Custom Hooks

Custom Hook is not mandatory but it will make your code more readable, more usable and modular.
↳ breaking into small component so that ~~it~~ will make each component having their own responsibility.
or single.

Custom Hooks are nothing but utility function or helper function so best place to create the Custom Hook is Utils.

⇒ Separate file for Separe Hook.
⇒ Name the file exactly same name as Hook
⇒ Name of Hook should start with "use"

In our code before Retaurant Menu Card is having two responsibility
~~So~~ - fetching data
- displaying

So we have fetched data using Custom hook which makes Restaurant Menu card single responsibility and now it is more modular and more testable.

To write any Hook first finalize the contract
↳ I/P to the Hook
↳ o/p ~~of~~ the Hook

Notes Provided by Akshay Sain is having all content for this lecture.

Please refer that.

~~Bundling~~

Chunking
Code splitting
Dynamic Bundling
Lazy loading
On Demand Loading.

To do this we are not importing Our Grocery ⓑ like previous

import Grocery from "./components/Grocery"; ✗

We will import using lazy function
↓
This lazy function takes the callback functn and that callback f^n ~~import~~ uses import function and that import f^n takes the path.

New way of importing using lazy loading.
↓
```
const Grocery = lazy(() => import("./components/Grocery"));
```

To work this we have to use Suspense, we will wrap our Component into suspense.

∵ When we click on Grocery code of Grocery is not there it takes some time to fetch but React is so fast that in that time it throws error. Suspence will solve it We will wrap our Grocery Component in Suspense

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

→ The time when Grocery component code is loading it will show this fallback code

TAILWIND Css

Command to Installe Tailwind CSS

```
npm install -D tailwindcss postcss
```

↳ tool for transforming css with Java Script.

If we have to give any hard coded value than we can use [ ] to provide the value

e.g. width of 200px is not available

We can use `w-[200px]`

This lecture is more practical. So have less Notes.

DATA IS THE NEW OIL

Higher Order Components

It's a function which takes a component and returns a component. after enhancing.

Higher order functn / component it returns a functn / component and component is a function that return some piece of JSX.

```
export const withOpenLabel = (ResturantCard) => {
    return (props) => {
        return (
                                          some css
            <div>
            <label className = " ...." > Open </label>
            <ResturantCard {...props}/>
            </div>
            )
    }
};
```

Accordian => Which we can expand and collapse

Acordion Header
Accordian Title
Accordian Body.

lifting the State

In Our App All Restaurant Catgories are maintaining their own state. and are uncontrolled component.

हमें Accordian बनाना है एक को click करें तो other one should collapse. for this we have to give power of deciding collapse and show to their parents instead of each child/Restaurant category. By lifting the state. Now, Parent component is controlling the child component and called as Controlled Component.

By doing this we want to make ~~showIndex~~ changes in ~~of~~ parent from child. We can't do this directly but we can do this from. indirect way.

```
{ categories. map ((category, index ) => (
    <Restaurant Category
    Key = { key++}
    data = {category?.card?.card}
    showList = {index === show Index && true }
    settingShow Index = {() => set show Index (index)}
    />
))}
```

Parent property
but changing it through passing as function.

In child we will access this as props and change it.

```
const hideShowList=()=>{
        settingshowIndex();
    }
```

⟹ Learn more from Notes of Akshay Saini

~~Context API~~

Sharing State B/w Components

Sometimes you want the state of two components to always change together. To do it remove state from both of them move it to their closest common parent, and then pass it down to them via props. This is known as lifting state up, and it's one of the most common things you will do writing react code.

Controlled & Uncontrolled Component

→ It is common to call a component with some local state "uncontrolled".

→ In contrast you may say a component is controlled when the important information in it is driven by props rather than its own local state. This lets parent component fully specify its behavior.

Props Drilling

Usually you will pass information from a parent component to a child component via props. But passing props can become verbose and inconvenient if you have to pass them through many components in the middle, or if many components in your app need the same information. Context lets the parent component make some information available to any component in the tree below - no matter how deep - without passing explicitly through ~~it~~ props.

Read More from Notes by Akshay Saini

There is some data which we needed anywhere in our app for that we use Context.

Context is like some global central Object. which can be accessible anywhere in the Object.

We ~~cant~~ put that data in Context which we needed anywhere in our Application. E.g User Name.

We did not put all the data in ~~Prof~~ Context otherwise what's the use of props then.

Creating a Context

```
import {createContext} from "react";
 Const UserContext = createContext({
     loggedInUser: "Default User",
 });
export default UserContext;
```

To use it we use hook. Named as {useContext}

```
const {loggedInUser} = useContext(UserContext);
console.log(loggedInUser);
```

How to use this Context in Class based Components:

In class based components we don't have hooks. We use Context

like

Logged In User

```
<UserContext.Consumer>
    {(data)=> console.log(data)}
</UserContext.Consumer>
```

JSX which is having call back fn and this call back fn gets access to the data.

How to Update Context

Let's use Dummy Data for this.

```
Const AppLayout =() => {
  const [UserName, setUserName] = useState();

   useEffect(() => {
     const data = {
       name : "Akshay Saini",
     };
     setUserName(data.name);
   }, []);
```

=> Dummy Data

~~set~~

```
return
  // Default
  <UserContext.Provider value =({ loggedInUser: userName}};
    {*/ //Akshay Saini */}   ← wrapped whole App- So whole App will have
                                updated value
    <div className = "app">
     <UserContext.Provider value = {{ loggedInUser: "Elon Musk"}}>
      {*/ Elon Musk */}   ← wrapped Header and updated only
                              that will have provide value
        < Header />
      <UserContext.Provider>
     <Outlet />
    </div>
  <UserContext.Provider >
 );
};
```

REDUX

* Redux is not mandatory in your application.
* Redux and React are different liabrianies.
* Redux is not the only library for state Management e.g zustand.
* Redux offers easy debugging.

⇒ Redux is a predictable State Container for JS Appr.
* Redux work in data layer. It is used to managing and centralising application state.

Redux Toolkit

The redux toolkit package is intended to be the standard way to write Redux Logic. It was originally created to help address three common concern about Redux.
* Configuring a Redux store is too complicated.
* I have to add a lot of package to get Redux to do anything useful
* Redux requires too much boiler plate code.

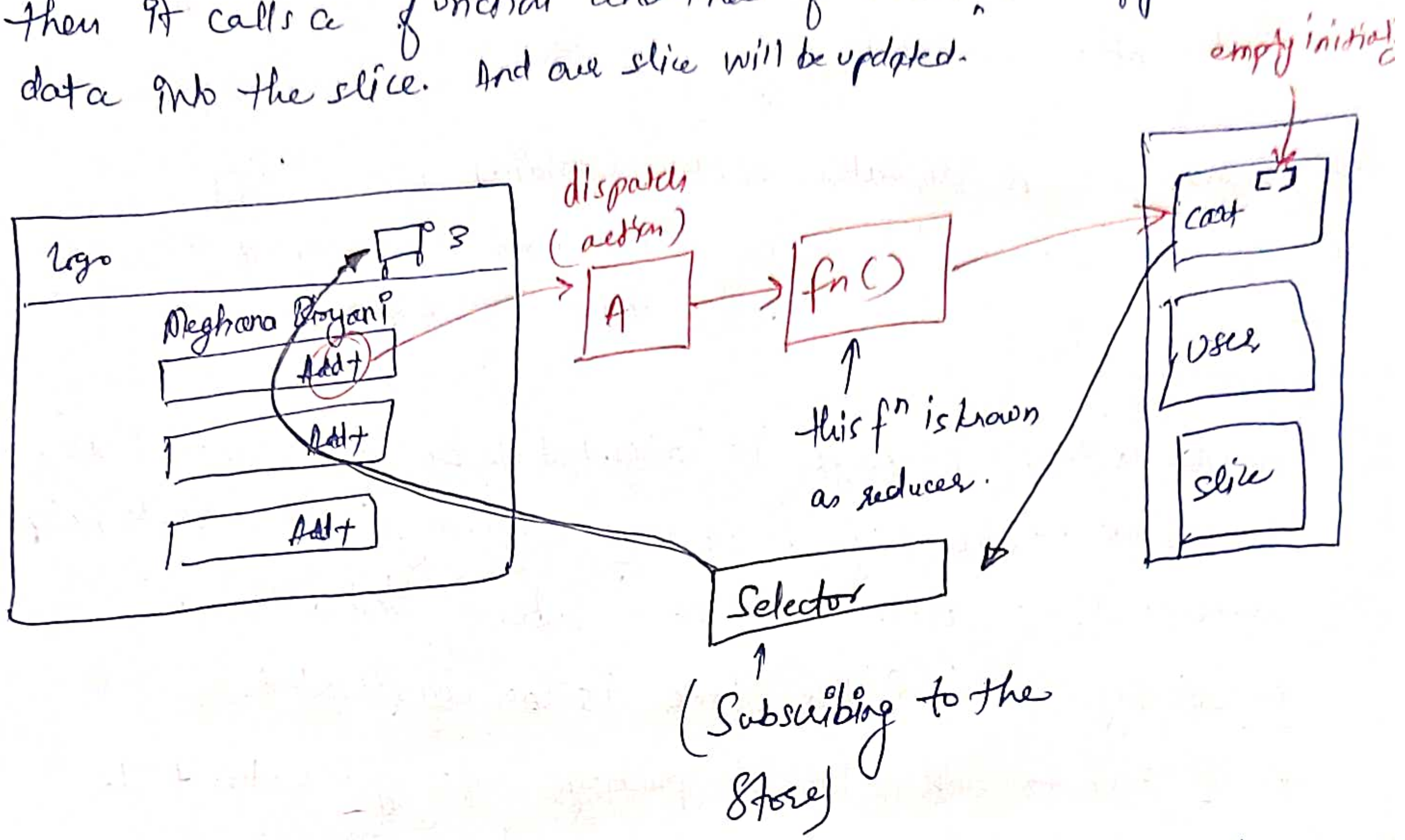
Using Redux we are building our own Cart

⇒ Redux store is kind of like very big JS object with lot of data inside it and it is kept in global central place.
* Is it good to keep all the data inside big ~~[illegible]~~ Redux store?
Yes it is absolutely fine. but so that our Redux store does not become very big very clumsy we have slices inside Redux Store.

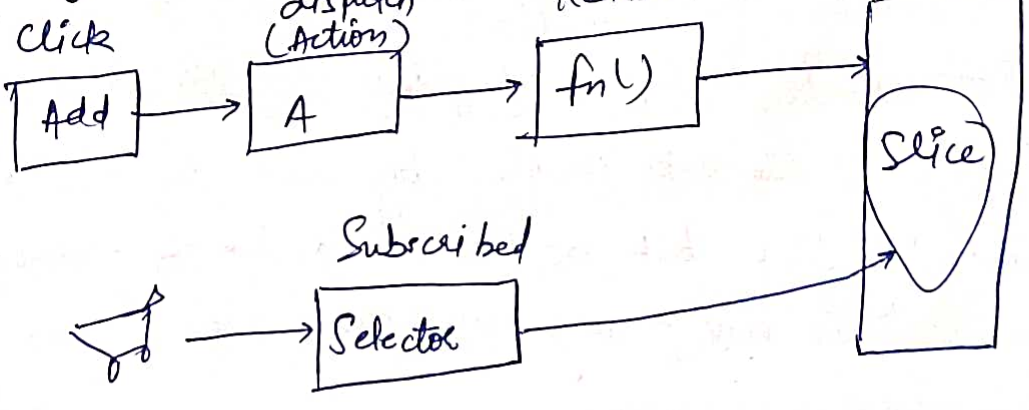
~~What~~ Whatever you wants to do there can be a slice for it in Redux Store.

We can't directly modify data into slice. There is a way.

- When we click on Add+ button it will dispatch an action and then it calls a function and that function internally modify the data into the slice. And our slice will be updated.



If the data inside the Cart changes, My Header component will update automatically. that is why we called Subscribing to the Store. My Header Component has subscribed to the ~~the~~ store using selector.



⇒ What we gonna do
- Installe @reduxjs/toolkit and react-redux
- Build our store
- Connect our store to our app
- Will create slice (cartSlice)
- dispatch(action)
- Selector.

Command
```
npm i @reduxjs/toolkit
npm i react-redux
```

we will pass our store as props.

*
```
<Provider store={ }>
      :
      :
</Provider>
```
Imported from react-redux library

* To create our store we will use configureStore();
↳ imported from @reduxjs/toolkit.

* To create Slice we use createSlice({});
it takes some configuration
```
const cartSlice = createSlice({
      name: 'cart',
      initialState: {
            items : []
      },
      reducers: {
            addItem : ()=>{}
      }
```
action

reducer fn.

We export action and reducers from this slice.

```
export const { addItem, removeItem, clearCart } = cartSlice.actions;
export default cartSlice.reducer;
```

Explanation of Code :

```
removeItem : (state, action) => {      ↙ We are mutating the state here.
      state.splice (action.payload, 1);
   };
```

This code is part of Redux reducer function, specially an action handler for 'remove

Redux Reducer

In Redux, a reducer is a function that takes the current state and an action as arguments and returns a new state. Reducers are used to specify how the application's state changes in response to actions sent to the store.

* Parameters

state : This represents the current state of the part of the Redux. store managed by this reducer

Action Payload

This contains the index of the item that should be removed from the state array. The 'payload' property is where the data passed with the action is stored.

state.splice (action. payload, 1); ← no. of elements to remove.

action.payload ↓ specifies the start index

state.splice ↓ This is an array method that changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

Similarly

no. of elements to remove ↓

state. splice (action. payload. index, 1, action. payload. item);

action. payload. index ↓ specifies the start index at which to begin changing that array.

action. payload. item ↳ New item that will be inserted at the specified Index.

```
Const appStore = configure Store({
    reducer: {
        cart: cart reducer
    },
});
```

reducer ← This is our whole main Appreducer and this reducer contains small reducers from slice.

// Subscribing to the store using a Selector:

```
Const ~~cart~~ cart Items = useSelector ((store) => store. cart. items);
```

← we will tell them what portion of store we need access.

useSelector ↓ use Selector Hook gives access to the store

We have useDispatch () hook for dispatching an action.

* Whenever you are using Selector. Make sure you are subscribing to the right portion of the store. If you don't subscribe to the right portion of the store it will be a big performance issue.

```
Const store = useSelector((store) => store);      } this is very
const CartItems = store.cart.items;                  less efficient.
              ↳ it is subscribed to whole store

const store = useSelector((store)=> store.cart.items)
   ↳                                ↳ More efficient.
   It will be affected only when this change. Nothing to do with other slice of store
```

Here we have nothing to do with whole store we have meaning from cart slice of store. If anything changes in store my store variable got affected.
We need to subscribe the selected portion of the store.

⇒ reducer for one big reducer
⇒ reducers for slice having multiple reducers.
⇒ While exporting we use reducer because exporting single reducer.

* In Vanilla (older) Redux ⇒ Don't Mutate State.

```
state.items.push(action.payload);  ← Prohibited in vanilla redux
                                     but valid in new Redux
So, In vanilla we do same like        Toolkit.

const newState = [...state];                 } In Redux Toolkit it is
newState.items.push(action.payload);         } done behind the scene
return newState;                             } using Immer library.
```

In Redux Toolkit, we have to mutate the state. and don't have to return anything earlier returning is mandatory.

Immer

Immer is a tiny package that allows you to work with immutable state in a more convenient way.

For Clear cart

state = [ ]; ← it will not works because we are not mutating the state we are changing the reference.

* In Redux

console.log(state) ← It will not works

console.log(current(state)) ← current comes from @redux/toolkit.

* while clear cart → Redux toolkit says either mutate the state or return a new state.

```
{ state. items . length = 0;
  ~~return []~~;
  return {items: []}; ← this new object will be
                        replaced inside original state.
```

* Redux dev Tools

↳ Very helpful in debugging

↳ It basically give console.log of every thing we are doing and alot more we can do.