Librería Redux-Thunk

React Asincrónico

Introducción

Como ya sabemos, ReactJS es una librería para el desarrollo de UI que se utiliza justamente en el Frontend de los proyectos. A menos que la página, aplicación o plataforma que estemos haciendo no depende de ningún dato de servidor, lo más común es que siempre necesitemos obtener información externa para poder poblar a nuestros componentes.
Si estamos haciendo una pagina de noticias tendremos que obtenerlas de una API o servicio web que nos la provea. Si estamos haciendo un login, lógicamente tendremos que pasarle las credenciales a un servicio para poder obtener un token de acceso.
Muchos son los posibles casos de uso que podemos encontrar, y queda claro que es un tema delicado que tenemos que incorporar.

Utilizar Fetch en componentes de React

Los componentes de clase de ReactJS vienen programados para que el desarrollador tenga la posibilidad de realizar acciones durante las diferentes etapas de vida de los mismos. Entre estos diferentes momentos se encuentra aquel en el cual el componente ya existe en el DOM, su estado ya fue inicializado y se encuentra disponible. Este es el método componentDidMount, y es el que vamos a utilizar **siempre** para pedir la información inicial de nuestros componentes.

Supongamos que estamos programando un componente el cual renderiza un listado de noticias.

[code]

```
export default class News extends Component {
  state = {
    news: []
  };

  render() {
    const { news } = this.state;
    return (
      <div className="News">
        {news.map(({ title, author, description, urlToImage: image }) => (
          <NewsItem
            title={title}
            author={author}
            description={description}
            image={image}
          />
        ))}
      </div>
    );
  }
}
```

[/code]

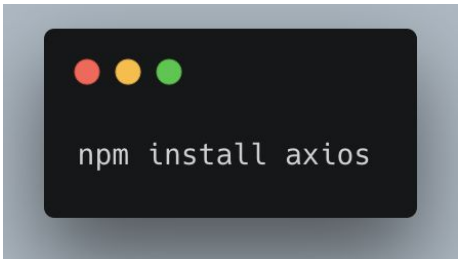
Por supuesto que en este mismo archivo tenemos que importar React y los componentes y estilos que necesitemos.
Ahora para obtener la información, es decir las noticias, podemos utilizar una API externa, como por ejemplo https://newsapi.org . La misma nos permite sacar un api key gratis y facil y empezar a utilizarla.
Agregamos entonces un método componentDidMount donde realizar la llamada con fetch:

[code]
```
//Antes está el state inicializado...

componentDidMount() {
   fetch(NEWS_API)
     .then(response => response.json())
     .then(news =>
       this.setState({
         news: news.articles
       })
     );
 }


//Después viene el render...
```
[/code]

NEWS_API es una constante que estamos importando de otro archivo que contiene la URL para realizar la llamada. Ahora nuestro componente obtiene su información de un Backend que la provee.

Librería Axios

Axios es una librería Javascript para realizar llamadas AJAX. Se puede instalar en nuestros proyectos fácilmente con NPM:



Un ejemplo básico con axios seria si queremos hacer una llamada GET para obtener información de un endpoint:

[code]

```
axios.get(url).then(response => console.log(response));
```

[/code]

Axios vs Fetch

Una de las primeras cosas que podemos pensar cuando leemos sobre axios es porque utilizar una librería cuando por defecto Javascript trae la API de fetch? Aca hay algunas razones:

1. Cuando utilizamos la API de fetch la primer respuesta que nos devuelve es un paso intermedio cuando los headers del response son recibidos, es por eso que siempre a la primera promesa de fetch le hacemos un .json() o el método de parseo que nos convenga y luego capturamos esa segunda promesa:

   [code]
   fetch(url).then(response => response.json()).then(data => console.log(data));
   [/code]

   Axios evita que yo tenga que realizar ese paso automatizando también el tipo de parseo que necesito.
2. En cuanto a manejo de errores, la API de fetch también suele tener sus problemas. Axios logra que cualquier respuesta HTTP con un código de status incorrecto siempre salga como una excepción y podamos capturar correctamente con el método .catch():

   [code]
   axios.get(url)
   .catch(error => console.log('BAD', error))
   .then(response => console.log('GOOD', response));
   [/code]

Podemos ver más información sobre lo que nos ofrece axios en su repositorio de github: https://github.com/axios/axios#features

Axios en nuestro componente

En este caso, lo único que necesitamos cambiar seria el metodo componentDidMount en donde hicimos el pedido de la información

[code]
//Antes está el state inicializado...

```
componentDidMount() {
   axios.get(NEWS_API).then(({ data: news }) =>
     this.setState({
       news: news.articles
     })
   );
 }


//Después viene el render...
```

[/code]

Método POST con axios.

Axios soporta los distintos métodos https que existen. Si queremos utilizar el método POST por ejemplo podemos utilizar la siguiente sintaxis:

[code]
axios.post(url, data).then(response => console.log(response.data));
[/code]

Redux middlewares

Introducción

Los middlewares en Redux siguen la misma filosofía que los middlewares en cualquier otro tipo de sistema. Simbolizan una forma de interceptar el código, logrando extenderlo y agregarle funcionalidad entre el momento en que algo se recibe y algo de resuelve.
Específicamente hablando de Redux, un middleware sirve para realizar o ejecutar un código en el momento intermedio entre que una acción es despachada y la misma llega al reducer.
Una de los puntos muy positivos que tiene trabajar con middlewares es que funcionan en forma de cadena, unos tras otros, realizando cada uno su acción pertinente.

Casos de uso

Supongamos por ejemplo que queremos loguear todas las acciones que llegan al store, para poder evaluar en desarrollo que está sucediendo. En este caso, una primera aproximación podría ser poner console.log en todas las acciones que realizamos, lo cual no sería para nada bueno.
Pensemos por un segundo que necesitamos, para poder controlar animaciones, que una acción que llega al store se retrase N milisegundos, que no se despacha inmediatamente. Tendríamos que poner un setTimeout en el action creator (que no es lo deseable).
Pensemos por un segundo que necesitamos que una acción resuelva de forma asincrónica, como sería el caso de que necesitemos pedir información a una API externa.
Todos estos ejemplos y muchisimos mas son casos en los cuales necesitamos de alguna forma capturar esas acciones y actuar de formas distintas para poder lograr el objetivo.
Para estos casos existen los middlewares.

Implementar middlewares en redux

al momento de crear un store, podemos utilizar una función que nos provee redux que es applyMiddlewares. Es una función que recibe como parámetro un listado de middlewares y la cual se utiliza al momento de crear el store
primero importamos la función:
[code]
import { createStore, applyMiddleware } from 'redux'
[/code]

Luego cuando creamos el store la utilizamos:
[code]
const store = createStore(todos, ['Use Redux'], applyMiddleware(customMiddleware))
[/code]

Pero que sería en este caso “customMiddleware”?

Creando middlewares para redux

Un middleware es básicamente una funciona con la siguiente forma:
[code]

```
const customMiddleware = store => next => action => {
  //Código a ejecutar
}
```

[/code]

Es decir, es una función que recibe el store y que devuelve una funciona que recibe next y devuelve una función que recibe la acción.
Suena todo como un trabalengua. Analicemoslo un poco:

- "Es una función que recibe el store": Hasta acá nada que no podamos comprender
- "Devuelve una función que recibe next": La primera devolución tiene que ser una función que recibe un parámetro llamada next. Next es una función que simboliza de forma genérica el siguiente paso en la cadena de middlewares. Lógicamente, si solo tenemos un middleware entonces next despachará definitivamente la acción al reducer.
- "Devuelve una función que recibe la acción": Es la función final de vuelta que recibe la acción con la cual vamos a poder trabajar.

En el código a ejecutar podemos hacer lo que nos sea conveniente y en última instancia determinar parar la ejecución de la acción, definir qué hacer si se cancela o decidir que siga la cadena invocando a next(action).

Redux-thunk

Redux-thunk es un middleware de redux que sirva para poder despachar funciones en vez de acciones planas en JSON. Esto nos dará la posibilidad de manejar a nuestros antojo y con facilidad efecto secundarios que necesitemos en nuestra aplicación, cómo retrasar una acción o realizar llamadas asíncronas las cuales tienen que esperar a que finalicen para despachar realmente la acción.

Incluirlo en redux

Primero que nada necesitamos instalar el paquete por NPM:

Una vez que lo tengamos, solo necesitamos importarlo y asignarlo como un middleware de redux:

[code]

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './rootReducer';

const store = createStore(
  rootReducer,
  applyMiddleware(thunk)
);
```

[/code]

Y solo con esto ya podemos comenzar a utilizarlo.

Firma de las funciones de thunk

Las funciones que podemos despachar tienen una firma, o parámetros específicos que podemos utilizar:

[code]
const thunkFunction = () => (dispatch) => {}
[/code]

Es decir, son funciones que reciben los parámetros que necesiten y que tienen que devolver otra función que reciba dispatch como primer parámetro.