UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN III
FRONT END

Anexo:

Rxjs

Anexo

2° Año – 3° Cuatrimestre

Índice

Anexo: RxJS y Programación Reactiva en Angular

¿Qué es RxJS?

RxJS (Reactive Extensions for JavaScript) es una librería que permite trabajar con flujos de datos de manera reactiva y asíncrona mediante métodos observables. Un observable es parecido a un “stream” (flujo de datos) al que te podes suscribir para recibir datos a medida que llegan (eventos, datos HTTP, timers, etc).

RxJS no sólo permite escuchar y reaccionar ante datos, sino que brinda poderosas herramientas para transformar, filtrar, combinar y manipular esos flujos mediante operadores.

¿Por qué RxJS en Angular?

Angular integra **RxJS** en su núcleo:

- Las peticiones **HTTP** con **HttpClient** retornan observables **RxJS**.
- Formularios reactivos, rutas, timers y WebSockets también pueden trabajarse como observables.
- Favorece la programación reactiva, en la que la UI se actualiza automáticamente al producirse cambios de datos, sin depender de eventos ni polling.

Conceptos Fundamentales

Observable

Un observable es un objeto que puede emitir múltiples valores en el tiempo (o ninguno), y se pueden observar esos valores con **subscribe()**.

```
const timer$ = new Observable<number>(observer => {
  let count = 0;
  setInterval(() => {
    observer.next(++count); // "emite" el siguiente valor
    if (count === 5) observer.complete();
  }, 1000);
});

timer$.subscribe({
                next: v => console.log('tick', v),
complete: () => console.log('Terminado')
});
```

Suscripción y Desuscripción

Cuando trabajas con observables en **RxJS**, te suscribís para empezar a recibir los valores que emiten:

```
const suscripcion = obs$.subscribe(valor => { ... });
```

Cada vez que el observable emite un valor (**next**), el **callback** pasado a subscribe se ejecuta. Pero, la suscripción sigue activa hasta que el observable complete, lance un error o vos la canceles manualmente.

¿Por qué es importante desuscribirse?

Si no cancelas (**desuscribes**) las suscripciones de observables “infinitos” (por ejemplo, streams de eventos, intervalos, sockets o streams que no se completan solos), tu aplicación puede:

- Seguir recibiendo eventos innecesarios, aunque el componente ya no exista en la pantalla.
- Consumir memoria y recursos de más (memory leaks).
- Generar bugs difíciles de rastrear, como actualizaciones sobre componentes ya destruidos.

Por eso, es fundamental liberar las suscripciones al destruir el componente.

Desuscripción Manual

```
const suscripcion = obs$.subscribe(valor => { ... });
// Cuando ya no necesitás escuchar más, o en ngOnDestroy:
   suscripcion.unsubscribe();

   //Si tenés varias suscripciones, podés agruparlas en una misma Subscription usando add().
     import { Subscription } from 'rxjs';

   subs = new Subscription();

   ngOnInit() {
     this.subs.add(obsA$.subscribe(...));
     this.subs.add(obsB$.subscribe(...));
   }

   ngOnDestroy() {
     this.subs.unsubscribe();
   }
```

Desuscripción Automática con takeUntil

En Angular es común escuchar observables que se mantienen activos mientras el componente está vivo (como eventos personalizados, **streams** de un servicio o datos en tiempo real). Una forma profesional y muy usada es usar el operador **takeUntil**, que cancela la suscripción automáticamente cuando se emite un valor en el observable "notificador", generalmente declarado como un **Subject** llamado **destroy$**:

```
      import { Subject, takeUntil } from 'rxjs';

destroy$ = new Subject<void>();

ngOnInit() {
 this.miServicio.datos$.pipe(
   takeUntil(this.destroy$)
 ).subscribe(datos => { ... });
}

ngOnDestroy() {
 this.destroy$.next();       // Emite valor y dispara cancelación
automática
 this.destroy$.complete();   // Limpia el subject
}
```

Ventajas de usar takeUntil:

- El código es claro y fácil de mantener.
- No importa cuántas suscripciones tengas, todas se desuscriben de forma limpia y segura.
- Se previenen memory leaks y errores por actualizaciones asíncronas en componentes destruidos.

¿Qué observables "se completan solos" y cuáles no?

Observables de **HttpClient** (ejemplo: this.http.get(...)) se completan solos después de emitir una respuesta, no requieren desuscripción manual.

Observables derivados de eventos, **Subject**, **BehaviorSubject**, intervalos (interval, timer), web sockets NO se completan solos y siempre deben ser correctamente gestionados/desuscritos.

Operadores RxJS Comunes

Los operadores se usan con **.pipe(...)** y permiten transformar, filtrar y combinar observables.

Operador	Uso	Ejemplo
map	Transforma cada valor emitido	map(x => x*2)
filter	Filtra valores según condición	filter(x => x>2)
tap	Efecto secundario, no cambia el valor	tap(console.log)
take	Toma los primeros n valores	take(1)
takeUntil	Cancela otro observable al emitir	takeUntil(this.destroy$)
switchMap	Cambia a un nuevo observable	switchMap(x => otroObs(x))
forkJoin	Ejecuta varios observables y espera todos	forkJoin([a$, b$])
combineLatest	Combina los últimos valores de varios obs	combineLatest([a$, b$])

Operadores RxJS

Los operadores permiten transformar, filtrar, mezclar o gestionar flujos de datos de un observable. Usan la función **.pipe()** para encadenarse entre sí y crear procesamiento secuencial de datos.

Ejemplo básico de uso de .pipe()

```
import { of } from 'rxjs';
import { map, filter } from 'rxjs/operators';
const origen$ = of(1, 2, 3, 4, 5);
origen$.pipe(
 filter(n => n % 2 === 1),    // Filtra sólo los impares: 1, 3, 5
 map(n => n * 10),           // Multiplica cada uno por 10: 10, 30, 50
        ).subscribe(val => console.log(val));
 // Output: 10, 30, 50
```

Tipos de métodos/operadores RxJS y ejemplo de cada uno

A continuación, explicación y ejemplo real de cada grupo importante de operador o método **RxJS**:

1. **Creación**

Crea observables a partir de distintos tipos de fuentes.

- **of**: crea a partir de una lista de valores

```
 import { of } from 'rxjs';
 of('a', 'b', 'c').subscribe(x => console.log(x));
 // Imprime: a b c
```

- **from**: a partir de un array, promesa, iterable

```
 import { from } from 'rxjs';
        from([1, 2, 3]).subscribe(x => console.log(x));
// Imprime: 1 2 3
```

- **interval:** emite números cada cierto tiempo

```
import { interval } from 'rxjs';
        const sub = interval(1000).subscribe(n => console.log(n));
        setTimeout(() => sub.unsubscribe(), 3100);
// Imprime: 0 1 2 (luego se detiene)
```

- **timer:** espera y luego empieza a emitir

```
import { timer } from 'rxjs';
        timer(2000, 1000).subscribe(n => console.log(n)); // Empieza en 2 segundos, luego
cada 1 seg.
```

2. Transformación

Transforman los valores que emite un observable.

- **map:**

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';
of(2, 4, 6).pipe(
 map(x => x / 2)
).subscribe(console.log);
// Imprime: 1 2 3
```

- **scan:** (acumula valores como un reduce)

```
import { of } from 'rxjs';
import { scan } from 'rxjs/operators';
of(1,2,3,4).pipe(
 scan((acum, val) => acum + val, 0)
).subscribe(console.log);
// Imprime: 1 3 6 10
```

3. Filtrado

Permiten limitar o decidir qué valores pasan.

- **filter:**

```
    import { of } from 'rxjs';
    import { filter } from 'rxjs/operators';
    of(1,2,3,4).pipe(
      filter(x => x > 2)
    ).subscribe(console.log);
// Imprime: 3 4
```

- **take(n):** toma los primeros n valores y luego completa.

```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';
interval(1000).pipe(
 take(3)
).subscribe(console.log);
// Imprime: 0 1 2
```

4. Combinación

Fusionan, concatenan o mezclan varios observables

- **merge:** emite de ambos en orden de llegada

```
  import { merge, of, interval } from 'rxjs';
   merge(of('a', 'b'), interval(1000).pipe(take(2)))
     .subscribe(console.log);
// Imprime: a b 0 1
```

- **forkJoin:** espera que todos los observables terminen y devuelve el resultado conjunto

```
 import { forkJoin, of } from 'rxjs';
   forkJoin([
     of(1),
     of(2),
     of(3)
   ]).subscribe(res => console.log(res));
// Imprime: [1, 2, 3]
```

5. Control de flujo y cancelación

- **takeUntil:** cancela el observable al emitirse otro observable

```
     import { interval, Subject } from 'rxjs';
   import { takeUntil } from 'rxjs/operators';

   const stop$ = new Subject<void>();
   interval(500)
     .pipe(takeUntil(stop$))
     .subscribe(n => {
       console.log(n);
       if (n === 3) stop$.next(); // Paramos a los 3
     });
// Imprime: 0 1 2 3
```

- **switchMap:** transforma y reemplaza el observable anterior por uno nuevo (ideal para peticiones sucesivas)

```
   import { fromEvent, interval } from 'rxjs';
import { switchMap, take } from 'rxjs/operators';

// Cada vez que hacés click, comienza una secuencia nueva de 3
intervalos, cancelando la anterior
fromEvent(document, 'click').pipe(
  switchMap(() => interval(500).pipe(take(3)))
).subscribe(console.log);
```

6. Efectos secundarios

- **tap:** ejecuta código sin modificar el **stream** (por ejemplo, logging)

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

of(4, 9).pipe(
  tap(x => console.log('Antes:', x)),
  map(x => x * x),
  tap(x => console.log('Después:', x))
).subscribe();
// Output:
// Antes: 4
// Después: 16
// Antes: 9
// Después: 81
```

El Encadenamiento y .pipe()

.pipe() es donde se encadenan los operadores en **RxJS**.

- Cada operador toma las emisiones y las transforma para el próximo.
- Se pueden combinar filtros, **mappers**, **side-effects**, y más.

Ejemplo real en Angular:

```
this.http.get<User[]>('/api/users').pipe(
 map(users => users.filter(u => u.activo)),
 tap(usersActivos => console.log('Activos:', usersActivos)),
 take(1)   // Solo el primer set de resultados, luego se desuscribe
).subscribe(usuarios => this.usuarios = usuarios);
```

Subject y BehaviorSubject en Angular

¿Qué es un Subject y para qué se usa?

Un **Subject** es un tipo especial de observable que puede emitir valores manualmente y a la vez ser observado por muchos suscriptores. Es útil para comunicar componentes o servicios, y para implementar flujos reactivos internos.

Ejemplo clásico: Un servicio avisa a varios componentes cuando pasa algo.

```
    import { Injectable } from '@angular/core';
  import { Subject } from 'rxjs';

  @Injectable({ providedIn: 'root' })
  export class NotificacionesService {
    private mensaje$ = new Subject<string>();

    get mensajes$() {
      return this.mensaje$.asObservable();
    }

    notificar(mensaje: string) {
      this.mensaje$.next(mensaje);
    }
  }

// En el componente receptor:
  this.notificacionesService.mensajes$.subscribe(msg => { ... });

// En el componente que emite:
  this.notificacionesService.notificar('Listo el pedido!');
```

Características importantes de Subject:

- Es **multicasting**: todos los suscriptores reciben los mismos valores.
- Si un nuevo suscriptor se suma, no recibe los valores anteriores; sólo recibe los valores futuros.

¿Qué es un BehaviorSubject y para qué se usa?

BehaviorSubject es un tipo de **Subject** que recuerda siempre el último valor emitido y lo entrega automáticamente al nuevo suscriptor.

Es ideal cuando necesitas que al suscribirte siempre tengas el estado actual, no te pierdas “el último valor”.

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

const user$ = new BehaviorSubject<string>('invitado');

user$.subscribe(user => console.log('Suscritor 1:', user)); //
Imprime 'invitado'
user$.next('Ana'); // imprime 'Ana' en todos los suscriptores

user$.subscribe(user => console.log('Suscritor 2:', user)); //
Imprime 'Ana' en el segundo
```

Diferencias clave con Subject:

Comienza con un valor inicial obligatorio.

Siempre provee el valor más reciente a nuevos suscriptores.

Ejemplo práctico:

Estado compartido con BehaviorSubject

Un servicio mantiene el usuario logueado a lo largo de la app:

```
// user.service.ts
   import { Injectable } from '@angular/core';
   import { BehaviorSubject } from 'rxjs';

   @Injectable({ providedIn: 'root' })
   export class UserService {
     private usuarioActual$ = new BehaviorSubject<string |
null>(null);

     get usuario$() {
       return this.usuarioActual$.asObservable();
     }

     setUsuario(nombre: string) {
       this.usuarioActual$.next(nombre);
     }
   }

// En el login:
   this.userService.setUsuario('Juan');

// En cualquier otro componente:
   this.userService.usuario$.subscribe(usuario => {
     if(usuario) alert('¡Hola, ' + usuario + '!');
   });
```

Analogía rápida

- **Subject**: sólo recibís los valores DESPUÉS de suscribirte.
- **BehaviorSubject**: siempre recibís el ÚLTIMO valor emitido (o el inicial) cuando te suscribes.

Resumen

- **RxJS** estructura y simplifica el manejo de datos asíncronos en Angular.
- Observables son la base para **streams**; todo lo asíncrono en Angular los usa.
- Operadores permiten transformar, filtrar, mezclar streams de forma poderosa y reactiva.
- **Subject** es ideal para emitir eventos o coordinar acciones entre componentes.
- **BehaviorSubject** es ideal para mantener y compartir estado reactivo “actual” en la app.

Bibliografía

- **Documentación oficial RXJS**

  https://v17.angular.io/guide/rx-library