upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

DESPLIEGUE DE SERVICIOS MULTIMEDIA
Máster Universitario en Ingeniería de Telecomunicación

AppGaztaroa

Commit 09: "Redux y Thunk"

Marko Galarza Galarza
Mikel Sagues García

Redux y Thunk

Commit 09: "Redux y Thunk"

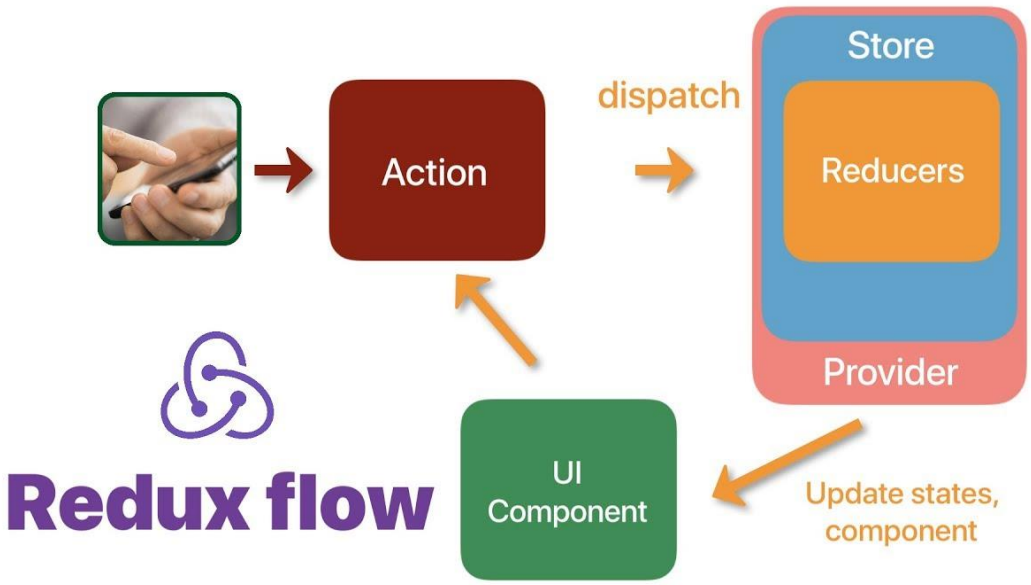
Antes de continuar avanzando en el desarrollo de nuestra aplicación, añadimos un elemento que podría entenderse como una complicación en un entorno tan sencillo como el que vamos a manejar en esta asignatura, pero que nos será de gran ayuda para tener ordenada la aplicación: Redux.

Redux es una implementación de la arquitectura Flux[1]. Redux nos "obliga" a trabajar en una implementación unidireccional, lo que, a la postre, aportará predictibilidad y trazabilidad a la aplicación, haciendo más sencillo cualquier modificación que queramos implementar en la misma.

Redux se basa en tres principios fundamentales:

1. Sólo hay una fuente de verdad.
2. El Estado es de solo lectura.
3. Los cambios se realizan con funciones puras.

La siguiente figura describe gráficamente la arquitectura de trabajo Redux[2] en la que, como podemos observar, el flujo de trabajo es unidireccional:



Además de Redux, emplearemos el middleware Redux-Thunk, lo que nos permitirá realizar llamadas asíncronas a nuestra base de datos.

[1] Hay cierta controversia sobre este hecho al añadir Redux ciertos elementos que no se definen en Flux. En todo caso, los propios creadores de Flux consideran Redux una implementación de esta arquitectura.
[2] https://react-redux.js.org/introduction/getting-started

1. Configurando Redux en nuestra aplicación

En este apartado vamos a configurar el repositorio Redux que emplearemos a partir de ahora en nuestra aplicación, dejando para el siguiente apartado la refactorización de la aplicación para conectarla a dicho repositorio.

Como ya se ha comentado más arriba, además de Redux emplearemos el middleware Thunk, que se instala por defecto con el paquete *reduxjs/toolkit*. Por lo tanto, el primer paso será instalar todas estas librerías en la carpeta principal de la aplicación:

npx expo install @reduxjs/toolkit react-redux

Una vez instaladas las librerías, creamos una carpeta de nombre *redux* en el directorio principal de la aplicación[3]. En dicho directorio descargaremos los ficheros JavaScript que se adjuntan a la práctica y que se describen brevemente a continuación. De este modo, todos los ficheros que tengan que ver con Redux estarán ordenados en una misma carpeta.

En nuestro caso, vamos a crear un repositorio diferente para cada uno de los tipos de datos que queremos almacenar (excursiones, cabecera, comentarios y actividades), y después emplearemos la función *combineReducers* para juntarlos todos. Podríamos haber concentrado toda la información en un único *reducer*, pero de este modo tenemos una organización más limpia. En todo caso, es cierto que volvemos a complicar un poquito nuestra implementación, por motivos "didácticos", ya que esto no estaría justificado para el volumen y complejidad de datos que manejamos.

Configurar el Store

La función *ConfigureStore()* definida en el fichero *configureStore.js* será la responsable de crear el store de nuestro Redux y de combinar los cuatro *reducers*. Además, también asociará a dicho store el *middleware* que emplearemos en nuestro Redux: Thunk.

```
import { configureStore } from '@reduxjs/toolkit'
import thunk from 'redux-thunk';
[...]

export const ConfigureStore = () => {
    const store = configureStore({
        reducer: {
            excursiones: excursiones,
            comentarios: comentarios,
            cabeceras: cabeceras,
            actividades: actividades,
        },
```

[3] Recordad, como siempre, respetar las mayúsculas y minúsculas en los nombres que asignamos a cada fichero, componente, variable, función, etc.

```
    });

    return store;
}
```

Action Types (tipos de acciones)

En el fichero *ActionTypes.js* se definirán todos los tipos de acciones asociados a nuestro repositorio Redux. Definiremos varios tipos de acciones para cada uno de los *reducers* que componen nuestro **store** (excursiones, cabecera, comentarios y actividades), de tal forma que dispongamos de diferentes tipos de acciones para: añadir información (ADD), identificar si la información está siendo cargada (LOADING) o si la operación no se ha podido completar (FAILED). En la siguiente figura se muestra un ejemplo, para el caso de las excursiones.

```
export const EXCURSIONES_LOADING = 'EXCURSIONES_LOADING';
export const ADD_EXCURSIONES = 'ADD_EXCURSIONES';
export const EXCURSIONES_FAILED = 'EXCURSIONES_FAILED';
```

Reducers

A continuación, construiremos cada uno de los cuatro repositorios Redux que darán soporte a la aplicación (*excursiones.js, cabeceras.js, actividades.js* y *comentarios.js*). Recordemos que la función de los *reducers* es modificar el estado de nuestra aplicación, a partir de la acción que se realice. Dichos *reducers* se construyen empleando un procedimiento estándar, en el que se identifica el tipo de acción a la que se pretende dar respuesta, mediante un *CASE*, para posteriormente actualizar y devolver el estado de la aplicación. Por ejemplo, el código fuente del fichero *excursiones.js* es el que se muestra a continuación:

```
import * as ActionTypes from './ActionTypes';

export const excursiones = (state = { isLoading: true,
                                errMess: null,
                                excursiones:[]}, action) => {
    switch (action.type) {
        case ActionTypes.ADD_EXCURSIONES:
            return {...state, isLoading: false, errMess: null, excursione
s: action.payload};

        case ActionTypes.EXCURSIONES_LOADING:
            return {...state, isLoading: true, errMess: null, excursiones
: []}

        case ActionTypes.EXCURSIONES_FAILED:
            return {...state, isLoading: false, errMess: action.payload};

        default:
          return state;
      }
};
```

Como vemos, mediante un *switch/case* se identifica el tipo de acción a realizar, para posteriormente devolver (*return)* el estado (*state*) con los valores apropiados en función del tipo de acción a realizar. En definitiva, el reducer es el encargado de actualizar el estado de la aplicación, a partir de la acción que recibe.

Action Creators

Finalmente, en el fichero *ActionCreators.js* implementaremos las funciones (*Thunk*) y las acciones que darán respuesta a los requerimientos de nuestra aplicación. Por ahora, lo único que vamos a hacer es crear las funciones que realizaran el "fetching" de los datos desde nuestra interfaz REST API. Dichas funciones tendrán una implementación basada en "*promises*" haciendo uso del dispatch().

```
export const fetchExcursiones = () => (dispatch) => {

    dispatch(excursionesLoading());

    return fetch(baseUrl + 'excursiones')
    .then(response => {
        if (response.ok) {
          return response;
        } else {
          var error = new Error('Error ' + response.status + ': ' + respo
nse.statusText);
          error.response = response;
          throw error;
        }
      },
      error => {
            var errmess = new Error(error.message);
            throw errmess;
      })
    .then(response => response.json())
    .then(excursiones => dispatch(addExcursiones(excursiones)))
    .catch(error => dispatch(excursionesFailed(error.message)));
};

export const excursionesLoading = () => ({
    type: ActionTypes.EXCURSIONES_LOADING
});

export const excursionesFailed = (errmess) => ({
    type: ActionTypes.EXCURSIONES_FAILED,
    payload: errmess
});

export const addExcursiones = (excursiones) => ({
    type: ActionTypes.ADD_EXCURSIONES,
    payload: excursiones
```

```
});
```

El código es muy similar para los cuatro tipos de datos, ya que, en todos los casos, se trata de obtener información del servidor, mediante una función de "*fetching"* que monitoriza posibles errores en dicha captura y responde en función de su estado (cargado, cargando o error de carga). Para ello, hace uso de tres *Actions* (funciones) cuyo "*type*" y "*payload*" varía en función del tipo de acción (ADD, LOADING o FAILED).

Estas funciones de "fetching" de datos deberán ser llamadas desde los componentes de nuestra aplicación, sustituyendo a las cargas de datos que hasta ahora veníamos haciendo en local (mediante *import*-s desde los ficheros JavaScript que habíamos almacenado en la carpeta *comun*). Esto será, precisamente, nuestro cometido en el segundo apartado de este ejercicio.

2. Refactorización de la aplicación

Cuando hablamos de refactorizar la aplicación, nos referimos a modificar la forma en la que ésta accede al estado de los diferentes componentes. Hasta ahora, cada componente era el responsable de mantener su estado, comunicando dicho estado a otros componentes en el momento en que los invocaba.

En una arquitectura Redux, el estado se almacena en un repositorio central a través del cual es accesible para los diferentes componentes que conforman la aplicación.

En consecuencia, una vez que hemos configurado nuestro repositorio Redux, es hora de conectar los componentes a dicho repositorio.

En primer lugar, cargamos en el *Store* (en el *state*) los datos desde la interfaz REST API empleando las funciones de *"fetching"* que ya hemos comentado. Esta operación se realiza en el fichero *CampoBaseComponent.js,* de tal forma que los datos estén disponibles inmediatamente después de cargarse la aplicación. Actualizamos el código de la siguiente manera:

```
[...]
import { connect } from 'react-redux';
import { fetchExcursiones, fetchComentarios, fetchCabeceras, fetchActivid
ades } from '../redux/ActionCreators';

const mapStateToProps = state => {
  return {
    excursiones: state.excursiones,
    comentarios: state.comentarios,
    cabeceras: state.cabeceras,
    actividades: state.actividades
  }
}

const mapDispatchToProps = dispatch => ({
  fetchExcursiones: () => dispatch(fetchExcursiones()),
  fetchComentarios: () => dispatch(fetchComentarios()),
  fetchCabeceras: () => dispatch(fetchCabeceras()),
  fetchActividades: () => dispatch(fetchActividades()),
})
[...]
class Campobase extends Component {

  componentDidMount() {
    this.props.fetchExcursiones();
    this.props.fetchComentarios();
    this.props.fetchCabeceras();
    this.props.fetchActividades();
```

```
  }
[...]
export default Campobase;
export default connect(mapStateToProps, mapDispatchToProps)(Campobase);
```

En la práctica, en este componente la función *mapStateToProps()* no necesita cargar todos los datos (en realidad no necesita ninguno), ya que no se emplean en este componente. Se añade el código para mayor claridad a la hora de entender el mecanismo a través del cual los componentes tienen acceso al repositorio (el estado de la aplicación) y la forma en que estos componentes están conectados a él mediante la función *connect()* de React-Redux.

Una vez que tenemos los datos cargados en nuestro estado, será necesario actualizar la forma en que cada uno de los componentes accede a ellos, de manera análoga a la que se muestra a continuación para el fichero *QuienesSomosComponent.js*:

```
[...]
import { ACTIVIDADES } from '../comun/actividades';
import { connect } from 'react-redux';

const mapStateToProps = state => {
    return {
      actividades: state.actividades
    }
  }

[...]
    constructor(props) {
        super(props);
        this.state = {
            actividades: ACTIVIDADES
        };
    }
[...]
                    data={this.state.actividades}
                    data={this.props.actividades.actividades}
[...]
export default QuienesSomos;
export default connect(mapStateToProps)(QuienesSomos);
```

Las claves serían las siguientes:

- Eliminamos de la cabecera del fichero todas las importaciones de datos que se hacían desde los ficheros JavaScript que habíamos almacenado en la carpeta *comun*.
- Dado que los componentes ya no necesitan almacenar su estado, debemos eliminar su constructor. La excepción será el componente *DetalleExcursion*, ya que todavía no hemos implementado ninguna forma alternativa para manejar el

array de *favoritos[]*. Por lo tanto, en este caso debemos mantener el constructor, pero almacenando en el estado únicamente la propiedad *favoritos[]*.

- El acceso a los datos ya no se hará a través del estado del componente, sino desde las propiedades (*props*) que hemos cargado mediante la función *mapStateToProps()* en cada uno de los componentes.

Por último, también debemos modificar el fichero *App.js* para conectarlo a nuestro repositorio Redux. Para ello, creamos el repositorio, empleando el componente *ConfigureStore* que hemos creado en el primer apartado del ejercicio y conectamos la aplicación al repositorio Redux mediante el componente *Provider* de *React-Redux*.

```
[...]
import { Provider } from 'react-redux';
import { ConfigureStore } from './redux/configureStore';

const store = ConfigureStore();

export default function App() {
  return(
  <Provider store={store}>
    <SafeAreaProvider>
      <View style={styles.container}>
        <Campobase/>
        <StatusBar style="auto" />
      </View>
    </SafeAreaProvider>
  </Provider>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
});
```

Una vez completado todo lo anterior, nuestra aplicación debería mostrar exactamente el mismo aspecto y funcionalidad del ejercicio anterior, ya que no hemos hecho ningún cambio en su interfaz gráfica. Podemos borrar los ficheros *javascript* de la carpeta *comun* a excepción de *comun.js*.

Si todo es correcto, ya podemos hacer el *commit* correspondiente a este capítulo y creando un documento con las claves del trabajo realizado. En este caso, vamos a establecer un requisito adicional para dicho documento: será necesario incorporar un diagrama de objetos/clases que nos permita seguir, gráficamente, el flujo de trabajo de nuestra aplicación y que presentaremos en clase en la fecha indicada por el profesor.

Se recomienda consultar los siguientes tutoriales para ver una posible forma en que se podría realizar dicho diagrama de clases[4], aunque se da absoluta libertad para elegir la forma en que se desee realizar dicha documentación.

3. Bibliografía

- **Flux**
  https://facebookarchive.github.io/flux/
  Un vídeo un poquito largo, pero que nos permite entender las motivaciones y filosofía que hay detrás de Flux:
  - https://www.youtube.com/watch?v=nYkdrAPrdcw
- **Redux**
  https://redux.js.org/
  https://redux-toolkit.js.org/introduction/getting-started
  Motivación para utilizar Redux:
  - https://redux.js.org/understanding/thinking-in-redux/motivation
  - https://redux.js.org/faq/general#when-should-i-use-redux

  Principios fundamentales:
  - https://redux.js.org/tutorials/fundamentals/part-1-overview
  - https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow
  - https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers
  - https://redux.js.org/tutorials/fundamentals/part-4-store

  Varios tutoriales. El primero de ellos del propio Dan Abramov, el creador de Redux:
  - https://egghead.io/courses/fundamentals-of-redux-course-from-dan-abramov-bd5cc867
  - https://medium.com/@debian789/implementaci%C3%B3n-de-redux-en-react-native-6324e2a8c4aa
  - https://www.knowledgehut.com/blog/web-development/redux-in-react-native
- **React-Redux**
  https://react-redux.js.org
  https://react-redux.js.org/introduction/getting-started
  Principales funciones que hemos empleado:
  - https://redux-toolkit.js.org/api/configureStore
  - https://react-redux.js.org/api/provider
  - https://react-redux.js.org/api/connect
  - https://react-redux.js.org/using-react-redux/connect-mapstate#connect-extracting-data-with-mapstatetoprops
  - https://react-redux.js.org/6.x/using-react-redux/connect-mapdispatch#connect-dispatching-actions-with-mapdispatchtoprops

[4] https://medium.com/@debian789/implementaci%C3%B3n-de-redux-en-react-native-6324e2a8c4aa
https://www.knowledgehut.com/blog/web-development/redux-in-react-native

- **Redux-Thunk**
  https://github.com/reduxjs/redux-thunk
  https://redux.js.org/usage/writing-logic-thunks
  https://redux.js.org/tutorials/fundamentals/part-4-store#middleware
- **React**
  https://reactjs.org/docs/react-component.html#componentdidmount
- **React Native**
  - Fetch
    https://reactnative.dev/docs/network
  - Promise
    https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve