Angular S5: Observables

Después de esta lección podrás:

1. Comprender qué son los Observables
2. Explorar la librería RxJS y la programación reactiva
3. Entender cómo trabajar con RxJS y los Observables en Angular

En esta lección aprenderemos a trabajar con el **patrón *observer***, y cómo Angular lo usa para manejo de eventos y programación asíncrona. Gracias a la librería de RxJS, podremos hacer uso de los **Observables** en nuestra aplicación Angular.

Observables

Un **observable** se basa en dos elementos fundamentales:

- `subject` el sujeto, elemento al que nos vamos a suscribir en busca de cambios o información
- `observers` los receptores de la información, pueden estar en diferentes puntos y son los que se suscriben a los cambios, para consumirlos. Como su nombre

indica, son objetos que están observando.

Con los **observables** nos vamos a iniciar en la programación reactiva. ¿Qué es eso? No os preocupéis, porque seguro que os suena el concepto de *streaming*.

Cuando nos conectamos a una retransmisión en directo en una red social, estamos recibiendo información que se está generando en ese mismo momento, sobre esta idea gira la programación reactiva.

Pongamos un sencillo ejemplo, una función que retransmite colores. Cuando nos **suscribimos** a su canal, nos reporta la información, para finalizar con un mensaje al **completarse** la información del *streaming* (así como info de la **desuscripción**):

```
// Emisor
function streamingColores(observer: Observer<string>) {
  observer.next('Rojo');
  observer.next('Azul');
  observer.next('Naranja');
  observer.complete();

  return {
    unsubscribe() {
      console.log('--> Suscripción finalizada!');
    }
  };
}
```

```
// Receptor
const sequence = new Observable(streamingColores);

// Nos suscribimos al canal, para mostrar cada reporte
sequence.subscribe({
  next(msg) {
    console.log(msg);
  },
  complete() {
    console.log('Ohh! Colores terminados!');
  }
});
```

```
Rojo
Azul
Naranja
Ohh! Colores terminados!
--> Suscripción finalizada!
```

En este ejemplo los colores se reportan de manera sencilla (síncrona). Pero más adelante veremos como se puede complicar esta comunicación. El concepto con el que nos debemos quedar, es el de **suscripción** a un **canal de comunicación**.

RxJS

Esta es la librería de programación reactiva en la que nos apoyaremos en **Angular**. Nos proveerá de una implementación del patrón observable, que nos hará más fácil trabajar con ellos.

Es importante comprender que la programación reactiva, se basa en el tratamiento de los datos y eventos, que viajan por el canal que tenemos abierto. Podríamos ver el canal de streaming, como un **array de datos**, y el receptor decide cómo consumir cada **dato** que viaja por él.

Para manejar cada dato existen funciones de manipulación de colecciones que nos provee RxJS, llamadas **Operadores**. Seguro que nos suenan algunas de ellas como `map`:

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

const nums = of(1, 2, 3);

const squareValues = map((val: number) => val * val);
const squaredNums = squareValues(nums);

squaredNums.subscribe(x => console.log(x));
```

Y en el caso de que necesitemos combinar varios operadores, usaremos el método `pipe`:

```
import { filter, map } from 'rxjs/operators';

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

const squareOddVals = pipe(
  filter((n: number) => n % 2 !== 0),
  map(n => n * n)
);

const squareOdd = squareOddVals(nums);

squareOdd.subscribe(x => console.log(x));
```

Con ello, ya tenemos los elementos básicos para empezar a trabajar en Angular con Observables.

¿Recordáis las **Promesas y Callbacks** de Javascript? Pues con RxJS aprenderemos a convertir estas operaciones en **observables**.

Por ejemplo, al suscribirnos a las respuestas de una petición *fetch*:

```
import { from } from 'rxjs';

// Creamos un Observable sobre una promesa devuelta por la llamada fetch
const data = from(fetch('/api/endpoint'));
// Nos suscribimos, a la espera de que se resuelva la promesa
data.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completed'); }
});
```

O en este otro ejemplo, suscribiéndonos al lanzamiento del evento *mousemove:*

```
import { fromEvent } from 'rxjs';

const el = document.getElementById('my-element');

// Creamos un Observable que estará atento al evento mousemove
const mouseMoves = fromEvent(el, 'mousemove');

// Nos suscribimos al evento
const subscription = mouseMoves.subscribe((evt: MouseEvent) => {
  // Imprimimos las coordenadas del puntero con cada movimiento
  console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);
});

// Nos desuscribimos pasados 5 segundos
setTimeout(() => subscription.unsubscribe(), 5000);
```

Observables en Angular

Veamos ahora la aplicación de lo aprendido, pero en un contexto Angular. Sin haber sido conscientes, ya hemos estado viendo observables en secciones anteriores, sobre todo en la parte de servicios, ya que por defecto el HttpClient de Angular devuelve Observables en sus métodos.

Cuando aprendimos sobre comunicación entre componentes padre-hijo, el uso de los Outputs era posible gracias al `EventEmitter` (clase que extiende se los `Subject` de **RxJS**). Incluso cuando vimos los formularios reactivos, la manera con la que Angular monitoriza los cambios, es gracias a observables.

```
import { FormGroup } from '@angular/forms';

@Component({
  selector: 'my-component',
  template: 'MyComponent Template'
})
export class MyComponent implements OnInit {
  nameChangeLog: string[] = [];
  heroForm: FormGroup;

  ngOnInit() {
    this.logNameChange();
  }
  logNameChange() {
```

```
    const nameControl = this.heroForm.get('name');
    // Existe un observable sobre los cambios en el valor del formulario
    nameControl.valueChanges.forEach(
      (value: string) => this.nameChangeLog.push(value)
    );
  }
}
```

Pero el uso más común que haremos será en los **Servicios de Datos**. En la lección pasada aprendimos sobre los servicios de Angular, y en la segunda parte en concreto, aprendimos a construir un servicio para recuperar datos atacando un API externa.

```
// ¿Recordáis la función del servicio a la que nos suscribíamos?
this.requestExampleService.getCharacters()
  .subscribe((data: CharacterResponseInterface) => {
    this.data = data;
});
```

Ahora ya tenemos capacidad para mejorar los servicios, comprendiendo el uso de los observables. Por ejemplo, vamos a añadir un **control de error** por si la petición HTTP fallase, y vamos a llevarnos el **modelado de datos** al servicio:

```
export class RequestExampleService {

  constructor(private http: HttpClient) { }

  getCharacters() {
    return this.http.get(characterUrl).pipe(
      map((res: CharacterResponseInterface) => {
        if (!res) {
          throw new Error('Value expected!');
        } else {
          const results: CharacterInterface[] = res.results;

          const formattedResults = results.map(({ id, name, image }) => ({
            id,
            nombre: name,
            imagen: image,
          }));
          return formattedResults;
        }
```

```
      }),
      catchError(err => {
        throw new Error(err.message);
      })
    );
  }
}
```

```
export class StudentListComponent implements OnInit {
  characterList: object[] = [];

  constructor(private requestExampleService: RequestExampleService) {}

  ngOnInit() {
    this.requestExampleService.getCharacters()
    .subscribe((res) => {
      this.characterList = res;
      console.log(this.characterList);
    }, (err) => {
      console.error(err);
    });
  }
}
```

Finalmente estamos listos para atacar cualquier servidor, construyendo servicios de angular, y codificando el comportamiento de los observables que harán las peticiones.

Juntando observables con ForkJoin

En más de una ocasión nos encontramos en la necesidad de esperar a que dos observables terminen y emitan un valor.

Es bastante típico encontrar está solución:

```
// Hacemos primera petición
this.httpClient.get(this.urlPeticionUno).subscribe((dataPeticionUno) => {
  console.log(dataPeticionUno);
  // Cuando nos lleguen datos de la primera, hacemos la segunda
  this.httpClient.get(this.urlPeticionDos).subscribe((dataPeticionDos) => {
```

```
    console.log(dataPeticionDos);
    console.log(dataPeticionUno);
    // Aquí tenemos disponibles ambos datos
  });
});
```

Como puedes observar se ha optado, por realizar una de petición HTTP y dentro de la función que se ejecuta cuando llegan datos, hacemos la llamada a la segunda. De esta forma podemos asegurarnos dentro del segundo subscribe, de que han llegado ambos datos.

Esta solución puede ser correcta en casos en los que necesitemos datos de la primera petición para lanzar la segunda. En caso de que sean peticiones independientes estaríamos desperdiciando mucho tiempo, ya que no es necesario esperar para lanzar la segunda, y podríamos lanzarlas en paralelo.

Ahora bien ¿cómo puedo lanzar ambas peticiones en paralelo y enterarme de que las dos han acabado? Si se te ha ocurrido hacerte unas variables a modo de "flags" que te indiquen cuando ha llegado cada petición y así verificar si han llegado ambas... estás optando por una solución poco escalable.

No te preocupes, RxJS tiene una solución para resto, el operador forkJoin:

```
// Creamos los observables sin subscribirnos
const obsUno: Observable<any> = this.httpClient.get(this.urlPeticionUno);
const obsDos: Observable<any> = this.httpClient.get(this.urlPeticionDos);

// Juntamos ambos con forkjoin
const joinedObservable: Observable<any> = forkJoin([obsUno, obsDos]);

// Nos suscribimos a los datos
joinedObservable.subscribe((datosPeticion: any[]) => {
  const dataPeticionUno = datosPeticion[0];
  const dataPeticionDos = datosPeticion[1];
});
```

Como puedes "observar" con el forkjoin generamos un nuevo Observable que devuelve un array de los datos resueltos por cada uno de los observables.

Esto nos permitirá hacer peticiones en paralelo ¡Yujuuu!

Ejercicio

Ahora que ya sabemos algo más sobre RxJS vamos a ponerlo en práctica:

https://gitlab.com/upgrade-hub/ejercicios-master/angular/angular-sesion-5