upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

DESPLIEGUE DE SERVICIOS MULTIMEDIA
Máster Universitario en Ingeniería de Telecomunicación

AppGaztaroa

Commit 10: "Activity Indicator y addFavoritos"

Marko Galarza Galarza
Mikel Sagues García

Activity Indicator y addFavoritos

Commit 10: "Activity Indicator y addFavoritos"

En este ejercicio avanzaremos en nuestra aplicación, dotándola de la capacidad de modificar el estado del *Store* para almacenar las excursiones "favoritas" del usuario. Además, intentaremos mejorar la experiencia de usuario, añadiendo información sobre el estado de ejecución de la aplicación, mediante un sencillo indicador de actividad (*loader*).

1. Activity Indicator

Un *loader* es un elemento que indica al usuario que la aplicación está ocupada realizando alguna tarea. En React Native, esta funcionalidad nos la proporciona el componente *Activity Indicator*, integrado en React Native, por lo que no será necesario importar ninguna nueva librería para ello.

Comenzamos por crear un nuevo componente en nuestra aplicación, de nombre *IndicadorActividad* y que será definido en el fichero *IndicadorActividadComponent.js*, dentro de la carpeta *componentes*. El código de dicho fichero será el siguiente:

```
import React from 'react';
import { ActivityIndicator, StyleSheet, Text, View } from 'react-native'
import { colorGaztaroaOscuro } from '../comun/comun';

export const IndicadorActividad = () => {
    return(
        <View style={styles.indicadorView} >
            <ActivityIndicator size="large" color={colorGaztaroaOscuro} />
            <Text style={styles.indicadorText} >En proceso . . .</Text>
        </View>
    );
};

const styles = StyleSheet.create({
    indicadorView: {
      alignItems: 'center',
      justifyContent: 'center',
      flex: 1
    },
    indicadorText: {
      color: colorGaztaroaOscuro,
      fontSize: 14,
      fontWeight: 'bold'
    }
});
```

El código, como puede observarse, es muy sencillo, siendo el componente *ActivityIndicator* el encargado de proporcionar la funcionalidad que buscamos (nosotros únicamente nos tenemos que preocupar de la hoja de estilos).

Emplearemos este componente para mejorar la experiencia de usuario en aquellas funcionalidades de nuestra aplicación en la que sea muy probable que se produzcan retrasos significativos desde la solicitud de una acción, por parte del usuario, hasta que se complete su ejecución.

En nuestro caso, cualquier petición al servidor tendrá que ser acompañada de un indicador de actividad, ya que la resolución de dichas peticiones estará sujeta a posibles retardos motivados, entre otros, por la calidad de la red.

Nuestra aplicación está preparada para poder hacer frente a esta funcionalidad, al disponer de acciones que monitorizan el proceso de *fetching* desde la aplicación. A modo de recordatorio, se muestra:

- Los tres tipos de acciones asociadas a la carga de la información relativa a las excursiones (*ActionTypes.js*):

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

- Y la forma en que la aplicación almacena en su estado una variable booleana para determinar si las excursiones están siendo cargadas o no (reducer *excursiones* en el fichero *excursiones.js*):

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

A partir de lo anterior, es evidente que, con sólo consultar el estado de la variable ***isLoading***, seremos capaces de determinar si la carga está en proceso o ya ha terminado.

Por ejemplo, en el componente *Home* podemos implementar esta funcionalidad de la siguiente manera:

- En primer lugar, debemos consultar el estado de la variable antes de renderizar el contenido de nuestra vista. En concreto, para el caso de las excursiones, esto puede hacerse dentro del *ScrollView* en el que se ejecuta la función *RenderItem*:

```
<RenderItem item={this.props.excursiones.excursiones.filter((excursion) =
> excursion.destacado)[0]}
    isLoading={this.props.excursiones.isLoading}
    errMess={this.props.excursiones.errMess}
/>
```

- Nótese que hemos "aprovechado el viaje" para consultar también el estado de la variable *errMess*, que nos permitirá saber si se ha producido un error en el proceso de carga.
- Una vez obtenido el valor de dichas variables desde el Store, actuamos en concordancia dentro de la función de renderizado. Por lo tanto, tendremos tres posibilidades, quedando como sigue:

```
function RenderItem(props) {

    const item = props.item;

    if (props.isLoading) {
        return(
            <IndicadorActividad />
        );
    }

    else if (props.errMess) {
        return(
            <View>
                <Text>{props.errMess}</Text>
            </View>
        );
    }

    else {

        const item = props.item;

        if (item != null) {
            return(
                <Card>
[...]
                </Card>
            );
        }
        else {
            return(<View></View>);
```

```
        }
    }
}
```

A continuación, repetiremos este mismo proceso para los componentes[1]:

- *Calendario*
  - En este caso, integraremos el indicador de actividad dentro del "cuerpo" del componente, de tal forma que el *Flatlist* se ejecute únicamente cuando se hayan cargados los datos.
- *QuienesSomos*
  - En este caso, integraremos el indicador de actividad dentro de un *Card*, de tal forma que *Historia* se muestre, aunque todavía no se haya cargado el contenido desde el servidor. En concreto, para el caso de que el contenido todavía se esté cargando:

```
        if (this.props.actividades.isLoading) {
            return(
                <ScrollView>
                    <Historia />
                    <Card>
                        <Card.Title>"Actividades y recursos"</Card.Title>
                        <Card.Divider/>
                        <IndicadorActividad />
                    </Card>
                </ScrollView>
            );
        }
```

2. Guardar Favoritos en el Store

En este segundo apartado completaremos la funcionalidad de marcar excursiones como "favoritas" que habíamos empezado a construir en un ejercicio anterior. Para ello, crearemos un nuevo reducer, con una acción asociada, a través del cual modificaremos el estado de nuestra aplicación para incorporar la información sobre las excursiones favoritas de nuestro usuario.

Comenzamos definiendo dos nuevos tipos de acciones en *ActionTypes.js*:

```
export const POST_FAVORITO = 'POST_FAVORITO';
export const ADD_FAVORITO = 'ADD_FAVORITO';
```

[1] Es cierto que, en estos momentos, dado que el fetching se realiza desde el componente *Campobase* al iniciar la aplicación, en estos componentes no veremos el indicador de actividad, salvo que modifiquemos la pantalla inicial en el menú o traslademos las cargas de datos a estos componentes. Sí que nos serán de utilidad los mensajes de error, en caso de que la interfaz REST API haya fallado.

El primer tipo de acción (POST_FAVORITO) se empleará para subir la información al servidor, mientras que el segundo tipo (ADD_FAVORITO) será ejecutada una vez completada la primera. Este es un procedimiento estándar, por lo que mantendremos ese "esquema" aunque, en la práctica, nuestra aplicación, por el momento, no va a modificar los datos en el servidor.

A continuación, creamos un nuevo *reducer*, de nombre *favoritos* en un fichero *favoritos.js* dentro de la carpeta *redux* de nuestra aplicación. El "esqueleto" de dicho fichero será el siguiente:

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

export const favoritos = (state = {favoritos: []}, action) => {
    switch (action.type) {
        case ActionTypes.ADD_FAVORITO:
[...]

        default:
          return state;
      }
};
```

A la hora de completar el código, tened en cuenta que:

- No se implementa ninguna respuesta para el caso en que la acción sea de tipo POST_FAVORITO. Por ahora, únicamente implementaremos la respuesta a ADD_FAVORITO.
- Nuestro *reducer* espera recibir como parámetro una acción que incluya un *payload*. Este *payload* será el parámetro *excursionId* que identifica una determinada excursión.
- En primer lugar, el *reducer* comprobará que dicha excursión no se encuentra ya entre los favoritos, devolviendo el estado sin ninguna modificación en caso de que lo esté. En caso contrario, incorporará dicho *excursionId* al array *favoritos* del estado.

En el fichero *ActionCreators.js* añadimos las siguientes funciones:

```
export const postFavorito = (excursionId)  => (dispatch) => {
    setTimeout(() => {
        dispatch(addFavorito(excursionId));
    }, 2000);
};

export const addFavorito = (excursionId) => ({
    type: ActionTypes.ADD_FAVORITO,
    payload: excursionId
});
```

Nótese que:

- La función *postFavorito()* (la cual es una función *Thunk*) se limita a introducir un retardo de dos segundos, para "simular" la comunicación con el servidor, para, a continuación, despachar la función *addFavorito* que será la encargada de devolver la acción de tipo ADD_FAVORITO.
- El parámetro con el que trabajan estas funciones es *excursiónId*, el cual finalmente constituye el *payload* de la acción que trasladaremos a nuestro *reducer*, tal y como habíamos comentado más arriba.

A continuación, debemos modificar el componente *DetalleExcursion* de tal forma que *favoritos* deje de ser una variable de estado "local" (del componente), empleando la variable que acabamos de crear mediante el nuevo *reducer*. Para ello:

- Importamos la variable *favoritos* desde el estado mediante la función *mapStateToProps*.
- Eliminamos el constructor del componente *DetalleExcursion*, ya que este no es necesario ahora que vamos a guardar el array en el estado global de la aplicación.
- Modificamos la función *marcarFavorito*:

```
  marcarFavorito(excursionId) {
      this.setState({favoritos: this.state.favoritos.concat(excursionId)});
      this.props.postFavorito(excursionId);
  }
```

- Creamos el método *mapDispatchToProps* para poder lanzar la función *Thunk* que hemos creado en *ActionCreators.js*:

```
const mapDispatchToProps = dispatch => ({
    postFavorito: (excursionId) => dispatch(postFavorito(excursionId))
})
```

- Actualizamos la exportación del componente a través de *connect()* para completar el proceso anterior.
- Actualizamos los parámetros de *RenderExcursion*:

```
        <RenderExcursion
            excursion={this.props.excursiones.excursiones[+excursionId]}
            favorita={this.state.favoritos.some(el => el === excursionId)}
            favorita={this.props.favoritos.favoritos.some(el => el === exc
ursionId)}
            onPress={() => this.marcarFavorito(excursionId)}
        />
```

Finalmente, no debemos olvidar actualizar el componente *ConfigureStore* para integrar este nuevo *reducer* en el Store.

Una vez completado todo lo anterior, nuestra aplicación debería mostrar la funcionalidad que se describe en el vídeo adjunto a la práctica. Nótese que todavía no hemos implementado la funcionalidad de "desmarcar" un favorito, por lo que éstos quedarán en la lista de favoritos hasta que salgamos de la aplicación. Tampoco hemos establecido ningún mecanismo para que los favoritos se almacenen de manera permanente en memoria del dispositivo ni en el servidor. En consecuencia, al reiniciar la aplicación la lista de excursiones favoritas volverá a estar vacía.

Como siempre, si todo es correcto, ya podemos hacer el *commit* correspondiente a este ejercicio incluyendo una breve descripción con las claves del trabajo realizado.

3. Bibliografía

- **Activity Indicator**
  https://reactnative.dev/docs/activityindicator