React (Day-12)

Redux Toolkit

- It is used to store state & data
- Works on data layer of the application.

Note: React and Redux are different libraries.

- Redux is not compulsory, it can be used when there is a use case, and only when it is required.

→ Alternatives of redux: zustand, recoil, etc.

→ Our application becomes easier to debug using redux (Courtesy: redux dev tools)

So what exactly is Redux?

- A predictable state container for JS Apps

So redux is used with other JS libraries too, but popularly used with react.

→ So, we'll be using React-Redux & Redux Toolkit

* Redux or Vanilla Redux is the older way of writing redux.
* So, we'll be exploring the latest ways by using Redux Toolkit, along with React Redux which bridges the gap between react & redux.

→ Redux Toolkit is the standard way to write Redux logic.

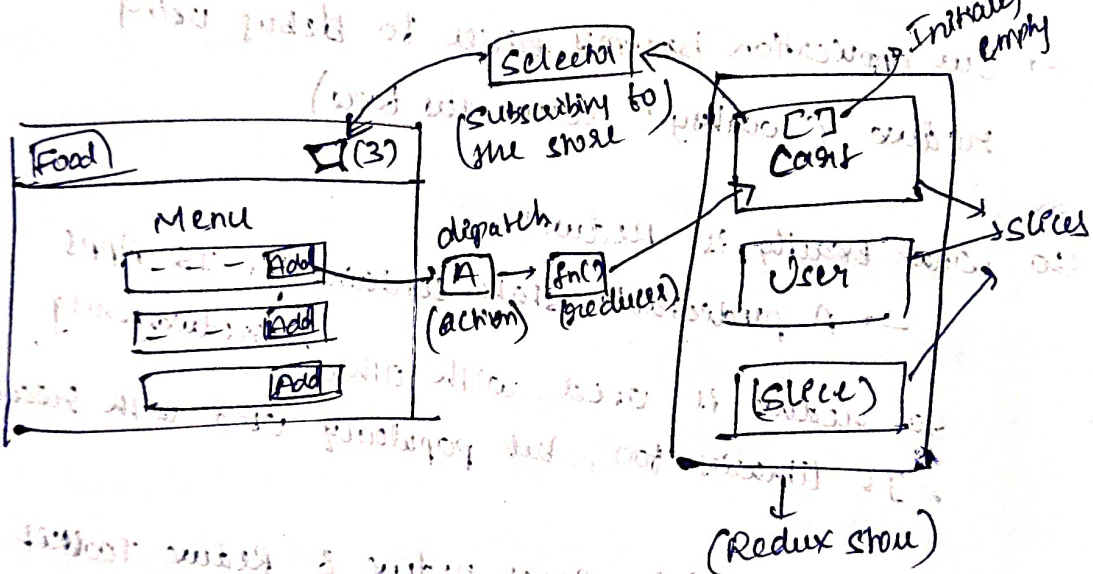
→ Redux was very complicated and used to come with a huge learning curve.

→ Redux Toolkit was created to help address three common concerns about Redux:

* Redux store config is too complicated
* Have to add lot of packages to make Redux do anything useful.
* Redux requires too much boilerplate code.

* So let's try to understand the redux architecture:-

→ Lets say we are building the cart feature for our food ordering app.



→ Redux store is like a big object that store a lot of data & states & allows all components to access / update the data / states.

→ Redux store divided into slices for different usecases in our project. Ex: (Cart, User, Theme, etc)

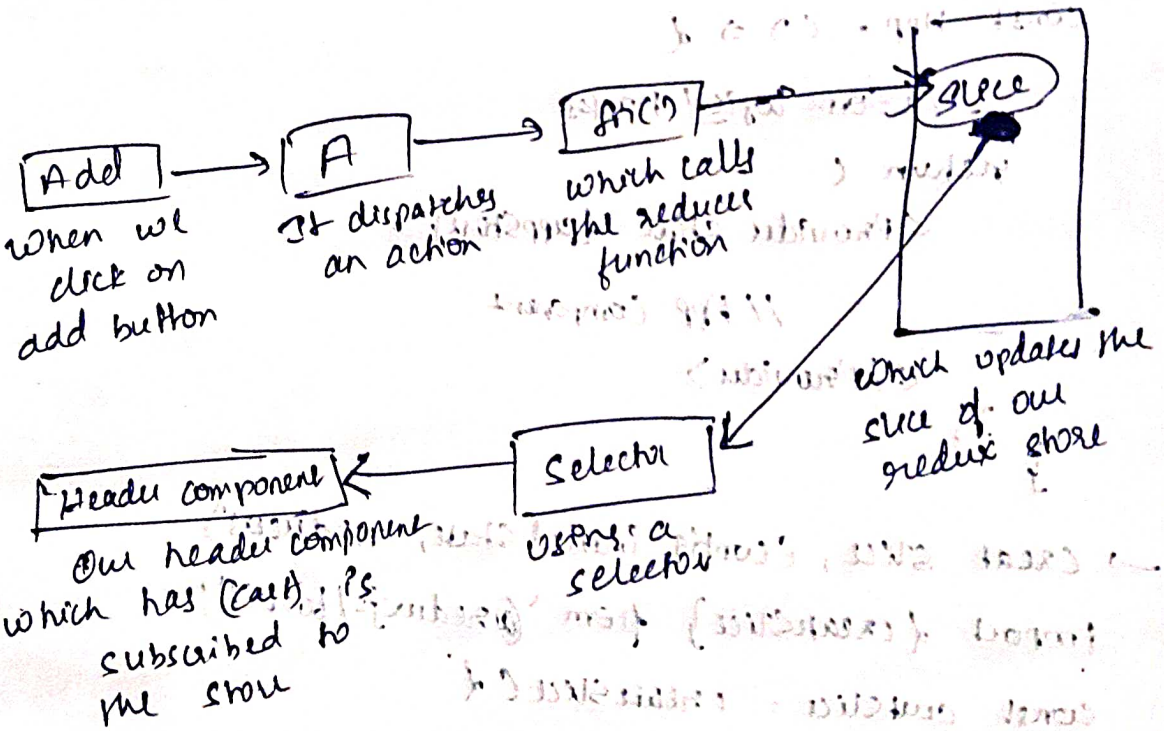
→ We can't update the data in store directly, so we (dispatch an action) which calls a (reducer function) which will update the data in the store. Ex: Adding item to cart.

→ Similarly, we can't read/access the data from store directly, we need to (subscribe to the store) using a (selector).

→ Subscribing to the store means, our component will now be dynamic with the data on the store & will be updated with the data on store.

Let's say our header subscribes to the store, then cart will show items dynamically from the data received through the store

→ Our component will be in sync with the store on subscribing to the store using selectors.



→ So, now we will be doing :-

* Install @reduxjs/toolkit and react-redux
* Build our store
* Connect our store to our app
* Slice (cart slice).
* dispatch (action)
* Selector (subscribe to store)

Configuring, creating & setting up our store

→ Create a store :-

appStore.js

```
import {configureStore} from '@reduxjs/toolkit';

const appStore = configureStore({});

export default appStore;
```

→ Wrap whichever component needs the access to store, with Provider

App.js

```
import {Provider} from 'react-redux';
import appStore from './utils/appStore';

const App = () => {
  // Some logic / hooks
  return (
    <Provider store={appStore}>
      // App Component
    </Provider>
  );
};
```

→ Creat slice, (config, initial state, reducers)

```
import {createSlice} from '@reduxjs/toolkit';

const cartSlice = createSlice({
  name: 'cart',
  initialState: {
    items: []
  },
  reducers: {
    addItem: (state, action) => {
      state.items.push(action.payload);
    },
    clearCart: (state) => {
      state.items.length = 0;
    }
  }
});
```

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

↳ Exporting actions & reducers
- ↳ actions to use in components, which will eventually dispatch & call reducer.
- ↳ reducers to add in the appStore
- : to let appStore (central store) know about the cartSlice.

Like this :- (appStore.js)

```
import { configureStore } from '@reduxjs/toolkit';
import cartReducer from './cartSlice.js';
const appStore = configureStore({
    reducer: {
        cart: cartReducer
        // can have other slices and their reducers too
        // like this ↓
        // user: userReducer
    }
});
export default appStore;
```

* Subscribing to the store :-
In any component, you can use useSelector:-

```
const cartItems = useSelector((store) => store.cart.items);
// now we can use this
```

* Using dispatch and calling reducer :-
In any component, on any event, we can dispatch an action which will call a reducer.

```
const dispatch = useDispatch();
const handleAddItem = (item) => {
    dispatch(addItem(item));
}
```

} : of useDispatch from 'react-redux' addItem-reducer from cartSlice.js

* Now handleAddItem() can be used as a callback function for any event listener in a component.

Ex: `<button onClick={() => handleAddItem(item)}></button>`

item can be taken from the component which should be added on button click

Note: All bridging between react & redux is done by 'react-redux'

So, we have Provider, useSelector, useDispatch from 'react-redux'

All core redux functionalities like configureStore, createSlice are imported from '@reduxjs/toolkit'.

Some imp things about redux :-

1) Be very careful and conscious while subscribing to the ~~cart~~ store

→ Here we are only subscribing to the cart items. So changes to cart items will result in changing / re-rendering our subscribed component.

Ex: `const cartItems = useSelector(store => store.cart.items);`

→ But, if we subscribe to the whole store and then use the cart items :-

```
const store = useSelector(store => store);
const cartItems = store.cart.items;
```

This might look exactly same as above example and it will even work fine.

→ But there will be a lot of performance gap between the first & second examples.

→ In the second one, we are subscribed to the whole store, so any changes to the store (user, them, etc if they exist) will affect our component resulting in unnecessary render cycles.
↳ which will degrade our app performance.

→ So, it is important to note that, we should only subscribe to the part of store, that concerns our component.

2) There might be confusion regarding usage of singular/plural forms of keywords like actions & reducer in store & slice configs.
We can't help it, just remember that redux defined those keywords in that way & we should remember it that way.
→ reducer in store config
→ reducers in slice config
→ reducer (counterSlice.reducer) in export (slice)
→ actions (counterSlice.actions) in export (slice)

3) Previously (Traditionally) in vanilla Redux, modifying the state in reducer was prohibited as the state was immutable.

So, we had to do something like,

```
addItem (state, action) => {
    const newState = [...state];
    newState.items.push(action.payload);
    return newState;
}
```

→ We had to create a copy of state, modify the copy, & then return the copy from reducer.

→ But now, in redux toolkit, we no longer are prohibited from mutating the state.

→ Infact, we "have" to modify the state in reducer.

→ By the way, even now redux toolkit performs the logic of modifying immutable state but it now abstracts that logic using "Immer"

↳ It is a tiny package that allows us to work with immutable state in more convenient way.

→ So, the whole process of old way of handling immutable state in redux is still carried out, but behind the scene.

↳ We as developers can directly mutate the state in the reducer function.

Note: In the reducer function, if you want to debug and use console.log to log the state, it will give some object which isn't in readable stream.

↳ Proxy(Object)

So, we can log the state in readable stream using "current"

⇒ console.log(current(state));

Note: Regarding mutating the state:

→ We can do it like this (which we've been doing)

(i) state.items.length = 0

→ or like this too :-

(ii) return {items: []}.

↳ returning the updated value to the state.

→ but not like this :-

(iii) state.items = []

↳ this will just change the reference of the local variable 'state' to an empty array from the original state.

Note: Explore :-

(i) Redux dev tools (for better debugging experience and a lot of features & options)

(ii) RTK Query (for advanced data fetching capabilities and caching options)