

Custom Hooks :-

We can build our own hooks to ensure

- readability,
- reusability,
- separation of concerns (maintainability / modularity),
- testable : because I can write separate test cases for each helper function/hook

Now, Hooks are functions at the end of the day and they are considered to be utility/helper functions. So, they are generally kept in a separate folder named “**utils**” altogether. In that folder we can create a file named “**helper.js**” to store our custom hooks.

- Generally, Hooks are mentioned in the utils folder inside another Hooks folder or just directly
- Always, mention “**use**” as the prefix of the filename containing our hook.
- Always do a **named export** from the hooks file because a default export fails to imply which function that file is exporting.

In the RestaurantMenu.js, we are actually doing 2 things :-

- Fetching details about the menu of a restaurant by making an API call to the server.
- Displaying or rather rendering the details of the component in the DOM

We will be making the first functionality as a hook in a file named “useRestaurant.js” in the utils folder.

```
6   const RestaurantMenu = () => {
7     // reading dynamic URL params
8     const { id } = useParams();
9     const [restaurant, setRestaurant] = useState(null);
10
11     useEffect(() => {
12       getRestaurantInfo();
13     }, []);
14     getRestaurantInfo = async () => {
15       const data = await fetch("https://www.swiggy.com/dapi/menu/v4,
16       const json = await data.json();
17       setRestaurant(json.data);
18     }
19 }
```

Right now, in the RestaurantMenu.js, lines 8 to 18 is actually fetching menu of a restaurant by using a restaurantId, got from the url using useParams(). Instead of these lines, it will be great if we can just make a hook named useRestaurant() and pass the restaurantID to it and get the menu data which will be stored in the ordinary restaurant variable, instead of using a state variable to store it. We can do like:

```
const restaurant = useRestaurant( id );
```

So, the custom hook should accept restaurantId (here id) as parameter. Then it should also make an API call to fetch data just like the above code from line 11 to 18. To store the API data, there should also be a state variable so that when the data is fetched, that state is updated and the hook returns the actual data. So, finally we do :-

```
src > utils > useRestaurant.js > ...
1   import { useState, useEffect } from "react";
2
3   const useRestaurant = (resId) => {
4     let [restaurant, setRestaurant] = useState(null);
5
6     // Get Data from the API call
7     useEffect(() => {
8       getRestaurantInfo();
9     }, []);
10    getRestaurantInfo = async () => {
11      const data = await fetch("https://www.swiggy.com/dap
12      const json = await data.json();
13      setRestaurant(json.data);
14    }
15
16    // Return the Restaurant Menu Data
17    return restaurant;
18  }
19
20  export default useRestaurant;
```

```
src > components > RestaurantMenu.js > ...
1   import { useEffect, useState } from "react";
2   import { useParams } from "react-router-dom";
3   import { IMG_CDN_URL } from "../config/restaurantImgConfig";
4   import Shimmer from "../Shimmer";
5   import useRestaurant from "../utils/useRestaurant";
6
7   const RestaurantMenu = () => {
8     // reading dynamic URL params
9     const { id } = useParams();
10
11     // Calling Custom Hook
12     const restaurant = useRestaurant(id);
13
14     return (!restaurant) ? (
15       <Shimmer />
16     ) : (
17       <div className="menu">...
35     </div>
36
37   )
38 }
```

In the RestaurantMenu.js, the restaurant variable does not need to be a state variable, the state of the required variable is maintained by the hook. Also, the way in which the restaurant variable of RestaurantMenu.js gets updated is :-

- In RestaurantMenu.js, in line 12 the Hook is called
- The Hook takes the resId as parameter and makes an API call to the server
- Since useEffect() is called after first render, at first the hook returns a null value to the above restaurant variable and hence the Shimmer UI is rendered.
- After the first render (here return statement is actually the render and instead of returning a JSX, its returning a value), the useEffect() is called, which then calls the reqd function to make an API call and then when the restaurant state variable gets updated i.e. it's state changes, the return statement is called again and this time it returns the fetched data to the normal restaurant variable.
- After that we can see the menu in our UI again.

We can further clean our hook by writing that URL in our constants file and importing it from there.

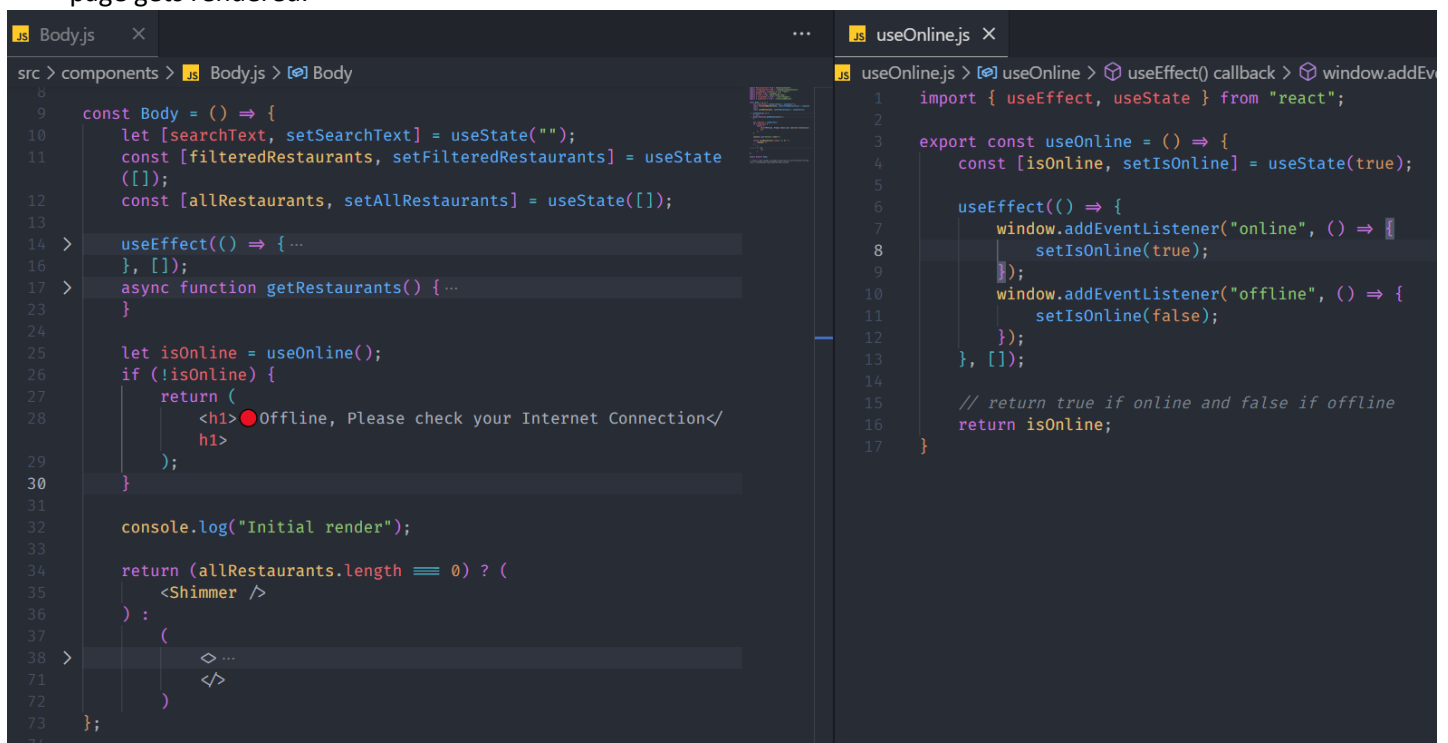
Building a new Feature (Are you online or not?)

We want to build a feature such that if the user is on the Home Page and he goes offline, when he tries to click on any restaurant or any other links, instead of him being able to click on any of the links, a pop saying "You are offline" or "Check your Internet connection" should appear. This will enhance the user experience. To know whether a system is online or offline, there is a JS event listener "online". You can read more about it from :- [Link1](#). For this, we can create a separate **custom hook** altogether named **useOnline()** to check whether the client system is online/offline. Things to remember :-

- In the custom hook, useOnline(), we have to return the status (here, stored in a variable named isOnline) of the system which tells whether the user is online or not. In first case, the status will be true else false.
- The return value will be stored in a normal variable (here named as online) in the Body.js file. So, if the online is true, then it will render the restaurants component as it is, else it will show a message to the user that the Internet connection should be checked.
- We also know that the client system can go from online to offline and vice-versa anytime. And each time this happens, we have to update the isOnline variable. So, we know that we have to make the isOnline a state variable.

Now, designing the useOnline() hook :-

- There should be a state variable to store the online or offline status of the client system.
- Now, the Home page will not even load at first if there is no internet connection, so we should only take care of the situation when the user is already on the Home Page and then if suddenly the user goes offline, then the popup should appear.
- So, we should set an event listener in our Home page which tells whether the user is online or offline
- Also, the event listener needs to be set only once and never again. So, the best place to do this is inside an useEffect hook in the useOnline() custom hook. This way the event listener is set to our page only once i.e. after the Home page gets rendered.



```
src > components > JS Body.js > [x] Body
  8
  9   const Body = () => {
10     let [searchText, setSearchText] = useState("");
11     const [filteredRestaurants, setFilteredRestaurants] = useState
12       ([]);
13     const [allRestaurants, setAllRestaurants] = useState([]);
14   >   useEffect(() => { ...
16     }, []);
17   >   async function getRestaurants() { ...
23     }
24
25     let isOnline = useOnline();
26     if (!isOnline) {
27       return (
28         <h1>🔴 Offline, Please check your Internet Connection</
29         h1>
30       );
31     }
32
33     console.log("Initial render");
34
35     return (allRestaurants.length === 0) ? (
36       <Shimmer />
37     ) :
38     >   (
39       < ...
40     </
41   );
42 };
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74

JS useOnline.js > useOnline > [x] useEffect() callback > [x] window.addEv
  1   import { useEffect, useState } from "react";
  2
  3   export const useOnline = () => {
  4     const [isOnline, setIsOnline] = useState(true);
  5
  6     useEffect(() => {
  7       window.addEventListener("online", () => {
  8         setIsOnline(true);
  9       });
10       window.addEventListener("offline", () => {
11         setIsOnline(false);
12       });
13     }, []);
14
15     // return true if online and false if offline
16     return isOnline;
17   }
```

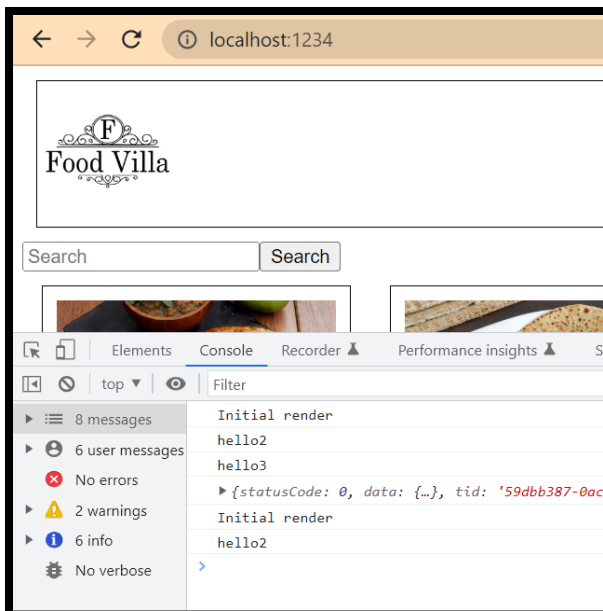
#NOTE :-

Just like we used a state variable (isOnline here) in our custom hook to change the return value from our hook based on the parameter passed during calling the hook, the prebuilt React Hooks are also like that.

For ex : useParams() returns the params of an url. So, this hook must also have a state variable to store the parameters and return it. And that state variable must change everytime the url changes.

Output 1 :-

```
JS Body.js x
src > components > JS Body.js > Body
9   const Body = () => {
10     let [searchText, setSearchText] = useState("");
11     const [filteredRestaurants, setFilteredRestaurants] = useState([]);
12     const [allRestaurants, setAllRestaurants] = useState([]);
13
14     > useEffect(() => { ...
15     }, []);
16     async function getRestaurants() {
17       const data = await fetch("https://www.swiggy.com/dapi/
18         restaurants/list/v5?lat=28.717111&lng=77.1575986
19         page_type=DESKTOP_WEB_LISTING");
20       const json = await data.json();
21       console.log(json);
22       setAllRestaurants(json?.data?.cards[2]?.data?.data?.cards);
23       setFilteredRestaurants(json?.data?.cards[2]?.data?.data?.
24         cards);
25     }
26     console.log("Initial render");
27     let isOnline = useOnline();
28     if (!isOnline) {
29       return (
30         <h1>🔴 Offline, Please check your Internet Connection</
31         h1>
32       );
33     }
34
35     > return (allRestaurants.length === 0) ? ( ...
36     ) : ( ...
37     )
38     > ( ...
39     )
40   };
41
42 JS useOnline.js M x
JS useOnline.js > useOnline > useEffect() callback > window.add
1   import { useEffect, useState } from "react";
2
3   export const useOnline = () => {
4     const [isOnline, setIsOnline] = useState(true);
5
6     useEffect(() => {
7       console.log("hello3");
8       window.addEventListener("online", () => {
9         console.log("hello");
10        setIsOnline(true);
11      });
12       window.addEventListener("offline", () => {
13        console.log("hello4");
14        setIsOnline(false);
15      });
16     }, []);
17
18     // return true if online and false if offline
19     console.log("hello2");
20     return isOnline;
21   }
```



So, when I am just starting my server and opening the home page for the first time, output in console is :-

The 1st line is because of line25 in Body.js

The 2nd line is because, in line27 of Body.js, a function call is made to the useOnline() hook and then it prints the line 18 of useOnline.js

The 3rd line is because of the useEffect() being called in Body.js and the function getRestaurants() gets called, where line20 gets executed In line 21,22, the state variables are getting updated so both the Body component and all it's child components will also get re-rendered.

Here, the custom Hook is also like a child component and it gets rerendered too after the state of state variables in Body.js get updated. Hence, only the "hello2" gets printed and the useEffect is not called again.

Again, if I go to the Networks tab and change the "No throttling" to "Offline", the setEventListener for offline system gets executed first, so the first line that gets printed is "hello4".

Also, in line 14 of useOnline.js, the state variable isOnline gets updated to false, so a rerender should happen of not only the useOnline.js component, but also the Body Component because the latter is also using the value of the isOnline state variable. (Remember that the purpose of using a state variable is that, whenever we update a state variables, all the variables dependent on it also gets updated, so here too the same thing happens, regardless of the fact the normal isOnline variable is defined in some other component altogether.

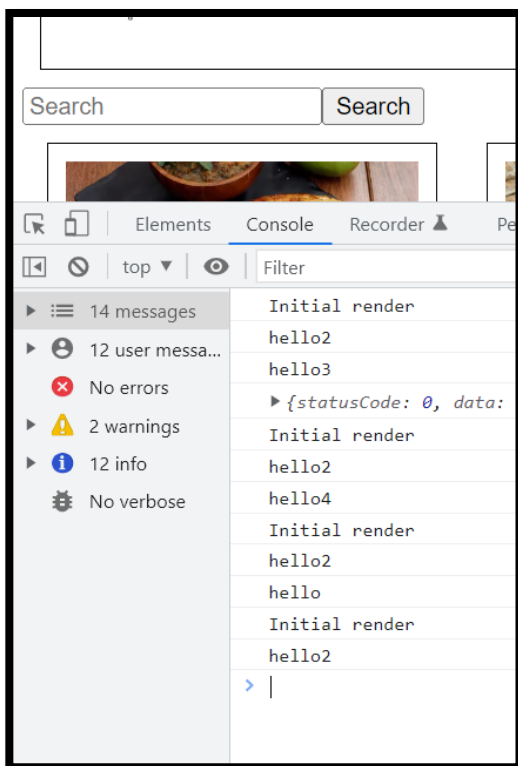
The flow of this kind of rerendering is : the parent component gets rerendered and along with it all the child components too sequentially. So, “Initial render” is printed first, then “hello2”. **Take note that not all parent components will be rerendered, but only those which are dependent on the state variable only**, that is why even though App.js is the parent component of Body.js, App component will not rerendered and neither will be its other child components : Header and Footer.



When I change the Network from “Offline” to “No throttling”, we get the last 3 lines.

The 3rd last line “hello” is printed because of the “online” eventListener we set on our webpage.

In that eventListener, we also update the state of the state variable named isOnline, so again the Parent component which uses the value of the state variable will be rerendered (here Body) and along with that the other child components will be too. Hence the last 2 lines.



Optimising the above functionality :-

There is a problem with the above code, which is, whenever our Body component is newly rendered (not rerendered), its child components are also rendered newly and not rerendered. This means that useOnline() also gets rendered.

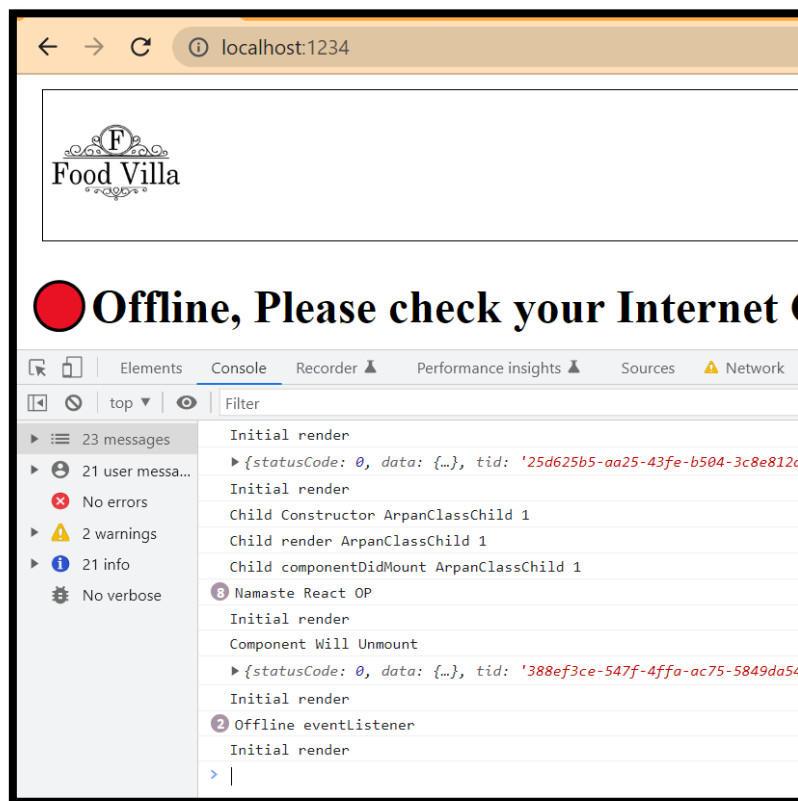
Now, earlier when we opened the website, we were already on the Home Page and that means we have already rendered the useOnline hook once and the eventListeners for both online and offline mode has been set for that Home Page.

But, if we now navigate to any other page like About/Contacts and then come back to the Home page again, then the Body along with the useOnline component gets rendered again.

This means that another additional eventListener has been added for both online and offline.

In the below example, we have started the server and then opened the Home Page at localhost:1234. That’s the reason behind the 1st three lines of output in the console. Then we navigated to the About page, which explains the next 4 lines. Then we again navigated to the Home page. The line 9 is because of the componentWillUnmout() function. The lines 8,10,11 are because of coming back to the Home Page.

Now, I go to the networks tab and make it offline. So, the isOnline state variable of the useOnline.js gets updated and the Body component gets rerendered, which explains the last and since there are now 2 eventListeners, we get 2 same output at 2nd last line.



```

src > components > Body.js > ...
9  const Body = () => {
10    let [searchText, setSearchText] = useState("");
11    const [filteredRestaurants, setFilteredRestaurants] = useState([]);
12    const [allRestaurants, setAllRestaurants] = useState([]);
13
14    useEffect(() => { ...
15    }, []);
16    async function getRestaurants() { ...
17    }
18
19    console.log("Initial render");
20
21    let isOnline = useOnline();
22    if (!isOnline) { ...
23    }
24
25    return (allRestaurants.length === 0) ? ( ...
26    ) : ( ...
27    );
28  };
29  export default Body;

```

```

src > utils > useOnline.js > ...
1  import { useEffect, useState } from "react";
2
3  export const useOnline = () => {
4    const [isOnline, setIsOnline] = useState(true);
5
6    useEffect(() => {
7      window.addEventListener("online", () => {
8        setIsOnline(true);
9      });
10     window.addEventListener("offline", () => {
11       console.log("Offline eventlistener");
12       setIsOnline(false);
13     });
14   }, []);
15
16   // return true if online and false if offline
17   return isOnline;
18 }

```

So, whenever we unmount the component, we have to make sure that the we also cleanup the eventListeners

Reusing the custom Hook in other places :

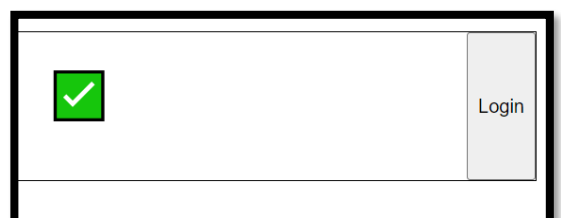
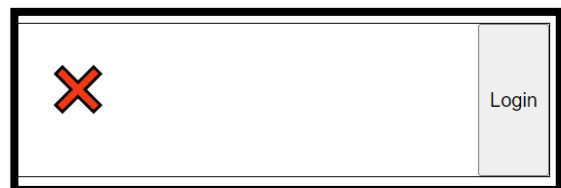
We can use the hook in our Header section to show beside the login button, a green tick if online and a cross if offline.

```

const Header = () => {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  let isOnline = useOnline();
  return (
    <div className="header">
      <Title />
      <div className="nav-items">...
      </div>
      <h1>{isOnline ? "✅" : "❌"}</h1>
      {
        (!isLoggedIn) ?
          (<button onClick={() => { setIsLoggedIn
            : (<button onClick={() => { setIsLogg
          }
        }
      }
    )
  );
}
export default Header;

```

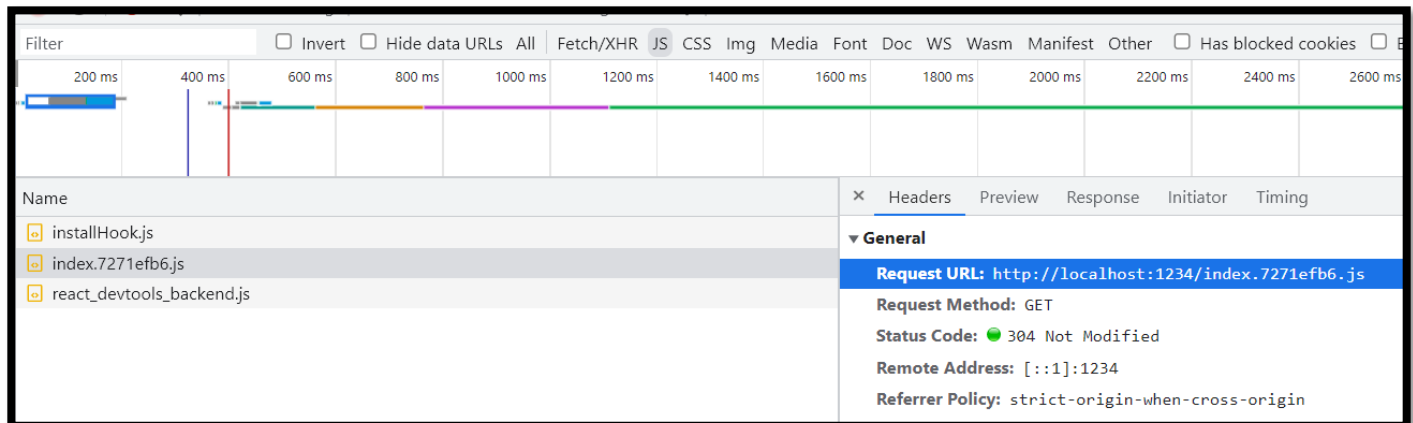


Chunking / Code-splitting / Dynamic Bundling / Lazy Loading / On-demand Loading / Dynamic

Import :

Now, in we know that the job of our bundler (here Parcel) is to bundle/compress/minify together our files in the production code. So, how many production ready js files does Parcel create from all the js files we have written?

To know, just open the website and move on to the networks tab. There, choose only the "JS" tab and you will see only 1 js file being called from the localhost (others seen in the below image are nothing but chrome extensions or other things)



However, if you build a large scale application, then there will be multiple components and bundling them all up in a single js file will take too much time to load. Therefore, we should break the entire production ready code into smaller parts. This concept is called Chunking/Code-splitting.

Is bundling good/bad ?

Bundling is good, but that does not mean that we have to bundle everything up in a single file. Instead, we should try to make logical bundles. This means we should code in such a way that the bundle corresponding to a specific use case only gets loaded. For ex :- See the video from 1:54:34 to 2:11:00

Using lazy() and Suspense element:

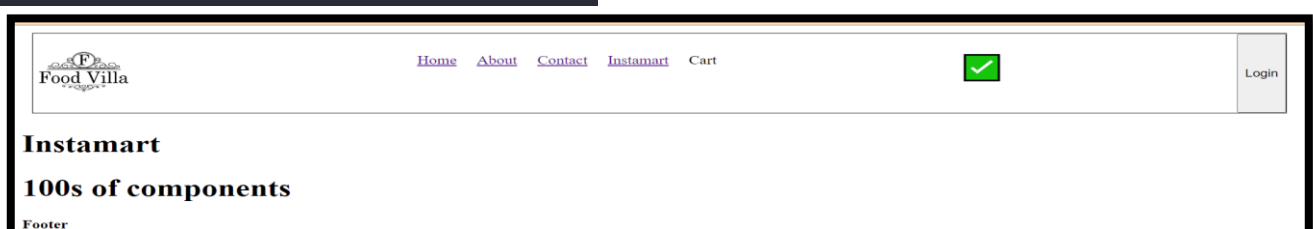
First we will make a simple component named Instamart and add it to the Header and App component (in the routes section).

```
JS Instamart.js U X JS Header.js M JS App.js M
src > components > JS Instamart.js > default
1  const Instamart = () => {
2    return (
3      <div>
4        <h1>Instamart</h1>
5        <h1>100s of components</h1>
6      </div>
7    )
8  }
9
10 export default Instamart;

JS Instamart.js U JS App.js M JS Header.js M X
components > JS Header.js > Header
return {
  <div className="header">
    <Title />
    <div className="nav-items">
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/contact">Contact</Link></li>
        <li><Link to="/instamart">Instamart</Link></li>
        <li>Cart</li>
      </ul>
    </div>
  </div>

JS Instamart.js U JS App.js M X JS Header.js
src > JS App.js > appRouter > children > element
49
50
51
52   path: "/instamart",
53   element: <Instamart />
54
55 },
56 ])
```

Now, we do not want this Instamart component to get loaded when anyone opens our website. Rather we want it to load only when someone clicks on the Instamart link in our Header section.



We know that in our App.js, we have mentioned in the Routes/Paths section that our Body, Header and Footer component will get loaded by default. And in the appRouter section of Body.js, we have also mentioned different other components like About, Contacts and Instamart, which will also get loaded by default, but just not be visible in that Home Page. But we don't want Instamart to get loaded along with the other components in the same bundle at the same time.

That's why we use a function called **lazy()** while importing the Instamart component in the App.js . This **lazy()** is a named import from the React library. We pass a callback function to this lazy() and in that callback function, we do **dynamic import**. Instead of the normally importing the Instamart component.

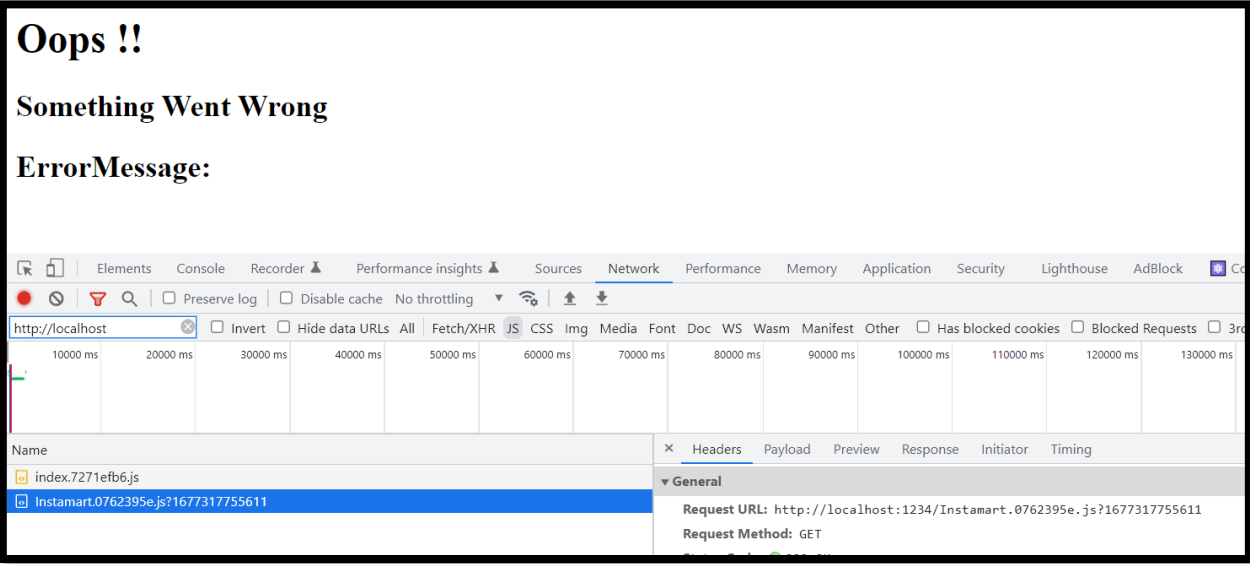
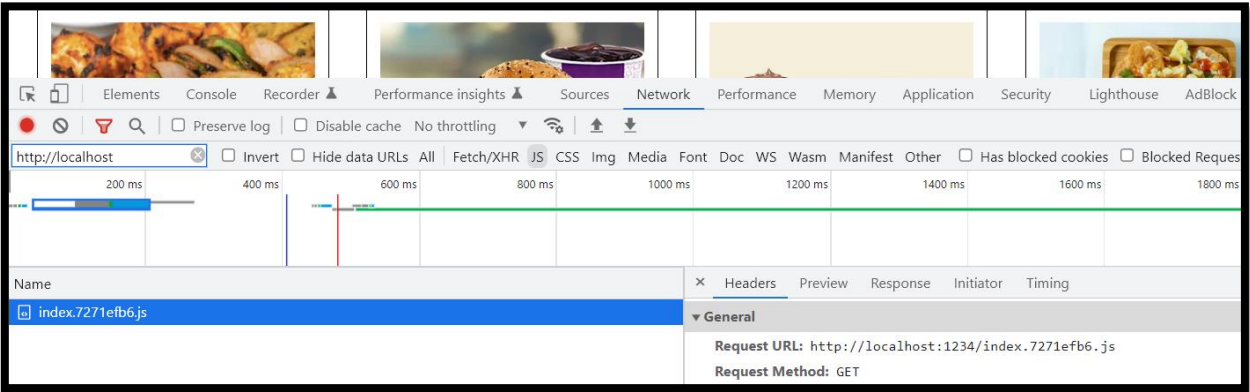
```
import React, { lazy, Suspense } from "react";

// We will not use this
// import Instamart from "./components/Instamart";

// Dynamic Import / Lazy loading
const Instamart = lazy(() => import("./components/Instamart"));

const AppLayout = () => (
  <Header />
  <Outlet />
  <Footer />
)
```

We are actually saying that the Instamart component should be imported lazily, that is, only when it is needed and thus there will be 2 separate JS bundles now :- 1 for the Instamart and the other for the rest JS code. Below are the pictures for the network calls made on just opening the webpage and the 2nd image shows the network calls made after clicking Instamart link in Header.



However, as you can see that the Instamart component is not rendering because as evident from the pic below, it takes an additional time for that bundle to get loaded. But React tries to render that component before that time and as such since it cannot find anything, it shows the error message.

Name	Status	Type	Initiator	Size	Time	Waterfall
index.7271efb6.js	304	script	(index)	347 B	109 ms	
Instamart.0762395e.js?167731787...	200	script	js-loader.j...	24.1 kB	36 ms	

So, we have to tell React to stop rendering a component until the bundle with that specific component gets loaded. To do this, we just wrap around that component with a element named **Suspense** like :-

```
{
  {
    path: "/instamart",
    element: (
      <Suspense>
        <Instamart />
      </Suspense>
    )
  }
}
```

Also, we have to name import the Suspense element from the React library. Wrapping Instamart within Suspense means telling React to wait till Instamart component is actually imported using the dynamic import. Also know that all imports actually returns a promise. So, Suspense tells React to wait for the promise to get resolved. Now, when we click on the Instamart link in the Header, it will get rendered successfully.

You know, that React is taking some additional time to render the Instamart component after we click the link, so during that time of loading, should we show a blank screen? No, we should show a Shimmer effect during that intermediate suspense time.

To let React know what to show during that suspense time, we can pass a prop in that Suspense component named **fallback** and pass the Shimmer component there.

```
{
  {
    path: "/instamart",
    element: (
      <Suspense fallback={<Shimmer />}>
        <Instamart />
      </Suspense>
    )
  }
}
```

I will do the same lazy loading for About Component too.

#Important Points :-

1. Never dynamically import 1 component inside another component. This is because if we do so, the former component will get dynamically imported in every render cycle of the latter component.
2. Dynamic Import : Learn more about it here <https://medium.com/@vincent.bocquet/dynamic-import-in-javascript-a-simple-guide-a808cff86458> , however, while using dynamic imports in these examples, do not use .then() functionality because React generally expects a module to be returned, so there's no use of it. Still if you want to use it then :-

```
// Dynamic Import / Lazy loading
const Instamart = lazy(() => import("./components/Instamart")
  .then((module) => module)
);
```