educación IT
CODE YOUR FUTURE

ReactJS Developer

Módulo 8

Redux-Saga

educación IT
CODE YOUR FUTURE

Redux-Saga

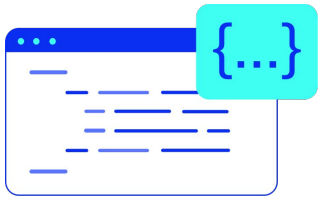
Redux-Saga es una **librería** que extiende la funcionalidad de Redux. Del mismo modo que *Redux-Thunk*, tiene como objetivo **manejar efectos secundarios o paralelos de las acciones de nuestra aplicación de forma sencilla, mantenible y testeable**.

El concepto principal a entender cuando nos iniciamos con *Redux-Saga* es verlo como **un hilo separado de tu aplicación que está escuchando todo el tiempo las acciones que ocurren**. Utiliza como herramienta principal **los generadores de JavaScript**, por eso, la primera parte del módulo 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 previa al envío de la acción al *store*.

En cambio, *Redux-Saga* **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**.

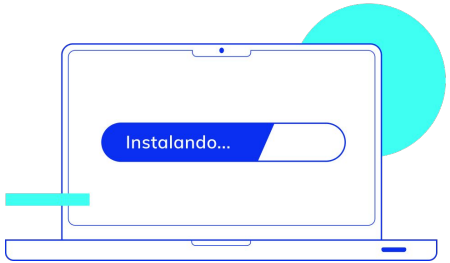


Instalación

Para agregar *Redux-Saga* a nuestro proyecto, utilizaremos **el manejador de paquetes NPM**:

```
npm install redux-saga.
```

Es importante recordar que **necesitamos tener a la librería Redux instalada**, ya que depende de la misma.



Implementación

Al igual que *Redux-Thunk*, *Redux-Saga* **es un *middleware* de Redux**. Entonces, en el archivo donde queramos utilizar sagas, tenemos que **importar las funciones pertinentes**:

```
1   import {createStore, applyMiddleware} from 'redux';
2   import createSagaMiddleware from 'redux-saga';
3
4   const sagaMiddleware = createSagaMiddleware();
5
6   const store = createStore(
7       rootReducer,
8       applyMiddleware(sagaMiddleware) // Incluímos reduxThunk acá
9   );
10
11  sagaMiddleware.run(/* rootSaga */);
12
13  export default store;
```

Promesas en *Redux-Saga*

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

En la siguiente función, 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.

```
1  function *worker() {
2      yield axios.get('/posts');
3      console.log("Exito");
4  }
```

Efectos en Saga

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 de que el código continuará su ejecución cuando las mismas finalicen**. En general reciben parámetros de configuración que les indican cómo funcionar. Veremos las mismas 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 Saga

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 Saga***.

Efectos en los *Watchers*

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

1. **`takeEvery`**: por cada acción especificada que recibamos, **ejecuta al *worker* correspondiente**.
2. **`takeLatest`**: toma solamente **la última llamada de la acción**. Se encarga de que un *worker* **no se ejecute más de una vez en simultáneo**.

Worker Saga

Un ***Worker Saga*** es una función generadora que **recibe un trabajo desde un *watcher* y realiza 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 **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`**. Este 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***:

```
export default function *rootSaga() {
    yield all([
        saga1(),
        saga2(),
        saga3()
    ]);
}
```

educación IT
CODE YOUR FUTURE

¡Sigamos trabajando!