

DISEÑO DE INTERFACES WEB  
TÉCNICO EN DESARROLLO DE APLICACIONES WEB

## Librerías para la creación de interfaces (I)

---

# ÍNDICE

<b>/ 1. Introducción y contextualización práctica</b>	<b>3</b>
<b>/ 2. Introducción Redux</b>	<b>4</b>
2.1. Creación de store con Redux	5
<b>/ 3. Acciones</b>	<b>6</b>
<b>/ 4. Caso práctico 1: “Implementar actions.js”</b>	<b>7</b>
<b>/ 5. Reducers</b>	<b>8</b>
5.1. Introducción y conceptos clave	8
5.2. Implementación y composición de reducers	9
<b>/ 6. Configuración Redux + React. js</b>	<b>10</b>
<b>/ 7. Acciones asíncronas y middleware</b>	<b>11</b>
<b>/ 8. Redux-thunk y redux-promise</b>	<b>12</b>
<b>/ 9. Caso práctico 2: “Implementa reducers.js”</b>	<b>13</b>
<b>/ 10. Resumen y resolución del caso práctico de la unidad</b>	<b>14</b>
<b>/ 11. Bibliografía</b>	<b>15</b>
<b>/ 12. Webgrafía</b>	<b>15</b>

# OBJETIVOS



*Utilizar y valorar distintas aplicaciones para el diseño de documento web.*

*Utilizar marcos, tablas y capas para presentar la información de manera ordenada.*

*Crear y utilizar plantillas de diseño.*

*Interpretar guías de estilo.*

*Utilizar la librería Redux para el desarrollo de nuevas aplicaciones.*



## / 1. Introducción y contextualización práctica

En este capítulo, estudiaremos una de las librerías desarrolladas para JavaScript que permite agilizar y simplificar el diseño e implementación de aplicaciones e interfaces web. En concreto, analizaremos el funcionamiento de Redux.

Redux es una librería que utiliza la arquitectura de software definida por el patrón de diseño Flux, cuyo objetivo principal es separar la parte de desarrollo visual del estado de los datos, es decir, la interfaz de los datos que son recibidos en una aplicación o al estado concreto de los componentes que se integran en una interfaz.

Para entender mejor el funcionamiento de esta librería, en primer lugar, se muestra cómo instalar y configurar los paquetes necesarios para la creación de nuestras aplicaciones utilizando Redux, asimismo se expone cómo realizar la configuración conjunta Redux + React.js.

Escucha el siguiente audio donde planteamos la contextualización práctica de este tema, encontrarás su resolución en el apartado Resumen y resolución del caso práctico.



Fig. 1. Logotipo Redux.



Audio intro. "Similitudes y diferencias.  
Redux vs Flux"  
<https://bit.ly/39y9Vsa>



## / 2. Introducción Redux

[Redux](#) consiste en una librería JavaScript que utiliza la arquitectura de software definida por el patrón de diseño Flux. El objetivo principal de esta librería es separar la parte de desarrollo visual del estado de los datos. Como se ha adelantado, esto hace referencia a los datos que se reciben en una aplicación o al estado concreto de los componentes que se integran en una interfaz.



Redux es un contenedor predecible del estado de aplicaciones JavaScript.

Te ayuda a escribir aplicaciones que se comportan de manera consistente, corren en distintos ambientes (cliente, servidor y nativo), y son fáciles de probar. Además de eso, provee una gran experiencia de desarrollo, gracias a [edición en vivo combinado con un depurador sobre una línea de tiempo](#).

Puedes usar Redux combinado con [React](#), o cual cualquier otra librería de vistas. Es muy pequeño (2kB) y no tiene dependencias.

Aprende Redux con su creador (en inglés): [Getting Started with Redux](#) (30 vídeos gratuitos)

### Testimonios

"Love what you're doing with Redux" Jing Chen, creador de Flux

"I asked for comments on Redux in FB's internal JS discussion group, and it was universally praised. Really awesome work." [Bill Fisher, author of Flux document](#) Bill Fisher, autor de la documentación de Flux

"It's cool that you are inventing a better Flux by not doing Flux at all." André Staltz, creador de Cycle

Fig. 2. Interfaz. Página web oficial.

El uso de Redux facilitará la implementación y desarrollo de aplicaciones.

Tal y como ocurre con otras librerías, la **instalación** de Redux se realiza por línea de comandos. Previamente, se debe instalar en el equipo el paquete Node.js. El comando **npm** y **npm** son claves para todo tipo de desarrollo a través de **la tecnología en JavaScript, puesto que permiten la instalación de programas y librerías implementadas en lenguaje de programación JavaScript**.

1. Instalación de la versión de Redux. Tras la instalación, puede ser importado en cualquier módulo utilizando **import redux from "redux";**

```
npm i -S redux
```

Código 1. Instalación de Redux.

2. También se aconseja la instalación de los paquetes relativos a la conexión con React y las herramientas de desarrollo.

```
npm i -S react-redux  
npm i -D redux-devtools
```

Código 2. Instalación de los paquetes necesarios de Redux.

Redux pretende simplificar el desarrollo basado en la arquitectura Flux y en la librería React. Para ello, permite almacenar en un único objeto todo el estado de la aplicación, es decir, en un único **store**.



## 2.1. Creación de store con Redux


En el capítulo anterior, pudimos ver que los *stores* son los almacenes de datos de los clientes y quedan unidos a las acciones a través de los disparadores (dispatcher).

En el caso de la arquitectura Flux, existen **múltiples stores**, lo que supone una de las diferencias más características con Redux, puesto que, en el caso de Redux, va a existir un único almacén.



Fig. 3. Modelo store en Flux.

A diferencia de Flux, en Redux, es habitual utilizar un único almacén que se implementa siguiendo una estructura en forma de árbol. Este store va a recoger todos los estados de la aplicación en dicho árbol.



Audio 1. "Funciones de store en Redux"  
<https://bit.ly/33yhLOU>

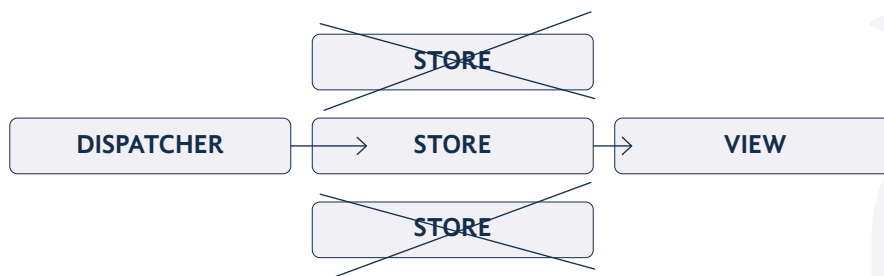



Fig. 4. Modelo store en Redux.

El código implementado para modelar los *stores* se ubica en la ruta `./src/store/index.js`. En Redux, a diferencia de lo expuesto en la arquitectura Flux, se implementa un único *store*, el cual presentará una estructura de árbol en la que cada nuevo dato se inserta como un nuevo nodo:

```
const initialState = {  
  store1: {  
    dato1:...,  
    dato2:...,  
  },  
  store2: {  
    dato1:...,  
    dato2:...,  
  }  
};
```

Código 3. Código de creación de store único en Redux.



## / 3. Acciones

La librería Redux solo permite leer el valor de los estados, pero no modificarlo, se necesita usar **acciones** para ello. Las acciones son objetos JavaScript cuya sintaxis se muestra a continuación. El código implementado para modelar los *stores* se ubica en la ruta `./src/store/nombrecomponente/actions.js`.

```
const ACCION= 'ACCION'
{
  type: ACCION,
  dato1: dato1
}
```

Código 4. Creación acción.

La implementación de estos objetos requiere de los siguientes atributos:

- Al menos, un atributo de **type** para indicar qué acción se va a llevar a cabo. Se aconseja que los tipos se definan como *String* constantes, tal y como se muestra en el caso anterior.
- Un atributo **payload** si el proceso de modificación implica que la acción lleve asociados los datos del cambio. En el código anterior, sería el `dato1`.

Por lo tanto, las acciones son objetos que permiten el envío de los datos entre la aplicación y el *store*, y para ello se utiliza la siguiente instrucción:

```
store.dispatch()
```

Código 5. Lanzamiento de acciones a la store.

Los **creadores de acciones** son funciones que permiten la creación de acciones, lo que resulta bastante aconsejable en términos de desarrollo. En la siguiente estructura de ejemplo, se crea una nueva función de tipo `ACCION` que utiliza como dato de creación el valor que `nombreFunción` ha recibido como parámetro.

```
function nombreFuncion (parámetro) {
  return {
    type: ACCION,
    parámetro
  }
}
```

Código 6. Creador de acciones.

Por lo tanto, en este punto, ya llevaríamos creado el objeto que modela la acción y el creador de acciones. Para crear la acción, solo restaría invocar a la función `dispatch` encargada del envío:

```
dispatch(nombreFuncion(parámetro))
```

Código 7. Ejecución creación de función.



## / 4. Caso práctico 1: “Implementar actions.js”

**Planteamiento:** Se quiere implementar el fichero actions.js completo, el cual debe contener las acciones que se indican en el siguiente listado. Recuerda que el modelado debe ser tal que puedan ser utilizados por el creador de acciones visto en el apartado anterior.

- ADD\_TODO
- COMPLETE\_TODO
- SET\_VISIBILITY\_FILTER

**Nudo:** Para la creación de cada una de las acciones, va a ser necesario desarrollar una función para cada caso. Estas contienen siempre un atributo *type* de tipo *string* y, de forma opcional, un atributo *payload* si se requiere la recepción de datos.

**Desenlace:** El código solución se muestra a continuación. En este caso, en todas las funciones se han utilizado constantes para almacenar el *string* que define el tipo de cada función. De esta forma, resultarán más accesibles para posibles cambios en el código de desarrollo.

```
/*
 * CONSTANTES DE TYPE
 */
export const ADD_TODO = 'ADD_TODO'
export const COMPLETE_TODO = 'COMPLETE_TODO'
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER'
/*
 * CREADORES DE ACCIONES
 */
export function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  }
}
export function completeTodo(index) {
  return {
    type: COMPLETE_TODO,
    index
  }
}
export function setVisibilityFilter(filter) {
  return {
    type: SET_VISIBILITY_FILTER,
    filter
  }
}
```

## / 5. Reducers

### 5.1. Introducción y conceptos clave

Los **reducers** son funciones comunes que reciben como parámetro un estado y el nombre de una acción, y devuelven un nuevo estado. Cuando un *reducer* es invocado, debe devolver siempre un estado, fruto del estado inicial y de la acción pasada por parámetro. La sintaxis de uso es la siguiente:

```
(previousState, action) => newState
```

Código 9. Sintaxis de *reducer*.

Para que una función pueda ser llamada utilizando la forma *reducer*, la función debe ser pura. Una función es **pura** cuando:

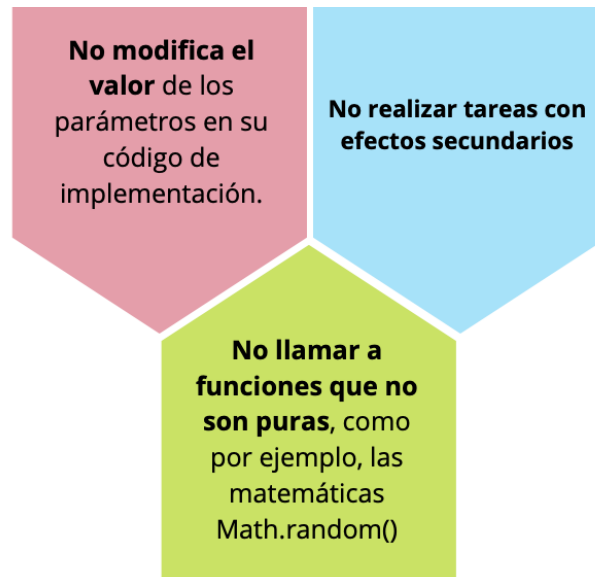


Fig. 5. Condiciones funciones puras.

En el siguiente ejemplo, recogido en la documentación oficial de Redux para *reducer*, se indica cómo se debe escribir una función para que esta sea *reducer*, y, por lo tanto, sea una función pura.

1. En primer lugar, se importa la **constante SET\_VISIBILITY\_FILTER** y se importa la **action VisibilityFilters**.

```
import {  
  SET_VISIBILITY_FILTER,  
  VisibilityFilters  
} from './actions'
```

Código 10. Valores importados.

2. En segundo lugar, se define la función **reducer**. En este caso, recibe el nombre de **todoApp**, la cual recibe dos parámetros: el **estado inicial** y la **acción** que ha sido lanzada con **dispatcher**, es decir, **la acción que ha sido disparada**.

```
function todoApp (state = initialState, action) {
```

Código 11. Definición de la función *todoApp* + parámetros entrada.





## 5.2. Implementación y composición de reducers

### a) Implementación de la función

Para la implementación de la función, se utiliza un **switch** que evalúa todos los valores posibles. En este caso, solo se ha implementado el case que evalúa el valor SET\_VISIBILITY\_FILTER.

Como se ve en el siguiente fragmento, al entrar en el case, se devuelve un objeto nuevo, para lo que se utiliza Object.assign(). Esta función va a crear un nuevo objeto a partir del estado inicial, modificando el estado inicial a través de la acción visibilityFilter.

```
switch (action.type){  
  case SET_VISIBILITY_FILTER:  
    return Object.assign({},state,{  
      visibilityFilter: action.filter  
    })  
  default:  
    return state  
}
```

Código 12. Código evaluado de la acción.

Finalmente, siempre se va a devolver un valor por defecto. Este valor será el anterior *state*. Por ejemplo, en este código, ante la llegada de cualquier acción desconocida, no se realiza ningún cambio en el estado para la acción, y se devuelven el valor del estado anterior.

```
default:  
  return state  
}
```

Código 13. Código default. Implementar siempre.

### b) Composición de reducers

Como veíamos al inicio de este capítulo, la librería Redux utiliza un solo *store*, pero es posible implementar una división que resultará realmente útil en aplicaciones muy grandes. Estamos hablando de la composición de **reducers**. Para ello, en primer lugar, se crea una *store* de la siguiente forma:

```
import { createStore } from 'redux'  
import reducer from './reducers/expenses'  
export default createStore(reducer)
```

Código 14. Creación base store.

Ahora, desde el fichero principal de la aplicación `./src/index.js`, se accede al estado de la aplicación almacenado en la *store* desde la función *render* de la siguiente forma:

```
const render = () => {  
  const state = store.getState()  
  ReactDOM.render(<App {...state} />, document.getElementById('root'))  
}  
render()
```

Código 15. Llamamiento estado de aplicación en store.



## / 6. Configuración Redux + React. js

Tal y como se ha visto hasta ahora, Redux nos va a permitir intercambiar los estados de una aplicación a través de los diferentes componentes. Para realizar un proyecto con Redux y React, se van a seguir los siguientes pasos:

1. En primer lugar, se crea una aplicación React utilizando la siguiente instrucción. Para el nombre del proyecto, no se pueden utilizar letras en mayúsculas.

- Incorrecto

```
npx create-react-app nombreProyecto
```

- Correcto

```
npx create-react-app nombre-proyecto
```

Success! Created nombre-proyecto at /Users/diana/Desktop/pruebas/nombre-proyecto  
Inside that directory, you can run several commands:

```
npm start
  Starts the development server.

npm run build
  Bundles the app into static files for production.

npm test
  Starts the test runner.

npm run eject
  Removes this tool and copies build dependencies, configuration files
  and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

cd nombre-proyecto
npm start
```

Happy hacking!

Fig. 6. Salida terminal instalación exitosa.

2. A continuación, se realiza la **instalación de los módulos de Redux, redux y react-redux**.

```
npm i redux
```

```
npm i react-redux
```

Tras la configuración, será necesario revisar la estructura de carpetas creada e implementar todos los ficheros necesarios. A lo largo de este capítulo, hemos indicado algunas de las rutas recomendadas para la colocación de los diferentes archivos de desarrollo relativos a los *store* o *actions*, entre otros.

Se va a realizar la **creación de los ficheros de componentes** que recogen los datos del usuario. Estos componentes se colocan en la carpeta de desarrollo **/src**, en concreto, dentro de una nueva carpeta llamada, por ejemplo, **/componentes**. Dentro de esta última ruta, **se creará una nueva carpeta para cada uno de los componentes que se vayan a implementar**. Y, en su interior, se colocan los ficheros que definen la interfaz visual, la lógica del componente y también la hoja de estilo asociada.

En cada una de las carpetas creadas, se coloca un nuevo fichero de tipo JavaScript, es decir, con extensión **.js**. En estos archivos, se implementa el código necesario para cada uno de los componentes, utilizando React.js.



Vídeo 1. "Creación de un proyecto Redux + React.js"  
<https://bit.ly/3fXtg6Y>





## / 7. Acciones asíncronas y middleware

### a) Middleware

Middleware es el código que se ejecuta entre dos framework, en particular, entre el que recibe la petición y el que modela la respuesta a dicha petición. Se trata de funciones de orden superior que van a permitir añadir una nueva funcionalidad a Redux, en concreto, al almacén de Redux: **store**.

Redux incorpora su propio middleware que, tal y como se indica desde el sitio oficial, “proporciona un punto de extensión por terceros entre el envío de una acción y el momento en que alcanza el reductor (*reducer*)”.

Las funciones middleware reciben como parámetro el *store* y devuelve una nueva función que tiene como parámetro una llamada a *next*, es decir, al siguiente middleware que va a ser utilizado. La estructura básica se muestra a continuación, donde los elementos principales son:

- **store**: almacén para el estado de la aplicación accesible a través de la función `getState()`.
- **next**: indica el siguiente middleware al que accede la aplicación en el flujo de ejecución.
- **action**: define la acción que ha sido lanzada (disparada con `dispatcher`).

```
const pruebaMiddleware = store => next => action => {  
  console.log('Esto es un middleware de prueba');  
}
```

Código 16. Diseño con middleware.

### b) Acciones asíncronas

La implementación que se ha visto en los apartados anteriores para Redux utiliza acciones síncronas. Ahora bien, en la actualidad, está cada vez presente la implementación de formas asíncronas, por ejemplo, utilizando AJAX en JavaScript.

La actualización del contenido de una página web utilizando JavaScript implica que para que la información mostrada en un sitio web se modifique, debe cambiar la respuesta del servidor, es decir, será necesario estar realizando peticiones constantes al servidor para actualizar la información y, con ello, que se modifique la interfaz de presentación. **Para evitar tener que estar realizando peticiones constantes, aparece AJAX.**

En el caso de Redux, será posible utilizar las acciones de forma asíncrona, para lo que es posible utilizar el **middleware `redux-thunk`** que permite utilizar acciones asíncronas.



Vídeo 2. “Ampliación de acciones asíncronas con middleware”

<https://bit.ly/37ssXNE>





## / 8. Redux-thunk y redux-promise

Los **middleware redux-thunk y redux-promise** son utilizados en Redux para implementar las acciones de forma asíncrona.

Definimos cuál es el objetivo principal de cada uno de estos middlewares a continuación:

- A través de **redux-thunk**, se lanza una determinada acción, la cual queda a la espera de recibir una respuesta, es decir, recibe la promesa (promise) de una respuesta.
- Como complemento a este middleware, se utiliza **redux-promise**, el cual permite lanzar las “promesas” que desde el **store** quedarán a la espera de ser completadas.

La instalación de estos middleware se realiza por línea de comandos:

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

*Código 17. Instalación middleware.*

Desde el fichero **store.js**, se implementa el siguiente código al que se le aplican los middlewares descritos:

```
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  
  )  
);
```

*Código 18. Llamamiento estado de aplicación en store.*

En el código anterior (código 18), han quedado definidos en el **store** de la aplicación los **middleware thunk y promise**, lo que va a permitir ejecutar todo tipo de acciones, las habituales síncronas y las asíncronas, ya que al recibir una función que devuelve una promesa (promise), esta será tratada en base a lo expuesto en este apartado.



## / 9. Caso práctico 2: “Implementa reducers.js”

**Planteamiento:** Se quiere implementar el fichero **reducers.js** completo, el cual debe contener las acciones que se indican en siguiente listado.

- Función 1 para evaluar el caso SET\_VISIBILITY\_FILTER
- Función 2 para evaluar las funciones ADD\_TODO y COMPLETE\_TODO

**Nudo:** Para la creación de cada una de las acciones, va a ser necesario desarrollar dos funciones. En ambos casos, será necesario utilizar una estructura de control que evalúa el tipo de acción que se está lanzando.

**Desenlace:** El código solución se muestra a continuación.

```
function visibilityFilter(state = SHOW_ALL, action) {  
  switch (action.type) {  
    case SET_VISIBILITY_FILTER:  
      return action.filter  
    default:  
      return state  
  }  
}  
  
function todos(state = [], action) {  
  switch (action.type) {  
    case ADD_TODO:  
      return [  
        state,  
        {  
          text: action.text,  
          completed: false  
        }  
      ]  
    case COMPLETE_TODO:  
      return state.map((todo, index) => {  
        if (index === action.index) {  
          return Object.assign({}, todo, {  
            completed: true  
          })  
        }  
        return todo  
      })  
    default:  
      return state  
  }  
}  
  
const todoApp = combineReducers({  
  visibilityFilter,  
  todos  
})  
  
export default todoApp
```



## / 10. Resumen y resolución del caso práctico de la unidad

En esta unidad, hemos comprobado que **Redux** es una librería que utiliza **la arquitectura de software definida por el patrón de diseño Flux**, cuyo objetivo principal es separar la parte de desarrollo visual del estado de los datos. En el caso de la arquitectura Flux, existen **múltiples store**, lo que supone una de las diferencias más características con Redux, puesto que va a existir un único almacén.

A diferencia de Flux, en Redux, es habitual utilizar un único almacén que se implementa siguiendo una estructura en forma de árbol. Este *store* va a recoger todos los estados de la aplicación en dicho árbol.

La librería Redux solo permite leer el valor de los estados, pero no modificarlo, para ello se necesita usar **acciones**.

También, hemos aprendido que el código para modelar los *stores* se centra en la implementación de estos objetos y requiere de los atributos: **type y payload**. El primero indica la acción que se va a llevar a cabo y el segundo es de carácter opcional, e indica los datos asociados a la acción.

Finalmente, recordemos otro de los elementos clave que hemos visto en este capítulo: los **reducers**. Son funciones que reciben como parámetro un estado y el nombre de una acción, y devuelven un nuevo estado. Cuando un reducer es invocado, debe devolver siempre un estado, fruto del estado inicial y de la acción pasada por parámetro.

### Resolución del caso práctico inicial

Como se ha visto a lo largo del capítulo, Redux está basado en la arquitectura Flux, lo que supone ciertos puntos de similitud como el diseño en capas, o el uso de acciones para acceder y modificar el valor de los estados.

Ahora bien, aunque está basado en la arquitectura Flux, Redux incorpora ciertas diferencias en cuanto a su implementación. ¿Qué diferencias crees que existirán entre Flux y Redux?

En la siguiente tabla, se recogen las tres claves principales que identifican el uso de la librería Redux:

Redux almacena todo el estado de la aplicación en un solo árbol, en un único store “una única fuente de verdad”
El estado de una aplicación solo es de lectura. Para alterar su contenido, se debe realizar a través de acciones específicas
Se utilizan las funciones puras, reducers, que permiten la modificación de los estados

Tabla 1. Diferencias Redux vs Flux

Redux almacena todo el estado de la aplicación en un solo árbol, en un único *store* “una única fuente de verdad”

El estado de una aplicación solo es de lectura. Para alterar su contenido, se debe realizar a través de acciones específicas

Se utilizan las funciones puras, reducers, que permiten la modificación de los estados



## / 11. Bibliografía

García-Miguel, D. (2019). *Diseño de Interfaces Web* (1.a ed.). Madrid, España: Síntesis.

## / 12. Webgrafía

Redux. Recuperado de:

<https://es.redux.js.org/docs/avanzado/middleware.html>

MEDAC