Almacenes

Tiempo estimado: 20min

Ya conocemos los dos elementos con los que comienza todo flujo de datos de la aplicación al origen de datos. Por un lado, las acciones representan una operación de datos. Mientras que el despachador es el objeto que se encarga de notificar estas acciones. Veamos ahora cómo trabajar con los almacenes. Los objetos que tienen como responsabilidad el acceso a los orígenes de datos como, por ejemplo, bases de datos o servicios webs REST.

La lección comienza introduciendo el concepto de almacén. Uno de los elementos claves de la arquitectura Flux. Seguimos con la clase base Store, del paquete flux, de los almacenes de nuestras aplicaciones. A continuación, volvemos a ver los controladores de acciones, ya presentados en la lección anterior. Pero más detenidamente, indicando claramente cuáles son sus responsabilidades. Después, introducimos las operaciones de acceso, aquellos métodos del almacén que realizan los accesos a los orígenes de datos asociados. Terminamos con los eventos de cambio, el medio a través del cual se notifica a la interfaz de usuario de que se han producido cambios. Así como el generador de Justo que nos asistirá en la creación de una plantilla almacén.

Al finalizar la lección, el estudiante sabrá:

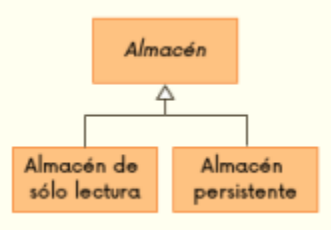
- Qué es un almacén.
- Cómo implementar un almacén en una aplicación React bajo Flux.
- Qué es un controlador de acciones en la arquitectura Flux.
- Dónde implementar los controladores de acciones en las aplicaciones React bajo Flux.
- Qué son las operaciones de acceso.
- Qué es el evento de cambio.
- Cuándo y cómo generar el evento de cambio.

Introducción

Un almacén (*store*) es un objeto que interactúa con el contenedor de datos de la aplicación, generalmente, una base de datos o un servicio web REST. Toda aplicación debe tener un almacén o más. Generalmente, se crea un almacén para representar o acceder a un determinado dominio de datos de la aplicación. De manera muy parecida a los espacios de nombres o esquemas de las bases de datos. Pero si lo deseamos, podemos implementar toda la lógica de acceso a los datos en un único almacén.

Los almacenes representan la capa de datos, en la aplicación React. Son los responsables del acceso a los datos. Ya sea leyéndolos y/o escribiéndolos en los orígenes de datos correspondientes.

Atendiendo al tipo de acceso que realice un almacén, se puede clasificar en de sólo lectura o persistente.



Un almacén de sólo lectura (*read-only store*) sólo accede al almacén en modo sólo lectura. Mientras que un almacén persistente (*persistent store*) es aquel que lo hace en L/E.

Los almacenes son los únicos objetos que pueden acceder a los datos. Por lo que cualquier dato, que deba ser presentado por un componente React, deberá proceder de alguno de los almacenes. Y

cualquier dato que deba escribirse en la base de datos, deberá dirigirse al almacén para que éste finalmente lleve a cabo el cambio.

Clase Store

El paquete flux viene con la clase Store que implementa un almacén base. Digamos que es la clase que, por convenio y buenas prácticas, debe de heredar directa o indirectamente todo almacén de nuestra aplicación React, si la diseñamos usando la arquitectura Flux. Por convenio y buenas prácticas, los almacenes se definen en la carpeta app/stores y llevan el sufijo Store como, por ejemplo, EmailStore.

La clase base Store se encuentra en el paquete flux/utils. He aquí una plantilla para ir abriendo boca:

```
import {Store} from "flux/utils";

class MyStore extends Store {
  //...
}
```

Esta clase dispone de varios miembros que hay que conocer. Ahora, vamos a presentar el método constructor y los demás a lo largo de la presente lección y de la siguiente.

Constructor de la clase

El constructor de la clase espera como argumento la instancia del despachador de la aplicación. En nuestro caso, pasaremos el objeto exportado por el archivo app/dispatacher/AppDispatcher.js.

```
class MyStore extends Store {
  constructor(dispatcher) {
    super(dispatcher);
  }
}
```

Este constructor lleva a cabo una tarea importante: registra el controlador de acciones del almacén en el despachador. Por lo que nunca hay que olvidar invocar al constructor base con la instancia del despachador.

Almacenes persistentes

Cada almacén se implementa mediante su propia clase, la cual dispone básicamente dos tipos de métodos:

- El controlador de acciones (*action handlers*), ya presentado en la lección anterior, que se encarga de gestionar las acciones del almacén.
- Las operaciones de acceso (*access operations*), aquellas que acceden a los datos de la aplicación.

  Por ejemplo, mediante este tipo de operaciones se lleva a cabo la inserción, actualización y supresión de registros de datos, así como la obtención de datos. Estos métodos suelen utilizar los paquetes http o request, el *framework* jQuery o las APIs Fetch o WebSocket.

Vamos a recordar cómo funciona el flujo de datos en la arquitectura Flux. El flujo lo comienza siempre una acción. Cada acción se genera mediante su creador de acción, el cual la crea y se la notifica al despachador. Entonces, el despachador recorre todos los controladores de acciones que tiene registrados, los cuales entonces analizan el objeto acción y, si están ante una acción que les concierne, la procesan. Tal como vamos a ver en breve, los controladores de acciones se definen en los almacenes y son los encargados de invocar las operaciones de acceso a datos. Una vez terminado el acceso, el mismo controlador de acción emite el evento de cambio sobre el almacén para que los componentes React se actualicen.

Controladores de acciones

Un controlador de acciones (*action handler*) es una método del almacén que se encarga de atender las acciones que se generan en la aplicación y procesarlas si son de incumbencia para el almacén. Recordemos que los controladores de acción se registran en el despachador, el cual los invoca uno a uno cuando la aplicación genera acciones mediante los creadores de acciones.

La signatura de estos controladores, tal como vimos en la lección anterior, la marca el despachador, pues es quien los invoca. Recordemos:

```
function handler(action)
```

Parámetro	Tipo de datos	Descripción
action	Object	Objeto acción: • type (string o number). Tipo de acción generada. • data (object). Datos asociados a esta acción particular.

Cuando se define un almacén mediante la clase Store del paquete flux, el método controlador debe ser __onDispatch(). Así pues, tendremos algo como:

```
class MyStore extends Store {
  /**
   * Constructor.
   *
   * @param dispatcher:Dispatcher  Despachador donde registrar el método controlador
   *                               de acciones.
   */
  constructor(dispatcher) {
    super(dispatcher);
  }

  /**
   * Método controlador de acciones.
   *
   * @override
   * @param action:object  Objeto acción.
   */
  __onDispatch(action) {
    //…
  }
}
```

Registro de controladores de acciones

Es importante recordar que los controladores de acciones deben registrarse en el despachador para que así puedan ser invocados cada vez que se genera una acción. Recordemos que esto se hace mediante el método register() del objeto despachador. Cuando se usa la clase Store del paquete flux, el registro lo realiza automáticamente el constructor de esta clase. Por lo que no debemos preocuparnos de hacerlo nosotros mismos, siempre que definamos su lógica en el método __onDispatch().

Responsabilidades de los controladores de acciones

Los controladores de acciones tienen una funcionalidad muy clara y bien definida:

1. Analizar el objeto acción y comprobar si debe ser atendida por el controlador. No debe hacer nada si no se trata de una acción de su interés.
2. Invocar la operación de acceso del almacén asociada a la acción para que así se lleve a cabo.

   Por ejemplo, si la acción tiene como objeto insertar un nuevo registro en la base de datos, el controlador debe invocar la operación del almacén que lo inserta, pasándole la propiedad data del objeto acción.
3. Una vez finalizada la operación de acceso, emitir el evento de cambio para que los componentes React que usan sus datos se actualicen.

He aquí un ejemplo ilustrativo:

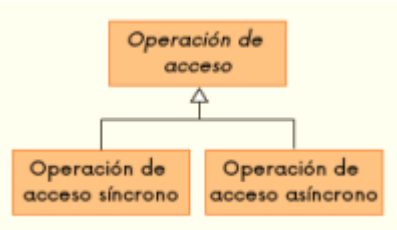
```
__onDispatch(action) {
  const type = action.type;
  const data = action.data;

  if (type == EmployeeAction.CREATE_EMPLOYEE) {
    this.createEmployee(data);
    if (this.hasChanged()) this.__emitChange();
```



```
    } else if (type == EmployeeAction.REMOVE_EMPLOYEE) {
      this.removeEmployee(data);
      if (this.hasChanged()) this.__emitChange();
    } else if (type == EmployeeAction.UPDATE_EMPLOYEE) {
      this.updateEmployee(data);
      if (this.hasChanged()) this.__emitChange();
    }
  }
```

Operaciones de acceso

Una operación de acceso (*access operation*) es el término con el que referirnos a un método del almacén que se encarga de realizar una operación contra su origen de datos. Atendiendo a cómo el método realiza su operación, se distingue entre operaciones síncronas y asíncronas.



Una operación de acceso síncrono (*synchronous access operation*) ejecuta su funcionalidad síncronamente. Esto quiere decir que cuando la operación finaliza su cuerpo, es porque la operación se ha llevado a cabo. Y hasta que no finaliza, la operación no termina. En cambio, una operación de acceso asíncrono (*asynchronous access operation*) la realiza asíncronamente y, por lo tanto, invocará una función parámetro cuando ha terminado para que el invocador pueda continuar con su flujo de ejecución. No son ningún misterio para los amantes de Node.

¿Por qué es importante cómo se invoca la operación? Porque el controlador de acciones implementará su funcionalidad de una manera u otra. Veamos.

Generalmente, cuando los datos se encuentran almacenados en la propia aplicación React, por ejemplo, en un objeto JavaScript, las operaciones son síncronas. En estos casos, el método controlador de acciones, primero, invocará la operación de acceso y, a continuación, emitirá el cambio. Ejemplo:

```
//operación de creación de empleado
createEmployee(data) {
  //crea el empleado en la base de datos síncronamente
}

//controlador de acciones
__onDispatch(action) {
  const type = action.type;
  const data = action.data;

  …
  else if (type == EmployeeAction.CREATE_EMPLOYEE)
    this.createEmployee(data);
    this.__emitChange();
  }
  …
}
```

Ahora bien, ¿qué ocurre si es asíncrono? ¿Se puede invocar el emisor de cambio nada más nos devuelva la operación el flujo de ejecución? La respuesta es un rotundo *no*. Porque no tenemos garantías de que la operación haya finalizado, ya que se ejecuta asíncronamente. En estos casos, la operación recibirá un parámetro adicional, conocido formalmente en Node como callback, el cual representa la función que debe ejecutar la operación cuando haya terminado. Esta función contendrá, pues, lo que debe ejecutarse cuando la operación de acceso haya terminado. Algo así como lo siguiente:

```
//operación de creación de empleado
createEmployee(data, callback) {
  //crea el empleado
  …

  //cuando ha terminado:
```



```
    callback();
  }

  //controlador de acciones
  __onDispatch(action) {
    const type = action.type;
    const data = action.data;

    …
    else if (type == EmployeeAction.CREATE_EMPLOYEE)
      this.createEmployee(data, (err) => {
        if (err) return alert(err);
        if (this.hasChanged()) this.__emitChange();
      });
    }
    …
  }
```

Las operaciones asíncronas también se pueden implementar mediante promesas. Para gustos, colores.

Evento de cambio

El evento de cambio (*change event*) se genera para notificar a los componentes React, esto es, a la interfaz de usuario, que se ha producido cambios en los datos que usan. El evento lo emite el almacén, más concretamente como acabamos de ver, su controlador de acciones, cuando ha finalizado la operación de acceso. Y claro está, siempre que se haya producido cambios en los datos; en otro caso, no hace falta.

Generación del evento de cambio

Para este fin, la clase Store implementa el método __emitChange(), el cual debe invocar el controlador de acciones del almacén cada vez que finalice una operación de acceso que haya generado cambios. Su signatura es muy sencilla:

```
__emitChange()
```

Este método sólo se puede invocar si la aplicación se encuentra atendiendo, en ese mismo instante, una acción.

He aquí un ejemplo de invocación tras la ejecución de una operación asíncrona, lo más común:

```
__onDispatch(action) {
  const type = action.type;
  const data = action.data;

  …
  else if (type == EmployeeAction.CREATE_EMPLOYEE)
    this.createEmployee(data, (err) => {
      if (err) alert(err);
      else if (this.hasChanged()) this.__emitChange();
    });
  }
  …
}
```

Método Store.hasChanged()

La clase Store tiene el método hasChanged() con el cual saber si la última acción has realizado cambios en el almacén. No siempre tienen que hacerlo, por ejemplo, una que ha desembocado en un error no hará ningún cambio. Este método se utiliza principalmente tras una operación de acceso de escritura para saber si se ha producido algún cambio y, entonces, el controlador de acciones debe generar el evento de cambio:

```
hasChanged() : boolean
```

Este método hay que sobrescribirlo en cada almacén. El almacén debe registrar, en algún atributo privado, si la última operación de escritura realizó algún cambio. Y entonces, este método devolver su valor. Cada vez que una operación de acceso de escritura finaliza, debe modificar este atributo. Por convenio y buenas prácticas, se utiliza el atributo changed. Haciendo que la sobrescritura del método se reduzca a algo tan sencillo como:


```
hasChanged() {
  return this.changed;
}
```

Generador de Justo

El generador de Justo proporciona el comando flux store para crear nuevos almacenes en la aplicación:

```
justo -g react flux store
```

Crea el archivo del almacén en la carpeta app/stores con una definición del almacén que hereda la clase Store y:

- Contiene una implementación básica del método controlador de acciones __onDispatch().
- La redefinido necesaria del método hasChanged().
- La definición del atributo privado changed que deben modificar las operaciones de acceso de escritura, cada vez que terminan su funcionalidad.

El módulo devuelve una instancia del almacén que debe utilizar la aplicación.

