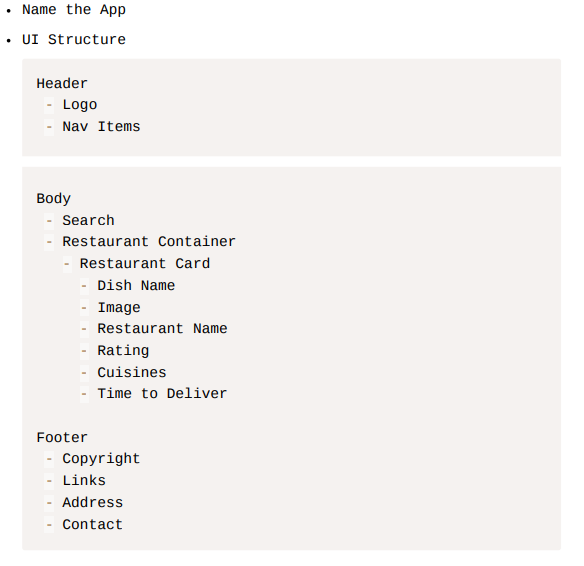
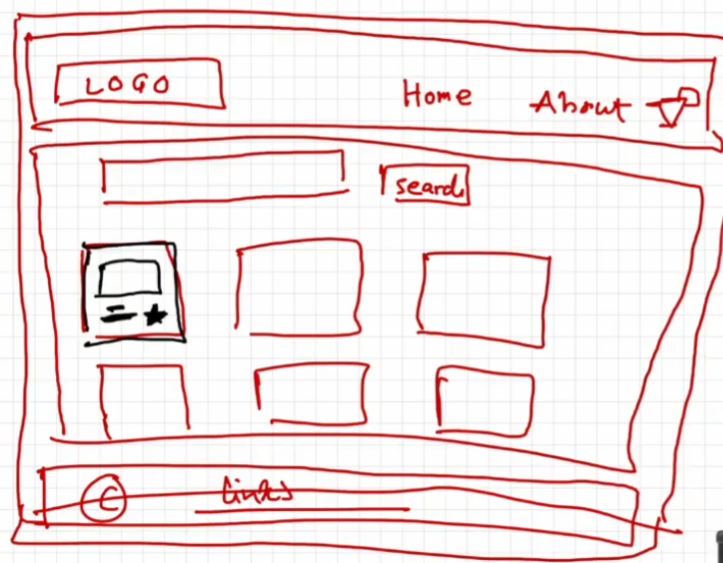
Chapter - 4: Talk is Cheap Show me the code

In this episode, we’ll start coding by creating a new project - a Food Ordering App.

### Planning the UI

Before we dive into coding, it’s important to plan everything out. Proper planning will make the development process smoother and easier to understand. We should have a clear idea of what we’re building before we write any code.

Low level design -

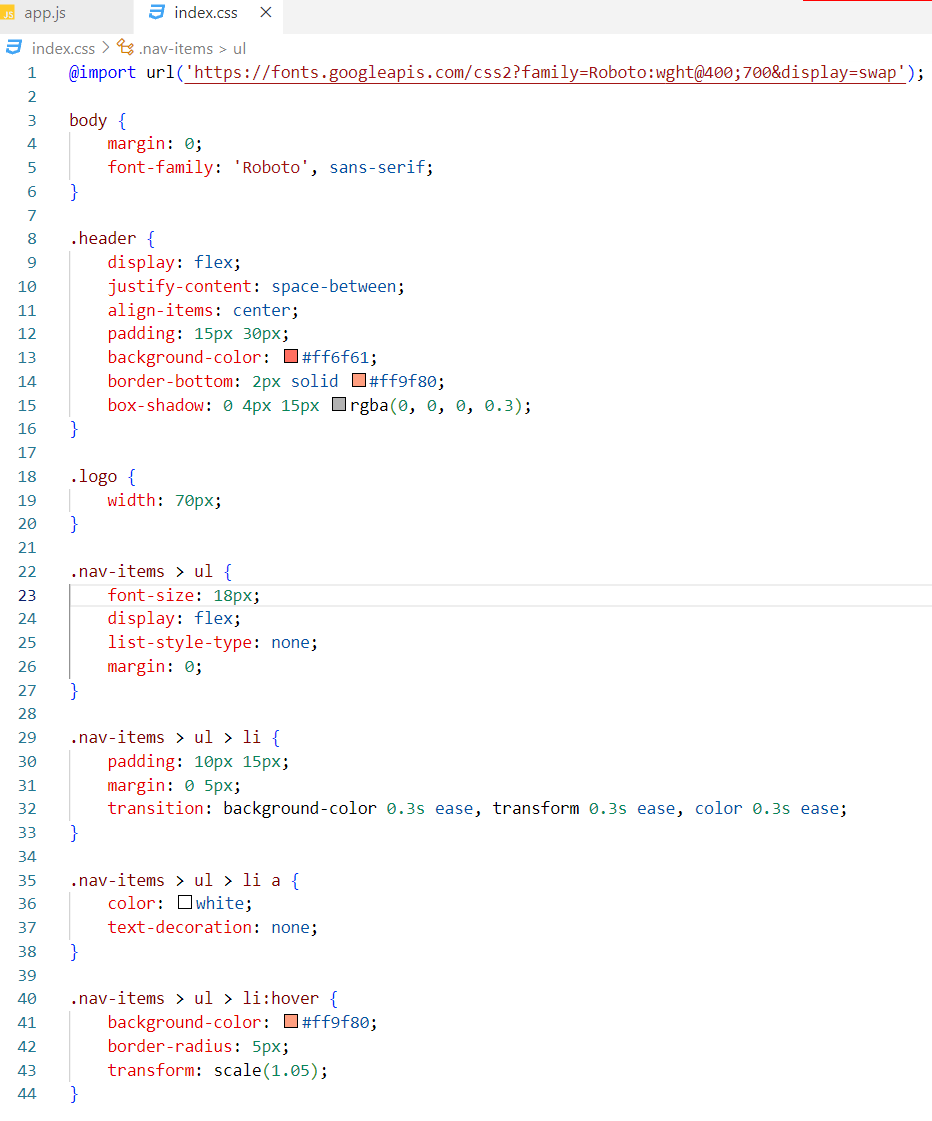


Awesome! Let’s dive in and start coding! 🚀

Implementing the Header Component in React -

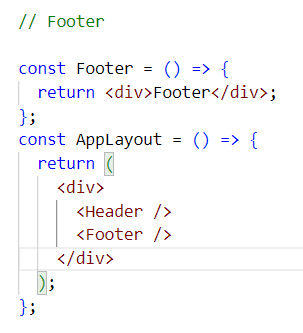


Designing the Header Component –





Implementing the Footer Component in React –



Note - JSX expressions must have one parent element.

In the AppLayout component, there are two JSX expressions - one for the Header component and another for the Footer component. These expressions are currently wrapped inside a <div> element, which introduces an extra node into the DOM tree when rendered.

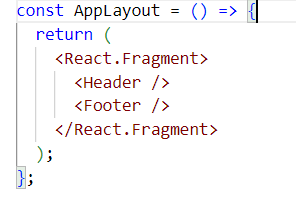
To avoid adding this unnecessary <div>, we can use React Fragments. Fragments allow us to group multiple elements without adding an extra DOM node, keeping the DOM cleaner and more efficient.



What is React Fragment?

React.Fragment is a special component provided by the React library. It lets you group multiple child elements without inserting an extra <div> or wrapper element into the DOM.

In our case, we use React.Fragment to wrap the Header and Footer components. This allows them to be rendered side by side without introducing any additional nodes into the DOM tree - keeping the structure clean and lightweight.



Additionally, you can use the shorthand syntax <>...</> instead of writing React.Fragment. However, it's important to note that **you cannot apply any props or styling** to this shorthand version - it’s purely for grouping elements.

Styling in React: Different Approaches

1. Inline Styling:

Inline styling in React can be done by passing a JavaScript object to the style attribute. The keys in the object represent CSS properties, and the values represent the corresponding styles. Note that the property names in the object are written in camelCase rather than hyphenated style (e.g., font-size becomes fontSize).



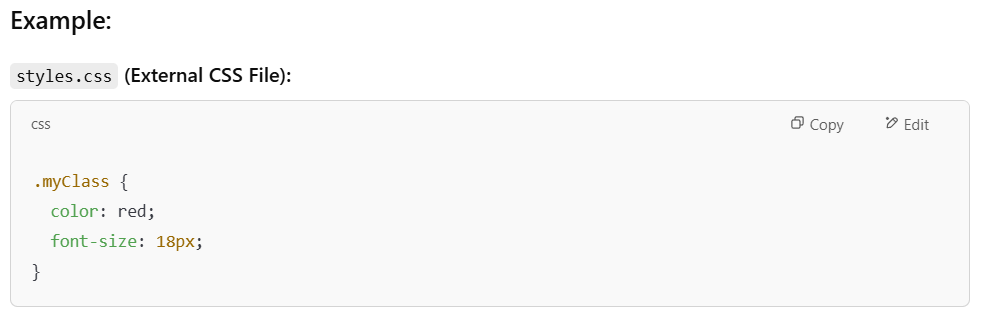
In this example:

* The color property is set to blue.
* The fontSize property is set to 20px.

This approach is useful when you want to apply styles directly within the JSX code without using external CSS files or stylesheets.

2. External CSS:

External CSS is a common way to apply styles to your React components. You can define your styles in a separate .css file and import that file into your component. Then, you apply the styles using the className attribute in JSX.





In this example:

* The styles.css file contains the .myClass class definition.
* The className="myClass" in the div element applies those styles.

Using external CSS is more maintainable for larger projects because it allows you to separate the structure (HTML/JSX) from the styling (CSS).

3.External Libraries (Tailwind, Bootstrap):

External libraries like Tailwind CSS and Bootstrap provide predefined classes that make styling faster and easier.

### *Example with* *****Tailwind CSS***** *-*

Tailwind CSS styles elements with utility classes, where each class controls one specific style, and combining them creates the final design.



In this example:

* text-blue-500 sets the text color to blue (a specific shade from Tailwind’s color palette).
* text-xl applies a larger font size.

4.Styled Components:

Styled Components is a CSS-in-JS library that allows you to write real CSS directly inside JavaScript files to style React components.



styled.div - styled is actually an object which has properties for every HTML tag like div, button, h1, etc. Each property is a function that creates a styled React component. styled.div is **not a normal function that takes a string in parentheses**. It is a **tagged template literal function**; it only works with backticks used **directly after** it.

StyledDiv - This is now your own custom component. It's like a normal <div>, but it already has styles built in.

Reusability - You can reuse StyledDiv anywhere in your app, and it will always look the same unless you change the styles.

This method works well with component-based architecture, as it keeps styles and logic in the same place, which aligns with modern React development practices.

### *What is Component-Based Architecture?*

* It’s a way of building applications where the **UI is divided into small, reusable pieces called components**.
* Each component is **self-contained**: it has its own logic, structure (HTML/JSX), and styling (CSS).
* These components can be combined like **building blocks** to create the full application.

What is Inline Styling in react?

Inline styling in React means applying styles directly to JSX elements using a JavaScript object.

You write the CSS properties inside an object using camelCase (instead of kebab-case), and then pass that object to the style attribute using double curly braces {}.



***Alternative version of the code -***

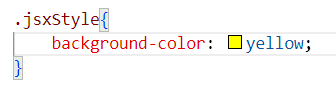


One pair of curly braces for JSX, and another one is for the object.



What is External Styling in react?

Assign a className to the JSX element and define the corresponding CSS rules in an external CSS file.



Can I use a React.Fragment inside another React.Fragment?

Yes, absolutely!

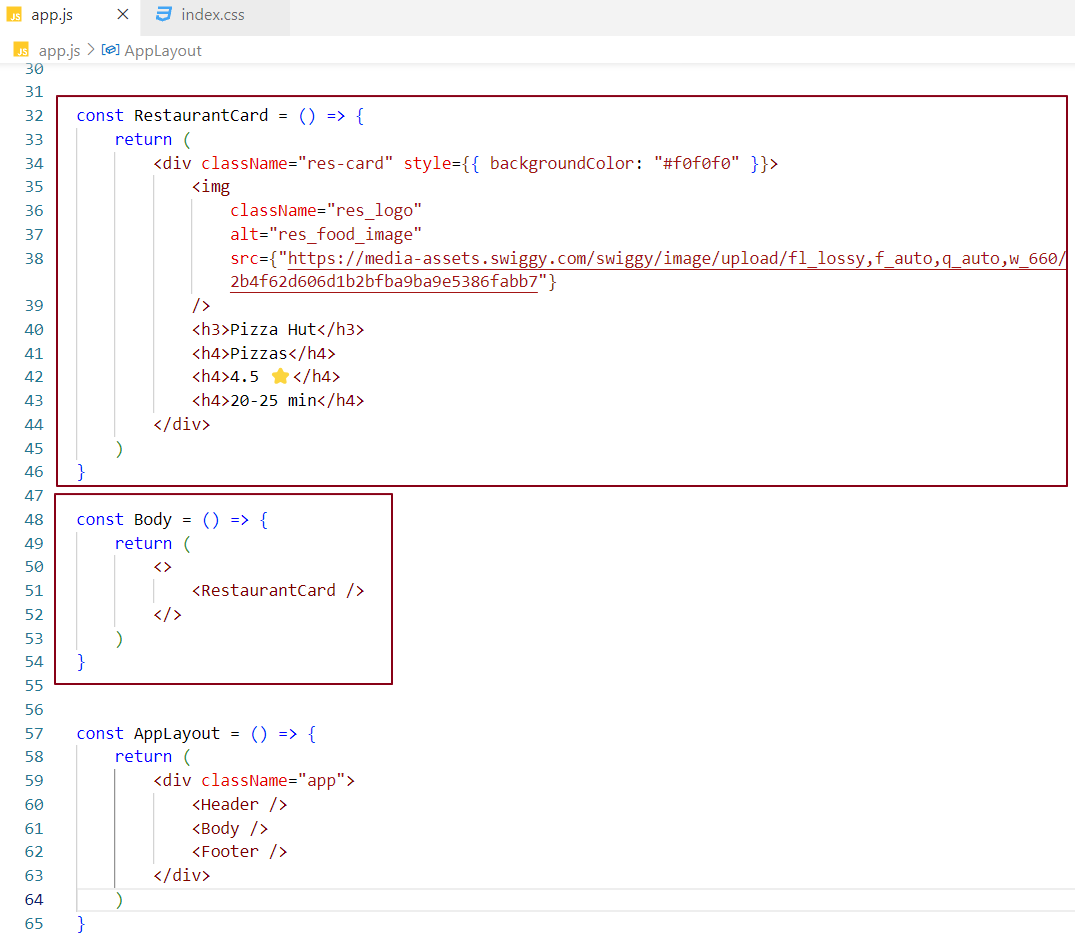


Empty Angular tags (like <>) are not rendered in the DOM; they are used purely to group child elements without adding extra nodes.

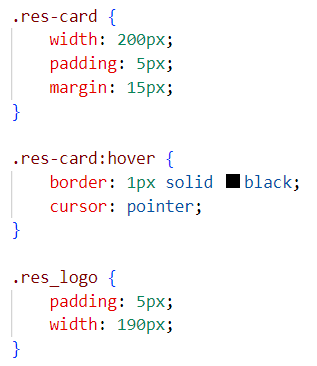
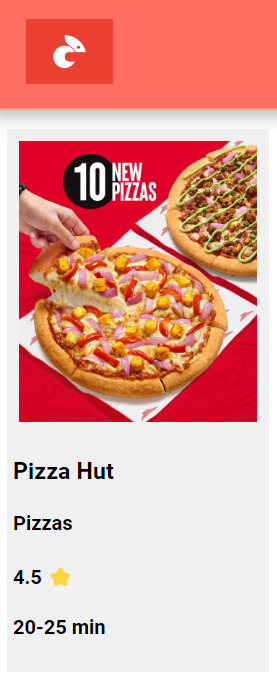


Implementing the Body Component in React -

To display a list of restaurant cards in the Body component, let's begin by creating a single Card component with hardcoded data.



Designing the Body Component -

As you know, JavaScript expressions can be used within JSX.

Let’s create a JavaScript object named pizzaHut and use it inside the RestaurantCard component.

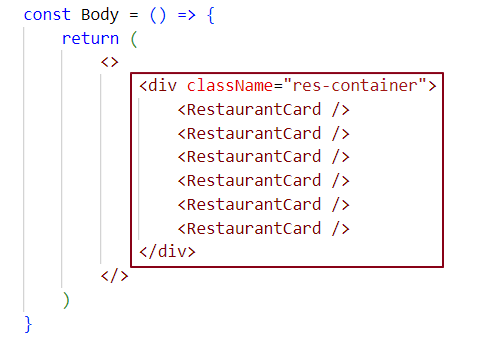
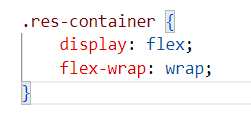
We can access the object’s properties using dot notation by wrapping them in curly braces within the JSX.

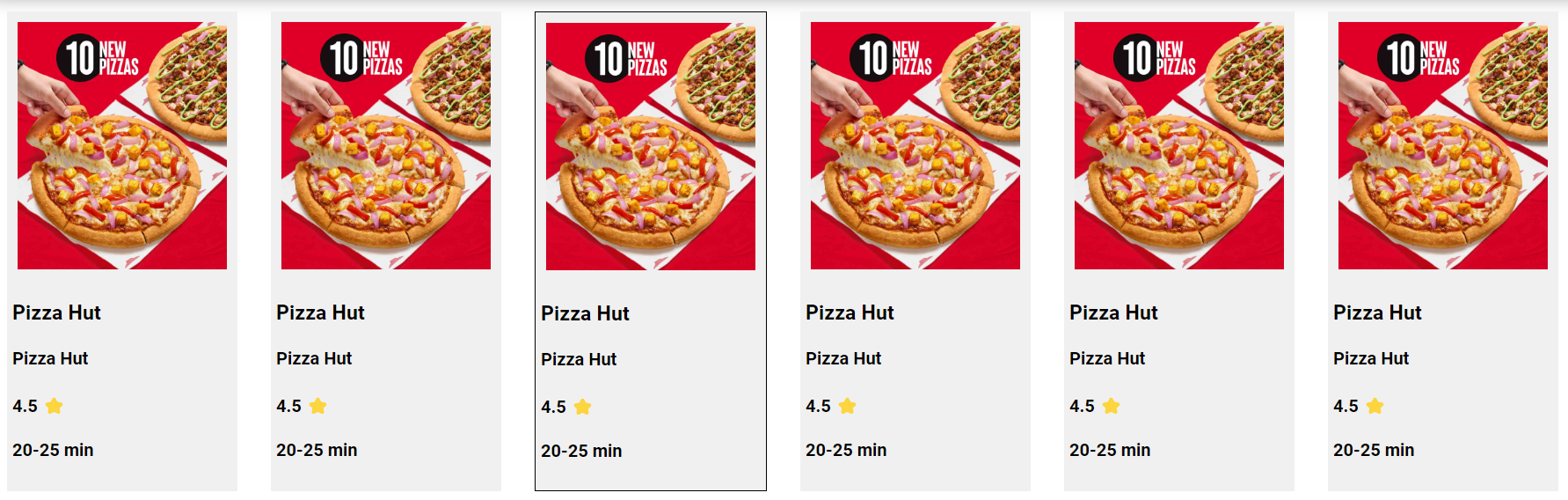


The app will continue to function as it did before, and the UI of the Body component will remain unchanged.

In a real-world application, we typically display a list of card components with different data.

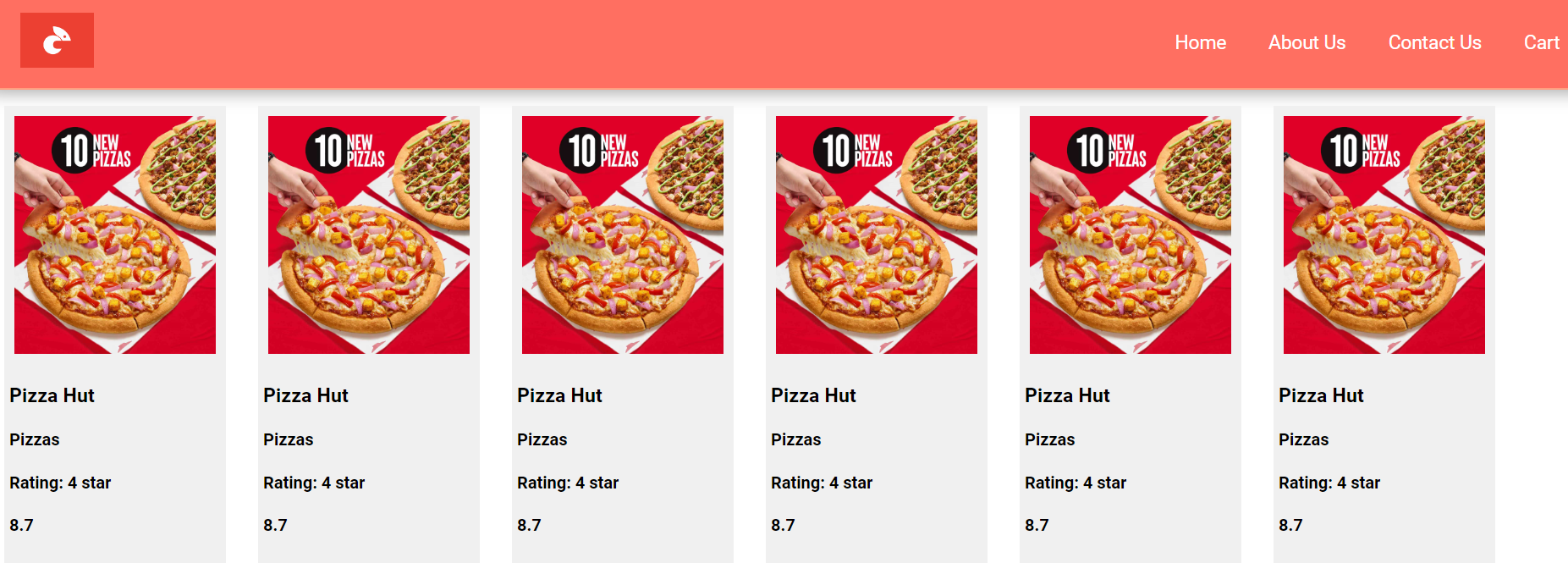
We can reuse the RestaurantCard component to render as many cards as needed. Let’s simulate this behavior.





Let’s fetch real-time restaurant data from the Swiggy API and display it dynamically using the RestaurantCard component in the UI.





Optional chaining (?.)

The optional chaining (?.) operator allows safe access to an object's properties or functions. If the object or function is undefined or null, the expression short-circuits and returns undefined, preventing errors.

So far, we’ve been able to duplicate card components inside the Body component. However, to display different data in each card, the data should be dynamic. To achieve this, we need to pass the data as props from the parent component (Body) to the child components (RestaurantCard). Each RestaurantCard will receive its specific data through props and display the relevant information accordingly.

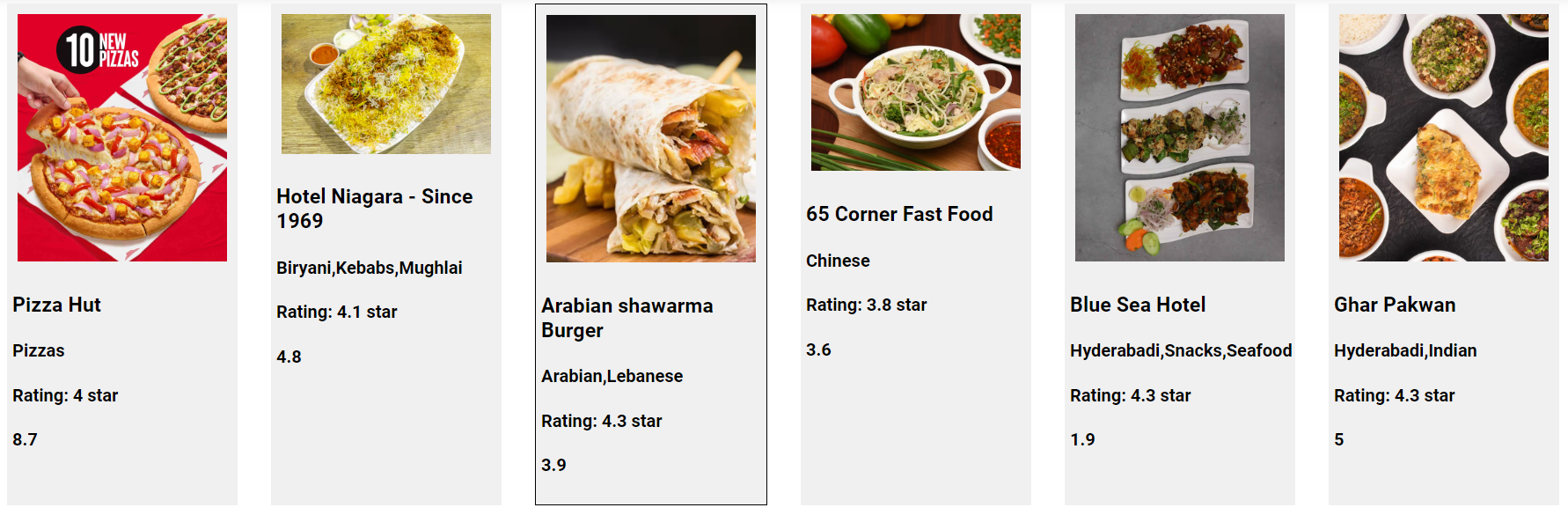
What are props?

Props (short for properties) are similar to passing arguments to a JavaScript function. When we pass props to a component, react automatically wraps them into a JavaScript object. This makes it easy to access and use the properties within the component.



In this example, we are passing the restaurant property from the parent component (Body) to the child component (RestaurantCard). As seen in the console, this property is automatically wrapped inside an object when passed as props.





How can we avoid wrapping the restaurant property inside an object?

We can achieve this by using **function call syntax** when passing props.





This is the restaurant object itself, not the restaurant object wrapped inside another object. Now, the child component receives the restaurant prop value directly, rather than accessing it from within an object.

Destructuring Props in the Child Component -

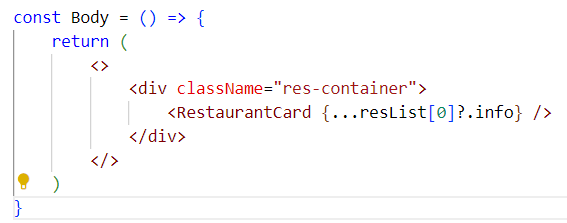




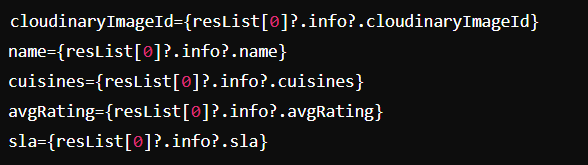
Destructuring props in Function Parameters Without the Spread Operator-



Destructuring Props in Function Parameters with the Spread Operator-



The expression ...resList[0]?.info uses the spread operator to copy all the properties of resList[0]?.info into a new object. This is not object destructuring - its object spreading (unpacking the properties). It’s equivalent to manually writing out each property of resList[0]?.info.



What is the Spread Operator (...) ?

The spread operator is used to expand or unpack elements from an Iterable - such as an array or object - into individual elements. It is commonly used where multiple values or key-value pairs are expected.

If we have 50 cards, manually rendering the RestaurantCard component each time would be inefficient and repetitive. Instead, we can use a loop (like .map()) to generate these components dynamically

In React, we usually use the map() function — a higher-order JavaScript method — to loop through a list of data and return a component for each item. Inside JSX, map () is used within curly braces {} so its output gets rendered on the UI.

In the code below, we loop through each restaurant object in the restaurant list and destructure each restaurant's data while rendering the RestaurantCard component.

We've created the basic structure for the Header, Footer, and Body components. A search bar will be added to the Body component later.

**Note:** Use the rafce snippet in VS Code (with ES7+ React/Redux snippets extension) to quickly scaffold a functional component with an export.

Why is React fast?

React is fast because it uses a Virtual DOM. Instead of touching the real DOM every time (which is slow), React first updates the Virtual DOM, finds what actually changed, and then updates only those parts in the real DOM. This avoids unnecessary work and makes UI updates efficient.

What is the Virtual DOM?

* A **lightweight copy** of the real DOM, stored in memory.
* Represented as a **JavaScript object** (also called a **Virtual DOM node**).
* Each node has 3 properties.
* **type** - the element type (div, span, ul, li, etc.)
* **props** - attributes, event handlers, and styles
* **children** - nested nodes or text

### *Example -*



***Virtual DOM Tree -***



<ul> - one node (one JS object).

First <li> - another node (another JS object).

Second <li> - another node (another JS object).

All together forms a tree of nodes (objects).

**One-Liner - The Virtual DOM is a tree of JavaScript objects where each node represents an element with** type**,** props**, and** children**. React updates this tree first, then syncs only the changes to the real DOM.**

## Why Do We Need the Virtual DOM?

***Real DOM updates are slow***

* The browser’s **real DOM** is very heavy.
* Every update (adding/removing/updating nodes) triggers **recalculations, reflows, and repaints**.
* If we directly updated the real DOM on every state change, the UI would lag.

***Virtual DOM makes updates faster***

* The **Virtual DOM** is just plain JavaScript objects (lightweight).
* React updates this **in-memory tree first** whenever state/props change.
* Then React **compares (diffs)** the new Virtual DOM with the old one.
* Finally, it updates **only the parts of the real DOM that changed**, not the entire UI.

***Efficiency with batching & reconciliation***

* React batches multiple updates together.
* Uses reconciliation (diffing algorithm) to apply **minimal changes** to the real DOM.

### JSX and the Virtual DOM

When you write JSX like <div>Hello, World! </div>, it is **compiled into** React.createElement, which returns a **React element** (a plain JavaScript object).

This object describes the element’s

* + **type** → "div"
  + **props** → attributes (like className)
  + **children** → "Hello, World!"

Together, these objects form the **Virtual DOM tree**.

### Functional Components and the Virtual DOM

When you define a functional component and render it, react does **not store the component itself in the Virtual DOM.**

Instead, react creates a **React element object** where

* + **type** - the function (component reference)
  + **props** - the props passed to it

***Example -***



***Output (simplified React element object) -***



When React renders it ((like root.render(<Greeting />)), it calls Greeting () which returns another object (<h1>Hello</h1>) → that object becomes part of the **Virtual DOM.**

One liner - JSX and functional components in React both compile into React elements (plain JS objects). These objects describe the UI (type, props, children) and together form the Virtual DOM tree.

## What is Reconciliation in React?

Reconciliation is the **process React uses to update the real DOM efficiently**.  
Instead of re-rendering the entire UI when state/props change, React:

1. Creates a new **Virtual DOM tree**.
2. **Diffs** it with the previous Virtual DOM tree.
3. Updates only the parts of the **real DOM** that actually changed.

This makes React **fast and efficient** compared to direct DOM manipulation.

## How Diffing Works (at object level)?

**Node Type Comparison** - If node type changes (<div> → <span>), React **replaces** the whole node.

**Props Comparison** - If type is the same but props change (e.g., className, onClick), React **updates only those props**.

**Children Comparison** - For nested elements, react **recursively compares children** to add, remove, or update only what’s needed.

### Example

JSX Before:



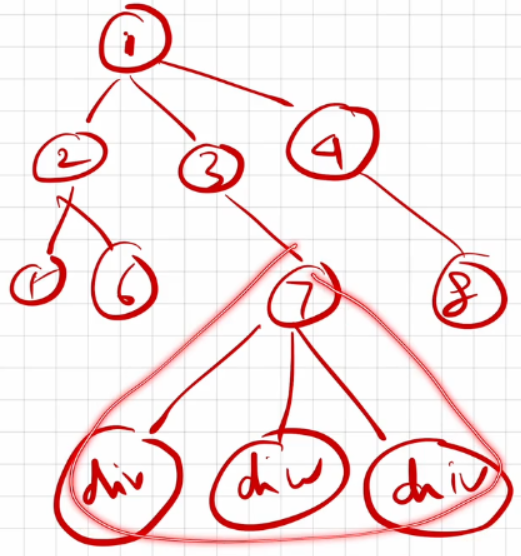
JSX After:



* Type = same (div) ✅
* Props = same (className="box") ✅
* Children = different (Hello → Hi) 🔄  
  👉 React updates **only the text**, not the entire <div>.

One-Liner - Reconciliation is React’s process of comparing the new Virtual DOM with the old one (diffing) and updating only what changed in the real DOM, making UI updates efficient

React’s Diffing Algorithm in Action -



If a change happens at the 7th node in the DOM tree, react does not re-render the entire DOM. Instead, through reconciliation, it re-renders only the affected part of the UI. The diffing algorithm pinpoints the exact change and updates just that section, making the process highly efficient

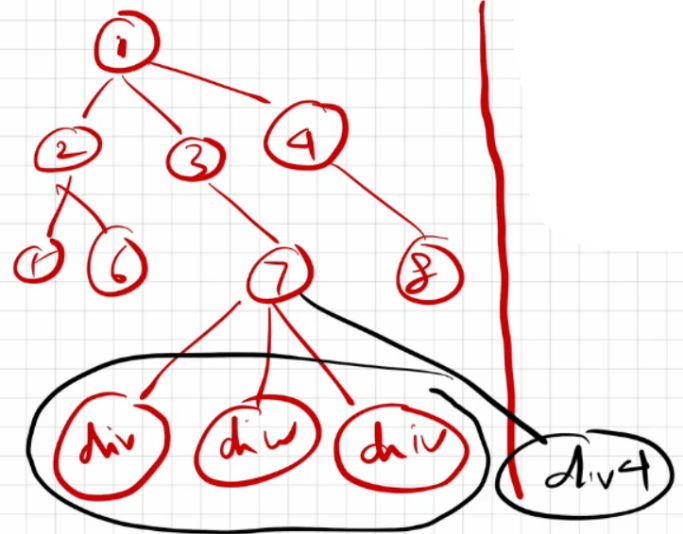
What are Keys in React?

Definition: Keys are special attributes used to uniquely identify elements in a list.

Purpose: They give elements a stable identity so React can track which items were added, changed, or removed during re-renders.

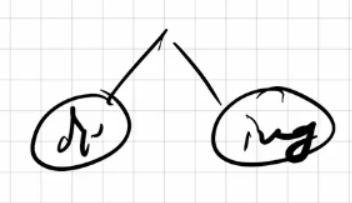
Benefit: Improves performance by allowing React to update only the changed items instead of re-rendering the entire list.

***case 1:***



In this scenario, if a new <div> (let's call it div4) is added at the 7th node, react may get confused because all the elements are <div> tags. Without unique keys, react won’t know which <div> was added or the correct order of the elements in the DOM. This lack of identification can lead to inefficient updates and unexpected behavior during re-renders.

***case 2:***



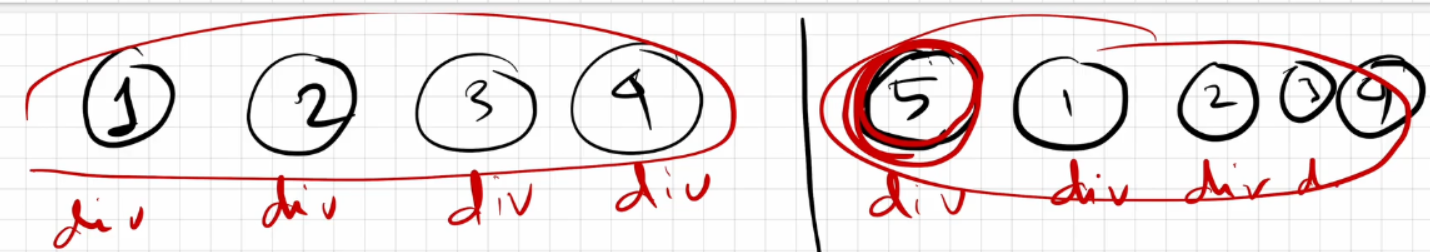
In this case, if a <div> is changed, an <img> tag is modified, or if both are swapped, react can accurately track these changes.

***case 3:***

When multiple children share the same tag name, react cannot effectively track them without unique keys.



Why Keys? A Detailed Explanation



In this example, the **left-hand side** shows the Actual DOM with 4 <div> elements (1, 2, 3, 4).

When an update occurs, react doesn’t directly change the Actual DOM. Instead, it creates a lightweight copy called the **Virtual DOM**, applies the changes there first, and then compares it with the previous Virtual DOM. The **right-hand side** represents this updated Virtual DOM, where a new 5th <div> is added at the beginning.

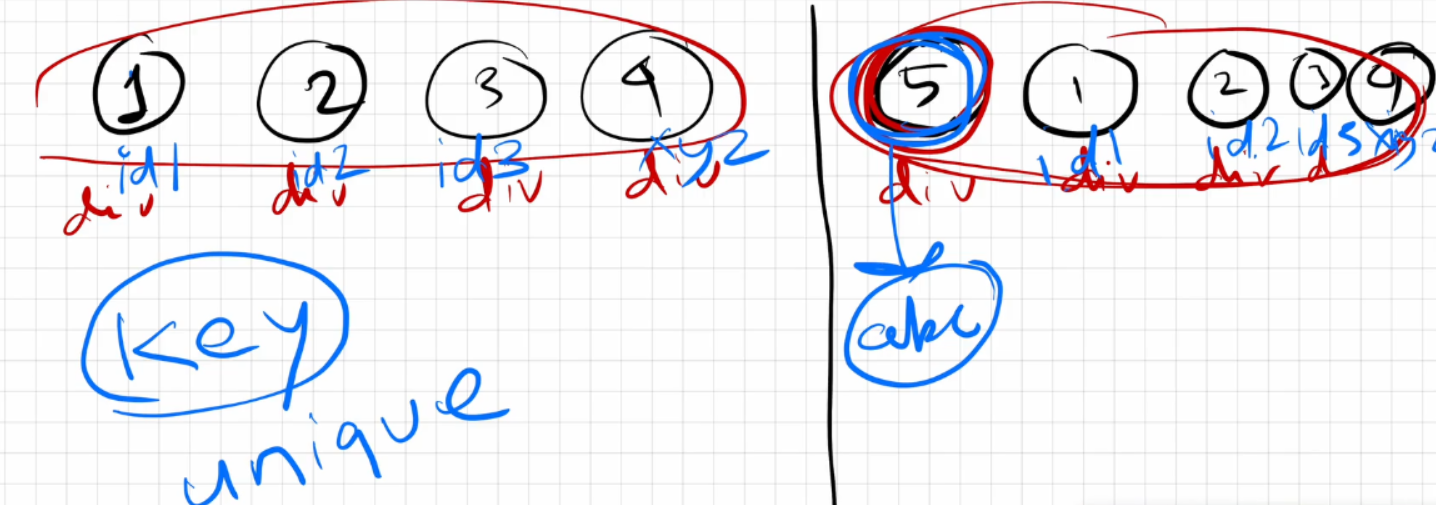
React then runs its **diffing algorithm** to detect changes.

* Without unique **keys**, react sees all <div> elements as identical.
* It cannot determine which <div> was newly added or where exactly it belongs.
* As a result, react **re-renders all 5** <div> **elements** in the UI, even though only one was new.

To solve this, react uses **keys.**

* **Keys** provide each element with a stable identity.
* With keys, react can compare old and new Virtual DOM trees and know exactly which element was added, removed, or updated.
* This allows React to **update only the changed element** (the new 5th <div>), instead of re-rendering the entire list.

One-liner - Keys help React uniquely identify elements in a list so its diffing algorithm knows exactly which items changed, ensuring only the necessary updates are made to the DOM.



Now that we’ve assigned **unique keys** to each <div> element, react can track them properly.

When we add a new 5th <div> at the beginning of the existing 4, React uses the **reconciliation process** with its **diffing algorithm:**

1. React compares the new Virtual DOM with the previous one.
2. It checks the **keys**:
   * Four keys (div1, div2, div3, div4) still match.
   * The new key (div5) does not match any existing one.
3. React concludes that only the new <div> needs to be rendered in the Actual DOM.

As a result, react **updates only the new element** instead of re-rendering all five. This makes the update **efficient and predictable.**

One-liner - With unique keys, React’s reconciliation process knows exactly which element is new or changed, so it only re-renders the necessary parts of the DOM instead of everything.

How did React know where to place the 5th div?

In the Virtual DOM, react compares the old and new trees. It sees that the new <div> has a unique key that doesn’t exist before and that it sits right before the first <div>. Using these keys, react tracks the position and updates the Actual DOM by placing the new 5th <div> before the first one.

Is JSX mandatory for React?

No, JSX is not mandatory for React. JSX and React are distinct concepts; JSX is a syntax extension that simplifies writing HTML-like code within JavaScript. It acts as syntactic sugar over pure React code, making it easier to create components in a more readable and convenient way.

Is ES6 mandatory for React?

No, you can write React code using pure JavaScript without ES6. However, ES6 introduces useful features like the spread operator and arrow functions, which can make building React applications easier and more concise. While ES6 is recommended for modern React development, it is not mandatory.

How to write comments in JSX?

JSX comments are written inside {/\* \*/}, and they can be used for both single-line and multi-line comments.

**Example:**

{/\* This is a single-line JSX comment \*/}

{/\*

This is a

multi-line

JSX comment

\*/}

What is React Fiber? Use cases of React Fiber.

In older versions of React (before version 16), React used the Stack Reconciler Engine (SRE) for reconciliation. However, it had some drawbacks, such as being synchronous.  
SRE works like a stack, processing tasks synchronously until the stack is empty.

For example, consider an application with a text box that makes an API call to fetch data before any input can be entered. While the data is being fetched, typing in the text box will not display anything. You can only type once the API call is complete. This means the app behaves synchronously, and both tasks cannot run in parallel without additional asynchronous handling.

To solve this problem, React introduced Fiber in version 16. Fiber is a completely rewritten reconciliation engine that supports asynchronous rendering and allows React to perform tasks more efficiently.

**Perks/Benefits**

* React Fiber focuses on animations & responsiveness.
* Fiber makes React faster and more efficient.
* Fiber splits work into chunks and prioritizes tasks based on their importance.
* Fiber can resume, pause, and restart rendering work on components as new updates come in.
* Fiber reuses previously completed work and can even abort tasks that are no longer needed.

Can we use the index as keys in React?

Yes, you can use the index of an item in a list as a key in React. However, it’s not a good practice for a few reasons:

1. Unexpected Behavior - If the list changes, using the index as a key can cause React to mix up the elements. This can lead to items being rendered incorrectly or in the wrong order.
2. State Loss - If you use the index as a key and reorder or change the list, react might not remember the state of each item.
3. Performance Issues - Using indexes can lead to performance problems when rendering large lists because React has to re-render more components than necessary.

It's better to follow this order for keys.

1. Use unique keys if you have them (like IDs).
2. Use indexes only if you have to, but be aware it can cause issues.
3. Don’t use no keys at all, because that can create problems.

What is Config Driven UI?

Config Driven UI is based on the configurations of the data that an application receives. It is managed from the backend.

A great example would be the Swiggy app's image carousel. Swiggy shows food offers based on geographical locations. For instance, if my home address is in Odisha and my friend’s home address is in Mumbai, we both access the Swiggy app from different places, but I might see different food offers in my region compared to what my friend sees in his region. These offers are typically displayed in the image carousel section of the Swiggy application. So, this image carousel section in the Swiggy app is config-driven, meaning we are viewing configured data in the UI based on our locations.

These configurations are managed from the backend. In some regions where there aren’t great offers, no data is sent from the backend to the UI, so people don’t see the image carousel section at all.