arielZarate / HenryAll Public

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main

HenryAll / FT-M2-master / 12-React-Redux /

arielZarate ...		8 days ago
..		
demo-mutation		27 days ago
demo		8 days ago
demoSubscribe		27 days ago
homework		8 days ago
redus		27 days ago
README.json		27 days ago
README.md		27 days ago

README.md

Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

React Redux

```
//Redux is a predictable state container for JavaScript apps.
```

Redux es una librería que nos va a ayudar a mantener el estado *global* de nuestra aplicación.

> Podemos usar Redux fuera de React, y React sin redux también. Pero ambas funcionan muy bien juntas, por esto la comunidad las adoptó rápidamente para usarlas juntas.

Si bien, podemos crear *containers* que mantengan el estado estado de sus *childrens*, lo que termina ocurriendo es que los Componentes *presentacionales* terminan demasiado acoplados a los *containers*, bajando su *resusabilidad*. Otro tema, es que en estos *containers* vamos a tener que escribir funciones que manejen el estado de varios Componentes, haciendo que este archivo se convierta en inmanejable de forma rápida. Justamente, `Redux` nos va a ayudar a resolver estos problemas con el paradigma que implementa.

Los tres principios de Redux

> Vamos a ir introducciendo cierta terminología específica de Redux, pueden leer el glosario completo aca

Única fuente de verdad (Single source of truth)

> El **estado** de toda tu aplicación está guardado en un árbol en una sola **store**.

Esto hace que se fácil crear apps universale, ya que el *estado* de tu servidor puede ser *serializado* fácilmente a todos los clientes sin efuerzo extra. Además hace que sea más fácil *debuggear* e *inspeccionar* tu aplicación.

```
console.log(store.getState())

/* Prints
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
```

```
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
*/
```

Los Estados son sólo lectura (State is read-only)

> La única forma de cambiar un estado es emitiendo una **acción**, que es un objeto que describe lo que ocurrió.

Esto asegura que ni la vista, ni ningún callback escriban directamente sobre el *estado*. En cambio, tienen que expresar una *intención* de transformar el estado. Como todas los cambios están *centralizados* y ocurren en un orden estricto, no vamos a tener que preocuparnos por qué cosas suceden primero (debido a la naturaleza *asincrónica*). Como las acciones son *objetos*, pueden ser logeados, serializados, guardados y pueden ser reproducidas en el futuro para *debuggear* o *testear* la aplicación.

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

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

Los cambios se hacen con funciones puras (Changes are made with pure functions)

> Para especificar cómo se transforma el *árbol de estado* se escriben funciones llamadas **reducers**.

Las funciones *reducers* son funciones puras, que toman el *estado anterior* y un *acción* y retornan el *nuevo estado*. Estas funciones deben retornar *nuevos objetos de estado* y no *mutar el estado anterior*. Como las *reducers* son sólo funciones, podemos manejar el orden en el que se ejecutan, pasar datos adicionales, o inclusive hacer *reducers* reutilizables para tareas comunes.

```
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
  }
}

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

Nuestra propia implementación de Redux

Para entender bien el concepto de *Redux*, vamos a crear nuestra propia mini-implementación del paradigma que utiliza. Luego vamos a usar la librería `redux`.

Para hacerlo, empecemos repasando las ideas principiales que tenemos que implementar:

- Toda la data mantenida por nuestra aplicación tiene que estar contenida en una única estructura de datos llamada el `state` de nuestra app. Esta estructura de datos debe estar guardada en el `store`.
- Nuestra app lee el **state** desde nuestra **store**.
- El **store** no puede ser manipulado directamente por el usuario.
- Los usuarios disparan **acciones** que describen qué sucedió.

- Un *nuevo estado* es generado, resultado de combinar el *viejo estado* y la *acción* del usuario. Este proceso lo realiza una función llamada **reducer**.

Reducers

Un *reducer* toma un estado *viejo (actual)* y una *acción* y devuelve un *estado nuevo*. Un *reducer* **debe ser una *función pura***, es decir:

- No debe mutar directamente el estado.
- No debe usar datos que no hayan sido pasada por argumentos.

Los *reducers* siempre deben tratar el *estado actual* como **sólo lectura**. De hecho, el *reducer* **no cambia el estado**, si no que **devuelve un estado nuevo**. Para crear nuestro primer reducer, entonces, vamos a necesitar:

- Una **acción**, que nos define que hacer (opcionalmente con argumentos).
- El **estado**, que *guarda* toda la información de nuestro *app*.
- El **reducer** *per se* que recibe el *estado* y la *acción* y retorna el *nuevo estado*.

Creando un Reducer

Empezemos por el *reducer* más simple posible, es el que devuelve el **estado actual**. Similar a la función Identidad.

```
var reducer = function(state, action) {
  return state;
}
```

Para los siguientes ejemplos, consideremos que el **estado** es un entero y que empieza en 0, podríamos llamarlo `counter`.

> En el común de los casos, los estados van a ser *sumamente* más complejos que sólo un entero!

Para incrementar o decrementar en uno nuestro estado, vamos a necesitar definir una **acción**, para que el usuario pueda indicar que tiene la *intención* de cambiar el estado de nuestra app.

Una acción sólo necesita una propiedad `type`:

```
// incrementar en uno el counter
var INCREMENTAR_COUNTER = {
  type: 'INCREMENTAR_COUNTER'
}
```

```
// decrementar en uno el counter
var DECREMENTAR_COUNTER = {
  type: 'DECREMENTAR_COUNTER'
}
```

Ahora que tenemos estas acciones las usemos en nuestros *reducers*:

```
var reducer = function(state, action){
  switch(action.type){
    case "INCREMENTAR_COUNTER":
      return state + 1;
    case "DECREMENTAR_COUNTER":
      return state - 1;
    default:
      return state; // caso por defecto
  }
}
```

Listo! Ahora vamos a tener el *nuevo estado* retornado según qué acciones hayamos pasado al reducer.

> Noten que dejamos el caso por **defecto** de tal modo que si no pasamos una acción *conocida*, el reducer vuelva el estado como estaba.

Argumentos en las acciones

En el ejemplo anterior, la acción siempre incrementaba en uno. Pero a veces, para describir la acción es necesario contar una serie de parámetros, por ejemplo en el caso que querramos incrementar nuestro `counter`, pero en un valor mayor a uno. Primero vamos a definir una **acción** en la cual podamos incrementar `7` veces el `counter`:

```
// incrementar en uno el counter
var INCREMENTAR_COUNTER = {
  type: 'INCREMENTAR_COUNTER'
}
// decrementar en uno el counter
var DECREMENTAR_COUNTER = {
  type: 'DECREMENTAR_COUNTER'
}
// incrementar el counter en n
var INCREMENTAR_7 = {
  type: 'INCREMENTAR_N',
  payload: 7
}
```

y la agregamos en nuestro **reducer**:

```
var reducer = function(state, action){
  switch(action.type){
    case "INCREMENTAR_COUNTER":
      return state + 1;
    case "DECREMENTAR_COUNTER":
      return state - 1;
    case "INCREMENTAR_N":
      return state + action.payload;
    default:
      return state; // caso por defecto
  }
}
```

En el ejemplo vemos que el type de accion es `INCREMENTAR_N`, pero sólo hemos creado la acción de ese tipo, pero con el `payload` en un número fijo, para generar varias acciones de este tipo vamos a hacer un *generador de acciones*, que no es otra cosa que una `factory`:

```
function  increment(n) {
  return {
    type: 'INCREMENTAR_N',
    payload: n
  }
}
```

Ahora podríamos invocar a nuestro reducer de la siguiente manera:

```
var state = 0;
reducer(state, increment(7)); // Incrementa siete // 7
reducer(state, INCREMENTAR_COUNTER) // incrementa de a uno. // 1
```

Si vemos el ejemplo, notamos que a pesar de pasar por los *reducers*, nuestro estado sigue siendo `0` siempre! Por lo tanto nos vemos en la necesidad de implementar un `Store`, que es la forma de guardar el estado en la filosofía `redux`:

```
 class Store {
   constructor(estadoInicial, reducer) {
     this._state = estadoInicial;
     this.reducer = reducer;
   }

   getState(): {
     return this._state;
   }

   dispatch(action) {
     this._state = this.reducer(this._state, action);
```

```
    }
  }
```

> En Redux, generalmente, tenemos un Store y un top level reducer.

Veamos que contiene nuestra **Store**:

- Cuando la inicializamos vamos a pasarle un *Estado Inicial* y un *reducer*.
- `getState()` retorna el *Estado Actual*.
- Por último tenemos la función `dispatch` que recibe una acción e invoca al `reducer` con el estado actual y la acción, y actualiza el estado con lo que retorna el `reducer` (el nuevo estado).

Noten que `dispatch` no retorna nada, simplemente *actualiza* el estado de la aplicación. Esto es un concepto importante de `redux` : cuando *despachamos* una acción, lo hacemos y nos olvidamos. **Despachar una acción no es una manipulación directa del estado, y no devuelve el nuevo estado**.

Usando el Store

Veamos como podemos usar nuestro `Store` :

```
var store = new Store(0, reducer);
console.log(store.getState()); //0
store.dispatch( INCREMENTAR_7 );
console.log(store.getState()); //7
store.dispatch( INCREMENTAR_COUNTER )
console.log(store.getState()); //8
store.dispatch( DECREMENTAR )
console.log(store.getState()); //7
```

Empezamos por crear una `Store` nueva y la guardamos, en este caso, en una variable también llamada `store` . Luego usaremos el objeto que guardamos para consultar el estado en el que se encuentra nuestra aplicación.

Pedirle al Store que nos notifique los cambios

Si vemos el ejemplo anterior, siempre que queremos ver el estado del `Store` tenemos que pedirle explicitamente. Sería interesante que nos enteremos que una acción fue despachada para que podamos responder si es necesario. Para esto vamos a implementar el patrón Observador, es decir que vamos a *suscribir* un callback para que escuche por cambios.

Esto es lo que queremos hacer:

- Registrar una función *listener* usando `suscribe` .

- Cuando `dispatch` es invocada, iteraremos sobre todos los *listeners* que tenga subscriptos y los invocaremos, notificandolos que el Estado ha cambiado.

Para esto vamos a agregar la funcionalidad de `suscribe` y `unsuscribe` a nuestro `Store`:

```
class Store {
  constructor(estadoInicial, reducer) {
    this._state = estadoInicial;
    this.reducer = reducer;
    this._listeners = []; // Empezamos con un arreglo vacío.
  }

  getState(): {
    return this._state;
  }

  dispatch(action) {
    this._state = this.reducer(this._state, action);
  }

  suscribe(listener): {
     _listeners.push(listener);
     return () => { // retorna una función unsuscribe
        this._listeners = this._listeners.filter(function(l){l !== listener});
     };
  }
}
```

La función `suscribe` recibe una función como listener, y retorna una función que al ser ejecutada, va a `filtrar` el listener con el que fue generada de la lista de listeners.

Ahora, para notificar a los listeners suscriptos, vamos a invocarlos cuando se ejecute el método `dispatch`:

```
dispatch(action) {
     this._state = this.reducer(this._state, action);
     this._listeners.forEach(function(listener){
        listener();
      })
  }
```

Ahora que nos podemos suscribir para obtener novedades, probemos lo siguiente:

```
var store = new Store(0, reducer);
var unsuscribe = store.suscribe( function() {
  console.log(store.getState());
})
```

```
store.dispatch( INCREMENTAR_7 ); // --> 7
store.dispatch( INCREMENTAR_COUNTER ); // --> 8
store.dispatch( DECREMENTAR ); // --> 7

unsuscribe(); // dejamos de recibir notificaciones;

store.dispatch( DECREMENTAR );
console.log(store.getState()); // 6
```

Si entendiste lo anterior, entonces ya tienes una idea de las bases sobre las que siente `redux`. La librería implementa otros patrones más complejos, y resuelve problemas comunes que pueden llegar a aparecer.

Instalando Redux

Redux viene en un paquete con el mismo nombre, asi que para instalarlo podemos hacer `npm install redux react-redux`. Con esto instalamos la librería de redux en sí, y los `helpers` de `react-redux`, donde están las funciones `bindActionsCreator`, `connect`, etc..

> Redux se instala así, asumiendo que ya tenemos un proyecto de React funcionando, con su webpack configurado.

Workflow de Redux

Lo primero que tenemos que incorporar para trabajar con Redux, es el workflow que nos propone. Básicamente, nuestra aplicación debería funcionar siempre siguiente este ciclo de vida, en el medio van a aparecer elementos de la API de redux que vamos a ir explicando.

> Se podría considerar a `redux` cómo una implementación del patrón `flux` para react, también se podría considerar cómo un patrón por si mismo. (De hecho, ni sus autores se ponen de acuerdo en eso) Lo cierto es que está influenciado por el patrón `flux`, usando por facebook.

1. El `árbol de Estado` define la UI y las acciones posibles a través de `props`.
2. Acciones realizadas por los usuarios son enviadas a un `action creator` que las normaliza.
3. Estas acciones normalizadas son pasadas a un `reducer`, que es donde se ejecuta el código que contiene la lógica de la acción.
4. El reducer crea un nuevo Estado y lo devuelve (`dispatches it`) al `Store`.
5. La UI es actualizada acorde al nuevo estado.

Acciones

Las **acciones** representan una *inteción* de cambiar el estado de nuestra `store`. Las *acciones* son las **únicas** fuente de información que llega a nuestras `stores`.

Las **acciones** son **objetos JavaScript planos**, deben tener una propiedad llamada `type`, que indica o describa la acción que se realiza. Por convención los `type` s debe estar escritos como `string constants`, es decir, todo en mayúsculas y con SNAKE_CASE. Además del type, las acciones van a contener cualquier otra propiedad que necesitemos para su funcionamiento. Se recomienda que las acciones tengan la menor cantidad de propiedades posibles.

Podemos ver algunas recomendaciones de cómo diseñar nuestras acciones en esta guía.

> Cuando tu app empieze a crecer, es buena idea separar las acciones en distintos módulos (archivos).

```
// incrementas likes
const INCREMENT_LIKES =  {
    type: 'INCREMENT_LIKES',
    index: 4,
  }

// agregar comentarios
const ADD_COMMENT = {
    type: 'ADD_COMMENT',
    postId: 'X4Yb4',
    author: 'Toni',
    comment: 'Esto es una acción en particular.',
  }
// remover comentarios
const REMOVE_COMMENT =  {
    type: 'REMOVE_COMMENT',
    postId: 'X4Yb5',
    index: 5,
  }
```

Action Creators

Los `actions creators` son funciones que **crean** acciones, o sea que *retornan* un objeto que representa una acción. Es fácil confundir los términos *action* y *action creator* así que hay que usarlos con cuidado.

```
// incrementas likes
export function  increment(index) {
  return {
    type: 'INCREMENT_LIKES',
    index
  }
}
```

```
// agregar comentarios
export function addComment(postId, author, comment) {
  return {
    type: 'ADD_COMMENT',
    postId,
    author,
    comment
  }
}
// remover comentarios
export function removeComment(postId, index) {
  return {
    type: 'REMOVE_COMMENT',
    postId,
    index
  }
}
```

Dispatch

Las *acciones* tienen que ser enviadas al *Store* para que surgan efecto. La función `dispatch` base es un método de `store`, esta manda una acción de forma sincrónica a los reductores del store, junto con el estado previo retornado por el store, para calcular el nuevo estado. Espera que las acciones que les pasemos sean objetos planos, listas para ser consumidas por los reducers.

También se puede envolver a los dispatchers en una serie de Middlewares, esto permite que los dispatchers puedan manejar *acciones asincrónicas*, además de poder transformar, demorar, ignorar o interpretar acciones antes de pasarla al siguiente Middleware.

> Cuando usamos redux con react, las funciones de dispatchers vienen bindeadas al componente

Reducers

Las *acciones* describen que algo sucedió en nuestra app, pero no especifican *cómo* esta accion impacta en el estado actual. Saber eso es el trabajo de los **reducers**.

Un *reducer* es una función que recibe el estado previo de un Store y un acción y retorna el nuevo estado. Justamente se llama reducer, porque toma al estado cómo una *acumulación* de acciones. Los reducers tienen que ser siempre funciones **puras**, estas son cosas que **nunca** deberías hacer en un reducer:

- Mutar sus argumentos.
- Realizar cosas que tengan efectos secundarios, como llamadas a APIs, o routeo.
- Llamar a funciones no puras adentro, como `Math.random()` o `Date.now()`

> Given the same arguments, a reducer should calculate the next state and return it. No surprises. No side effects. No API calls. No mutations. Just a calculation.

```
function posts( state = [], action) {
  switch(action.type){
    case 'INCREMENT_LIKES':
      console.log('increment Likes');
      const i = action.index;
      return [
        ...state.slice(0,i), // antes del que estamos actualizando
        {...state[i], likes: state[i].likes + 1},
        ...state.slice(i+1)// despues del actualizado
      ]
    default:
      return state;

  }
}

export default posts;
```

Cuando la app es grande, podemos poner cada reducer en archivos separados, agrupandolos según los datos que manejen, haciendolos independientes entre ellos. Para hacer esto, redux nos ofrece la funcionalidad de `combineReducers()`, lo que hace es convertir un objeto que tiene varios reducers en una sola función reducers que los contiene, y que puede ser pasada a `createStore`.

```
// Redux sólo puede tener un reducer.
// por eso vamosa  tener el Root Reducer
// donde vamos a incorporar los demas

import { combineReducers } from 'redux';
import { routerReducer } from 'react-router-redux';

import posts from './posts.js';
import comments from './comments.js';

const rootReducer = combineReducers( {posts, comments, routing: routerReducer })

export default rootReducer;
```

Store

La **Store** tiene la siguientes responsabilidades:

- Mantiene el estado de la aplicación.
- Permite el acceso al estado a través de `getState()`.

- Permite actualizar el estado a través de `dispatch(action)`.
- Registra *listeners* con `subscribe(listener)`.
- Maneja la desuscripción de *listeners*.

Hay que notar que vamos a tener exactamente *una* Store por cada application que hagamos usando Redux. Para crear una Store, vamos a usar la función `createStore` que recibe un reducer como argumento, y opcionalmente el *estado inicial* de la app.

```
import { createStore, compose } from 'redux';
import { syncHistoryWithStore} from 'react-router-redux';
import { browserHistory } from 'react-router';

// Import root reducer
import rootReducer  from './reducers/index.js';

import comments from './data/comments.js';
import posts from './data/posts.js';

const defaultState = {
  posts,
  comments
}

const store = createStore(rootReducer, defaultState);
```

Usando Redux con React

Para usar redux con react, vamos a usar un paquete llamado `react-redux` que nos ofrece los `bindings` de redux con react. Para instalarlo hacemos:

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

Componentes

Los bindings de `react-redux` están realizados pensandos en el patrón de **separar los Componentes Presentacionales de los Containers**.

	Presentacionales	Containers
Próposito	Cómo se ven las cosas (markup, estilos)	Cómo funcionan las cosas (traer datos, actualizar estados)
Sabe de Redux	NO	SI

	Presentacionales	Containers
Para leer datos	Lee de props	Se suscribe a los estados de Redux
Para cambiar datos	Invoca callbacks de sus props	Envía acciones a Redux
Son escritos	A mano	Generados por React Redux

> Técnicamente podríamos codear los Containers por nosotros mismos usando `subscribe`, pero Redux nos desaconseja de hacer esto, ya que nos proveen de la función `connect`, que nos permite generarlos, y estos Componentes generados están optimizados en términos de performance.

Para generar un Componente que tenga todos los bindings de redux con react, primero vamos a tener que definir las siguientes funciones:

- **mapStateToProps**: Recibe el estado de la apliación y lo mapea a props de react.
- **mapDispatchToProps**: Recibe el método `dispatch` y retorna callbacks props que vamos a poder pasar a los Componentes presentacionales.

Finalmente, usando `connect` de `react-redux` y pasandole estas dos funciones obtenemos una función lista para darle los binding a un Componente React. Finalmente elegimos que Componente queremos que tenga los bindings y luego lo exportamos. Por ejemplo, vamos a darle los binding al Componente `Main` y lo vamos a exportar como `App`:

```
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import * as actionsCreators from '../actions/actionCreators.js';

import Main from './Main.js';

function mapStateToProps(state) {
  return {
    posts: state.posts,
    comments: state.comments
  }
}

function mapDispatchToProps(dispatch) {
  return bindActionCreators(actionsCreators, dispatch);
}

const App = connect(mapStateToProps, mapDispatchToProps)(Main);

export default App;
```

Provider

Todos los Componentes Containers deben tener acceso al `Store` para que puedan suscribirse a ella. Una opción seria pasar el store como un prop a cada componente Container, pero esto se volvería tedioso muy rápidamente, y un posible punto de error. Lo que nos recomienda `redux` es usar un Componente especial de `react-redux` llamado `<Provider>` que mágicamente hace que el Store esté disponible para todos los Container de nuestra app, sin pasarla explícitamente.

```
...
import { Provider } from 'react-redux'; //Bindings from redux and React
import store, { history } from './store.js';

const router = (
  <Provider store={store}>
    <Router history={ history }>
      <Route path="/" component={App}>
        <IndexRoute component={PhotoGrid}/>
        <Route path="view/:postId" component={Single}/>
      </Route>
    </Router>
  </Provider>
)
...
```

Ahora vamos a poder acceder a nuestro `store` en cada Componente a través de sus props.

Homework

Completa la tarea descrita en el archivo README