

# Episode-09 | Optimising Our App



Please make sure to follow along with the whole "Namaste React" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React development.



I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-09** first. Understanding what "Akshay" shares in the video will make these notes way easier to understand.

## Q ) When and why do we need `lazy()` ?

The `lazy()` function is a feature in React that allows us to `load` components dynamically, or lazily, only when they are needed . This can be beneficial for improving the performance and load times of our web application, especially if it contains a large number of components or if some components are rarely used. Here's when and why we might need to use `lazy()`:

?? *Code Splitting and Reducing Initial Bundle Size* ? In large React applications, bundling all components into a single JavaScript file can result in a large initial bundle size. This can lead to slower load times for users. By using `lazy()`, we can split our code into smaller, more manageable chunks. These chunks are loaded on-demand, reducing the initial bundle size and improving the time it takes for our application to load.

?? *Improved Performance* ? Lazy loading can lead to better application performance. Components that are only loaded when needed reduce the amount of code that needs to be executed

during the initial page load, which can lead to faster rendering times and a smoother user experience.

?? *Faster Initial Load* ? When we use `lazy()`, only the essential code is loaded initially. Less code to parse and execute means that our application can start up faster, especially in scenarios where not all components are used right away.

?? *Better User Experience* ? By deferring the loading of components until they are needed, we can provide a more responsive user experience. Users don't have to wait for unnecessary components to load, and they can interact with the parts of the application that are immediately visible.

?? *Reducing Browser Caching Overhead* ? Smaller initial bundles produced by `lazy()` can also benefit from browser caching. Since components are loaded as separate chunks, once loaded, they are less likely to change frequently. This can result in a better caching strategy and faster subsequent visits to our site for returning users.

?? *Optimizing Mobile Performance* ? On mobile devices with limited bandwidth and processing power, lazy loading is even more important. Smaller initial bundles can make our application more accessible and usable on mobile devices.



Here's an example of how to use `lazy()` to load a component dynamically

```
import React, { lazy, Suspense } from 'react';

const LazyComponent = lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

In this example, the `LazyComponent` is only loaded when it is rendered.

The `Suspense` component allows us to specify a loading indicator while the component is being loaded. This way, we can ensure a smooth user experience even during the asynchronous loading process.

In summary, we need to use `lazy()` when we want to optimize the performance and user experience of our React application by reducing the initial bundle size and deferring the loading of components until they are needed. This is particularly beneficial in large applications or when targeting slower connections and devices.

## Q ) What is `suspense` ?

In React, `Suspense` is a feature that allows us to declaratively manage asynchronous data fetching and code-splitting in our applications. It is primarily used in combination with the `lazy()` function for dynamic imports and with the `React.lazy()` component to improve the user experience when loading data or components asynchronously.

Here are the main aspects and use cases of `Suspense`:

**Data Fetching** 📄 Suspense can be used to handle the loading of asynchronous data, such as data from an API. It provides a way to specify a fallback UI (e.g., a loading spinner or a message) that is displayed while the data is being fetched. This is especially useful for making our application more user-friendly and responsive.

**Code Splitting** 📄 When used with `lazy()` or `React.lazy()`, Suspense can manage the loading of code-split components. We can specify a fallback component or loading indicator to display while the component is being loaded. This helps in reducing the initial bundle size and improving the application's performance.

**Error Handling** 📄 Suspense can also handle errors that might occur during data fetching or code splitting. We can specify how to render an error component or message in case an error occurs during the asynchronous operation.



Here's a basic example of using Suspense for data fetching:

```
import React, { Suspense } from 'react';

const fetchData = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data fetched!");
    }, 2000);
  });
};
```

```
function DataFetchingComponent() {
  const data = fetchData();

  return (
    <div>
      <Suspense fallback={<div>Loading data...</div>}>
        <AsyncDataComponent data={data} />
      </Suspense>
    </div>
  );
}

function AsyncDataComponent({ data }) {
  return <div>{data}</div>;
}
```

In this example, when the `DataFetchingComponent` is rendered, it starts fetching data asynchronously. The `Suspense` component wraps the `AsyncDataComponent`, specifying a fallback UI to display while the data is being fetched.

`Suspense` can also handle errors by using an error boundary. If an error occurs during data fetching or code-split component loading, we can catch and handle the error gracefully.

While `Suspense` simplifies managing asynchronous operations and loading states in our React application, it's essential to be aware of the version of React we are using. `Suspense` for data fetching was introduced in React 18 and may have different usage patterns compared to `Suspense` for code-splitting, which has been available since React 16.6. Depending on the version of React, we might need to adjust our code accordingly.

**Q ) Why do we get this error: A component was suspended while responding to synchronous input. This will cause the UI to be replaced with a loading indicator. To fix this, updates that suspend should**

## be wrapped with start transition ? How does suspense fix this error?

The error message you provided, "A component was suspended while responding to synchronous input. This will cause the UI to be replaced with a loading indicator. To fix this, updates that suspend should be wrapped with start transition," is related to React's Suspense feature and is typically encountered in asynchronous contexts where components are fetching data or handling code splitting.

To understand this error and how to fix it, you need to know a bit about how Suspense works and why it's important. Suspense is used to manage asynchronous data fetching and code-splitting, allowing you to display a loading indicator while the data or code is being fetched. When React encounters a Suspense boundary (created using `<Suspense>`), it knows that there might be a delay in rendering, and it can handle that situation gracefully.

The error message you received is telling you that a component that was responding to synchronous input (meaning it's not supposed to be waiting for anything) encountered a suspension. This should not happen because Suspense is primarily designed to handle asynchronous operations, and you generally don't want to introduce delays in the rendering of synchronous user interactions.



Here's how to fix this error:

The error message you provided, "A component was suspended while responding to synchronous input. This will cause the UI to be replaced with a loading indicator. To fix this, updates that suspend should be wrapped with start transition," is related to React's Suspense feature and is typically encountered in asynchronous contexts where components are fetching data or handling code splitting.

To understand this error and how to fix it, you need to know a bit about how Suspense works and why it's important. Suspense is used to manage asynchronous data fetching and code-splitting, allowing you to display a loading indicator while the data or code is being fetched.

When React encounters a Suspense boundary (created using `<Suspense>`), it knows that there might be a delay in rendering, and it can handle that situation gracefully.

The error message you received is telling you that a component that was responding to synchronous input (meaning it's not supposed to be waiting for anything) encountered a suspension. This should not happen because Suspense is primarily designed to handle asynchronous operations, and you generally don't want to introduce delays in the rendering of synchronous user interactions.



Here's how to fix this error:

**Identify the Issue** You should identify which part of your code is causing the synchronous component to suspend. This could be due to a network request, a dynamic import of a component, or another asynchronous operation.

**Wrap Asynchronous Code** Ensure that the asynchronous code, which might suspend, is wrapped within a Suspense boundary (using `<Suspense>`) and that you provide a fallback UI to display while waiting for the operation to complete.



Here's an example of how to properly structure your code:

```
import React, { Suspense, lazy } from 'react';

const AsyncComponent = lazy(() => import('./AsyncComponent'));

function App() {
  // Synchronous code
  return (
    <div>
```

```

    <h1>Your App</h1>
    <Suspense fallback={<div>Loading...</div>}>
      <AsyncComponent />
    </Suspense>
  </div>
);
}

export default App;

```

In this example, the AsyncComponent is loaded asynchronously, and it is wrapped within a boundary. The fallback attribute specifies what to display while the component is loading. The rest of the application, which is synchronous, doesn't get affected and will continue to respond to user input without unnecessary delays.

Suspense helps in maintaining a smooth and responsive user experience by handling asynchronous operations gracefully and ensuring that synchronous interactions are not interrupted by loading indicators.

## Q ) Advantages and Disadvantages of using this code splitting pattern ?

Code splitting is a technique used to break down a large monolithic JavaScript bundle into smaller, more manageable pieces, which can be loaded on-demand. This pattern has several advantages and some potential disadvantages, depending on how it's implemented and the specific use case. Let's explore the advantages and disadvantages of using code splitting:

### Advantages:

**Faster Initial Load Time** Smaller initial bundles result in faster load times for your web application. Users can start interacting with the application sooner because they don't have to download unnecessary code.



**Improved Performance** ? Code splitting can lead to better performance, as smaller bundles can be parsed and executed more quickly. This can reduce the time it takes to render the initial page and improve the overall responsiveness of the application.

**Optimized Resource Usage** ? Code splitting helps optimize resource usage. Components or features that are rarely used may never be loaded unless needed. This conserves bandwidth and memory, making your application more efficient.

**Enhanced Caching** ? Smaller bundles can benefit from browser caching. Since they are less likely to change frequently, browsers can cache them, resulting in faster subsequent visits for returning users.

**Simpler Maintenance** ? Smaller bundles are easier to maintain. When you make updates to specific parts of your application, you can be more confident that you won't introduce unexpected issues in unrelated components.

**Better Mobile Performance** ? On mobile devices with limited bandwidth and processing power, code splitting can significantly enhance the user experience by reducing the amount of data that needs to be loaded and processed.

### **Disadvantages:**

**Complex Configuration** ? Setting up code splitting and configuring it correctly can be complex, especially in large applications. You may need to make adjustments to your build tools and bundler settings.

**Initial Loading Delay** ? When a component is loaded on-demand, there may be a slight delay the first time it is needed, which can impact user perception of your application's speed. However, this delay is usually minimal, and it's often a tradeoff for the benefits of code splitting.

**Asynchronous Loading** ? Handling asynchronous loading and rendering of components requires careful design to ensure a seamless user experience. You need to consider scenarios such as loading indicators and error handling.

**Route-Based Splitting** ? To maximize the benefits of code splitting, you should implement it on a route or feature basis. This can lead to a more granular structure, but it may require some restructuring of your application.

**Tool and Framework Support** ? Not all frameworks and libraries have built-in support for code splitting. You may need to rely on specific tools and configurations, which can vary depending on your stack.

**Testing Complexity** ? Testing code-split components can be more challenging because you must ensure that they load correctly in different scenarios and that they don't introduce unexpected issues.

In summary, code splitting is a valuable technique for improving the performance and user experience of your web applications, but it comes with some complexities and trade-offs. The advantages, especially in terms of faster initial load times and optimized resource usage, often outweigh the disadvantages, which can be mitigated with careful implementation and testing.

## Q ) When do we and why do we need suspense ?

React Suspense is a feature introduced in React to help manage asynchronous operations, such as data fetching and code splitting, in a more declarative and user-friendly manner. You need to use Suspense in your React application when you want to:

**Improve User Experience** ? Suspense helps in providing a better user experience by managing the loading state of asynchronous operations. Instead of showing loading spinners or handling loading states manually, Suspense allows you to specify fallback UI components to be displayed while data is being fetched or code is being loaded.

**Optimize Performance** ? Suspense, in combination with code splitting, can significantly improve the performance of your application. It allows you to load code and data only when it's needed, reducing the initial bundle size and making your application faster to load.

**Simplify Code** ? Suspense simplifies your code by providing a more declarative way to handle asynchronous operations. It reduces the need for complex state management and error handling for data fetching or code splitting.

**Avoid Callback Hell** ? In traditional async patterns, managing multiple asynchronous operations can lead to "callback hell" or nested promises. Suspense provides a more structured way to handle multiple asynchronous operations concurrently.

**Error Handling** ? Suspense is also useful for handling errors gracefully. You can specify how to render error components or messages when an error occurs during data fetching or code splitting, making it easier to provide a clear userfacing error message.

Here's a brief overview of when and why you might need Suspense in different scenarios:

**Data Fetching** ? Use Suspense for data fetching when you want to make your application more responsive and provide a smooth loading experience for data-driven components. It simplifies the management of loading states and error handling.

**Code Splitting** ? Use Suspense for code splitting when you want to improve your application's initial load time and performance. It allows you to load parts of your application on-demand, which can lead to faster rendering times and better resource usage. Concurrent Mode:

React Suspense is particularly valuable when using React Concurrent Mode. Concurrent Mode leverages Suspense to handle asynchronous rendering and data fetching more concurrently and efficiently.

In summary, you need to use Suspense in React when you want to create a more responsive, efficient, and user-friendly application by simplifying the handling of asynchronous operations and providing a better user experience during data fetching and code splitting.

## Self-Note –

In this episode we will see how we can write code in a better way, how we can optimize our app, how we can make our app performant, how we can make our app very fast, how we can make our app lightweight so we can load the webpage very fast.

But first we will learn about the single responsible principle –

It means suppose you have a function, class or you have any single identity of your code that should have single responsibility. All the components that we have created are the functions and each component has the different function so according to the principle each of the components should have single responsibility. What do you mean by single responsibility? Suppose we have RestaurantMenu, the only job of the RestaurantMenu should have to be display Restaurant Menu so similarly each component that we have created we should give it single responsibility and we should not load of things in the component if we do loads of things in the single components needs to break it down in the different component.

So this is the good way of maintaining in the modular fashion.

Modularity means you break down the code in small, small modules so your code becomes more maintainable and more testable. So, we can catch the bug very easily.

If you follow the single responsible principle, you get feature of reusability.  
So, your code become more Reusable, Maintainable, Testable.

## Custom hook -

We can create the own custom hooks

How we can use the custom hooks to make the code more modular, to abstract the extra responsibility to the components to the different hook.

What are hooks?

It is the utility function (normal function)

We will just abstract / takeout some responsibility from component and extract inside a hook so that our hook and our component become more modular and become more readable.

Create the own custom hook and how we can optimize when we write own custom hook –

RestaurantMenu file –

Why we need to create the own custom hook and why did I choose

Creating the custom hook is not the mandatory thing but it's a very good thing that will make your code more readable, more your code more modular, make your code reusable.

There are 2 major responsibilities of RestaurantMenu component first responsibility fetching the data and second is displaying the data on UI

Don't you think that RestaurantMenu should only be concern about displaying the data it should not worry about where the data coming from, how the data coming from , what is the API being called.

Let's take the example of useParams we had a useParams hook this gives us the resId and we don't know what the code is written behind the scene. useParams is just the utility function and this utility function we don't know how this useParams function getting the parameters. There would be some code written for useParams that will go and check the Url, check what is the resId over here and abstract the resId and give it to you over here in the variable resId we are not bother about the implementation about the hook.

Similarly, what we are trying to achieve is can I abstract my fetching data also so I will just get the data. What I am trying to over here is I am just fetching the menu so somehow I just want a hook , custom hook which is useRestaurantMenu and this gives us resInfo. Suppose we had a hook and we do not have to worry about how the custom hook fetching the data.

#### Remove the login from RestaurantMenu

So, what we are trying to is I am trying to abstract takeout the fetch data logic and put inside this hook. I will create the useRestaurantMenu hook and this RestaurantMenu() component not worry about how to fetch the data, it just worry about this is the resInfo I have to get my restaurant data inside it and I just want to display it now. It doesn't have to manage its state now. It somehow magically gets access it resInfo.

What we are going to is **we will create the own custom hook that will fetch the data and give it to RestaurantMenu.**

ResturantMenu have just single responsibility it is just concern about displaying the restaurant menu on to the UI and code become cleaner.

So, now let us just see how we can create the own custom hook useRestaurantMenu to fetch the data of restaurant and give it us to the component ResturantMenu

So ResturantMenu component doesn't worry about how we are fetching the data , it will just concern about I have got my resId and with this resId for the particular restaurant I just want my restaurant menu. So I will just give my resId to my custom hook and somehow magically I want restaurant information into my component. ResturantMenu doesn't worry about the implementation of this hook, implement how we will be fetching the data. ResturantMenu doesn't worry about it. Single responsibility of ResturantMenu get the data and display it. How to get the data abstract it.

Write the own ResturantMenu hook –

Custom hook is the utility function they are like helper functions

So when there helper functions so generally the best way, best place to create to helper functions are in utils. Will create the custom hooks inside utils and always prefer to create the separate file for a separate hook. This is the good convention/ pattern to follow.

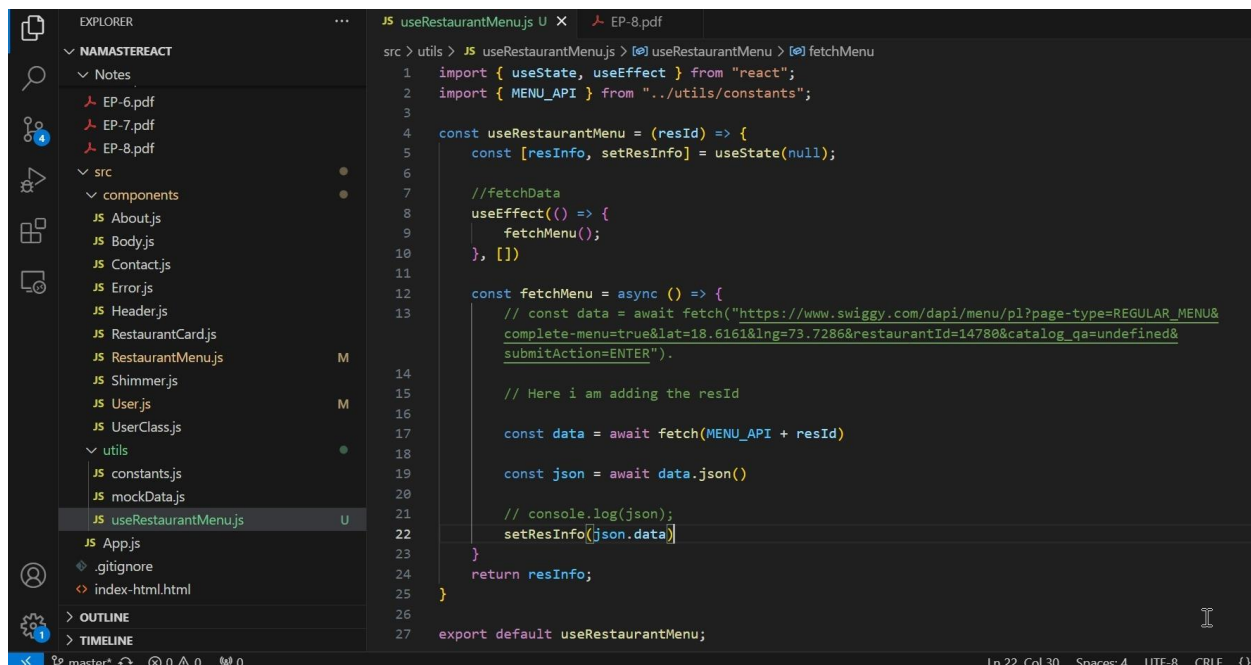
Always create the hook with the word “use”

Why we do that because that is the way to react to know if the function name starting from “use” that's means it a hook, react understand well so we use that.

Hook is the normal JavaScript function, what is the job of this useRestaurantMenu hook, it takes the resId and its job to return the restaurant information(resInfo)

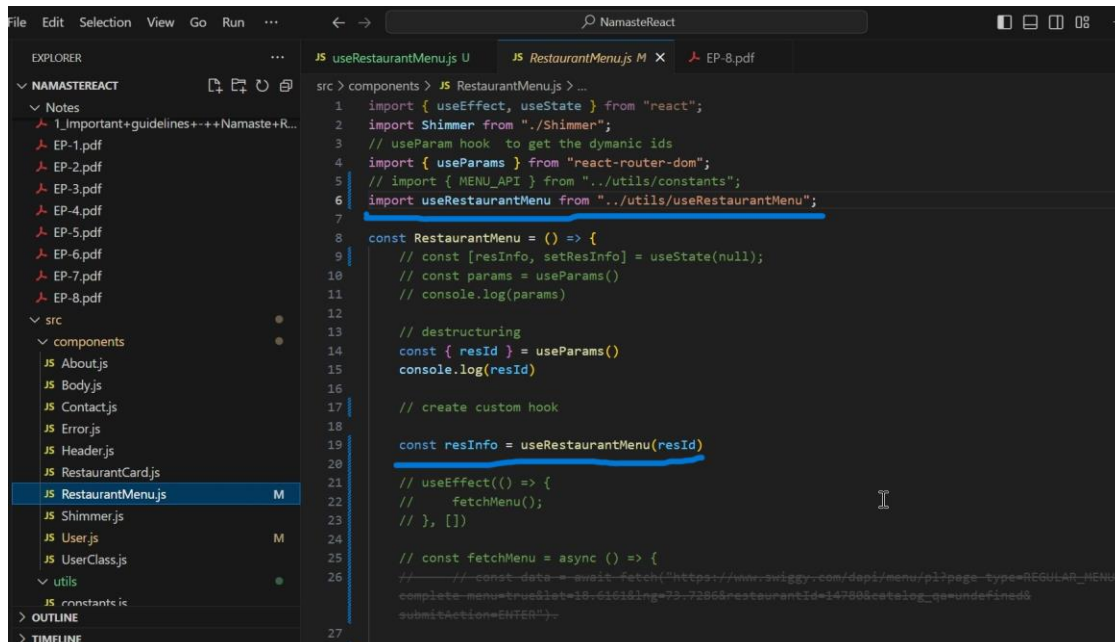
So we get the resId like id of any restaurant pizza hut now it has to fetch the data and return the restaurant information back to where the hook its been called. Inside it will write the logic.

### Custom hook-



```
1 import { useState, useEffect } from "react";
2 import { MENU_API } from "../utils/constants";
3
4 const useRestaurantMenu = (resId) => {
5   const [resInfo, setResInfo] = useState(null);
6
7   //fetchData
8   useEffect(() => {
9     fetchMenu();
10  }, []);
11
12  const fetchMenu = async () => {
13    // const data = await fetch("https://www.swiggy.com/dapi/menu/pl?page-type=REGULAR_MENU&complete-menu=true&lat=18.6161&lng=73.7286&restaurantId=14780&catalog_qa=undefined&submitAction=ENTER");
14
15    // Here i am adding the resId
16
17    const data = await fetch(MENU_API + resId)
18
19    const json = await data.json()
20
21    // console.log(json);
22    setResInfo(json.data)
23  }
24  return resInfo;
25 }
26
27 export default useRestaurantMenu;
```

### Call/ import the custom hook-



```
1 import { useEffect, useState } from "react";
2 import Shimmer from "../Shimmer";
3 // useParam hook to get the dynamic ids
4 import { useParams } from "react-router-dom";
5 // import { MENU_API } from "../utils/constants";
6 import useRestaurantMenu from "../utils/useRestaurantMenu";
7
8 const RestaurantMenu = () => {
9   // const [resInfo, setResInfo] = useState(null);
10   // const params = useParams()
11   // console.log(params)
12
13   // destructuring
14   const { resId } = useParams()
15   console.log(resId)
16
17   // create custom hook
18
19   const resInfo = useRestaurantMenu(resId)
20
21   // useEffect(() => {
22   //   fetchMenu();
23   // }, []);
24
25   // const fetchMenu = async () => {
26   //   // const data = await fetch("https://www.swiggy.com/dapi/menu/pl?page-type=REGULAR_MENU&complete-menu=true&lat=18.6161&lng=73.7286&restaurantId=14780&catalog_qa=undefined&submitAction=ENTER");
27   // }
```

This code working the same way as it was working before. There is no change in the feature it's a good way of writing code.

There is no issue if you want to use API call in the previous way but if you want to API call in the different hook that make much more scene, it make testable because now if I want to test the fetching data logic I just need to test useRestaurantMenu. Suppose if there is an issue in my fetching data logic I know where I want to make the changes.

So if there is an bug related to fetching data I have to go to useRestaurantMenu hook and if there is an issue related to display the data will check RestaurantMenu component.

## Create the custom hook for online offline feature-

Somehow, I should hook that give me the status of my internet. what I am trying to build is suppose my website should know whether my user is online or offline/ internet is active or inactive. We see such type of features in WhatsApp, messengers

Will implement the feature when you are active /online/ internet connect active then its shows green as soon as goes its shows red. Suppose the internet goes off when the user click on card it will throw an error- website is not loading, 404 error, you are offline please check your internet connect.

For implement the logic we just return the Boolean values so useOnlineStatus return the boolean. But how we can check we are offline or online will check using event listener.

[https://developer.mozilla.org/en-US/docs/Web/API/Window/online\\_event](https://developer.mozilla.org/en-US/docs/Web/API/Window/online_event)

Initially useOnlineStatus we don't have any information from the caller , caller means whenever the hooks will be used so now the caller is RestaurantMenu take the example of param so to get the parameter from the url and no need to particular information from the caller the component where it is being called I don't need. I just access my url we can do it by using utility functions, browsers functions.

Similarly, in useOnlineStatus hook I don't need extra information from the component just need to return the Boolean value.

But how we can check we are offline or online will check using event listener.

Just check window.online event so window gives us access to the window object and the window object already has the event listener online to it. So will use the event listener whether you are online or offline.

Que is how many times we add the event listener to our browser

Ans is just once. How we can add the event listener just once so we will use useEffect with the empty dependency array so whatever we are written inside it will execute just once.

Even listener job is keeping tracking whether the internet is online or offline and return the status back.

```
src > utils > JS useOnlineStatus.js > useOnlineStatus
1  import { useEffect, useState } from "react";
2
3  const useOnlineStatus = () => {
4      const [onlineStatus, setOnlineStatus] = useState(true);
5
6      useEffect(() => {
7          window.addEventListener("offline", () => {
8              setOnlineStatus(false);
9          });
10
11          window.addEventListener("online", () => {
12              setOnlineStatus(true);
13          });
14      }, []);
15
16      // boolean value
17      return onlineStatus;
18  };
19
20  export default useOnlineStatus;
21
```

If I will go online it will change the onlineStatus , onlineStatus is a normal state variable

Every time my status changes I will update my onlineStatus and will return it.

Initially value of online status will be true (green)

Now in the body component implement – If the online status of internet is true then show the cards and if the online status is false then show the message you are offline



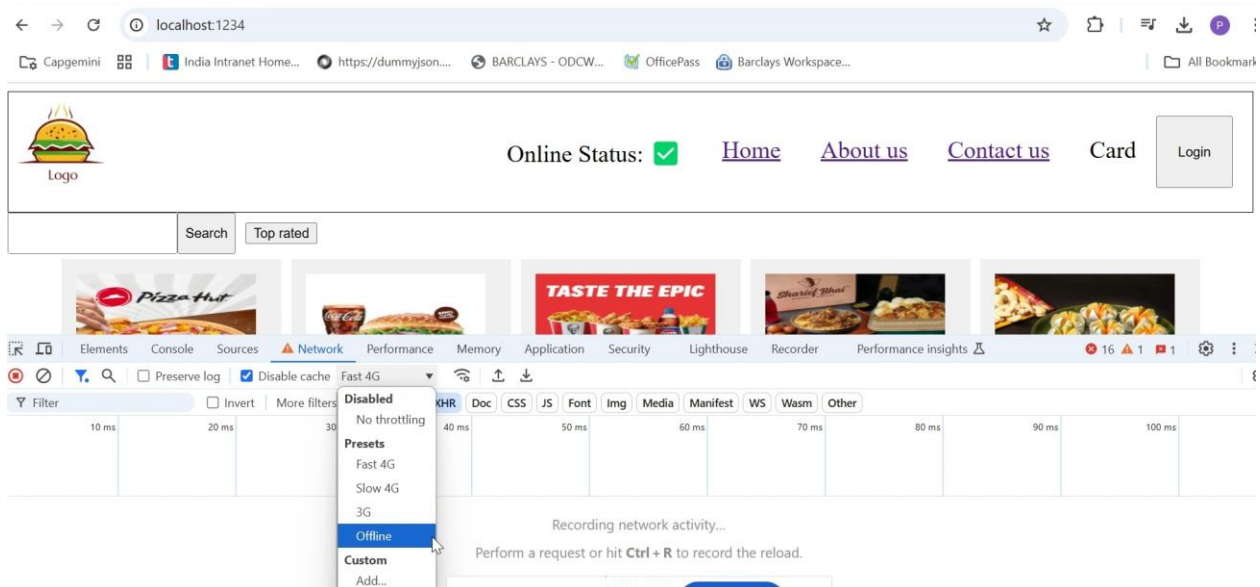
```
// use the custom hook here
const onlineStatus = useOnlineStatus();

if (onlineStatus === false)
  return (
    <h1>
      Looks like you're offline!! Please check your internet connection;
    </h1>
  );
```

Import the custom hook in body component and write the condition for offline status.

In browser itself we can turn on and off the internet –

Inspect -> Network -> No throttling, there is option offline



Now adding the green and red logo in the header



Write the below code –

Call the custom hook in the header component and write the condition in li for online status.

```

src > components > JS Header.js > [🔍] Header
4 | import useOnlineStatus from '../utils/useOnlineStatus';
5 |
6 | // Named Export
7 | export const Header = () => {
8 |     const [btnNameReact, setBtnNameReact] = useState("Login")
9 |     // console.log("Header Render")
10 |
11 |     // If no dependency array => useEffect is called on every render
12 |     // If dependency array is empty = [] => useEffect is called on initial render (just once)
13 |
14 | // call the custom hook |
15 | const onlineStatus = useOnlineStatus();
16 |
17 | useEffect(() => {
18 |     // console.log("useEffect Called")
19 | }, [btnNameReact]);
20 |
21 | return (
22 |     <div className="header">
23 |         <div className='logo-container'>
24 |             <img className="logo" src={LOGO_URL} />
25 |         </div>
26 |         <div className="nav-items">
27 |             <ul>
28 |                 <li>
29 |                     Online Status: {onlineStatus ? "✅" : "❌"}
30 |                 </li>
31 |                 <li>
32 |                     <Link to="/">

```

When we create the own custom hook is it mandatory to use the word “use” before for your write the name of it.

**Ans-** It’s not mandatory if you don’t write the word “use”. If you check the react docs, they recommend using the word “use”.

Similarly, the component name you should always create the component name with capital letter but its not mandatory, it will work without it.

But most of time react developer setup the linting process and the linter start throw an error if you don’t follow the convention. So always do what the documentation and library recommended.

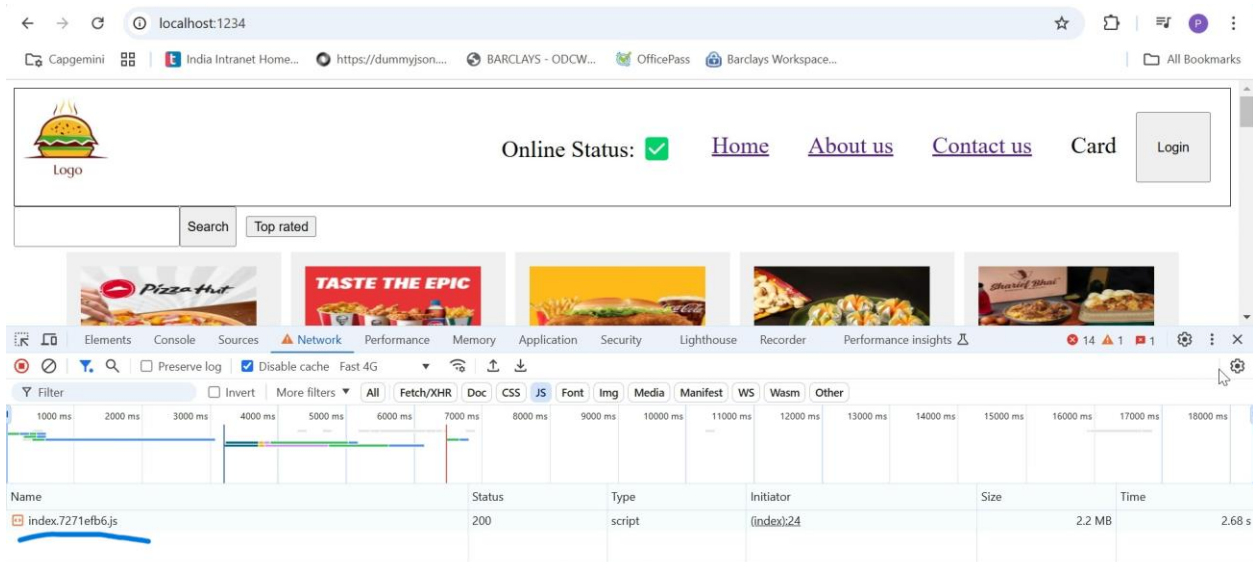
One more reason for “use” before the developer easily understand that there is using the custom hook.

Whenever we develop large scale application which have 1000 of components and they are nested the performance of applications are not good. So what we do to make it performant –

What is the problem for developing such app.

The size of the js file increase a lot, actually bundler takes all your file and make it into one.

If you go to the dist folder where you can see bundler take all the files and generates one js file and it will give it you the browser an dthis is the only file which will loaded on to the browser.



And this one js file loaded and everything happen on the webpage all the code present in one js file and everything happen on the webpage is happening through one js file.

If you will make thousand of things in one js file so the size of the JS file increase a lot here is 2.2 mb but this is for the development build when it goes to the production build it will much smaller than the 2.2mb but still significantly high. So this js file take lot of time to load. This JS file is so huge when you load this it will take lot of time to get.

So how to optimize it-

Can you have to break down into smaller pieces and I do something that my allocation is not just in one js file but smaller js files.

But there are two things either we can bundle or we can not, should we do bundling – yes of course we have 1000 of components. suppose we have 1000 of components do don't load 1000 of file on to the webpage, that's useless.

And also we do not put 1000 of the files on to the one file.

Both solution is not true.

**So what we have to do we will try to make smaller bundles of these files and this processes is known as Chunking, Code Splitting, dynamic bundling, lazy loading, on demand loading , dynamic import.**

So, what we are trying to solve is you have to chunk your application, you have to make the application into smaller chunk, you have to split your code, you have to make dynamic bundling. These are the same things for the same process to break down your app into smaller logical chunks.

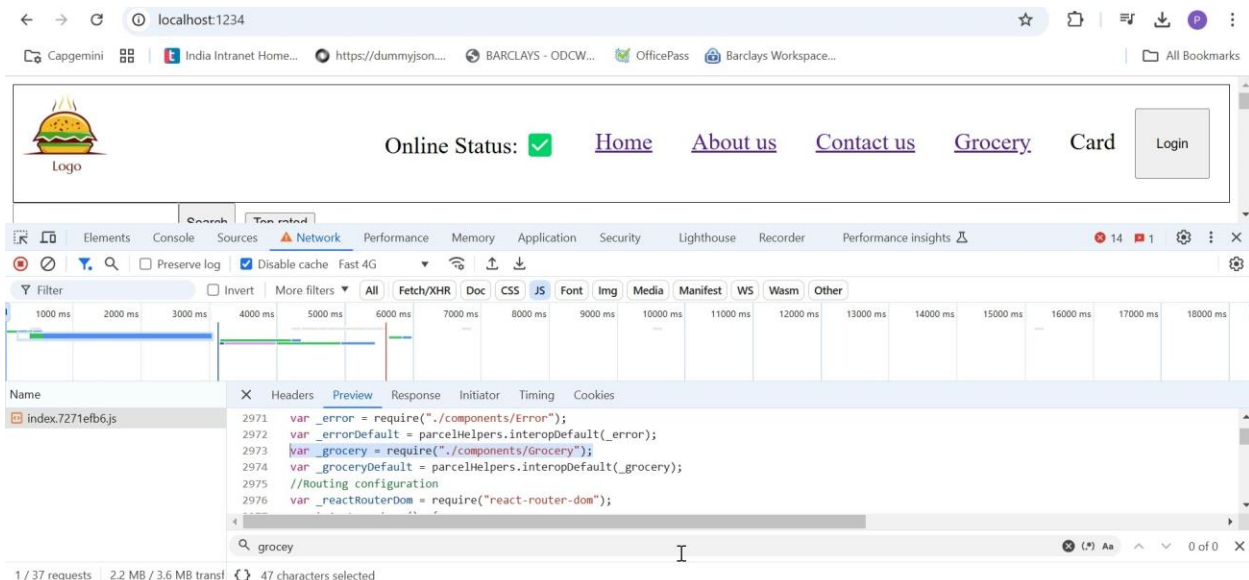
How to make smaller bundles and when to make smaller bundles and what should be there in smaller bundles.

Will do the logical separation of the bundles. That means the bundles should have enough code of the feature of the website. So, you don't put a load on the single bundle. So you don't have request of js doesn't become so heavy that take lot of time to get on the browser.

Suppose our app is not delivery only food it also selling grocery example just like swiggy instamart or we can take the example of uber has cabs, uber has food delivery, uber has grocery delivery. So our app suppose also had grocery delivery vertical that the separate entity all together. Will create the grocery delivery button if I click on it takes me to a grocery page where I can see all the grocery and order it online and the grocery vertical also has lot of component. So, what we will do we will make the different bundle of grocery and other will make for main bundle.

```
1  const Grocery = () => {
2    return (
3      <h1>
4        {" "}
5        Our grocery online store, and we have a lot of child components inside
6        this web page!!!
7      </h1>
8    );
9  };
10
11  export default Grocery;
12
```

But this grocery code still inside index.js bundle



So I want to logically distribute my application that my grocery and all component of grocery should come from different bundle.

### Create the separate bundle of grocery-

Don't import grocery directly like this, will import grocery by using lazy loading.

Why we called lazy loading because when our app loads initially it will not load the code for grocery, only when I will go to the grocery page then will be grocery code will be there in app. So, we will not load everything directly but we will do lazy loading when required.

Lazy is the function that is given to us by react it comes from react package and it is the named export.

```
import React, {lazy} from 'react';
```

```
const Grocery = lazy(() => import("./components/Grocery"));
```

this lazy function takes the callback and the import.

this import is the function this import is not the first one import and this function take the path of grocery where it needs to come from.

Now, we will got the new JavaScript file over there so now split our code into two javascript bundles. Grocery has its own bundle and main bundle are separated now the main bundle doesn't code for grocery component and grocery.js has own grocery code.

localhost:1234/grocery

# Oops!!

## Something went wrong

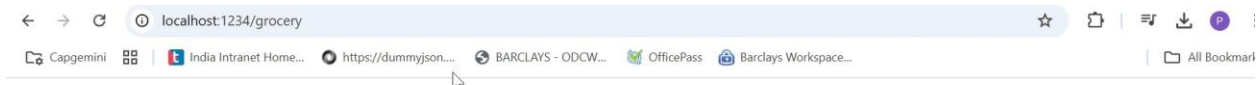
Network tab showing requests:

Name	Status	Type	Initiator	Size	Time
index.7271efb6.js	200	script	(index):24	2.2 MB	2.96 s
Grocery.c52c2ad2.js?1731607447907	200	script	index.7271efb6.js:71	25.1 kB	277 ms

```

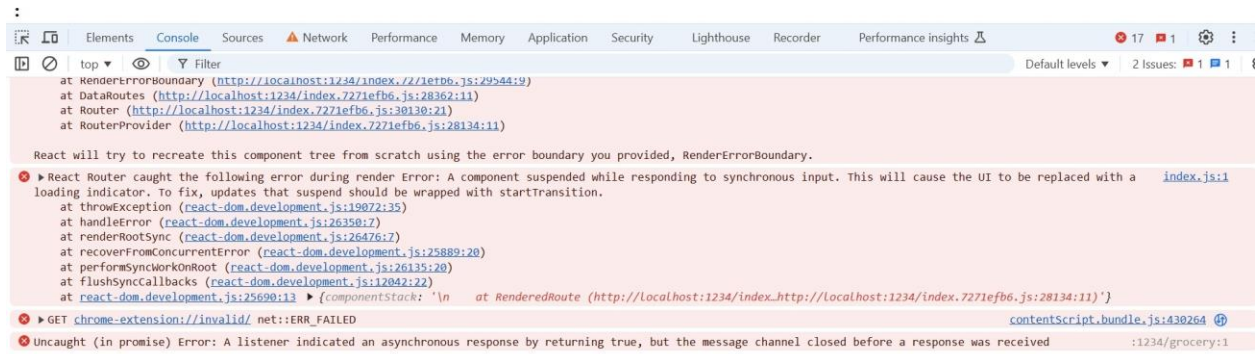
NAMASTEREACT
├── .parcel-cache
└── dist
    ├── JS Grocery.c52c2ad2.js
    ├── JS Grocery.c52c2ad2.js.map
    ├── JS index.4f22b734.js
    ├── JS index.4f22b734.js.map
    ├── # index.9e4c6fd2.css
    ├── # index.9e4c6fd2.css.map
    ├── # index.15f0a4b1.css
    ├── # index.15f0a4b1.css.map
    ├── index.15f0a4b1.css.map.16016.14
    ├── index.15f0a4b1.css.map.34268.24
    ├── JS index.73bac2e7.js
    ├── JS index.73bac2e7.js.map
    ├── # index.652f138f.css
    ├── # index.652f138f.css.map
    ├── JS index.7271efb6.js
    ├── JS index.7271efb6.js.map
    ├── JS index.7826abd7.js
    ├── JS index.7826abd7.js.map
    ├── JS index.bc93b592.js
    └── JS index.bc93b592.js.map
  
```

Now getting this error



# Oops!!

## Something went wrong



Why-

This grocery code takes 12milisecond to come to the browser and react is very fast react try to load the grocery component, but the code is not there. That's react suspend the rendering so the grocery code was not there, so react did it will not load that is why it will throw an error.

Will handle this error by using Suspense, it the component comes from react library.

So we just wrap our component into Suspense

Import the Suspense-

```
import React, {lazy, Suspense} from 'react';
```

Wrap the component by Suspense

```
{
  path: "/grocery",
  element: <Suspense><Grocery /></Suspense>,
},
```

Now you have to give the fallback or the placeholder. What should react render when the code is not available so basically kind of loading screen.

When you are on the home page your code of grocery not there and react try to load something it can load until the grocery is there so in the meantime intermediately react wants something



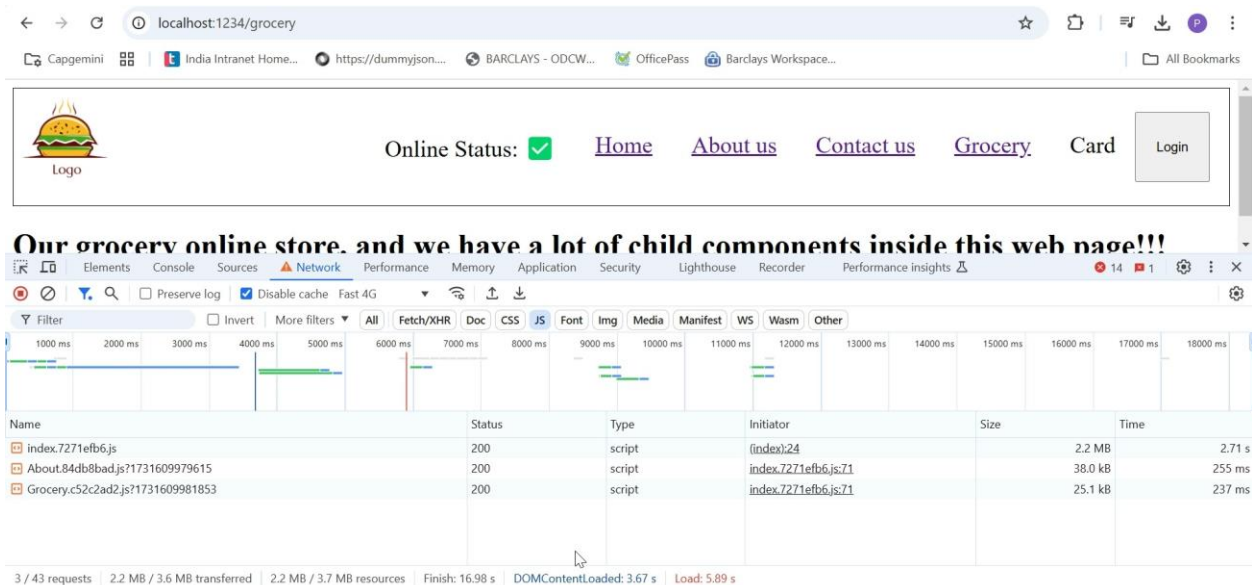
to be present on screen and you can give that inside fallback. How you can pass, you can pass the JSX. You can also pass the simmer UI over here

```
{
  path: "/grocery",
  element: <Suspense fallback={<h1>Loading.....</h1>}><Grocery /></Suspense>,
},
```

So when I click on grocery it will take some second for the millisecond will see the loading. It is so fast we can not see it. Just choose the slow network so that you can see it loading... screen.

Like this way we can optimize our code.

Do the lazy loading of about component .



This is how I will distribute my application to smaller smaller chunks and this make our application very performant.