Xseed

Curso de React _ Technisys

Xseed

Clase 12 -

Generators - Redux Sagas

by Diego Cáceres

Repaso

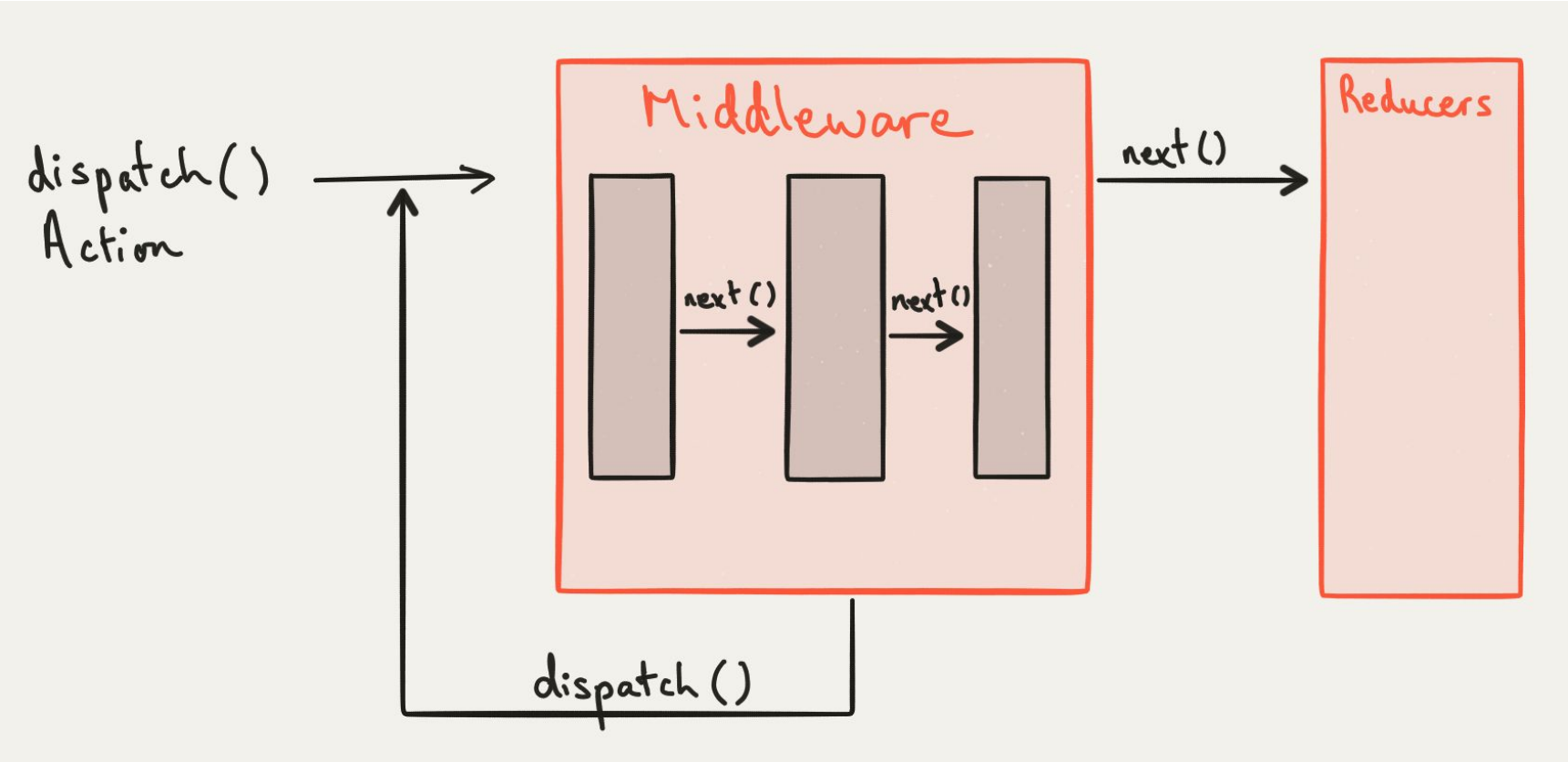
Redux Middleware

Los **Middlewares** son un punto de extensión en Redux, ya que nos proveen una forma de interactuar con todas las acciones que son emitidas al Reducer.

Hay varios ejemplos de uso, como loguear las acciones, reportar errores, hacer llamadas asíncronas, o dispatchear (emitir) nuevas acciones.

Los **Middlewares** se encadenan uno detrás de otro, y son llamados en orden para todas las acciones que se emitan. Tienen la posibilidad de modificar la Acción, o incluso de detener su propagación, osea, que no se envíe al siguiente **Middleware** ni al Reducer.

Redux Middleware



Redux Middleware - Implementación

Básicamente, es una función que recibe una acción e interactúa de alguna forma con ella:

```
function middleware(action) { /*...*/ }
```

Cómo funcionan en cadena, cada Middleware le debe enviar la acción al siguiente en la cadena, y si fuera el ultimo, seria al Reducer:

```
function middlewareWrapper(nextDispatch) {
 return function(action) {
     /*...*/
     nextDispatch(action);
 }
}
```

Redux Middleware - Implementación

Pero además, cada Middleware tiene acceso a una copia del Store (osea, tiene acceso a la función Dispatch y al getState para ver el State). Esto se logra siendo que la función final de un Middleware tiene otro wrapper:

```
function storeWrapper(store) {
 function middlewareWrapper(nextDispatch) {
   return function(action) {
     nextDispatch(action);
   }
 }
}
```

=

```
function middleware(store) {
 return function(nextDispatch) {
   return function(action) {
     nextDispatch(action);
   }
 }
}
```

=

```
const middleware = store => nextDispatch => action => {
 nextDispatch(action);
}
```

Redux Middleware

Para usar Middlewares se componen las distintas funciones utilizando un método de Redux llamado 'applyMiddleware', y dándole el resultado al createStore de Redux. En este caso, `middlewareOne` será el que reciba las acciones primero, y `middlewareTwo`, como es el último, es el que dispatcheara las acciones al Reducer.

```
import { applyMiddleware, createStore } from "redux";

const store = createStore(
 reducers,
 initialState,
 applyMiddleware(
   middlewareOne,
   middlewareTwo
 )
);
```

Redux Middleware – Redux Thunk

Para resolver los Side Effects en Redux, podemos utilizar un Middleware en particular llamado Redux Thunk Middleware.

Esta librería permite emitir un nuevo tipo de acciones, que son funciones, y se conocen como Thunks**.** El Middleware cuando detecta que es una función simplemente la invoca en vez de enviarla a los Reducers, y esta función tiene la posibilidad de "dispatchear" más acciones (ya sean ***thunks*** o acciones comunes).

Se crearán en el mismo lugar que las acciones hasta ahora, en un archivo llamado `actions.js` o similar.

```
export function fetchPosts() {
 // Thunk middleware se encarga de invocar la función con el dispatch.
 return function (dispatch) {
   // Dispatch para avisar que comienza la llamada a la API (loading).
   dispatch(requestPosts());
   return axios.get(`https://www.myapi.com/getPosts)
     .then(
       response => {
           console.log('Success getting posts.', response);
           dispatch(receivePosts(response.data));
       },
   // Pasamos una función para el error en vez de utilizar un catch
   // porque el catch agarra errores del dispatch tambien
       error => {
           console.log('An error occured.', error)
           dispatch(receivePostsFail(error));
       }
     );
 }
}
```

```
function requestPosts() {
 return {
     type: REQUEST_POSTS
 }
}

function receivePosts(posts) {
 return {
   type: RECEIVE_POSTS,
   payload: {
     posts: posts,
     receivedAt: Date.now()
   }
 }
}
```

Redux Middleware – Redux Thunk

La implementación de este Middleware es bastante simple, una versión simplificada del mismo sería así:

```
const thunkMiddleware = ({ dispatch, getState }) => next => action => {
 if (typeof action === 'function') {
   return action(dispatch, getState);
 }

 return next(action);
}
```

Xseed

ES6 Generators

Generators

Estamos acostumbrados a que las funciones en Javascript siempre se ejecutan hasta que se completan: ‘run to completion’.
En este caso por ejemplo, por más que el callback trate de ejecutarse, no puede interrumpir el loop muy largo de la función foo (Javascript es single-threaded, solo un comando o función se puede ejecutar en un momento dado).

```
setTimeout(function(){
 console.log("Hello World");
},1000);
function foo() {
 for (var i=0; i<=1E10; i++) {
     console.log(i);
 }
}
foo();
// 0..1E10
// "Hello World"
```

Generators

Sin embargo, con **Generators** tenemos un nuevo tipo de funciones, que pueden ser pausados en el medio de su ejecución, y continuadas **luego**, permitiendo que otro código se ejecute en estas pausas. El contexto de la función se mantiene entre las diferentes pausas.

Las funciones **Generators** son Cooperativas, lo que significa que ellas deciden cuándo permitir una interrupción y de esta forma cooperar con el resto del programa. Dentro del cuerpo de la función, podemos utilizar la nueva palabra clave '**yield**' para pausar la función desde dentro. Nada puede pausar un Generator desde fuera, pausa cada vez que encuentra un **yield**.

Sin embargo, una vez pausada, no puede continuar su ejecución sola, sino que necesita que un control externo le indique que debe continuar.

Generators - Declaración

Los **Generators** se declaran con el carácter especial * que se coloca inmediatamente luego de la palabra **function**. (es válido tanto contra function, o contra el nombre del Generator que estamos declarando)

```
function* foo() {

    // ......

}
```

```
function *foo() {

    // ......

}
```

Generators - yield

Cuando vemos una línea del estilo “**yield** ____” se denomina una “expresion yield” o “*yield expression*”. No es un statement, porque cuando se retome la ejecución de esta función, se enviará un valor hacia adentro, y eso que enviemos será computado como el resultado de la expresión **yield** (en este ejemplo, lo sumará a 1 y lo guardará en x).

Al momento de pausarse, envía hacia afuera el valor “foo”, osea que con los **Generators** tenemos una comunicación en 2 sentidos, “*2-way communication*”

```
function* foo() {
    var x = 1 + (yield "foo");
    console.log(x);
}
```

Generators - yield

- **yield** es utilizado para pausar la función, y puede enviar un valor hacia afuera (si no se declara nada, se envia undefined).
- En **yield** también podemos recibir un valor desde fuera al momento de retomar la ejecución.
- **yield*** es utilizado para delegar la ejecución a otra función de tipo **Generator.**

```
function* foo() {
 let x = yield 'Please give me a value for x'
 let y = yield 'Please give me a value for y'
 let z = yield 'Please give me a value for z'

 return (x + y + z)
}
```

Generators - Como ejecutar un function*

Cuando ejecutamos un Generator, nos devuelve un “Generator Iterator”, un iterador para poder controlar la ejecución de ese Generator, invocando el método next() sobre el. Al llamar a next(), ejecutará la función hasta el primer o siguiente yield.
El next() nos devuelve un objeto con propiedades value y done, que indica si el Generator se terminó de ejecutar:

```
function *foo() {
      yield 1;
      yield 2;
      yield 3;
      yield 4;
      yield 5;
}
```

```
let it = foo();
var message = it.next();
console.log(message); // { value:1, done:false }

console.log( it.next() ); // { value:2, done:false }
console.log( it.next() ); // { value:3, done:false }
console.log( it.next() ); // { value:4, done:false }
console.log( it.next() ); // { value:5, done:false }
// Tecnicamente aun no termino, podriamos pasar un valor a yield 5
console.log( it.next() ); // { value:undefined, done:true }
```

Generators - Como ejecutar un function*

Cuando retornamos algo desde un Generator, el valor vendrá en value luego de la última interacción:

```
function* foo() {
 let x = yield 'Give me a value for x'
 let y = yield 'Give me a value for y'
 let z = yield 'Give me a value for z'
 return (x + y + z)
}
```

```
let generatingFoo = foo()
//Tenemos el objeto iterable
console.log( generatingFoo.next() )
//{ value: 'Give me a value for x', done: false }
console.log( generatingFoo.next(8) )
//{ value: 'Give me a value for y', done: false }
console.log( generatingFoo.next(4) )
//{ value: 'Give me a value for z', done: false }
console.log( generatingFoo.next(8) )
//{ value: 20, done: true }
```

Generators - Como ejecutar un function*

Aca tenemos otro ejemplo, que puede ser un poco entreverado al comienzo.

- Cuando enviamos 12, lo sustituye por el ‘yield (x + 1)’ y por eso y queda en 24.
- Cuando enviamos 13, lo sustituye por el ‘yield (y/3)’ y por eso z queda en 13
- Luego suma 24 + 13 + 5 = 42.

```
function *foo(x) {
 var y = 2 * (yield (x + 1));
 var z = yield (y / 3);
 return (x + y + z);
}
```

```
let it = foo( 5 );

// No enviamos nada al `next()` aca
console.log( it.next() );       // { value:6, done:false }
console.log( it.next( 12 ) );   // { value:8, done:false }
console.log( it.next( 13 ) );   // { value:42, done:true }
```

Generators - Como ejecutar un function*

- Podemos utilizar un for..of loop para recorrer las pausas de un Generator, pero en este caso el valor que se return es descartado:

```
function *foo() {
 yield 1;
 yield 2;
 yield 3;
 yield 4;
 yield 5;
 return 6;
}

for (var v of foo()) {
 console.log( v );
}
// 1 2 3 4 5
```

Generators - Usos

Un uso común para Generators es para funciones asíncronas, que llaman a next cuando la se resuelve la promesa.

```
function request(url) {
 axios.get(url).then((reponse) => {
   it.next(response);
 })
}
function* main() {
  const result1 =  yield request('http://somapi.com');
  const result2 = yield request('http://some.otherapi?id='  + result1.id );
  console.log('Your response is: '  + result2.value);
}
var it = main();
it.next()
```

Xseed

Redux Saga

Redux Saga

En Redux solo podemos manejar acciones síncronas. Para agregar side effects, podemos usar Redux-Thunk, o Redux Saga que es otra alternativa.

redux-saga es una librería que trata de solucionar los Side Effects en React/Redux de una forma mejor y más sencilla.

Podemos considerar a una 'saga' como un thread separado en nuestra aplicación que solo se encarga de los Side Effects, y como es un Middleware, tiene acceso al store, y por ende, a emitir nuevas acciones, y escuchar las acciones que se emiten.

Está basada en Generators, pero el Middleware se encarga de pausarlo y reanudarlo por nosotros. Esto hace que nuestro código asíncrono, se vea más sincronico, haciendolo mas facil de leer, escribir y testear.

Repositorio y Documentación.

Redux Saga

El Setup es bastante sencillo:

```
import { createStore, applyMiddleware } from 'redux';
import createSagaMiddleware from 'redux-saga'
import reducer from './reducers';
import rootSaga from './sagas';

//Creamos el middleware
const sagaMiddleware = createSagaMiddleware()
//Se lo damos al store
const store = createStore(
 reducer,
 applyMiddleware(sagaMiddleware)
)

// Iniciamos la Saga
sagaMiddleware.run(rootSaga)
```

Redux Saga

En el archivo principal de Sagas, vamos a dividir las Sagas en dos categorías, **Workers** y **Watchers**

- Una saga **Watcher** se encarga de observar todas las acciones emitidas al Store de Redux y si el type coincide con la acción que le corresponde, se la asigna a un Worker Saga. Normalmente desde el archivo de sagas, vamos a exportar una combinación de todos los watchers.

- La **Worker** saga es la que se encarga de procesar esa acción, y realizar todas las demás acciones que sean necesarias.

Redux Saga

En el archivo principal de Sagas (sagas.js en este caso) vamos a tener varias sagas creadas, por lo tanto necesitamos exportar una función rootSaga que inicie todas a la vez.

```
// Iniciamos todas las sagas a la vez
export default function* rootSaga() {
 yield all([
   watchGetProducts(),
   watchCheckout()
 ])
}
```

Redux Saga

```
//En sagas.js
import { types, receiveProducts } from './actions'
import * as api from './api'

//WATCHER SAGA
export function* watchGetProducts() {
     yield takeEvery(types.GET_ALL_PRODUCTS,getAllProducts)
}

//WORKER SAGA
export function* getAllProducts() {
     const products = yield call(api.getProducts)
     yield put(receiveProducts(products))
}
```

takeEvery y **call** son metodos de redux-saga que veremos mas adelante.

```
//WATCHER SAGA
export function* watchCheckout() {
 while(true) {
   yield take(types.CHECKOUT_REQUEST)
   yield call(checkout)
 }
}

//WORKER SAGA
export function* checkout() {
 try {
   const cart = yield select(getCart);
   yield call(api.buyProducts, cart)
   yield put(checkoutSuccess(cart))
 }
 catch(error) {
   yield put(checkoutFailure(error))
 }
}
```

No esta mal visto utilizar **while(true)** dentro de un watcher, ya que es un **Generator** que se pausara.

Redux Saga - Helpers

Contamos con determinadas funciones proporcionadas por la librería para utilizar en nuestras sagas:

- **takeEvery:** Toma todas las acciones que coinciden con el type definido, e invoca la saga que se pasa en el segundo parámetro. Permite concurrencia.

```
function* watchAction() {
 yield takeEvery('ACTION', workerSagaToRun)
}
```

- **takeLatest:** Toma todas las acciones que coinciden con el type definido y ejecuta la saga correspondiente, pero cancela la saga anterior si aun esta corriendo.

```
function* watchLastAction() {
 yield takeLatest('ACTION', workerSagaToRun)
}
```

Redux Saga - Effect Creators

Los Effect Creators retornan un objeto plano de Javascript que no hace nada, pero describe una acción a realizar para el Middleware. La ejecución de la acción es realizada por el Middleware. Los helpers y los effects, los importamos desde 'redux-saga/effects'

- **call:** ejecuta una función, si retorna una promesa, pausa la saga hasta que se resuelva. Luego de la función, podemos indicar con qué parámetros invocarla.
- **put**: Emite una accion (dispatch).
- **select:** Nos permite acceder al state de Redux
- **take, fork, race, spawn, join, cancel** son otros efectos disponibles.

```
const state = yield select()
```

```
function* effects() {
 let result = yield call(fnToRun, optionalArgsToPassToFn)
 yield put(actionToDispatch(result))
}
```

Redux Saga - Effect Creators

El effect **take** es similar al **takeEvery**, pero en vez de disparar siempre una función cuando llega una acción que coincida, nos permite a nosotros decidir qué hacer luego de que llegue la acción, e incluso no escuchar más por esa acción.
En este ejemplo, la aplicación esperaría a que el usuario agregue sus primeros 3 ToDos antes de felicitarlo:

```
import { take, put } from 'redux-saga/effects'

function* watchFirstThreeTodosCreation() {
 for (let i = 0; i < 3; i++) {
   const action = yield take('TODO_CREATED')
 }
 yield put({type: 'SHOW_CONGRATULATION'})
}
```

Redux Saga - Effect Creators

Otro ejemplo, si queremos implementar un flujo para el Login y otro para Logout, utilizando **takeEvery** (o redux thunk) tendríamos que crear dos tasks. Sin embargo, utilizando **take**, podemos dejar esta responsabilidad en la misma tarea:

```
function* loginFlow() {
 while (true) {
   yield take('LOGIN')
   // ... Toda la logica del login
   yield take('LOGOUT')
   // ... toda la logica del Log out
 }
}
```

Redux Saga - Effect Creators

El effect **race** indica a redux-saga que inicie una carrera entre dos o más efectos, y el resultado contendrá el que se resolvió primero. En este ejemplo, si se emite un action CANCEL_FETCH antes de que se resuelva la llamada a la api, la primera se cancelara y se retorna un objeto con una clave ‘cancel’. Sino, un objeto con una clave ‘response’.

```
import { take, call, race } from `redux-saga/effects`
import fetchUsers from './path/to/fetchUsers'

function* fetchUsersSaga {
 const { response, cancel } = yield race({
   response: call(fetchUsers),
   cancel: take(CANCEL_FETCH)
 })
}
```

Redux Saga - Testing

Xseed

```
//Application
import { call } from 'redux-saga/effects'
import Api from '...'
function* fetchProducts() {
 const products = yield call(Api.fetch, '/products')
 // ...
}
```

```
//Test
import { call } from 'redux-saga/effects'
import Api from '...'

const iterator = fetchProducts()
//expects a call instruction
assert.deepEqual(
 iterator.next().value,
 call(Api.fetch, '/products'),
 "fetchProducts should yield an Effect call(Api.fetch, './products')"
)
```

```
// Effect -> llamar a la funcion Api.fetch con
`./products` como argumento
{
 CALL: {
   fn: Api.fetch,
   args: ['./products']
 }
}
```

Redux Saga - Ventajas

- El código se ve más ‘sincrónico’, y por ende es más fácil de seguir paso a paso el proceso.
- Permite escribir flujos más complicados.
- Es simple de escribir tests sobre las Sagas.
- Las sagas se pueden componer (yield*).
- Los Action Creators siguen siendo puros, y siempre retornan objetos planos.
- Aisla los side-effects a una sola area de la aplicación (archivo de sagas).
- Hay varios helpers y buena documentación.

Ejercicios

Ejercicios

Descargar el zip `EjerciciosClase12`. Cada carpeta dentro corresponde a un ejercicio y es un proyecto creado con Create React App, por lo que para ejecutarlo es necesario:

- Si usan yarn:
  - `$ yarn install`
  - `$ yarn start`
- Si usan npm:
  - `$ npm install`
  - `$ npm start`

Se debería abrir automáticamente el explorador con el ejercicio corriendo en http://localhost:3000/

Ejercicio 1

El ejercicio ya contiene redux, es un simple contador.

- Primero se debe conectar correctamente el componente Counter para que emita las acciones increment y decrement.
- Luego crear un archivo llamado sagas.js y agregar la primera saga:

```
export function* helloSaga() {
 console.log('Hello Sagas!')
}
```

- Luego se debe configurar en index.js el middleware de redux-saga y correrlo, importando la saga de sagas.js.

Ejercicio 1 - continuación

- Luego agregar un nuevo botón en Counter que invoque una nueva función de props llamada ‘onIncrementAsync’. El boton puede decir ‘Increment after 1 second’
  - Se debe crear un nuevo action creator correspondiente para utilizar en el mapDispatchToProps para esta nueva función.

- Luego en sagas.js crear dos nuevas sagas:
  - Un watcher saga que escuche cada action con type ‘INCREMENT_ASYNC’ e invoque la saga worker. (hay un helper de redux-saga llamado delay: delay(1000))
  - Un worker saga que cuando lo invoque espere 1 segundo, y luego emita un action ‘increment’

- Exportar ahora un ‘rootSaga’ que haga un yield del efecto ‘all’ para iniciar tanto el helloSaga como el watcher creado. Corregir en index.js para iniciar este rootSaga.

Ejercicio 2

Este ejercicio parte de la solución del ejercicio 2 de la clase 11, implementado con redux-thunk.

- Hay que migrar el proyecto a redux-sagas.
  - Instalar la librería
  - Crear el archivo de sagas.js
  - Configurar en index.js el store para utilizar este middleware.
  - Comentar los actionCreators que retornan una funcion (thunks).
  - Modificar los componentes que dispatcheaban los thunks, para que simplemente emitan acciones de tipo Request (FETCH_USERS_START, FETCH_POSTS_START)
  - Crear las sagas correspondientes para que todo vuelva a funcionar.