(lecture - 9)

(optimizing our App)

Date.....................

→ Single Responsibility Principle →

Acc. to this principle each of the component should have a single responsibility.

ex → If we have component RestaurantMenu the only job of this component should have to be displaying our Restaurant Menu.

→ Modularity means that we breakdown our code into different small modules so that our code become more maintainable and more testable.

→ whenever we are testing the RestaurantCard and all the test case pass in the code if we can catch the bug very easily and the bug that can be caught by just testing the RestaurantCard.

If we have very big application where all these components are interlinked one inside the another then we will get a random bug and we will have to check whole big component to see ~~that~~ where that bug is coming from.

→ Distributing our code into smaller pieces and keeping it modular makes our code testable and maintainable.

→ If we follow Single Responsibility Principle we get the feature of Reusability, testable & maintainable.

Date...................

→ We can create our own custom hooks.
Hook are just kind of like utility function.

→ We will just abstract/take out some responsibility from a component and extract it inside the hook, so that our hook and our component becomes more modular and more readable.

→ we well make a custom hook for fetching the data and ~~p~~ RestaurantMenu component doesn't worry about how to fetch the Data, it just have to worry about that we got the resInfo data inside the custom hook and it just want to display it now, it doesn't have to manage it's own state, it just have magically access to the resInfo.

→ Single responsibility to display the data on UI.

→ we will create the hook inside utils.
Always create a separate file for separate hook.

→ We well name the file exactly the same name of that hook.

utils

→ useRestaurantMenu.js

Scanned with CamScanner

Date.....................

→ Whenever we are writing hook think like input & output terms.

First of all try to see the contract, we got the resId and now it has to fetch the data ~~ruf~~ return the resInfo back to where that hook is being called from.

```
• import {useEffect, useState} from "react";
• import {Menu_API} from "../utils/constants";

const useRestaurantMenu = (resId) => {
  const [resInfo, setResInfo] = useState(null);

  useEffect(() => {
    fetchData();
  }, []);

const fetchData = async () => {
  const data = await fetch(Menu_API + resId);
  const json = await data.json();
  setResInfo(json.data);
};

return resInfo;
};

export default useRestaurantMenu;
```

Spiral

Scanned with CamScanner

25:02

Date.....................

- We have taken out the logic of fetching the data from RestaurantMenu component.

- Now RestaurantMenu just needs to worry about showing data on UI, by passing the resId

RestaurantMenu.js →

```
  const RestaurantMenu = () => {
    const {resId} = useParams()

  const resInfo = useRestaurantMenu(resId);


      - - -
       -
      -
      -
      -
      -
```

- Now we don't need a State inside RestaurantMenu

→ we will create a hook for internet status
- Online/ Offline

useOnlineStatus.js →

```
import {useEffect, useState} from "react";
 const useOnlineStatus = () => {
```

Date.....................

```
const [onlineStatus, setOnlineStatus] = useState(true);

useEffect (() => {
window. addEventListener ("offline", () =>
  { setOnlineStatus (false);
});
window . addEventListener ("online", () => {
  setOnlineStatus (true);
}):
}, []);

return onlineStatus ;
};

export default useOnlineStatus ;
```

Body.js →

```
import useOnlineStatus from "../utils/useOnlineStatus";
const onlineStatus = useOnlineStatus ();
```

→ use word before hook is not compulsory, it is recommanded only.

Date..................

if (onlineStatus === false) return
<h1> Looks like You're Offline!! </h1>
please check your's internet

- Bundler bundles the whole ~~code~~ component code in single .js file.
- The size of this .js file is increase a lot,

→ The size of JS file increases by how many component it hold, how to optimise it?

→ we can't build large scale frontend application if we can't take care of this.

→ we have to breakdown our App into smaller piece.

→ Can we do something that our application not just one JS file but small JS files.

→ But we don't need 1000 files to load on our browser, making 1000 calls are difficult to do. and also don't want to put 1000 files into one file

Both of these sol's are not true.

we will try to make smaller bundles of these files

This process is known as chunking / code splitting / dynamic bundling / lazy loading / ondemand loading

Lazy loading → when our App loads, it will not load the code for grocery initially, only when we go to our grocery page then only that grocery code be there in our app

Date.............

• It is the process to breakdown our App into smaller chunks.

Q-> How to make smaller bundles? when to make smaller bundles? what should be there in these bundles

A-> We want to do a logical separation of our bundles, that means a bundle should have enough code for a feature.

→ So we can split our bundles into these logical chunks.

a separate component for grocery

• import React, {lazy} from "react";
• ~~import Grocery from "./components/Grocery";~~

const Grocery = lazy(() => import(---path))

It is not same like upper import

• Now we wouldn't write import a path for grocery on top of the page, we will import it from lazy function.

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

Date.....................

→ Now ~~that~~ a single ~~bundle~~ JS file ~~has~~ does not have code for Grocery component. It has not loaded the code for grocery component.

→ after calling Grocery we will get a new JS file of grocery component.

→ But it will throw an error

→ Because this grocery code took 12 milliseconds to come to the browser & React is very fast, React try to load the grocery component but code was not there, that's why React suspends the Rendering That's why it throws an error

→ We will use Suspense, it is a component, comes from React library. We can wrap our component in Suspense.

App.js import {Suspense} from "react";

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

Scanned with CamScanner

Date......................

• Our component is not available at the moment we just ~~j~~ put the suspense and wrap the component around it.

• Now we will give it a placeholder (fallback), what should react render when the code is not available ~~or~~ basically kind of like a loading screen

```
<Suspense fallback={<h1> loading.... </h1>}>
    <Grocery />
</Suspense>
```

Chunking / code splitting / Dynamic Bundling / lazy loading / Ondemand loading / dynamic import → All the code doesn't come at once and it is come when it is requested.