Chap 09 - Optimizing Our App

Date : 22|Jan|2023 @ashrayaa.

CUSTOM HOOKS

→ Why? When? & How?

→ Why should we build hooks?

* Reusability
* Readability
* Seperation of concerns
* Maintainability.

→ Hooks are just normal javascript functions.

→ Functions are used to wrapup a logic and can reuse it anywhere we want to.

Eg:- In Body.js,

FilterData() is a function written seperately. We didn't write it between the code because it breaks modularity, reusability & readability.

If I need to use this fn anywhere else then I can reuse it.

- Great place to keep re-usable functions :- ↓
  - → Make a folder utils (It's utility, helper or common or whatever name).
  - → Inside create a file util.js and write the function filterData() inside it and make it named export.
  - → Any no. of helperfns can be kept inside util.js.
  - → Advantage :
    - * functions become readable & reusable
    - * This makes our code more testable because I can write seperate testcases for this.
    - * Maintainable because it is easy to debug.
    - * Modularity means we've broken down code into small pieces and every pieces have it's own responsibility.

Let us take <RestaurantMenu/> component.

Job of this component is to show resturant menu.

(i) find out restaurant id (resId)

(ii) get details of the restaurant from server.

(iii) to display.

We'll try to extract this logic.

We will create a custom hook that will help us to get the restaurant details.

[NB: Create a hook with "use" name infront of it]

* Create a custom hook useRestaurant.

→ Create a new file named useRestaurant.js:

```
* Const useRestaurant = (resId) => {
    Const [restaurant, setRestaurant] = UseState(null);
    //Get data from API
    UseEffect (() => { getRestaurantInfo(); }, []);
    async function getRestaurantInfo () {
        const data = await fetch (
          "https://www.swiggy..." + resId);
const json =await data.json ();
SetRestaurant (json.data);
}
//return restaurant Data
return restaurant;
};
export default useRestaurant;
```

Meanwhile, in <RestaurantMenu/> :

```
Const RestaurantMenu =() => {
    const {resId} = useParams ();
    const restaurant = UseRestaurant (resId);
    return !restaurant ? (<shimmer/>) :
     (
        ... );};
export default RestaurantMenu;
```

Next feature :→

Online & Offline

- If the user have no internet connection, then it should show "You are offline, check your internet connection".
- Else it should show the data.

In Body.js ↓

```
const Body = () => {
          isOnline
  const ~~offline~~ = useOnline ( );
       !isOnline
  if (~~offline~~) {
      return (
            <h1>Check your internet
                    connectivity </h1>
      );
  }
```

Now, we can create custom hook useOnline()

Scanned with CamScanner

useOnline.js

```
import {useState, useEffect} from "react";
const useOnline = () => {
  const [isOnline ,setIsOnline] = useState(true);
  useEffect (() => {
    window.addEventListener ("online", () => {
          setIsOnline (true);
    });
   Window.addEventListener ("offline", () => {
          setIsOnline (false);
   });
  }, []);
  return isOnline ;
}
export default useOnline ;
```

To fake offline

Go to chrome devtools → Network tab
→ Change speed option (fast, slow 3G, offline).

Important

* Cleaning cache

- Whenever eventListener is added, we should clean it up.
- ~~Because~~ [whenever you are going offline & getting back online, ~~a new~~ eventListener is created only once because we've empty dependency array].
- It is always a good practice to clear the eventListener when we go out of the component. otherwise browser will keep hold those.
- to do that :↓

```
useEffect(() => {
    const handleOnline = () => {
        setIsOnline (true);
    };
Window. addEventListener ("online", handleOnline);
window. addEventListener ("offline", () => {
        setIsOnline (false);
  });
  return () => {
   Window. removeEventListener ("online", handleOnline);
   window. removeEventListener ("offline", handleoffline);
};
}, []);
return isOnline;
```

This fn

handleoffline()

Scanned with CamScanner

For the whole code, parcel creates only one .js file.
In this file, all the ~~ft~~ code is bundled together.
So, the size of this index.js file is large.
But in production bundle, size of this file should be small.

There would be a 100s of components in a large website like "makemytrip".
Suppose if all these are bundled together in a single index.js file, It will blast. It will make our app very slow.

So, to build a large-scale production ready application, we should do :-

"CHUNKING"

It is also called as :-

→ Code Splitting
→ Dynamic Bundling
→ Lazy Loading.

We cannot bundle everything in our app.

→ On Demand Loading.

→ Dynamic Import

Making a new different bundle in our App

Let us create "Instamart"

→ create an instamart component.

→ In App.js file, do chunking :-

App.js :-

Instead of importing like this :-

⊛ import Instamart from "./components/Instamart";

Do lazy loading :-

⊛ Const Instamart = lazy (() =>
import ("./components/Instamart"));

So, now the "index.js" file in dist folder won't have code of instamart. It is created as seperate file while loading.
This is called ON-DEMAND LOADING.

"When you are loading your component in demand, react tries to suspend it".

So, when instamart is loaded for the first time, we see an error message on screen.
This is because, instamart file took 27ms to get loaded. But react tries to render it before it get loaded. That's why error.

Solution for this

"Suspense" → We can wrap instamart inside suspense.

App.js :-

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

React now knows that when there is a suspense, what will be loaded.

In the intermediate time, a shimmer should be shown.

So, there is a prop known as "fallback".

So, write :-

```
<Suspense fallback = {<Shimmer/>} >
        <Instamart />
</Suspense >
```

NB :-

* Never ever dynamically load your component inside another component.