UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Programador Web Inicial
Front End Developer

REDUX

Presentación:

Como los requisitos en aplicaciones JavaScript de una sola página se están volviendo cada vez más complicados, **nuestro código, más que nunca, debe manejar el estado**. Este estado puede incluir respuestas del servidor y datos cacheados, así como datos creados localmente que todavía no fueron guardados en el servidor. El estado de las UI también se volvió más complejo, al necesitar mantener la ruta activa, el tab seleccionado, si mostrar o no un spinner, si deben mostrarse los controles de paginación o no.

Controlar ese cambiante estado es difícil. Si un modelo puede actualizar otro modelo, entonces una vista puede actualizar un modelo, el cual actualiza otro modelo, y esto causa que otra vista se actualice. En cierto punto, ya no se entiende que está pasando en la aplicación ya que **perdiste control sobre el cuándo, el por qué y el cómo de su estado**. Cuando un sistema es opaco y no determinista, es difícil reproducir errores o agregar nuevas características.

Como si no fuera suficientemente malo, considera que **los nuevos requisitos son comunes en el desarrollo front-end**. Como desarrolladores, se espera que manejemos actualizaciones, renderizado en el servidor, obtener datos antes de realizar cambios de rutas, y más cosas.

Esta complejidad es difícil de manejar debido a que **estamos mezclando dos conceptos** que son difícil de entender para la mente humana: **mutaciøn y asincronicidad**. Librerías como React tratan de resolver este problema en la capa de la vista removiendo tanto la asincronicidad como la manipulación directa del DOM. De todas formas, controlar el estado de tus datos es todavía tu responsabilidad. Acá es donde entra Redux.

Siguiendo los pasos de Flux, CQRS y Event Sourcing, **Redux intenta hacer predecibles las mutaciones del estado** imponiendo ciertas restricciones en cómo y cuándo pueden realizarse las actualizaciones. Estas restricciones se reflejan en los tres principios de Redux.

Conceptos básicos

Redux de por sí es muy simple.

Imagine que el estado de su aplicación se describe como un simple objeto. Por ejemplo, el estado de una aplicación de tareas (TODO List) puede tener el siguiente aspecto:

```
{
  todos: [{
    text: 'Comer',
    completed: true
  }, {
    text: 'Hacer ejercicio',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

Este objeto es como un "modelo" excepto que no hay *setters*. Esto es así para que diferentes partes del código no puedan cambiar el estado arbitrariamente, causando errores difíciles de reproducir.

Para cambiar algo en el estado, es necesario enviar una acción. Una acción es un simple objeto en JavaScript (observe cómo no introducimos ninguna magia) que describe lo que sucedió. A continuación mostramos algunos ejemplos de acciones:

```
{ type: 'ADD_TODO', text: 'Ir a nadar a la piscina' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

Hacer valer que cada cambio sea descrito como una acción nos permite tener una claro entendimiento de lo que está pasando en la aplicación. Si algo cambió, sabemos por qué cambió. Las acciones son como migas de pan (el rastro) de lo que ha sucedido.

Finalmente, para juntar el estado y las acciones entre sí, escribimos una función llamada reductor (reducer). Una vez más, nada de magia sobre él asunto, es sólo una función que toma el estado y la acción como argumentos y devuelve el siguiente estado de la aplicación. Sería difícil escribir tal función para una aplicación grande, por lo que escribimos funciones más pequeñas que gestionan partes del estado:

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  if (action.type === 'SET_VISIBILITY_FILTER') {
    return action.filter;
  } else {
    return state;
  }
}

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

Y escribimos otro reductor que gestiona el estado completo de nuestra aplicación llamando a esos dos reductores por sus respectivas *state keys*:

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

Esto es básicamente toda la idea de Redux. Tenga en cuenta que no hemos utilizado ninguna API de Redux. Ya se incluyen algunas utilidades para facilitar este patrón, pero la idea principal es que usted describe cómo su estado se actualiza con el tiempo en respuesta a los objetos de acción, y el 90% del código que se escribe es simplemente JavaScript, sin uso de Redux en sí mismo, sus APIs, o cualquier magia.

Tres Principios

Redux puede ser descrito en tres principios fundamentales:

Única fuente de la verdad

El estado de toda tu aplicación está almacenado en un árbol guardado en un único store.

Esto hace fácil crear aplicaciones universales, ya que el estado en tu servidor puede ser serializado y enviado al cliente sin ningún esfuerzo extra. Como un único árbol de estado también hace más fácil depurar una aplicación; te permite también mantener el estado de la aplicación en desarrollo, para un ciclo de desarrollo más veloz.

Algunas funcionalidades que históricamente han sido difíciles de implementar - como Deshacer/Rehacer, por ejemplo - se vuelven triviales si todo tu estado se guarda en un solo árbol.

```
console.log(store.getState())
/* Imprime
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Considerar usar Redux',
      completed: true
    },
    {
      text: 'Mantener todo el estado en un solo árbol',
      completed: false
    }
  ]
}
*/
```

El estado es de solo lectura

La única forma de modificar el estado es emitiendo una acción, un objeto describiendo qué ocurrió.

Esto te asegura que ni tu vista ni callbacks de red van a modificar el estado directamente. En vez de eso, expresan un intento de modificar el estado. Ya que todas las modificaciones están centralizadas y suceden en un orden estricto, no hay que preocuparse por una carrera entre las acciones. y como las acciones son objetos planos, pueden ser registrados,

serializados y almacenados para volver a ejecutarlos por cuestiones de depuración y pruebas.

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
});

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
});
```

Los cambios se realizan con funciones puras

Para especificar cómo el árbol de estado es transformado por las acciones, se utilizan reducers puros.

Los reducers son funciones puras que toman el estado anterior y una acción, y devuelven un nuevo estado.

Recuerda devolver un nuevo objeto de estado en vez de modificar el anterior.

Puedes empezar con un único reducer, y mientras tu aplicación crece, dividirlo en varios reducers pequeños que manejan partes específicas del árbol de estado.

Ya que los reducers son funciones puras, puedes controlar el orden en que se ejecutan, pasarle datos adicionales, o incluso hacer reducers reusables para tareas comunes como paginación.

```
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.slice(0, action.index),
        Object.assign({}, state[action.index], {
          completed: true
        }),
        ...state.slice(action.index + 1),
      ];
    default:
      return state;
  }
}

import { combineReducers, createStore } from 'redux';
let reducer = combineReducers({ visibilityFilter, todos });
let store = createStore(reducer);
```

Herencia

Redux tiene una herencia mixta. Es similar a ciertos patrones y tecnologías, pero también es diferente en varias formas importantes. Vamos a ver las similitudes y diferencias debajo.

Flux

¿Puede Redux ser considerado una implementación de Flux? Sí y no.

(No te preocupes, los creadores de Flux lo aprueban, si es lo que necesitas saber.)

Redux se inspiró en muchas de las cualidades importantes de Flux. Como Flux, Redux te hace concentrar la lógica de actualización de modelos en una capa específica de tu aplicación ("stores" en Flux, "reducers" en Redux). En vez de dejar al código de la aplicación modificar los datos directamente, ambos te hacen describir cada mutación como objetos planos llamados "acciones".

A diferencia de Flux, **en Redux no existe el concepto de Dispatcher**. Esto es porque se basa en funciones puras en vez de emisores de eventos, y las funciones puras son fáciles de componer y no necesitan entidades adicionales para controlarlas. Dependiendo de como veas Flux, puedes ver esto tanto como una desviación o un detalle de implementación. Flux

siempre fue descripto como (state, action) => state. En ese sentido, Redux es una verdadera arquitectura Flux, pero más simple gracias a las funciones puras.

Otra diferencia importante con Flux es que **Redux asume que nunca modificas los datos**. Puede usar objetos planos o arrays para tu estado y está perfecto, pero modificarlos dentro de los reducers no es recomendable. Siempre deberías devolver un nuevo objeto, lo cual es simple con la propuesta del operador spread, o con librerías como Immutable.

Mientras que es técnicamente *posible* escribir reducers impuros que modifican los datos en algunos por razones de rendimiento, desalentamos activamente hacer eso. Características de desarrollo como time travel, registrar/rehacer, o hot reloading pueden romperse. Además características como inmutabilidad no parecen causar problemas de rendimiento en aplicaciones reales, ya que, como Om demostró, incluso si pierdes la posibilidad de usar asignación en objetos, todavía ganas por evitar re-renders y recalculos caros, debido a que sabes exactamente qué cambió gracias a los reducers puros.

Elm

Elm es un lenguaje de programación funcional inspirado por Haskell y creado por Evan Czaplicki. Utiliza una arquitectura 'modelo vista actualizador", donde el actualizador tiene el siguiente formato: (action, state) => state. Los "actualizadores" de Elm sirven para el mismo propósito que los reducers en Redux.

A diferencia de Redux, Elm es un lenguaje, así que puede beneficiarse de muchas cosas como pureza forzada, tipado estático, inmutabilidad, y coincidencia de patrones (usando la expresión case). Incluso si no planeas usar Elm, deberías leer sobre la arquitectura de Elm, y jugar con eso. Hay un interesante playground de librerías de JavaScript que implementa ideas similares. ¡Debemos mirar ahí por inspiración en Redux! Una forma de acercarnos al tipado estático de Elm es usando una solución de tipado gradual como Flow.

Immutable

Immutable es una librería de JavaScript que implementa estructuras de datos persistentes. Es una API JavaScript idiomática y performante.

Immutable y la mayoría de las librerías similares son ortogonales a Redux. ¡Siéntete libre de usarlos juntos!

Redux no se preocupa por si guardas el estado en objetos planos, objetos de Immutable o cualquier otra cosa. Probablemente quieras un mecanismo de (de)serialización para crear aplicaciones universales y rehidratar su estado desde el

servidor, pero aparte de eso, puedes usar cualquier librería de almacenamiento de datos *mientras soporte inmutabilidad*. Por ejemplo, no tiene sentido usar Backbone para tu estado de Redux, ya que estos son mutables.

Ten en cuenta que que incluso si tu librería inmutable soporta punteros, no deberías usarlos en una aplicación de Redux. Todo el árbol de estado debería ser considerado de solo lectura, y deberías usar Redux para actualizar el estado, y suscribirse a los cambios. Por lo tanto, actualizar mediante recuerdos no tiene sentido en Redux. **Si tu único caso de uso para los punteros es desacoplar el árbol de estado del árbol de UI y gradualmente refinar tus punteros, deberías revisar los selectores mejor.** Los selectores son funciones getter combinables. Mira reselect para una muy buena y concisa implementación de selectores combinables.

Baobab

Baobab es otra popular librería para implementar inmutabilidad para actualizar objetos planos en JavaScript. Aunque puedes usarlo con Redux, hay muy pocos beneficios de usarlos juntos.

La mayoría de las funcionalidades que provee Baobab están relacionadas con actualizar los datos con punteros, pero Redux impone que la única forma de actualizar los datos es despachando acciones. Por lo tanto, resuelven el mismo problema de forma diferente, y no se complementan uno con otro.

A diferencia de Immutable, Baobab todavía no implementa ninguna estructura de datos especialmente eficiente, así que no ganas nada realmente por usarlo junto a Redux. Es más fácil simplemente usar objetos planos en su lugar.

Rx

Reactive Extensions (y su reescritura moderna en proceso) es una forma magnífica de manejar la complejidad de aplicaciones asíncronas. De hecho hay un esfuerzo de crear una librería para controlar la interacción entre humano y computadora como observables independientes.

¿Tiene sentido usar Redux junto con Rx? ¡Seguro! Funcionan genial juntos. Por ejemplo, es fácil exponer el store de Redux como un observable:

```
function toObservable(store) {
  return {
```

```
    subscribe({ onNext }) {
      let dispose = store.subscribe(() => onNext(store.getState()));
      onNext(store.getState());
      return { dispose };
    },
  };
}
```

De forma similar, puedes componer diferentes streams asíncronos para convertirlos en acciones antes de enviarlos al store.dispatch().

La pregunta es: ¿de verdad necesitas Redux si ya usas Rx? Probablemente no. No es difícil reimplementar Redux en Rx. Algunos dicen, que son solo 2 líneas usando el método .scan() de Rx. ¡Y probablemente lo sea!

Si tienes dudas, revisa el código fuente de Redux (no hay mucho ahí), así como su ecosistema (por ejemplo, las herramientas de desarrolladores). Si no te interesa tanto eso y quieres que los datos reactivos simplemente fluyan, probablemente quieras usar algo como Cycle en su lugar, o incluso combinarlos con Redux. ¡Déjanos saber cómo resulta eso!

Ejemplos

Redux es distribuido con algunos ejemplos en su código fuente.

Counter Vanilla

Ejecuta el ejemplo de Counter Vanilla

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/counter-vanilla
open index.html
```

No requiere un sistema de build o un framework de vista y existe para mostrar la API pura de Redux usando ES5.

Counter

Ejecuta el ejemplo de Counter:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/counter
npm install
npm start

open http://localhost:3000/
```

Este es el más básico ejemplo usando Redux junto a React. Por simpleza, vuelve a renderizar el componente de React manualmente cuando el store cambia. En un proyecto real, deberías usar algo con mejor rendimiento como React Redux.

Este ejemplo incluye pruebas.

Todos

Ejecuta el ejemplo de Todos:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/todos
npm install
npm start

open http://localhost:3000/
```

Este es el mejor ejemplo para obtener un conocimiento más profundo de cómo las actualizaciones de estado funcionan en Redux. Muestra como los reducers pueden delegar el manejo de acciones a otros reducers, y como usar React Redux para generar componentes contenedores para tus componentes presentacionales.

Este ejemplo incluye pruebas.

Todos with Undo

Ejecuta el ejemplo de Todos with Retroceder:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/todos-with-undo
npm install
npm start

open http://localhost:3000/
```

Este es una variación del ejemplo anterior. Es prácticamente idéntico, pero además muestra cómo envolver tus reducers con Redux Undo te permite agregar la funcionalidad de Deshacer/Rehacer a tus aplicaciones con una pocas líneas.

TodoMVC

Ejecuta el ejemplo de TodoMVC:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/todomvc
npm install
npm start

open http://localhost:3000/
```

Este es el clásico ejemplo de TodoMVC. Está aquí por razones de comparación, pero cubre los mismos puntos que el ejemplo de Todos.

Este ejemplo incluye pruebas.

Shopping Cart

Ejecuta el ejemplo de Shopping Cart:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/shopping-cart
npm install
```

```
npm start

open http://localhost:3000/
```

Este ejemplo muestra importantes patrones de Redux que se vuelven más importantes mientras tu aplicación crece. En particular, muestra cómo guardar entidades de una forma normalizada usando sus IDs, como componer reducer en varios niveles, y cómo definir selectores junto a los reducers así el conocimiento de la forma del estado está encapsulado. Además muestra cómo registrar cambios con Redux Logger y despachar acciones con el middleware Redux Thunk.

Tree View

Ejecuta el ejemplo de Tree View:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/tree-view
npm install
npm start

open http://localhost:3000/
```

Este ejemplo muestra cómo renderizar una vista de un árbol profundamente anidado y representar su estado de forma normalizada así es fácil actualizarlo mediante reducers. Un buen rendimiento al renderizar al hacer que los componentes contenedores se suscriban solo a los nodos del árbol que renderizan.

Este ejemplo incluye pruebas.

Async

Ejecuta el ejemplo de Async:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/async
npm install
npm start
```

```
open http://localhost:3000/
```

Este ejemplo incluye leer desde un API asíncrono, obtener datos en respuestas a la acciones del usuario, mostrar indicadores de cargando, cachear respuestas e invalidar la cache. Usa el middleware It uses Redux Thunk para encapsular efectos secundarios asíncronos.

Universal

Ejecuta el ejemplo de Universal:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/universal
npm install
npm start

open http://localhost:3000/
```

Esta es la más básica demostración de renderizado en el servidor con Redux y React. Muestra cómo preparar el estado inicial en el servidor y pasarlo al cliente así se inicia desde el estado existente.

Real World

Ejecuta el ejemplo de Real World:

```
git clone https://github.com/reactjs/redux.git

cd redux/examples/real-world
npm install
npm start

open http://localhost:3000/
```

Este es el ejemplo más avanzado. Es grande por diseño. Cubre cómo mantener entidades en una cache normalizada, implementando un middleware personalizado para llamadas a API, renderizando páginas parcialmente cargadas, paginación, cacheo de respuestas, mostrar mensajes de error y ruteo. Adicionalmente, incluye las Redux DevTools.

Básico

Acciones

Primero, vamos a definir algunas acciones.

Las **acciones** son un bloque de información que envía datos desde tu aplicación a tu store. Son la *única* fuente de información para el store. Las envias al store usando store.dispatch().

Aquí hay unas acciones de ejemplo que representan agregar nuevas tareas pendientes:

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

Las acciones son objetos planos de JavaScript. Una acción debe tener una propiedad type que indica el tipo de acción a realizar. Los tipos normalmente son definidos como strings constantes. Una vez que tu aplicación sea suficientemente grande, quizás quieras moverlos a un módulo separado.

```
import { ADD_TODO, REMOVE_TODO } from '../actionTypes'
```

Nota

No necesitas definir tus tipos de acciones constantes en un archivo separado, o incluso definirlas. Para proyectos pequeños, probablemente sea más fácil usar strings directamente para los tipos de acciones. De todas formas, hay algunos beneficios explícitos en declarar constantes en grandes bloques de código. Lee Reduciendo el Boilerplate para más consejos prácticos sobre cómo mantener tu código limpio.

Además del type, el resto de la estructura de los objetos de acciones depende de tí. Si estás interesado, revisa Flux Standard Action para recomendaciones de cómo una acción debe armarse.

Vamos a agregar una acción más para describir un usuario marcando una tarea como completa. Nos referimos a una tarea en particular como su index ya que vamos a almacenarlos en un array. En una aplicación real, es más inteligente generar un ID único cada vez que creamos una nueva.

```
{
  type: COMPLETE_TODO,
  index: 5
}
```

Es una buena idea pasar la menor cantidad de información posible. Por ejemplo, es mejor pasar el index que todo el objeto de tarea.

Por último, vamos a agregar una acción más para cambiar las tareas actualmente visibles.

```
{
  type: SET_VISIBILITY_FILTER,
  filter: SHOW_COMPLETED
}
```

Creadores de acciones

Los **creadores de acciones** son exactamente eso—funciones que crean acciones. Es fácil combinar los términos "acción" con "creador de acción", así que haz lo mejor por usar los términos correctos.

En implementaciones de Flux tradicional, los creadores de acciones ejecutan el despacho cuando son invocadas, algo así:

```
function addTodoWithDispatch(text) {
  const action = {
    type: ADD_TODO,
    text
  }
  dispatch(action)
}
```

En cambio, en Redux los creadores de acciones simplemente regresan una acción:

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

```
}
```

Esto las hace más portables y fáciles de probar. Para efectivamente inicial un despacho, pasa el resultado a la función dispatch():

```
dispatch(addTodo(text))
dispatch(completeTodo(index))
```

Alternativamente, puedes crear un **creador de acciones conectados** que despache automáticamente:

```
const boundAddTodo = (text) => dispatch(addTodo(text))
const boundCompleteTodo = (index) => dispatch(completeTodo(index))
```

Ahora puedes llamarlas directamente:

```
boundAddTodo(text)
boundCompleteTodo(index)
```

La función dispatch() puede ser accedida directamente desde el store como store.dispatch(), pero comúnmente vas a querer usar utilidades como connect() de react-redux. Puedes usar bindActionCreators() para automáticamente conectar muchos creadores de acciones a dispatch().

Los creadores de acciones pueden además ser asíncronos y tener efectos secundarios. Puedes leer más sobre las acciones asíncronas en el tutorial avanzado para aprender cómo manejar respuestas AJAX y combinar creadores de acciones en un flujo de control asíncrono. No salta ahora mismo hasta las acciones asíncronas hasta que completes el tutorial básico, ya que cubre otros conceptos importantes que son prerrequisitos para el tutorial avanzado y las acciones asíncronas.

Código fuente

```
actions.js

/*
 * tipos de acciones
 */

export const ADD_TODO = 'ADD_TODO'
export const COMPLETE_TODO = 'COMPLETE_TODO'
```

```
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER'

/*
 * otras constantes
 */

export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
}

/*
 * 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 }
}
```

Reducers

Las acciones describen que *algo pasó*, pero no especifican cómo cambió el estado de la aplicación en respuesta. Esto es trabajo de los reducers.

Diseñando la forma del estado

En Redux, todo el estado de la aplicación es almacenado en un único objeto. Es una buena idea pensar en su forma antes de escribir código. ¿Cuál es la mínima representación del estado de la aplicación como un objeto?

Para nuestra aplicación de tareas, vamos a querer guardar dos cosas diferentes:

- El filtro de visibilidad actualmente seleccionado;

- La lista actual de tareas.

Algunas veces verás que necesitas almacenar algunos datos, así como el estado de la UI, en el árbol de estado. Esto está bien, pero trata de mantener los datos separados del estado de la UI.

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

Nota sobre relaciones

En aplicaciones más complejas, vas a necesitar tener diferentes entidades que se referencien una a otra. Sugerimos mantener el estado tan normalizado como sea posible, sin nada de anidación. Mantener cada entidad en un objeto con el ID como llave, y usa los IDs para referenciar otras entidades, o para listas. Piensa en el estado de la aplicación como una base de datos. Este enfoque se describe en la documentación de normalizr más detalladamente. Por ejemplo, manteniendo todosById: { id -> todo } y todos: array<id> dentro del estado es mejor para una aplicación real, pero lo vamos a mantener simple para el ejemplo.

Manejando Acciones

Ahora que decidimos cómo se verá nuestro objeto de estado, estamos listos para escribir nuestro reducer. El reducer es una función pura que toma el estado anterior y una acción, y devuelve en nuevo estado.

(previousState, action) => newState

Se llama reducer porque es el tipo de función que pasarías a Array.prototype.reduce(reducer, ?initialValue). Es muy importante que los reducer se mantengan puros. Cosas que **nunca** deberías hacer dentro de un reducer:

- Modificar sus argumentos;

- Realizar tareas con efectos secundarios como llamas a un API o transiciones de rutas.
- Llamar una función no pura, por ejemplo Date.now() o Math.random().

En la guía avanzada vamos a ver como realizar efectos secundarios. Por ahora, solo recuerda que los reducers deben ser puros. **Dados los mismos argumentos, debería calcular y devolver el siguiente estado. Sin sorpresas. Sin efectos secundarios. Sin llamadas a APIs. Sin mutaciones. Solo cálculos.**

Con esto dicho, vamos a empezar a escribir nuestro reducer gradualmente enseñándole cómo entender las acciones que definimos antes.

Vamos a empezar por especificar el estado inicial. Redux va a llamar a nuestros reducers con undefined como valor del estado la primera vez. Esta es nuestra oportunidad de devolver el estado inicial de nuestra aplicación.

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

const initialState = {
  visibilityFilter: VisibilityFilters.SHOW_ALL,
  todos: []
}

function todoApp(state, action) {
  if (typeof state === 'undefined') {
    return initialState
  }

  // Por ahora, no maneja ninguna acción
  // y solo devuelve el estado que recibimos.
  return state
}
```

Un estupendo truco es usar la sintáxis de parámetros por defecto de ES6 para hacer lo anterior de forma más compacta:

```
function todoApp(state = initialState, action) {
  // Por ahora, no maneja ninguna acción
  // y solo devuelve el estado que recibimos.
  return state
}
```

Ahora vamos a manejar SET_VISIBILITY_FILTER. Todo lo que necesitamos hacer es cambiar la propiedad visibilityFilter en el estado. Fácil:

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

Nota que:

1. **No modificamos el state.** Creamos una copia con Object.assign(). Object.assign(state, { visibilityFilter: action.filter }) también estaría mal: esto modificaría el primer argumento. **Debes** mandar un objeto vacío como primer parámetro. También puedes activar la propuesta del operador spread para escribir { ...state, ...newState }.

2. **Devolvemos el anterior state en el caso default**. Es importante devolver el anterior state por cualquier acción desconocida.

Nota sobre Object.assign

Object.assign() es parte de ES6, pero no está implementado en la mayoría de los navegadores todavía. Vas a necesitar usar ya sea un polyfill, el plugin de Babel, o alguna otra función como _.assign().

Nota sobre switch y Boilerplate

La sentencia switch no es verdadero boilerplate. El verdadero boilerplate de Flux es conceptual: la necesidad de emitir una actualización, la necesidad de registrar el Store con el Dispatcher, la necesidad de que el Store sea un objeto (y las complicaciones que existen para hacer aplicaciones universales). Redux resuelve estos problemas usando reducers puros en vez de emisores de eventos.

Desafortunadamente muchos todavía eligen un framework basados en si usan switch en su documentación. Si no te gusta switch, puedes usar alguna función createReducer personalizada que acepte un mapa, como se ve en "Reduciendo el Boilerplate".

Manejando más acciones

¡Todavía tenemos dos acciones más que manejar! Vamos a extender nuestro reducer para manejar ADD_TODO.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    default:
      return state
  }
}
```

Tal cual como antes, nunca modificamos directamente state o sus propiedades, en cambio devolvemos un nuevo objeto. El nuevo todos es igual al viejo todos agregándole un único objeto nuevo al final. La tarea más nueva es creada usando los datos de la acción.

Finalmente, la implementación de COMPLETE_TODO no va a venir con ninguna una sorpresa:

```
case COMPLETE_TODO:
  return Object.assign({}, state, {
    todos: state.todos.map((todo, index) => {
      if (index === action.index) {
        return Object.assign({}, todo, {
          completed: true
        })
      }
```

```
      return todo
    })
  })
```

Debido a que queremos actualizar un objeto específico del array sin recurrir a modificaciones, necesitamos crear un nuevo array con los mismos objetos menos el objeto en la posición. Si te encuentras realizando mucho estas operaciones, es una buena idea usar utilidades como react-addons-update, updeep, o incluso una librería como Immutable que tienen soporte nativo a actualizaciones profundas. Solo recuerda nunca asignar nada a algo dentro de state antes de clonarlo primero.

Separando Reducers

Este es nuestro código hasta ahora. Es algo Here is our code so far. It is rather verboso:

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

```
        })
      })
    default:
      return state
  }
}
```

¿Hay alguna forma de hacerlo más fácil de entender? Parece que todos y visibilityFilter se actualizan de forma completamente separadas. Algunas veces campos del estado dependen uno de otro y hay que tener en cuenta más cosas, pero en nuestro caso podemos fácilmente actualizar todos en una función separada:

```
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
  }
}

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

```
      })
    case ADD_TODO:
    case COMPLETE_TODO:
      return Object.assign({}, state, {
        todos: todos(state.todos, action)
      })
    default:
      return state
  }
}
```

Fijate que todos acepta state—¡Pero es un array! Ahora todoApp solo le manda una parte del estado para que la maneje, y todos sabe cómo actualizar esa parte. **Esto es llamado *composición de reducers*, y es un patrón fundamental al construir aplicaciones de Redux.**

Vamos a explorar la composición de reducers un poco más. ¿Podemos extraer a otro reducer el control de visibilityFilter? Podemos:

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

Ahora podemos reescribir el reducer principal como una función que llama a los reducers que controlan distintas partes del estado, y los combina en un solo objeto. Ni siquiera necesita saber el estado inicial. Es suficiente con que los reducers hijos devuelvan su estado inicial cuando reciben undefined la primera vez.

```
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
  }
}

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

function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

Nota que cada uno de estos reducers está manejando su propia parte del estado global. El parámetro `state` es diferente por cada reducer, y corresponde con la parte del estado que controla.

¡Esto ya se está viendo mejor! Cuando una aplicación es muy grande, podemos dividir nuestros reducers en archivos separados y mantenerlos completamente independientes y controlando datos específicos.

Por último, Redux viene con una utilidad llamada combineReducers() que realiza la misma lógica que usa todoApp arriba. Con su ayuda, podemos reescribir todoApp de esta forma.

```
import { combineReducers } from 'redux'

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Fíjate que esto es exactamente lo mismo que:

```
export default function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

Además puedes darles diferentes nombres, o llamar funciones diferentes. Estas dos formas de combinar reducers son exactamente lo mismo:

```
const reducer = combineReducers({
  a: doSomethingWithA,
  b: processB,
  c: c
})
function reducer(state = {}, action) {
  return {
    a: doSomethingWithA(state.a, action),
    b: processB(state.b, action),
    c: c(state.c, action)
  }
}
```

Todo lo que combineReducers() hace es generar una función que llama a tus reducers **con la parte del estado seleccionada de acuerdo a su propiedad**, y combinar sus resultados en un único objeto. It's not magic..

Nota para expertos de ES6

Ya que combineReducers espera un objeto, podemos poner todos nuestros reducers en un archivo separado, export cada función reductora, y usar import * as reducers para obtenerlos todos juntos como objetos con sus nombres como propiedades.

```
import { combineReducers } from 'redux'
import * as reducers from './reducers'

const todoApp = combineReducers(reducers)
```

Ya que import * es todavía una sintaxis nueva, no la usamos más en la documentación para evitar confusiones, pero probablemente te encuentres con algunos ejemplos en la comunidad.

Código fuente

```
reducers.js
import { combineReducers } from 'redux'
import { ADD_TODO, COMPLETE_TODO, SET_VISIBILITY_FILTER, VisibilityFilters } from
'./actions'
const { SHOW_ALL } = VisibilityFilters

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
```

Store

En las secciones anteriores, definimos las acciones que representan los hechos sobre "lo que pasó" y los reductores son los que actualizan el estado de acuerdo a esas acciones.

El ***Store*** es el objeto que los reúne. El *store* tiene las siguientes responsabilidades:

- Contiene el estado de la aplicación;
- Permite el acceso al estado vía getState();
- Permite que el estado sea actualizado via dispatch(action);
- Registra los *listeners* vía subscribe(listener);
- Maneja la anulación del registro de los *listeners* vía el retorno de la función de subscribe(listener).

Es importante destacar que sólo tendrás un *store* en una aplicación Redux. Cuando desees dividir la lógica para el manejo de datos, usarás composición de reductores en lugar de muchos *stores*.

Es fácil crear una *store* si tienes un reductor. En la sección anterior, usamos combineReducers() para combinar varios reductores en uno solo. Ahora lo vamos a importar y pasarlo a createStore().

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp)
```

Opcionalmente puedes especificar el estado inicial a través del segundo argumento para createStore(). Esto es útil para hidratar el estado del cliente para que coincida con el estado de una aplicación Redux que se ejecuta en el servidor.

```
let store = createStore(todoApp, window.STATE_FROM_SERVER)
```

Enviar Acciones

Ahora que hemos creado un *store*, vamos a verificar que nuestro programa funciona! Incluso sin ninguna interfaz de usuario, ya podemos verificar la lógica de actualización.

```
import { addTodo, toggleTodo, setVisibilityFilter, VisibilityFilters } from
'./actions'

// Mostramos el estado inicial
console.log(store.getState())

// Cada vez que el estado cambie, lo mostramos
// Tenga en cuenta que subscribe() devuelve una función para anular el registro del
listener
let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
)

// Enviamos algunas acciones
store.dispatch(addTodo('Aprender sobre acciones'))
store.dispatch(addTodo('Aprender sobre reductores'))
store.dispatch(addTodo('Aprender sobre stores'))
store.dispatch(toggleTodo(0))
store.dispatch(toggleTodo(1))
store.dispatch(setVisibilityFilter(VisibilityFilters.SHOW_COMPLETED))

// Anulamos el monitoreo de las actualizaciones al estado
```

```
unsubscribe()
```

Puedes ver cómo esto hace que el estado del *store* cambie:

```
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[0]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[1]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[2]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▼ Object {visibleTodoFilter: "SHOW_COMPLETED", todos: Array[3]}
  ▼ todos: Array[3]
    ▼ 0: Object
        completed: true
        text: "Learn about actions"
      ▶ __proto__: Object
    ▼ 1: Object
        completed: true
        text: "Learn about reducers"
      ▶ __proto__: Object
    ▼ 2: Object
        completed: false
        text: "Learn about store"
      ▶ __proto__: Object
      length: 3
    ▶ __proto__: Array[0]
    visibleTodoFilter: "SHOW_COMPLETED"
  ▶ __proto__: Object
```

Hemos especificado el comportamiento de nuestra aplicación incluso antes de empezar a escribir la interfaz de usuario. No lo haremos en este tutorial, pero en este momento usted puede escribir pruebas para sus reductores y creadores de acción. No necesitarás hacer *mock* de nada porque son funciones puras. Invóquelas, y haga afirmaciones (assertions) sobre lo que devuelvan.

Código Fuente

```
index.js
import { createStore } from 'redux'
import todoApp from './reducers'

let store = createStore(todoApp)
```

Flujo de datos

La arquitectura Redux gira en torno a un **flujo de datos estrictamente unidireccional**.

Esto significa que todos los datos de una aplicación siguen el mismo patrón de ciclo de duración, haciendo que la lógica de tu aplicación sea más predecible y más fácil de entender. También fomenta la normalización de los datos, de modo que no termines con múltiples copias independientes de la misma data sin que se entere una de la otra.

Si todavía no estás convencido, lea Motivación y *The Case for Flux* para un argumento convincente a favor del flujo de datos unidireccional. Aunque Redux no es exactamente Flux, comparten las mismas ventajas clave.

El ciclo de duración de la data en cualquier aplicación Redux sigue estos 4 pasos:

1. **Haces una llamada a** store.dispatch(action).

   Una acción es un simple objeto describiendo *qué pasó*. Por ejemplo:

```
{ type: 'LIKE_ARTICLE', articleId: 42 }
{ type: 'FETCH_USER_SUCCESS', response: { id: 3, name: 'María' } }
{ type: 'ADD_TODO', text: 'Leer la documentación de Redux.' }
```

   Piense en una acción como un fragmento muy breve de noticias. "A María le gustó el artículo 42." o "'Leer la documentación de Redux.' fue añadido a la lista de asuntos pendientes."

   Puedes invocar store.dispatch(action) desde cualquier lugar en tu aplicación, incluyendo componentes y *XHR callbacks*, o incluso en intervalos programados.

2. **El *store* en Redux invoca a la función reductora que le indicaste.**

   El *store* pasará dos argumentos al reductor: el árbol de estado actual y la acción. Por ejemplo, en el caso de la aplicación de asuntos pendientes, el reductor raíz podría recibir algo como esto:

```
// El estado actual de aplicación (listado de asuntos pendientes y un filtro)
let previousState = {
  visibleTodoFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Leer la documentación.',
```

```
      complete: false
    }
  ]
}

// La acción que se está realizando (agregando un asunto)
let action = {
  type: 'ADD_TODO',
  text: 'Entendiendo el flujo.'
}

// Tu reductor devuelve el siguiente estado de aplicación
let nextState = todoApp(previousState, action)
```

Tenga en cuenta que un reductor es una función pura. Sólo *evalúa* el siguiente estado. Debe ser completamente predecible: invocarla con las mismas entradas muchas veces debe producir las mismas salidas. No debe realizar ningún efecto alterno como las llamadas al API o las transiciones del *router*. Esto debe suceder antes de que se envíe la acción.

3. **El reductor raíz puede combinar la salida de múltiples reductores en un único árbol de estado.**

   Cómo se estructura el reductor raíz queda completamente a tu discreción. Redux provee una función combineReducers() que ayuda, a "dividir" el reductor raíz en funciones separadas donde cada una maneja una porción del árbol de estado.

   Así es como funciona combineReducers(). Digamos que usted tiene dos reductores, uno para una lista de asuntos y otro para la configuración del filtro asunto actualmente seleccionado:

```
function todos(state = [], action) {
  // Calcularlo de alguna manera...
  return nextState
}

function visibleTodoFilter(state = 'SHOW_ALL', action) {
  // Calcularlo de alguna manera...
  return nextState
}

let todoApp = combineReducers({
```

```
  todos,
  visibleTodoFilter
})
```

Cuando usted emite una acción, todoApp devuelta por combineReducers llamará a ambos reductores:

```
let nextTodos = todos(state.todos, action)
let nextVisibleTodoFilter = visibleTodoFilter(state.visibleTodoFilter, action)
```

A continuación se combinará ambos conjuntos de resultados en un único árbol de estado:

```
return {
  todos: nextTodos,
  visibleTodoFilter: nextVisibleTodoFilter
}
```

Mientras combineReducers() es una utilidad de gran ayuda, no tienes que usarla; ten la liberta de escribir tu propio reductor raíz!

4. **El *store* en Redux guarda por completo el árbol de estado devuelto por el reductor raíz.**

   ¡Este nuevo árbol es ahora el siguiente estado de tu aplicación! Cada *listener* registrado usando store.subscribe(listener) será ahora invocado; los listeners podrán invocar store.getState() para obtener el estado actual.

   Ahora, la interfaz de usuario puede actualizarse para reflejar el nuevo estado. Si utilizas herramientas como React Redux, este es el momento donde invocas component.setState(newState).

Uso con React

Comencemos enfatizando que Redux no tiene relación alguna con React. Puedes escribir aplicaciones Redux con React, Angular, Ember, jQuery o vanilla JavaScript.

Dicho esto, Redux funciona especialmente bien con librerías como React y Deku porque te permiten describir la interfaz de usuario como una función de estado, y Redux emite actualizaciones de estado en respuesta a acciones.

Usaremos React para crear nuestra aplicación sencilla de asuntos pendientes (To-do).

Instalando React Redux

React Redux no está incluido en Redux de manera predeterminada. Debe instalarlo explícitamente:

```
npm install --save react-redux
```

Si no usas npm, puedes obtener la distribución UMD (Universal Module Definition) más reciente desde unpkg (ya sea la distribución de desarrollo o la de producción). La distribución UMD exporta una variable global llamada window.ReactRedux por si la añades a tu página a través de la etiqueta <script>.

Componentes de Presentación y Contenedores

Para asociar React con Redux se recurre a la idea de **separación de presentación y componentes contenedores**. Si no está familiarizado con estos términos, lea sobre ellos primero, y luego vuelva. ¡Son importantes, así que vamos a esperar!

Repasemos sus diferencias:

	Componentes de Presentación	**Componentes Contenedores**
Propósito	Como se ven las cosas (*markup*, estilos)	Cómo funcionan las cosas (búsqueda de datos, actualizaciones de estado)
Pertinente a Redux	No	Yes
Para leer datos	Lee datos de los *props*	Se suscribe al estado en Redux
Para manipular datos	Invoca llamada de retorno (callback) desde los *props*	Envia acciones a Redux
Son escritas	Manualmente	Usualmente generados por React Redux

La mayoría de los componentes que escribiremos serán de presentación, pero necesitaremos generar algunos componentes contenedores para conectarlos al *store* que maneja Redux. Con esto y el resumen de diseño que mencionaremos a continuación no implica que los componentes contenedores deban estar cerca o en la parte superior del árbol de componentes. Si un componente contenedor se vuelve demasiado complejo (es decir, tiene componentes de presentación fuertemente anidados con innumerables devoluciones de llamadas que se pasan hacia abajo), introduzca otro contenedor dentro del árbol de componentes como se indica en el FAQ.

Técnicamente usted podría escribir los componentes contenedores manualmente usando store.subscribe(). No le aconsejamos que haga esto porque React Redux hace muchas optimizaciones de rendimiento que son difíciles de hacer a mano. Por esta razón, en lugar de escribir los componentes contenedores, los generaremos utilizando el comando connect(), función proporcionada por React Redux, como verás a continuación.

Diseño de la jerarquía de componentes

Recuerda cómo diseñamos y dimos forma al objecto del estado raíz? Es hora de diseñar la jerarquía de la interfaz de usuario para que coincida con este objeto del estado. Esto no es una tarea específica de Redux. *Thinking in React* es un excelente tutorial que explica el proceso.

Nuestro breve resumen del diseño es simple. Queremos mostrar una lista de asuntos pendientes. Al hacer clic, un elemento de la lista se tachará como completado. Queremos mostrar un campo en el que el usuario puede agregar una tarea nueva. En el pie de página,

queremos mostrar un *toggle* para mostrar todas las tareas, sólo las completadas, o sólo las activas.

Diseño de componentes de presentación

Podemos ver los siguientes componentes de presentación y sus *props* surgir a través de esta breve descripción:

- **TodoList** es una lista que mostrará las tareas pendientes disponibles.
  - todos: Array es un arreglo de tareas pendientes que contiene la siguiente descripción { id, text, completed }.
  - onTodoClick(id: number) es un *callback* para invocar cuando un asunto pendientes es presionado.
- **Todo** es un asunto pendiente.
  - text: string es el texto a mostrar.
  - completed: boolean indica si la tarea debe aparecer tachada.
  - onClick() es un *callback* para invocar cuando la tarea es presionada.
- **Link** es el enlace con su *callback*.
  - onClick() es un *callback* para invocar cuando el enlace es presionado.
- **Footer** es donde dejamos que el usuario cambie las tareas pendientes visibles actualmente.
- **App** es el componente raíz que representa todo lo demás.

Cada artículo describe la *apariencia* pero no conoce *de dónde* vienen los datos, o *cómo* cambiarlos. Sólo muestran lo que se les da. Si migras de Redux a otra cosa, podrás mantener todos estos componentes exactamente iguales. No dependen de Redux en absoluto.

Diseño de componentes contenedores

También necesitaremos algunos componentes contenedores para conectar los componentes de presentación a Redux. Por ejemplo, el componente de presentación TodoList necesita un contenedor comoVisibleTodoList que se suscribe al *store* de Redux y debe saber cómo aplicar el filtro de visibilidad. Para cambiar el filtro de visibilidad, proporcionaremos un componente contenedor FilterLink que renderiza un Link que distribuye la debida acción al hacer clic:

- **VisibleTodoList** filtra los asuntos de acuerdo a la visibilidad actual y renderiza el TodoList.
- **FilterLink** obtiene el filtro de visibilidad actual y renderiza un Link.
  - filter: string es el tipo del filtro de visibilidad.

Diseño de otros componentes

A veces es difícil saber si un componente debe ser componente de presentación o contenedor. Por ejemplo, a veces la forma y la función están realmente entrelazadas, como en el caso de este pequeño componente:

- **AddTodo** es un campo de entrada con un botón "Añadir tarea"

Técnicamente podríamos dividirlo en dos componentes, pero podría ser demasiado pronto en esta etapa. Está bien mezclar presentación y lógica en un componente que sea muy pequeño. A medida que crece, será más obvio cómo dividirlo, así que lo dejaremos en uno solo.

Implementación de componentes

Vamos a escribir los componentes! Comenzaremos con los componentes de presentación por lo que no es necesario pensar en la relación con Redux todavía.

Implementación de componentes de presentación

Todos estos son componentes normales de React, por lo que no los examinaremos en detalle. Escribiremos componentes funcionales sin-estado a menos que necesitemos usar el estado local o los métodos del ciclo de duración. Esto no significa que los componentes de presentación *tengan que ser* funciones - es solo que es más fácil definirlos de esta manera. Si, y cuando necesites agregar un estado local, métodos de ciclo de duración u optimizaciones de rendimiento, puedes convertirlos a clases.

```
components/Todo.js
import React, { PropTypes } from 'react'

const Todo = ({ onClick, completed, text }) => (
  <li
    onClick={onClick}
    style={{
      textDecoration: completed ? 'line-through' : 'none'
    }}
  >
    {text}
  </li>
)

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
```

```
  completed: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default Todo
```

```
components/TodoList.js
import React, { PropTypes } from 'react'
import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map(todo =>
      <Todo
        key={todo.id}
        {...todo}
        onClick={() => onTodoClick(todo.id)}
      />
    )}
  </ul>
)

TodoList.propTypes = {
  todos: PropTypes.arrayOf(PropTypes.shape({
    id: PropTypes.number.isRequired,
    completed: PropTypes.bool.isRequired,
    text: PropTypes.string.isRequired
  }).isRequired).isRequired,
  onTodoClick: PropTypes.func.isRequired
}

export default TodoList
```

```
components/Link.js
import React, { PropTypes } from 'react'

const Link = ({ active, children, onClick }) => {
  if (active) {
    return <span>{children}</span>
  }
```

```
  return (
    <a href="#"
       onClick={e => {
         e.preventDefault()
         onClick()
       }}
    >
      {children}
    </a>
  )
}

Link.propTypes = {
  active: PropTypes.bool.isRequired,
  children: PropTypes.node.isRequired,
  onClick: PropTypes.func.isRequired
}

export default Link
```

```
components/Footer.js
import React from 'react'
import FilterLink from '../containers/FilterLink'

const Footer = () => (
  <p>
    Show:
    {" "}
    <FilterLink filter="SHOW_ALL">
      Todos
    </FilterLink>
    {", "}
    <FilterLink filter="SHOW_ACTIVE">
      Activo
    </FilterLink>
    {", "}
    <FilterLink filter="SHOW_COMPLETED">
      Completado
    </FilterLink>
  </p>
)
```

```
export default Footer
```

```
components/App.js
import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
  </div>
)

export default App
```

Ahora es el momento de conectar los componentes de presentación a Redux mediante la creación de algunos contenedores. Técnicamente, un componente contenedor es sólo un componente de React que utiliza store.subscribe () para leer una parte del árbol de estado en Redux y suministrar los *props* a un componente de presentación que renderiza. Puedes escribir un componente contenedor manualmente, pero sugerimos generar los componentes contenedores con la función connect() de la librería React Redux, ya que proporciona muchas optimizaciones útiles para evitar *re-renders* innecesarios. (Un beneficio de utilizar esta librería es que usted no tiene que preocuparse por la implementación del método shouldComponentUpdate recomendado por React para mejor rendimiento.)

Para usar connect(), es necesario definir una función especial llamada mapStateToProps que indique cómo transformar el estado actual del *store* Redux en los *props* que desea pasar a un componente de presentación. Por ejemplo, VisibleTodoList necesita calcular todos para pasar a TodoList, así que definimos una función que filtra el state.todos de acuerdo con el state.visibilityFilter, y lo usamos en su mapStateToProps:

```
const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
```

```
      return todos.filter(t => !t.completed)
  }
}

const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

Además de leer el estado, los componentes contenedores pueden enviar acciones. De manera similar, puede definir una función llamada mapDispatchToProps() que recibe el método dispatch() y devuelve los *callback props* que deseas inyectar en el componente de presentación. Por ejemplo, queremos que VisibleTodoList inyecte un *prop* llamado onTodoClick en el componente TodoList, y queremos que onTodoClick envíe una acción TOGGLE_TODO:

```
const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}
```

Finalmente, creamos VisibleTodoList llamando connect() y le pasamos estas dos funciones:

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Estos son los conceptos básicos de la API de React Redux, pero hay algunos atajos y opciones avanzadas por lo que le animamos a revisar su documentación en detalle. En caso de que te preocupe el hecho que mapStateToProps esté creando objetos nuevos con

demasiada frecuencia, quizás desees aprender acerca de computar datos derivados con *reselect*.

El resto de los componentes contenedores están definidos a continuación:

```
containers/FilterLink.js
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => {
  return {
    active: ownProps.filter === state.visibilityFilter
  }
}

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    onClick: () => {
      dispatch(setVisibilityFilter(ownProps.filter))
    }
  }
}

const FilterLink = connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)

export default FilterLink
```

```
containers/VisibleTodoList.js
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'

const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
```

```
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
  }
}

const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Implementación de otros componentes

```
containers/AddTodo.js
import React from 'react'
import { connect } from 'react-redux'
import { addTodo } from '../actions'

let AddTodo = ({ dispatch }) => {
  let input

  return (
    <div>
      <form onSubmit={e => {
        e.preventDefault()
        if (!input.value.trim()) {
```

```
          return
        }
        dispatch(addTodo(input.value))
        input.value = ''
      }}>
        <input ref={node => {
          input = node
        }} />
        <button type="submit">
          Añadir tarea
        </button>
      </form>
    </div>
  )
}
AddTodo = connect()(AddTodo)

export default AddTodo
```

Transferir al store

Todos los componentes contenedores necesitan acceso al *store Redux* para que puedan suscribirse a ella. Una opción sería pasarlo como un *prop* a cada componente contenedor. Sin embargo, se vuelve tedioso, ya que hay que enlazar store incluso a través del componentes de presentación ya que puede suceder que tenga que renderizar un contenedor allá en lo profundo del árbol de componentes.

La opción que recomendamos es usar un componente *React Redux* especial llamado <Proveedor> para mágicamente hacer que el *store* esté disponible para todos los componentes del contenedor en la aplicación sin pasarlo explícitamente. Sólo es necesario utilizarlo una vez al renderizar el componente raíz:

```
index.js
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

let store = createStore(todoApp)
```

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

Example: Todo List

Este es el código fuente completo de la mini aplicación to-do que desarrollamos durante el tutorial básico.

Punto de entrada

```
index.js
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

let store = createStore(todoApp)

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

Creadores de acciones

```
actions/index.js
let nextTodoId = 0
export const addTodo = (text) => {
  return {
    type: 'ADD_TODO',
    id: nextTodoId++,
    text
  }
}

export const setVisibilityFilter = (filter) => {
  return {
    type: 'SET_VISIBILITY_FILTER',
    filter
  }
}
```

```
export const toggleTodo = (id) => {
  return {
    type: 'TOGGLE_TODO',
    id
  }
}
```

Reductores

```
reducers/todos.js
const todo = (state = {}, action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return {
        id: action.id,
        text: action.text,
        completed: false
      }
    case 'TOGGLE_TODO':
      if (state.id !== action.id) {
        return state
      }

      return Object.assign({}, state, {
        completed: !state.completed
      })

    default:
      return state
  }
}

const todos = (state = [], action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        todo(undefined, action)
      ]
    case 'TOGGLE_TODO':
```

```
      return state.map(t =>
        todo(t, action)
      )
    default:
      return state
  }
}

export default todos
```

```
reducers/visibilityFilter.js
const visibilityFilter = (state = 'SHOW_ALL', action) => {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}

export default visibilityFilter
```

```
reducers/index.js
import { combineReducers } from 'redux'
import todos from './todos'
import visibilityFilter from './visibilityFilter'

const todoApp = combineReducers({
  todos,
  visibilityFilter
})

export default todoApp
```

Componentes de presentación

```
components/Todo.js
import React, { PropTypes } from 'react'

const Todo = ({ onClick, completed, text }) => (
```

```
  <li
    onClick={onClick}
    style={{
      textDecoration: completed ? 'line-through' : 'none'
    }}
  >
    {text}
  </li>
)

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  completed: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default Todo
```

```
components/TodoList.js
import React, { PropTypes } from 'react'
import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map(todo =>
      <Todo
        key={todo.id}
        {...todo}
        onClick={() => onTodoClick(todo.id)}
      />
    )}
  </ul>
)

TodoList.propTypes = {
  todos: PropTypes.arrayOf(PropTypes.shape({
    id: PropTypes.number.isRequired,
    completed: PropTypes.bool.isRequired,
    text: PropTypes.string.isRequired
  }).isRequired).isRequired,
  onTodoClick: PropTypes.func.isRequired
}
```

```
export default TodoList
```

```
components/Link.js
import React, { PropTypes } from 'react'

const Link = ({ active, children, onClick }) => {
  if (active) {
    return <span>{children}</span>
  }

  return (
    <a href="#"
       onClick={e => {
         e.preventDefault()
         onClick()
       }}
    >
      {children}
    </a>
  )
}

Link.propTypes = {
  active: PropTypes.bool.isRequired,
  children: PropTypes.node.isRequired,
  onClick: PropTypes.func.isRequired
}

export default Link
```

```
components/Footer.js
import React from 'react'
import FilterLink from '../containers/FilterLink'

const Footer = () => (
  <p>
    Show:
    {" "}
    <FilterLink filter="SHOW_ALL">
      Todos
```

```
    </FilterLink>
    {", "}
    <FilterLink filter="SHOW_ACTIVE">
      Activos
    </FilterLink>
    {", "}
    <FilterLink filter="SHOW_COMPLETED">
      Completados
    </FilterLink>
  </p>
)

export default Footer
```

```
components/App.js
import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
  </div>
)

export default App
```

Componentes contenedores

```
containers/VisibleTodoList.js
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'

const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
```

```
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
  }
}

const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

```
containers/FilterLink.js
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => {
  return {
    active: ownProps.filter === state.visibilityFilter
  }
}

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
```

```
    onClick: () => {
      dispatch(setVisibilityFilter(ownProps.filter))
    }
  }
}

const FilterLink = connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)

export default FilterLink
```

Otros componentes

```
containers/AddTodo.js
import React from 'react'
import { connect } from 'react-redux'
import { addTodo } from '../actions'

let AddTodo = ({ dispatch }) => {
  let input

  return (
    <div>
      <form onSubmit={e => {
        e.preventDefault()
        if (!input.value.trim()) {
          return
        }
        dispatch(addTodo(input.value))
        input.value = ''
      }}>
        <input ref={node => {
          input = node
        }} />
        <button type="submit">
          Añadir tarea
        </button>
      </form>
    </div>
```

```
  )
}
AddTodo = connect()(AddTodo)

export default AddTodo
```

Fuente : https://es.redux.js.org/docs/basico/ejemplo-todos.html