Redux: Continued

Combining Reducers

Redux allows multiple reducers to be merged into one, which is sent to the creatStore using a function called **combineReducers.**

The way we link reducers is very simple, we create a file for the reducer in the reducer directory. We will also create a file called index.js in the reducer directory.
In the Index.js file, we import the combineReducers function from Redux and we also import all the individual Reducer files.

We then call the combineReducers function and send it as an argument that contains all the different reducers.

combineReducers: This method allows us to create a **root reducer** automatically which combines child reducers into a single reducing function. This gathers the result of every child object into a single state object. It automatically takes two arguments that are state.object and action.
The reducers passed to the combineReducers function must pass three conditions:

- For an action that is not recognised, it must return the **state** given as the first argument.
- It must never return undefined. It is very easy to accidentally do this via the initial return statement, so if you do this then the **combineReducer** throws, instead of letting the error appear elsewhere.
- If the given state is undefined, it must return the **initial state** for this specific reducer. According to the previous rule, the initial state should

not be undefined. It is easy to specify this with ES6 optional arguments, but you can also explicitly check the first argument for being undefined.

Currying

Curying is an advanced technique of working with functions. It is used not only in JavaScript but also in other languages. Currying is the **transformation** of a function that makes a function callable as **f (a, b, c)** from **f (a) (b) (c).**

In other words, when a function, instead of taking all the arguments at once, takes the first and takes the second function, returns a new function and returns a new function that takes the third argument, and after that until Not, all arguments are fulfilled.

Uses of Curry Functions:

- It helps to avoid the use of the same variable.
- It helps in event handling.

Syntax:

```
function Myfunction(a) {
   return (b) => {
      return (c) => {
        return a * b * c
        }
       }
    }
```

Middleware

Redux middleware provides a third-party extension point between sending an action and the time when it reaches the reducer. People use Redux

middleware for crash reporting,logging, an asynchronous API, routing, and more.

Why do we need middleware?

Redux is **synchronous** in itself, so how do network requests such as **Async operations** work with Redux? Middleware is useful here. As discussed earlier, the reducer is the place where all execution arguments are written. The reducer has nothing to do with how long it is taking, how long it is taking or logging in the status of the app before and after the action starts.

In this case, the Redux middleware function provides a means of interacting with the action being sent to them before reaching the reducer. Customized middleware functions can be created by writing higher-order functions (a function that returns another function), which wrap around some logic. Several middlewares can be added together to add new functionality, and each middleware has no idea about what came before and after.

Redux Thunk

Thunk is a functional programming technique used to delay computation. Now instead of doing some work, you produce a function body or an invaluable expression ("thunk"), which can be optionally used for later work.

Thunk In Redux:

In React / Redux, thunks enable us to directly avoid causing wrong effects in our action creators, actions, or components. Instead, anything impure can be wrapped inside thunk. Later, that thunk would be invited by the middleware to actually cause the effect. By moving our side effects to run (at the middleware level) at one point of the Redux loop, the rest of our apps remain relatively pure.

What is the major use of thunk?

When we install our thunk package using ***npm-install-thunk***, it finally helps our action creators to return functions. This further helps in **API calls** or **event handling.**

When thunk is implemented by middleware, it performs an asynchronous effect. When that async is complete, the callback or handler can send a normal action to the store. So thunk lets us temporarily "escape" from the normal loop, along with an async handler that eventually re-enters the loop.

Summarising It

Let's summarise what we have learnt in this module:

- Learned about combining reducers in detail.
- Learned about currying and middleware.
- Learned about Redux Thunk in detail.

Some References:

● combineReducers
https://redux.js.org/api/combinereducers

● Middleware
https://redux.js.org/understanding/history-and-design/middleware

● Thunk
https://medium.com/fullstack-academy/thunks-in-redux-the-basics-85e538a3fe60