Chapter 9 – Optimising App

Let's optimize our app using custom hooks and dynamic bundling.

Why use Hooks in React Instead of Regular JavaScript Functions?

1. State Management

**What it Does:** Hooks like useState let you create state variables directly inside functional components.

**Why It Matters:** React can keep track of these state variables automatically. If their values change, react knows to re-render the component. Without hooks, react wouldn't know when internal data changes, which could lead to an outdated user interface (UI).

2. Lifecycle Management

**What it Does:** Hooks like useEffect help you manage side effects, like fetching data or setting up subscriptions.

**Why It Matters:** Hooks let you choose when these side effects occur in a component’s lifecycle. Regular JavaScript functions don’t have this ability, making it harder to control how the component behaves over time.

3. Automatic Reconciliation

**What it Does:** Hooks help React update the DOM quickly when state or props change.

**Why It Matters:** This means that only the parts of the UI that need to be updated will change, which makes everything run faster. Regular JavaScript functions don’t work with React’s update system, so they can’t trigger changes in the UI when state changes.

Why Build Custom Hooks?

We build custom hooks to keep our code organized and reusable. They let us share code between different components without repeating ourselves. This follows the Single Responsibility Principle (SRP), which means each part of our code should only do one job. By using custom hooks, we make our code easier to understand and maintain.

Benefits of Custom Hooks

**Reusability:**

Custom hooks are JavaScript functions. You can write your logic once and use it in multiple places, promoting code reusability.

**Readability:**

Meaningful names for custom hooks help clarify their purpose. This makes your code easier to understand at a glance.

**Modularity:**

React components should focus on rendering data. By moving functional logic into custom hooks, you keep components simpler and more focused.

Modularity means breaking your code into smaller, manageable pieces, each with its own responsibility.

**Testability:**

Smaller pieces of code can be tested more easily. By organizing code into custom hooks, you can write more effective test cases.

Example Scenario -

Imagine a component that:

* Makes an API call to fetch data.
* Displays the fetched data in the UI.

Problem:

This component has two responsibilities:

* Fetching data (API call).
* Displaying data in the UI.

#### Solution:

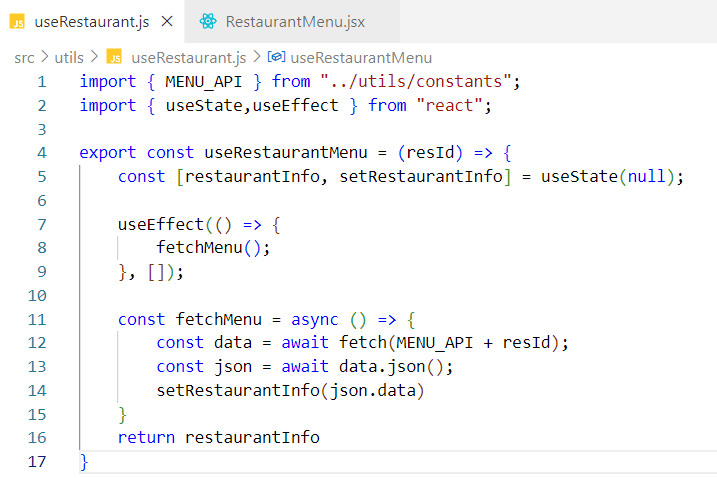
To follow the Single Responsibility Principle:

* Move the API call logic to a custom hook (e.g., useFetchData).
* Keep the component focused only on displaying the data.

Currently, the RestaurantMenu Component violates the principle of modularity by handling two responsibilities: fetching data and displaying it in the UI. Ideally, the component should focus solely on rendering the data and not concern itself with where the data comes from, how it is fetched, or which API is being called. This concept is known as data abstraction.

To improve this situation, we can create a custom hook useRestaurantMenu to manage the data-fetching logic separately. By doing so, we simplify the component's role, allowing it to focus exclusively on displaying the data without worrying about the data-fetching process.

Let's create a custom hook called useRestaurantMenu in the utils folder. Since this hook serves as a utility function, it is best practice to place it in the utils folder. We can then use this hook within the RestaurantMenu component.

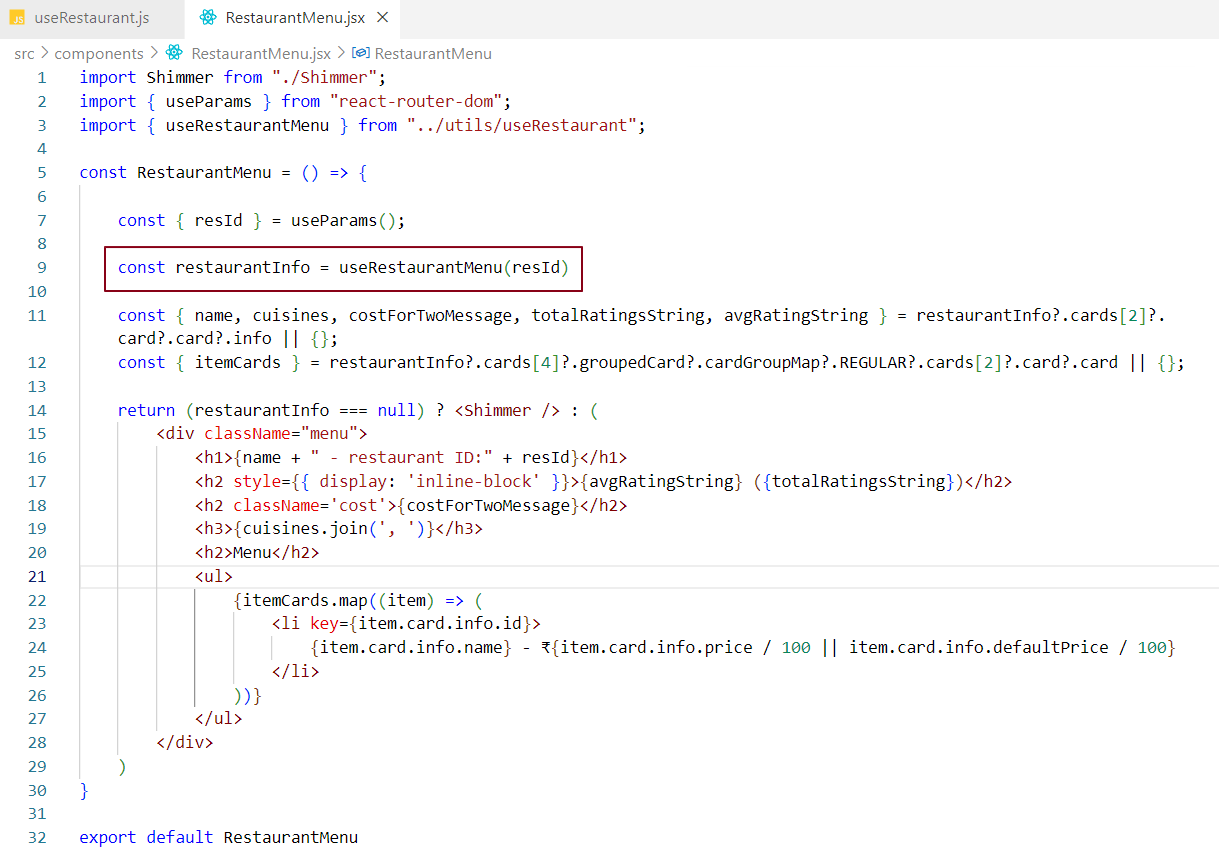


The useRestaurantMenu hook fetches and returns the menu data for a restaurant based on the given resId, which helps the RestaurantMenu component display the correct menu information.

### Constants and Hook Usage

**Constant**: resId (this is the unique ID used to get the specific restaurant's menu)

**Hook Usage**: The hook is called in the RestaurantMenu component, where resId is passed in to get the right menu data.

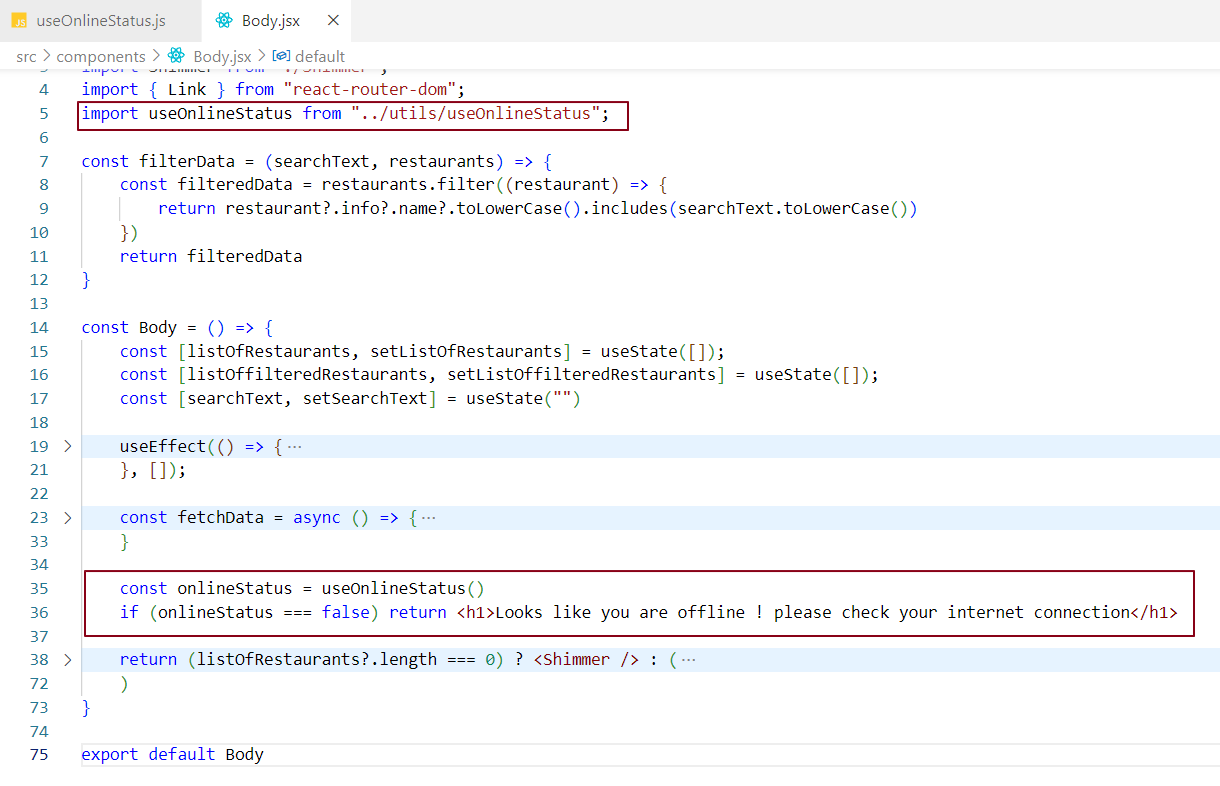


We initially included the API call logic within the RestaurantMenu component. However, according to the Single Responsibility Principle (SRP), this component should solely be responsible for rendering data in the UI and should not handle how the data is fetched from the API. To adhere to SRP, we have separated this logic from the RestaurantMenu component and encapsulated it within a custom hook in the helper module, useRestaurantMenu.js. Now, whenever the RestaurantMenu component needs this functionality, it can simply import it from the hook.

Building Online and Offline features -

If the user has no internet connection, display 'You are offline. Please check your internet connection.' Otherwise, show the actual data.





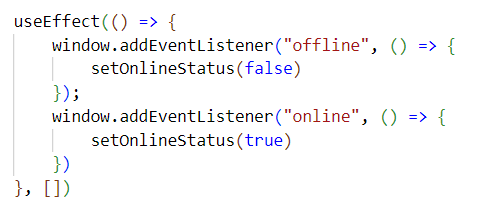
### Event Handling in useOnlineStatus

**1.Adding Event Listeners**:

In the useEffect hook, we use window. addEventListener to listen for two specific events:

**"offline"**: This event is triggered when the browser loses internet connectivity.

**"online"**: This event is triggered when the browser regains internet connectivity.



**2.Anonymous Functions**:

Each addEventListener call uses an anonymous function that sets the onlineStatus state:

When the offline event is fired, it calls the function that sets onlineStatus to false.

When the online event is fired, it calls the function that sets onlineStatus to true.

**3.State Updates**:

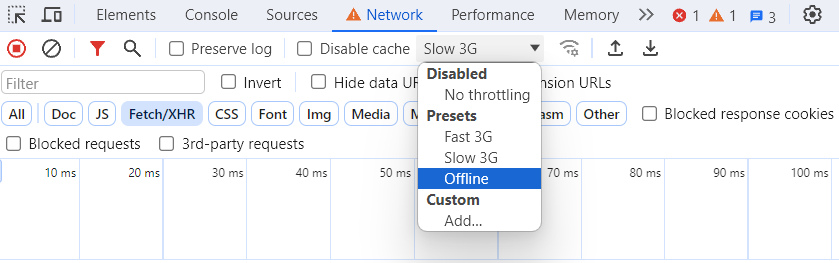
The setOnlineStatus function is part of the useState hook. It updates the onlineStatus state variable based on the event:

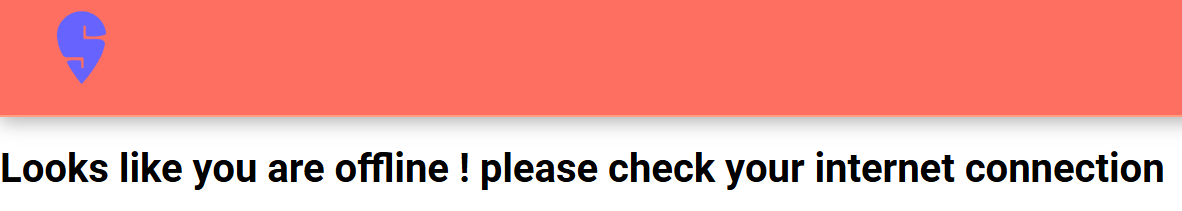
If the user is offline, setOnlineStatus(false) updates the state to reflect that the user is not connected to the internet.

If the user is back online, setOnlineStatus(true) updates the state to indicate a restored connection.

Simulating Offline and Online Behavior in the Chrome Browser

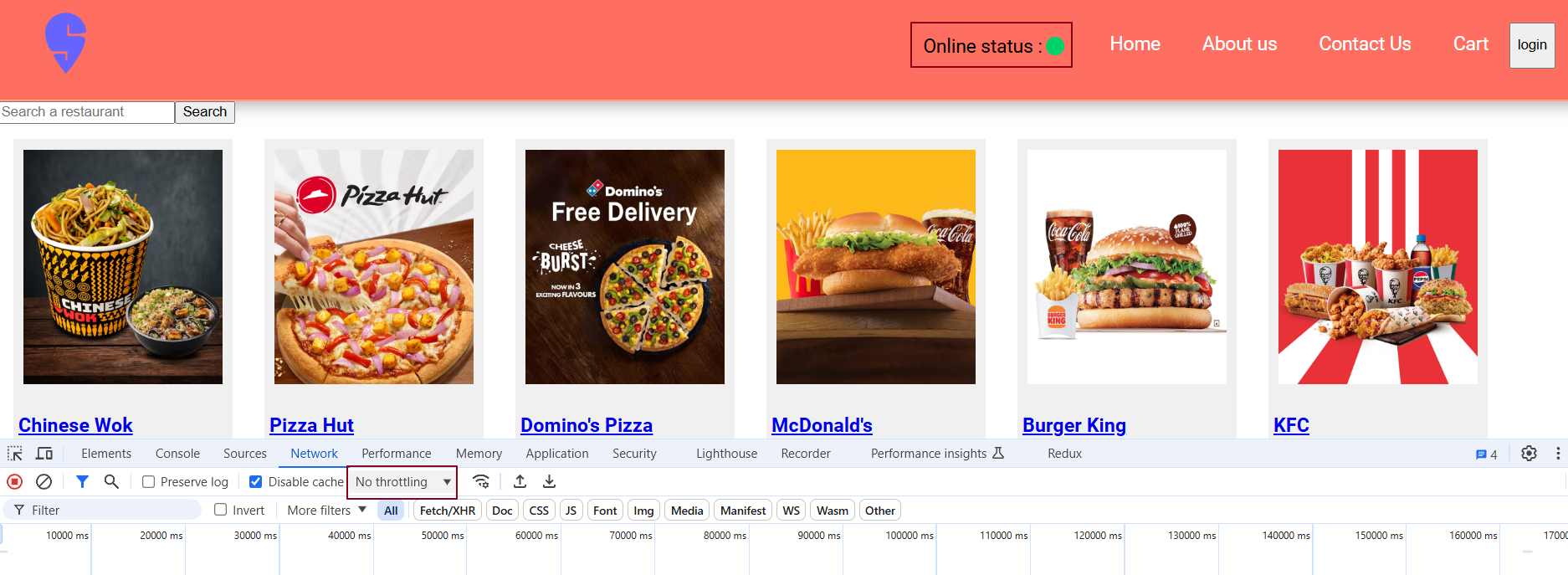
**Offline** - No internet connection, **no throttling** - Internet access restored.

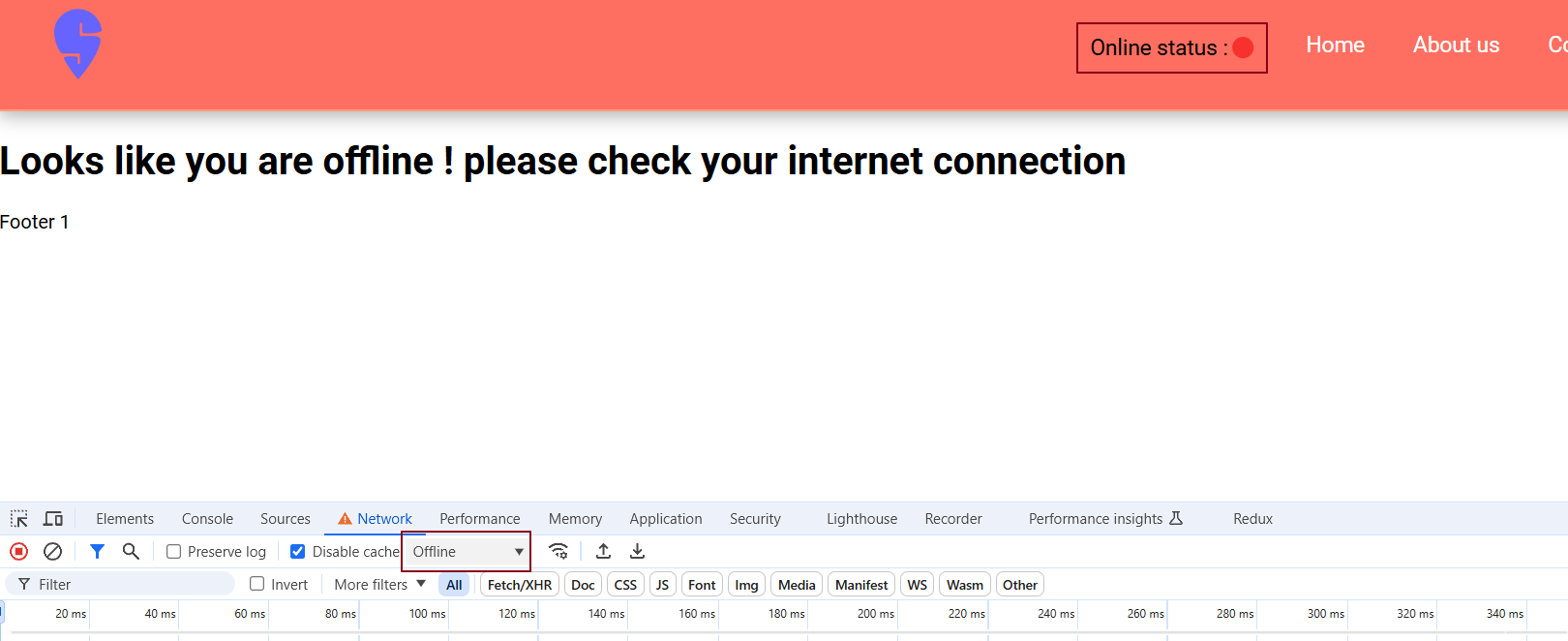




We can use this feature to enable chat notifications with color indicators: green when the user is online, red when user is offline. Let’s apply this feature in our header component.



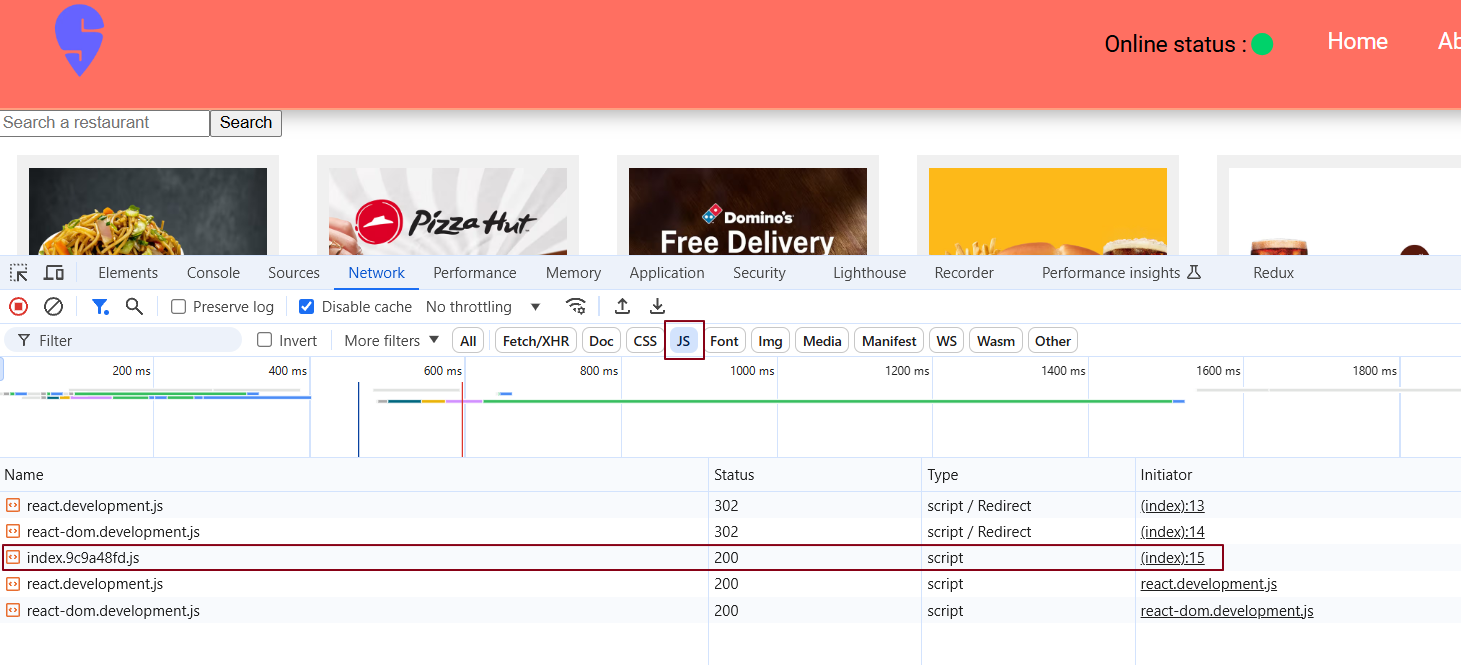




**Note:** It’s good practice to start custom hook names with 'use'.

Although not strictly required, some projects use a linter that enforces this naming convention. If we don’t follow it, the linter may throw an error.

In our application, Parcel has created a single JavaScript file called index.js, where all of our code is bundled and minified together. This index.js file, located inside the dist folder, is the file served to the browser. We can clearly view this in the developer tools as shown below.



The size of the index.js bundle is smaller in the production build.

In large-scale applications with thousands of components, bundling everything into a single file can make the application slow, as it forces all components to load at once on the first page load. To improve scalability, performance, and efficiency, we should split this large bundle into smaller chunks. These mini-bundles can then load only when specific conditions are met, ensuring faster load times and a smoother user experience.

What is bundling? Who handles the bundling process?

Bundling is the process of merging several code files into one optimized and compressed file to improve the speed and performance of an application. The goal is to reduce the number of files the browser needs to load, which helps enhance loading times. Bundler tools like Parcel or Webpack carry out this process by combining and compressing files, removing unnecessary parts to create a more efficient version of the application. The final bundled file is then sent to the browser when a user loads the page, making the overall experience smoother and faster.

What is Chunking / Code Splitting / Dynamic bundling / Lazy Loading / On demand Loading / Dynamic Import / Dynamic bundling?

If we build a large-scale production-ready application with multiple components, bundling all of them into a single JavaScript file can lead to longer rendering times in the UI. To improve performance, we should break the application code into smaller chunks, a concept known as chunking or code splitting. This approach allows only the necessary parts of the code to load when needed, enhancing the application's speed and efficiency.

Is bundling good?

Bundling is useful, but only to a point. We shouldn’t combine all the component code into one file and load it in the UI. Instead, we should make smaller, logical bundles. This means we should load only the bundle that is needed for a specific task.

For example, consider MakeMyTrip (MMT), which is an app mainly used for booking flights. When users open MMT, they usually start on the flights page. From a developer's view, we only load the flight components when the app starts, not every service at once. Even though we could load the train booking bundle, users don’t want to see it right away. When someone wants to book a train and goes to that part of the app, then we load the train components.

In short, developers should bundle components based on what users need instead of loading everything at once. For small apps, it's often fine to have just one bundle because it won't slow things down.

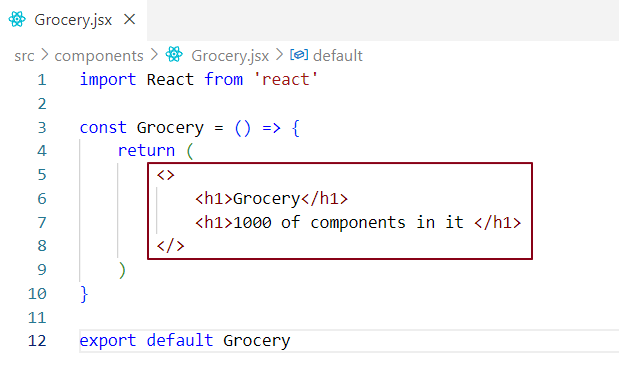
Where do we implement on-demand loading?

We perform on-demand loading of a component in situations where the bundle will be loaded only as needed.

Why on-demand?

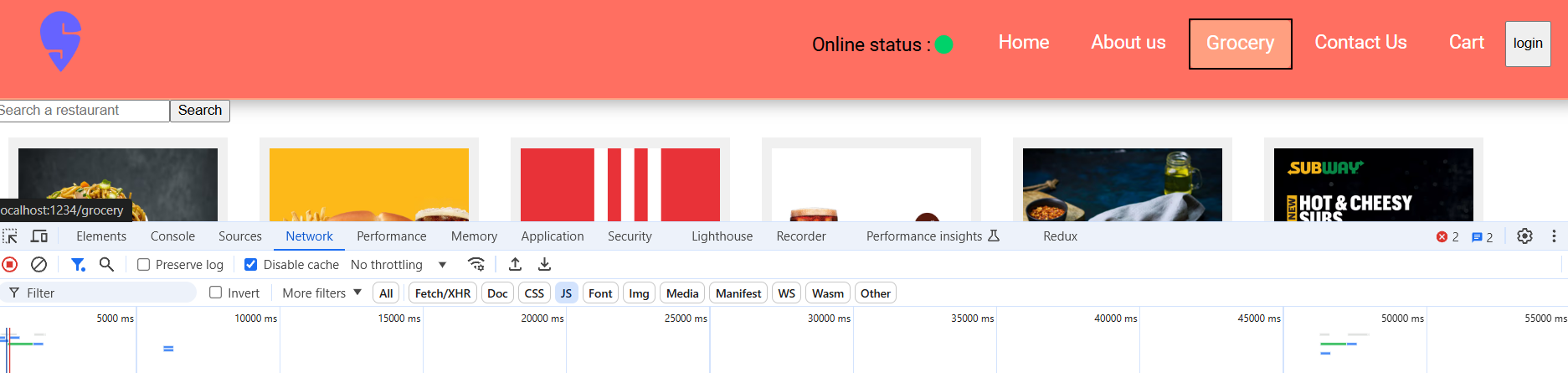
Because we only want to load the bundles when the user navigates to that specific page.

Let us create a component Grocery and load this component on demand inside app.js.

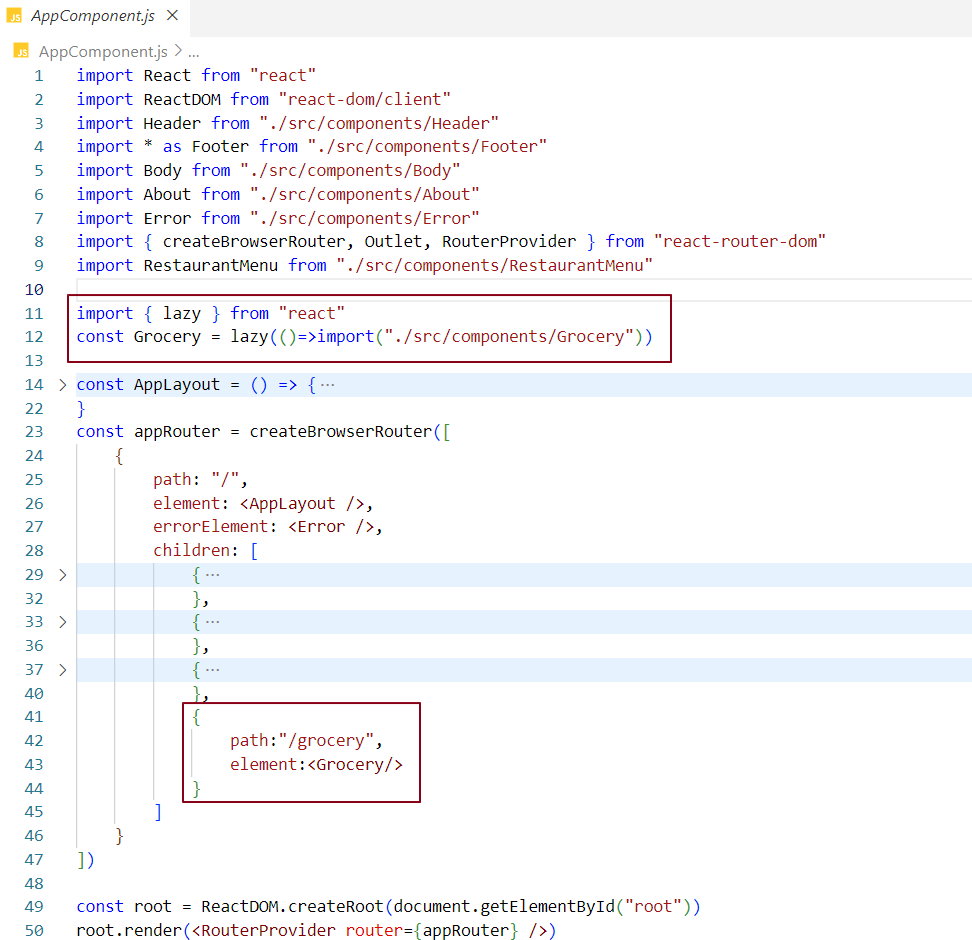


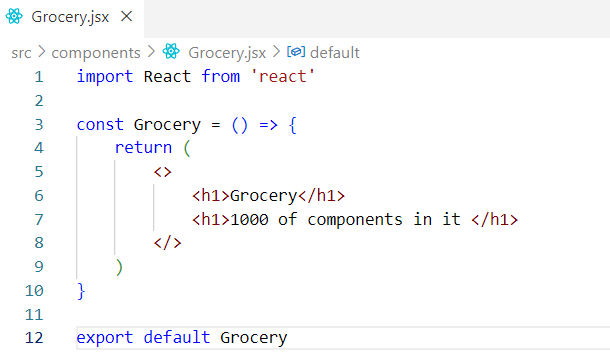
This code is still present in our main bundle, index.js. However, this is not efficient because this single file contains hundreds of components, all loaded into the index.js bundle. As a result, the size of the bundle file increases, leading to performance issues.

Let's dynamically load the **Grocery** bundle when the user clicks on the **Grocery** link in the header.

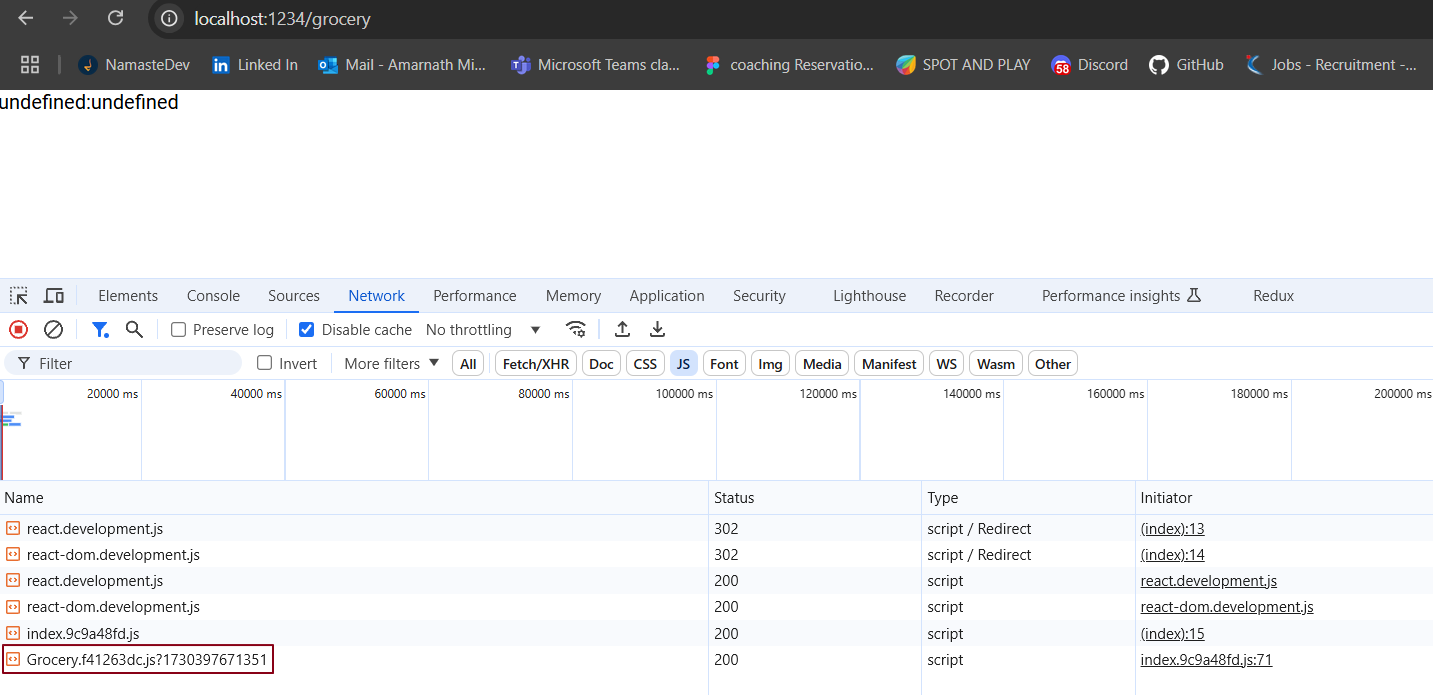








When we click on the Grocery link, we load the Grocery component's bundle. This is known as **on-demand loading.**



### Theory of React.lazy

**1.Purpose of React.lazy**

The lazy function in React is used for **code splitting**, allowing you to load components only when they are needed. This helps improve the performance of your application by reducing the initial load time.

**2.Callback Function**

React.lazy accepts a **callback function** as an argument. This function, when executed, will call the import function to load the component dynamically.

**3.The import Function**

The import function is a **dynamic import** that returns a **promise**. When you call import("./path/to/component"), it attempts to load the specified module asynchronously.

Once the component is loaded successfully, the promise resolves, allowing the lazy-loaded component to be rendered.

**4.How It Works Together**

When you use React.lazy, it takes the callback function you provided, which contains the import statement.

When the Grocery component is needed (like when the user clicks a link), React calls this callback function, triggering the import.

While the component is loading, you can show a fallback UI (like a loading spinner) until the component is ready to be displayed.

In our application, when we navigate to the Grocery component, we encounter an undefined:undefined error. What could be causing this?

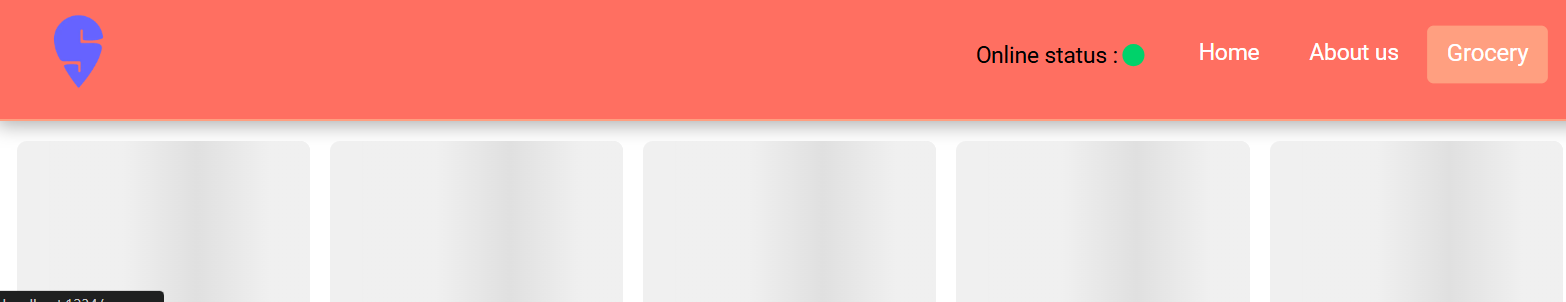
When we load the Grocery component, the Grocery component bundle takes some time to load. Meanwhile, react tries to render the Grocery component, which does not exist, meaning this component is inside the bundler and the bundler has not loaded it. In this process, react suspends this rendering. That’s why we are getting the error component page in the UI. At this moment, err.status and err.statusText will have undefined stored in them. When we are loading our component on demand, react tries to suspend it.

The Grocery card component is a suspense component because we don’t know whether it will be loaded in the UI or not. It will load only when the bundle for this component is loaded.

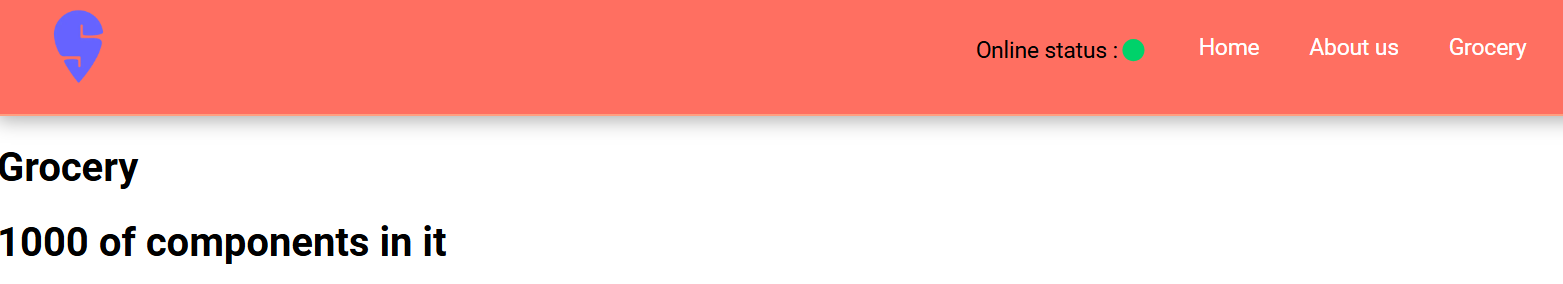
What is suspend?

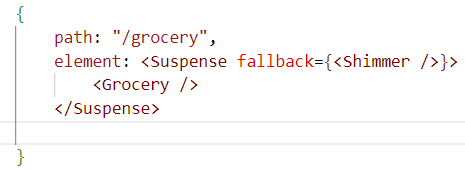
<Suspense> is a React component that suspends the rendering of lazy-loaded components until the corresponding bundle for that component is fully available. It allows you to show a fallback UI (like a loading spinner) while waiting for the component to load.

When I click on the Grocery link, the bundle for the Grocery component is not immediately available. We have provided a Shimmer UI inside the fallback property of the <Suspense> tag, which means this Shimmer will be displayed until the bundle for the Grocery component is fully loaded.



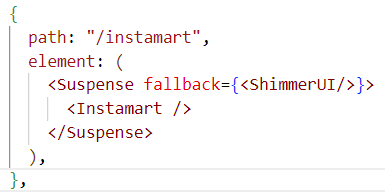
Once the bundle is loaded, we can retrieve the Grocery component and display its data in the UI.





What is Fallback in react?

When the bundle is not loaded, react suspends the component rendering until the bundle becomes available. During this time, to enhance user experience, we should display a shimmer UI on the screen. The Suspense component accepts a prop called fallback, where we can specify the shimmer UI component. We can also write JSX within the fallback attribute using curly braces {}.



Why should you avoid lazy loading a component inside a parent component?

Because if you do, the child component will be lazily loaded on every render due to any state or prop change in the parent component. This means that the bundle for the child component will be loaded each time, which is not efficient.