REDUX

Ori Calvo, 2017

oric@trainologic.com

https://trainologic.com

Redux Introduction

- Managing data in web applications has become extremely complex
- Fetching data from server, updating models and views might result in a total mess

State Management Patterns

- Design patterns for managing and synchronizing UI
- Popular patterns/libraries
  - Flux
  - Redux
  - Ngrx
  - Mobix
  - Home made

Redux Introduction

- Based on Flux ideas
- Single store
- Immutable store
- Components dispatch actions
- Reducers reacts to actions by changing state
- Subscribers are notified

Redux Introduction- the state

- The state is a **single** JavaScript object which contains all data inside application

State shape example

```
{
    visibilityFilter: 'SHOW_ALL',
    tasks: [
        {
                text: 'Consider using Redux',
                status: true,
        },
        {
                text: 'Keep all state in a single tree',
                status: false
        }
      ]
}
```

1st Principal Rule

- The first principle refers to the change of data in an application

- Every change in the application’s data, including the data and the UI state, is contained in a single object, called **the state** or the **state tree.**

2nd Principal Rule

- The second principle of Redux is that the state tree **is read only.**
- You cannot modify or write to it
- Instead, anytime you want to change the state, you need to **replace the data (actions)**

2nd Principal Rule

Initial state

```
const initialState = {
    tasks:[
        {
            text: 'Clean the House',
            completed: true,
        }
    ]
};
```

```
const taskApp = (state = initialState,action)=>{
    switch (action.type){
        case ADD_TASK:
            return Object.assign({},state,{
                tasks:[
                    ...state.tasks,
                    {
                        text: action.text,
                        complete: false
                    }
                ]
            });
    }
};
```

Function that takes the action type and when task is add it replace the entire state with a new state that also contains the new data

3rd Principal Rule

- An action is a plain JavaScript object, describing the changes in the application. any data that gets into the Redux application gets there by actions.

```
{
    type : 'Add Task',
    index : 0
}
```

3rd Principal Rule

- The third principal refers to a pure functions
- Pure functions are the functions whose returned value depends solely on the values of their arguments
- Pure functions are predicted

```
const incrementByOne = (number)=>{
     number += 1;
     return number
}
```

3rd Principal Rule

- state change, is handled by a function that takes the previous state of the app, the action being dispatched, and returns the next state of the app.

- This function has to be pure

3rd Principal Rule

```
const taskApp = (state = initialState,action)=>{
    switch (action.type){
        case ADD_TASK:
            return Object.assign({},state,{
                tasks:[
                    ...state.tasks,
                    {
                        text: action.text,
                        complete: false
                    }
                ]
            });
    }
};
```

Pure function that takes the current state and action
Return a new object with the new add task.
Result is expected

Actions

- Actions are JavaScript objects which by convention holds a ‘type’ property that specify the action description
- The actions role is to send a notification in order to change the state

Actions

- The structure of the action is up to us and can contain additional property according to the application state structure

Simple action example

```
{
    type : 'Add Task',
    index : 0
}
```

Actions creators

- Action creators are simply functions that return an action object

Action task example

```
const addTask = ()=>{
     return{
        type:'ADD_TASK',
        index: 0
     }
};
```

- Action creators are useful for asyc operations will be learned in the future

Action dispatch

- Dispatch is a redux method which send the action's object to the reducer function which according to the action type, changes the state

```
store.dispatch(addTask('Clean the House'));
store.dispatch(addTask('Feed the Dogs'));
store.dispatch(addTask('Play Guitar'));
store.dispatch(addTask('Buy Flowers'));
```

Action Example

- Consider a task list application which the user can add tasks, mark the tasks status and eventually filter tasks according to user selection

Action Example

./actions/actions.js

```
export const ADD_TASK = 'ADD_TASK';
export const ENTER_TASK = 'ENTER_TASK';
export const SHOW_TASK = 'SHOW_TASK';

export const showTaskOptions = {
    SHOW_ALL: 'SHOW_ALL',
    SHOW_COMPLETED: 'SHOW_COMPLETED',
    SHOW_ACTIVE: 'SHOW_ACTIVE'
};
```

First it will be good practice to store actions types as constant

```
export function addTask(text) {
    return { type: ADD_TASK, text }
}

export function enterTask(index) {
    return { type: ENTER_TASK, index }
}

export function showTask(filter) {
    return { type: SHOW_TASK, filter }
}
```

Secondly, creating an actions container for each type with an additional property to pass data to the state

Reducers

- Reducers are pure functions that takes the **previous state** and an **action** as arguments and returns the new state according to the action's type

Reducers

- In addition, reducers can sometimes update just a portion of the application’s state object
- For example, one reducer will handle the state’s tasks list object while other reducer will handle the change of task status

Reducer example

./reducers/task_reducer

```
import {ADD_TASK,
        ENTER_TASK,
         SET_SHOW_TASK,
        SHOW_TASK_OPTIONS} from '../actions/actions';

const initialState = {
    showOption: showTaskOptions.SHOW_ALL,
    tasks:[]
};

const taskApp = (state = initialState,action)=>{
    switch (action.type){
        case SET_SHOW_TASK:
            return Object.assign({},state,{showOption:action.filter});
        default:
            return state
    }
};
```

Importing the action's types

Creating initial state

Reducer example

- TaskApp is the reducer's name and it contains a switch function that will behave according to the action type

- If none action is taken or an unknown action will occur the reducer will by default return the current and unchanged state

Reducer example - immutability

- The most important rule of a reducer is that **reducers should not be mutated**
- Remember, reducers must be pure functions that the return value is expected
- Pure functions are easier to track down and will result in less errors

Reducer example - immutability

- That is why the example used the Object.assign() method

- The method will create a new different object that will take the state object and change the specific part respectively

Reducer example- handling actions

- Of course the reducer can handle several actions

Case for using the add task action to add task to the tasks list

```
case ADD_TASK:
    return Object.assign({},state,{
        tasks:[
            ...state.tasks,
            {
                text: action.text,
                complete: false
            }
        ]
    });
```

```
case ENTER_TASK:
    return Object.assign({},state,{
        tasks: state.tasks.map((task,index)=>{
            if(index === action.index){
                return Object.assign({},task,{
                    complete: !task.complete
                })
            }
            return todo
        })
    });
```

Accessing a task in order to change its status

Conceptual aside

- Notice the strange syntax over the ADD_TASK case? ('…')?
- it's the new EcmaScript 6 spread operator
- The spread operator let us add to a list, other list's items

```
const listA = [1,2,3,4];
const listB = [...listA,'item1','item2']; // => [1,2,3,4,'item1','item2']
```

Reducer - reducer splits

- Reducers might handle lots of cases which result in a long code block

- It's a good practice to separate the reducer to small reducers that handle non related state fields

Reducer – reducer composition

- As mentioned actions are triggers that notify the reducer that something needs to be changed in the state

- Then, the reducer might change the entire state or might change only a part of the state

Reducer – reducer composition

- The pattern of a reducer consist of several cases that each case change part of the state is called **reducer composition**

reducer splits - example

- before splitting the reducer, it is crucial to understand which state's fields are related to each other

- The shown example suggest that ADD_TASK and ENTER_TASK are related and can be separated from the SET_SHOW_TASK

reducer splits - example

./reducers/task_reducer

```
const taskHandlerReducer = (state = [],action)=>{
    switch (action.type){
        case ADD_TASK:
            return [
                ...state,
                {
                    text: action.text,
                    complete: false
                }
            ];
        case ENTER_TASK:
            return state.map((task,index)=>{
                if(index === action.index){
                    return Object.assign({},task,{
                        complete: !task.complete
                    })
                }
                return task;
            });
        default:
            return state;

            }
    };
```

Both cases are dealing with actions that reflect changes only on the tasks property (which is an array)

reducer splits - example

./reducers/task_reducer

```
const {SHOW_ALL} = SHOW_TASK_OPTIONS;

const showOptionHandlerReducer = (state = SHOW_ALL,action)=>{
    switch(action.type){
        case SET_SHOW_TASK:
            return action.filter;
        default:
            return state;
    }
};
```

The other split will result in a reducer which handles the other part of the state which is the property showOption

reducer splits - example

- Finally, combine the splits into the root reducer like so

./reducers/task_reducer

```
const taskApp = (state={},action)=>{
    return {
        showOption: showOptionHandlerReducer(state.showOption,action),
        tasks: taskHandlerReducer(state.tasks,action)
    }
};
```

The root reducer takes state as an empty object and action, then it returns an object which consist of two state's properties with the split reducers as values.
When the user will trigger an action, each split reducer will handle the action, if the split reducer wont recognize the action it will return the current state unchanged part

CombineReducers

- A helper function which takes the split reducers and gathers them result into a single object

```
import { combineReducers } from 'redux';
const taskApp = combineReducers({
    taskHandlerReducer,
    showOptionHandlerReducer
});
```

=

```
const taskApp = (state={},action)=>{
    return {
        showOption: showOptionHandlerReducer(state.showOption,action),
        tasks: taskHandlerReducer(state.tasks,action)
    }
};
```

Store

- Store is an object that combine the actions and the reducer together in order to change the application state

Store – conceptual aside

- A redux application will use **only** one store

- If there's a need to separate the application logic it will be only through code splitting and reducers composition as learned

Store's abilities

- The store:
  - Storing the current state
  - Allow access to state with getState()
  - Allow updating the state with dispatch(action)
  - Registers listeners with subscribe(listener)
  - Unsubscribe a listener

getState()

- Will return the current state that the store holds

```
store.getState()
```

dispatch(action)

- The only function that is able to change the current state
- When calling dispatch(), the reduce function of the store will be called with two argument which are the getState() and the action
- Eventually return the new state

```
store.dispatch(addTask('Clean the House'))
```

subscribe(listener)

- Listens to the state’s change
- Will be called anytime an action will dispatch
- The listener is a callback function which invoked when an action dispatches and the state has been changed

```
store.subscribe(()=>{
    console.log(store.getState());
});
```

Store

- To create a store simply npm install redux and import createStore from redux

```
import { createStore } from 'redux';
import  taskApp  from '../reducers/task_reducer';

let store = createStore(taskApp);
```

Redux: Actions, Reducers, Store

- The example shows how to use the store:

```
import { createStore } from 'redux'
import taskApp from './reducers/tasks_reducers'

import{addTask,
       enterTask,
        showTask,
       SHOW_TASK_OPTIONS} from './actions/actions';


let store = createStore(taskApp);

console.log(store.getState());

let usubscribe = store.subscribe(()=>{
    console.log(store.getState());
});

store.dispatch(addTask('Clean the House'));
store.dispatch(addTask('Feed the Dogs'));
store.dispatch(addTask('Play Guitar'));
store.dispatch(addTask('Buy Flowers'));
store.dispatch(enterTask(0));
store.dispatch(enterTask(1));
```

Will follow any change and unsubscribed the listener

What will be the eventual state?

Presentational & containers components

- React-redux support the idea of separating presentational and containers components
- that pattern is useful especially because it makes the application easier to understand
- With the separation pattern we can reuse the presentational components with a whole different state sources

Containers components

- containers will deal with the logic behind the presentational components
- Will deal with the how things work
- Are stateful, and will provide the states data to the respective presentational component

Presentational components

- Concerns with how things should be presented
- Stateless components, they are rarely connected to any state
- Receive data and callbacks exclusively via props

Async Actions

- Todays applications usually communicate with a server, requesting for data
- Some of those requests might take time to fetch so it will return a promise
- We must have a tool to handle those kind of actions

Async Actions

- For asyc actions, there are two crucial time stamps that a sync actions should inform the reducer to change the current state
  - When requesting the data from the server
  - When receiving the data from the server
  - When getting an error from the server

Async Actions

```
const requestUsersPosts = (subreddit) => {
    return {
        type: REQUEST_POSTS,
        subreddit
    }
};
```

For example we might consider the next example when we use action creators to notify the reducer that the state should be change when requesting the data and receiving the requested data

```
const receiveUsersPosts = (subreddit, json) => {
    return {
        type: RECEIVE_POSTS,
        subreddit,
        posts: json.data.children.map(child => child.data),
        receivedAt: Date.now()
    }
};
```

Async Actions - reducer

- There are no change in creating the reducer which is simply gets actions via dispatch to change the state

Async Action Creator – Thunk actions

- An action that return a function of sorts will be considers as a thunk action
- The returned function does not have to be pure and can also contain an asyc request from a certain API

Async Action Creator – Thunk actions

```
export const fetchPosts = (subreddit) => {
    return dispatch => {
        dispatch(requestUsersPosts(subreddit));

        return fetch(`https://www.reddit.com/r/${subreddit}.json`)
            .then(response => response.json(),
                error => console.log('An error occured.', error))
            .then(json => dispatch(receiveUsersPosts(subreddit, json)))
    }
};
```

This is an example of an action which returns an async function to fetch users data from reddit. When the promise resolves, the response turns into json which then dispatches a sync action

Async Action Creator – Thunk actions

- Thunk actions have another useful feature as they can dispatch results of each other

```
export const fetchPostsIfNeeded=(subreddit) => {
    return (dispatch, getState) => {
        if (shouldFetchPosts(getState(), subreddit)) {
            return dispatch(fetchPosts(subreddit))
        } else {
            return Promise.resolve();
        }
    }
};
```

Returns true or false

The example reflects a use case where a thunk action returns a function which dispatches another thunk action that will eventually result in an async call

Redux Thunk middleware

- The Thunk middleware is a separate library that can be use to handle thunk actions

- Thunk middleware enables function creators to not only return objects but also function

- Will be cover in details later

Redux Thunk middleware

```
import thunkMiddleware from 'redux-thunk';
import { createStore, applyMiddleware } from 'redux';
import { selectSubreddit, fetchPosts } from './actions/actions';
import rootReducer from './reducers/reducer';

const store = createStore(
    rootReducer,
    applyMiddleware(
        thunkMiddleware,
    )
);
store.dispatch(selectSubreddit('Reactjs'));
store
    .dispatch(fetchPosts('Reactjs'))
    .then(() => console.log(store.getState()));
```

Applying middleware with the applyMiddleware() method which will consis a specific thunkMiddleware to handle the asyc actions

Middlewares

- With the help of a middleware, a Redux store can handle asyc actions

- By enhancing the store with the help of the method applyMiddleware() we can use some asyc middleware to handle asyc actions

Middlewares

- In general, a middleware is a code block that sits between the request and the response in order to manipulate the response before generate it

Redux Middleware

- When it comes to Redux, a middleware is a function that takes an action, and according to the actions type, shape or other factors can manipulate the action

Chainning Middlewares

- applyMiddleware method chains middlewares
- To apply a middleware to the store we can import applyMiddleware() from the Redux library
- The method takes middleware as arguments to handle functionality to the application

applyMiddleware

```
import thunkMiddleware from 'redux-thunk';
import { createLogger } from 'redux-logger';
import { createStore, applyMiddleware } from 'redux';
import { selectSubreddit, fetchPosts } from './actions/actions';
import rootReducer from './reducers/reducer';

const loggerMiddleware = createLogger();

const store = createStore(
    rootReducer,
    applyMiddleware(
        thunkMiddleware,
        loggerMiddleware
    )
);
store.dispatch(selectSubreddit('Reactjs'));
store
    .dispatch(fetchPosts('Reactjs'))
    .then(() => console.log(store.getState()));
```

The create store is modified with two chained middleware which one will handle asyc actions and the other one will log any state change

Custom Middleware

Custom Middleware

```
export default function createLogger({ getState }) {
    return (next) =>
        (action) => {
            const console = window.console;
            const prevState = getState();
            const returnValue = next(action);
            const nextState = getState();
            const actionType = String(action.type);
            const message = `action ${actionType}`;
            console.log(`%c prev state`, `color: #9E9E9E`, prevState);
            console.log(`%c action`, `color: #03A9F4`, action);
            console.log(`%c next state`, `color: #4CAF50`, nextState);
            return returnValue;
        };
}
```

Custom Middleware

- A simple custom logger which will log out all the actions, previous and new states of the application
- It accepts a state via the applyMiddleware which then
- and returns the a variable with the next parameter which will return the next chained middleware function or the main dispatch action

Normalizing State Shape

- The shape of the state's structure is crucial for a Redux application
- Its important that the state's data structure will not repeat the data it contains
- Deeply nested data might re-renders unrelated UI components for the parent object needs to change as well

Normalizing State Shape

```
const stateBadShape = [
    {
        id : "post1",
        author : {username : "user1", name : "User 1"},
        body : "......",
        comments : [
            {
                id : "comment1",
                author : {username : "user2", name : "User 2"},
                comment : ".....",
            }

        ]
    },
    {
        id : "post2",
        author : {username : "user2", name : "User 2"},
        body : "......",
        comments : [
            {
                id : "comment3",
                author : {username : "user3", name : "User 3"},
                comment : ".....",
            },

        ]
    }

];
```

Consider the next state shape as a bad practice structure.

1) It has repeatable objects
2) UI components that renders due to change in the parent components will eventually re render even if the change only occur on child object like 'comment'

Normalizing State Shape

- basically treats the application’s store like a database and keeping the data in a normalize form

Normalizing State Shape

Table posts

```
posts : {
    byId : {
        "post1" : {
            id : "post1",
            author : "user1",
            body : "......",
            comments : ["comment1", "comment2"]
        },
        "post2" : {
            id : "post2",
            author : "user2",
            body : "......",
            comments : ["comment3", "comment4",
"comment5"]
        }
    },
    allIds : ["post1", "post2"]
},
```

Table Comments

```
comments : {
    byId : {
        "comment1" : {
            id : "comment1",
            author : "user2",
            comment : ".....",
        },
        "comment4" : {
            id : "comment4",
            author : "user1",
            comment : ".....",
        },

    },
    allIds : ["comment1", "commment4"]
},
```

Table users

```
users : {
    byId : {
        "user1" : {
            username : "user1",
            name : "User 1",
        },
        "user2" : {
            username : "user2",
            name : "User 2",
        }
    },
    allIds : ["user1", "user2", "user3"]
}
```

The example shows a good practice to normalize the state's data

Normalizing State Shape

- When comparing both example the difference is clearly visible

- Each item is defined in one place which means that only one place needs to be updated according to the id pointer to the table's key

Normalizing State Shape

- Also, when changing a child object will not re-render any UI component that rely on the parent object

```
comments : {
    byId : {
        "comment1" : {
            id : "comment1",
            author : "user2",
            comment : ".....",
        },
        "comment4" : {
            id : "comment4",
            author : "user1",
            comment : ".....",
        },

    },
    allIds : ["comment1", "commment4"]
},
```

```
posts : {
    byId : {
        "post1" : {
            id : "post1",
            author : "user1",
            body : "......",
            comments : ["comment1", "comment2"]
        },
        "post2" : {
            id : "post2",
            author : "user2",
            body : "......",
            comments : [
                             "comment3", "comment4",
                              "comment5"
                           ]
        }
    },
    allIds : ["post1", "post2"]
},
```

Trade-off using normalization

- Normalization is an important part of structuring the state's shape

- However, one must understand the downside of normalizing the data

Trade-off using normalization

- Consider both examples:

Not normalized data

```
const whatsapp = [
    {
        group1: {
    contact1: {name: 'asaf', number: 888},
    contact2: {name: 'ori', number: 999}
    }},
    {
        group2:{
        contact1:{name:'itay',number:777},
        contact2: {name:'gilad',number:555}
    }}
];
```

How much effort will it takes to grab a contact's number from each one of the structures?

Normalized data

```
const whatsAppNrmlz = {
    byGroup:{
        'group1':[
            'asaf',
            'ori'
        ],
        'gourp2':[
            'gilad',
            'itay'
        ]
    },
    byContactName:{
        'asaf':888,
        'ori':999,
        'itay':777,
        'gilad':555
    }
};
```

Trade-off using normalization

- It seems like it will be much harder to grab data from the normalized structure

- Which is exactly its down-side:

Not normalized data

```
console.log(whatsapp[0].group1.contact1.number);
```

Normalized data

```
console.log(whatsAppNrmlz.byContactName[whatsAppNrmlz.byGroup.group1[1]]);
```

fetching data from a normalized structure, a pointer must be used to fetch the data