






# Angular 17

## Tema 7. Servicios





## Objetivos

---

-  Diseño de servicios
-  Conexión con servidores
-  Patrón Observer

## Contenidos

---

-  Configuración e inyección de servicios
-  Diseño correcto de un servicio
-  Patrón Observer. Introducción a RxJS
-  Servicio HttpClient

## Configuración e inyección de dependencia de los servicios

Sólo se pueden inyectar (a un componente o a otros servicios) durante el «contexto de inyección». Podemos usar el **constructor** o la función **inject**:

```
constructor(private logica:LogicaService,  
            private almacen:AlmacenService) {}  
private logica=inject(LogicaService);  
private almacen=inject(AlmacenService);
```

El servicio puede registrarse en el **componente** que va a usarlo. Se creará y destruirá junto con el componente, cada vez que éste se dibuje y borre:

```
@Component({  
  ...  
  providers:[LogicaService]  
})
```

Y también se puede configurar de forma global a toda la aplicación usando **appConfig**. Sólo se creará una vez, y puede utilizarse para mantener el estado de la aplicación:

```
export const appConfig: ApplicationConfig = {  
  providers: [provideRouter(routes), AlmacenService]  
};
```

## Diseño de servicios

Reglas básicas:

- Debe ocultar la complejidad de la aplicación a los componentes
- Debe mantener el estado de la aplicación
- Lo más cerrado posible. Ante la duda, «private»

Debe ser **asíncrono**:

- callback
- promesas
- **patrón observer**

```
new Promise<T>((fnAceptado, fnRechazado)=> {  
  Operación lenta. Al acabar:  
  correcto ---> fnAceptado(datos)  
  error     ---> fnRechazado()  
});
```

```
promesa  
  .then(datos:T=> {  
    procesar los datos para la plantilla  
  })  
  .error()=>{  
    procesar el error para la plantilla  
  })
```

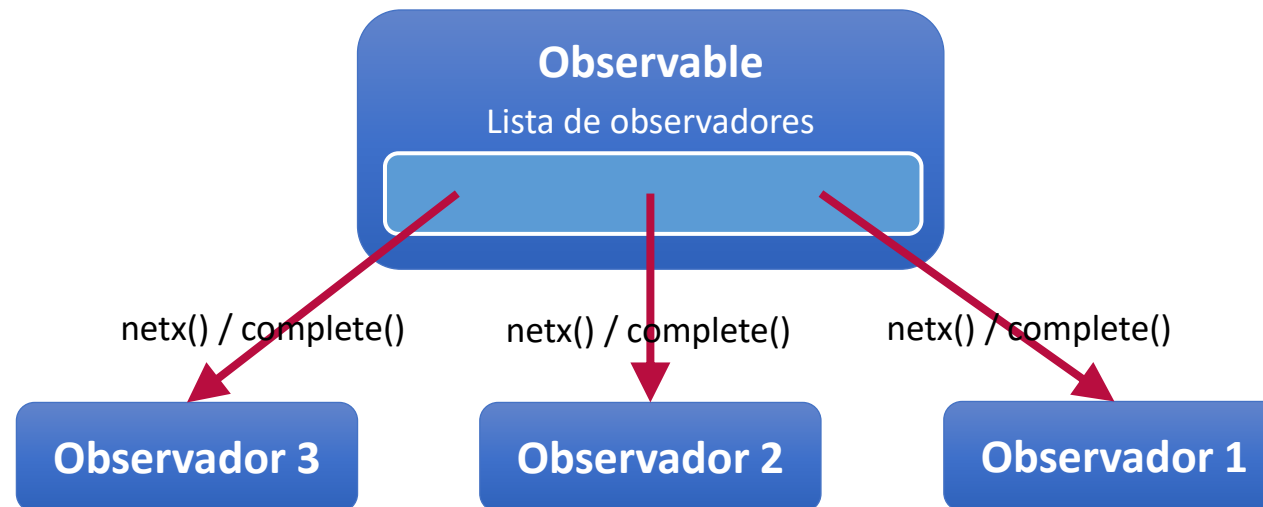
```
observador$=new Subject<T>();  
solicitarDatos() {  
  Operación lenta. Al acabar:  
  correcto ---> this.observador$.next(datos)  
  error     ---> this.observador$.error(error)  
  final     ---> this.observador$.complete()  
}
```

```
observable$.subscribe(  
  (datos:T)=> { procesar los datos para la plantilla },  
  (error)=> { procesar el error para la plantilla },  
  ()=>      { procesar el final para la plantilla }  
);  
  
solicitarDatos();
```

Es un patrón de diseño que permite a un **observable** notificar cualquier cambio de estado a sus **observadores**.

**Observable** (observable, editor, publisher) es el «sujeto» que «emite» el flujo de datos, el código que provoca los eventos.

**Observers** (observadores, suscriptores, subscribers) son los objetos a los que se les notifican los cambios producidos en el observable. Para que esto se produzca, el observador se **suscribe** al observable.



**RxJS** (Reactive Extensions for JavaScript) es una biblioteca de JavaScript para tareas asíncronas que implementa el patrón observable. Es un elemento esencial para múltiples aspectos de Angular.

```
interface Observer<T> {  
  next: (value: T) => void  
  error: (err: any) => void  
  complete: () => void  
}
```

```
interface Subscribable<T> {  
  subscribe(observer: Partial<Observer<T>>): Unsubscribable  
}
```

**subscribe(observador)** Un observador se suscribe a un observable

**next: (datos) => {...}** El observable emite datos para ser procesador por el observador.

**error: (error) => {...}** Se ha producido un error en el observable. Finaliza la suscripción.

**complete: () => {...}** La emisión de eventos ha finalizado. Finaliza la suscripción.

## Biblioteca RxJS. Clases y métodos usados en los ejemplos

---

<b>class Subject</b>	Observador / observable diseñado para comunicar un productor con un consumidor de eventos, por ejemplo un servicio y un componente.
<b>class BehaviourSubject</b>	Como el anterior, pero emite un evento adicional al suscribirse (define un valor emitido inicial).
<b>interface Observer&lt;T&gt;</b>	Define el conjunto de campos que debe tener un observador: <b>next</b> , <b>complete</b> , <b>error</b> .
<b>subscribe()</b>	Método de observables para que se suscriban observadores.
<b>pipe()</b>	Método de observables para aplicar operadores, funciones que crean o modifican el flujo de datos emitido.
<b>map()</b>	Operador de transformación que permite alterar el dato emitido.
<b>of()</b>	Operador de creación que crea un flujo de datos para cada parámetro suministrado.
<b>from()</b>	Operador de creación que crear un flujo de datos para cada elemento del array suministrado.
<b>delay()</b>	Operador que retrasa la emisión de un dato (para pruebas, por ejemplo).
...	



## Ejercicio 07 A. Crear

Quiero que añadas dos opciones al ejemplo del video anterior. La primera debe permitir la **creación** de valores nuevos:

Valores con observables adicionales

Valores con observables directos

Crear con observables directos

Modificar con observables adicionales

### Crear utilizando el servicio "datos-directos"

Este componente usa un servicio cuyos métodos devuelven directamente un observable que ejecuta "next", "error" y "complete". Por lo tanto el observable y sus suscripciones se invalidan después de cada uso, y hay que suscribirse una y otra vez, con cada ejecución del método del servicio.

Código	<input type="text" value="100"/>
Nombre	<input type="text" value="Nuevo valor"/>
Valor	<input type="text" value="1200"/>
<div>Crear nuevo valor</div>	

Utiliza y modifica el servicio **datos-directos**. Quiero que los métodos que añadas al servicio devuelvan observables que generen «next», «error» y «complete», con pausas simuladas de un segundo. Fallará si el «id» ya existe, y de forma aleatoria un 50% de las veces.

Emplea formularios reactivos y las reglas de validación que consideres oportunas.

## Ejercicio 07 A. Modificar

La segunda opción sirve para **modificar** valores nuevos:

Valores con observables adicionales

Valores con observables directos

Crear con observables directos

Modificar con observables adicionales

### Modificar usando el servicio "datos-complejos"

El servicio empleado proporciona observables que sólo ejecutan "next". Las suscripciones nunca finalizan (al menos en lo que respecta al servicio), y por ese motivo puede separar la obtención del observable y la suscripción del componente del método del servicio que realiza la acción, siempre de tipo void.

A cambio el código es algo más complejo, necesito definir observables adicionales en el servicio... y es más flexible cuando se producen cambios en el programa.

Aglutine SA

Seleccionar

Código	<input type="text" value="20"/>
Nombre	<input type="text" value="Aglutine SL"/>
Valor	<input type="text" value="1200"/>
<div>Modificar</div>	

Muestra una lista de valores, y cuando se selecciona uno aparece el segundo formulario para modificar el nombre o el precio.

Utiliza y modifica el servicio **datos-complejos**. El servicio tendrá métodos adicionales para proporcionar observables que nunca finalizan, y obviamente los métodos que ejecutan las acciones serán «void». Como siempre, pausas simuladas de un segundo y fallará si el «id» no existe, y de forma aleatoria un 50% de las veces.

Emplea formularios reactivos y las reglas de validación que consideres oportunas. No hace falta que añadas subcomponentes; dibuja dos veces el formulario «crear/modificar».

## Servicio HttpClient



Servicio estándar de Angular para realizar peticiones **HTTP** a servidores, generalmente servidores **REST**.

Internamente emplea RxJS.

Métodos: request, get, post, put, delete ...

Se importa con el método «provideHttpClient()»

Por defecto emplea JSON para comunicarse con el servidor.

Debes inyectarlo en otros servicios, nunca directamente en el componente.

Recuerda que las peticiones HTTP tradicionales sólo envían **una** respuesta.

```
private http=inject(HttpClient);
private urlBase='https://api.restful-api.dev/objects';

leerConSuscripciónDirecta$():Observable<Dato[]> {
  return this.http.request<Dato[]>('get',this.urlBase);
}

private leer$=new Subject<Dato[] | null>();

getLeer$() {
  return this.leer$;
}

leerUsandoUnSubjectAdicional():void {
  this.http.get<Dato[]>(this.urlBase).subscribe({
    next:lista=>this.leer$.next(lista),
    error:e=>this.leer$.next(null),
    complete:()=>{}
  });
}
```

## Servidores REST para pruebas

---

- <https://free-apis.github.io>

**Free APIs**

- <https://restful-api.dev>

**RESTFUL-API**


- <https://jsonplaceholders.typicode.com>

**{JSON} Placeholder**

- <https://api.nasa.gov>

 **{ APIs }**

- <https://fakeapi.platzi.com>

 **Platzi Fake Store API**

- <https://fakestoreapi.com>

**Fake Store API**

## ¿Qué hemos aprendido?

---

- Diseño y configuración de un servicio
- Patrón Observer
- Introducción a RxJS y su uso en Angular
- Uso de HttpClient

## Resumen de comandos

---

### npm

npm install [-g] paquete  
npm install  
npm uninstall paquete  
npm -version

### ng (1)

ng version  
ng help  
ng serve [-o]  
ng build  
ng new nombre\_proyecto  
    --routing false  
    --skip-tests  
    --skip-git  
    --no-standalone

### ng (2)

ng generate component nombre\_componente / ng g c  
ng generate service nombre\_servicio / ng g s  
ng generate module nombre\_módulo / ng g m

## Ejercicio 07 B

Quiero que solicites la «foto del día» al servidor de la NASA, y que el servicio recuerde todos los datos recibidos. Si ya los tiene, no vuelve a solicitarlos.

La url será similar a ésta: **[https://api.nasa.gov/planetary/apod?api\\_key=TU\\_API\\_KEY&date=2007-09-11](https://api.nasa.gov/planetary/apod?api_key=TU_API_KEY&date=2007-09-11)**

Tres componentes:

- **Ver.** Muestra un formulario dirigido por plantilla para la fecha
- **Lista.** El resumen de las fotos recuperadas
- **Detalle.** Usado por los dos componentes anteriores, muestra el detalle de una foto

Usa estas dos clases para recuperar los datos del servidor, almacenarlos o dibujar las plantillas:

```
export class Foto {  
  constructor(  
    public date:string,  
    public copyright:string,  
    public explanation:string,  
    public hdurl:string,  
    public media_type:string,  
    public service_version:string,  
    public title:string,  
    public url:string  
  ){}  
}
```

```
export class Resumen {  
  constructor(  
    public date:string,  
    public title:string  
  ){}  
}
```