Chapter 5 - Let’s get hooked

Before we dive into React hooks, let’s first discuss the folder structure.

Is there a recommended way to organize React projects?

**Answer:** React **doesn’t enforce** any specific folder structure — you're free to organize your files however you like. But for better readability, scalability, and maintainability, it's **recommended** to follow a consistent and modular structure.

Recommended Modular Folder Structure for Large Projects



Explanation of Key Folders

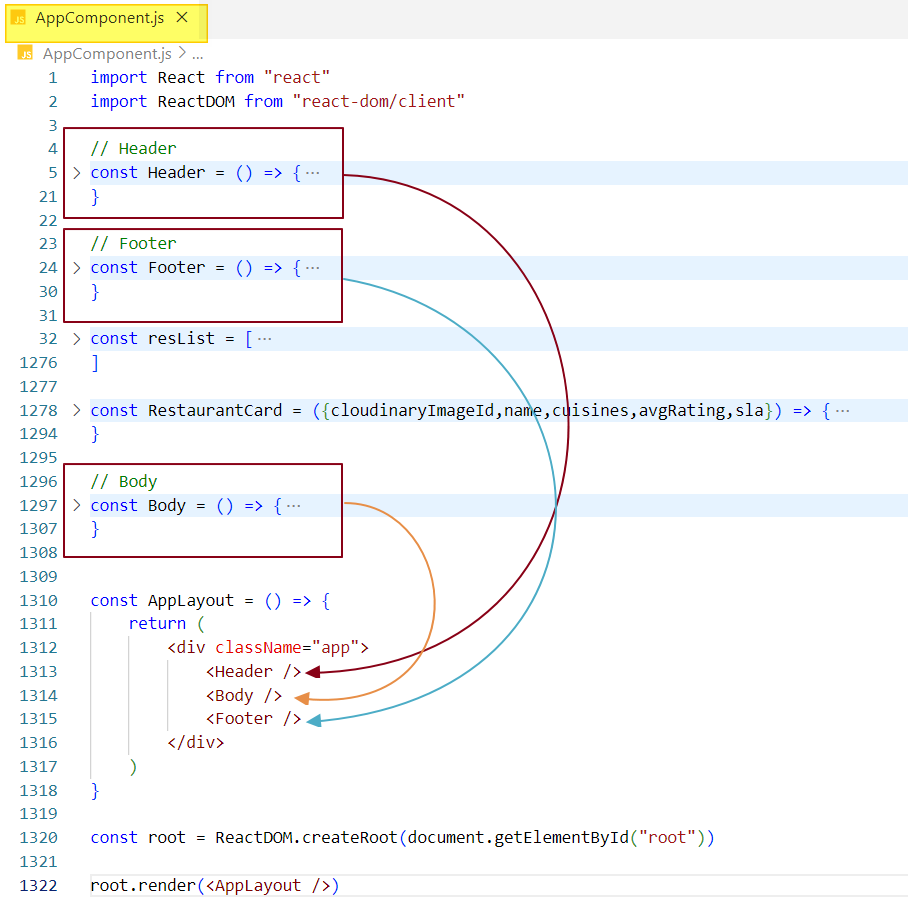
* assets/: Static files (images, fonts, icons) that don’t change frequently.
* components/: Reusable UI components (e.g., buttons, cards). Each component typically lives in its own folder with its styles and logic.
* features/: Group files by feature (e.g., auth, dashboard). Each feature has its own subfolders for components, services, and hooks, making it modular and self-contained.
* hooks/: Custom hooks shared across the app (e.g., useFetch or useAuth).
* services/: Handles all API calls and external services (e.g., authService for login, api.js for global API setup).
* store/: Contains your state management logic (Redux, Zustand, or Context API). The store is organized into actions, reducers, and slices to keep state management clean and scalable.
* utils/: Helper functions used across the app (e.g., formatting dates or validating inputs).
* styles/: Global styles like global.css, or Tailwind configuration, that are applied throughout the app.
* routes/: Handles routing with React Router or any other routing solution.
* layout/: Contains shared layout components like the Header, Sidebar, and Footer used across various pages.

We can create our own custom folder structure to keep the project organized. However, before doing that, let's first understand an important design pattern in React: The Single Responsibility Principle (SRP).

Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) is a design pattern that states each component should have only one specific responsibility or purpose.

In our application we created three components - Header, Footer, and Body - inside the AppComponent. This violates SRP because the AppComponent is handling more than one task: it not only creates the Header, Footer, and Body components, but also renders them inside the AppLayout component.



To adhere to the **Single Responsibility Principle (SRP),** each component should be responsible only for its specific role. Therefore, we’ll create separate files for the **Header**, **Footer**, and **Body** components. These components can then be imported into the **AppLayout** component as needed.

Next, we’ll restructure the project following common industry practices, organizing all code within a 'src' folder. While using a 'src' folder is not mandatory, we’ll follow this approach to align with industry standards.

It’s a best practice to create separate files for each component. We’ll have the following components:

1. Header
2. RestaurantCard
3. Body
4. Footer

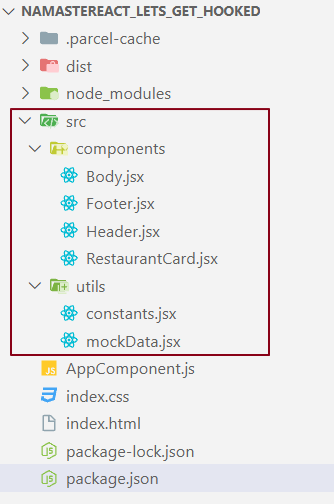
These components will be placed inside a 'components' folder, which will be a subfolder within the 'src' folder. When naming component files, always start with a capital letter. In this project, we are using the '.js' extension, but '. jsx' can also be used, depending on your preference.

Additionally, we will create a config or constants file to store hardcoded information. This file will be placed in a 'utils' folder, as per industry standards. The reason for this is that these files are considered utilities, providing helpful configurations or resources for the application.

We will also create a mockData file within the 'utils' folder to store hardcoded data for the restaurant list.

Note: In some cases, developers may use 'common' instead of 'utils' for the folder name.

Current folder structure -



Import and Export in React

***Import:***

The import statement allows us to bring in and use code (like components, functions, or variables) from another file that has been exported. In React, there are two main types of imports: default imports and named imports.

**Named Import** - Used to import specific exports by their exact names.



**Default Import** - Used to import the default export from a file. You can give it **any name** when importing. since there’s only one default.



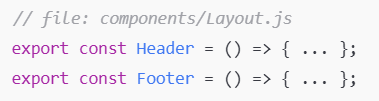
You can also combine both in one line if needed:



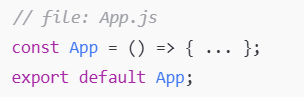
***Export:***

The export statement allows us to share code from one file (such as functions, components, or variables) so it can be imported and used in other files. In React, there are two main types of exports: default exports and named exports.

**Named Export -** Exports specific items by name.



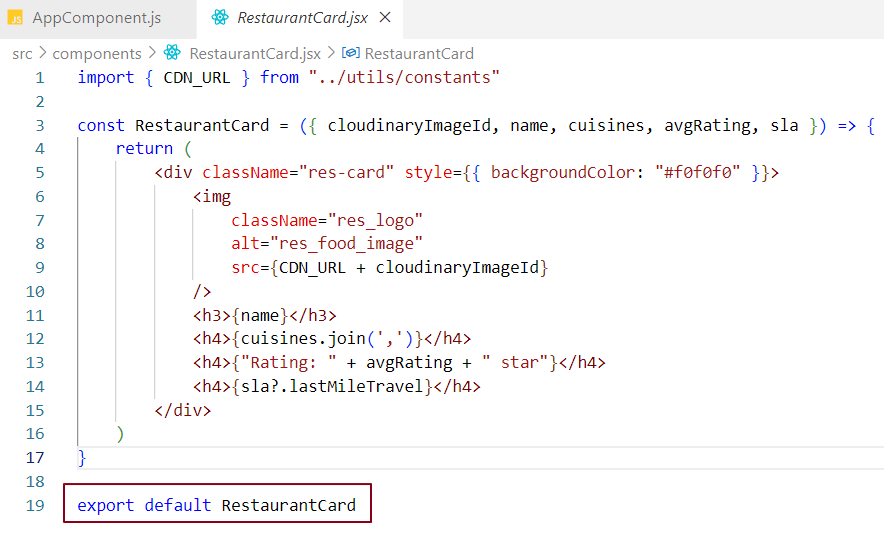
**Default Export -** Exports a single value (function, component, or variable) as the default export.



Default Export - (Export Single component)

A **default export** in React is used to export a single value (such as a component, function, or variable) from a file. It allows the value to be imported **without using curly braces.**

syntax - export default ComponentName;

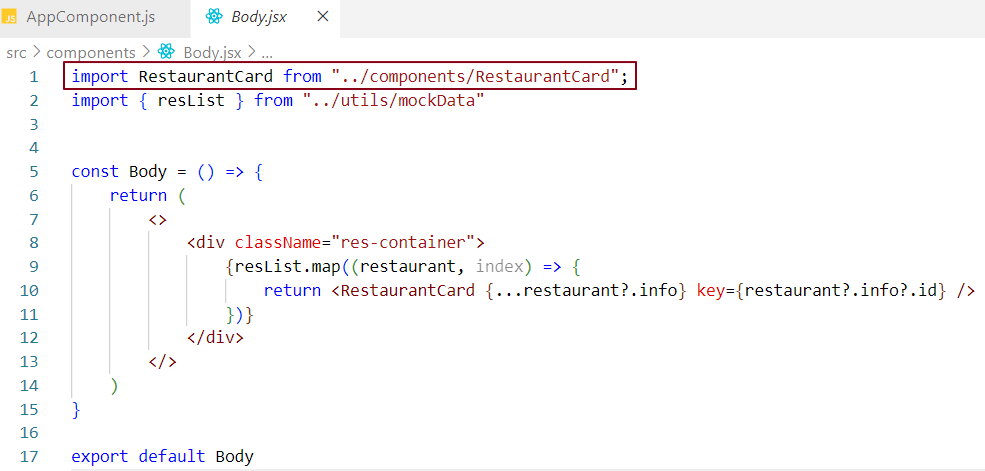


Default Import - (Import single component)

A **default import** in React is used to import a single value that was exported using export default from another module. One key benefit is that you can name the imported value anything you like in the importing file.

Syntax - import ComponentName from './FilePath';

In the code snippet below, the RestaurantCard component is imported into the Body.js file. While importing, including the .js extension in the file path is optional - it won’t affect the functionality. JavaScript automatically resolves .js files even if the extension is omitted.



With **default imports,** you can assign any name to the imported component. For example, if RestaurantCard is imported as XYZ, it won’t cause any error - just make sure you use XYZ consistently throughout the Body.js file.



Named Export - (Export Single component)

A **named export** allows you to export one or more values from a module using their exact names. These values must be imported **using the same names** in other files.

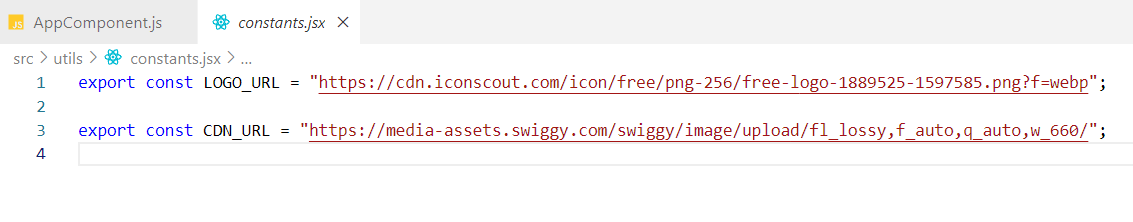
syntax -

To export specific values from a module using **named exports,** you can either export them directly or at the end of the file.

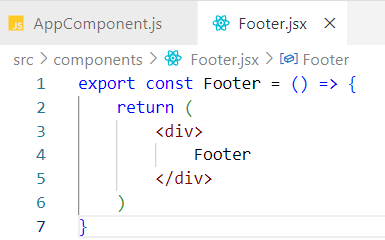


Both approaches are valid and widely used. Just make sure to import them using the **exact same names.**

Exporting Constant Values Using Named Export -



Exporting a Component Using Named Export-



Named Import - (Import Single component)

A **named import** lets you import specific values (such as functions, variables, or components) from a module by referencing their **exact names,** wrapped in **curly braces**.

Named import syntax - import { value1, value2 } from './filePath';

The names used in the import **must exactly match** the named exports from the source file.



**Note:** In the context of named imports, { componentName } is **not** object destructuring. It simply refers to importing a specific export by its name from a module.

Default Export - (Export multiple components)

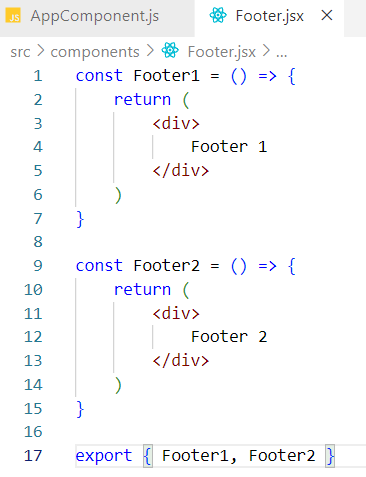
Not possible

Default Import - (Import multiple components)

Not possible

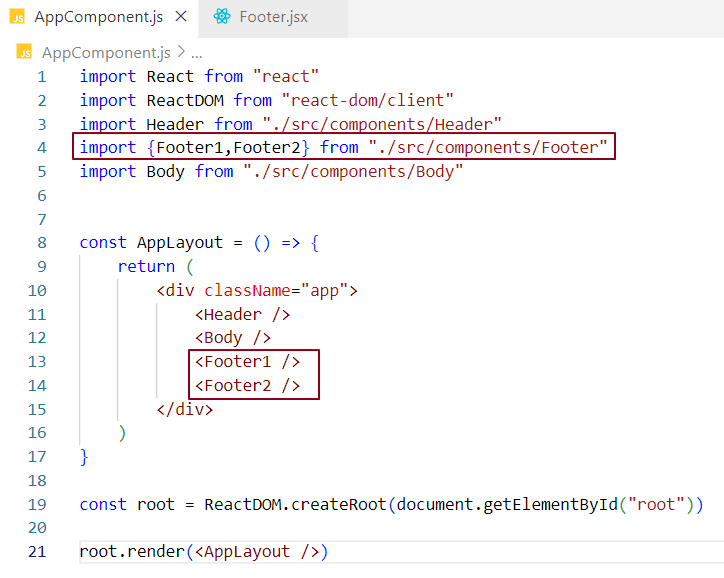
Named Export - (Export Multiple components)

Exporting two-Footer Components in React. You can export multiple footer components from a single module using **named exports.**



Named Import - (Import Multiple components)

Importing Two-Footer Components in React



Another way -



You can mix **default** and **named** imports and exports within the same file, but the rules for each still apply.

### Adding Functionality to Display Top-Rated Restaurants

In the previous chapter, we created the skeleton of our application. Now, let's enhance its functionality. We will add a button labelled **"Top-Rated Restaurants"**. When this button is clicked, it will filter out restaurants that have an average rating greater than 4.0 and display them in the UI.

### Steps:

* **Inside the Body Component**, add a button labelled **"Top Rated Restaurants"** above the restaurant list.
* Attach an **onClick event handler** to the button. This event handler will:
  + Filter the restaurants based on the average rating (greater than 4.0).
  + Update the UI to display only the top-rated restaurants.



Syncing Data Layer with UI in React

* In the code snippet, when we click the **"**Top Rated Restaurants**"** button, the onClick event handler is invoked, which filters the list of restaurants. After filtering, the listOfRestaurants variable is updated with the filtered data and logged to the console.
* At this point, we expect the filtered data to appear in the UI since we're using the map() function to render the filtered listOfRestaurants. However, there are no visible changes in the UI because listOfRestaurants is just a regular JavaScript variable, not a state variable managed by React.
* React does not automatically re-render components when regular variables are updated. To ensure the UI layer stays synced with the data layer, we need to use React state variables, which are designed to trigger a re-render when their values change.

How to Fix It?

To sync the UI layer with the data layer, we need to use React's useState hook to store and update the list of restaurants.

Why do I need state or state variable when I have local JS variable?

* React only tracks state variables - it does not monitor regular local variables. In our code, if we modify a local variable like listOfRestaurants using a function, react remains unaware of that change. Even if the variable is used throughout the component, react won’t know why or when it changed. As a result, the UI will not update to reflect the new value.
* This is because localvariablesarestateless - meaning they do notretaintheirvalue between re-renders. Every time a component re-renders (which happens when state or props change), all local variables inside the component function are re-initialized from scratch. React does not remember what value they held previously, and it has no way to detect changes in them.
* On the other hand, state variables are persistent - their values are preserved between re-renders, and React actively monitors them. If a state variable changes, react triggers a re-render to reflect the updated value in the UI.
* To ensure our data stays in sync with the UI, we need to use state variables, which React can track and manage over time. This is where the useState hook comes in - it allows us to create, update, and preserve values across renders. React listens for changes in state variables and automaticallyupdatestheUI when they change.

### What is State in React?

State in React is a special mechanism used to store data that can change over time within a component. It allows the component to remember information between renders and automatically re-renders the UI when that data changes.

#### **Key Points:**

* Storage for Data: State stores things like user input, whether a button is clicked, or how many items are in a cart. It can store simple values (like numbers, text, or true/false) or more complex ones (like objects).
* Dynamic Updates: State can change as the user interacts with the app. When something changes (like a button click), React updates the state and automatically updates the UI without you needing to manually manipulate the DOM.
* Automatic UI Updates: Whenever the state changes, react automatically updates the UI to match the new state. This means you don’t need to manually update the DOM to reflect changes.
* Component-Specific: Each component can have its own state, keeping it separate from other components unless shared intentionally.

#### **Example:**

Imagine a simple counter app:

* **State**: Holds the current count (count).
* **Button Click**: When you click a button to increase the count, the state (count) updates to the new number.
* **UI Update**: The UI automatically displays the new count after the state change without any manual intervention.

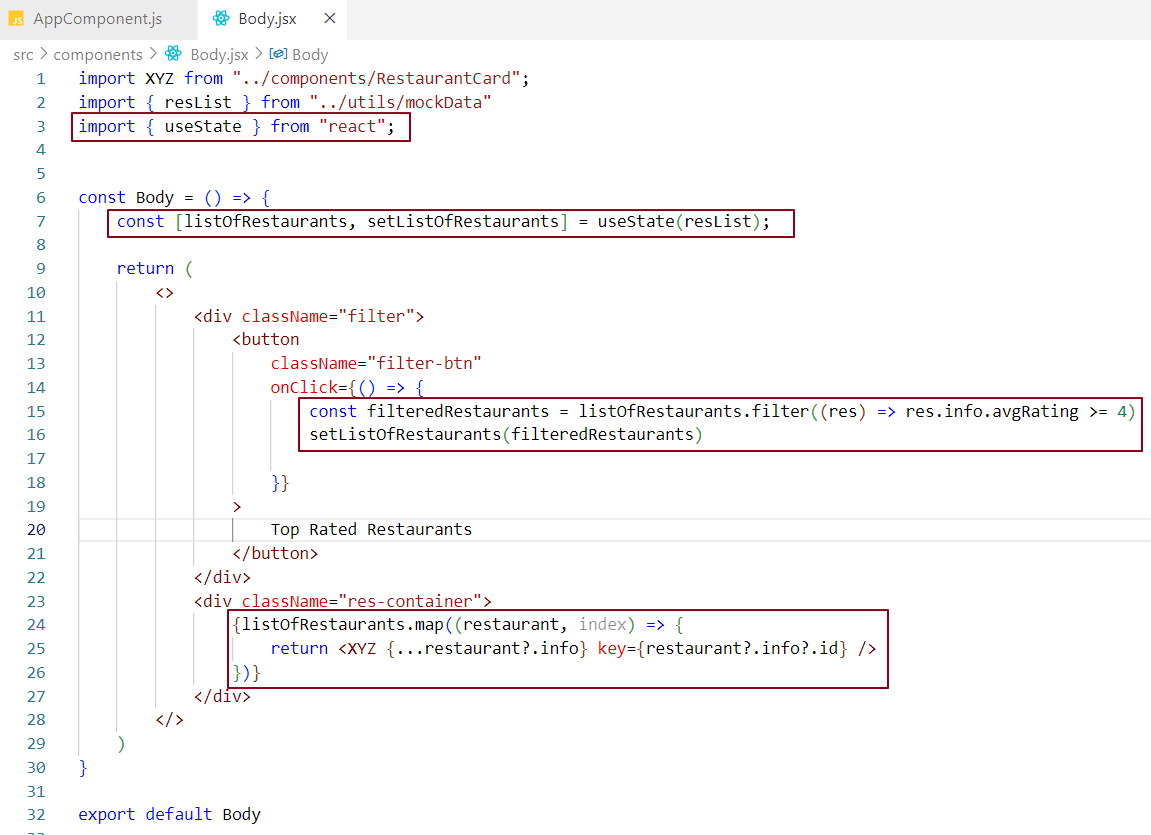
What is a Hook?

A hook is a JavaScript function provided by React to enable specific functionality, like managing state, in functional components. React introduced hooks to make it easier to work with features that were previously only available in class components. One of the most commonly used hooks is the useState hook, which allows you to add state to a functional component.

What is a useState Hook?

* The useState hook is a function provided by React that allows you to create state variables in a functional component.
* The useState hook returns an array where the first element is the state variable, and the second element is the setter function used to update the variable.
* The useState function expects an initial value as an argument to set up the state variable.

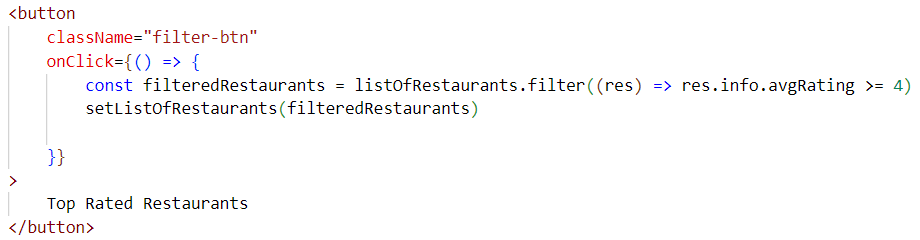
Using a React state variable with the useState hook -





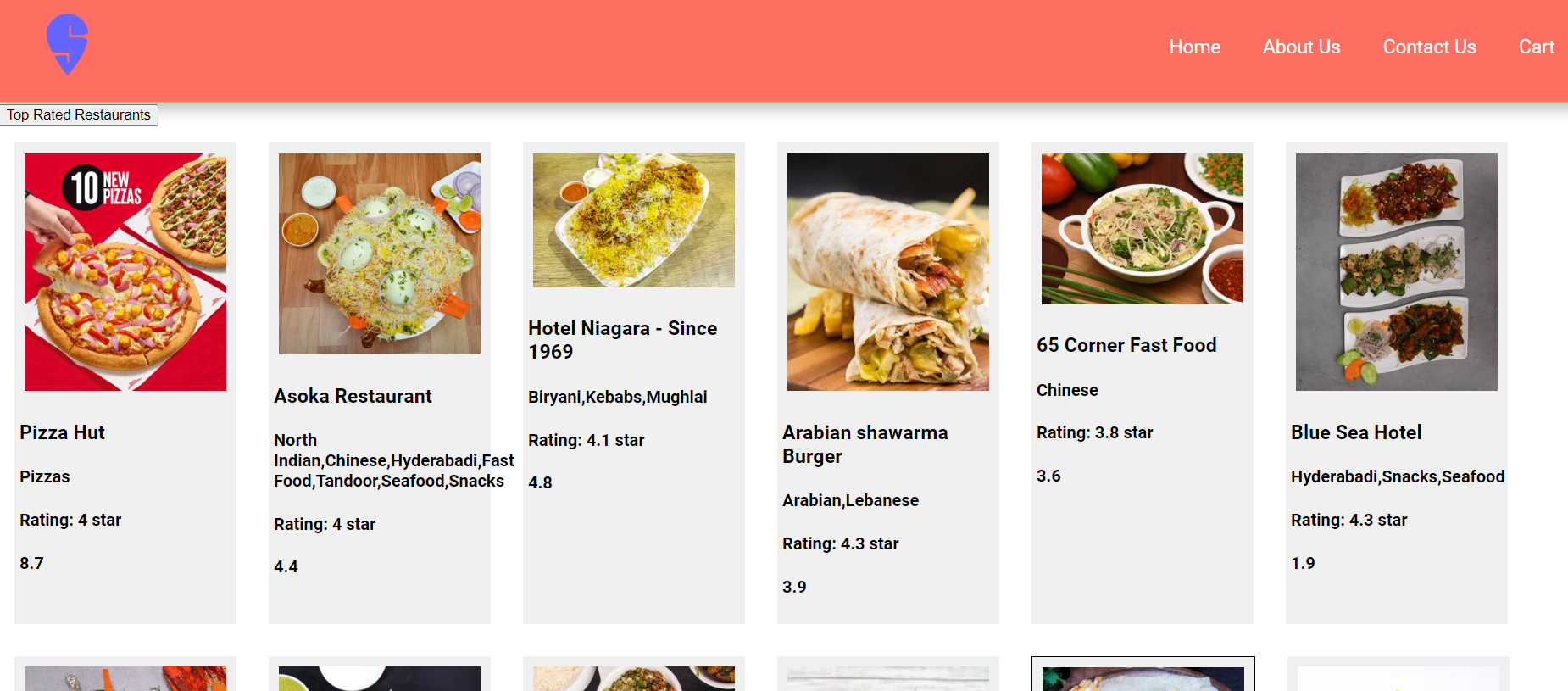
listOfRestaurants is the state variable that holds the list of restaurants, initially set to the value of resList.

setListOfRestaurants is the function used to update listOfRestaurants. When called with a new value, it triggers a re-render of the component to display the updated list.

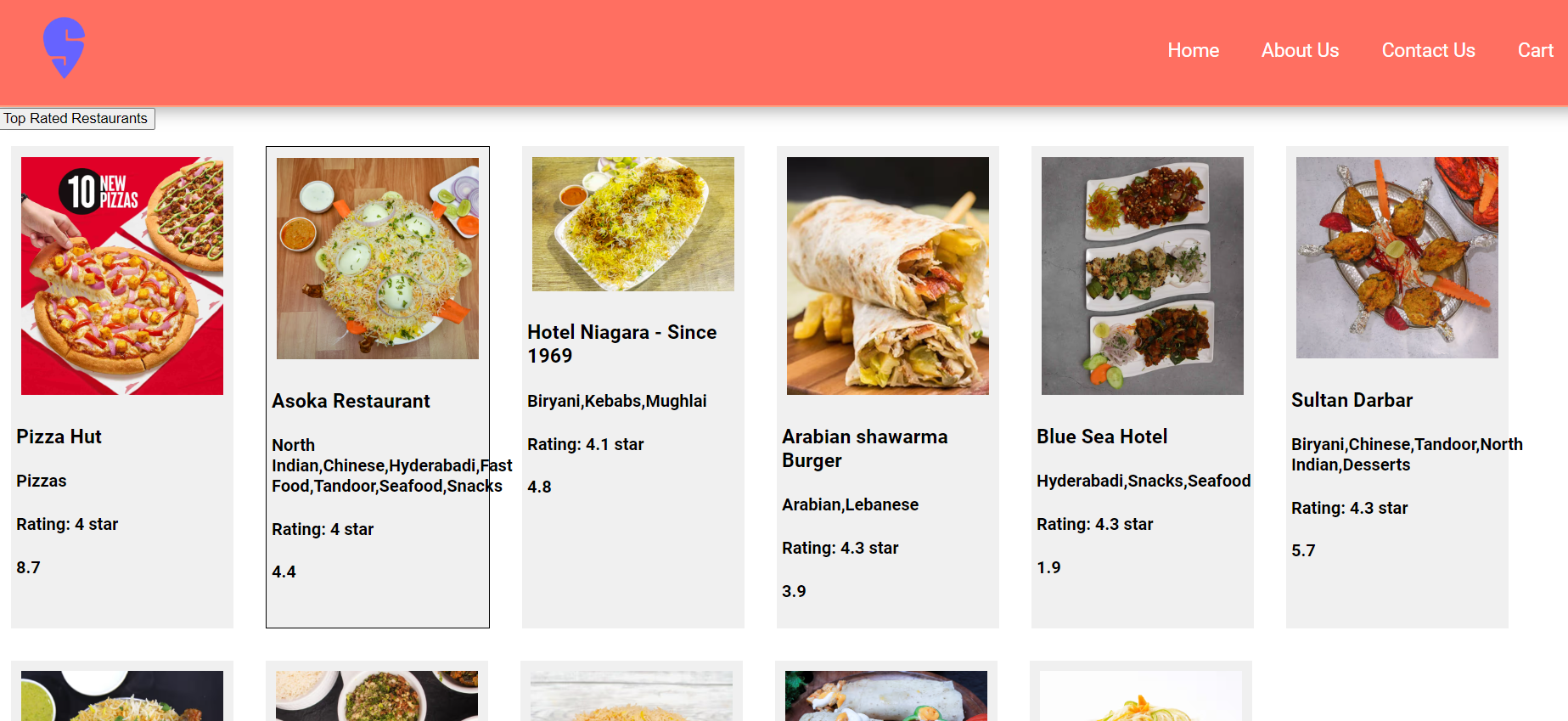


Clicking the 'Top Rated Restaurants' button triggers the onClick event handler, which performs the following actions:

* Filtering: The code filters listOfRestaurants to retain only those restaurants with an average rating of 4 or higher.
* Updating State: The setListOfRestaurants function updates the state with the filtered list.
* Reconciliation: React re-renders the component, checks for changes in the state, and updates the UI to display the new list of top-rated restaurants.



After clicking the Top-Rated Restaurants button, we see the top-rated restaurant cards displayed based on our filter logic.



Virtual DOM and Reconciliation in React -

In React, when you create elements, you're actually creating virtual DOM objects. These virtual copies are kept in sync with the real DOM through a process called "Reconciliation" (also known as the "diffing" algorithm). During each rendering cycle, react compares the new virtual DOM (the updated version) with the old virtual DOM (the previous version) and makes only the necessary changes to the actual DOM. This efficient process ensures that updates happen quickly, optimizing performance.

State Updates and Reconciliation -

When the useState setter function is invoked or a component's state changes, react triggers a reconciliation process to determine the differences between the current and previous **virtual DOMs**. This process, known as the diffing algorithm (or React Fiber), compares the **current virtual DOM (VDOM)** with the **previous virtual DOM (VDOM)** and identifies which parts of the UI have changed. React then updates only the altered portions of the UI in the **real DOM,** minimizing unnecessary changes and optimizing performance. As a result, when the setter is called to update a state variable, the component re-renders, and React ensures that only the modified sections of the UI are updated.