Redux

lunes, 31 de octubre de 2022 17:29

* Flux implementations

Redux A Flux-like library that achieves modularity through functions instead of objects.

↳ Based on Flux. 99 lines of code.

↳ It is Flux-like: it has actions, action creators, a store, and action objects that change state through.

↳ Not exactly Flux: removes the dispatcher, and application's state is a single immutable object. Introduces reducers, not part of Flux.

↓

pure functions that return a new state based on the current state and action.

```
(state, action) => newState
```

→ Redux stores a state in one place, a single store.

(How can we achieve this with different types of data?

↓

A single source of truth : The Redux Store

→ The State Tree is an object where each key is a branch of the tree.

* Actions

→ Actions provide instructions about what changes in the app state along with necessary data.

→ Actions are the only way to update the state of a redux app.

→ In a redux app, instead of object-oriented, we think verb-oriented.

→ Once you identify the actions needed to change state, you can list them in a file called

Constants.js

```
const constants = {
    SORT_COLORS: "SORT_COLORS",
    ADD_COLOR: "ADD_COLOR",
    RATE_COLOR: "RATE_COLOR",
    REMOVE_COLOR: "REMOVE_COLOR"
}
export default constants
```

→ An action is a Javascript object that has a minimum field for type

```
{ type: "ADD_COLOR" }
{ type: "ADD_COOLOR"}
```

→ an action type is a string that describes what should happen.

→ It's easy to make mistakes with strings This type of mistake does not trigger warnings, so it's hard to spot.

```
import C from "./constants"

{ type: C.ADD_COLOR }
```

This prevents from making such mistakes, because a typo in a javascript object throws an error in the browser.

* Action Payload Data

→ actions are Javascript objects that provide the instructions to change state. Most changes require some data: the action's payload.

↓

which record should I remove?
what new info is added to a post?

→ When we dispatch an action like RATE_COLOR, we need to know what color

to rate and with what rating. This information can be passed with the action in the same object.

```
{
    type: "RATE_COLOR",
    id: "a5685c39",
    rating: 4
}
```

→ action object

* Reducers

→ Reducers are functions used to update the state tree.

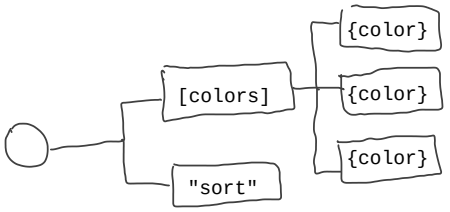
↳ are functions that take the current state along with an action as arguments and use those to create and return a new state.

↳ Designed to update a specific part of the state tree: leaves or branches.

↓

We can combine reducers into one

→ In an app of color rating, we have the state tree:



A separate reducer will be used to handle each part of the state tree.

↳ Each reducer is simply a function

↓

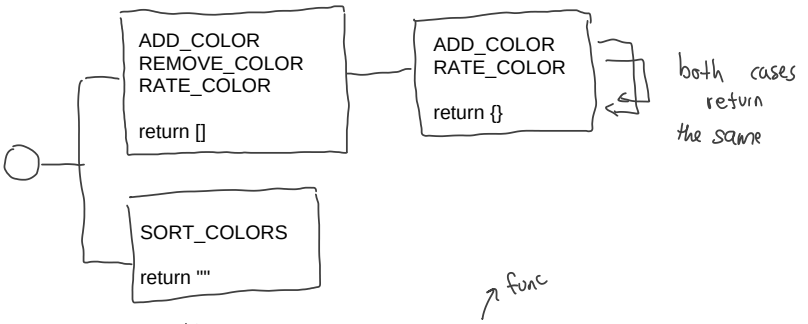
if we put them together

```
import C from "../constants"

export const color = (state={}, action) => {
    return {}
}

export const colors = (state=[], action) => {
    return []
}

export const sort = (state="", action) => {
    return ""
}
```

The returned value and initial state correspond to their data type in the state tree.

- param state and return state must be the same type

→ Each reducer is designed to handle only the actions necessary to update its part of the state tree



both cases return the same

→ Each reducer is combined into a single reducer that will use the store

↑ func

→ Both the color and colors reducer handle ADD_COLOR and RATE_COLOR but each reducer handles a different part of the tree.

↓

ADD_COLOR: an array with a new color

↖ each reducer focuses on what a specific action means for its part of tree

ADD_COLOR: ← a new color object with the input attributes

```
export const color = (state = {}, action) => {
     switch (action.type) {
          case C.ADD_COLOR: ← handle each action type
               return {
                    id: action.id,
                    title: action.title,
                    color: action.color,
                    timestamp: action.timestamp,
                    rating: 0
               }
          case C.RATE_COLOR:
               return (state.id !== action.id) ?
                    state :
                         {
                         ...state,
                         rating: action.rating
                         }
          default :
               return state
     }
}
```

A common way of coding reducers is with a switch

Reducers should always return something. The default state returns the current state

→ ADD_COLOR: returns a new object with the payload data

RATE_COLOR: returns a new color object with the desired rate

```
const action = {
     type: "ADD_COLOR",
     id: "4243e1p0-9abl-4e90-95p4-8001l8yf3036",
     color: "#0000FF",
     title: "Big Blue",
     timestamp: "Thu Mar 10 2016"
}
console.log( color({}, action) ) // create a new color
```

Let's look at the Colors Reducer:

```
export const colors = (state = [], action) => {
     switch (action.type) {
          case C.ADD_COLOR :
               return [
                    ...state,
                    color({}, action)
                    ]
          case C.RATE_COLOR :
               return state.map(
                    c => color(c, action)
                    )
          case C.REMOVE_COLOR :
               return state.filter(
                    c => c.id !== action.id
                    )
          default:
               return state
     }
}
console.log( colors(currentColors, action) )
```

→ Color reducer is designed to manage as on the colors branch of the state tree.

→ Colors reducer manages the entire colors branch.

ADD_COLOR: returns a new array to concat the new color

RATE_COLOR: returns a new array with the desired color rated.

REMOVE_COLOR: returns a new array without the desired color to delete.

↑ an array of { } id, title, color, TS

↑ an { } with type, id, title, color, TS

→ The sort reducer is used to change the Sort State variable

→ To recap, state updates are handled by reducers.

→ Reducers are pure functions that take the current state and an action as arguments

→ Modularity is achieved by reducers

* The Store

→ The store is what holds the app's state data and handles all the state updates.

→ Flux allows many Stores, Redux only one.

→ The store handles the updates by passing the current state and action through a single reducer: combine all your reducers

combineReducers() → combines all your reducers into one. These reducers are used to build your state tree. The names of the fields match the names of the reducers that are passed in.

```
import { createStore, combineReducers } from 'redux'
import { colors, sort } from './reducers'

const store = createStore(
     combineReducers({ colors, sort })
)
console.log( store.getState() )
// create a stor with init value
const store = createStore(
     combineReducers({ colors, sort }),
     initialState ──→ {} with the same structure
)
console.log( store.getState().colors.length ) // 3
console.log( store.getState().sort )
// "SORTED_BY_TITLE"
```

Output

```
{
  colors: [],
  sort: "SORTED_BY_DATE"
}
```

↑

→ The only way to change the state of your application is by dispatching actions through the store.

→ The store has a dispatch() method that takes actions as an argument.

→ When you dispatch an action through the store, the action is sent through the reducers and state is updated.

```
console.log(
"Length of colors array before ADD_COLOR",
store.getState().colors.length
)
// Length of colors array before ADD_COLOR 3
store.dispatch({
     type: "ADD_COLOR",
     id: "2222e1p5-3abl-0p523-30e4-8001l8yf2222",
     title: "Party Pink",
     color: "#F142FF",
     timestamp: "Thu Mar 10 2016 01:11:12 GMT-0800 (PST)"
})
console.log(
"Length of colors array after ADD_COLOR",
store.getState().colors.length
)
```

Output 3

Output 4