Chapter 9 – Optimising the App

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

Why should we use hooks in React instead of regular JavaScript functions?

### 1. **State Management**

* **What it does:** Hooks like useState allow you to create state variables directly inside functional components.
* **Why it matters:** React automatically tracks these state variables. When their values change, react re-renders the component to reflect the new state. Without hooks, react wouldn't be aware of internal data changes, which could result in an outdated user interface (UI).

### 2. **Lifecycle Management**

* **What it does:** Hooks like useEffect help you manage side effects such as fetching data, setting up subscriptions, or manually manipulating the DOM.
* **Why it matters:** Hooks give you control over when side effects are executed during a component's lifecycle (e.g., on mount, update, or unmount). Regular JavaScript functions don’t have access to the React lifecycle, making it harder to manage component behavior over time.

### 3. **Automatic Reconciliation**

* **What it does:** Hooks allow components to respond to state and prop changes, helping React update the DOM efficiently.
* **Why it matters:** React only updates the parts of the UI that have changed, improving performance. Regular JavaScript functions don’t integrate with React’s reconciliation system, so they can't trigger UI updates when state or props change.

Why Build Custom Hooks?

Custom hooks help keep our code organized, reusable, and maintainable. They allow us to extract and share logic across multiple components, avoiding code duplication. This adheres to the Single Responsibility Principle (SRP), which states that each part of our code should have one clear responsibility. By using custom hooks, we make the codebase cleaner, easier to understand, and easier to maintain.

Benefits of Custom Hooks

Reusability:Custom hooks are JavaScript functions that allow you to write logic once and reuse it in multiple components, promoting code reusability.

Readability:Using meaningful names for custom hooks clarifies their purpose, making the code easier to understand at a glance.

Modularity:React components should focus on rendering UI. By moving functional logic into custom hooks, you keep components simpler and more focused. Modularity involves breaking code into smaller, manageable pieces, each with a single responsibility.

Testability:Smaller, modular code is easier to test. Organizing logic into custom hooks allows you to write more effective and focused 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 adhere to the **Single Responsibility Principle (SRP)**:

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

### **Current Situation:**

The RestaurantMenu component violates the principle of modularity by handling two responsibilities:

1. Fetching data (API calls).
2. Displaying the data in the UI.

Ideally, the component should focus solely on rendering the UI and not on how the data is fetched or which API is being used. This concept is known as **data abstraction.**

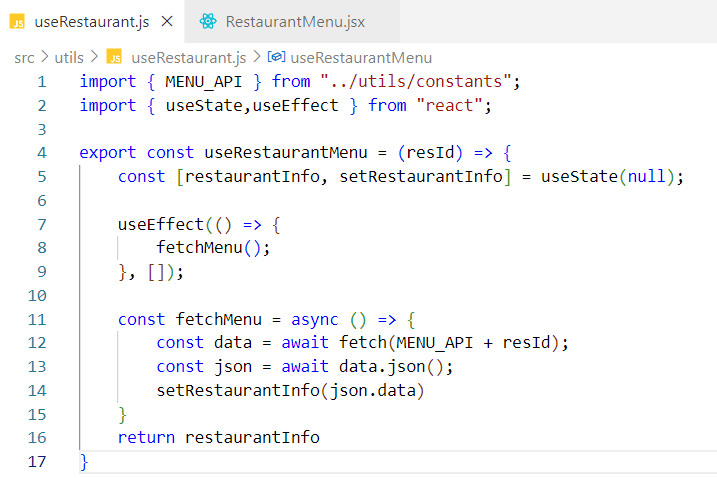
### ****Solution:****

To adhere to modularity, we can create a custom hook, useRestaurantMenu, to handle the data-fetching logic separately. This will simplify the component's role, allowing it to focus exclusively on displaying the data without worrying about the data-fetching process.

We’ll create the custom hook useRestaurantMenu in the **utils** folder, as it's a utility function. Afterward, we can use this hook within the RestaurantMenu component.

**Why This Works:**

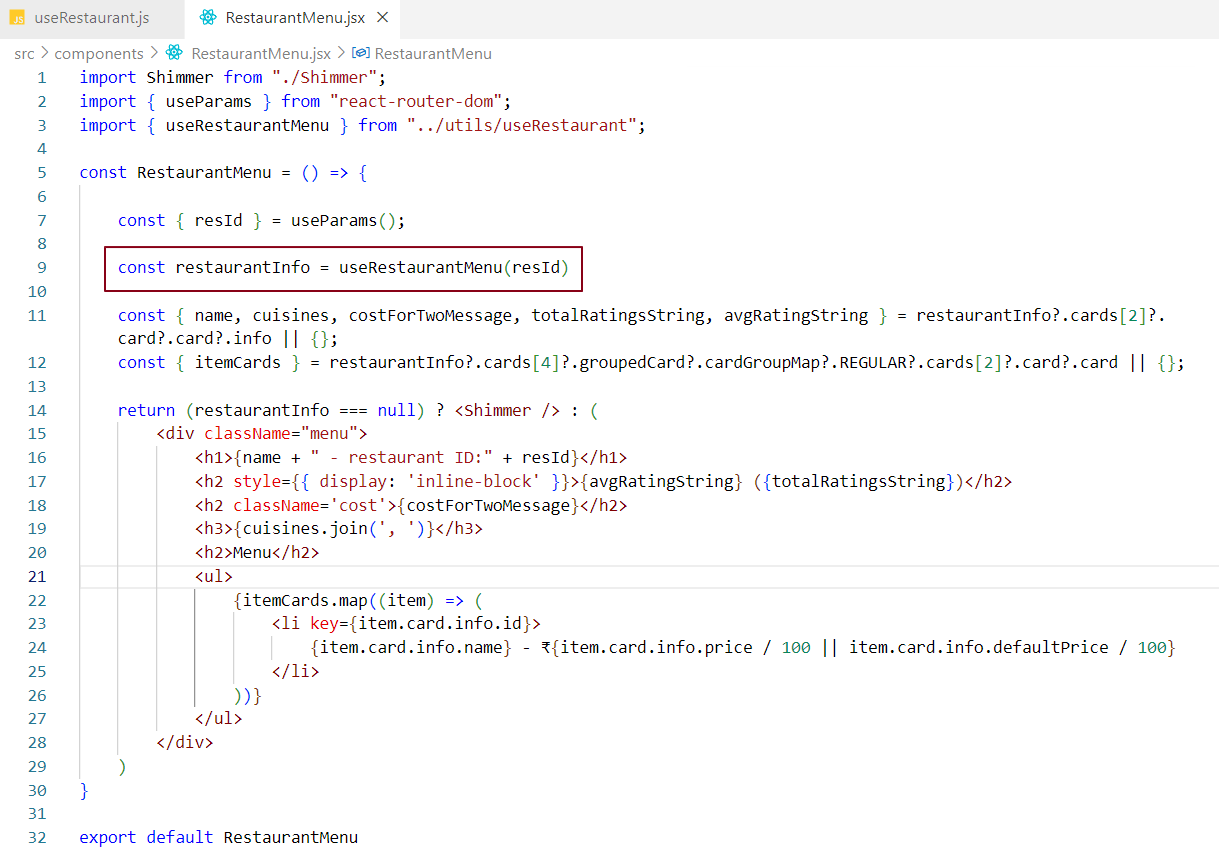
* **Data Abstraction:** The component no longer worries about how or where the data is fetched.
* **Modularity:** The logic for fetching data is separated into a custom hook, making the component cleaner and more focused.
* **Reusability:** The useRestaurantMenu hook can be reused in other components if needed.



### The useRestaurantMenu hook fetches and returns the menu data for a given resId, allowing the RestaurantMenu component to display the correct menu information.

### **Constants and Hook Usage**

* **Constant – resId:** A unique identifier used to fetch the specific restaurant's menu data.
* **Hook Usage:** The useRestaurantMenu hook is called inside the RestaurantMenu component, with resId passed as an argument to retrieve the correct menu data.



Refactoring for Single Responsibility Principle (SRP)

**Problem:** Initially, the RestaurantMenu component was responsible for both fetching data and rendering the UI, which violates the Single Responsibility Principle (SRP) — a core software design principle that states each component should have only one specific responsibility.

**Solution:** To adhere to SRP, we extracted the data-fetching logic into a custom hook called useRestaurantMenu, placed inside the helper module.

**Benefits:**

* ✅ The RestaurantMenu component now focuses purely on presentation.
* ✅ The useRestaurantMenu hook handles all API interaction.
* ✅ Improves modularity, readability, and maintainability of the codebase.

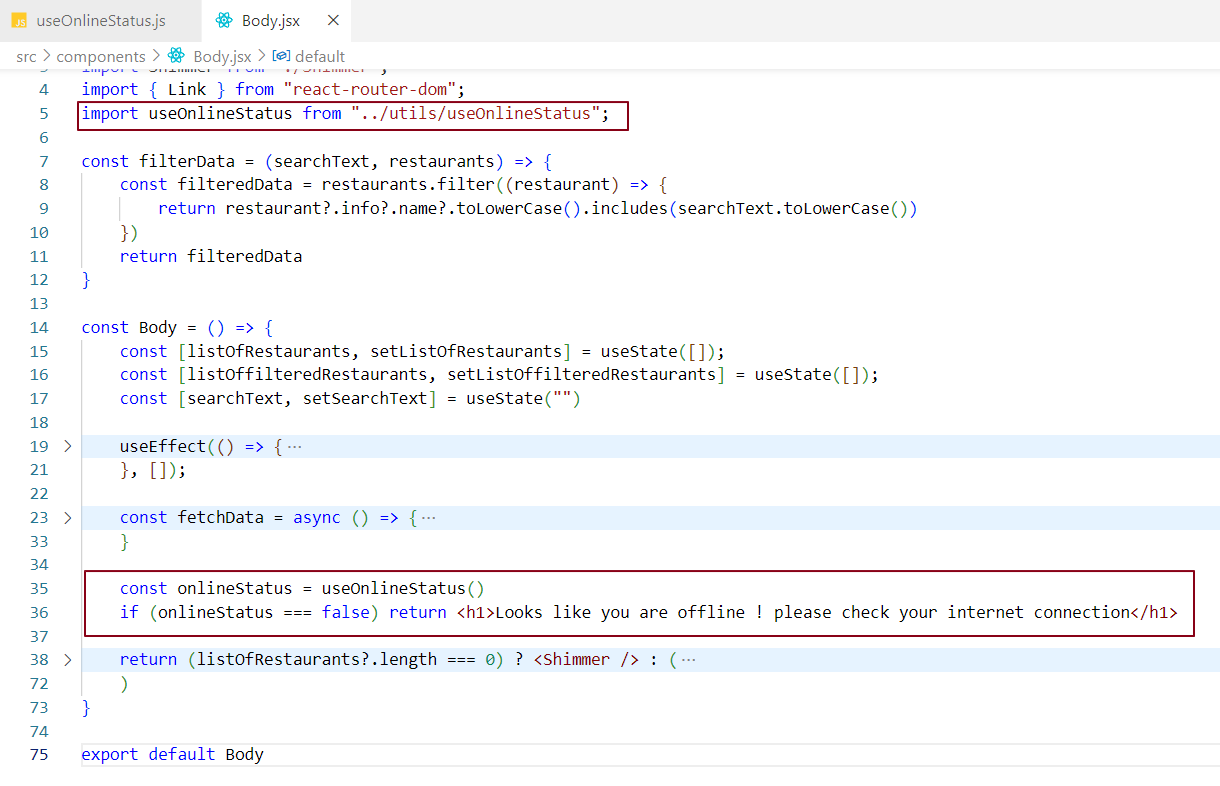
**Usage:** Whenever the RestaurantMenu component needs menu data, it simply imports and uses the custom hook.

🌐 Building Online and Offline Features

To improve user experience, let’s add a feature that checks the user’s internet connection and responds accordingly:

* If the user is offline: We’ll display a friendly message like: "You are offline. Please check your internet connection."
* If the user is online: We’ll go ahead and fetch and display the actual data as normal.

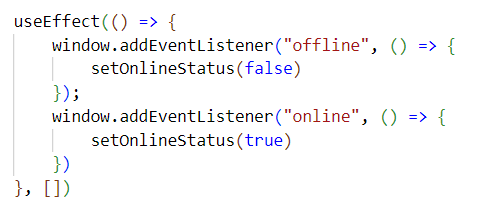




### Event Handling in useOnlineStatus Hook

**Adding Event Listeners:**

* The window.addEventListener is crucial for detecting the browser's online/offline status.
* The "offline" event triggers when the browser loses internet connectivity, while the "online" event triggers when the connectivity is restored.
* These listeners ensure the component is aware of changes in the network status in real-time.



**Anonymous Functions:**

* The anonymous functions handle the logic of updating the onlineStatus state.
* When the "offline" event is fired, the function calls setOnlineStatus(false) to set the state to false, indicating no internet connection.
* Similarly, when the "online" event is fired, the function calls setOnlineStatus(true) to set the state to true, indicating the internet connection is back.

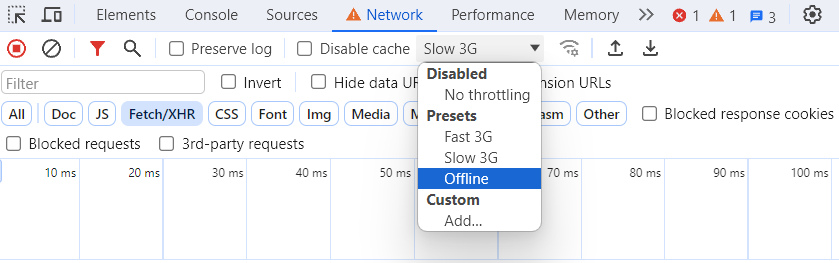
**State Updates:**

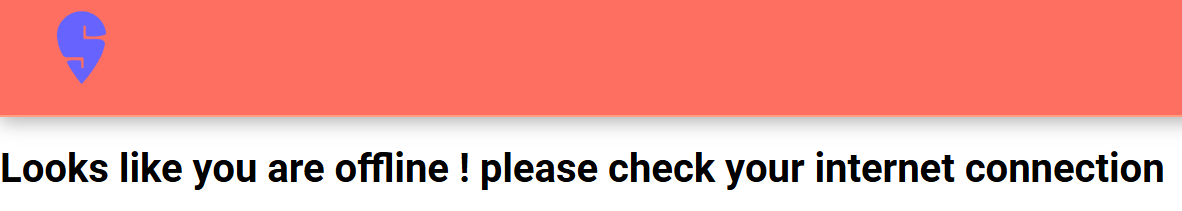
* The useState hook manages the onlineStatus state, which tracks whether the browser is online or offline.
* Whenever an event triggers a state update, the component re-renders to reflect the new network status.
* This is a reactive pattern where the component immediately responds to changes in the network state.

Simulating Offline and Online Modes in Chrome

**Offline** - No internet connection

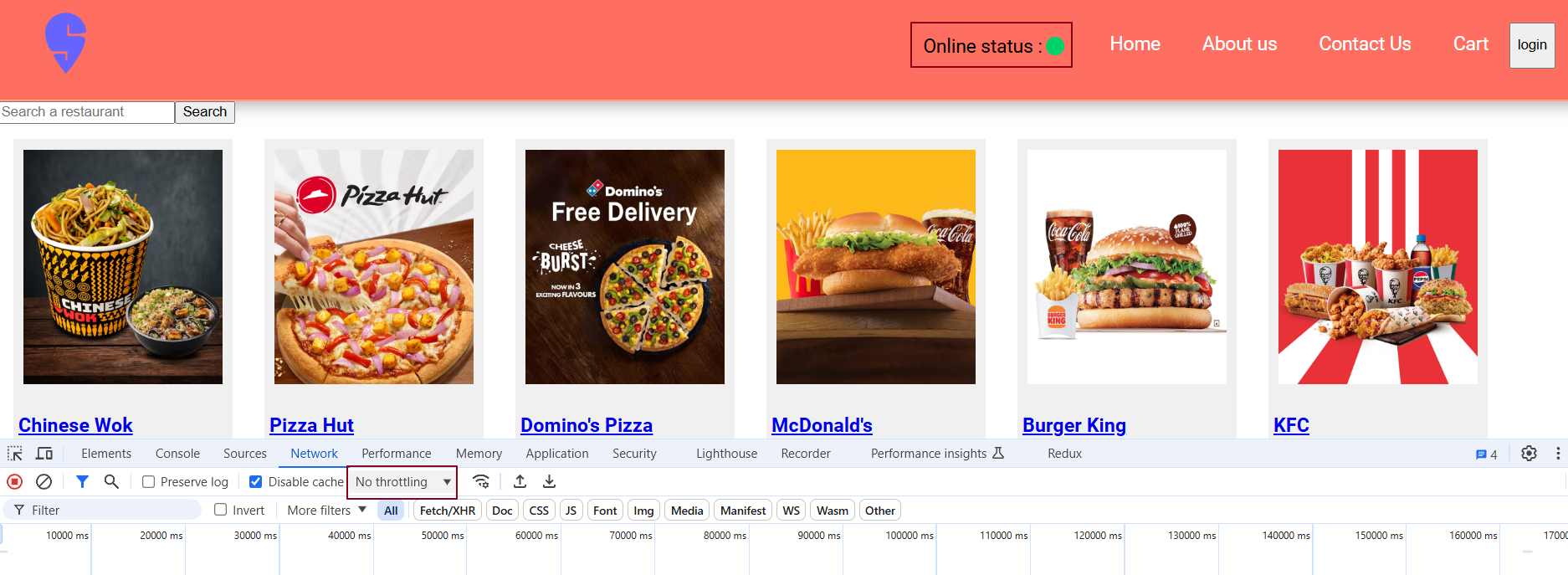
**No throttling** - Internet access restored.

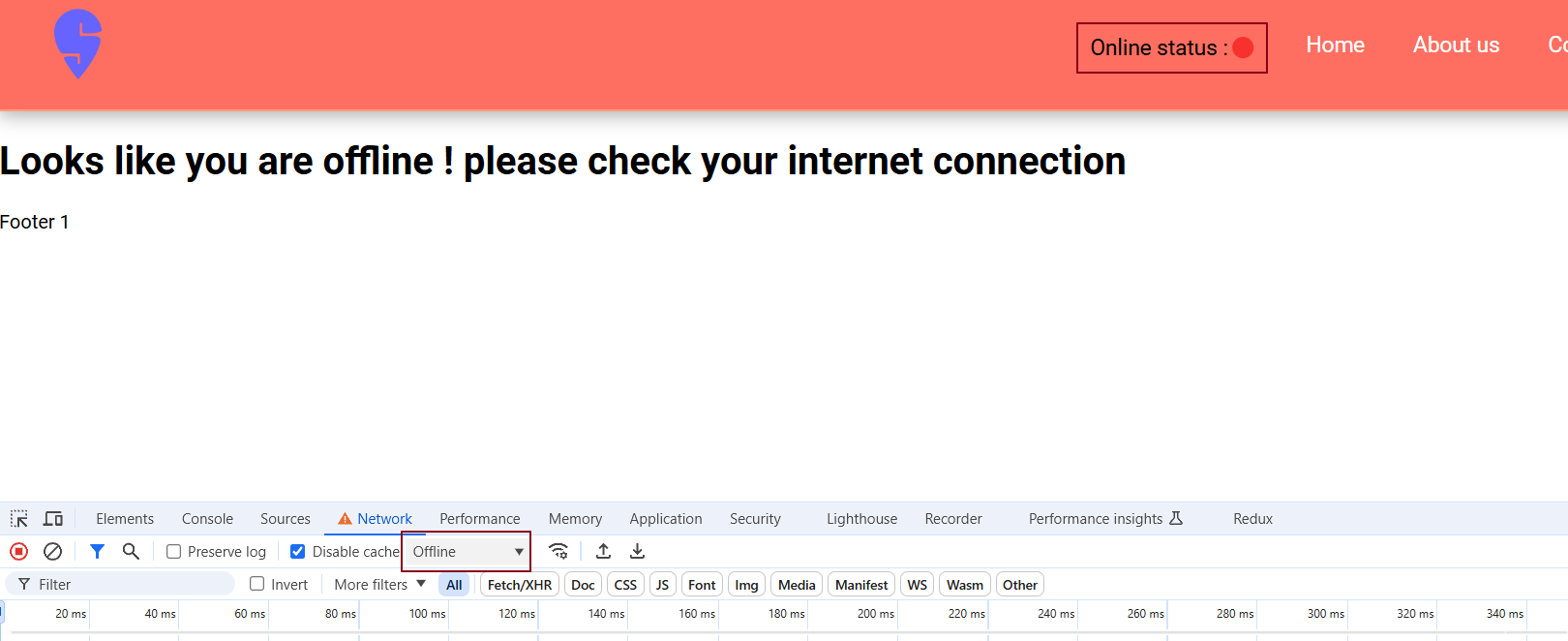




We can use this feature to enable chat notifications with color indicators: green for when the user is online and red for when the user is offline. Let’s implement this feature in our header component.



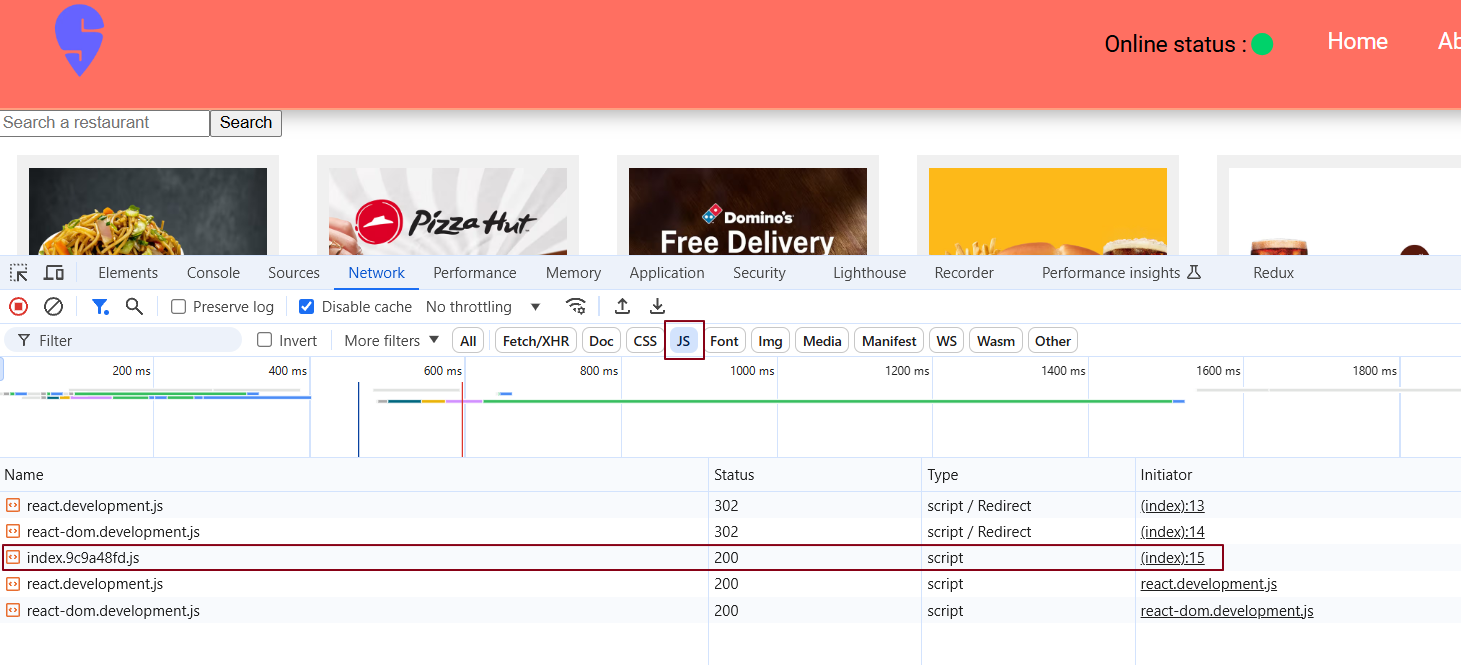




**Note:** It's a good practice to start custom hook names with 'use'. While not strictly required, some projects use a linter that enforces this naming convention. If we don’t follow it, the linter may throw an error.

Introduction to Bundle

In our application, Parcel bundles and minifies all of our code into a single JavaScript file called index.js. This file, located inside the dist folder, is what gets served to the browser. We can easily 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 slow down the application, as it forces all components to load at once during the first page load. To improve scalability, performance, and efficiency, we should split this large bundle into smaller chunks. These mini-bundles can load only when specific conditions are met, ensuring faster load times and a smoother user experience.

✅ What is Bundling?

Bundling is the process of combining multiple JavaScript (and other asset) files into a single or smaller set of optimized files. This helps in reducing the number of HTTP requests the browser makes, improving the application's load time and performance.

**During bundling -**

* Code splitting may occur to load only necessary parts of the code.
* Minification removes whitespace, comments, and shortens variable names.
* Tree shaking removes unused code.
* Transpilation (via Babel or TypeScript) may also be integrated in the process to ensure compatibility.

🛠️ Who Handles Bundling?

Bundling is handled by build tools called bundlers, such as:

* Webpack - Highly configurable and widely used.
* Parcel - Zero-config bundler, good for small to mid-sized projects.
* Vite - Modern and very fast, leverages ES modules.
* Rollup - Great for libraries and supports tree shaking out-of-the-box.
* esbuild - Extremely fast bundler and transpiler.

These tools handle merging files, resolving dependencies, and optimizing the final output for deployment.

How does bundling help improve a web application's load time and performance?

🔄 Without Bundling:Imagine a typical modern web app with many JavaScript files:



When a browser loads your app without bundling, it has to make one HTTP request for each file. That means 6 separate requests just for JS files. Add CSS, images, etc., and this number grows.

* More requests = More time to connect, download, parse.
* Each request has network latency and overhead (like headers).
* Browsers have a limit on concurrent HTTP requests per domain (usually ~6).

**✅** With Bundling:Bundlers like Webpack or Vite combine all those files into one single bundle.js (or a few split chunks). Now the browser:

* Makes 1 request instead of 6.
* Downloads everything it needs in one go.
* Reduces latency and overhead.
* Improves Time to Interactive (TTI).

**📈** Result: Fewer requests = faster load = better user experience.

### Chunking, Code Splitting, Dynamic Bundling, Lazy Loading, On-Demand Loading, and Dynamic Imports

These concepts are all related to optimizing the loading and performance of web applications. While they share similar goals, each has a unique role. Below is an explanation of each concept, how they improve performance, and how they relate to each other.

**1. Chunking**

Chunking is the process of dividing your application's code into smaller, manageable pieces (chunks). Instead of bundling the entire application code into a single large JavaScript file, chunking breaks it down into smaller files. This allows only the relevant parts of the code to be loaded at any given time, reducing the initial load time of the application.

**How it improves performance:**

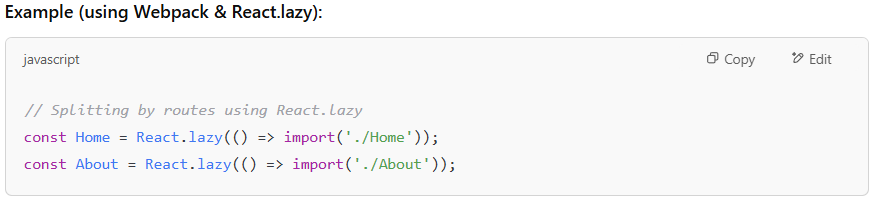
* **Reduced initial load time:** By splitting the code into chunks, only the necessary chunk is loaded initially, making the application load faster.
* **Smaller file sizes:** The chunks are smaller, so the browser can download and parse them more quickly.

**2. Code Splitting**

Code Splitting is essentially the same as chunking. It involves splitting your application's codebase into multiple smaller files (or chunks) that can be loaded dynamically. The primary goal of code splitting is to load only the parts of the application that are needed at a specific moment, instead of loading the entire application upfront.

**How it improves performance:**

* **Faster initial loading:** By loading only the essential code at first, you reduce the initial payload, improving the page’s load time.
* **On-demand loading:** Parts of the application are loaded only when required, reducing unnecessary downloads and improving efficiency.



**3.** **Dynamic Bundling**

Dynamic Bundling refers to dynamically creating bundles of code at runtime based on user interactions, specific routes, or other runtime conditions. Unlike static bundling, where all bundles are created ahead of time, dynamic bundling generates and loads code bundles when required during runtime.

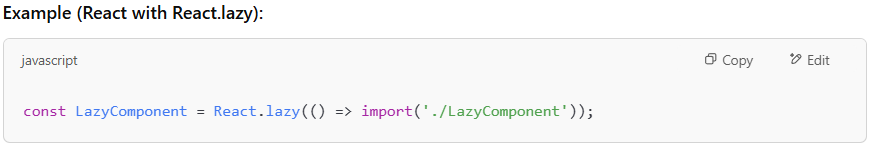
**How it improves performance:**

* **Bundles are created as needed:** Instead of loading all bundles upfront, dynamic bundling ensures only the required bundles are loaded, improving efficiency.
* **Flexibility in resource loading:** Resources are loaded based on the user’s behavior, providing a smoother experience by optimizing when and how resources are fetched.

**4. Lazy Loading**

**Lazy Loading** is a performance optimization technique where resources—like images, components, or JavaScript files—are loaded **only when they are actually needed**. Instead of fetching everything during the initial page load, non-essential content is **delayed** until the user **interacts with it** or it **enters the visible part of the screen** (the viewport).**How it improves performance:**

* **Defers non-essential content:** Non-critical resources like images or components are only loaded when necessary, reducing the initial page load time.
* **Improved perceived performance:** The user can start interacting with the page faster because only essential content is loaded first.

In this example, **LazyComponent** is only loaded when required, rather than upfront.

**5**. **On-Demand Loading**

On-Demand Loading is a broader concept that involves loading resources (JavaScript, images, etc.) only when they are requested by the user. Unlike lazy loading, which generally loads content automatically when it comes into view, on-demand loading is explicitly triggered by user actions, such as clicking a button or navigating to a new page.

**How it improves performance:**

* **Only loads what’s required:** Resources are only loaded when the user requests them, which reduces unnecessary network requests and improves initial load times.
* **Faster time-to-interactivity:** As the app doesn’t load unnecessary resources upfront, users can interact with the page faster.

**6. Dynamic Imports**

Dynamic Imports is a JavaScript feature (using import()) that allows you to load modules dynamically at runtime. Instead of bundling everything upfront, dynamic imports let you load specific modules when they are required, which is especially useful for implementing code splitting and lazy loading.

**How it improves performance:**

* **Modular code:** By splitting the code into smaller chunks that can be dynamically imported, the browser only loads the necessary code for the current state of the application.
* **Optimized loading:** Only the modules or components that are needed are loaded, reducing the overall JavaScript bundle size.



### ❓ Are they All the Same?

**No**, these techniques are closely related but have different roles in optimizing performance. Here's a quick breakdown:

* **Chunking** -Breaks the entire codebase into smaller, manageable pieces (chunks) to avoid loading everything at once.
* **Code Splitting** - Specifically splits code based on routes or features so only what's needed is loaded at a given time.
* **Dynamic Bundling -** Creates and loads code bundles at runtime based on user interaction or context.
* **Lazy Loading** - Loads non-essential resources like components or images only when they’re needed, not during the initial load.
* **On-Demand Loading** - Loads any resource (JavaScript, data, images, etc.) only when requested by the user.

### How do they improve performance together ?

When combined, these techniques create a highly efficient loading strategy for web applications:

* **Initial Load Speed:** Only the necessary resources are loaded initially, reducing the page load time and speeding up user interactions.
* **User Experience:** Users can interact with the application sooner, as only essential resources are loaded first. Non-essential resources are loaded later when needed.
* **Efficient Resource Management:** By reducing unnecessary network requests and minimizing initial payloads, the application optimizes resource utilization, leading to better performance.

### Is Bundling Good?

**Yes**, bundling is useful - but only up to a point. While it helps reduce the number of HTTP requests by grouping files together, bundling **everything into one large file** can actually hurt performance.

Instead, we should create **smaller, logical bundles**. This means loading only the bundle needed for a specific part of the app.

📱 **Example (MakeMyTrip - MMT):**

When users open MMT, they usually land on the **Flights** page. So, from a developer’s perspective, we only load the flight-related components first. There's no need to load **Train Booking** or **Hotel Booking** components right away.  
When a user later clicks on **Train Booking**, then and only then do we load that part of the app.

**🔁 in short,**

* Bundle based on user flow and features.
* For large apps, use multiple smaller bundles.
* For small apps, a single bundle is often fine.

This approach ensures faster load times and a smoother user experience.

Where do we implement On-Demand loading?

We use **on-demand loading** in situations where we don’t want to load a component or resource upfront. Instead, we load it **only when the user requests it or interacts with a specific feature.**

🧠 **Typical scenarios include:**

* Navigating to a new route/page e.g., going from Home to Profile.
* Clicking a button that opens a rarely used modal or dashboard.
* Expanding an accordion or section that shows additional data or graphs.
* Loading large libraries like charts or maps only when needed.

✅ **Why use it?**

To keep the initial load light and improve performance by avoiding unnecessary downloads.

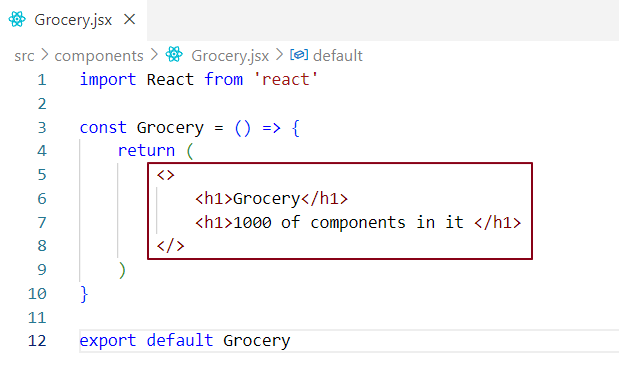


Why should we use on-demand loading?

The on-demand loading strategy ensures that a component is only loaded when it is actually needed by the user, rather than loading everything upfront. This is particularly beneficial for features or pages that are not immediately required when the application starts. For example, a user might only need to view a specific page or open a feature after interacting with the application (like clicking a button to navigate to a new page or opening a modal). By loading these components only when necessary, we improve the application's performance and reduce initial loading time.

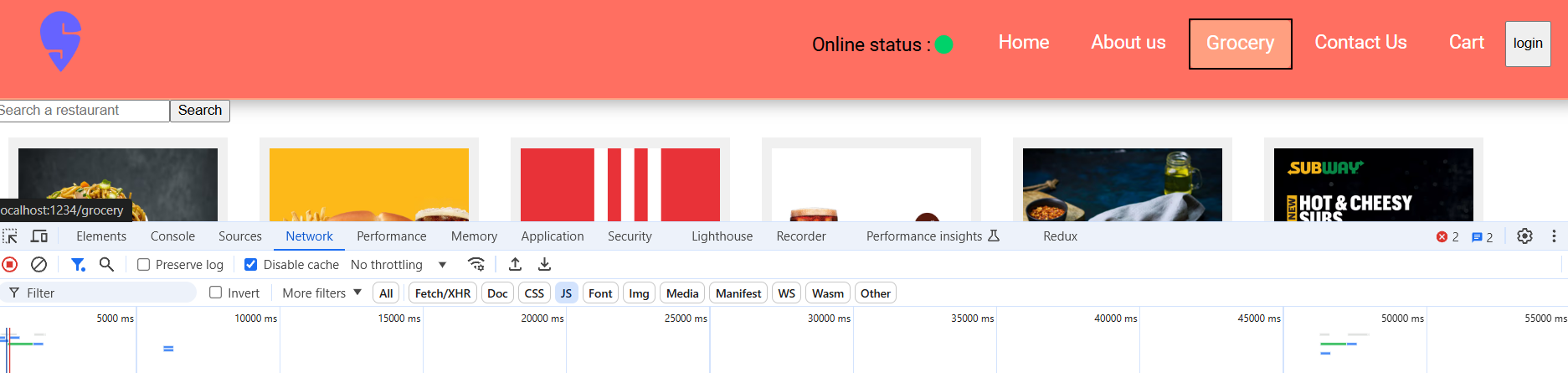
Example: On-Demand Loading with a "Grocery" Component in React

Let's say you have a **Grocery** component that should only be loaded when the user navigates to the grocery page.

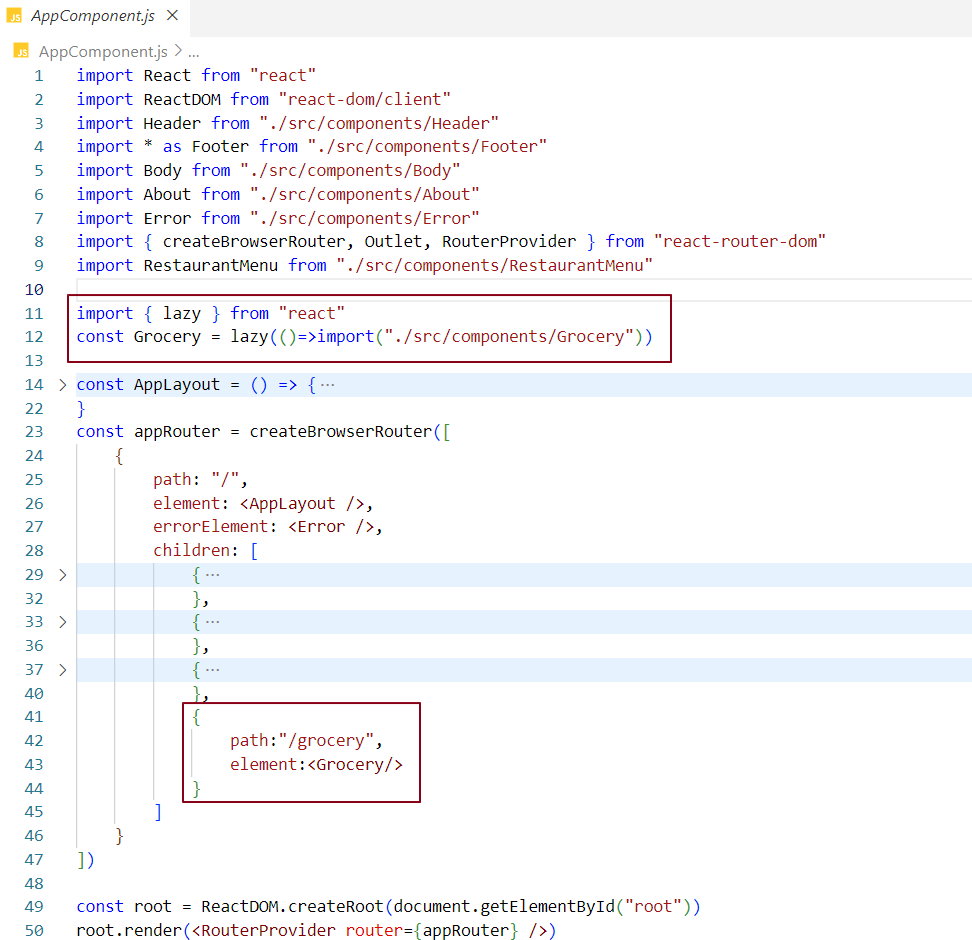


This code is still included in our main bundle, index.js. However, this approach is inefficient because the single file contains hundreds of components, all of which are loaded into the index.js bundle. As a result, the bundle size increases, leading to performance issues.

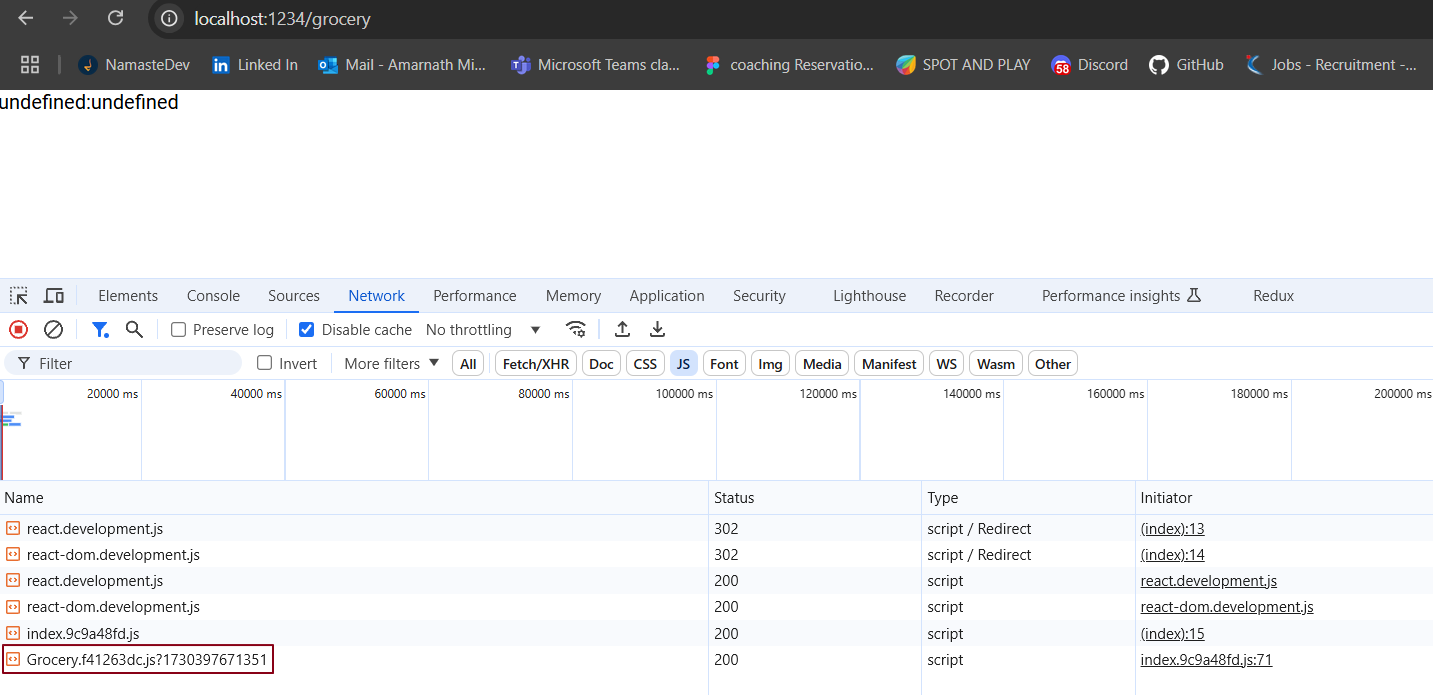
Let's dynamically load the Grocery bundle when the user clicks on the Grocery link in the header.







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



Understanding React.lazy

**1. Purpose of React.lazy**

The React.lazy function is used for codesplitting, allowing components to be loaded only when they are needed. This strategy helps to improvetheperformance of your application by reducing the initial load time, as only the essential components are loaded upfront. Non-essential components are loaded dynamically, only when the user interacts with the app or navigates to specific sections, thus enhancing the overall efficiency of your application.

### 2. Callback Function in React.lazy

When you use **React.lazy**, you pass a **callback function** as an argument. This function contains an import() statement that **dynamically loads the component** when it's required. Let's break it down.

#### What is the callback function doing?

#### Answer - The callback function is essentially a **trigger** that tells React when to load the component. The import() inside this function is the dynamic import that tells React to fetch the component from the specified path.

#### Why is this necessary?

#### Answer - By using a callback function, react doesn't load the component immediately. Instead, it waits until the component is needed - for example, when a user navigates to a specific route or interacts with a button. This approach splits the code and boosts performance by loading only the required components when needed, rather than all at once.

#### Example:

const Grocery = React.lazy(() => import('./Grocery'));

* React.lazy takes the **callback function** () => import('./Grocery').
* import('./Grocery') is a **dynamic import** that returns a **Promise**. When this promise resolves, the Grocery component is available to be rendered.

#### How it works ?

1. Initially, react doesn't load the Grocery component because it's inside the import() call.
2. When the component is needed (for example, when a user clicks a button or navigates to a page where this component is used), the **callback function** is executed.
3. The **import** triggers the loading of the Grocery component.
4. While the component is loading, you can show a **loading indicator** (fallback UI).
5. Once the component is loaded, react renders it.

### How React.lazy Works Behind the Scenes ?

* **Callback function** = a trigger to start loading the component.
* **Dynamic import (**import()**)** = tells React to load the component when it's actually needed.

So, the callback function acts as a delayed trigger to fetch the component dynamically. React executes this function only when the component is actually needed, saving time and improving performance by avoiding its initial load.

**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. This means that the JavaScript engine will try to fetch the component or module only when it's needed, rather than upfront.

Once the component is successfully loaded, the Promise resolves, which triggers React to render the lazy-loaded component. This ensures that the component isn't included in the initial bundle, improving the initial load time and enabling code-splitting for more efficient resource management.

**4. How It Works Together**

When you use React.lazy, it takes a callback function containing the import statement. This function is executed only when the component is actually needed—like when a user navigates to a specific page or clicks a button. For example, if the user clicks the 'Grocery' link, react triggers the callback, which then dynamically imports and loads the Grocery component.

While a component is being fetched using React.lazy, you can display a fallback UI (like a loading spinner or placeholder) to inform the user that loading is in progress. Once the component loads successfully, it replaces the fallback and is rendered. This approach defers component loading until needed, enhancing both performance and user experience.

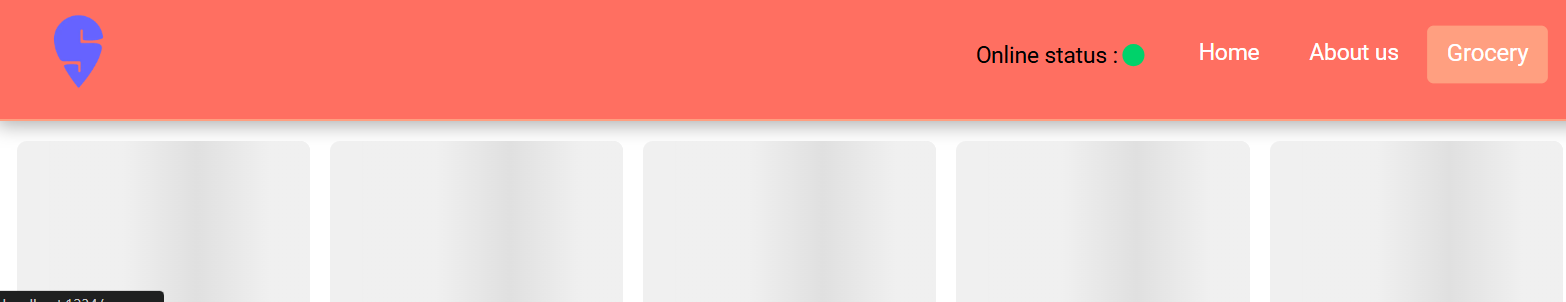
What could be causing the "undefined:undefined" error when navigating to the Grocery component in our application?

When we load the Grocery component, the component bundle takes time to load. Meanwhile, react attempts to render the Grocery component, but it doesn't exist yet because the bundler hasn't loaded it. In this process, react suspends the rendering. This causes the error message to appear in the UI. During this time, err.status and err.statusText are undefined. Since we're loading the component on demand, react tries to suspend it. The Grocery card component is a suspense component because its existence in the UI depends on the bundle being loaded. It will only load once the bundle for this component is available.

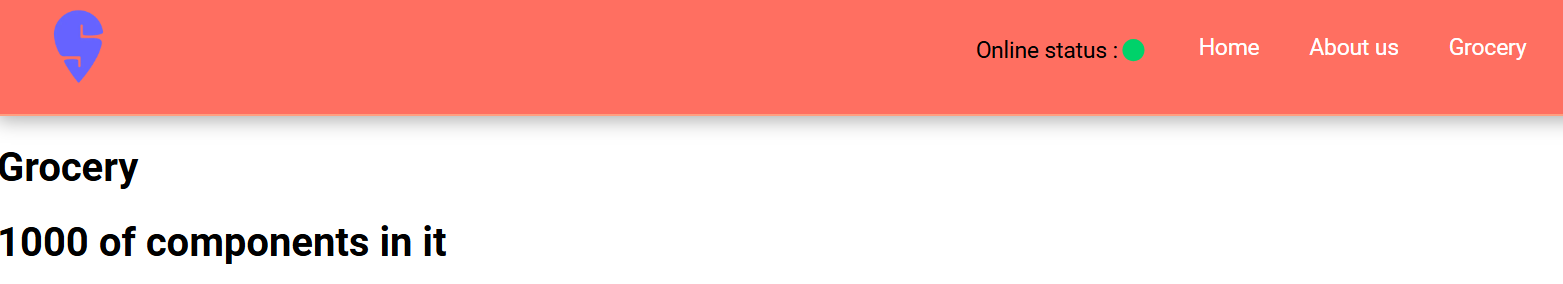
What is suspend?

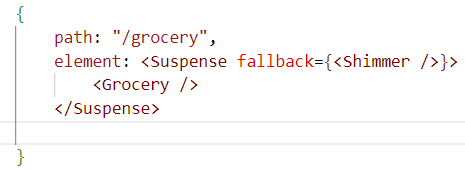
<Suspense> is a React component that pauses the rendering of lazy-loaded components until the corresponding bundle for that component is fully available. It enables you to show a fallback UI (like a loading spinner or shimmer effect) while waiting for the component to load.

For example, when you click on the Grocery link, the bundle for the Grocery component isn't immediately available. In this case, we've provided a Shimmer UI inside the fallback property of the <Suspense> tag. This means the Shimmer UI will be displayed until the Grocery component's bundle is fully loaded.



Once the bundle has loaded, react will retrieve the Grocery component and display it in the UI.

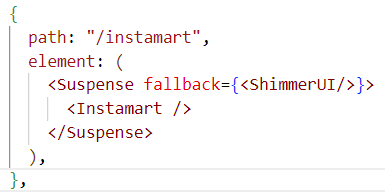




What Is fallback in React?

In React, the fallback is a prop of the <Suspense> component that specifies the UI to display while a lazy-loaded component's bundle is being fetched. When the bundle is not loaded, react suspends the rendering of the component until it becomes available. During this time, to improve the user experience, we show a placeholder or loading UI, like a shimmer effect, inside the fallback prop.

For example, you can use the fallback prop to display a shimmer UI or any other placeholder component while waiting for the actual component to load. You can even write JSX directly inside the fallback using curly braces {}.



In this case, the Shimmer component will be displayed until the Grocery component is ready to be rendered.

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

**The Problem -** When you use React.lazy() inside a parent component’s function, the lazy-loaded component gets re-initialized on every render. This can cause unnecessary performance issues because React may attempt to re-fetch the component’s bundle even if it has already been loaded.

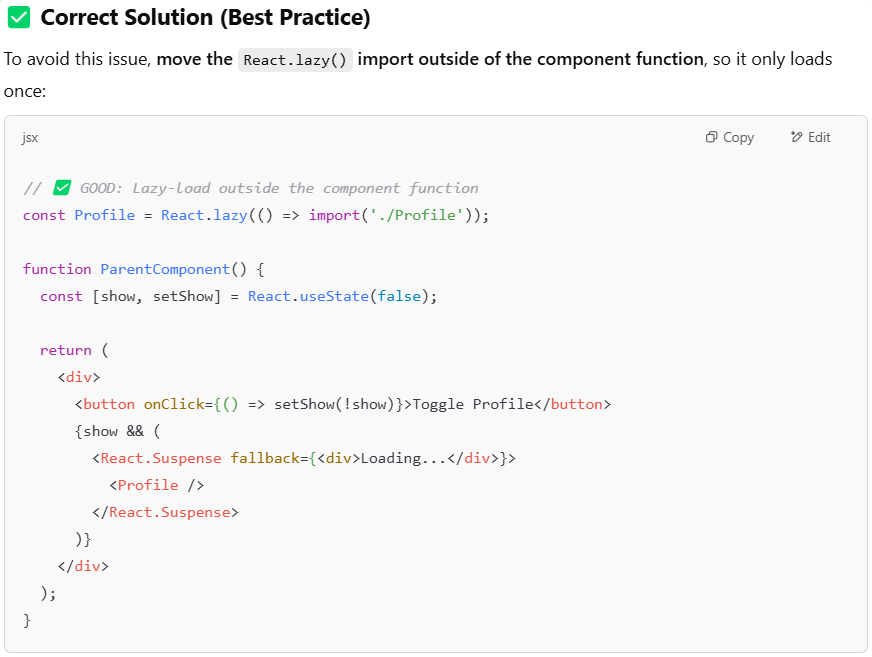
### ****Why this Happens ?****

* React.lazy() is a **dynamic import** that tells React to load a component asynchronously when needed.
* If React.lazy() is called **inside the component function**, it gets executed **every time the parent component renders** (e.g., due to state or prop changes).
* This means React **treats the component as a new import each time,** leading to:
  + Unnecessary re-imports
  + Performance degradation
  + Potential flickering issues when re-rendering

### 

### Why is this a problem?

* Every time setShow toggles, ParentComponent re-renders.
* Because React.lazy() is inside the function, it **gets executed again** on every render.
* Even though React caches the import, putting React.lazy() inside a component causes it to run again every time, which wastes resources.



**Why does this work?**

* React.lazy() is defined **only once** when the module is loaded.
* The Profile component is now **cached**, so it doesn’t get reloaded unnecessarily.
* Toggling visibility only **shows or hides** the component without affecting performance.

### ****Key Takeaways****

✔ **Don’t define** React.lazy() **inside the component function -** it will reinitialize on every render.  
✔ **Always declare lazy-loaded components outside** the component body to ensure they only load once.  
✔ This approach **improves performance,** prevents redundant imports, and optimizes resource usage.