DAY8

Virtual DOM

Normal dom:

Relods the whole page

Html

Head body

Title script meta table

Tr

Td td

React js:

Dose not reload the whole page

Manipulated the data, the whole structure

Dose not reload the DOM

Yield – can return anything

Generator

function \*anotherGenerator(i){

yield i+1;

yield i+2

yield i+3

}

function \*myGen(i){

yield i

yield \*anotherGenerator(i)

yield i+10

}

var gen=myGen(10)

console.log(gen.next().value)

console.log(gen.next().value)

console.log(gen.next().value)

console.log(gen.next().value)

console.log(gen.next().value)

console.log(gen.next().value)

// function \*idMaker(){

// var index=1011;

// while(true)

// yield index++;

// }

// var gen=idMaker()

// console.log(gen.next().value)

// console.log(gen.next().value)

// console.log(gen.next().value)

// console.log(gen.next().value)

// console.log(gen.next().value)

// function \*generatorMessage(){

// yield 1;

// yield 2;

// yield 3+10

// console.log("END")

// }

// const gen=generatorMessage()

// console.log(gen.next().value)

// console.log(gen.next())

// console.log(gen.next())

// console.log(gen.next())

// function \*generateForLoop(num){

// for(let i=0;i<num;i++){

// yield console.log(i)

// }

// }

// const genLoop=generateForLoop(10)

// genLoop.next()

// genLoop.next()

// genLoop.next()

// function generateForLoop(num){

// for(let i=0;i<num;i++){

// console.log(i)

// }

// }

// generateForLoop(10)

class Foo{

\*[Symbol.iterator](){

yield 1;

yield 2;

}

}

class Another{

\*[Symbol.iterator](){

yield 'a';

yield 'b';

}

}

console.log(Array.from(new Foo))

console.log(Array.from(new Another))

class Foo{

\*[Symbol.iterator](){

yield 1;

yield 2;

}

}

class Another{

\*[Symbol.iterator](){

yield 'a';

yield 'b';

}

}

console.log(Array.from(new Foo))

console.log(Array.from(new Another))

MIDDLEWARE

REDUX MIDDLEWARE:

Redux Middleware. Middleware provides a way to interact with actions that have been dispatched to the store before they reach the store's reducer. Examples of different uses for middleware include logging actions, reporting errors, making asynchronous requests, and dispatching new actions.

MIDDLE WARE SISTS BETWEEN dispatch and reducers.

The middleware sits in between the dispatch and reducers, which means we can alter our dispatched actions before they get to the reducers or execute some code during the dispatch. An example of a redux middleware is redux-thunk which allows you to write action creators that return a function instead of an action

Redux Middleware. Middleware provides a way to interact with actions that have been dispatched to the store before they reach the store's reducer. Examples of different uses for middleware include logging actions, reporting errors, making asynchronous requests, and dispatching new actions.

Why to use middleware:

1. Logging
2. Authentication request
3. Caching
4. Error tracking

As expressed above, actions go through middleware before getting to the reducers, which gives us a great way of applying logic or filters to all actions. This means that the logic is grouped in one place instead of being spread across reducers, that we can easily identify where to investigate if a bug occurs, and we can swap that code out if we ever need to.

Some use-cases that benefit from using middleware:

Logging: every action goes through this middleware, so we can log its type and payload for debugging or tracking purposes.

Error tracking: if any asynchronous action returned an error, this middleware can display a notification.

Caching: Only call your API for the same action once, and cache the result for future calls.

Auth requests: For API calls, apply an authentication token before sending out the request.

So much more :)

Writing a middleware

To define your own middleware, you need to write a function with the following signature:

store => next => action => result

This looks very confusing at first glance - I hear you - so let’s break it down a little bit:

store is the Redux store instance that will be passed to your middleware.

next is a function that you need to call with an action when you want to continue the flow execution, which means passing the action to the next in line: either the following middleware or a reducer.

action is the action that was originally dispatched so that you can access it, apply logic based on the action, and eventually pass it on using next.

result is the value used as the result of the dispatch call.

-------------------------

Finally, to apply this middleware to the Redux store, you need to call applyMiddleware when creating the store through createStore(). Here’s a nice example from the official Redux documentation:

import { createStore, combineReducers, applyMiddleware } from 'redux'

let todoApp = combineReducers(reducers)

let store = createStore(

todoApp,

// applyMiddleware() tells createStore() how to handle middleware

applyMiddleware(logger, crashReporter)

)

In the example above, the middleware logger will be called first, followed by the crashReporter middleware since this is the order in which they were passed to applyMiddleware.