Tema 11

Redux

¿Qué es Redux?

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.

Instalación

Redux no tiene relación con React, puede usarse en casi cualquier framework que use JavaScript (Angular, Ember, jQuery, vanilla javascript, etc).

Para instalar redux para react podemos hacerlo mediante el siguiente comando:

npm install --save react-redux

Es posible que también necesitemos instalar redux:

npm install --save redux

Componentes de Presentación y Contenedores

Este es un patrón muy útil a la hora de trabajar con React, si dividimos los componentes de nuestras aplicaciones en dos categorías descubriremos que es mucho más fácil y reusable.

Estas categorías se llaman **Container** y **Presentational components (Contenedores y Componentes de Presentación**). También se les puede llamar de otra forma pero la idea final es la misma.

Componentes de Presentación

Componentes de Presentación:

- Indican cómo las cosas se ven.
- Pueden contener ambos tipos de componentes (presentacion y contenedor).
- No tienen dependencias con el resto de la aplicación.
- No cargan datos.
- Reciben los datos de llamadas o props.
- Se escriben como componentes funcionales.

Contenedores

Contenedores:

- Indican cómo funcionan las cosas.
- Pueden tener ambos componentes pero normalmente no tendrán componentes de presentación.
- Obtienen los datos para guardarlos y pasarlos a los componentes de presentación.
- Hacen llamadas Flux y otros tipos de llamadas a los componentes de presentación.

Los pondremos en distintas carpetas para diferenciarlos.

Ventajas

Este patrón nos ofrece las siguientes ventajas en nuestras aplicaciones:

- Mejor separación de conceptos, tendremos una mejor vista de cada parte de la aplicación.
- Mejor reusabilidad, podemos reutilizar los componentes fácilmente al estar separados.
- Abstracción, al tener los componentes separados es más fácil detectar dónde ha ocurrido un fallo y solucionarlo o cambiar aspectos de esos componentes.

Podéis ver un ejemplo de este patrón en:

https://gist.github.com/chantastic/fc9e3853464dffdb1e3c

Conceptos principales de Redux

Imagina que tienes una aplicación como la siguiente:

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

Conceptos principales de Redux

Este objeto es un modelo pero no tiene métodos setters, de esta forma no se puede modificar el modelo.

Para cambiar algo en el estado necesitamos realizar un action. Aquí tienes algunos ejemplos de acciones:

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

Conceptos principales de Redux

Mediante estas acciones podemos detectar cuando se produce un cambio y por qué. Las acciones son como migas de pan de lo que ha sucedido. Por último, para unir el estado y las acciones juntos, escribimos una función llamada de un reducer.Es simplemente una función que toma el estado y la acción como argumentos, y devuelve el siguiente estado de la aplicación. Sería difícil escribir una función de este tipo para una gran aplicación, por lo que escribir funciones que gestionan pequeñas partes del estado:

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

Conceptos principales de Redux

```
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
  }
}
```

Conceptos principales de Redux

Ahora escribimos otro reducer que maneja el estado completo de la aplicación llamando a los dos reducer que hemos creado arriba. Al añadirlos les asignamos las correspondientes acciones.

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

Esta es la idea principal de Redux, controlar el estado de la aplicación y saber qué ocurre y por qué.

Tres principios de Redux

El estado de toda la aplicación se almacena en un árbol de objetos dentro de un store.

Esto hace que sea fácil crear aplicaciones universales. Un árbol de estados también hace que sea más fácil de depurar o inspeccionar una aplicación; sino que también le permite conservar el estado de su aplicación en el desarrollo, para un ciclo de desarrollo más rápido.

Por ejemplo: todo el estado se almacena en un solo árbol.

Conceptos principales de Redux

```
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
    }
  ]
}
*/
```

Conceptos principales de Redux

La única manera de cambiar el estado es emitir una acción, un objeto que describe lo que sucedió.

Esto asegura que ni las vistas ni las llamadas cambian el estado. Debido a que todos los cambios están centralizados y suceden uno tras otro en un orden estricto, no hay condiciones de carrera sutiles a tener en cuenta.

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

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

Conceptos principales de Redux

Para especificar cómo el árbol de estados es transformado por acciones, se escriben reducers.

Los Reductores son funciones que tienen el estado anterior y una acción, y devuelven el siguiente estado. Se puede comenzar con un único reductor, y conforme la aplicación crezca, dividirlo en reductores de menor tamaño que gestionan las partes específicas del árbol de estados. Debido a que los reductores son sólo funciones, se puede controlar el orden en que sean llamados, pasar datos adicionales, o incluso hacer reductores reutilizables para tareas comunes tales como la paginación.

Conceptos principales de Redux

```
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'
const reducer = combineReducers({ visibilityFilter, todos })
const store = createStore(reducer)
```

Redux

Aquí tienes un enlace a unos vídeos explicativos para entender un poco mejor el funcionamiento de redux y sus principios básicos:

https://egghead.io/courses/getting-started-with-redux