Redux 

“A predictable state container for JavaScript apps”

Información

- Documentación: https://redux.js.org
- Creada por **Dan Abramov**
  - Twitter: https://twitter.com/dan_abramov
  - Github: https://github.com/gaearon
  - Blog: https://overreacted.io
  - Egghead: https://egghead.io/instructors/dan-abramov

Introducción

- Librería javascript para el manejo del ***estado*** de aplicaciones
- **Objetivos**
  - Aplicaciones *predecibles* y *testeables*
  - Estado *centralizado*: persistencia, undo/redo…
  - *Debuggable*: **Redux DevTools**
- Librería javascript
  - `npm install -- save redux`
  - Funciona con cualquier librería de vistas (React, Angular, Vue), incluso en back
  - Gran comunidad: addons, librerias, recursos…

Tres principios

- Fuente única de la verdad
  El estado de la aplicación es almacenado dentro de un único *store*

- El estado es de solo lectura

  El único modo de cambiar el estado es *despachar* una *acción*, un objeto que describe el cambio

- Los cambios se realizan con funciones puras

  Los *reducers* especifican como cambia el estado en respuesta a las acciones

Primer ejemplo

```
// importamos la utilidad createStore de la librería
import { createStore } from 'redux';
// definimos un reducer
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}
// creamos el store que almacena el estado con su API { subscribe, dispatch, getState }
const store = createStore(counter);
// nos subscribimos a los cambios y leemos el estado
store.subscribe(() => console.log(store.getState()));
// despachamos acciones
store.dispatch({ type: 'INCREMENT' }); // 1
store.dispatch({ type: 'INCREMENT' }); // 2
store.dispatch({ type: 'DECREMENT' }); // 1
```

Implementamos createStore

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

  const getState = () => state;

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

  const subscribe = listener => {
    listeners.push(listener);
    return function () {
      listeners = listeners.filter(l => l !== listener);
    }
  };

  dispatch({});
  return { getState, dispatch, subscribe };
}
```

Conceptos básicos

- Acciones
- Estado
- Reducers
- Store
- Flujo de datos
- Uso con React: react-redux

Acciones (I)

- Objetos que representan una *intención* de cambiar el estado
  - Se definen con una propiedad obligatoria **type** que identifica el tipo de acción
  - Pueden contener otros datos que describen completamente la acción (**payload**)
  - Es **buena práctica** definir los distintos types como constantes, incluso ponerlos en un fichero aparte

```
const ADD_TODO = 'ADD_TODO’;
// o importamos desde otro fichero
import { ADD_TODO } from '../actionTypes';

const action = {
  type: ADD_TODO,
  text: 'Build my first Redux app'
};
```

Acciones (II) – Actions creators

- Funciones que crean y devuelven acciones
  - Acciones reusables
  - Fácilmente testeables
  - Mediante *middleware*, pueden devolver funciones, acceder al estado, ejecutar side-effects (asincronía)…

```
import { ADD_TODO } from '../actionTypes';

function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  };
}
```

Estado

- El estado puede tener cualquier forma, es responsabilidad nuestra *modelar* el estado para que se ajuste a la aplicación. Puede ser:
  - Un dato primitivo: Number, String, Boolean
  - Un Array, Object, o cualquier estructura *serializable*

```
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
```

Reducers (I)

- Especifican cómo cambia el estado en respuesta a las acciones enviadas al store
  - `(previousState, action) => newState`
  - Debe ser una **función pura**, por lo que no pueden, bajo ningún concepto:
    - Mutar sus argumentos
    - Ejecutar side-effects (API, BBDD, DOM…)
    - Llamar funciones no puras Date.now(), Math.random()
  - Fácilmente testeables y predecibles, ya que, a igual estado y acción, **siempre** generan el mismo estado

Reducers (II)

```
const initialState = {
  visibilityFilter: VisibilityFilters.SHOW_ALL,
  todos: []
}
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return {...state, visibilityFilter: action.filter }
    case ADD_TODO:
      return {
        ...state,
        todos: [
            ...state.todos,
            {
              text: action.text,
              completed: false
            }
        ]
      };
    case TOGGLE_TODO:
      return {
        ...state,
        todos: state.todos.map(todo =>
          todo.id === action.id
            ? { ...todo, completed: !todo.completed }
            : todo
        )
      };
    default:
      return state;
  }
}
```

Reducers (III) – combineReducers

```
function todos(state = initialState.todos, action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ];
    case TOGGLE_TODO:
      return state.map(todo =>
        todo.id === action.id
          ? { ...todo, completed: !todo.completed }
          : todo
      );
    default:
      return state;
  }
}

function visibilityFilter(state = initialState.visibilityFilter, action
) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state;
  }
}
```

```
function todosApp(state = initialState, action) {
  return {
    todos: todos(state.todos, action),
    visibilityFilter: visibilityFilter(state.visibilityFilter, action)
  };
}
```


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

const todosApp = combineReducers({
  todos,
  visibilityFilter
});
```

Store

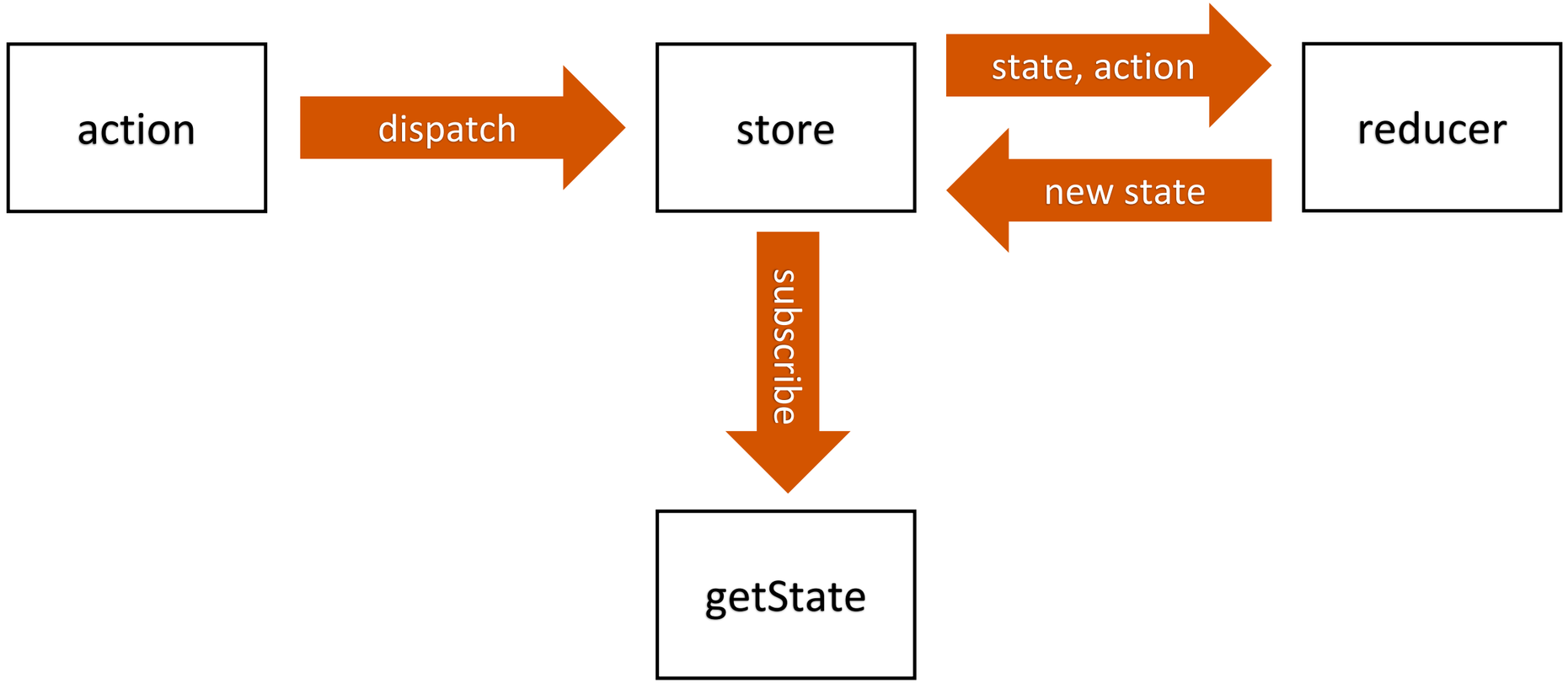
- Objeto *core* de Redux, enlaza *acciones* con *reducers*
- Guarda el estado de la aplicación
- Permite el acceso al estado con **store.getState()**
- Permite despachar acciones con **store.dispatch(action)**
- Registra y mantiene subscripciones con **store.subscribe()**
- Una vez definido el reducer, crear el store es tan sencillo como:

```
import { createStore } from 'redux';
import todoApp from './reducers';
const store = createStore(todoApp, initialState);
```

Flujo de datos (I)

- En Redux el flujo de datos es **unidireccional**
  1. Se despacha una acción: **store.dispatch(action)**
  2. El store llama al reducer pasándole el estado y la acción
  3. El reducer principal combina el resultado de los diferentes reducers, produciendo el nuevo estado
  4. Redux almacena el nuevo estado y llama a los subscriptores para que puedan consultar el valor

Flujo de datos (II)



Uso con React (I) – react-redux

- Libería que proporciona utilidades para enlazar Redux con React
  - `npm install -- save react-redux`
- `<Provider />`
- `connect()`

Uso con React (II) – <Provider />

- Componente que envuelve toda la aplicación React
- Le pasamos el store creado como prop
- Crea un contexto React donde pone el store para que pueda ser accedido por otros componentes

```
import { Provider } from 'react-redux';
import App from './components/App';

const store = createStore(todoApp);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Uso con React (III) – connect

- *Hoc* que crea componentes conectados con el store Redux

```
import { connect } from 'react-redux';
import TodosList from './components/TodosList';

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

- **mapStateToProps**: extrae datos del estado
- **mapDispatchToProps**: crea funciones que despachan acciones
- **connect()(Component)**: pasa la función dispatch como prop

Uso con React (IV) – mapStateToProps

- `mapStateToProps(state, ownProps)`
- Se ejecuta cuando hay un cambio en el estado
- Accede al estado y a las props del componente generado y devuelve un objeto donde cada clave pasa como prop al componente envuelto

```
function mapStateToProps(state, ownProps) {
  return {
    todos: state.todos,
    visibleTodos: getVisibleTodos(state.todos, ownProps.filter)
  };
}
```

Uso con React (V) – selectores

- Dentro de mapStateToProps, en lugar de leer el estado directamente, ejecutamos **funciones** que calculan los datos que necesitan los componentes. Estas funciones son los **selectores.**
- Quitamos lógica de tratamiento de datos en los componentes y la centralizamos en los selectores:
  - Componentes cada vez más “tontos”
  - Reutilización de selectores
  - Los selectores son fáciles de testear
  - Podemos *memoizar* (https://github.com/reduxjs/reselect)
- Consejo: **usa selectores**, incluso aunque pueda parecerte que el cáculo que realizan es muy sencillo.

Uso con React (VI) – mapDispatchToProps

- `mapDispatchToProps(dispatch, ownProps)`
- Accede a dispatch y a las props del componente generado y devuelve un objeto donde cada clave es una función que despacha una acción

```
import { addTodo, setVisibilityFilters } from '../actions';

function mapDispatchToProps(dispatch, ownProps) {
  return {
    addTodo: text => dispatch(addTodos(text)),
    setVisibilityFilter: () => dispatch(setVisibilityFilter(ownProps.filter))
  };
}
// o en modo objeto
const mapDispatchToProps = {
  addTodo,
  toggleTodo
};
```

Conceptos avanzados

- Acciones asíncronas
- Flujo asíncrono
- Middleware
- Uso con React Router

Acciones Asíncronas (I)

- Hasta ahora todas las acciones son síncronas, ¿cómo emitimos acciones **asíncronas**?
- En peticiones AJAX a APIs identificamos varios momentos y en cada uno de ellos podemos emitir una acción síncrona
  - El momento de iniciar la petición

```
{ type: 'FETCH_TODOS_REQUEST' };
```

  - El momento en que la petición finaliza con éxito

```
{ type: 'FETCH_TODOS_SUCCESS', todos: {…} };
```

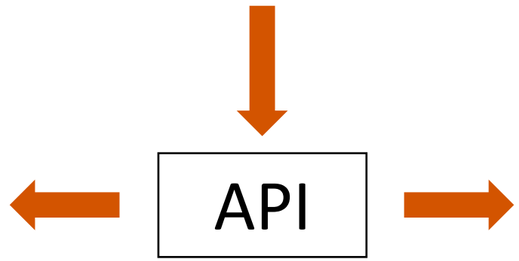
  - El momento en que la petición falla

```
{ type: 'FETCH_TODOS_FAILURE', error: 'Error fetching todos' };
```

Acciones Asíncronas (II) – Actions creators

- Actions creators para las acciones síncronas

```
function fetchTodosRequest() {
  return {
    type: FETCH_TODOS_REQUEST
  }
}
```




```
function fetchTodosFailure(error) {
  return {
    type: FETCH_TODOS_FAILURE,
    error
  }
}
```

```
function fetchTodosSuccess(todos) {
  return {
    type: FETCH_TODOS_SUCCESS,
    todos,
    receivedAt: Date.now()
  }
}
```

- Los reducers tienen que manejar un estado mas complejo, (*isFetching, error*…)

Acciones Asíncronas (III) – Redux Thunk

- ¿Cómo despachamos estas acciones desde el componente?
- Opción 1: Emitimos acciones directamente en el componente
- Opción 2: **Redux Thunk Middleware**
  - Action creator puede devolver una función (thunk)
  - La función es ejecutada por el middleware
  - **Extra**: el thunk tiene acceso al estado (dispatch, getState)

```
function fetchTodos () {
  return function (dispatch, getState) {
    dispatch(fetchTodosRequest());
    return TodoService.getTodos()
      .then(todos => dispatch(fetchTodosSuccess(todos)))
      .catch(error => dispatch(fetchTodosFailure(error)));
  }
}
```

Acciones Asíncronas (IV) – Configurar middleware

- Tenemos que indicar al store que vamos a usar middleware
- **applyMiddleware**: configura el store para poder usar diferentes middlewares

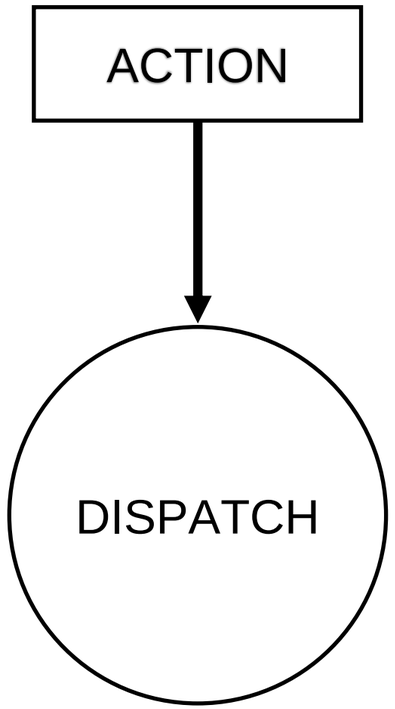
```
import thunkMiddleware from 'redux-thunk';
import { createLogger } from 'redux-logger';
import { createStore, applyMiddleware } from 'redux';

const loggerMiddleware = createLogger();

const store = createStore(
  rootReducer,
  applyMiddleware(
    thunkMiddleware, // podemos emitir funciones
    loggerMiddleware // middleware de log de acciones
  )
);
```

Flujo asíncrono

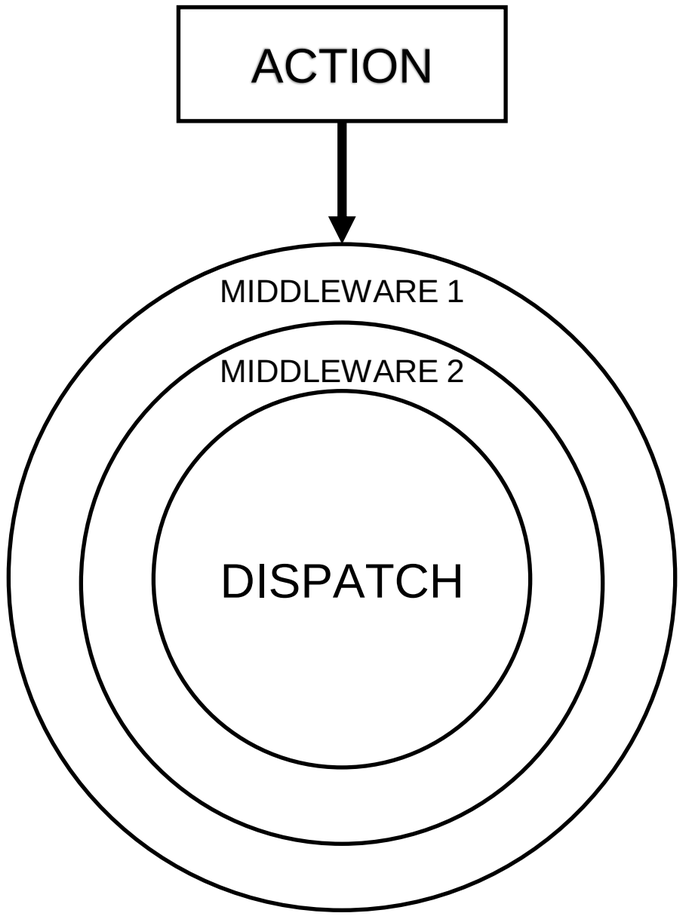
- **Sin middleware** -> flujo sincrono
- **Con middleware** -> flujo asíncrono
  - Cada middleware envuelve a dispatch e intercepta la acción
  - Podemos emitir cosas distintas a acciones síncronas (funciones, promesas, …)
  - Podemos acceder al estado dentro del middleware
  - Cada middleware puede pasar acciones al siguiente middleware
  - El último middleware envuelve al dispatch original de Redux, por lo que debe asegurarse que las acciones que le pase son acciones normales de Redux (objetos con type)

Flujo asíncrono (II)

- Sin middleware



- Con middleware



Middleware

- **Extiende el store**, entre el momento en que se emite una acción y el momento en que la acción alcanza el reducer
- Podemos **encadenar** middleware
- `store => next => action => {}`

```
// Implementación básica
function applyMiddleware(store, middlewares) {
  middlewares = middlewares.slice();
  middlewares.reverse();
  let dispatch = store.dispatch;
  middlewares.forEach(middleware => (dispatch = middleware(store)(dispatch)));
  return Object.assign({}, store, { dispatch });
}
```

Uso con React Router (I)

- **Redux**: fuente de la verdad para **datos**
- **React Router**: fuente de la verdad para URL
- Integración:
  - Conectamos React Router con la App Redux
  - Navegamos con React Router
  - Obtenemos datos de la URL

Uso con React Router (II)

1) Configuración

```
const Root = ({ store }) => (
  <Provider store={store}>
    <Router>
      <Route path="/:filter?" component={App} />
    </Router>
  </Provider>
);
```

2) Navegacion con React Router

```
const FilterLink = ({ filter, children }) => (
  <NavLink
    exact
    to={filter === 'SHOW_ALL' ? '/' : `/${filter}`}
    activeStyle={{
      textDecoration: 'none',
      color: 'black'
    }}
  >
    {children}
  </NavLink>
);
```

3) Leer datos de la URL

```
const App = ({ match: { params } }) => (
  <div>
    <AddTodo />
    <VisibleTodoList filter={params.filter || 'SHOW_ALL'} />
    <Footer />
  </div>
);

const mapStateToProps = (state, ownProps) => ({
  todos: getVisibleTodos(state.todos, ownProps.filter)
});
```

Redux DevTools

- Browser extension: Chrome web store, Firefox add-ons
- http://extension.remotedev.io/