Sergio Xalambrí

# Desarrollo de Aplicaciones Web con React.js y Redux.js

sdx.im

# Desarrollo de Aplicaciones Web con React.js y Redux.js

Sergio Daniel Xalambrí

Este libro está a la venta en http://leanpub.com/react-redux

Esta versión se publicó en 2018-06-20

# ¡Twitea sobre el libro!

Por favor ayuda a Sergio Daniel Xalambrí hablando sobre el libro en Twitter!

El tweet sugerido para este libro es:

Acabo de comprar "Desarrollo de Aplicaciones Web con React.js y Redux.js" por @sergiodxa

El hashtag sugerido para este libro es #react-redux.

Descubre lo que otra gente está diciendo sobre el libro haciendo click en este enlace para buscar el hashtag en Twitter:

#react-redux

# Índice general

**Introducción a Redux.js** . . . . . **1**
Principios . . . . . 1
Instalando Redux en nuestro proyecto . . . . . 2
Conceptos básicos . . . . . 2
Conclusión . . . . . 6

**Combinando React.js y Redux.js** . . . . . **7**
Instalando react-redux . . . . . 7
Encapsulando la aplicación . . . . . 7
Accediendo al Store . . . . . 8
Optimizando . . . . . 8
Despachando acciones . . . . . 9
Funcionamiento sin decoradores . . . . . 10
Conectando componentes puros . . . . . 11
Conclusión . . . . . 11

**Middlewares en Redux.js** . . . . . **12**
API . . . . . 12
Middleware de logging . . . . . 13
Middleware de errores . . . . . 14
Usando un middleware . . . . . 14
Conclusión . . . . . 14

**Acciones asíncronas en Redux.js** . . . . . **15**
Manualmente . . . . . 15
Usando middlewares . . . . . 16
Con redux-thunk y redux-promise . . . . . 16
Otras opciones . . . . . 17
Conclusión . . . . . 17

**Pruebas unitarias en Redux.js** . . . . . **19**
Preparando el ambiente de pruebas . . . . . 19
Creadores de acciones . . . . . 19
Reducers . . . . . 20

Middlewares . . . 21
Conclusiones . . . 23

**Estructura de archivos Ducks para Redux.js . . . 24**
Que es . . . 24
Como funciona . . . 24
Reglas . . . 25
Como usarlo . . . 25
Conclusión . . . 25

**Creando código modular con ducks de Redux . . . 26**
Instalando dependencias . . . 26
Creando nuestro duck . . . 26
Definiendo tipos de acciones . . . 26
Creando nuestros creadores de acciones . . . 27
Creando nuestra función reductora . . . 27
Código final . . . 28
Conclusión . . . 29

**Manejo de errores en Redux.js . . . 30**
Creando nuestro middleware . . . 30
Usando uno ya hecho . . . 31
Conclusión . . . 31

**Usando Redux en el servidor con Socket.io . . . 32**
Instalación de dependencias . . . 32
Creando nuestro Store y Reducers . . . 32
Servidor de WebSockets . . . 34
Cliente web . . . 34
Conclusión . . . 35

**Renderizando aplicaciones de Redux en el servidor . . . 37**
Instalando dependencias . . . 37
Preparando el servidor . . . 37
Renderizando React.js . . . 38
Implementando Redux . . . 39
Renderizado con props . . . 40
Conclusión . . . 41

**Obteniendo datos en aplicaciones de Redux . . . 42**
Definiendo el API . . . 42
Creando un cliente para el API . . . 42
Middleware para acciones asíncronas . . . 43
Implementando el Middleware . . . 45
Conclusión . . . 45

**Estado inmutable con Redux e Immutable.js** . . . . . **46**
Usándolo en un reducer . . . . . 46
Combinando reducers . . . . . 46
Combinando reducers con Immutable.js . . . . . 47
Conclusión . . . . . 47

**Componentes de Alto Orden en React.js** . . . . . **48**
Conclusión . . . . . 51

**Migrando a Redux** . . . . . **52**
Desde Flux . . . . . 52
Desde Backbone . . . . . 53

**Glosario de términos** . . . . . **54**
Estado . . . . . 54
Acción . . . . . 54
Reducer . . . . . 54
Función despachadora . . . . . 55
Creador de acciones . . . . . 55
Acción asíncrona . . . . . 56
Middleware . . . . . 56
Store . . . . . 56
Creador de store . . . . . 57
Potenciador de store . . . . . 57

# Introducción a Redux.js

Redux es una librería para controlar el estado de nuestras aplicaciones web fácilmente, de una forma consistente entre cliente y servidor, testeable y con una gran experiencia de desarrollo.

Redux está en gran parte influenciado por la arquitectura Flux[1] propuesta por Facebook para las aplicaciones de React.js[2] y por el lenguaje Elm[3], esta muy pensado para React.js, pero también se puede usar con Angular.js, Backbone.js o simplemente con Vanilla JS.

## Principios

Redux se basa en tres principios:

### Una sola fuente de la verdad

> Todo el estado de tu aplicación esta contenido en un único store

Esto facilita depurar nuestra aplicación y crear aplicaciones universales cuyo estado en el servidor pueda serializarse para luego usarlo en el navegador sin mucho esfuerzo. Otras funcionalidades como atras/adelante se hacen más fáciles de implementar cuando tenemos un solo store con todo el estado de nuestra aplicación.

### El estado es de solo lectura

> La única forma de modificar el estado es emitir una acción que indique que cambió

Esto te asegura que ninguna parte de la aplicación, como pueden ser eventos de la UI, callbacks o sockets, alteren directamente el estado de tu aplicación, en vez de eso emiten una intención de modificarlo.

Y gracias a que todas las modificaciones se centralizan y se ejecutan una por una es imposible que se pisen cambios. Por último como las acciones son objetos planos pueden mostrarse en consola o almacenarse para volverlas a ejecutar durante el debugging.

[1] https://facebook.github.io/flux/
[2] https://facebook.github.io/react/
[3] http://elm-lang.org/

### Los cambios se hacen mediante funciones puras

> Para controlar como el store es modificado por las acciones se usan reducers puros

Los Reducers son funciones puras que reciben el estado actual de la aplicación y la acción a realizar y devuelven un **nuevo** estado, sin modificar directamente el estado actual. Podemos tener un solo Reducer encargado de toda nuestra aplicación o si esta crece dividirlo en múltiples funciones las cuales podemos controlar en que orden se ejecutan.

## Instalando Redux en nuestro proyecto

Para instalar Redux es igual que cualquier librería/framework que esté en npm:

```
1  npm i -S redux
```

Luego podemos importarlo como cualquier otro módulo:

```
1  import redux from "redux";
```

Internamente va a descargar dos dependencias:

- loose-envify[4] - Inyecta variables de entorno
- lodash[5] - Colección de funciones utilitarias para programación funcional

## Conceptos básicos

Redux es bastante fácil de aprender, aunque a simple vista no lo parezca, incluso es tan fácil que la librería es increíblemente pequeña (2kb minificada).

### Acciones

Las Acciones son POJOs (Plain Old JavaScript Objects) con al menos una propiedad que indica el tipo de acción y, de ser necesario, otras propiedades indicando cualquier otro dato necesario para efectuar nuestra acción. Normalmente se usa el formato definido en el Flux Standard Action (FSA)[6].

[4] https://github.com/zertosh/loose-envify
[5] https://github.com/lodash/lodash
[6] https://github.com/acdlite/flux-standard-action

```
{
  "type": "ADD_TODO",
  "payload": {
    "text": "Aprender Redux"
  }
}
```

Para enviar una acción a nuestro Store usamos la función `store.dispatch()` pasando nuestra acción como único parámetro.

## Creadores de acciones

Estos son simplemente funciones que pueden o no recibir parámetros y devuelven una acción (un POJO), es muy buena idea, para evitar problemas de consistencia, programar una función por cada tipo de acción y usarlas en vez de armar nuestros objetos a mano.

```
/**
 * Devuelve una acción de tipo ADD_TODO
 * @param {String} text Texto del TODO
 * @return {Object} Objecto de acción
 */
function addTodo(text) {
  return {
    type: "ADD_TODO",
    payload: {
      text
    }
  };
}
```

Debido a que normalmente son funciones puras son fáciles de testear. Luego de ejecutar nuestra función, para poder despachar la acción, es simplemente llamar a la función `dispatch(addTodo('Aprender Redux'))`.

## Reducers

Mientras que las acciones describen que algo ocurrió no especifican como nuestra aplicación reacciona a ese algo. De esto se encargan los **Reducers**.

Ya que el estado de nuestra aplicación es un único objeto es buena idea empezar a pensar cual va a ser la forma más básica antes de empezar a programar, como ejemplo vamos a suponer que hacemos una aplicación de TODOs por lo que nuestro store va a tener el siguiente formato:

```
{
  "todos": []
}
```

Ahora que definimos la forma de nuestro store podemos empezar a crear reducers. Un reducer es una función pura que recibe el estado actual y una acción y devuelve el nuevo estado.

```
(prevState, action) => nextState;
```

Se llaman reducers porque son el tipo de funciones que pasarías a `Array.prototype.reduce(reducer[, initialValue])`. Es muy importante que se mantengan como funciones puras. Algunas cosas que **nunca** deberías hacer en un reducer:

- Modificar sus argumentos directamente (lo correcto es crear una copia)
- Realizar acciones con efectos secundarios como llamadas a una API o cambiar rutas
- Ejecutar funciones no puras como Date.now() o Math.random()

Que se mantengan puros quiere decir que pasandole los mismos parámetros debería siempre devolver el mismo resultado. Ahora vamos a programar un reducer para nuestra acción `ADD_TODO`:

```
// cargamos el método de Redux para
// poder combinar reducers
import { combineReducers } from "redux";

function todos(state = [], action = {}) {
  switch (action.type) {
    case "ADD_TODO":
      // creamos una copia del estado actual
      const copy = Array.from(state);
      // modificamos lo que necesitamos
      copy.push(action.payload.text);
      // retornamos el nuevo estado
      return copy;
    default:
      // si el action.type no existe o no concuerda
      // con ningunos de los casos definidos
      // devolvemos el estado sin modificar
      return state;
  }
}

// combinamos nuestros reducers
// los keys que usemos para nuestros reducers
// van a ser usados como keys en nuestro store
// en este ejemplo sería: { todos: [], }
const reducers = combineReducers({
  todos
});

export default reducers;
```

Como se ve arriba Redux nos provee una función llamada `combineReducers()` que recibe un objeto con los reducers que definimos y los combina.

El nombre que le pongamos a cada reducer es usado como propiedad del store que creemos y es donde se va a guardar el estado devuelto por el reducer.

## Store

Por último necesitamos crear nuestro **Store**, el store va a tener cuatro responsabilidades:

1. Almacenar el estado global de la aplicación
2. Dar acceso al estado mediante `store.getState()`
3. Permitir que el estado se actualice mediante `store.dispatch()`
4. Registrar listeners mediante `store.subscribe(listener)`

Es importante recordar que solo podemos tener un store en nuestras aplicaciones de Redux, cuando quieras separar la lógica de manipulación de datos usa la composición de reducers en vez de muchos stores.

Para crear un store necesitamos una función de Redux y el reducer (o los reducers combinados) que vamos a usar:

```
1  // cargamos la función para crear un store
2  import { createStore } from "redux";
3  // cargamos nuestros reducers (ya combinados)
4  import reducers from "./reducers.js";
5  // definimos el estado inicial
6  const initialState = {
7    todos: []
8  };
9  // creamos el store
10 const store = createStore(reducers, initialState);
11 export default store;
```

La función `createStore` simplemente recibe nuestros reducers como primer parámetro y nuestro estado inicial como segundo (y opcional), en el estado inicial podemos desde enviar simplemente la forma básica de nuestro store hasta enviar los datos recibidos desde el servidor.

## Obteniendo nuestro estado

Una vez tenemos nuestro store creado podemos acceder al estado que almacena con `store.getState()` desde cualquier parte de nuestra aplicación donde importemos el store.

```
import store from "./store.js";
// vemos el estado actual del store
console.log(store.getState());
```

## Subscribirse a los cambios de estado

Podemos suscribirnos al store para enterarnos cuando cambia y poder modificar nuestra aplicación en consecuencia usando `store.subscribe(callback)`.

```
import store from "./store.js";
// nos suscribimos al store, esto nos devuelve
// una función que nos sirve para desuscribirnos
const unsubscribe = store.subscribe(() => {
  // vemos el nuevo store
  console.log(store.getState());
  // nos desuscribimos
  unsubscribe();
});
```

# Conclusión

Como se puede ver Redux es bastante simple de empezar a usar, y gracias a que es tan simple es posible combinarlo con prácticamente cualquier framework o librería que exista, ya sea React[7], Backbone[8], Angular 1[9]/2[10], Ember[11], etc.

[7] https://medium.com/react-redux/combinando-react-js-y-redux-js-7b45a9dc39ac
[8] https://github.com/redbooth/backbone-redux
[9] https://github.com/wbuchwalter/ng-redux
[10] https://github.com/wbuchwalter/ng2-redux
[11] http://www.ember-redux.com/ddau/

# Combinando React.js y Redux.js

En el capítulo anterior vimos como funciona Redux.js[12] y dijimos que era posible usarlo con cualquier framework o librería de JavaScript.

Y, aunque esto es cierto, Redux es especialmente bueno al usarlo con librerías como React.js[13], ya que podés describir tu UI como funciones puras y usar Redux para tener todo el estado de nuestra aplicación y pasarlo a nuestras vistas.

## Instalando react-redux

La conexión de React con Redux no esta incluida directamente en Redux, para esto necesitamos bajar react-redux, así que vamos a descargar lo necesario:

```
npm i -S react react-dom react-redux redux
```

## Encapsulando la aplicación

Lo primero que necesitamos es encapsular nuestra aplicación con el componente `Provider` que trae `react-redux`. Este componente recibe un único parámetro llamado store el cual es, como su nombre indica, la instancia del Store que usamos.

```
import React from "react";
import { render } from "react-dom";
import { Provider } from "react-redux";

import store from "./store";
import App from "./components/App";

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("app")
);
```

Este componente `Provider` define en el contexto global de React nuestra instancia del store.

[12] https://sergiodxa.com/essays/introduccion-a-redux
[13] https://platzi.com/react/

## Accediendo al Store

Una vez encapsulada nuestra aplicación de React nos toca definir que componentes van a acceder a nuestro Store, ya que no todos lo van a necesitar.

Para hacer eso necesitamos conectar nuestros componentes a Redux, esto se logra con un decorador que trae `react-redux` llamado `connect`.

```
// importamos el decorador @connect de react-redux
import { connect } from "react-redux";
import React from "react";
import UserItem from "./UserItem";

// aplicamos el decorador @connect a nuestro componente
@connect()
class UserList extends React.Component {
  render() {
    // renderizamos el listado de usuarios que
    // recibimos como props del Store
    return (
      <section>
        {this.props.users.map(user => <UserItem {...user} key={user.id} />)}
      </section>
    );
  }
}
export default UserList;
```

De esta forma nuestro componente UserList va a tener dentro de sus `props` todos los datos del Store. Con esto ya podemos renderizar nuestra aplicación usando los datos almacenados en el Store de Redux.

## Optimizando

Aunque el método anterior sea más que suficiente no es lo mejor a nivel de performance, ya que de esta forma cada vez que cambie algo del Store se va a volver a renderizar `UserList`, incluso si la lista de usuario no cambio.

Para mejorar esto el decorador `connect` puede recibir una función que define que datos pasar al componente conectado.

```
function mapStateToProps(state, props) {
  // armamos un objeto solo con los
  // datos del store que nos interesan
  // y lo devolvemos
  return {
    users: state.users,
  };
}

// aplicamos el decorador @connect pasándole
// nuestra función mapStateToProps
@connect(mapStateToProps)
class UserList extends React.Component {
  ...
}
```

De esta forma podemos solo enviar a `UserList` el listado de usuarios, así cuando se modifique otra cosa que no sea la lista de usuarios no se va a volver a renderizar el componente.

## Despachando acciones

Entre las `props` que el decorador `connect` inyecta a nuestro componente se encuentra la función `dispatch` del Store, con la cual podemos despachar acciones.

```
// cargamos nuestro creador de acciones
import sendData from '../actions/send-data';

@connect()
class UserList extends React.Component {
  handleSendData() {
    const action = sendData();
    // despachamos la acción al store
    this.props.dispatch(action);
  }
  ...
}
```

Resulta que `connect` como segundo argumento podemos pasarle una función que nos permite controlar la función `dispatch` para mandar una personalizada.

```
import sendData from '../actions/send-data';
// importamos el método bindActionCreators de Redux
import { bindActionCreators } from 'redux';

function mapStateToProps(state, props) { ... }

function mapDispatchToProps(dispatch, props) {
  // creamos un objeto con un método para crear
  // y despachar acciones fácilmente y en
  // una sola línea
  const actions = {
    sendData: bindActionCreators(sendData, dispatch),
  };

  // devolvemos nuestras funciones dispatch
  // y los props normales del componente
  return { actions };
}

// decoramos nuestro componente pasándole las
// funciones mapStateToProps y mapDispatchToProps
@connect(mapStateToProps, mapDispatchToProps)
class UserList extends React.Component {
  handleSendData() {
    // creamos y despachamos la acción sendData
    this.props.actions.sendData();
  }
  ...
}
```

De esta forma podemos mandar a nuestro componente las acciones que necesitamos y que con solo ejecutarlas ya haga el dispatch de estas.

# Funcionamiento sin decoradores

Aunque `connect` esta pensado como decorador es posible usarlo como una función normal sin necesidad de usar Babel con el plugin babel-plugin-transform-decorators-legacy[14] para soportar decoradores de la siguiente forma.

```
export default connect(mapStateToProps, mapDispatchToProps)(Component);
```

Como siempre `connect` recibe `mapStateToProps` y `mapDispatchToProps` como parámetros, solo que además devuelve una función que recibe el componente a conectar y nos devuelve el componente conectado, el cual simplemente exportamos y listo, conseguimos el mismo resultado que usándolo como un decorador.

Esto es util si no queremos usar decoradores todavía, ya que la actual propuesta es muy posible que cambie a futuro.

[14] https://github.com/loganfsmyth/babel-plugin-transform-decorators-legacy

## Conectando componentes puros

Aunque lo normal es usar `connect` con componentes hechos con clases es completamente posible usarlo con componentes puros si hacemos uso del decorador como una función.

```
import React from 'react';
import { connect } from 'react-redux';
function mapStateToProps(state, props) { ... }
// el componente puro a conectar
function UserItem(props) { ... }
// exportamos el componente conectándolo gracias a connect
export default connect(mapStateToProps)(UserItem);
```

Así podríamos crear toda nuestra aplicación solamente con componentes puros, sin necesidad de usar clases. Esto nos obligaría, de verdad, a mantener el estado de toda nuestra aplicación en Redux y dejar React para la UI.

## Conclusión

Integrar React y Redux es bastante simple y gracias a `connect` es muy fácil controlar qué datos del Store le llegan a cada componente permitiendo una mejor performance y evitando renders innecesarios.

# Middlewares en Redux.js

Luego de ver como funciona Redux.js[15] y como usarlo con React.js[16], vamos a aprender a crear middlewares, estos son funciones de orden superior que nos permiten agregar funcionalidad a los Stores de Redux.

Un middleware sirve para muchas tareas diferentes, registrar datos, capturar errores, despachar promesas, cambiar rutas, etc. básicamente cualquier tarea con efectos secundarios se puede hacer en un middleware.

## API

Como dije un middleware es una función, esta función va a recibir como parámetro una versión reducida del Store, con solamente los métodos `dispatch` y `getState`.

Esta función debe devolver otra función que va a recibir una función normalmente llamada `next` que nos sirve para llamar al siguiente middleware.

Por último devolvemos una nueva función que va a recibir la acción que se esta ejecutando. Para entenderlo bien veámoslo en código.

```
1  // nuestro middleware
2  function middleware(store) {
3    // recibimos { dispatch, getState }
4    return function wrapDispatch(next) {
5      // recibimos next
6      return function dispatchAndDoSomething(action) {
7        // recibimos la acción despachada
8        // acá va nuestro código
9      };
10   };
11 }
```

Esta es básicamente la estructura de un middleware, dentro de nuestra función `dispatchAndDoSomething` es donde vamos a colocar todo nuestro código. Otra forma escribir este código es usando arrow functions para que quede más simple y conciso.

[15] https://medium.com/@sergiodxa/introducci%C3%B3n-a-redux-js-8bdf4fe0751e#.39z09rgjd
[16] https://medium.com/@sergiodxa/combinando-react-js-y-redux-js-7b45a9dc39ac#---25-92.srpbnzc1s

```
// esto:
const middleware = store => next => action => {
  // acá va nuestro código
};
// es lo mismo que esto:
const middleware = function(store) {
  return function(next) {
    return function(action) {
      // acá va nuestro código
    };
  };
};
```

De esta forma nos quedan todos los argumentos en una línea y simplemente escribimos el código necesario.

## Middleware de logging

Supongamos que queremos crear un middleware que muestre en consola cada acción despachada, el cambio en el Store y cuanto tarda en ejecutase.

```
const logger = store => next => action => {
  // agrupamos lo que vamos a mostrar en
  // consola usando el tipo de la acción
  console.group(action.type);
  // mostramos el estado actual del store
  console.debug("current state", store.getState());

  // mostramos la acción despachada
  console.debug("action", action);

  // empezamos a contar cuanto se tarda en
  // aplicar la acción
  console.time("duration");

  // pasamos la acción al store
  next(action);

  // terminamos de contar
  console.timeEnd("duration");

  // mostramos el estado nuevo
  console.debug("new state", store.getState());
  // terminamos el grupo
  console.groupEnd();
};
```

Este simple middleware registra en consola el estado del Store, la acción despachada, el nuevo estado y cuanto se tarda en realizar los cambios (perfecto para identificar problemas de performance).

## Middleware de errores

Creemos otro middleware de ejemplo, hagamos uno que en caso de un error nos muestre que ocurrió.

```
const catchError = store => next => action => {
  try {
    // aplicamos la acción
    next(action);
  } catch (error) {
    // mandamos nuestro error a algún servicio
    // como Sentry o Track:js donde luego
    // podamos revisarlo con detalle
    errorLogger.send(error);
  }
};
```

Este middleware super simple nos permitiría capturar cualquier error que ocurra y registrarlo ya sea en consola o en algún servicio.

## Usando un middleware

Luego de creado nuestro middleware para empezar a usarlo tenemos que aplicarlo al Store al momento de crearlo.

```
import { applyMiddleware, createStore } from "redux";
import reducer from "./reducer";
import logger from "./middlewares/logger";
import catchError from "./middlewares/catch-error";
const store = createStore(reducer, applyMiddleware(catchError, logger));
```

De esta forma aplicamos nuestros dos middlewares al momento de crear el Store. Cabe destacar que el orden en que se pasen los middlewares importa en como se van a ejecutar, en nuestro caso primero se ejecuta catchError y luego logger de forma que si tanto logger como los reducers tienen algún error catchError lo va a registrar y si no logger va a mostrar la información en la consola.

## Conclusión

Usar middlewares nos permite extender fácilmente la funcionalidad de nuestros Stores de Redux.js, sin escribir mucho código, lo que puede resultarnos muy útil en aplicaciones medianas o grandes para implementar fácilmente features que necesitemos.

# Acciones asíncronas en Redux.js

Redux, por su naturaleza puramente funcional, esta pensado para realizar tareas síncronas:

```
(state, action) => newState;
```

Sin embargo, debido a como funciona JS lo más común es trabajar de forma asíncrona, por ejemplo hacer una petición AJAX a una API es asíncrono y luego de esto seguramente vamos a querer modificar el Store en base a la respuesta.

Para esto se usan las **acciones asíncronas**, hay varias formas de trabajar con acciones asíncronas en Redux, una es hacerlo a mano, otra opción es usar es usar distintos middlewares.

## Manualmente

Para hacerlo manualmente primero necesitamos nuestro creador de acciones, por ejemplo.

```
export default function addTodo(content) {
  return {
    type: "ADD_TODO",
    payload: { content }
  };
}
```

Luego es bastante simple, al terminar de ejecutar nuestra función asíncrona vamos a despachar nuestra acción normalmente.

```
import store from "./store";
import addTodo from "./actions/add-todo";

fetch("/api/todos/1")
  .then(response => response.json()) // obtenemos los datos
  .then(addTodo) // creamos la acción
  .then(store.dispatch) // despachamos la acción
  .catch(error => {
    // si hay algún error lo mostramos en consola
    console.error(error);
  });
```

Con esto al terminar nuestra petición creamos la acción en base a la respuesta y luego la despachamos, bastante simple. Esto podría ocurrir como resultado de que el usuario envíe un formulario o haga click sobre un botón.

# Usando middlewares

Si no queremos que este proceso sea manual para cada función asíncrona, podemos usar middlewares que se encarguen de esto automáticamente, para esto hay varias opciones.

# Con redux-thunk y redux-promise

Estos dos middlewares nos permiten hacer un dispatch de una función que devuelva una promesa y que se haga el dispatch de la acción automáticamente.

Primero vamos a bajarlos con **npm**.

```
npm i -S redux-thunk redux-promise
```

- redux-thunk[17]: permite despachar funciones que devuelvan promesas y el Store se encarga de ejecutarlos
- redux-promise[18]: permite despachar promesas y el Store se encarga de esperar que se completen, las promesas deben devolver una acción

Para que estos funcionen primero necesitamos crear nuestro Store aplicándole los middlewares.

```
import { createStore, applyMiddleware } from "redux";
import promise from "redux-promise";
import thunk from "redux-thunk";

import reducers from "./reducers";

export default createStore(reducers, applyMiddleware(thunk, promise));
```

De esta forma creamos nuestro store con ***redux-thunk*** y ***redux-promise*** como middlewares. Desde ahora además de despachar objetos (para acciones síncronas), podemos despachar funciones que devuelvan promesas para acciones asíncronas.

[17] https://github.com/gaearon/redux-thunk
[18] https://github.com/acdlite/redux-promise

```
import store from "./store";
import addTodo from "./actions/add-todo";
// creamos un pequeño cliente para un api
const api = {
  todos: {
    retrieve() {
      // hacemos el request y nos encargamos de convertir
      // la respuesta en una acción válida
      return fetch("/api/todo")
        .then(response => response.json()) // obtenemos los datos
        .then(addTodo); // los convertimos en una acción
    }
  }
};

store.dispatch(api.todos.retrieve());
```

Gracias a los middlewares que aplicamos Redux va a esperar que la función `api.todos.post` se termine de ejecutar y que el resultado sea una acción estándar de Flux[19] (con `type` y `payload`) y cuando se complete va a, efectivamente, despachar nuestra acción igual que si hubiésemos hecho todo a mano.

## Otras opciones

Además de usar **redux-thunk** y **redux-promise** hay otros middlewares que nos pueden servir al momento de trabajar con acciones asíncronas.

- redux-simple-promise[20]: Para trabajar con promesas, similar a **redux-promise**.
- redux-async[21]: Para trabajar con promesas, similar a **redux-promise**.
- redux-future[22]: Para trabajar con promesas (mónadas), similar a **redux-promise**.
- redux-rx[23]: Para trabajar con Observables de RxJS[24].
- redux-saga[25]: Para trabajar con generadores de ES6.

Todos estos son buenas opciones, **redux-saga** en particular esta tomando mucha fuerza, y es apoyado por el mismo creador de Redux[26].

## Conclusión

Como ven, trabajar con acciones asíncronas es en realidad muy fácil, incluso haciéndolo a mano. La gran ventaja de usar middlewares que nos solucionen esto es que si en muchas partes vamos a

[19] https://github.com/acdlite/flux-standard-action
[20] https://github.com/alanrubin/redux-simple-promise
[21] https://github.com/symbiont-io/redux-async
[22] https://github.com/stoeffel/redux-future
[23] https://github.com/acdlite/redux-rx
[24] https://github.com/ReactiveX/rxjs
[25] https://github.com/yelouafi/redux-saga
[26] https://github.com/gaearon

usar acciones asíncronas (y es muy probable que pase) nos ahorramos mucho trabajo cada vez que despachamos la acción.

# Pruebas unitarias en Redux.js

Cuando desarrollamos una aplicación con Redux.js la mayor parte del código que escribas van a ser funciones puras, esto hace que crear pruebas unitarias para nuestra aplicación sea más fácil que nunca.

## Preparando el ambiente de pruebas

Lo primero que necesitamos para empezar a hacer pruebas es configurar nuestro ambiente de desarrollo local para correr las pruebas. Para esto vamos a usar las librerías tape y tap-spec.

```
npm i -D tape tap-spec
```

- tape: nos sirve para realizar nuestras pruebas
- tap-spec: formatea y colorea el resultado de las pruebas en consola.

Para usarlos vamos a crear una carpeta llamada `test` y dentro un `index.js` que es nuestro entry point para pruebas. Luego vamos a crear un script en `package.json` llamado `test`.

```
{
  ...
  "scripts": {
    "test": "tape -r babel-core/register test/index.js | tap-spec"
  }
  ...
}
```

Como se ve vamos a estar usando Babel.js[27] para transpilar el código de nuestras pruebas. Cuando vayamos a correr nuestras pruebas simplemente usamos el comando de npm.

```
npm test
```

Y con esto ya ejecutamos las pruebas.

## Creadores de acciones

Un creador de acciones es simplemente una función pura que recibe ciertos parámetros y devuelve un objeto que describe una acción. Por ejemplo.

[27] https://babeljs.io/

```
export default function addTodo(message) {
  return {
    type: "ADD_TODO",
    payload: {
      message
    }
  };
}
```

Ese es un ejemplo de un creador de acciones común. Vamos a crearle una prueba.

```
import test from "tape";
import addTodo from "../actions/add-todo";
test("ADD_TODO action creator", t => {
  t.plan(1);
  t.deepEquals(
    addTodo("hello world"),
    {
      type: "ADD_TODO",
      payload: {
        message: "hello world"
      }
    },
    "it should return the expected object"
  );
});
```

Con esto ya tenemos una prueba para verificar que nuestro creador de acciones funciona correctamente. En general todos los creadores de acciones van a funcionar de esta forma así que probarlos es bastante sencillo.

## Reducers

Los Reducers son la parte más importante de cada aplicación de Redux, y deberían todos tener pruebas unitarias para asegurarse su correcto funcionamiento. Un ejemplo simple de un reducer puede ser el siguiente.

```
// definimos el estado inicial
const initialState = [];

function todos(state = initialState, action = {}) {
  switch (action.type) {
    // si la acción es ADD_TODO
    case "ADD_TODO":
      // copiamos el estado actual
      const newState = Array.from(state);
      // agregamos el mensaje nuevo
      newState.push(action.payload.message);
      // devolvemos el nuevo estado
      return newState;
    // si no identificamos la acción
    default:
      // devolvemos el estado sin tocarlo
      return state;
  }
}

export default todos;
```

Para poder hacer pruebas sobre un reducer necesitamos simplemente ejecutarlo pasándole un estado y una acción y ver el resultado que devuelve.

```
import test from "tape";
import addTodo from "../actions/add-todo";
import todos from "../reducers/todos";

test("todos reducer", t => {
  t.plan(2);
  const defaultState = todos();
  t.deepEquals(defaultState, [], "it should return an empty array as default");
  const message = "Hello world";
  const state = todos(defaultState, addTodo(message));
  t.deepEquals(state, [message], "it should return an array with the message");
});
```

Con esto ya probamos que ocurre cuando no recibe una acción y que ocurre cuando recibe una acción de tipo `ADD_TODO`. Gracias a estas dos simples pruebas podemos asegurarnos de que nuestro reducer funciones correctamente.

# Middlewares

Acá ya se puede volver más complicado de probar, principalmente porque depende del middleware que estemos probando el como vamos a escribir nuestras pruebas. Para este ejemplo vamos a usar el middleware socket.io-redux[28]. Primero veamos el código del middleware.

[28] https://github.com/sergiodxa/socket.io-redux

```
const socketIO = socket => () => next => action => {
  if (action.meta && action.meta.socket && action.meta.socket.channel) {
    socket.emit(action.meta.socket.channel, action);
  }
  return next(action);
};
export default socketIO;
```

Ese es el código de nuestro middleware, básicamente recibe una acción y valida que tenga la propiedad `meta`, que esta sea un objeto con una propiedad `socket` que a su vez sea otro objeto con la propiedad `channel`. Si posee todo esto entonces emite la acción a través del canal especificado en la metadata de la acción. Por último pasa la acción al siguiente middleware o al reducer.

Ahora veamos como hacer una prueba de este middleware.

```
import test from 'tape';
import socketIO from 'socket.io-redux';
// simulamos la función next que va a usar el middleware
function next(action) { return action; }
test('socket.io middleware', t => {
  t.plan(3);
  // nuestra acción de prueba con los datos necesarios
  const testAction = {
    type: 'ADD_NEW',
    payload: 'hello world!',
    meta: { socket: { channel: 'add:new' } },
  };
  // simulamos el objeto socket con el método emit
  const socket = {
    emit(channel, data) {
      // nuestro falso método emit
      // acá hacemos las pruebas
      t.equals(
        channel,
        'add:new',
        'it should have the channel "add:new"'
      );
      t.deepEquals(
        data,
        testAction,
        'it should have the action as data'
      );
    },
  };
  // ejecutamos el middleware y guardamos el resultado
  const action = socketIO(socket)()(next)(testAction);
  t.deepEquals(
    action,
    testAction,
    'it should return the passed action'
```

```
  );
});
```

## Conclusiones

Hacer pruebas unitarias de nuestro código es super importante para evitarnos problemas y encontrar errores más rápido.

En el caso de aplicaciones de Redux ya que la mayor parte de nuestro código no depende directamente de Redux para funcionar es muy simple hacer pruebas unitarias en este por lo que hay no excusa para no hacer pruebas y ser mejores desarrolladores.

# Estructura de archivos Ducks para Redux.js

Al realizar una aplicación con Redux es muy común manejar la siguiente estructura de archivos:

```
|_ /actions   # Los creadores de acciones
|_ /constants # Las constantes, como los tipos de acciones
|_ /reducers  # Los reducers de la aplicación
```

Aunque esta forma funciona, con el tiempo uno se encuentra casos donde un reducer tiene un solo tipo de acción posible y por lo tanto un solo creador de acciones. Y sin embargo terminamos creando al menos tres archivos para eso (aunque los tipos de acciones se pueden guardar todos juntos).

Para solucionar eso existe Ducks[29].

## Que es

Ducks es una forma de modularizar partes de una aplicación de Redux juntando reducers, tipos de acciones y creadores de acciones juntos de una forma fácil de entender y portar.

El nombre del formato (**ducks**) viene de la pronunciación de la última sílaba de Redux en inglés.

## Como funciona

Veamos un ejemplo simple en código primero.

```
// tipo de acción
const CHANGE_FILTER = "my-app/filters/CHANGE_FILTER";
// nuestro reducer
export default function reducer(state = "", action = {}) {
  switch (action.type) {
    case CHANGE_FILTER:
      return action.payload;
    default:
      return state;
  }
}
// creador de acciones
export function changeFilter(filter) {
  return { type: CHANGE_FILTER, payload: filter };
}
```

[29] https://github.com/erikras/ducks-modular-redux

Un módulo de **Ducks** debe seguir ciertas reglas (que se ven reflejadas en el código anterior).

## Reglas

Un módulo...

1. DEBE exportar por defecto una función llamada `reducer()`.
2. DEBE exportar sus creadores de acciones como funciones.
3. DEBE definir sus tipos de acciones en el formato `modulo-app/reducer/ACTION_TYPE`.
4. PUEDE exportar sus tipos de acciones como `UPPER_SNAKE_CASE` si otro reducer la va a usar o si esta publicada como una librería reusable.

## Como usarlo

Para usarlo simplemente importas el **duck** en tu listado de reducers de la siguiente forma.

```
import { combineReducers } from "redux";
// los reducers
import filters from "./ducks/filters";
export default combineReducers({
  filters
});
```

También es posible importar los creadores de acciones.

```
import * as filtersActions from "./ducks/filters";
// o
import { changeFilter } from "./ducks/filters";
```

Y así vas a importarlos listos para ser usados

## Conclusión

Implementar e incluso migrar a esta estructura es muy fácil y ayuda mucho a mejorar nuestra experiencia como desarrolladores al hacer nuestros proyectos son más mantenibles a largo plazo y fácil de entender para nuevos desarrolladores.

# Creando código modular con ducks de Redux

En el capítulo anterior hablamos que una buena práctica al momento de ordenar nuestro código de Redux.js es usar el formato de módulos ducks[30].

Este formato nos dice que nuestros módulos deben tener sus tipos de acciones, sus creadores de acciones y su reducer en un solo archivo, y debe exportar estos últimos dos para que sean usados en nuestra aplicación.

Ahora vamos a ver como podemos crear un módulo usando este formato fácilmente con una librería que nos ahorra mucho trabajo.

## Instalando dependencias

Primero vamos a instalar lo que necesitamos:

```
npm i -S redux redux-duck immutable
```

- redux-duck[31]: Librería utilitaria para crear ducks fácilmente.
- immutable[32]: Librería para trabajar con estructuras de datos inmutable.

## Creando nuestro duck

Vamos a crear un duck para controlar un listado de mensajes en una aplicación de chat. Para eso vamos a crear un archivo donde vamos a tener nuestro módulo, dentro vamos a colocar el siguiente código.

```
import { createDuck } from "redux-duck";
```

Primero importamos la función `createDuck` de **redux-duck**.

```
const duck = createDuck("messages", "chat");
```

Luego vamos a crear nuestro duck, el primer parámetro que vamos a pasar es el nombre del mismo, el segundo es el nombre de la aplicación, este parámetro es opcional.

## Definiendo tipos de acciones

[30] https://github.com/erikras/ducks-modular-redux
[31] https://github.com/sergiodxa/redux-duck
[32] https://github.com/facebook/immutable-js

```
const ADD_MESSAGE = duck.defineType("ADD_MESSAGE");
const REMOVE_MESSAGE = duck.defineType("REMOVE_MESSAGE");
```

Luego vamos a definir los tipos de acciones que vamos a tener en nuestro módulo. El método `defineType` recibe como único parámetro un string con el nombre de la acción y devuelve un nuevo string con el formato:

```
app - name / duck - name / ACTION_NAME;
```

Donde el nombre de la aplicación es opcional, como dijimos antes. En nuestro ejemplo quedarían dos strings con este formato:

```
ADD_MESSAGE === "chat/messages/ADD_MESSAGE";
REMOVE_MESSAGE === "chat/messages/REMOVE_MESSAGE";
```

## Creando nuestros creadores de acciones

```
export const addMessage = duck.createAction(ADD_MESSAGE);
export const removeMessage = duck.createAction(REMOVE_MESSAGE);
```

Luego vamos a exportar el resultado de ejecutar el método `createAction` pasándole los tipos de acciones que definimos antes.

Este método nos devuelve una función que crea objetos de acciones con la propiedad `type` igual al valor que le indicamos al crearla. Esta función puede recibir cualquier valor como parámetro y lo va a definir como `payload` de la acción devuelta.

## Creando nuestra función reductora

```
const initialState = {
  list: Immutable.List(),
  data: Immutable.Map()
};
export default duck.createReducer(
  {
    [ADD_MESSAGE]: (state, { payload }) => {
      return {
        list: state.list.push(payload.id + ""),
        data: state.data.set(payload.id + "", payload)
      };
    },
    [REMOVE_MESSAGE]: (state, { payload }) => {
      return {
        list: state.list.filterNot(id => id === payload.id),
        data: state.data
```

```
      };
    }
  },
  initialState
);
```

Por último creamos y hacemos un `export default` del valor devuelto por el método `createReducer`.

Este recibe dos parámetros, el primero es un objeto cuyos nombres de propiedades sean los strings creados anteriormente al definir los tipos de acciones y los valores sean funciones que reciben el estado y la acción.

Y como segundo parámetro recibe el estado inicial de nuestro reducer, este puede ser un string o un objeto. Esta función nos devuelve nuestra función reductora (reducer) que luego exportamos para que se pueda usar.

# Código final

```
// importamos createDuck de redux-duck
import { createDuck } from "redux-duck";
// importamos Immutable para usarlos luego en nuestro estado inicial
import Immutable from "immutable";
// creamos nuestro duck
const duck = createDuck("messages", "chat");
// definimos los tipos de acciones
const ADD_MESSAGE = duck.defineType("ADD_MESSAGE");
const REMOVE_MESSAGE = duck.defineType("REMOVE_MESSAGE");
// creamos nuestros creadores de acciones
export const addMessage = duck.createAction(ADD_MESSAGE);
export const removeMessage = duck.createAction(REMOVE_MESSAGE);
// definimos el estado inicial de nuestro
const initialState = {
  list: Immutable.List(),
  data: Immutable.Map()
};
// creamos nuestra función reductora (reducer)
export default duck.createReducer(
  {
    [ADD_MESSAGE]: (state, { payload }) => {
      return {
        list: state.list.push(payload.id + ""),
        data: state.data.set(payload.id + "", payload)
      };
    },
    [REMOVE_MESSAGE]: (state, { payload }) => {
      return {
        list: state.list.filterNot(id => id === payload.id),
        data: state.data
      };
```

```
    }
  },
  initialState
);
```

## Conclusión

Modularizar nuestro código en ducks nos ayuda tener código más fácil de mantener, probar y reutilizar, y la librería redux-duck nos facilita crear estos módulos de una forma sencilla y organizada.

# Manejo de errores en Redux.js

¿Que ocurre si una acción llega con un dato mal formado? ¿Si a un reducer le falta un punto y coma? Cuando trabajamos con código no hay forma de evitar al 100% los errores. Por esa razón es muy importante capturarlos para que no rompan nuestra aplicación y enterarnos si pasó algo.

En Redux nuestro código donde es muy probable que hayan errores son los reducers[33] y los middlewares[34], si quisiéramos capturar los errores en ambos por separado tendríamos que a cada reducer y a cada middleware envolverlos en un `try/catch` y manejar sus errors individualmente.

¡Pero para evitar esto podemos usar un middleware!

## Creando nuestro middleware

Vamos a ver entonces como crear un middleware que capture nuestros errores:

```
const tryCatch = errorHandler => () => next => action => {
  try {
    return next(action);
  catch (error) {
    return errorHandler(error);
  }
};
```

Ese va a ser nuestro middleware. Básicamente lo que hace es intenta llamar al siguiente middleware (o reducer) y si en algún momento hay un error lo captura y ejecuta la función `errorHandler` que le hayamos pasado al instanciarlo.

Ahora para aplicar nuestro middleware es tan fácil como hacer lo siguiente al crear nuestro Store[35].

```
import { createStore, applyMiddleware } from "redux";
// nuestro middleware
import tryCatch from "./middlewares/try-catch";
// otro posible middleware de un tercero
import someMiddlware from "some-middleware";
import reducer from "./reducer";
function errorHandler(error) {
  // acá hacemos algo con el error como mostrarlo en consola
  // o mejor aún mandarlo a algún sistema como Sentry o track:js
  // ¡incluso a nuestro propio servicio interno!
  console.error(error);
```

[33] http://es.redux.js.org/docs/basico/reducers.html
[34] https://medium.com/react-redux/middlewares-en-redux-js-88081fcd6c91
[35] http://es.redux.js.org/docs/api/create-store.html

```
}
export default createStore(
  reducer,
  applyMiddleware(tryCatch(errorHandler), someMiddleware)
);
```

Como se puede ver es bastante fácil de capturar nuestros errores con un simple middleware, solo tenemos que asegurarnos de que siempre sea el primer middleware para que pueda atrapar los errores de todo nuestro código.

# Usando uno ya hecho

Si no queremos crear nuestro propio middleware para capturar errores podemos simplemente usar uno ya hecho descargándolo de npm:

```
npm i -S redux-catch
```

- redux-catch[36]: Middleware para capturar errores
  Ahora solo tendríamos que importar **redux-catch** en vez de nuestro propio middleware y pasarle nuestra función `errorHandler` para que sepa que hacer con los errors y ya estamos listo.

Adicionalmente el `errorHandler` que le pasemos a **redux-catch** recibe como segundo parámetro el método `getState` que nos puede servir para saber el estado de la aplicación en el momento del error.

# Conclusión

Nuestra aplicación puede tener errores por un montón de razones, ya sean errores de sintaxis, algún bug o que simplemente un dato llegó mal formado y nuestro código no supo que hacer.

Y capturarlos es importante para evitar que nuestra aplicación se rompa y ya no le funcione al usuario e implementar un middleware tan simple nos puede ayudar mucho a dar una mejor experiencia de usuario e incluso a arreglar bugs antes de que el usuario se entere.

[36] https://github.com/sergiodxa/redux-catch

# Usando Redux en el servidor con Socket.io

Redux fue hecho para controlar el estado de la UI de una aplicación. Resulta que mientras podamos tener una única instancia del store Redux también puede servir en Backend, por ejemplo en aplicaciones Real-time usando Socket.io, donde el estado de la aplicación se mantendría, e incluso compartiría entre varios usuarios conectados.

## Instalación de dependencias

```
npm i -S redux socket.io socket.io-client redux-duck
```

- socket.io[37]: Librería para trabajar fácilmente con WebSockets en Node.js
- socket.io-client[38]: Cliente para conectarse a un servidor de WebSockets.
- redux-duck[39]: Librería para crear ducks de Redux.

## Creando nuestro Store y Reducers

Lo primero que vamos a hacer es crear los reducers, imaginemos que tenemos una aplicación de chat, el estado de nuestra aplicación podría ser algo así:

```
type user = {
  id: number,
  username: string
};
type message = {
  id: number,
  author: number,
  content: string
};
type state = {
  users: Map<user>,
  messages: List<message>
};
```

Vamos entonces a crear los ducks de nuestra aplicación.

[37] https://github.com/socketio/socket.io/
[38] https://github.com/socketio/socket.io-client
[39] https://github.com/sergiodxa/redux-duck

```
import { createDuck } from "redux-duck";
import { List as list } from "immutable";
const duck = createDuck("messages", "chat");
const ADD = duck.defineType("ADD");
const REMOVE = duck.defineType("REMOVE");
export const addMessage = duck.createAction(ADD);
export const removeMessage = duck.createAction(REMOVE);
export default duck.createReducer(
  {
    [ADD]: (state, { payload = {} }) => {
      return state.push(map(payload));
    },
    [REMOVE]: (state, { payload = {} }) => {
      return state.filterNot(message => message.get("id") === payload.id);
    }
  },
  list()
);
```

Ese va a ser nuestro duck para los mensajes del chat.

```
import { createDuck } from "redux-duck";
import { Map as map } from "immutable";
const duck = createDuck("user", "chat");
const ADD = duck.defineType("ADD");
export const addUser = duck.createAction(ADD);
export default duck.createReducer(
  {
    [ADD]: (state, { payload = {} }) => {
      return state.set(payload.id + "", map(payload));
    }
  },
  map()
);
```

Y ese va a ser nuestro duck para manejar los usuarios. Como vemos nuestros ducks son muy simples, solo podemos agregar usuarios y en cuanto a los mensajes solo podemos agregar y quitar mensajes.

Ahora para crear nuestro reducer nos traemos los de nuestros ducks:

```
import { combineReducers } from "redux";
import messages from "./ducks/messages";
import users from "./ducks/users";
export default combineReducers({
  messages,
  users
});
```

Y con eso ya tenemos nuestro reducer listo. Ahora vamos a crear nuestro servidor de WebSockets.

## Servidor de WebSockets

Una vez que tenemos nuestro reducer y nuestros creadores de acciones vamos a crear un servidor de WebSockets usando socket.io.

```
import Server from "socket.io";
export default function startServer(store) {
  // creamos el servidor escuchando el puerto que
  // recibimos como variable de entorno
  const io = new Server().attach(process.env.PORT);
  // nos suscribimos a los cambios del store y
  // mandamos el estado actual en cada cambio por
  // el canal 'state' de socket.io
  store.subscribe(() => io.emit("state", JSON.stringify(store.getState())));
  // cuando el usuario se conecte
  io.on("connection", socket => {
    // le emitimos el estado actual
    socket.emit("state", JSON.stringify(store.getState()));
    // y escuchamos cada acción que mande
    socket.on("action", store.dispatch.bind(store));
  });
  return io;
}
```

Luego vamos a iniciar nuestro servidor y pasarle nuestro store.

```
import { createStore } from "redux";
import reducer from "./reducer";
import startServer from "./start-server";
const store = createStore(reducer);
const server = startServer(store);
```

Ahora solo nos queda levantar nuestro servidor usando Node.js

```
npm start
```

Con esto ya tenemos un servidor de WebSockets cuyo estado se guarda en Redux. Ahora vemos como sería un cliente sencillo.

## Cliente web

Primero vamos a crear un duck para nuestra aplicación.

```
import { createDuck } from 'redux-duck';
const duck = createDuck('client', 'chat');
const UPDATE_STATE = duck.defineType('UPDATE_STATE');
export const updateState = duck.createAction(UPDATE_STATE);
export default duck.createReducer({
  [UPDATE_STATE]: (state, { payload = {} }) => {
    return payload;
  },
], {});
```

Ese super simple duck va a ser toda nuestra aplicación de Redux en el Frontend. Ahora vamos a iniciar nuestro Store y conectarnos al servidor de sockets.

```
import io from 'socket.io-client';
import { createStore } from 'redux';
import reducer, { updateState } from './duck/client';
// nos conectamos al servidor
const socket = io('http://localhost');
// al conectarnos
socket.on('connect', initialState => {
  // recibimos el estado inicial y creamos el store
  const store = createStore(reducer, initialState);
  // cuando el servidor nos mande una actualización
  // despachamos la acción updateState
  socket.on('state', nextState => {
    store.dispatch(updateState(nextState);
  });
});
```

Con eso ahora el estado de nuestra aplicación nos va a llegar por WebSockets y con eso vamos a iniciar nuestro Store, además en cada cambio que se realice en el servidor vamos a recibir todo el nuevo estado y vamos a actualizar el Store.

Por último en nuestra aplicación nos tocaría que cada acción despachada se envíe por WebSockets en el canal `action` de forma que llegue al servidor y se actualice el Store ahí guardado.

Una última idea podría ser implementar un middleware en nuestro Store del lado del servidor que se encargue de guardar en una base de datos el payload de cada acción para que no se pierdan datos si se cae el servidor.

# Conclusión

Este ejemplo es super simple, y no recomiendo que se use tal cual en producción. En una aplicación de verdad donde queramos replicar el Store en el servidor lo ideal sería que nuestro servidor de WebSockets nos mande el estado inicial del Store al conectarnos y luego cada acción que se realice, las cuales deberían crearse en el cliente que genera la acción y mandarlas por socket.io.

De esta forma en el cliente solo actualicemos lo necesario y no todo el estado de nuestra aplicación de golpe, esto reduciría la cantidad de datos enviados por WebSockets (menor consumo de datos en

móviles), de la misma forma el servidor debería despachar a su Store propio la acción, así el estado ahí almacenado se mantendría actualizado para la próxima persona en conectarse y mientras cada cliente tiene su propio Store y se encarga de actualizarse.

Un problema que podría llegar a ocurrir de usar esta forma es que si un cliente se desconecta no le llegarían algunas acciones, lo cual puede significar una perdida de datos y en dejar de estar sincronizado con el Store. Esto se puede solucionar ya sea enviando todo el estado cada X tiempo o crear alguna especie de cola de acciones cuando el servidor detecte una desconexión del cliente.

# Renderizando aplicaciones de Redux en el servidor

Renderizar en el servidor una aplicación hecha con React.js[40] nos da una gran mejora de performance, o más bien de percepción de performance, lo cual de cara al usuario se convierte en una mejor UX al parecer que el sitio carga más rápido.

Incluso gracias a renderizar en el servidor es posible hacer aplicaciones que funcionen sin JS (Server First[41]) y que una vez descargado e iniciado JS funcionen como una aplicación de una página (SPA[42]).

Cuando es solo una aplicación en React.js es fácil realizarlo. Pero cuando lo combinamos con Redux[43] necesitamos crear una instancia del Store del mismo en cada petición para que esto funcione.

## Instalando dependencias

Primero, como siempre, vamos a instalar nuestras dependencias.

```
1  npm i -S react react-dom react-redux redux micro
```

- micro[44] librería para crear micro servicios HTTP.

## Preparando el servidor

Una vez instaladas vamos a crear un servidor muy básico. El servidor lo vamos a crear como un micro servicio, de esa forma podemos renderizar nuestra aplicación sin importar si usamos Node.js o no como backend.

[40] https://medium.com/@sergiodxa/renderizando-react-js-en-el-server-con-express-js-y-react-engine-903de08c3df6
[41] https://ponyfoo.com/articles/server-first-apps
[42] https://es.wikipedia.org/wiki/Single-page_application
[43] https://medium.com/@sergiodxa/combinando-react-js-y-redux-js-7b45a9dc39ac
[44] https://github.com/zeit/micro

```
// cargamos micro para crear el server
import micro, { json, send, sendError } from "micro";
async function serverRender(request, response) {
  try {
    // obtenemos el body del request para saber los datos
    const body = await json(request);
    // mandamos la respuesta
    return send(response, 200, "hola mundo");
  } catch (error) {
    // si hay un error mandamos el error
    sendError(request, response, error);
  }
}
const server = micro(serverRender);
server.listen(process.env.PORT || 3000);
```

Ese servidor va a ser nuestra base. Dentro vamos a colocar el código. Pero primero vamos a explicar como va a funcionar esto.

Cuando el usuario entre a nuestra aplicación, digamos que en Django, vamos recibir la petición, Django se tiene que encargar de obtener todos los datos necesarios para nuestra vista y va a mandar una petición HTTP a nuestro micro servicio.

Nuestro micro servicio entonces va a renderizar el HTML que corresponde y va a devolvérselo a Django para que lo inyecte en alguna plantilla HTML y lo mande al usuario, luego ya podríamos renderizar en el navegador y que funcione como una aplicación de una página.

## Renderizando React.js

Lo primero es que vamos a definir que datos va a recibir nuestro micro servicio.

```
type body = {
  component: string, // el path al componente
  props: Object // los datos necesarios
};
// ejemplo
const body = {
  component: "/build/server/Home.js",
  props: {
    list: []
  }
};
```

Ahora que ya sabemos esto vamos a hacer que nuestro micro servicio renderice React.

```
import micro, { json, send, sendError } from "micro";
import React from "react";
import { renderToString } from "react-dom/server";
async function serverRender(request, response) {
  try {
    // obtenemos component y props del body
    const { component, props } = await json(request);
    // cargamos el componente que recibimos
    const Component = require(component);
    // lo renderizamos a HTML
    const html = renderToString(<Component {...props} />);
    // respondemos con el HTML
    return send(response, 200, html);
  } catch (error) {
    sendError(request, response, error);
  }
}
const server = micro(serverRender);
server.listen(process.env.PORT || 3000);
```

Con esto ya tenemos un pequeño micro servicio que al recibir un request con el componente a renderizar y los datos necesarios devuelve el HTML generado.

# Implementando Redux

Si queremos usar Redux en nuestra aplicación, para poder usarlo en el servidor vamos a necesitar instanciar un Store de Redux por cada request y darle ese Store a nuestra aplicación de React.

Para hacer esto podemos hacer que nuestro servidor se encargue de crear el Store y luego de mandárselo a React, pero ya que nuestro micro servicio no sabe que estamos renderizando, solo recibe el componente y los datos y renderiza, no nos sirve hacer esto.

Para nuestro caso lo que vamos a hacer es que el componente que renderizamos reciba los datos que genera la instancia del servidor, algo similar a esto:

```
import React from "react";
import { createStore } from "redux";
import { Provider } from "react-redux";
import reducer from "my-app/reducer.js";
import App from "my-app/containers/App.jsx";
function ServerProvider(props) {
  const store = createStore(reducer, props);
  return (
    <Provider store={store}>
      <App />
    </Provider>
  );
}
export default ServerProvider;
```

De esta forma el entry point de nuestra aplicación para servidor va a ser nuestro componente `ServerProvider`, el cual va a recibir como props el estado inicial de la aplicación, va a crear el Store y devolver una aplicación de React conectada a Redux.

Con esto hecho de esta forma ya ni siquiera necesitamos modificar nuestro servidor, ya que con solo pasarle el path a nuestro `ServerProvider` y el estado inicial ya tenemos todo listo para generar el HTML para hacer server-render.

## Renderizado con props

Puede pasar, y es muy común, que nuestro componente `App` reciba sus propios props, datos que no es necesario o no tiene sentido que estén guardados en el estado global (por ejemplo si son inmutables).

En ese caso nuestro micro servicio no nos permite usar esa funcionalidad, así que vamos a modificar tanto el `ServerProvider` como el micro servicio para poder realizarlo.

```
function ServerProvider(props) {
  const store = createStore(reducer, props.initialState);
  return (
    <Provider store={store}>
      <App {...props.initialProps} />
    </Provider>
  );
}
```

Ahora nuestro `ServerProvider` va a recibir dos datos vía props, el primero es el `initialState` el cual es usado para crear el Store de Redux. El segundo es `initialProps` el cual es usado como los props de nuestro componente `App`.

```
async function serverRender(request, response) {
  try {
    // obtenemos component, initialState e initialProps del body
    const { component, initialState, initialProps = {} } = await json(request);
    // cargamos el componente que recibimos
    const Component = require(component);
    // lo renderizamos a HTML pasando
    // el estado y los props iniciales
    const html = renderToString(
      <Component initialState={initialState} initialProps={initialProps} />
    );
    // respondemos con el HTML
    return send(response, 200, html);
  } catch (error) {
    sendError(request, response, error);
  }
}
```

Con esta modificación a nuestro micro servicio podemos obtener de los datos que recibimos de la petición el estado y los props iniciales y mandarlos al componente (nuestro `ServerProvider`) para que renderice la aplicación y gracias al valor por defecto del `initialProps` en caso de no recibir nada igual va a funcionar.

# Conclusión

Como se puede ver renderizar una aplicación con React y Redux no es complicado y solo es necesario realizar un paso más que si usáramos solo React y los beneficios para la UX son muy buenos gracias a que el usuario desde el primer momento puede recibir datos.

Mi recomendación es que siempre rendericen en el servidor sus aplicaciones, incluso que apliquen la metodología Server-First o Progressive Enhancement para que su aplicación no requiere de JS para sus funciones básicas. Con React y Redux renderizar en el servidor es más fácil que nunca.

# Obteniendo datos en aplicaciones de Redux

Ya sabemos como usar Redux y como despachar acciones para modificar el estado, pero ¿Qué pasa si queremos traernos más datos desde el servidor?

Es muy común que esto ocurra ya que nuestras aplicaciones web interactúan con un servidor constantemente mediante peticiones HTTP, ya sea usando AJAX o Fetch.

## Definiendo el API

Para poder hacer esto vamos a suponer que tenemos un API de artículos de un blog. Nuestros endpoint van a ser algo así:

```
GET    /api/v1/posts/
POST   /api/v1/posts/
GET    /api/v1/posts/:id
PUT    /api/v1/posts/:id
DELETE /api/v1/posts/:id
```

## Creando un cliente para el API

Lo primero que vamos a hacer es crear un objeto para consumir este API

```
import "isomorphic-fetch";
const endpoint = "/api/v1/posts";
const api = {
  posts: {
    async read(id = null) {
      try {
        if (!id) {
          const response = await fetch(endpoint);
          const data = await response.json();
          return Promise.resolve(data);
        }
        const response = await fetch(`${endpoint}/${id}`);
        const data = await response.json();
        return Promise.resolve(data);
      } catch (error) {
        return Promise.reject(error);
      }
```

```
    },
    async create(data) {
      try {
        const response = await fetch({
          url: endpoint,
          method: "POST",
          body: data
        });
        const data = await response.json();
        return Promise.resolve(data);
      } catch (error) {
        return Promise.reject(error);
      }
    },
    async update(id, data) {
      try {
        const response = await fetch({
          url: `${endpoint}/${id}`,
          method: "PUT",
          body: data
        });
        const data = await response.json();
        return Promise.resolve(data);
      } catch (error) {
        return Promise.reject(error);
      }
    },
    async delete(id) {
      try {
        const response = await fetch({
          url: `${endpoint}/${id}`,
          method: "DELETE"
        });
        const data = await response.json();
        return Promise.resolve(data);
      } catch (error) {
        return Promise.reject(error);
      }
    }
  }
};
export default api;
```

Este objeto nos va a servir como cliente para consumir el api, de esta forma podemos hacer peticiones con líneas como `api.posts.read(1)` el cual devolvería el post con el ID uno.

## Middleware para acciones asíncronas

Ya que trabajamos con Redux tiene sentido que nuestras peticiones sean acciones, en este caso asíncronas. Para poder trabajar de esta forma vamos a crear un pequeño middleware que nos permita

despachar acciones asíncronas.

```
function asyncAwait(asyncResolver) {
  return store => next => action => {
    // despachamos la acción normalmente
    const result = next(action);
    // ejecutamos el asyncResolver pasándole
    // la acción y la función dispatch
    asyncResolve(action, store.dispatch);
    // retornamos la acción despachada
    return result;
  };
}
```

Este pequeño middleware nos va a permitir despachar acciones y que estas lleguen a una función que llamamos `asyncResolver`, la misma va a ser una función asíncrona que al recibir ciertas acciones va a empezar el proceso asíncrono para obtener datos.

```
import api from 'my-app/utils/api.js';
async function requestPost(payload, dispatch) {
  try {
    // hacemos el request
    const data = await api.posts.read(payload.id);
    // despachamos una acción con los datos
    dispatch({
      type: 'POST_SUCCEED'
      payload: data,
    });
  } catch (error) {
    // despachamos el error para mostrarlo en la UI
    return dispatch({
      type: 'POST_FAILED',
      payload: error.message,
      error: true,
    });
  }
}
async function asyncResolve(action, dispatch) {
  switch (action.type) {
    case 'POST_REQUEST':
      return requestPost(action.payload, dispatch);
    default:
      return action;
  }
}
export default asyncResolver;
```

Como vemos en el código de arriba, el `asyncResolver` al igual que los reducer se encarga de verificar cual es el tipo de acción que recibimos y se encarga realizar la petición y despachar una acción con al respuesta, o en caso de un error despachar el mensaje para que el usuario se pueda enterar.

## Implementando el Middleware

Por último, necesitamos hacer que nuestro store sepa que existe el middleware y le pase las acciones.

```
import { createStore, applyMiddleware } from "redux";
import reducer from "my-app/reducer.js";
import asyncAwait from "my-app/async-await.js";
import asyncResolver from "my-app/async-resolver";
export default initialState =>
  createStore(
    reducer,
    initialState,
    applyMiddleware(asyncAwait(asyncResolver))
  );
```

Con esto podemos crear un archivo store.js que reciba el estado inicial y nos devuelva un store aplicándole el middleware que creamos y nuestro asyncProvider.

## Conclusión

Ahora ya estamos listos para empezar a despachar acciones para iniciar peticiones y que luego el asyncProvider sepa que tiene que hacer para cada tipo de acción y actuar en consecuencia.

Por último, la forma en que estamos centralizando todo es más o menos como funciona Redux Saga, una de las librerías más populares para trabajar con flujos de datos asíncronos haciendo uso de los Generadores de ES2015.

# Estado inmutable con Redux e Immutable.js

Redux nos propone tratar nuestro estado como inmutable. Sin embargo los objetos y array en JavaScript no lo son, lo que puede causar que mutemos directamente el estado por error.

Immutable.js es una librería creada por Facebook para usar colecciones de datos inmutables como listas, mapas, sets, etc. Usándolo con Redux nos permite expresar nuestro estado como colecciones de Immutable.js para evitarnos estos posibles problemas al mutar datos.

## Usándolo en un reducer

La primera forma de usar Immutable.js en Redux es usándolo directo en un reducer. Simplemente definiendo el estado inicial como una colección inmutable y luego modificándolo según la acción despachada.

```
import { Map as map } from "immutable";
function reducer(state = map(), { type, payload }) {
  switch (type) {
    case "ADD": {
      return state.set(payload.id.toString(), map(payload));
    }
    default: {
      return state;
    }
  }
}
export default reducer;
```

De esta forma podemos empezar a hacer uso de Immutable.js. Un pequeño detalle al usar mapas inmutables es que el key usado debe ser siempre un string, puede ser un número, pero por experiencia esto pueda dar errores de que Immutable.js no encuentre el valor al hacer `collection.get(1)`, por esa razón cuando agregamos el dato a nuestro mapa usamos `.toString()`sobre el ID para evitarnos este problema.

## Combinando reducers

Aunque es posible tener un único reducer para toda la aplicación, a medida que esta crece lo común es empezar a dividirlo en múltiples reducers y usar `redux.combineReducers` para unirlos en uno solo que usamos al crear el Store.

```
import { combineReducers } from "redux";
import data from "./reducers/data.js";
export default combineReducers({
  data
});
```

De esta forma nuestro estado ahora es un objeto con una propiedad `data` la cual posee nuestra colección inmutable, pero ¿Qué pasa si queremos que todo nuestro estado sea un conjunto de colecciones inmutables anidadas?

## Combinando reducers con Immutable.js

Si decidimos tratar todo el estado como una colección inmutable debemos entonces hacer uso de redux-immutable[45]. Esta librería nos ofrece una función `combineReducers` personalizada la cual funciona con exactamente la misma API que la oficial de Redux, por lo que hacer el cambio de una a otra consiste en cambiar de donde importamos la función.

```
import { combineReducers } from "redux-immutable";
import data from "./reducers/data.js";
export default combineReducers({
  data
});
```

Como vemos simplemente pasamos de importar desde redux a hacerlo desde redux-immutable, con ese simple cambio estamos usando Immutable.js en todo nuestro store, ahora cuando conectemos nuestros componentes a este podemos usar una sintaxis 100% de Immutable.js.

```
function getItem(state, props) {
  return state
    .get('data')
    .get(props.id.toString())
    .toJS(),
}
```

Ese selector por ejemplo se encarga de traerse del mapa de datos el item con el ID recibido como prop y devolverlo convertido a un objeto de JS común que podemos recibir en un componente y usarlo sin problemas.

## Conclusión

Usar Immutable.js nos permite trabajar con un estado verdaderamente inmutable evitando problemas comunes como pueden ser mutar directamente una propiedad sin crear una copia del estado lo cual puede causar errores de inconsistencia de datos y dolores de cabeza a muchos desarrolladores.

Además que Immutable.js es bastante fácil de usar por lo que incluso nos facilita nuestro trabajo como desarrolladores enormemente, por lo que vale mucho la pena empezar a usarlo.

[45] redux-immutable

# Componentes de Alto Orden en React.js

Algo que ocurre muy seguido es que varios componentes de React vayan a necesitar usar una misma funcionalidad o extenderse con funciones de terceros (como el acceso a un Store o internacionalización).

Originalmente esto se lograba gracias a la utilización de Mixins, estos eran, básicamente, objetos que poseían los métodos que queríamos compartir, un ejemplo (de la misma documentación).

```
var SetIntervalMixin = {
  componentWillMount() {
    this.intervals = [];
  },
  setInterval() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount() {
    this.intervals.forEach(clearInterval);
  }
};
```

Este mixin contiene la lógica para poder crear fácilmente un intervalo (como con `window.setInterval`) el cual se borre automáticamente al desmontarse el componente, evitándonos tener que pensar en hacer esto a mano y en causar problemas de memoria si nos olvidamos.

—

Resulta que desde que se incorporaron las clases como forma de crear componentes, y más aún con las funciones para componentes puros, ya no es posible usar mixins en React.js.

> Mixins Are Dead. Long Live Composition[46] por Dan Abramov

Las razones para esto fueron que ES2015 no tiene soporte a mixins de forma nativa por lo que en vez de crear una API propia decidieron no soportarlos. El hacer esto significó tener que buscar nuevas formas de extender componentes para agregar funcionalidades comunes.

La solución llego desde el lado de la programación funcional gracias a las Funciones de Alto Orden[47]. Estas son funciones que reciben una o más funciones como argumentos y devuelven una nueva función.

En React esto se traslada a **Componentes de Alto Orden**. Haciendo un paralelismo, es una función que recibe uno o más componentes y devuelve uno nuevo. Veamos un ejemplo super básico.

[46] https://medium.com/@dan_abramov/mixins-are-dead-long-live-higher-order-components-94a0d2f9e750#.ci1qkuncy
[47] https://medium.com/@yeion7/funciones-de-alto-orden-en-javascript-42d04769d9b5#.70vmxjj0c

```
import React from "react";
function getViewport() {
  return {
    height: window.innerHeight,
    width: window.innerWidth
  };
}
// nuestro componente de alto orden
function withViewport(WrappedComponent) {
  return function WithViewport(props) {
    return <WrappedComponent getViewport={getViewport} {...props} />;
  };
}
export default withViewport;
```

Como vemos en el ejemplo tenemos una función `getViewport` el cual nos devuelve un objeto con el width y height de nuestro navegador y nuestro HOC (High Order Component—Componente de Alto Orden) que recibe un componente y devuelve un nuevo componente puro, el cual a su vez le pasa los props que recibe al componente envuelto y adicionalmente la función `getViewport`.

Ahora nuestro `WrappedComponent` recibiría, además de sus props normales, una función llamada `getViewport`. De esta forma muy simple podemos empezar a extender la funcionalidad de nuestros componentes igual que hacíamos con los mixins. Volviendo al ejemplo anterior de mixins veamos ahora como haríamos eso mismo usando un **HOC**.

```
import React, { Component } from 'react';
function setIntervalHOC(WrappedComponent) {
  return class WithSetInterval extends Component {
    componentWillMount() {
      this.intervals = [];
    }
    setInterval(..args) {
      const id = setInterval.apply(null, args);
      this.intervals.push(id);
      return id;
    }
    componentWillUnmount() {
      this.intervals.forEach(clearInterval);
    }
    render() {
      return (
        <WrappedComponent
          setInterval={this.setInterval.bind(this)}
          {...this.props}
        />
      );
    }
  }
}
```

Esa es la versión **HOC** del mixin para usar `setInterval`, la diferencia es que ahora es una función que recibe un componente y lo renderiza pasándole el `setInterval` propio, y es el componente `WithSetInterval` el cual posee la lista de intervalos y se encarga de borrarlos al desmontarse.

De esta forma el componente envuelto solo sabe que si llama `props.setInterval` va a crear un intervalo y que automáticamente se va a borrar al desmontarse el componente. Veamos por último como lo usaríamos:

```
// ./components/App.jsx
import React, { Component } from "react";
import setIntervalHOC from "./decorators/set-interval.js";
class App extends Component {
  static propTypes = {
    timer: PropTypes.number
  };
  static defaultProps = {
    timer: 500
  };
  state = {
    amount: 0
  };
  componentDidMount() {
    this.props.setInterval(this.tick.bind(this), this.props.timer);
  }
  tick() {
    this.setState({
      amount: this.state.amount + 1
    });
  }
  render() {
    return <div>{this.state.amount}</div>;
  }
}
export default setIntervalHOC(App);
```

```
// ./index.js
import React from "react";
import { render } from "react-dom";
import App from "./components/App.jsx";
render(<App timer={1000} />, document.getElementById("app"));
```

Como vemos simplemente creamos nuestro componente que haga uso del intervalo y se exporte envuelto en `setIntervalHOC`, de esa forma cuando importemos `App` vamos a importar en realidad el componente devuelto por el **HOC** y al renderizarse mostraría primero un 0, luego de un segundo un 1, y así, cada segundo (o lo que hayamos indicado a la propiedad `timer` de `App`, o si no indicamos nada 500ms),iría aumentando hasta que se desmonte.

> Gist con el ejemplo:
> https://gist.github.com/sergiodxa/09fa274d68c929a4059bdb8000c03e49

# Conclusión

Los **Componentes de Alto Orden** son una forma excelente, y fácil de usar, para extender componentes. Este patrón es usado por ejemplo en React Redux para conectar un componente al Store de Redux.

Hay otras formas de usarlos además de la vista acá, el patrón que usamos se conoce como **PropsProxy** ya que estamos manipulando los props que llega al componente. Otro patrón es **Inheritance Inversion** que consiste en devolver un nuevo componente que extienda el componente que estamos envolviendo.

Como detalle extra, los HOC se pueden usar como decoradores siguiendo la propuesta actual (y capaz obsoleta), permitiendo usarlos de esta forma:

```
1 @setIntervalHOC
2 class App extends Component { ... }
3 export default App;
```

# Migrando a Redux

Redux no es un framework monolítico, sino un conjunto de contratos y algunas funciones que hacen que todo funcione en conjunto[48]. La mayor parte de tu "código de Redux" ni siquiera va a hacer uso de la API de Redux, ya que la mayor parte del tiempo vas a crear funciones.

Esto hace fácil migrar a o desde Redux.

¡No queremos encerrarte!

## Desde Flux

Los reducers capturan "la esencia" de los Stores de Flux, así que es posible migrar gradualmente de un proyecto Flux existente a uno de Redux, ya sea que uses Flummox[49], Alt[50], Flux tradicional[51] o cualquier otra librería de Flux.

También es posible hacer lo contrario y migrar de Redux a cualquier de estas siguiendo los siguiente pasos:

Tu proceso debería ser algo como esto:

Crea una función llamada `createFluxStore(reducer)` que cree un store de Flux compatible con tu aplicación actual a partir de un reducer. Internamente debería ser similar a la implementación de createStore[52] (código fuente[53]) de Redux. Su función dispatch solo debería llamar al reducer por cada acción, guardar el siguiente estado y emitir el cambio.

Esto te permite gradualmente reescribir cada Store de Flux de tu aplicación como un reducer, pero todavía exportar `createFluxStore(reducer)` así el resto de tu aplicación no se entera de que esto esta ocurriendo con los Stores de Flux.

Mientras reescribes tus Stores, vas a darte cuenta que deberías evitar algunos anti-patrones de Flux como peticiones a APIs dentro del Store, o ejecutar acciones desde el Store. Tu código de Flux va a ser más fácil de entender cuando lo modifiques para que funcione en base a reducers.

Cuando hayas portado todos los Stores de Flux para que funcionen con reducers, puedes reemplazar tu librería de Flux con un único Store de Redux, y combinar estos reducers que ya tienes usando `combineReducers(reducers)`

.
Ahora lo único que falta es portar la UI para que use react-redux o alguno similar.
Finalmente, seguramente quieras usar cosas como middlewares para simplificar tu código asíncrono.

[48] http://es.redux.js.org/docs/api/index.html
[49] http://github.com/acdlite/flummox
[50] http://github.com/goatslacker/alt
[51] https://github.com/facebook/flux
[52] http://es.redux.js.org/docs/api/create-store.html
[53] https://github.com/rackt/redux/blob/master/src/createStore.js

## Desde Backbone

La capa de modelos de Backbone es bastante diferente de Redux, así que no recomendamos combinarlos. Si es posible, lo mejor es reescribir la capa de modelos de tu aplicación desde cero en vez de conectar Backbone a Redux. Igualmente, si no es posible reescribirlo, capaz debería usar backbone-redux[54] para migrar gradualmente, y mantener tu Store de Redux sincronizado con los modelos y colecciones de Backbone.

[54] https://github.com/redbooth/backbone-redux

# Glosario de términos

Este es un glosario de los términos principales en Redux, junto a su tipo de dato. Los tipos están documentados usando la notación Flow.

## Estado

```
type State = any;
```

Estado (también llamado árbol de estado) es un termino general, pero en la API de Redux normalmente se refiere al valor de estado único que es manejado por el Store y devuelto por `getState()`. Representa el estado de tu aplicación de Redux, normalmente es un objeto con muchas anidaciones.

Por convención, el estado a nivel superior es un objeto o algún tipo de colección llave-valor como un `Map`, pero técnicamente puede ser de cualquier tipo. Aun así, debes hacer tu mejor esfuerzo en mantener el estado serializable. No pongas nada dentro que no puedas fácilmente convertirlo a un JSON.

## Acción

```
type Action = object;
```

Una acción es un objeto plano (POJO—Plan Old JavaScript Object) que representa una intención de modificar el estado. Las acciones son la única forma en que los datos llegan al store. Cualquier dato, ya sean eventos de UI, callbacks de red, u otros recursos como WebSockets eventualmente van a ser despachados como acciones.

Las acciones deben tener un campo type que indica el tipo de acción a realizar. Los tipos pueden ser definidos como constantes e importados desde otro módulo Es mejor usar strings como tipos en vez de `Symbols`[55] ya que los strings son serializables.

Aparte del type, la estructura de una acción depende de vos. Si estás interesado, revisa Flux Standard Action[56] para recomendaciones de como deberías estar estructurado una acción.

Revisa acción asíncrona debajo.

## Reducer

[55] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Symbol
[56] https://github.com/acdlite/flux-standard-action

```
type Reducer<S, A> = (state: S, action: A) => S;
```

Un *reducer* (también llamado *función reductora*) es una función que acepta una acumulación y un valor y devuelve una nueva acumulación. Son usados para reducir una colección de valores a un único valor.

Los reducers no son únicos de Redux—son un concepto principal de la programación funcional. Incluso muchos lenguajes no funcionales, como JavaScript, tienen una API para reducción. En JavaScript, es `Array.prototype.reduce()`[57].

En Redux, el valor acumulado es el árbol de estado, y los valores que están siendo acumulados son acciones. Los reducers calculan el nuevo estado en base al anterior estado y la acción. Deben ser *funciones puras*—funciones que devuelven el mismo valor dados los mismos argumentos. Deben estar libres de efectos secundarios. Esto es lo que permite características increíbles como hot reloading y time travel.

Los reducers son el concepto más importante en Redux.

*No hagas peticiones a APIs en los reducers.*

# Función despachadora

```
type BaseDispatch = (a: Action) => Action;
type Dispatch = (a: Action | AsyncAction) => any;
```

La *función despachadora* (o simplemente *función dispatch*) es una función que acepta una acción o una acción asíncrona; entonces puede o no despachar una o más acciones al store.

Debemos distinguir entre una función despachadora en general y la función base `dispatch` provista por la instancia del store sin ningún middleware.

La función base `dispatch` siempre envía síncronamente acciones al reducer del store, junto al estado anterior devuelto por el store, para calcular el nuevo estado. Espera que las acciones sean objetos planos listos para ser consumidos por el reducer.

Los middlewares envuelven la función `dispatch` base. Le permiten a la función `dispatch` manejar acciones asíncronas además de las acciones. Un middleware puede transformar, retrasar, ignorar o interpretar de cualquier forma una acción o acción asíncrona antes de pasarla al siguiente middleware. Lea más abajo para más información.

# Creador de acciones

[57] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce

```
type ActionCreator = (...args: any) => Action | AsyncAction;
```

Un creador de acciones es, simplemente, una función que devuelve una acción. No confunda los dos términos—otra vez, una acción es un pedazo de información, y los creadores de acciones son fabricas que crean esas acciones.

Llamar un creador de acciones solo produce una acción, no la despacha. Necesitas llama al método `dispatch` del store para causar una modificación. Algunas veces decimos *creador de acciones conectado*, esto es una función que ejecuta un creador de acciones e inmediatamente despacha el resultado a una instancia del store específica.

Si un creador de acciones necesita leer el estado actual, hacer una llamada al API, o causar un efecto secundario, como una transición de rutas, debe retornas una acción asíncrona en vez de una acción.

# Acción asíncrona

```
type AsyncAction = any;
```

Una acción asíncrona es un valor que es enviado a una función despachadora, pero todavía no esta listo para ser consumido por el reducer. Debe ser transformada por un middleware en una acción (o una serie de acciones) antes de ser enviada a la función dispatch() base. Las acciones asíncronas pueden ser de diferentes tipos, dependiendo del middleware que uses. Normalmente son primitivos asíncronos como una promesa o un thunk, que no son enviados inmediatamente a un reducer, pero despachan una acción cuando una operación se completa.

# Middleware

```
type MiddlewareAPI = { dispatch: Dispatch, getState: () => State };
type Middleware = (api: MiddlewareAPI) => (next: Dispatch) => Dispatch;
```

Un middleware es una función de orden superior que toma una función despachadora y devuelve una nueva función despachadora. A menudo convierten acciones asíncronas en acciones.

Los middlewares son combinables usando funciones. Son útiles para registrar acciones, realizar efectos secundarios como ruteo, o convertir una llamada asíncrona a una API en una serie de acciones síncronas.

Revisa `applyMiddleware(...middlewares)`[58] para más detalles de los middlewares.

# Store

[58] http://es.redux.js.org/docs/api/apply-middleware.html

```
type Store = {
  dispatch: Dispatch
  getState: () => State
  subscribe: (listener: () => void) => () => void
  replaceReducer: (reducer: Reducer) => void
}
```

Un store es un objeto que mantiene el árbol de estado de la aplicación.

Solo debe haber un único store en una aplicación de Redux, ya que la composición ocurre en los reducers.

- `dispatch(action)` es la función dispatch base descrita arriba.
- `getState()` devuelve el estado actual de la aplicación.
- `subscribe(listener)` registra una función para que se ejecute en cada cambio de estado.
- `replaceReducer(nextReducer)` puede ser usada para implementar hot reloading y code splitting. Normalmente no la vas a usar.

Revisa la referencia del API del Store[59] completa para más detalles.

## Creador de store

```
type StoreCreator = (reducer: Reducer, initialState: ?State) => Store;
```

Un creador de store es una función que crea un store de Redux. Al igual que la función despachante, debemos separar un creador de stores base, `createStore(reducer, initialState)` exportado por Redux, por los creadores de store devueltos por los potenciadores de store.

## Potenciador de store

```
type StoreEnhancer = (next: StoreCreator) => StoreCreator;
```

Un potenciador de store es una función de orden superior que toma un creador de store y devuelve una versión potenciada del creador de store. Es similar a un middleware ya que te permite alterar la interfaz de un store de manera combinable.

Los potenciadores de store son casi el mismo concepto que los componentes de orden superior de React, que ocasionalmente se los llamada "potenciadores de componentes".

Debido a que el store no es una instancia, sino una colección de funciones en un objeto plano, es posible crear copias fácilmente y modificarlas sin modificar el store original. Hay un ejemplo en la documentación de compose demostrándolo.

[59] http://es.redux.js.org/docs/api/Store.html

Normalmente nunca vas a escribir un potenciador de store, pero capaz uses el que provee las herramientas de desarrollo[60]. Es lo que permite que el time travel sea posible sin que la aplicación se entere de que esta ocurriendo. Curiosamente, la forma de implementar middleware en Redux[61] es un potenciador de store.

[60] https://github.com/gaearon/redux-devtools
[61] http://es.redux.js.org/docs/api/apply-middleware.html