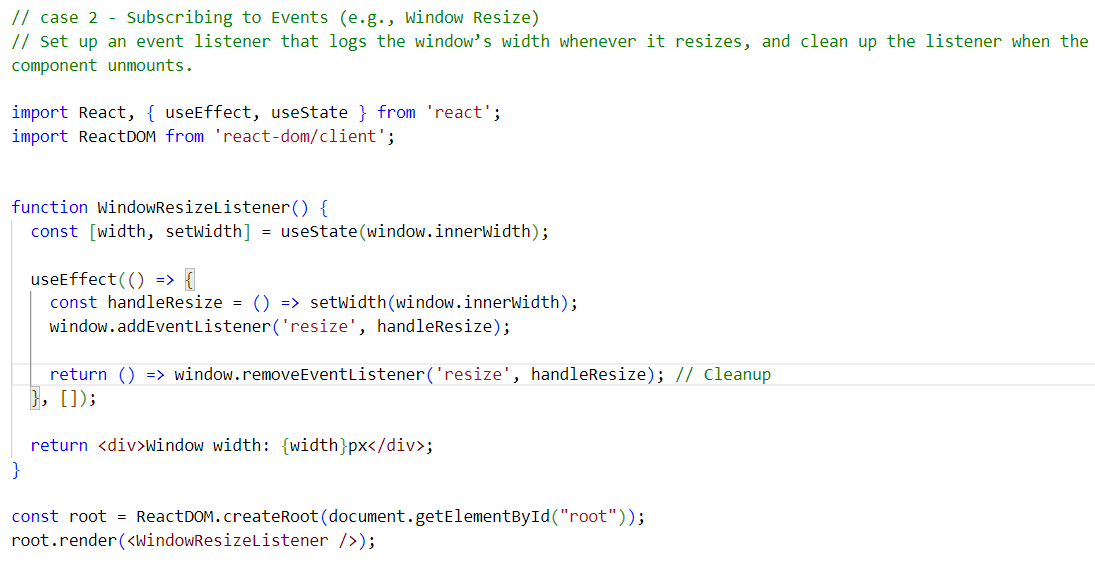
When our page loads for the first time, the component renders. The component re-renders whenever there is a change in its state or props

What are the use cases of useEffect Hook?

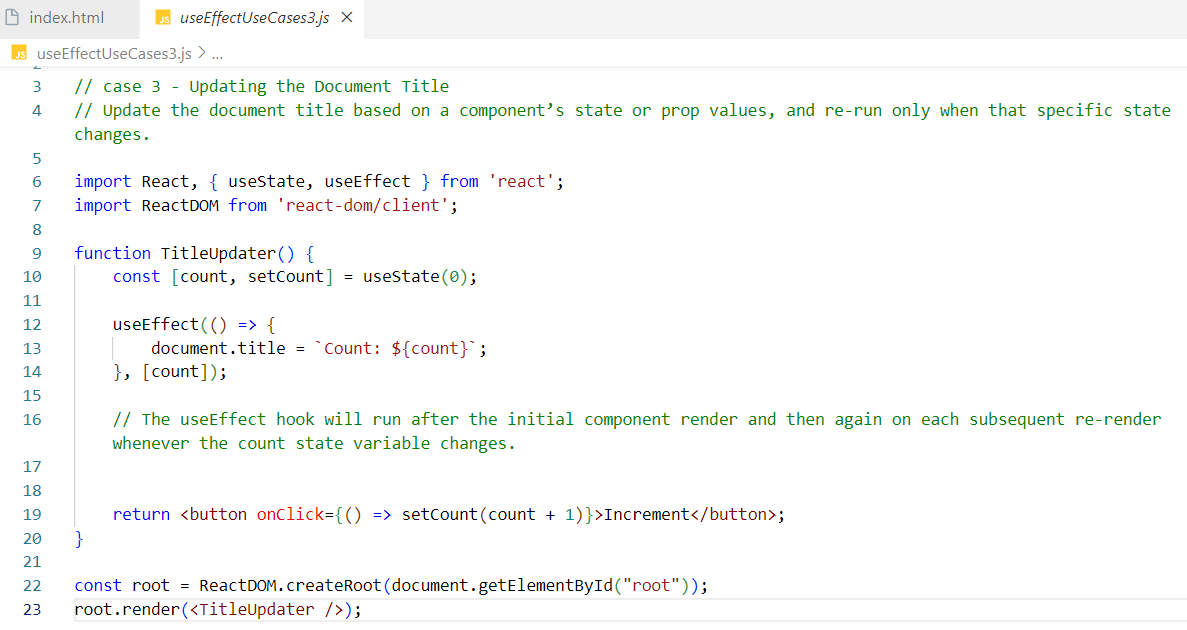
1.Data Fetching: Use useEffect to fetch data from an API when the component mounts. This is often done with an empty dependency array to ensure the data is fetched only once.



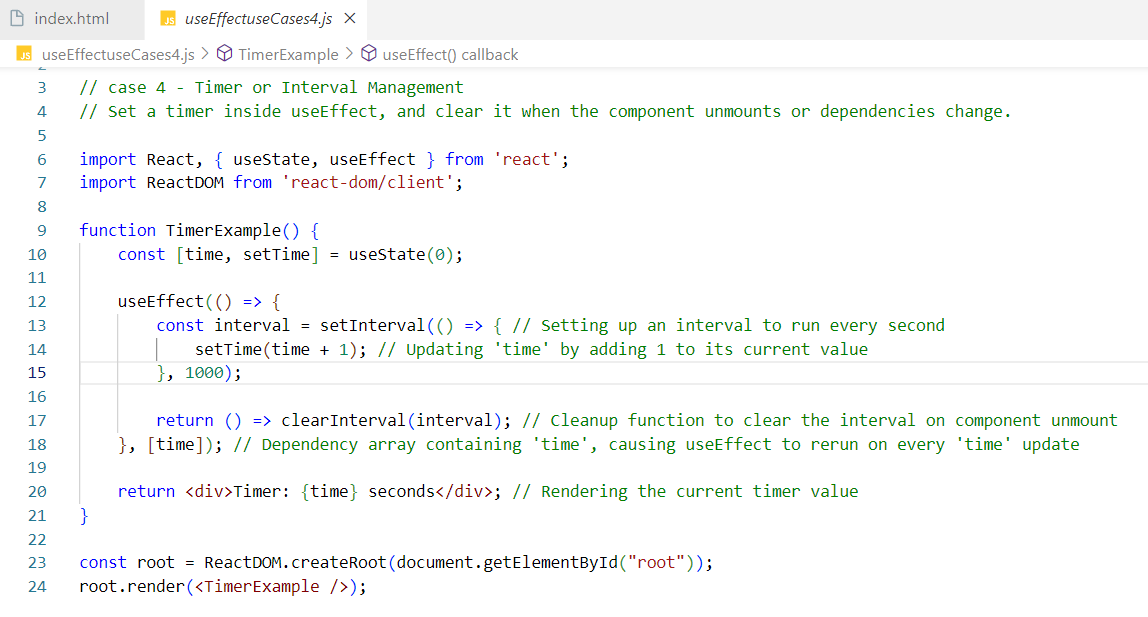
2.Subscribing to Events: Set up subscriptions to events (like WebSocket connections or event listeners) when the component mounts, and clean them up when it unmounts.



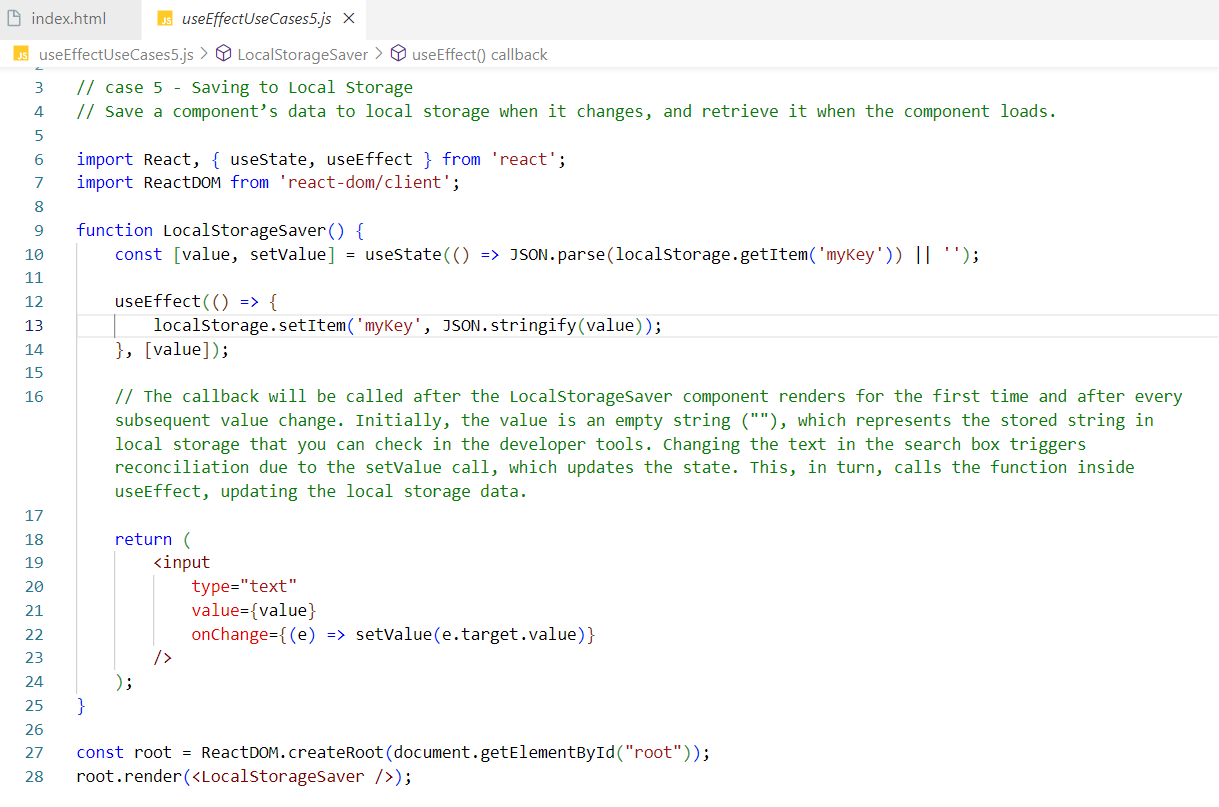
3.Updating the Document Title:Use useEffect to change the document title based on the component's state or props.



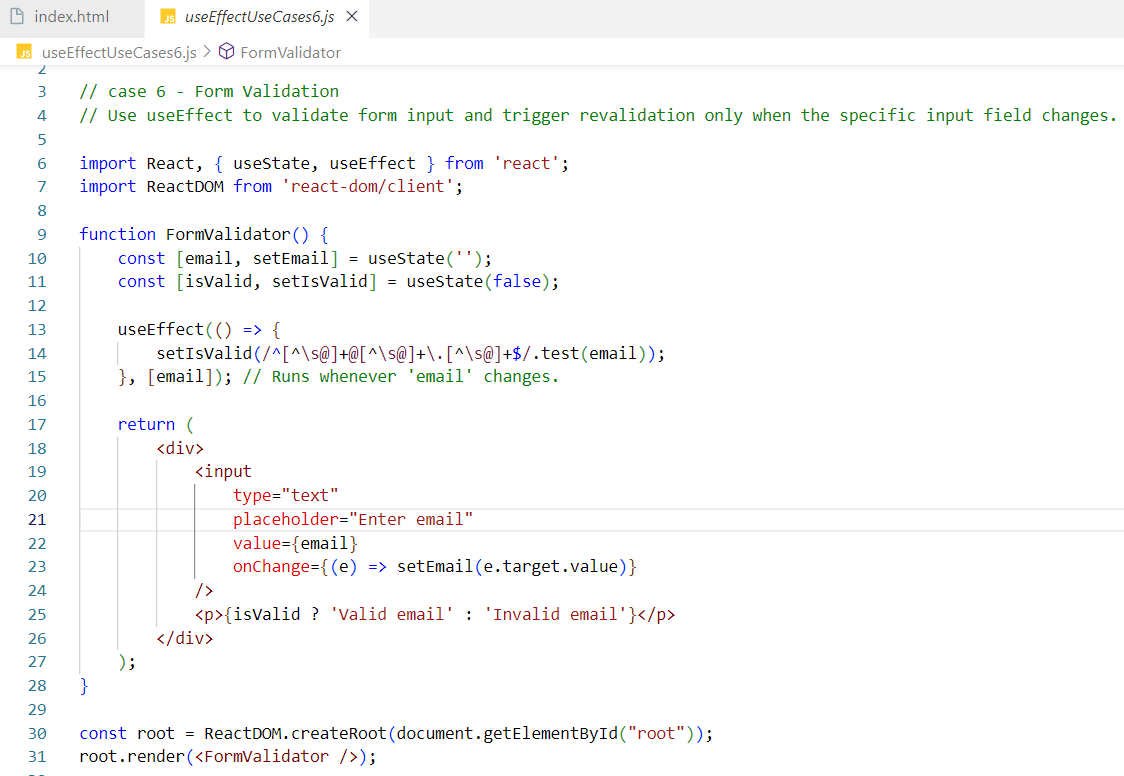
4.Timer or Interval: Manage timers or intervals by setting them up in useEffect and clearing them when the component unmounts or when dependencies change.



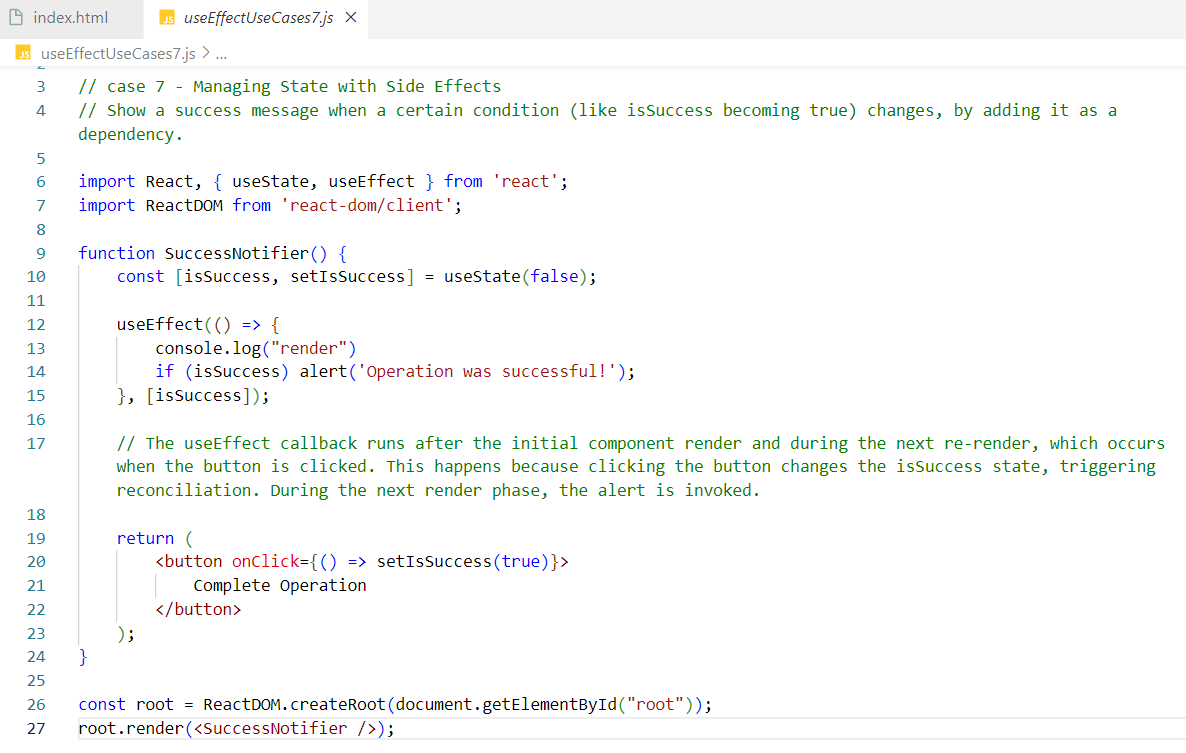
5. Local Storage: Save data to local storage whenever it changes and retrieve it when the component mounts.



6.Form Validation: Use useEffect to validate form inputs and update the component state based on the validation results.

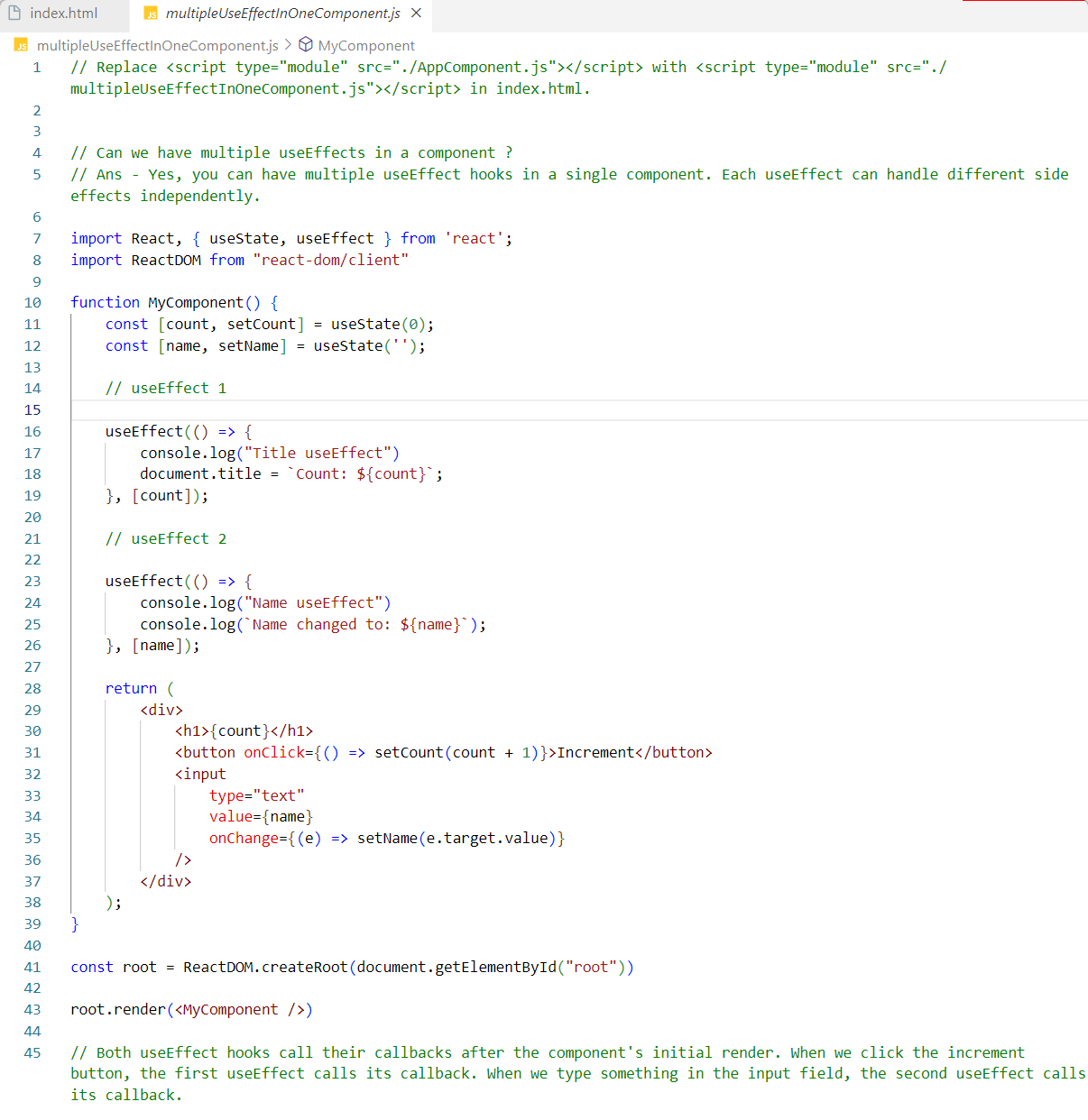


7.Managing State with Side Effects: Use useEffect to perform actions in response to state changes, such as resetting forms or displaying success messages.

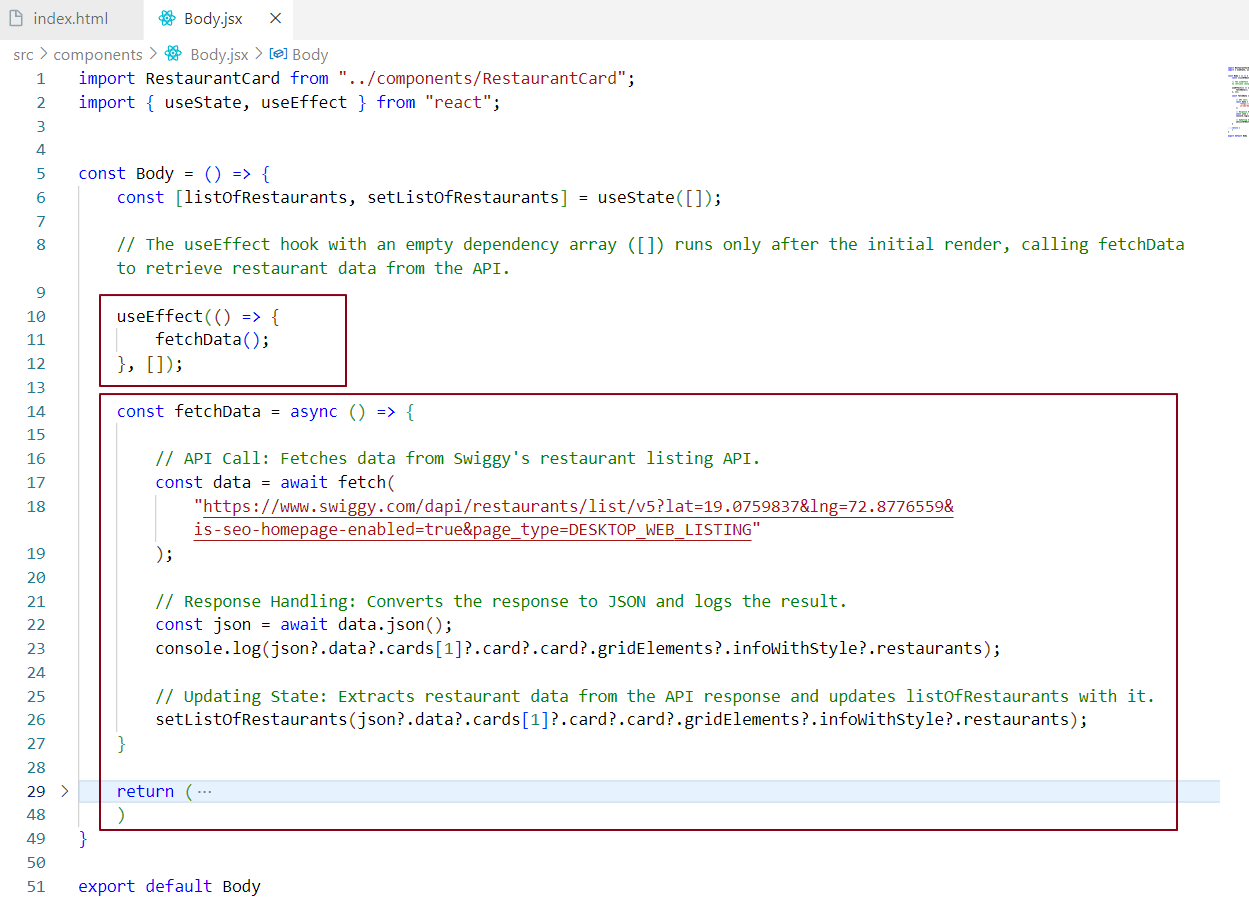


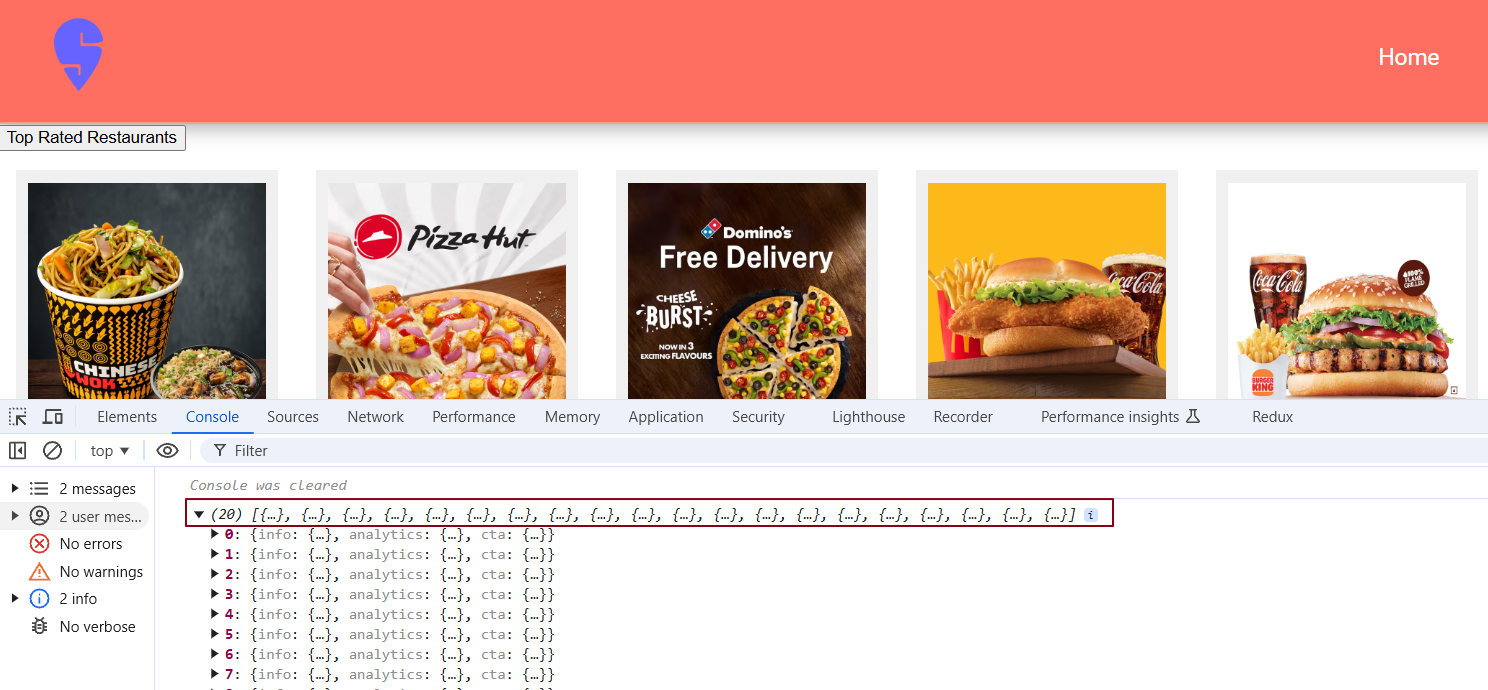
Can we use multiple useEffect in a component?

Yes, we can.



Let's make Swiggy API call we discussed earlier inside the useEffect hook.





**Potential Error:** When calling the Swiggy API, a CORS error may occur because the browser blocks requests from localhost due to CORS (Cross-Origin Resource Sharing) policy restrictions.

#### What is CORS?

CORS (Cross-Origin Resource Sharing) is a security feature enforced by browsers that restricts JavaScript code from accessing resources from a different origin (domain) than the one it was served from. This system uses HTTP headers to determine whether the browser should allow or block requests to cross-origin URLs. For example, calling Swiggy’s API from localhost is considered a cross-origin request, so the browser blocks it to maintain security.

Solutions for CORS Errors

1. **Using a CORS Browser Extension**: To bypass CORS restrictions for development purposes, you can use a CORS extension, which temporarily allows cross-origin requests by modifying browser behavior.
2. **Using a Proxy**: For a more robust solution that doesn’t rely on a browser extension, you can use a proxy server to handle the CORS issue. For example:

const data = await fetch (

"https://corsproxy.io/?https://www.swiggy.com/dapi/restaurants/list/v5?lat=19.0759837&lng=72.8776559&is-seo-homepage-enabled=true&page\_type=DESKTOP\_WEB\_LISTING"

);

This setup forwards your request through a proxy server, which bypasses the browser’s CORS restrictions.

When fetching restaurant list data from the API, there’s often a slight delay before the data loads. To enhance the user experience during this loading period, consider adding a visual loading indicator to let users know that data is being retrieved.

1. **Loading Spinner**: We could implement a loading spinner to appear while the data is being fetched. This provides immediate feedback to users that the application is working.

if (listOfRestaurants.length === 0) {

return <div>Loading...</div>;

}

1. **Shimmer UI**: For an even smoother experience, consider adding a “Shimmer UI,” which displays placeholder elements while the data loads. This approach offers a more polished look and helps set user expectations by showing an animated skeleton screen until the data is ready.

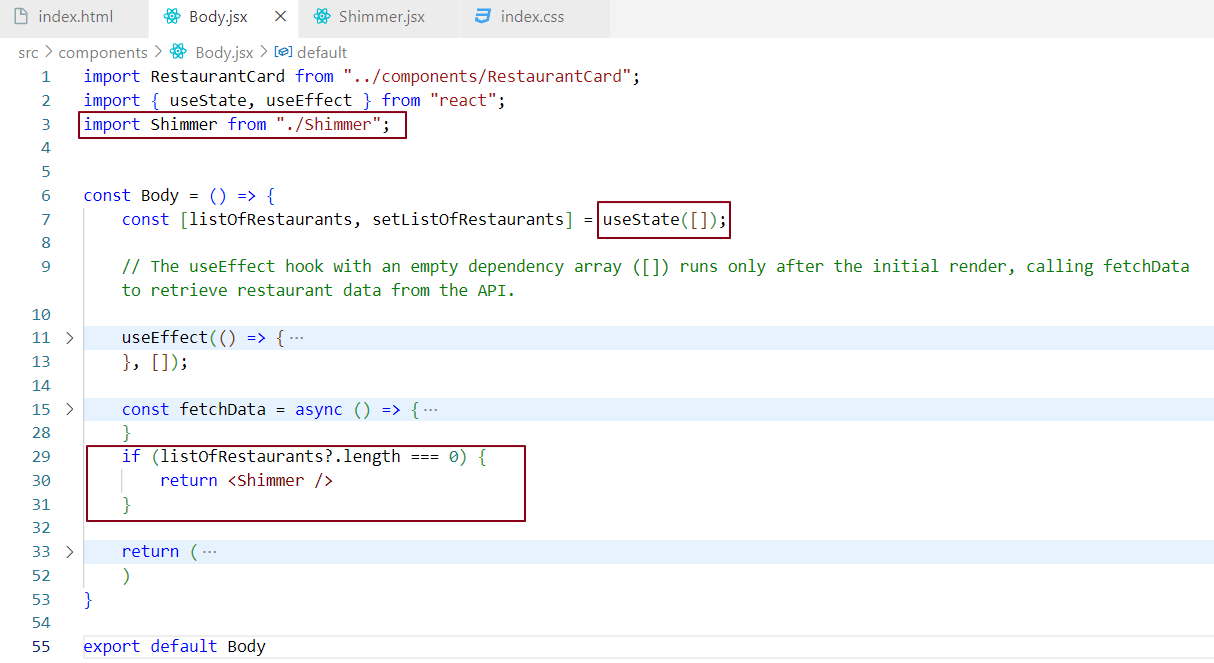
What is Shimmer UI?

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, giving the end user a more engaging UI.

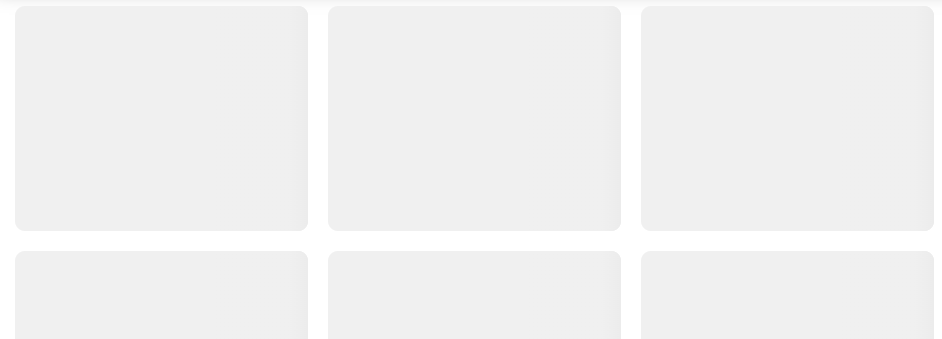
In the figure below, the end user can anticipate the appearance of cards that will load within the Shimmer UI. Once the data is fetched from the API, the placeholders are replaced with actual card content.



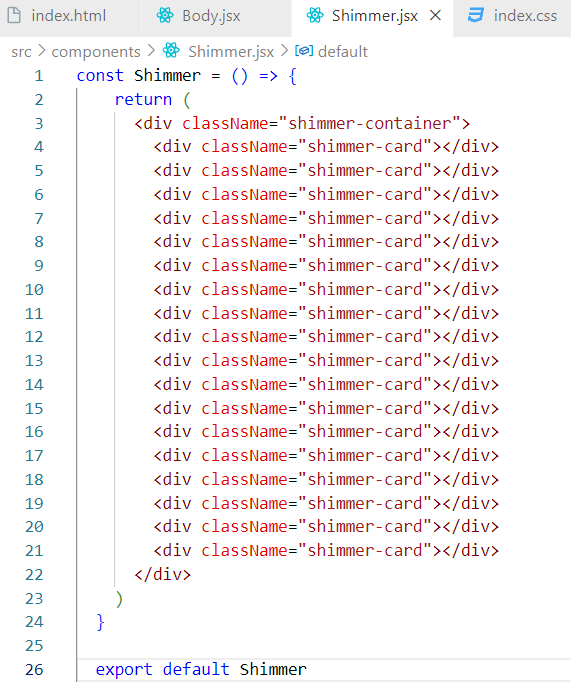
Let's integrate Shimmer UI into our application.



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



**Shimmer UI Component -**



We can ideally use map () to refactor this code, but let's keep 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 is available or 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 returned and rendered in the UI; otherwise, Component2 is returned and rendered in the UI.



In the example above, we display a shimmer effect in the UI when the restaurant list data is not available. As we know, useEffect is called after a component renders and re-renders. In our case, useEffect will be executed only once because we provided an empty dependency array.

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

Once the component's initial render is complete, useEffect is triggered, invoking the callback function that 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, when the restaurants state changes, the Body component re-renders. This time, according to our conditional rendering logic, the restaurants state variable contains data, allowing the list of restaurants to be displayed in the UI.

Q) Why do we need State variable?

To understand this, let’s introduce a feature in our app: a dynamic **' Login/Logout '**

button inside the Header component.

.



**Step-by-Step Explanation**

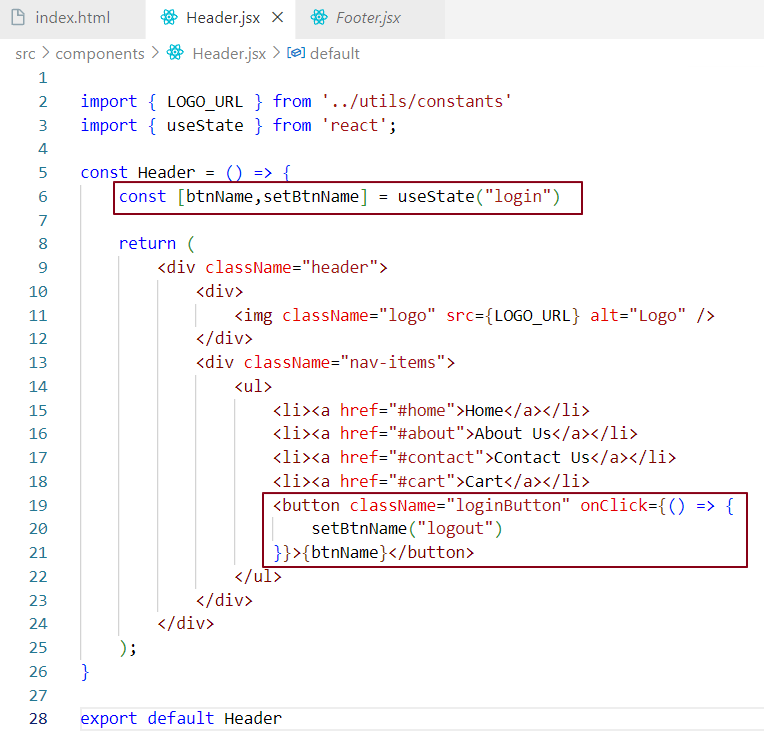
1. **Step 1** - We create a variable, btnName, with an initial value of "Login" and use it as the button text.
2. **Step 2** - We want the button text to change to "Logout" upon clicking. However, despite updating btnName and seeing the change logged in the console, the button text in the UI remains unchanged.

**\* Why doesn’t it change?** The issue here 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.

1. **Solution: Using useState ()** - To ensure that 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, automatically triggering a re-render each time it’s updated.

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

**Fixing the Issue with useState: Dynamic Button Text in React –**



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**

1. **Create a State Variable with useState**
   * Instead of a regular variable, we defined btnName as a state variable using useState, 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 that the UI reflects the latest value.

1. **Update State on Button Click**
   * We added an onClick event to the button, which calls setBtnName("logout") to change the value of btnName to "logout".
   * Syntax:

<button onClick= {() => setBtnName("logout")}>{btnName}</button>

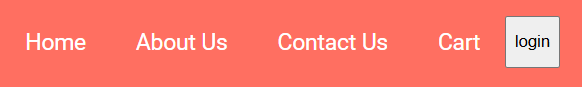
* + **Why**: Using setBtnName directly updates the state, triggering a re-render and updating the button text.

1. **Automatic UI Update**
   * Thanks to useState, the button text automatically updates on the UI to match the current state ("login" or "logout"), without requiring any manual re-rendering logic.

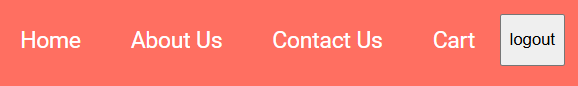
Building a Login/Logout Toggle Feature



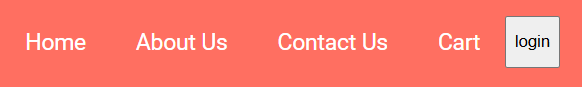
The code toggles btnName between "login" and "logout" by checking its current value with a ternary operator; if btnName is "login", it sets it to "logout", and vice versa, automatically re-rendering the UI with the updated state.



When user clicks on login button, button text changes from login to logout



When user clicks on login button, button text changes from logout to login



We have successfully implemented the toggle button functionality.

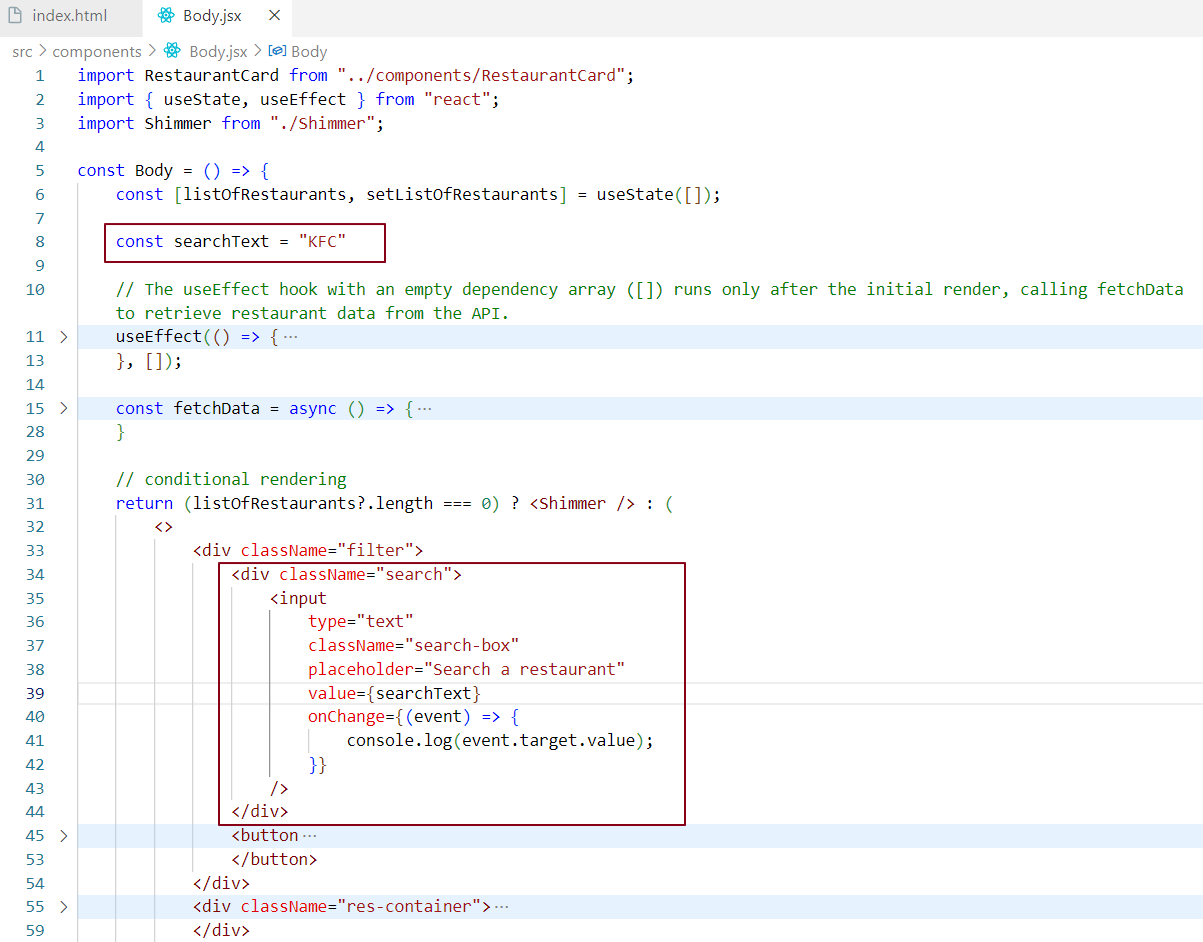
The interesting aspect of this example is how we’re able to modify a const variable like btnName

, which typically isn't possible. However, since React re-renders the entire component whenever a state variable changes, it effectively creates a new instance of btnName

with the updated value. So, rather than directly updating btnName

, React generates a new one with the modified value on each state change. This is the elegance of React’s design.

Building a Search Functionality -



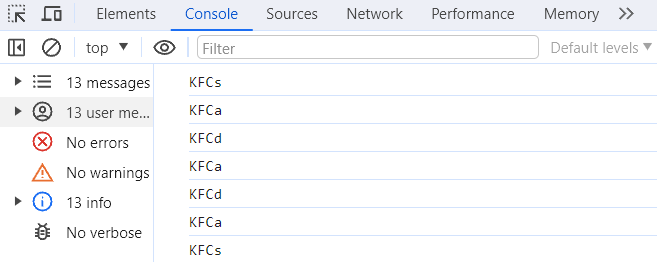
We added a search bar and created a local JavaScript variable called searchText inside the Body component. We set this variable as the value of the search bar, which is why "KFC" shows up in the search box.

**If we try to edit this value, nothing changes in the UI. Why is that?**

\* This happens because it depends on the direction of data flow. In our case, data flows in one direction—from the local variable to the search textbox value. We're simply displaying the data from the local variable in the UI. To see updates, we need to read data from the textbox and update the local variable, creating a backward flow of data. It may sound complex, but it’s straightforward in practice.

\* Another reason is due to the way React renders components. When a change event occurs—like typing in the search bar - React quickly re-renders the Body component. During each render, the value "KFC" is reassigned to the searchText variable, which then updates the search box value. The bottom line is, no matter how many times we type in the search box, React’s render cycle resets it to "KFC" each time.

We’ve written the code so that each time the text changes, the value is logged in the console.



This is not what we want; we want to see the updated text instead of a hard-coded value.

To achieve this functionality, regular local JavaScript variables are insufficient. We need to maintain the state of the variable 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 will not be aware of the change. Even if the variable is used in multiple places, react won't know why the local variable was modified, and it won't display the updated value in the UI because these variables are stateless. They do not maintain their state, and as we already know, react only tracks state variables.

**Note:** If we want our variables to stay in sync with the UI state, we need to use state variables. This enables React to track them by maintaining their state. With state, we can achieve two-way data binding in React.

Data Binding in React

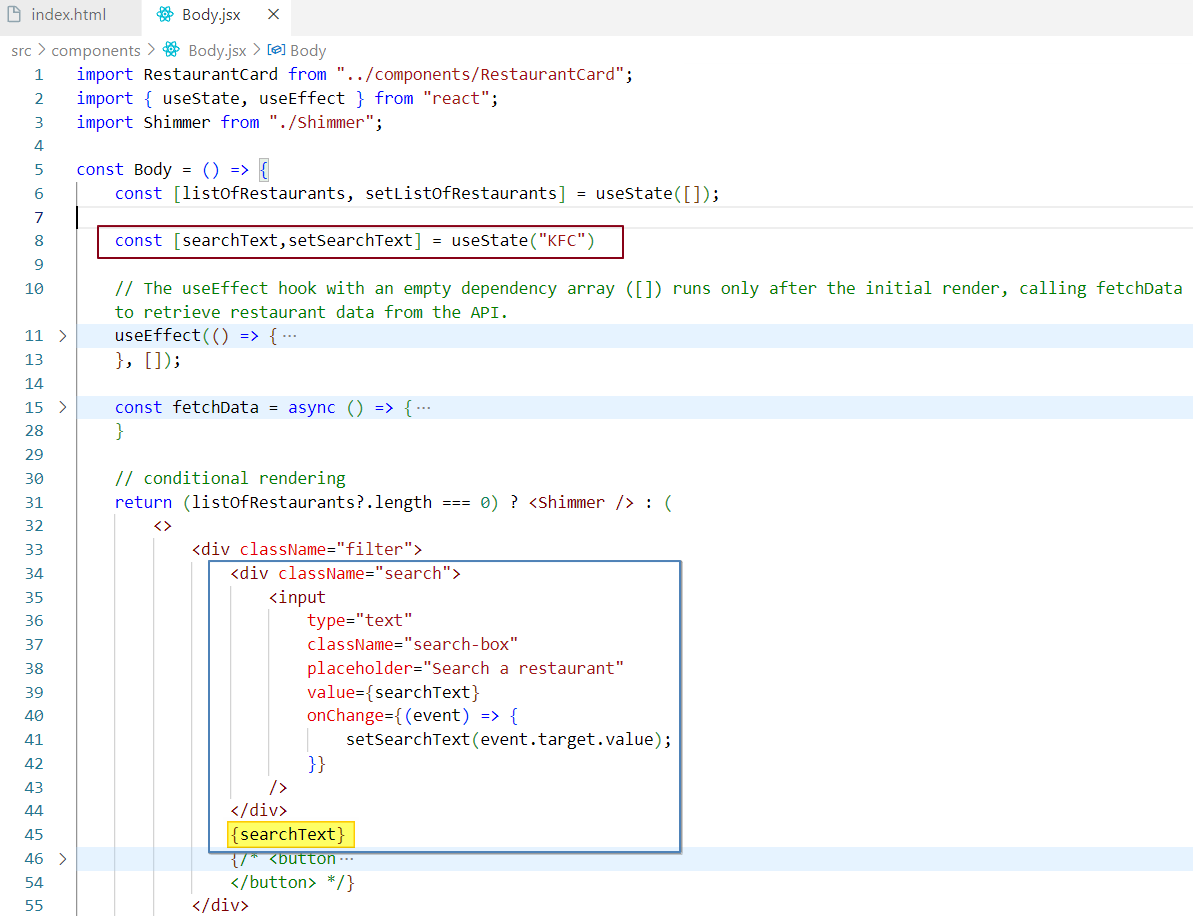
In React, there are two types of data binding:

1. **One-Way Data Binding**: Data flows in one direction.
2. **Two-Way Data Binding**: Data flows in both directions.

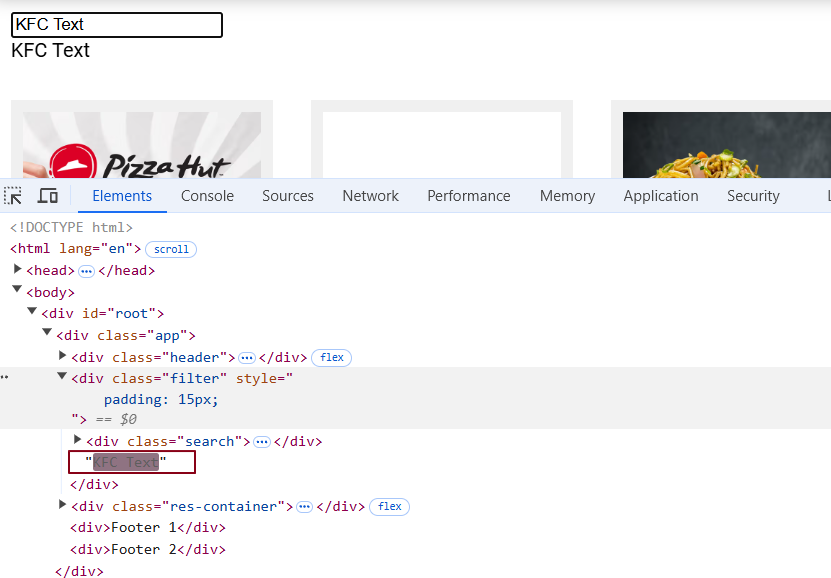
In our code example, we simply placed a local JavaScript variable into the textbox, which writes the variable's data into the textbox. This demonstrates React’s one-way data binding. However, our goal is to edit the textbox value, meaning our local variable should be updated when the textbox value changes. In other words, we need to read data from the textbox and update our local variable simultaneously.

This concept of reading and writing data simultaneously is called two-way data binding.

Up to this point, we have observed one-way data binding in action. Now, let’s introduce two-way data binding, which we can implement using a hook called useState.



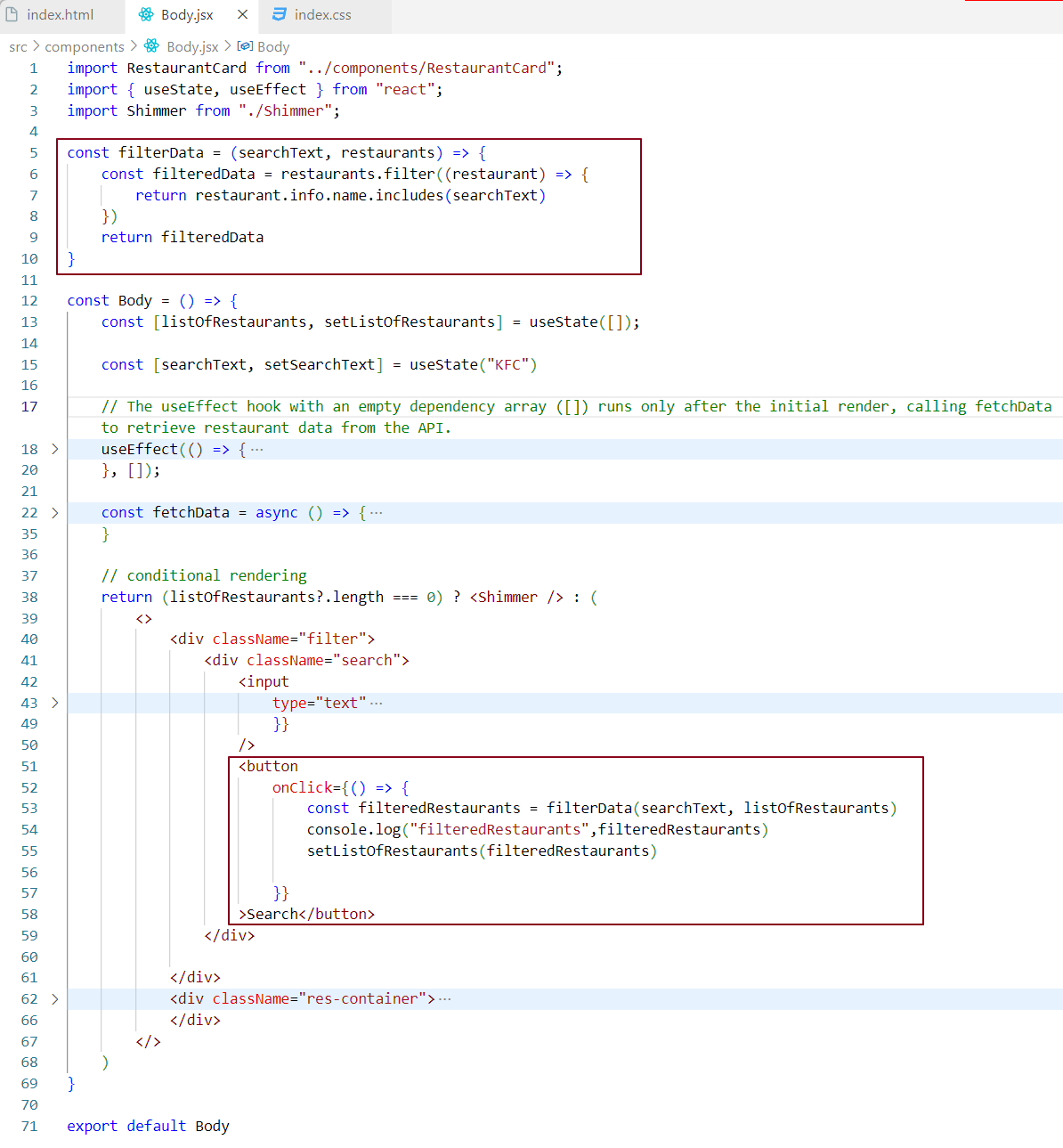
I have placed the React variable right below the search input text. The setSearchText function, provided by the useState hook, is used to update the React variable searchText. React watches and tracks this searchText.



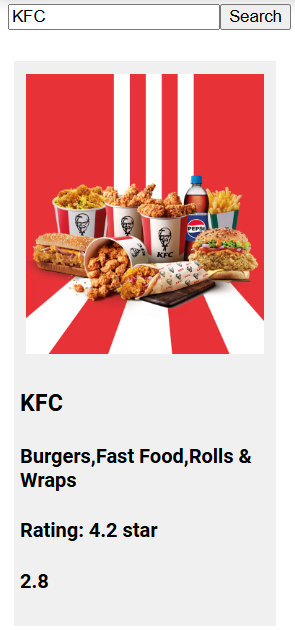
Whenever searchText is updated or changed, react re-renders the entire Body component and only updates the changed portion in the DOM (as shown in the figure above). This process involves reconciliation, which happens behind the scenes, and the diffing algorithm handles all the heavy lifting.

**Note**: When we say React re-renders the Body component, it means React effectively destroys the current instance of the Body component and creates a new one very quickly. Now, if I type anything in the search textbox in the UI, the searchText is updated, which we can see visually. This demonstrates two-way data binding in action.

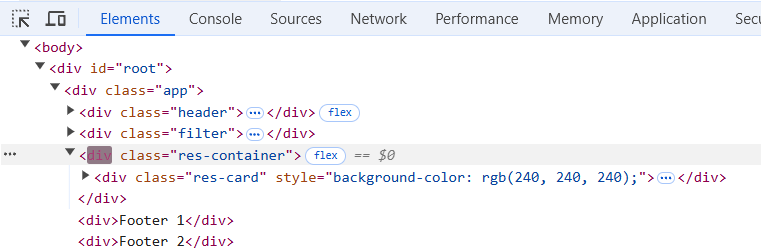
Let’s add a button next to the search bar and implement filter logic inside it.



When the search button is clicked, it filters the list of restaurants based on the search text entered in the input field and returns the filtered restaurant list. This updates the state variable listOfRestaurants, which is then rendered in the UI.



When the search button is clicked, the state variable listOfRestaurants is updated, triggering reconciliation. This invokes the diffing algorithm to compare the new virtual DOM with the old virtual DOM. Once it identifies the changes, it injects the updated portion into the actual DOM. You can visually see this, as shown in the figure below. The highlighted div represents the updated filtered restaurant that will be 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.

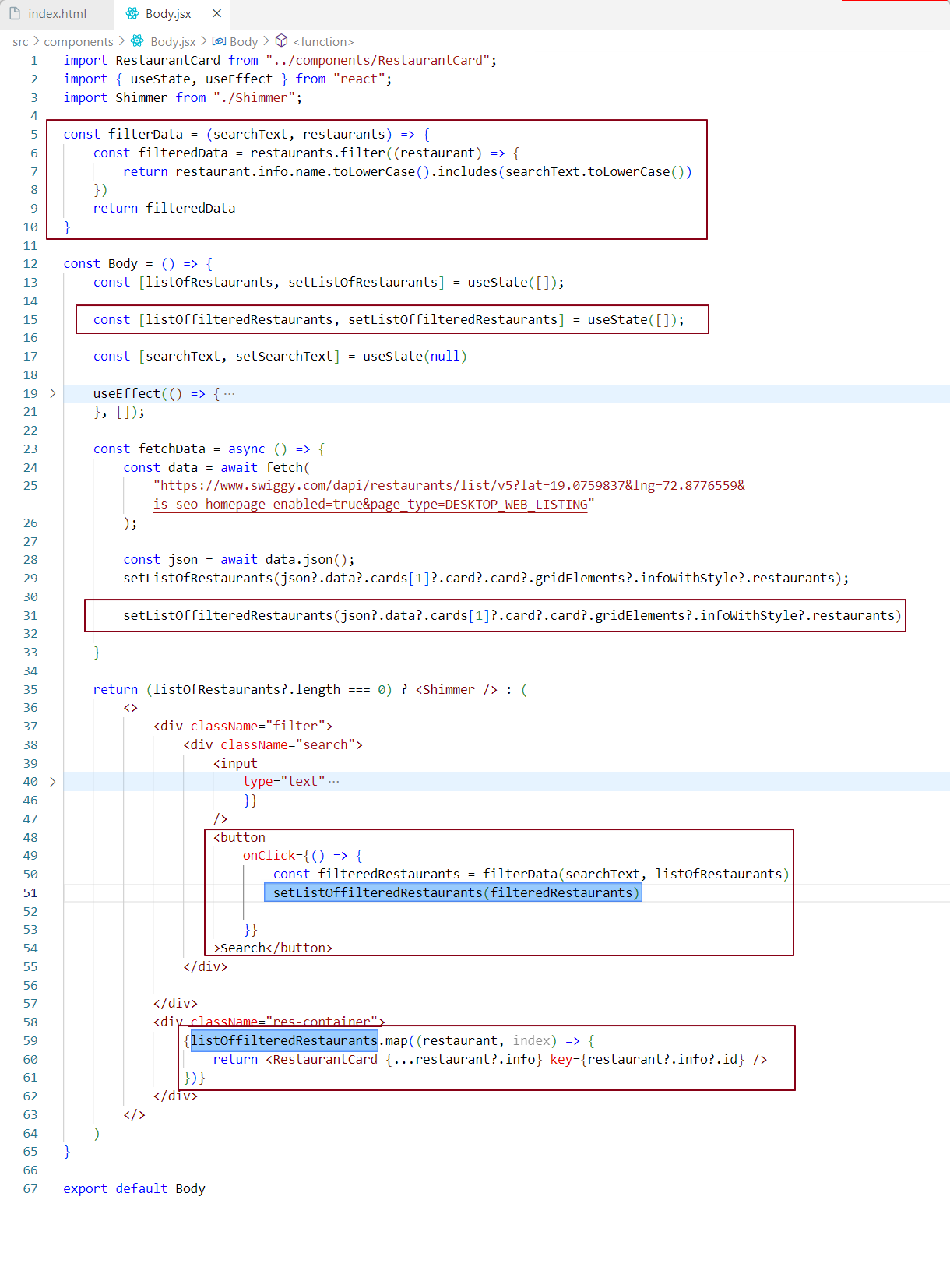
However, there is one issue with this approach: the search functionality will only work once. If we search for a restaurant a second time, it will not function as expected.

Why is the search behaving in a way that it shouldn't?

When we type some text in the textbox and click the search button, the filterData function is invoked, returning a list of filtered restaurant data. This filtered data updates the listOfRestaurants state variable using setListOfRestaurants(filteredData). As a result of this state change, the Body component re-renders, displaying only the filtered restaurant card components in the UI.

At this point, the listOfRestaurants state variable contains only the filtered data. Therefore, during the next search, the input text is compared against this stored filtered data. If the searched restaurant isn't found in the filtered list, we get a blank page, indicating that no matches were found.

To resolve this issue, the search operation should be performed on the complete list of all restaurants, rather than just the filtered ones. To achieve this, we need two state variables: one for the complete list of restaurants and another for the list of filtered restaurants. This approach allows us to maintain access to both the full restaurant list and the filtered list as needed. Consequently, we will create two state variables inside the Body component.



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

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

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

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

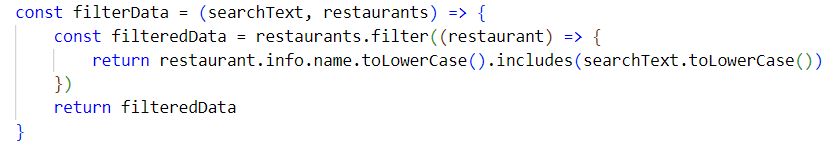
When the API is first called, both state variables are populated with real-time data, initially starting empty.

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.

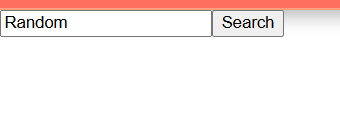
**Updated Filter Logic:**

The search functionality is now case-insensitive. Previously, the search was case-sensitive, so the search text's case mattered. With the updated logic, case-sensitive searching is no longer required.



Now, the search function works as expected.

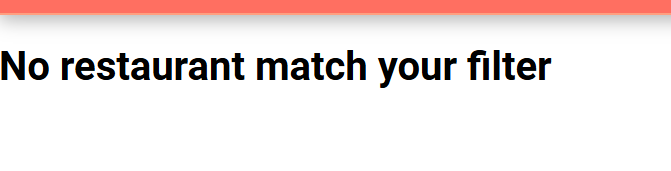
However, we have encountered another issue in our app. If no matching data is found, the UI displays a blank area instead of a helpful message.



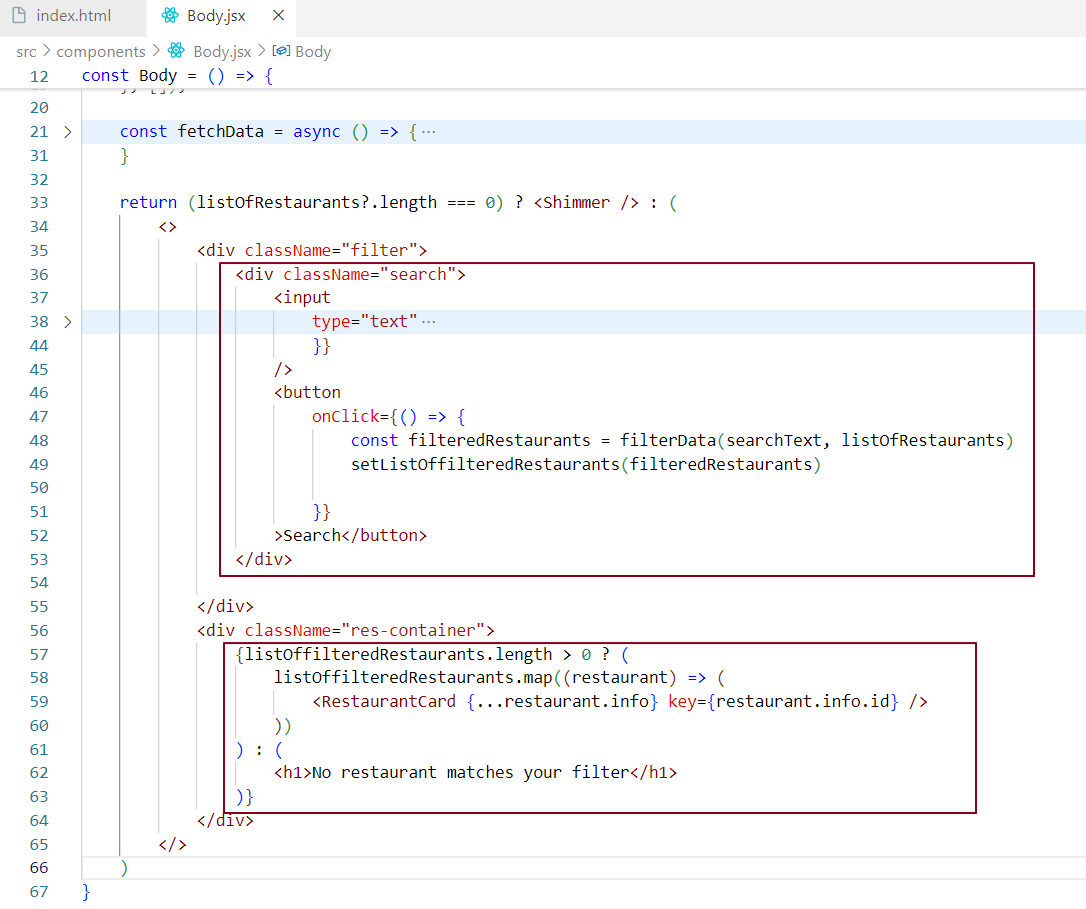
Let’s address this issue.

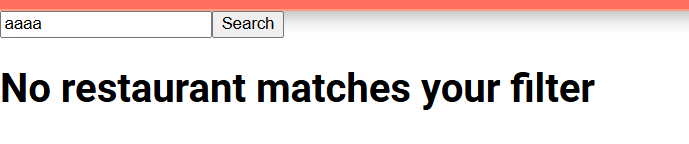


Now, for an invalid search, we receive a notification saying, "No restaurant matches your filter." However, this introduced another issue: our search bar disappears.

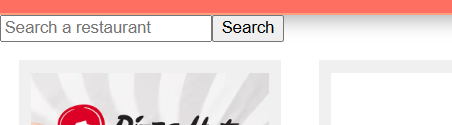


Let’s address this issue by moving the conditional logic for listOfFilteredRestaurants inside the return statement and using conditional rendering. This way, we can ensure that the search bar remains visible even when no restaurants match the filter.



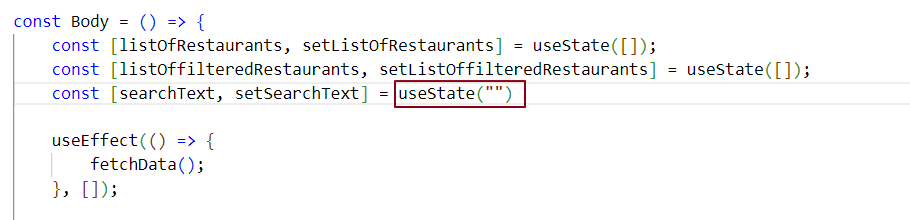


When we first load the page, the searchText is initialized to null. When we click the search button, it results in an error





This error occurs because restaurant?.info?.name can be null, and we are attempting to call toLowerCase () on null. To resolve this issue, we should initialize searchText with an empty string ("") instead of null.



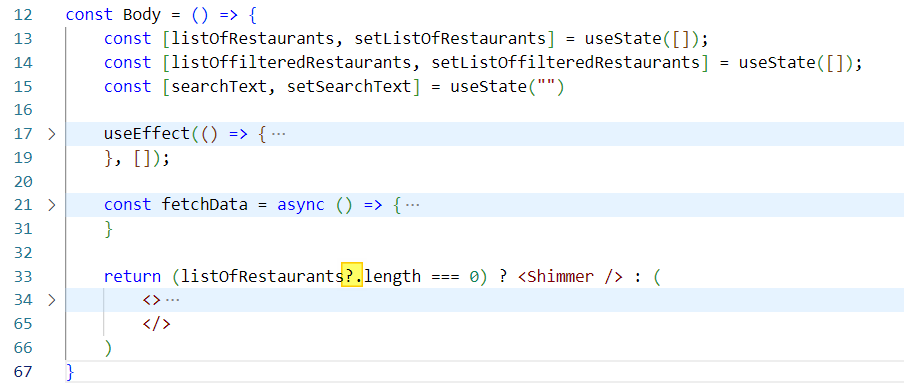
This will resolve the issue. Now let’s understand a concept Early Return in react.

What is Early Return? Why do we use it?

**Early return** refers to exiting a component before it finishes rendering. We use early returns to prevent rendering a component when the necessary data is unavailable. If we try to render a component based on a condition when the required data is missing, the condition may fail, leading to unexpected errors in the UI.

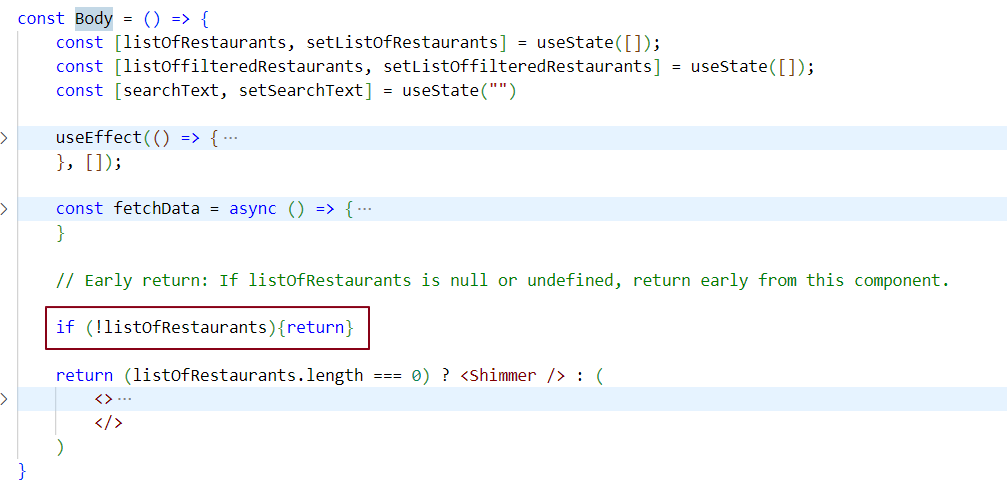
There are two ways to avoid this error

Way 1: Optional chaining



Way 2: Early return

We prevent rendering the Shimmer component until listOfRestaurants is defined. We use early return to achieve this. In our case, before rendering the Shimmer component, we perform an early return to ensure that if listOfRestaurants is not available or is undefined, we return null or a valid piece of JSX. This way, we can stop the Shimmer component from being rendered prematurely on the UI.



JavaScript Expressions vs. Statements

**JavaScript Expression:**

An expression is any valid unit of code that resolves to a value. It can be a combination of values, variables, operators, and functions that produces a result.

**Examples of Expressions:**

1. **Arithmetic Expression:**

1 + 2 // evaluates to 3

1. **String Method:**

"foo”. toUpperCase () // evaluates to 'FOO'

1. **Logical Expression:**

isTrue? true: false // evaluates to true or false based on the value of isTrue

1. **Function Call:**

console.log (2) // logs '2' to the console

**JavaScript Statement:**

A statement is a command that performs an action but does not return a value. Statements control the flow of execution.

**Examples of Statements:**

1. **Variable Declaration:**

let x; // declares a variable x

1. **Conditional Statement:**

if (condition) {

// some code here

} // checks a condition and executes code based on it

1. **Function Declaration:**

function myFunction () {

// some code here

} // defines a function

Using Expressions and Statements in JSX

**Using Expressions in JSX:**

* To use a JavaScript expression in JSX, wrap it in curly braces {}. For example:

jsx

Copy code

<h1> {1 + 2} </h1> // This will render: <h1>3</h1>

<h1>{"foo".toUpperCase()} </h1> // This will render: <h1>FOO</h1>

**Using Statements in JSX:**

* You **cannot** directly use JavaScript statements in JSX. JSX is designed to work with expressions that return values. If you try to use a statement directly in JSX, it will not work as expected.

**Recommended Approach:**

* If you need to perform logic or control flow that results in an expression for JSX, do it **outside** of the JSX code or inside a function:
  + For example, you can use a variable or a function to return an expression:

const greeting = "Hello, World!";

return (

<div>

<h1>{greeting}</h1> // Using a variable as an expression

</div>

);

**For conditional rendering:**

const isLoggedIn = true;

return (

<div>

{isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please log in.</h1>} // Using a conditional expression

</div>

);

Important Points

* **DOM Updates**: DOM updates occur through the React DOM library.
* **Diffing Algorithm**: The diffing algorithm is implemented within the React core library.
* Whenever there is a change in state, react rerenders the components that utilize that state.
* Whenever there is a change in props, react rerenders the components that receive those props.