TEMA 2

Programación reactiva:

RxJS + NgRX

Desarrollo front-end avanzado

Máster Universitario en Desarrollo de sitios y aplicaciones web

Universitat Oberta de Catalunya

Contenido

- Programación reactiva usando RxJS
- Fundamentos del patrón Redux
- NgRX: estados, acciones, reducers, effects y selectores

Programación reactiva usando RxJS

Tal y como ya comentamos en la primera práctica de esta asignatura, la programación reactiva es un paradigma centrado en los *streams* de datos de manera asíncrona. Por lo tanto, el primer aspecto a considerar en este paradigma es conocer qué es un *stream.* Un *stream* es una estructura de datos similar a la que podemos encontrar en cualquier lenguaje de programación (*array*, tablas hash, etc.), pero con la diferencia de que no disponemos de la información sobre el mismo de manera asíncrona, sino que desconocemos su tamaño y sus propios datos.

En nuestro caso concreto, desarrollo web, podemos encontrar *streams* generados a partir de las siguientes fuentes:

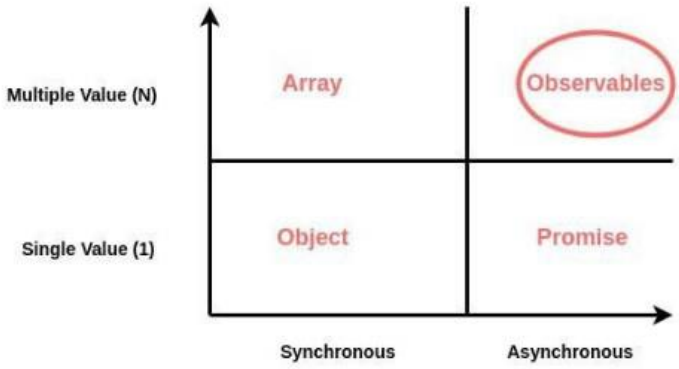
- Eventos DOM (ratón, teclado, formulario, ventana).
- Peticiones HTTP (con un servidor).
- WebSockets.
- Lecturas de ficheros o bases de datos.
- Animaciones.

La biblioteca que nos permite trabajar con este paradigma en el mundo web es RxJS (implementación Rx en JavaScript). Esta biblioteca está construida sobre dos patrones de diseño: 1) Observable e 2) Iterador. La biblioteca RxJS es un port de la original extensión reactiva (Rx desarrollada por Microsoft). Por lo tanto, se encuentran implementaciones en diferentes lenguajes como son RxJava, RxNet, etc.

La biblioteca RxJS trabaja sobre un tipo de datos básico denominado **observable** el cual representa en sí el *stream* de datos. No obstante, existen otros interesantes tipos que complementan a éste, tales como observer, schedulers, subjects.

Otro de los puntos fundamentales que han hecho de RxJS una biblioteca ampliamente utilizada, son su amplio conjunto de operadores que permiten manipular los *streams* de datos antes de que un observador se subscriba a los cambios de los *streams*. Algunos de los operadores más famosos están inspirados en los propios de *array* tales como map, reduce o filter, pero tratarán la información de manera asíncrona.

En la siguiente imagen vemos el uso de cada una de las técnicas/recursos de programación que se pueden utilizar para tratar estructuras de datos en el lenguaje JavaScript, y nos encontramos que los *observables* serían la herramienta perfecta para trabajar con secuencias de datos que son asíncronos:



La mejor documentación para arrancar con esta potente biblioteca proviene de la página oficial de la misma:

https://rxjs-dev.firebaseapp.com/guide/overview.

Los conceptos fundamentales de esta biblioteca se basan en los siguientes puntos:

- **Observable:** Representa la idea de una colección invocable de valores o eventos futuros.
- **Observer:** Es una colección de *callbacks* que saben cómo escuchar los valores entregados por el Observable.
- **Subscription:** Representa la ejecución de un Observable, es principalmente utilizada para cancelar la ejecución de un Observable.
- **Operators:** Son funciones puras que permiten desarrollar siguiendo el estilo de programación funcional. Son utilizadas normalmente para modificar el *stream*.
- **Subject:** Es el equivalente a un EventEmmiter, y el único modo de *multicasting* de un valor o evento a múltiples Observer.
- **Schedulers*:*** Son *dispatchers* centralizados para controlar la concurrencia.

En el siguiente enlace se muestran algunos ejemplos utilizando la tradicional manera con *event listeners* frente a la manera *reactiva:*

https://rxjs-dev.firebaseapp.com/guide/overview#first-examples

Para hacer uso de la biblioteca RxJS sólo basta con enlazar el CDN con la última versión de tal manera que lo único que habría que hacer es añadir:

```
<script src="https://unpkg.com/@reactivex/rxjs@6.4.0/dist/global/rxjs.umd.js"></script>
```

NOTA: La biblioteca RxJS 5.5 ha modificado la cadena de operadores a operadores combinables. De este modo, cuando en un ejemplo se encuentre con operadores encadenados del siguiente modo:

```
observable$
  .map(método1)
 .filter(método2)
 .scan(método3)
```

La nueva nomenclatura es utilizando el operador pipe de la siguiente manera:

```
observable$.pipe(
                map(método1),
                filter(método2),
                scan(método3)
      )
```

Por lo tanto, como resumen, sabemos que la programación reactiva es un paradigma de programación asíncrona que trata sobre los flujos de datos y la propagación de cambios. RxJS es una librería que permite realizar la programación reactiva usando los Observables, lo que hace más sencillo componer código asíncrono.

RxJS provee de un conjunto de funciones de utilidad que permiten crear y trabajar con Observables. Estas funciones de utilidad pueden usarse para:

- Convertir código existente para operaciones asíncronas en observables
- Iterar a través de valores de flujo
- Mapear valores a diferentes tipos
- Filtrar flujos
- Componer múltiples flujos

Esta biblioteca es muy extensa, pero a continuación se muestran las características y funciones que pueden ser más relevantes:

Operadores

Los operadores son funciones que utilizan como base los Observables, con el fin de realizar manipulaciones complejas al flujo de eventos que generan los Observables.

Los operadores se clasifican según su función.

- Operadores de creación
- Operadores de transformación
- Operadores de filtrado
- Operadores de combinación
- Operadores de utilidad
- Operadores de errores
- Operadores matemáticos

Operadores de creación

Este tipo de operador crea un nuevo observable.

➔ create

Convierte una función en un Observable.

```
const observable = Observable.create(function(observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
});

observable.subscribe(
  value => console.log(value),
  err => {},
  () => console.log("this is the end")
);
```

➔ from

Convierte un **array**, un **iterable** o una **promesa** en una secuencia observable.

```
const arraySource = from([1, 2, 3, 4, 5]);

//output: 1,2,3,4,5
const subscribe = arraySource.subscribe(val =>
console.log(val));
```

➔ of

Convierte una lista de valores en una secuencia observable.

```
const source = of(1, 2, 3, 4, 5);

//output: 1,2,3,4,5
const subscribe = source.subscribe(val => console.log(val));
```

➔ interval

Emite números consecutivos en intervalos.

```
const source = interval(1000); //in ms

//output: 0,1,2,3,4,5....
const subscribe = source.subscribe(val => console.log(val));
```

➔ timer

Tiene dos posibles comportamientos:

- Emite un ítem especificado (o si no, una consecución de números) tras esperar un tiempo.

```
const source = timer(1000);

//output: 0
const subscribe = source.subscribe(val => console.log(val));
```

- Una mezcla entre timer e Interval que permite crear un intervalo tras un tiempo.

```
//Wait 1 second, and repeat every 4 seconds
const source = timer(1000, 4000);

//output: 0,1,2,3....
const subscribe = source.subscribe(val => console.log(val));
```

Operadores de transformación

Transforman un origen/conjunto de datos en un Observable.

➔ map

Aplica una proyección a cada uno de los elementos de un origen de datos (array, str).

```
const source = from([1, 7, 4, 2, 5]);
//add 3 to each value
const example = source.pipe(map(val => val + 3));
const subscribe = example.subscribe(val => console.log(val));
```

➔ **reduce**

Convierte los valores de un origen observable a un único valor, que es emitido cuando **todos** los orígenes son completados.

```
// RxJS v6+
import { of } from 'rxjs';
import { reduce } from 'rxjs/operators';

const source = of(1, 2, 3, 4);
const example = source.pipe(reduce((acc, val) => acc + val));
//output: Sum: 10'
const subscribe = example.subscribe(val => console.log('Sum:',
val));
```

Operadores de filtrado

Estos operadores permiten desechar los datos provenientes del flujo que no satisfagan ciertas condiciones. Esto nos permite reducir lógica imperativa en la que se suelen hacer comprobaciones utilizando la estructura de control if-else debido a que, el dato puede ser desechado directamente utilizando alguno de los operadores de esta família.

➔ **filter**

Emite los valores que cumplan una condición asignada.

```
const source = from([1, 2, 3, 4, 5]);
//filter out non-even numbers
const example = source.pipe(filter(num => num % 2 === 0));
```

➔ debounceTime

Descarta los valores que fueron **emitidos antes** del tiempo especificado **antes de la salida**.

```
// RxJS v6+
import { fromEvent, timer } from 'rxjs';
import { debounceTime, map } from 'rxjs/operators';

const input = document.getElementById('example');


//for every keyup, map to current input value
const example = fromEvent(input, 'keyup').pipe(map(i =>
i.currentTarget.value));

//wait .5s between keyups to emit current value
//throw away all other values
const debouncedInput = example.pipe(debounceTime(500));

//log values
const subscribe = debouncedInput.subscribe(val => {
  console.log(`Debounced Input: ${val}`);
});
```

➔ take

En un flujo de datos no se conoce a priori el número total de elementos que pueden ser emitidos desde el elemento observado. No obstante, a veces es necesario cerrar los flujos (observables) bien por un espacio de tiempo, o por un número de elementos. El operador take permite definir el número de datos que se escucharán desde el observable antes de cerrar la subscripción.

En el siguiente ejemplo se muestra que el observable solamente toma el primer valor del flujo, el resto del flujo no es tenido en cuenta, debido a que el flujo se ha cerrado.

```
const source = of(5, 12, 7, 1, 2);
//take the first emitted value then complete
const example = source.pipe(take(1));
//output: 5
const subscribe = example.subscribe(val => console.log(val));
```

➔ takeUntil

Devuelve los valores de los observables de origen **hasta** que un origen especificado produzca un valor.

```
const source = interval(300);
//after 4 seconds, emit value
const timer$ = timer(4000);
//when timer emits after 4s, complete source
const example = source.pipe(takeUntil(timer$));
//output: 0,1,2,3
const subscribe = example.subscribe(val => console.log(val));
```

Operadores de combinación

En ocasiones se disponen varios flujos de datos provenientes de más de un observable y necesitamos combinar los datos provenientes de diferentes observables. Estos operadores son los que nos permiten realizar combinaciones de datos de los flujos de los diferentes observables.

➔ combineLatest

Se combinan en un único valor (un *array*) el último valor emitido por cada uno de los observables. Es importante darse cuenta de que cada vez que llega un valor por cada uno de los observables se combinará este último con los últimos valores producidos por los otros observables.

```
const timerOne = timer(1000, 4000);
//timerTwo emits first value at 2s, then once every 4s
const timerTwo = timer(2000, 4000);
//timerThree emits first value at 3s, then once every 4s
const timerThree = timer(3000, 4000);

//when one timer emits, emit the latest values from each timer
as an array
const combined = combineLatest(timerOne, timerTwo, timerThree);

const subscribe = combined.subscribe(
  ([timerValOne, timerValTwo, timerValThree]) => {
    /*
      Example:
       timerOne first tick: 'Timer One Latest: 1, Timer Two
Latest:0, Timer Three Latest: 0
       timerTwo first tick: 'Timer One Latest: 1, Timer Two
Latest:1, Timer Three Latest: 0
      timerThree first tick: 'Timer One Latest: 1, Timer Two
Latest:1, Timer Three Latest: 1
  */
    console.log(
      `Timer One Latest: ${timerValOne},
     Timer Two Latest: ${timerValTwo},
     Timer Three Latest: ${timerValThree}`
    );
  }

);
```

➔ concat

Emite los valores de los observables en el mismo orden en el que se suscriben. Es decir, espera a que el observable anterior termine, antes de continuar con el otro.

```
const sourceOne = of(1, 2, 3);
const sourceTwo = of(4, 5, 6);
const example = sourceOne.pipe(concat(sourceTwo));
//output: 1,2,3,4,5,6
const subscribe = example.subscribe(val =>
  console.log("Example: Basic concat:", val)
);
```

➔ startWith

Antepone el valor especificado ante el resto de los valores.

```
const source = of(5, 4, 7);
const example = source.pipe(startWith(12));
//output: 12, 5, 4, 7
const subscribe = example.subscribe(val => console.log(val));
```

➔ merge

Convierte varios observables en uno solo.

```
const first = interval(300);
const second = interval(600);
const third = interval(100);
const fourth = interval(1000);

const example = merge(
  first.pipe(mapTo('1')),
  second.pipe(mapTo('2")),
  third.pipe(mapTo('3')),
  fourth.pipe(mapTo('4'))
);
//output: 3, 1, 2, 4
```

Operadores de utilidad

En ciertas ocasiones son necesarios operadores que nos permitan realizar operaciones que no encajan en ninguna de las anteriores categorías. Normalmente son utilizados para realizar conversiones o para realizar operaciones que no tienen implicación directa con el flujo (operaciones de lado, *side-effects*).

➔ toPromise

Convierte una secuencia de observables en una promesa.

```
//return basic observable
const sample = val => Rx.Observable.of(val).delay(5000);
//convert basic observable to promise
const example = sample('First Example')
  .toPromise()
  //output: 'First Example'
  .then(result => {
    console.log('From Promise:', result);
  });
```

➔ do/tap

Realiza acciones o efectos secundarios de forma transparente, es decir, realiza operaciones que no tienen una relación directa con el flujo (aunque pueden depender de un resultado parcial de los datos del flujo en un determinado momento).

```
// RxJS v6+
import { of } from 'rxjs';
import { tap, map } from 'rxjs/operators';

const source = of(1, 2, 3, 4, 5);
//transparently log values from source with 'do'
const example = source.pipe(
  tap(val => console.log(`BEFORE MAP: ${val}`)),
  map(val => val + 10),
  tap(val => console.log(`AFTER MAP: ${val}`))
);

//'do' does not transform values
//output: 11...12...13...14...15
const subscribe = example.subscribe(val => console.log(val));
```

➔ **toArray**

Convierte una secuencia de observables en un array.

```
// RxJS v6+
import { interval } from 'rxjs';
import { toArray, take } from 'rxjs/operators';

interval(100)
  .pipe(
    take(10),
    toArray()
  )
  .subscribe(console.log);

// output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Operadores de errores

Algunos operadores se utilizan para gestionar los errores que se producen en los datos de los flujos del observable. De este modo, se disponen de operadores que nos permiten capturar los errores o incluso repetir una función hasta que se produzca una circunstancia particular.

➔ **catch**

Captura los errores del Observable. Es recomendable retornar un nuevo observable de tipo cathError.

```
const source = throwError("This is an error!");
//output: 'I caught: This is an error'
const example = source.pipe(catchError(val => of(`I caught: $
{val}`)));
```

Actividades complementarias

"Hay que comentar que este tipo de actividades complementarias no son obligatorias ni evaluables, son ejercicios para practicar que recomendamos realizar para asimilar todos los conceptos progresivamente."

Es importante que se asienten los conceptos de RxJS antes de continuar. Por ello se recomiendan realizar las siguientes tareas:

- Realizar los 40 ejercicios (con soluciones) de la siguiente Web:

  http://reactivex.io/learnrx/

- Mirar el siguiente video-tutorial en el cual se hace una migración de un código basado en promesas hacia RxJS:

  https://www.youtube.com/watch?v=Y33g1sCoo60

- Mirar el siguiente vídeo en el cual se muestra el uso de RxJS Observable en una aplicación Angular:

  https://www.youtube.com/watch?v=q--U25yPTrA

- Haz una lectura final de la guía base sobre programación reactiva:

  "The introduction to Reactive Programming you've been missing":
  https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

Fundamentos del patrón Redux

El patrón Redux ha derivado en una biblioteca *open-source* de JavaScript para gestionar el estado de una aplicación. Redux es una pequeña biblioteca con una simple y limitada API la cual opera con funciones reductoras y programación funcional. Ésta es la biblioteca más ampliamente utilizada junto a React o Angular para construir las interfaces de usuario.

Redux se basa en tres fundamentos (léase https://es.redux.js.org/docs/introduccion/tres-principios.html):

1. **Única fuente de la verdad**. El estado de toda tu aplicación está almacenado en un árbol guardado en un único store. En conclusión, sería como disponer de un único almacén donde se encuentra la información actualizada desde la cual los demás componentes y servicios de la aplicación podrán consultar.

2. **El estado es de sólo lectura**. La única forma de modificar el estado es emitiendo una acción que genera un objeto describiendo qué ocurrió.

3. **Los cambios se realizan con funciones puras** (i.e. Operators). Para especificar cómo el árbol de estado es transformado por las acciones se utilizan reducers (funciones puras).

Los principales beneficios de utilizar Redux son:

1. Arquitectura escalable de datos.

2. Mayor control en el flujo de datos.

3. Programación funcional.

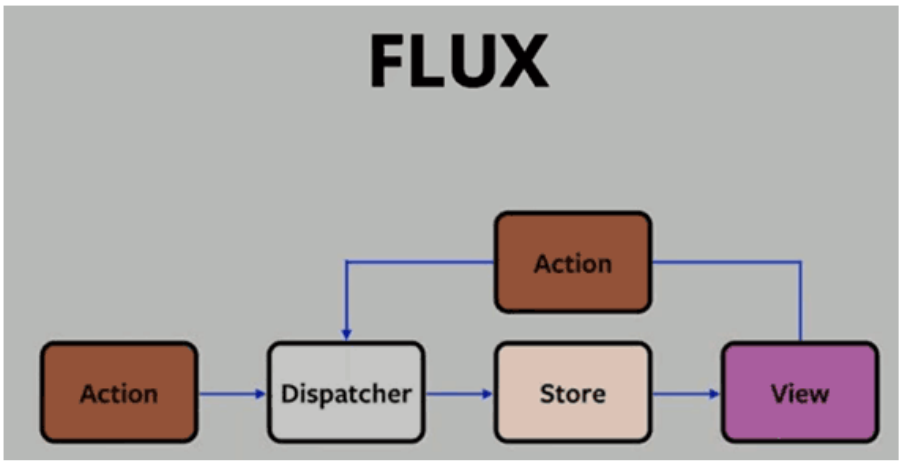
4. Control de estados.

La arquitectura de datos

Existen varios patrones de diseño que imponen una arquitectura de datos. El más popular de estos es el clásico MVC (Modelo-Vista-Controlador), aunque no es el único que se aplica, sino que podemos encontrar variantes como la aplicada por defecto en Angular llamado MVVM (Modelo-Vista-VistaModelo). En estos patrones se definen cómo será el flujo de la información entre los componentes de una aplicación (vista, modelos, controladores, servicios).

La arquitectura de datos de Redux difiere radicalmente de éstos puesto que sólo se puede modificar el estado desde un único punto, mientras que en las demás arquitecturas diferentes puntos (varios servicios) pueden modificar el estado de una aplicación. El hecho de que se pueda modificar el estado desde diferentes puntos de la aplicación eleva la complejidad de su desarrollo y la hace más propensa a errores.

Una de las aplicaciones más potentes en el campo de interacción con los usuarios desde el front-end es Facebook, la cual ideó un nuevo patrón denominado FLUX. En FLUX el flujo de datos sólo se realiza en una única dirección. Observa el siguiente diagrama que describe el flujo de datos donde aparecen conceptos como Action, Dispatcher, Store y View (para profundizar en la comprensión de FLUX, lee el siguiente artículo: https://facebook.github.io/flux/).

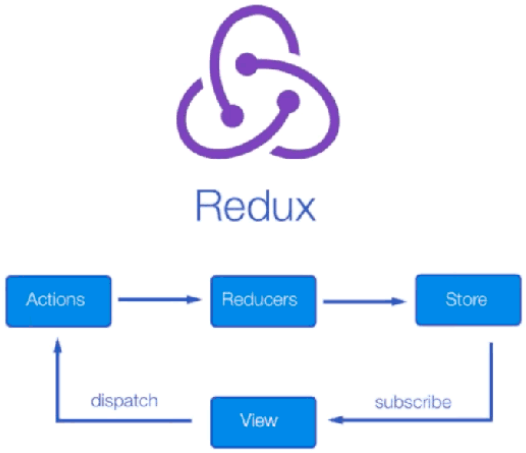


El Store es el almacén donde se almacenan todos los datos de la aplicación, lo que conocemos como el estado. Todos los datos se conectan directamente con las vistas (View, ahí es donde se visualiza el estado de la aplicación) y desde las vistas se disparan acciones que solicitan modificar el almacén (Store, el cual es inmutable y se creará un nuevo estado del almacén de datos).

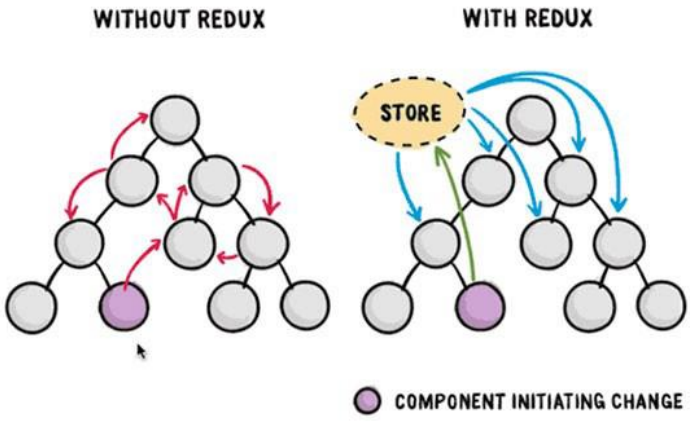
La realidad es que el patrón que se está utilizando (la implementación) en FLUX es la versión conocida como Redux. Redux es independiente de cualquier framework, se suele confundir y asociar Redux con React por ser React la biblioteca que primero integró su modelo de trabajo junto a Redux.

El diagrama de Redux se simplifica ligeramente frente al de FLUX pero el concepto es muy parecido.

En este nuevo diagrama aparece un bloque denominado Reducers que consiste en funciones puras que generan un nuevo estado que se almacena en el Store.



En la siguiente imagen se puede ver una comparativa del flujo de datos de una arquitectura utilizando *data-binding* con una y doble dirección en la que es un caos conocer quién modifica el estado de la aplicación frente a la propuesta de Redux en la cual todos los componentes de la aplicación se subscriben a los cambios del store (usando observables) y cuando se produce un cambio son notificados. Del mismo modo, las vistas son las que solicitan a los reducers crear un nuevo estado a partir de acciones.



https://css-tricks.com/learning-react- redux/

NgRX: definir estados, acciones, reducers, effects y selectores

NgRx es un *framework* para construir aplicaciones reactivas en Angular aplicando un gestor de estado (Redux). Además, se incluyen paquetes que reducen el *boileplater* de la aplicación del patrón Redux.

Los principios en los que se basa esta biblioteca son los siguientes:

1. EL estado es una estructura de datos única e inmutable.
2. Los componentes delegan la responsabilidad a los efectos de lado (*side-effects*), los cuales son gestionados de manera aislada.
3. Se incluye la seguridad de los tipos de los datos/atributos en la arquitectura haciendo uso de TypeScript.
4. Se promueve la programación funcional cuando se construyen aplicaciones reactivas.
5. Se proporciona una fuerte estrategia de test para validar la funcionalidad.

Los paquetes que componen esta biblioteca son los siguientes:

- **Store**: Gestor de estado para Angular inspirado en Redux y utilizando RxJS.
- **Store devtools**: Permite realizar *debugging* de nuestras aplicaciones.
- **Effects**: Efectos de lado (i.e. conexiones con servidor, bases de datos, *loggers* o cualquier operación que no tenga que interactuar directamente con el store).
- **Router Store**: Conecta Angular Router con una rama del árbol de estados.
- **Entity**: Adaptador para simplificar el *boileplater* de las acciones más comunes en un CRUD del estado.
- **Schematics**: *Scaffolidng* para las aplicaciones que usan NgRX.

Inicialmente, para trabajar necesitamos **Store**, **Store devtools** y **Effects** como base para nuestras primeras aplicaciones.

Los Effects

En una aplicación Angular, los componentes son los responsables de interactuar con los recursos externos directamente a través de los servicios. En lugar de eso, los efectos (*effects*) proporcionan un modo de interactuar con los servicios y aislarlo de los componentes. Los efectos son los encargados de manejar las tareas tales como recuperar datos de un servidor, ejecutar tareas pesadas que producen múltiples eventos, y otras interacciones externas donde los componentes no necesitan especificar el conocimiento de esas interacciones.

Los conceptos clave de los efectos son:

- Los efectos aíslan los efectos de lado de los componentes, permitiendo que los componentes sean más puros, ya que sólo se concentrarían en seleccionar el estado que quieren manipular y dispararían acciones.
- Los efectos escuchan a un observable de cada acción que es disparada desde el Store.
- Los efectos filtran aquellas acciones que son basadas en el tipo de acción que ellos están interesados. Esto normalmente es hecho por un operador de RxJS.
- Los efectos hacen tareas, las cuales son síncronas o asíncronas, y retornan una nueva acción.

La documentación oficial sobre *effects* de NgRX explica el paso a paso de cómo instalarlo y configurarlo (https://ngrx.io/guide/effects).