

Tema 4



Redux

¿Qué es Redux?



Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as [live code editing combined with a time traveling debugger](#).

You can use Redux together with [React](#), or with any other view library.
It is tiny (2kB, including dependencies).

```
npm install -S redux  
npm install -S react-redux
```

¿Qué es Redux?

- ¿Un **contenedor de estado predecible**?
- Para escribir aplicaciones consistentes y fácil de testear
- Gran experiencia de desarrollo
- Muy muy apropiado para usar con React, aunque se puede usar con otros frameworks / librerías

¿Qué es el estado de una aplicación?

- ¿Los datos cargados del servidor vía API REST?

(Ej: lista de productos en una página de catálogo, personajes de una serie de TV...)

¿Qué es el estado de una aplicación?

- ¿El estado actual de la UI?

(Ej: usuario ha iniciado sesión o no, ha ocurrido un error al guardar un dato, un panel está colapsado o no, un DatePicker está desplegado...)

¿Qué es el estado de una aplicación?

- Ambas respuestas son complementarias. El estado es todo eso.

¿Qué es el estado de una aplicación?

- Cualquier información necesaria para representar la interfaz de usuario en un momento dado
- Simplificado: **"datos externos" + "estado UI"**

Motivos

- En aplicaciones complejas, tenemos:
 - Multitud de **cambios** (mutaciones)
 - Operaciones asíncronas (APIs, eventos de usuario, etc)
- Redux aporta claridad y sencillez en la gestión de este estado global
- Para conseguir una aplicación **determinista**

Los 3 principios de Redux

- **Todo el estado** de la aplicación se almacena en un único **objeto global inmutable**
- Los cambios en el estado se **describen** mediante **acciones**
- Los cambios en el estado son **aplicados** por **funciones puras** (*reducers*).

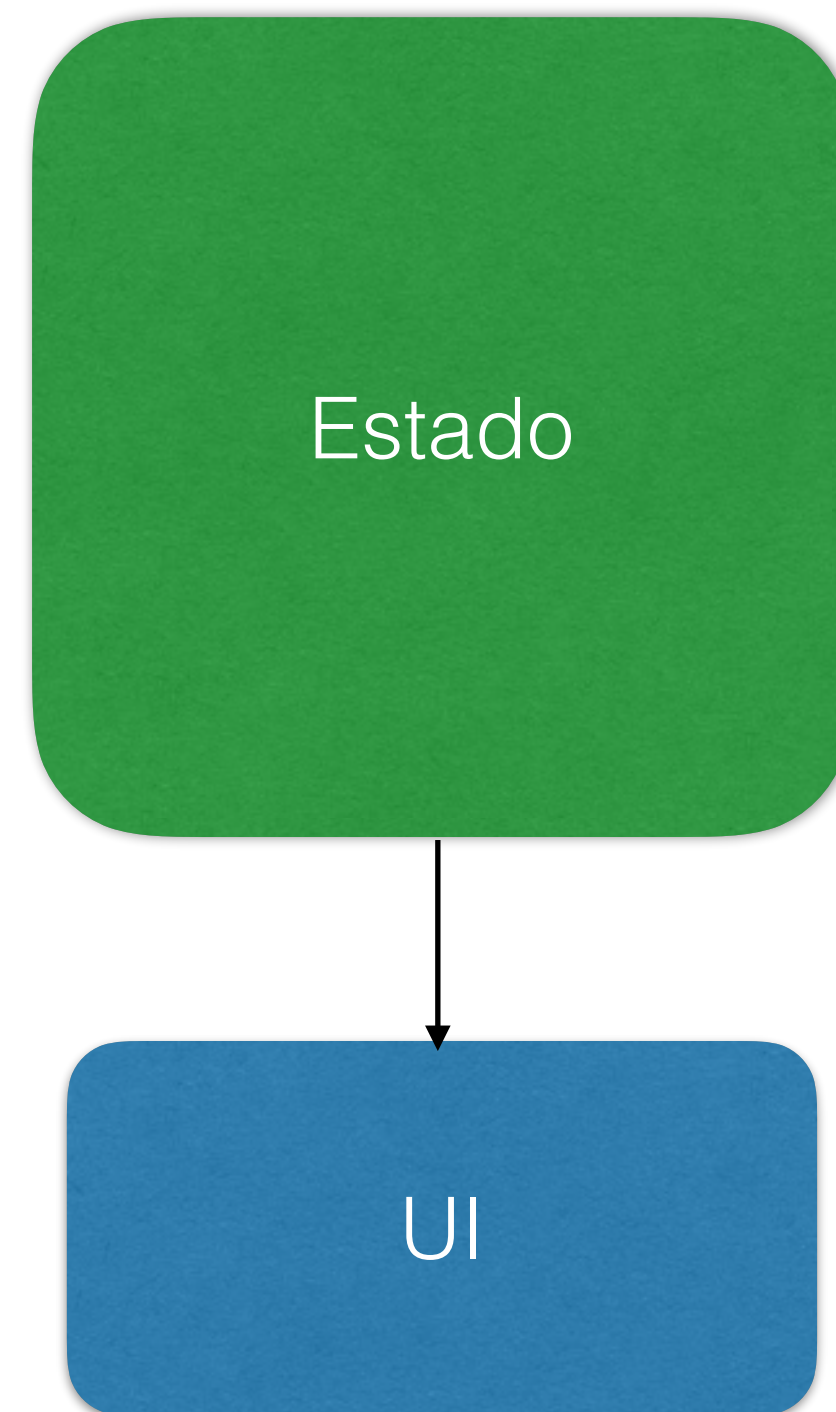
Estado global

- **Única** fuente de verdad
- Permite:
 - Generarlo en el servidor e "hidratarlo" en el cliente (aplicaciones universales o isomórficas)
 - Simplicidad para tests unitarios (sin mocks)
 - Depuración sencilla (un único sitio donde "buscar")
 - Cosas que creíamos "difíciles": Hacer / deshacer, hot-loading manteniendo el estado, guardar en localStorage...

Estado global

```
{
  characters: [ ... ],
  familyNames: ['Lannister', 'Stark', ...],
  allSeasons: [1,2,3,4,5],
  filter: {
    name: 'John',
    family: '',
    aliveOnly: false,
    seasons: []
  }
}
```

Estado global



Acciones

- **El estado global es de SOLO LECTURA**
- La única manera de mutar el estado es emitiendo una **acción**
- Una acción es un **objeto que describe qué ha ocurrido.**
- Ni las vistas ni los callbacks de APIs etc. escriben nunca directamente al estado: sólo podrán expresar la **intención de cambiar algo.**

Acciones

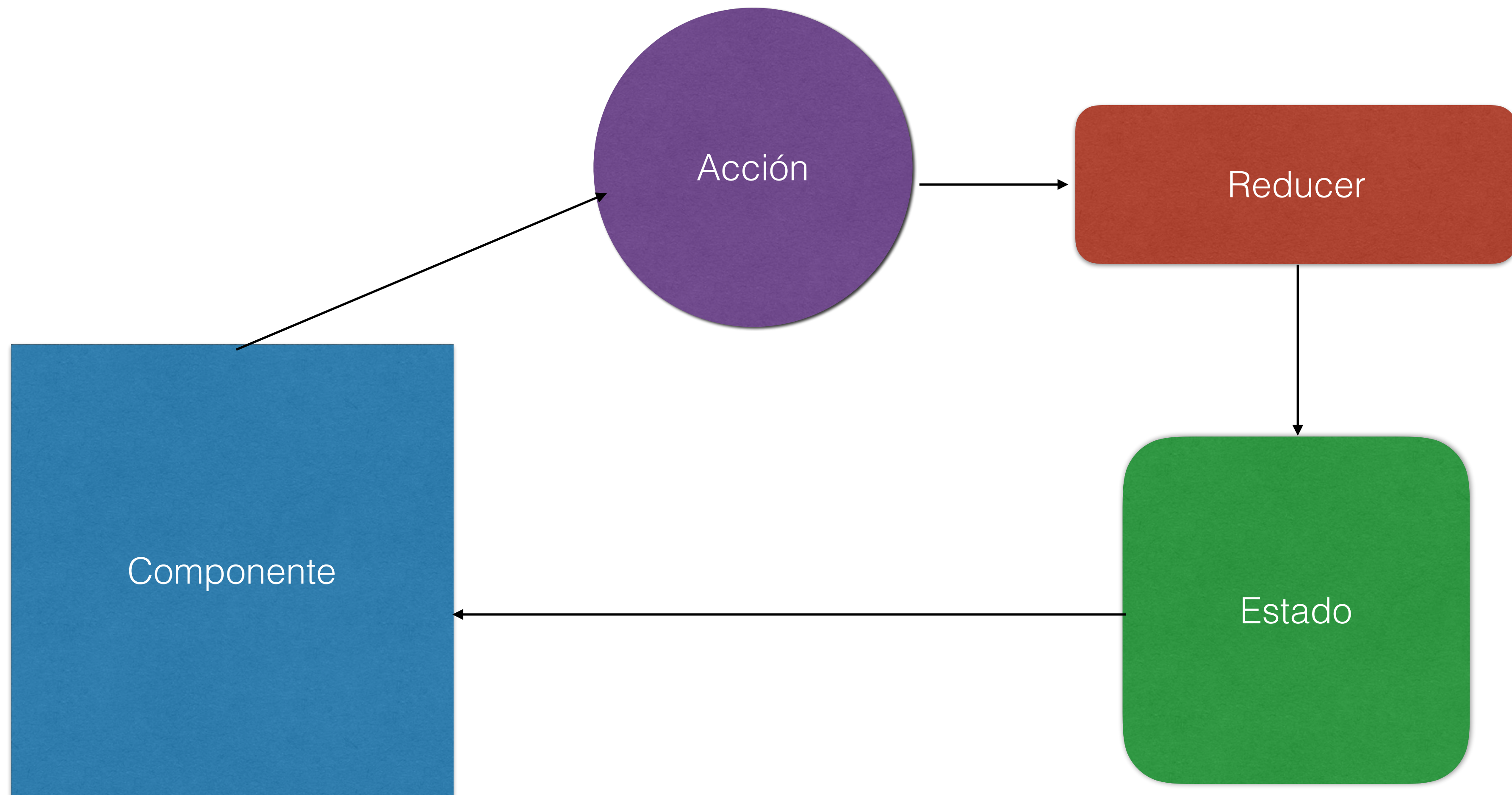
- Conseguimos que los cambios estén **centralizados**
- Las acciones pueden ser serializadas, registradas en un log, almacenadas e incluso reproducidas más tarde para test o depuración
- Como las acciones portan los datos asociados, se puede reproducir el estado de la aplicación incluso en otro browser

Acciones

- Ej: Cambiar el nombre del actor en el filtro de búsqueda
- Convención: las acciones tienen una propiedad **type** que identifica a la acción

```
{  
  type: 'CHANGE_FILTER',  
  query: {  
    name: 'Tyrion'  
  }  
}
```


Acciones



Reducers

- Para especificar como el árbol de estado es transformado por las **acciones**, escribimos funciones puras (**reducers**).

```
function filter(state={}, action) {  
  switch(action.type) {  
    case 'CHANGE_FILTER':  
      return {  
        ...state,  
        ...action.query  
      }  
    default:  
      return state  
  }  
}
```

Reducers

- Otro ejemplo: guardar datos recibidos del servidor

```
{
  type: 'SAVE_CHARACTERS',
  payload: [
    {
      name: "Jon Snow",
      family: "Stark",
      actor: "Kit Harington",
      seasons: [1,2,3,4,5],
      alive: true
    },
    ...
  ]
}
```

Reducers

- Otro ejemplo

```
function characters(state=[], action) {  
  switch(action.type) {  
    case 'SAVE_CHARACTERS':  
      return [ ...action.payload ];  
    default:  
      return state;  
  }  
}
```

Reducers - ventajas

- Testing muy sencillo: sólo hay que testar **una función!**
- Cambios centralizados: ya sabemos el único lugar de nuestro código donde ocurren cambios
- Se puede programar la "lógica" de la aplicación completa, y probarla, sin crear ni una sola vista

Reducers - ventajas

```
function characters(state=[], action) {  
  switch(action.type) {  
    case 'SAVE_CHARACTERS':  
      return action.payload;  
    default:  
      return state;  
  }  
}  
  
describe('Characters reducer', () => {  
  it('Saves character data', () => {  
    const data = [1,2,3];  
    const res = characters([], { type: 'SAVE_CHARACTERS', payload: data });  
    assert.equal(res, data);  
  })  
})
```

¿Cómo unimos todo?

- Emitir o **despachar** las acciones...
- Que pasen por nuestros reducers, junto con el estado actual, para transformarlo
- Y necesitamos **recuperar** el estado en un momento dado, a ser posible cada vez que haya cambios

Store de Redux

- Sólo existe **una instancia de Store** para toda la aplicación
- Para crear un Store, necesitamos indicar a Redux el **reducer** que debe utilizar para procesar las **acciones**, y opcionalmente el estado inicial
- Se usa la función **createStore** de redux.

Store de Redux

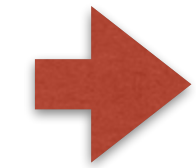
- **createStore** devuelve un objeto con la siguiente interfaz:
 - **getState()** - devuelve el estado actual
 - **dispatch(*action*)** - despacha una acción a través de nuestros reducers
 - **subscribe(*callback*)** - permite suscribirse a los cambios del estado. Devuelve una función que podemos llamar para eliminar la subscripción.

Ejemplo: contador

```
import { createStore } from 'redux';
```

```
function counter(state=0, action){  
  switch(action.type){  
    case 'INCREMENT':  
      return state + 1;  
    default:  
      return state;  
  }  
}
```

```
const store = createStore(counter, 0);
```



```
store.subscribe(() => console.log('Nuevo estado', store.getState()))  
store.dispatch({ type: 'INCREMENT' })  
//Nuevo estado 1  
store.dispatch({ type: 'INCREMENT' })  
//Nuevo estado 2  
store.dispatch({ type: 'ICRRREMENT' })  
//Nuevo estado 2
```

Ejemplo: lista de contadores

```
function counterList(state=[], action) {  
  switch(action.type) {  
    case 'ADD_COUNTER':  
      return [...state, 0];  
  
    case 'REMOVE_COUNTER':  
      let index = action.index;  
      return state.filter((value, i) => i !== index);  
  
    case 'INCREMENT':  
      let index = action.index;  
      return state.map((value, i) => {  
        if (i === index) {  
          return value + 1;  
        }  
        return value;  
      });  
    default:  
      return state;  
  }  
}
```

Ejemplo: lista de contadores

```
const store = createStore(counterList);

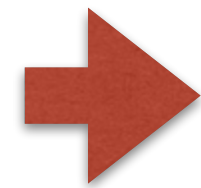
store.subscribe(() => console.log('Nuevo estado', store.getState()))
store.dispatch({ type: 'ADD_COUNTER '});
//Nuevo estado [0]
store.dispatch({ type: 'ADD_COUNTER '});
//Nuevo estado [0, 0]
store.dispatch({ type: 'INCREMENT', index: 0 })
//Nuevo estado [1, 0]
store.dispatch({ type: 'INCREMENT', index: 0 })
//Nuevo estado [2, 0]
store.dispatch({ type: 'REMOVE_COUNTER', index: 1 })
//Nuevo estado [2]
```

Store de Redux

- **createStore** no es magia, de hecho se puede implementar en 5 minutos

Store de Redux artesano

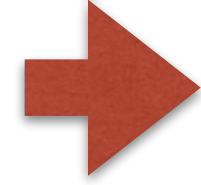
```
function createStore(reducer, initialState){  
  let state = initialState,  
      listeners = []
```



```
  const dispatch = (action) => {  
    state = reducer(state, action);  
    listeners.forEach((callback) => callback());  
  }
```

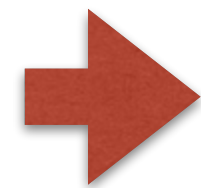


```
  const getState = () => state;
```



```
  const subscribe = (listener) => {  
    listeners = [ ...listeners, listener ];  
    return function(){  
      listeners = listeners.filter(l => l !== listener);  
    }  
  }
```

```
  //accion falsa para configurar el estado inicial  
  dispatch({ type: '@@@INIT' });
```



```
  return {  
    getState,  
    dispatch,  
    subscribe  
  }  
}
```

Combinación de reducers

- Para mantener la simplicidad, nos gustaría tener un **reducer** independiente por cada parte del estado global de la aplicación
- Pero **createStore** recibe un único reducer como argumento (una función).
- Podemos hacerlo por fuerza bruta...

Combinación de reducers

```
const initialState = {
  characters: [],
  filter: {}
}

function complexReducer(state=initialState, action){
  switch(action.type){
    case 'SAVE_CHARACTERS':
      return {
        ...state,
        characters: action.payload
      }
    case 'CHANGE_FILTER':
      return {
        ...state,
        filter: {
          ...state.filter,
          ...action.payload
        }
      }
  }
}

const store = createStore(complexReducer);
store.dispatch({ type: 'SAVE_CHARACTERS', payload: [ ... ]});
store.dispatch({
  type: 'CHANGE_FILTER',
  payload: { name: 'John' }
});
```

Combinación de reducers

```
function characters(state=[], action) {  
  switch(action.type) {  
    case 'SAVE_CHARACTERS':  
      return action.payload;  
    default:  
      return state;  
  }  
}  
  
function filter(state={}, action) {  
  switch(action.type) {  
    case 'CHANGE_FILTER':  
      return {  
        ...state,  
        ...action.payload  
      }  
  }  
}  
  
function complexReducer(state=initialState, action) {  
  return {  
    characters: characters(state.characters, action),  
    filter: filter(state.filter, action)  
  }  
}  
  
const store = createStore(complexReducer);  
...
```


Combinación de reducers

- Redux nos ofrece una función que hace esto por nosotros: **combineReducers**
- Recibe como único argumento un objeto con la forma del estado, y los reducers que se ocupan de cada parte
- Devuelve una función, un reducer

Combinación de reducers

```
import { createStore, combineReducers } from 'redux';
```

```
const initialState = {  
  characters: [],  
  filter: {}  
}
```

```
function characters(state=[], action){  
  ...  
}
```

```
function filter(state={}, action){  
  ...  
}
```

```
const complexReducer = combineReducers({  
  characters: characters,  
  filter: filter  
})
```

```
const store = createStore(complexReducer);  
...
```

Combinación de reducers

```
import { createStore, combineReducers } from 'redux';

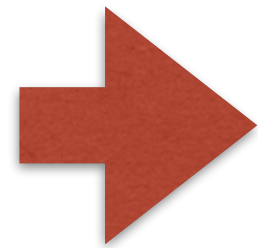
const initialState = {
  characters: [],
  filter: {}
}

function characters(state=[], action){
  ...
}

function filter(state={}, action){
  ...
}

const complexReducer = combineReducers({
  characters,
  filter
})

const store = createStore(complexReducer);
store.dispatch({ type: 'SAVE_CHARACTERS', payload: [ ... ]});
console.log(store.getState().characters);
```



Combinación de reducers

- Combinar reducers se puede hacer a cualquier nivel de profundidad
- Podemos combinar reducers, siendo alguno de ellos a su vez la combinación de otros reducers
- De este modo dividimos la lógica de negocio en funciones sencillas, independientes entre sí
- El reducer de una aplicación completa, es la combinación de los diferentes reducers que definen el estado

Redux - resumen

- Estado global de sólo lectura, guardado en Store
- Acciones que despachamos al Store
- Reducer que transforma el estado de acuerdo a esas acciones
- Suscripción a cambios y recuperación del estado

Conectar React con Redux

- Nivel básico:
 - Re-render con `store.subscribe()`
 - Pasar `dispatch`, `getState` a los componentes

Conectar React con Redux

```
const store = createStore(counter);
```

```
const App = ({ store }) => {  
  const { getState, dispatch } = store,  
        clicks = getState();  
  return (  
    <button onClick={ () => dispatch({ type: 'INCREMENT' }) }>  
      Has hecho click { clicks } veces  
    </button>  
  )  
}
```

```
const update = () => {  
  render(<App store={ store } />, document.getElementById('app'))  
}
```

```
store.subscribe(update);  
//primer render  
update();
```

Conectar React con Redux

- Esto hace exactamente la función **connect** de **react-redux**
- **npm install -S react-redux**

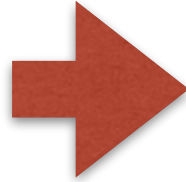
Conectar React con Redux

- **connect**(*mapStateToProps*)
Devuelve una función con la que envolver nuestro componente (connect es una función *curryficada*)
- *mapStateToProps(state)*
Función que recibe el estado y mapea datos de éste a props del componente

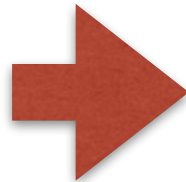
Conectar React con Redux

```
const store = createStore(counter);
```

```
const App = ({ clicks, dispatch }) => {  
  return (  
    <button onClick={() => dispatch({ type: 'INCREMENT' }) }>  
      Has hecho click { clicks } veces  
    </button>  
  )  
}
```



```
const mapStateToProps = state => {  
  return {  
    clicks: state  
  }  
}
```



```
const ConnectedApp = connect(mapStateToProps)(App);  
  
render(<ConnectedApp store={ store } />, document.getElementById('app'))
```

Conectar React con Redux

- ¿Y si queremos pasar el store a componentes hijos, nietos...?
- Pasar constantemente **store={ store }** NO es una buena idea
- **react-redux** nos proporciona un componente **Provider** que se encarga de eso: hacer que el **store** esté disponible para cualquier **connect** que tengamos "más abajo" en el árbol de componentes.

Conectar React con Redux

```
import { connect, Provider } from 'react-redux';

// ...

const ConnectedApp = connect(mapStateToProps)(App);

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

Organización del código

- Hay 3 buenas prácticas que utilizamos con Redux habitualmente:
 - Action types como constantes
 - Action creators (funciones) para construyen las acciones
 - Módulos para agrupar funcionalidad (acciones y reducers que atienden esas acciones)

Action Types

- Para una aplicación no trivial, usar las strings mágicas tipo "CHANGE_FILTER" etc, es una mala idea
- Mejor definimos esas constantes explícitamente y las utilizamos en los reducers y en las acciones

Action Types

```
import { createStore } from 'redux';
```

➔

```
export const SAVE_CHARACTERS = 'SAVE_CHARACTERS'
```

```
function characters(state=[], action) {  
  switch(action.type) {  
    ➔ case SAVE_CHARACTERS:  
      return action.payload;  
    default:  
      return state;  
  }  
}
```

➔

```
const store = createStore(characters);  
store.dispatch({ type: SAVE_CHARACTERS, payload: [ ... ]});
```

Action Creators

- Lo mismo con las acciones en sí, es mejor abstraer la construcción a una función

Action Creators

```
export const ADD_COUNTER = 'ADD_COUNTER';
export const REMOVE_COUNTER = 'REMOVE_COUNTER';
export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';

function addCounter() {
  return {
    type: ADD_COUNTER
  }
}

function removeCounter(index) {
  return {
    type: REMOVE_COUNTER,
    index
  }
}

function incrementCounter(index) {
  return {
    type: INCREMENT_COUNTER,
    index
  }
}

store.dispatch(addCounter());
store.dispatch(incrementCounter(0));
```

Módulos

- Tenemos 3 archivos ahora:
 - Reducer(s)
 - Action creators (funciones)
 - Action types (constantes)

Módulos

- Podemos crear una carpeta por cada área de la funcionalidad, donde guardamos estos 3 archivos.
- Lo que deberá exportar el módulo es:
 - El reducer principal del módulo
 - Los action creators como funciones

Módulos

```
// src/modules/counters/actionTypes.js
```

```
export const ADD_COUNTER = 'ADD_COUNTER';
```

```
export const REMOVE_COUNTER = 'REMOVE_COUNTER';
```

```
export const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
```

Módulos

```
// src/modules/counters/actions.js
```

```
import { ADD_COUNTER, REMOVE_COUNTER, INCREMENT_COUNTER } from '../actionTypes'
```

```
export function addCounter() {  
  return {  
    type: ADD_COUNTER  
  }  
}
```

```
export function removeCounter(index) {  
  return {  
    type: REMOVE_COUNTER,  
    index  
  }  
}
```

```
export function incrementCounter(index) {  
  return {  
    type: INCREMENT_COUNTER,  
    index  
  }  
}
```

Módulos

```
// src/modules/counters/index.js
```

```
import * as actionTypes from './actionTypes';  
export * from './actions';
```

```
function counters(state=[], action) {  
  switch(action.type) {  
    case actionTypes.ADD_COUNTER:  
      ...  
  }  
}
```

```
export default counters;
```

Módulos

```
// app index.js
```

```
import { createStore, combineReducers } from 'redux';  
import counters, { addCounter } from './modules/counters';  
import anotherReducer from './modules/another';
```

```
const appReducer = combineReducers({  
  counters,  
  anotherReducer  
})  
const store = createStore(appReducer);  
  
store.dispatch(addCounter())
```

Ejercicio: Ecommerce con Redux

- Vamos a crear un módulo por cada parte de la aplicación:
 - route
 - catalog
 - cart
 - order (checkout)

Ejercicio: Módulo route

- Sólo hay una cosa que podemos hacer con la ruta: establecerla
- 1 solo action creator: **setRoute**(route)
- 1 reducer muy simple que sólo gestiona la ruta

Crear el store y lanzar la aplicación

- Vamos a extraer la creación del store con los reducers, etc. a un archivo aparte
- Convención: **configureStore**
- Desde ahí importamos nuestros modulos, construimos la Store de Redux y la devolvemos

Ejercicio - Componente index

- Será un contenedor (connect) para recibir la ruta actual y poder decidir qué componente pintar

Ejercicio - Módulo catalog

- Solo gestiona la lista de productos
- Su única acción (de momento) será **saveProducts** que recibe la lista de productos

Ejercicio - Componente catalog

- Muestra la lista de productos usando componentes CatalogItem
- Del estado necesita los productos
- Vamos a cargar aquí los datos "desde el servidor" (despachamos una acción con los datos en **src/data/catalog.js**)

Encapsular el acceso al estado

- *mapStateToProps* tiene una vinculación fuerte con el módulo / reducer de turno: necesitamos conocer la **forma** del estado para extraer los datos
- Con datos básicos no hay problema, pero ¿y si además queremos transformar esos datos para la UI?
- ¿O si queremos entregar un módulo que sea totalmente independiente y reutilizable, como un paquete de npm?

Selectores con reselect

- Podemos definir los **selectores** como parte del módulo. `selector = (state) => datos`
- El caso básico serán funciones que reciben **state** y devuelven el dato
- Pero podemos hacer cosas más avanzadas con **reselect**

npm install -S reselect

selectores básicos

- Selector básico

```
// extraer el carrito del estado  
const cartItems = state => state.cart;  
// extraer el catálogo del estado  
const catalogItems = state => state.catalog;
```

- Lo mismo que estamos escribiendo en
mapStateToProps

selectores complejos

- Selector complejo: se compone de uno o varios selectores simples más una función de transformación o de cálculo.
- El resultado de esa función es lo que devuelve finalmente el selector complejo
- Los creamos con **createSelector** de **reselect**

selectores complejos

```
import { createSelector } from 'reselect';
// Combina el estado del carrito y del catálogo para
// generar un Array de { ...product, quantity }
const cartProducts = createSelector(
  cartItems,
  catalogItems,
  function mapItems(cart, catalog) {
    return Object.keys(cart).map(productId => {
      const quantity = cart[productId],
            product = catalog.find(p => p.id.toString() === productId);
      return {
        ...product,
        ...{ quantity }
      }
    });
  }
);
```

selectores complejos

```
// Reduce sobre los productos del carrito para  
// calcular el total del carrito  
const cartTotal = createSelector(  
  cartProducts,  
  function calculateTotal(products) {  
    return products.reduce((acc, item) => {  
      return acc + (item.quantity * item.price);  
    }, 0).toFixed(2);  
  }  
) ;
```

Uso de selectores con connect()

- Exportamos los selectores desde un módulo
- Los usamos **en lugar de mapStateToProps**, o combinándolo
- Dejamos de mapear "a mano" desde el state, puesto que la forma del estado queda oculta tras el selector.

Ejercicio: módulo y componente Cart

- Definimos acciones: **addToCart**, **removeFromCart**, **changeQuantity**
- Escribimos el reducer
- Creamos los selectores: productos del carrito "desnormalizados" y total del carrito
- Y conectamos todo esto al componente Cart

Action creators como props

- Podemos evitar escribir constantemente:

this.props.dispatch(changeQuantity(...))

- Sería mucho mejor si pudiésemos hacer:

this.props.changeQuantity(...)

mapDispatchToProps

- Resulta que **connect** recibe un segundo parámetro opcional:

`connect(mapStateToProps, mapDispatchToProps)`

- Si es una función, recibe como argumento **dispatch**, igual que `mapStateToProps` recibía **state** y debe devolver un objeto que será entregado como props al componente conectado.

mapDispatchToProps

```
import { changeQuantity } from '../modules/cart';
import { goToCatalog, goToCheckout } from '../modules/route';

//... Cart

const mapDispatchToProps = (dispatch) => {
  return {
    changeQuantity: (id, qty) => dispatch(changeQuantity(id, qty)),
    goToCatalog: () => dispatch(goToCatalog()),
    goToCheckout: () => dispatch(goToCheckout())
  }
}

export default connect(selectors.cartState, mapDispatchToProps)(Cart);
```


mapDispatchToProps

- Si es un objeto formado por Action Creators, *connect* los envolverá con **dispatch** automáticamente por nosotros

```
connect(mapStateToProps, {  
  propName: actionCreator,  
  ...  
}, dispatch)
```

- De modo que podamos crear la acción y despacharla en una sola llamada

mapDispatchToProps

```
import { bindActionCreators } from 'redux';
import { changeQuantity, selectors } from '../modules/cart';
import { goToCatalog, goToCheckout } from '../modules/route';

// ...

Cart.propTypes = {
  items: PropTypes.arrayOf(PropTypes.object).isRequired,
  total: PropTypes.string.isRequired,
  changeQuantity: PropTypes.func.isRequired,
  goToCatalog: PropTypes.func.isRequired,
  goToCheckout: PropTypes.func.isRequired
}

const mapDispatchToProps = {
  changeQuantity,
  goToCatalog,
  goToCheckout
}

export default connect(selectors.cartState, mapDispatchToProps)(Cart);
```

Ejercicio - terminar Cart

- Usando *mapDispatchToProps*, vamos a terminar las interacciones del carrito.
- Queremos modificar las cantidades de un producto en el carrito, eliminarlo del carrito, y volver al catálogo.

Redux middleware

- Hasta ahora Redux es simple pero potente para muchos casos de uso
- Sin embargo, sólo funciona de forma **síncrona**
- Cada llamada a **dispatch** ejecuta todos los reducers de forma inmediata, en el momento de la llamada

Redux middleware

- Redux proporciona un punto de extensión justo cuando se despacha una acción y **antes de que llegue** al reducer: middleware
- Un middleware puede atender ciertos "tipos" de acciones, o dejarlas pasar a otros middlewares, hasta que finalmente llegan a los reducers de forma habitual

Redux middleware

- El más utilizado: **redux-thunk**

npm install -S redux-thunk

- Permite despachar **funciones** en lugar de objetos como acciones
- Esas funciones reciben a su vez **dispatch** (y **getState**), y serán ejecutadas por el middleware, por lo que pueden despachar varias acciones "simples" desde un único action creator

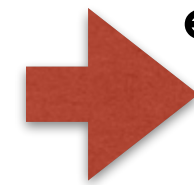
Redux middleware

```
import { get } from '../lib/api';

export const SAVE_CATALOG = 'SAVE_CATALOG';
export function saveCatalog(data) {
  return {
    type: SAVE_CATALOG,
    payload: data
  }
}

export const LOAD_CATALOG = 'LOAD_CATALOG';
export function fetchProducts() {
  return (dispatch, getState) => {
    //para spinner!
    dispatch({
      type: LOAD_CATALOG
    });

    get('/products.json')
      .then(products => {
        dispatch(saveCatalog(products));
      });
  }
}
```



applyMiddleware

- Para que esto funcione, tenemos que inyectar el middleware durante la creación del store, para que "intercepte" la función **dispatch**
- **applyMiddleware**(middleware, [middleware, ...])
- Permite aplicar varios middlewares simultáneamente, y lo usamos con **createStore**

applyMiddleware

```
// src/configureStore.js
```

```
import { createStore, combineReducers, applyMiddleware } from 'redux';
import catalog from './modules/catalog';
import cart from './modules/cart';
import order from './modules/order';
import route from './modules/route';
import thunkMiddleware from 'redux-thunk';

export default function configureStore() {
  const appReducer = combineReducers({
    catalog,
    cart,
    order,
    route
  });

  return createStore(appReducer, applyMiddleware(thunkMiddleware));
}
```

Escribir middleware

- Escribir nuestro propio middleware es muy sencillo
- Tenemos que escribir una función que devuelve una función que devuelve una función.
- La primera recibe el store
- La segunda recibe **next** (el siguiente middleware o el store estándar)
- La tercera recibe **action** en cada dispatch

Escribir middleware



logger middleware

```
const loggerMiddleware = store => next => action => {  
  //aquí tenemos acceso a todo  
  console.log('Acción', action);  
  next(action);  
  console.log('Estado', store.getState());  
}  
  
export default loggerMiddleware;
```

Delay middleware

```
const delayMiddleware = store => next => action => {  
  //retrasa todas las acciones 200 ms  
  setTimeout(() => next(action), 200);  
}  
  
export default delayMiddleware;
```

Crash reporter middleware

```
/**
 * Sends crash reports as state is updated and listeners are notified.
 */
const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception in the reducer!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
}
```

thunk middleware

```
const thunk = store => next => action =>  
  typeof action === 'function' ?  
    action(store.dispatch, store.getState) :  
    next(action)
```

¿Cuándo necesitamos un middleware?

- En general, cuando el mecanismo de despachar acciones básicas se nos queda corto
- O si queremos simplificar los action creators, eliminando código repetitivo
- Solo tenemos que "marcar" o "identificar" las acciones que queremos procesar
- Y actuar sobre ellas en el middleware

Ejercicio - módulo Order

- La pantalla de Checkout tiene un formulario con validación
- Es un gran ejemplo donde necesitamos **redux-thunk**: al pulsar Enviar, tenemos que validar los datos
- Si es correcto, guardamos los detalles y navegamos a la última pantalla de agradecimiento
- Si no, guardaremos los errores en el state y nos quedamos en la misma página

Ejercicio - módulo Order

- ¿Formularios con Redux?
- Si no queremos usar estado interno, tendremos que guardar los datos "parciales" en el Store
- Existen librerías para gestionar formularios, las más conocidas: **redux-form** y **react-redux-form**
- Vamos a implementarlo a mano nosotros

Ejercicio - componente Checkout

- Tendremos que conectar los **onChange** de los campos con el reducer
- Mediante un action creator que guarde el valor de un campo
- Y recuperar del Store los valores de todos los campos en **connect**, para pintarlo

Ejercicio - refactor

- Ahora que sabemos escribir thunks, podemos hacer una carga de datos en el Catálogo más inteligente
- Extraemos toda la lógica a un único action creator: **fetchProducts**, que, **sólo si no están ya cargados los productos**, simulará la llamada a la API y despachará la acción de guardar los datos
- Usamos este nuevo fetchProducts en el componente Catalog

Ejercicio - Pantalla final

- Para terminar nuestra aplicación, nos falta implementar el último componente: ThankYou
- Tiene que mostrar los datos del pedido recién validado (nombre, dirección)