Generadores y Redux-saga

Generadores en Javascript

Concepto

Hasta este momento aprendimos que las funciones en javascript y en casi cualquier lenguaje de programación son bloques de código los cuales tienen un nombre y pueden ser llamadas reiteradas veces a lo largo de nuestro script. Estos bloques de código son un conjunto de instrucciones que se ejecutan una detrás de la otra hasta que no haya más instrucciones o hasta que encontremos un "return" el cual devuelve un valor y detiene la ejecución del resto del código.
Las funciones generadoras son denominadas generalmente "pausables". Son funciones que se ejecutan a demanda y que se detienen cada vez que encuentran la palabra yield. A diferencia de las funciones normales, cuando guardamos una función generadora en una variable, guardamos el "Objeto generator" que es un objeto especial que cumple con el concepto de Iterator.

Declaración

Las funciones generadoras se declaran de la misma forma que declaramos una función normal pero agregando un * al lado de la palabra function:

[code]

```
function* generador() {
  //código a ser ejecutado
}
```

[/code]

Variantes de declaración

Existe muchas otras maneras de declarar una función generadora, aunque la anunciada anteriormente es la más común y utilizada. Otras de las formas que existen son:
[code]

```
function * generator () {}
function* generator () {}
function *generator () {}

let generator = function * () {}
let generator = function* () {}
let generator = function *() {}

class MyClass {
  *generator() {}
  * generator() {}
}
```

```
const obj = {
  *generator() {}
  * generator() {}
}
```

[/code]

Yield

Cuando definimos lo que eran las funciones generadoras, dijimos que las mismas se ejecutan parando o pausando cada vez que encuentran la palabra "yield".
Yield funciona, salvando las distancias, como un return pero que no termina la ejecución de la función sino que la detiene hasta la próxima vez que se pida que avance.

[code]

```
//Función con return
function impares(){
  var impar = 1;
  return impar;
  impar += 2;
  return impar;
}
```

[code]

En este ejemplo la función impares solo correrá hasta el primer return. Cuando la llamemos solo obtendremos el valor 1 y no podremos obtener nada más. Ahora vamos a ver como quedaría con yield.

[code]

```
function* impares(){
  var impar = 1;
  yield impar;
  impar += 2;
  yield impar;
}
```

[/code]

Ahora la función impares es generadora. Cuando vayamos necesitando sus valores la función sabrá cuándo detenerse y cuándo devolver información.

Utilizando una función generadora

Cuando llamamos a una función generadora para invocar, la misma devuelve un objeto Generador. Lo normal es guardar este objeto generador en una variable para luego utilizar el método next() para ir avanzando entre sus valores.

[code]

```
function* impares(){
  var impar = 1;
  yield impar;
  impar += 2;
  yield impar;
}

const generadorImpares = impares();

console.log(generadorImpares.next());
console.log(generadorImpares.next());
console.log(generadorImpares.next());


[/code]
```

Cada vez que llamemos a impares() estamos generando una nueva instancia del generador, por eso tenemos que guardar en una variable esa instancia para poder trabajarla.
El método next() lo que hará será iniciar o continuar la ejecución de la función. Devuelve un objeto con "value" y "done". Value es el valor que yield tiene, y done es un flag para que sepamos si la función terminó su ejecución o no.

En el caso del código que hicimos recién, las respuestas que obtendremos son:

```
Object {
  done: false,
  value: 1
}
Object {
  done: false,
  value: 3
}
Object {
  done: true,
  value: undefined
}
```

Vamos a entender qué está pasando. Cuando llamamos por primera vez a next, la función inicia su ejecución y devuelvo el primer valor de impar, es decir 1. Como todavía hay más instrucciones, el valor de done es false.
Cuando llamamos a next por segunda vez, la función resume su ejecución desde donde había quedado, entonces, el yield termina de ejecutarse (más adelante veremos qué significa esto), suma 2 a la variable impar y llega hasta el segundo yield que nos devuelve el valor actual de impar que es 3. Entonces value va a ser 3 y done va a ser false, porque si bien no quedan mas lineas de código por ejecutar, todavia no termine de ejecutarse la línea de yield propiamente dicha.
La tercera vez que ejecutemos next() la función retoma desde este punto, finaliza la línea de yield y como no hay más instrucciones termina su ejecución. Devuelve por lo tanto un value undefined y un done true.

Return

Podemos utilizar return para mejorar un poco el comportamiento de nuestra función:

[code]
function* impares(){
  var impar = 1;
  yield impar;
  impar += 2;
  return impar;
}
[/code]

Si hacemos esto, ahora la segunda vez que llamemos a next() el valor de done será true, ya que se ejecuta un return que por defecto finaliza la ejecución de la función.

Pasando variables al generador

Ya vimos que el método next devuelve el próximo valor que la función me quiere devolver y un flag de si existen más líneas de código para ejecutar o no. Nosotros también podemos pasarle valores a esa función generadora como parámetros del método next(). Ejemplo:

[code]

const personajes = [
  'Ariel',
  'Fernando',
  'Jose'
];

function* saludarPersonaje(){
  while(true) {
   const id = Math.floor(Math.random() * 3) + 0;
   const personaje = yield id;
   yield console.log('Hola ' + personaje);
  }
}

const generadorDeSaludos = saludarPersonaje();

const {value} = generadorDeSaludos.next();
generadorDeSaludos.next(personajes[value]);

[/code]

Analicemos el ejemplo anterior. Primero creo una lista de personajes, imaginemos que en realidad esta información puede venir desde un servidor.
Después declaro mi función generadora. Ponemos la generacion de informacion dentro de un ciclo infinito, esto es una practica muy comun.
Dentro del ciclo primero creo un ID al azar y hago yield de ese id. Cuando se llame al método next() por primera vez, va a devolver el valor de ese id generado.
Luego de esto, asignado el valor de yield id a const personaje. Es decir, esperamos recibir información desde el exterior de la función. Esto lo vemos en la última línea del script cuando llamamos al método next pasándole como argumento el personaje en cuestión finalmente realizar un console.log saludando al personaje.

Si quisiéramos saludar varias veces, bastaría con repetir estas dos líneas:

[code]
const {value} = generadorDeSaludos.next();
generadorDeSaludos.next(personajes[value]);

[/code]

Quizás dentro de un ciclo (no infinito).

Redux-saga

Introducción

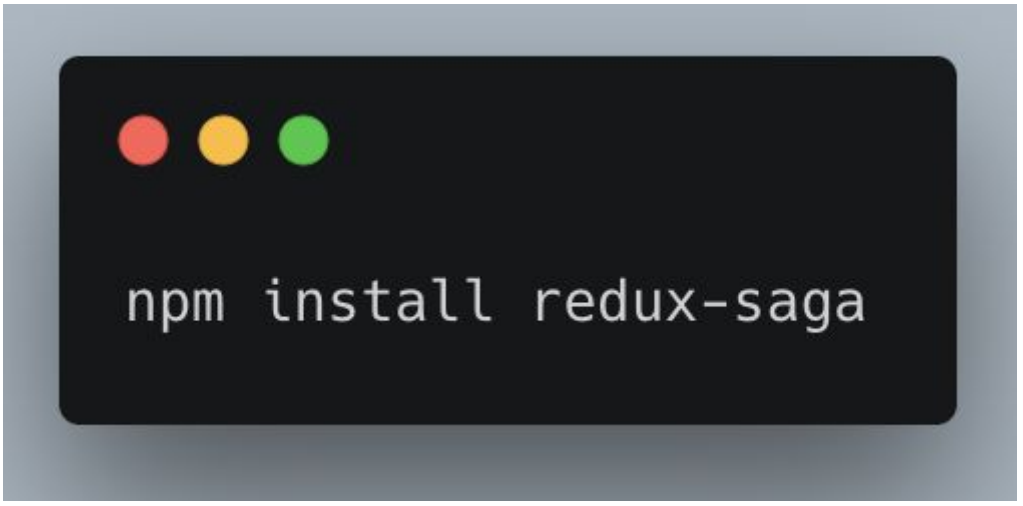
Redux-saga es una librería que extiende la funcionalidad de redux. Al igual que redux-thunk tiene como objetivo manejar efectos secundarios o paralelos de las acciones de nuestra aplicación de forma sencilla, mantenible y testeable.
El concepto principal que hay que entender cuando empezamos con redux-saga es que lo veamos como un hilo separado de tu aplicación que esté todo el tiempo escuchando las acciones que ocurren
Utiliza como herramienta principal los generadores de Javascript, motivo por el cual la primera parte de la clase aprendimos sobre ellos.

Diferencia con redux-thunk

Redux-thunk es una librería de Redux que nos permite que nuestras acciones, las que despachamos, puedan ser funciones y no solamente objetos JSON planos. Provee una gran libertad para realizar cualquier tipo de instrucción previo al envío de la acción al store.
Redux-saga en cambio sigue trabajando con acciones planas, con objetos JSON. Tiene una filosofía de trabajo totalmente distinta porque en este caso captura las acciones y ejecuta otro código despachando nuevas acciones.

Instalación

Para agregar redux-saga a nuestro proyecto utilizaremos el manejador de paquetes NPM.



Obviamente, necesitamos tener a la librería redux instalada también ya que depende de la misma.

Implementación

Al igual que redux-thunk, redux-saga es un middleware de redux. Entonces, en el archivo donde nosotros queramos utilizar sagas, tenemos que importar las funciones pertinentes:

[code]
import { createStore, applyMiddleware } from 'redux'
import createSagaMiddleware from 'redux-saga'

const sagaMiddleware = createSagaMiddleware()
const store = createStore(
  reducer,
  applyMiddleware(sagaMiddleware)
)
sagaMiddleware.run(/* Aca va el saga o el rootSaga */)

[/code]

Promesas en redux-sagas

Cuando trabajamos con redux-sagas, al igual que vimos anteriormente, las promesas que sean usadas con yield para la ejecución de la función hasta que las mismas sean resueltas. Esto es una gran ventaja a la hora de trabajar con código asincrónico.

[code]

function* worker() {
  yield axios.get(URL);
  console.log("Éxito");
}

[/code]

En la función de arriba, la línea de console.log no se ejecutará hasta que axios devuelva la promesa resuelta. Recién en ese caso la ejecución de la función continuará

Efectos

Los efectos en redux-saga es un concepto muy importante. Representan las operaciones provistas por la librería que podemos ejecutar y esperar con tranquilidad y certeza que el código continuará su ejecución cuando las mismas finalicen.
En general reciben parámetros de configuración indicandoles como funcionan.
Veremos las mismas más en acción más adelante.
En términos simples, los efectos son objetos planos JSON de javascript que representan algún tipo de operación dentro del middleware.

Watcher Sagas

Un watcher saga es una función generadora que se encarga de observar continuamente las acciones que están intentando llegar al reducer del store. En el caso de que esas acciones

correspondan a las que nos interesa interceptar, los watchers serán los encargados de realizar los pasos pertinentes y enviarle trabajo a los Worker Sagas

Efectos de los Watchers

Para lograr escuchar continuamente las acciones y ejecutar a los workers, los watchers tienen distintos métodos:

1. takeEvery: Por cada accion especificada que recibamos, ejecuta el worker correspondiente
2. takeLatest: Toma solamente la última llamada de la acción. Se encarga de que no se ejecute más de una vez en simultáneo un worker.

Worker Sagas

Un Worker saga es una función generadora la cual recibe un trabajo desde un watcher y realizar los efectos secundarios que necesitemos en nuestra aplicaciones, como animaciones o llamadas asíncronas.

Efectos de los Workers

Al igual que los watchers, los workers tienen sus propios métodos para poder trabajar las acciones:

1. put: Sirve para poder despachar nuevas acciones al store
2. call: Recibe una función y sus parámetros y la llama. Sirve para evitar usar funciones asíncronas directamente en los generadores. Ayuda a que el testing sea más sencillo. (Puede ser usado dentro de un try...catch para el manejo de errores)

Root Saga

Cuando trabajamos en una aplicación real lo más normal es que tengamos más de un saga funcionando. En general cada saga se ocupa solamente de observar y trabajar sobre una acción en particular.
Para lograr trabajar con múltiples sagas, redux-saga nos provee de un efecto llamado all. all recibe un array con todos los watchers que declaramos y se encarga de crear el objeto que nos servirá como nuestro rootSaga para pasarle al método run de nuestro middleware.

[code]

```
export default function* rootSaga() {
  yield all([
    saga1(),
    saga2()
  ])
}
```

[/code]