# **Servicios en Angular:**

#### **¿Qué son los Servicios en Angular?**

Los **servicios** en Angular son componentes reutilizables que encapsulan lógica de negocio y funciones comunes para ser utilizados por múltiples componentes. Su propósito principal es compartir datos y funcionalidades a través de la aplicación de manera eficiente.

**Características Clave:**

* **Reutilizables**: Se pueden usar en múltiples componentes.
* **Modulares**: Centralizan la lógica de negocio.
* **Basados en DI**: Utilizan la Inyección de Dependencias (Dependency Injection) para ser accesibles desde otros componentes.

#### **¿Cómo crear un servicio en Angular?**

Hay dos formas de crear servicios en Angular:

**Usando Angular CLI**  
ng g s services/NombreDelServicio --skip-tests

1. **Explicación del comando:**
   * ng g s: Genera un nuevo servicio.
   * services/NombreDelServicio: Crea el servicio dentro de la carpeta "services" con el nombre NombreDelServicio.
   * --skip-tests: Omite la creación de archivos de prueba (.spec.ts).
2. **Creación Manual**
   * Crear un archivo **MiServicio.ts**.
   * Importar Injectable desde @angular/core.
   * Decorar la clase con @Injectable({ providedIn: 'root' }).

**Ejemplo de código:**

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root'

})

export class MiServicio {

constructor() {}

getData() {

return ['Item 1', 'Item 2', 'Item 3'];

}

}

# **Iteradores y Promesas**

En Angular y JavaScript, conceptos como el **patrón iterador** y las **promesas** son fundamentales para trabajar con colecciones y tareas asincrónicas. Ambos permiten manejar datos de forma eficiente y estructurada, simplificando tareas complejas.

## **Patrón iterador**

El **patrón iterador** facilita recorrer colecciones de elementos (como arrays o listas) sin necesidad de conocer su estructura interna. Esto permite separar la lógica de iteración de los datos, haciendo el código más modular y reutilizable.

### **Características**

* Itera sobre colecciones mediante el método next().
* Devuelve un objeto con dos propiedades:
  + **value**:El valor actual.
  + **done**: Indica si se cocina la iteración.

### **Ejemplo práctico**

const miArray = [1, 2, 3];

const iterador = miArray[Symbol.iterator]();

console.log(iterador.next()); // { value: 1, done: false }

console.log(iterador.next()); // { value: 2, done: false }

console.log(iterador.next()); // { value: 3, done: false }

console.log(iterador.next()); // { value: undefined, done: true }

En este caso:

* **Symbol.iterator**Crea un iterador para la matriz.
* Cada llamada a next()devuelve el siguiente elemento hasta que donesea true.

## **Promesas**

Una **promesa** es un objeto que representa el resultado eventual de una operación asincrónica. Simplifica la ejecución de tareas como llamadas a API, temporizadores o procesos en segundo plano.

### **Estados de una Promesa**

1. **Pendiente (pendiente):** La operación está en progreso.
2. **Resuelta (cumplido):** La operación fue exitosa.
3. **Rechazada (rechazada):** La operación falló.

### **Métodos principales**

* **then()**: Maneja el resultado cuando la promesa se resuelve.
* **catch()**: Maneja errores cuando la promesa es rechazada.

### **Ejemplo práctico**

const promesa = new Promise((resolve, reject) => {

const exito = true;

if (exito) {

resolve("Operación exitosa");

} else {

reject("Error en la operación");

}

});

promesa

.then((resultado) => console.log(resultado)) // Operación exitosa

.catch((error) => console.error(error)); // Error en caso de fallo

## **Promesas vs. Observables**

Aunque las promesas son útiles para manejar tareas asincrónicas, en aplicaciones más complejas Angular utiliza **observables** (RxJS) para gestionar flujos continuos de datos.

| **Aspecto** | **Promesas** | **Observables** |
| --- | --- | --- |
| **Respuesta** | Una sola vez | Varias veces (flujo continuo) |
| **Cancelación** | No soportada | Soportado |
| **Manejo** | Simple ( then, catch) | Complejo ( subscribe, unsubscribe) |

**Conclusión:** El **patrón iterador** permite recorrer colecciones de estructura manerada, mientras que las **promesas** simplifican la gestión de tareas asincrónicas. Estos conceptos son esenciales para manejar datos y flujos en aplicaciones modernas.

## ¿Qué es un Injection Token?

Un **Injection Token** permite crear identificadores únicos para proveer valores a través de la Inyección de Dependencias.

**Ejemplo de código:**

import { InjectionToken } from '@angular/core';

export const API\_URL = new InjectionToken<string>('apiUrl');

Para usar el **Injection Token**:

@NgModule({

providers: [

{ provide: API\_URL, useValue: 'https://api.com' }

]

})

# **Observables y Sujetos**

En Angular, los **observables** y **sujetos** son herramientas esenciales del patrón reactivo proporcionado por RxJS. Permiten manejar flujos de datos asincrónicos de forma eficiente, lo que es clave para crear aplicaciones dinámicas y reactivas.

## **¿Qué son los observables?**

Los observables son **productores de datos** que emiten valores a lo largo del tiempo. Estos valores pueden ser datos únicos, múltiples, o incluso un error o señal de completado. Son ampliamente utilizados en Angular para gestionar eventos, solicitudes HTTP, formularios reactivos y más.

### **Características de los Observables:**

1. **Productores de Datos:** Generan y emiten valores.
2. **Asincrónicos:** Procesan datos en el tiempo, sin bloquear el flujo principal.
3. **Composición:** Permiten aplicar operadores como map, filtery mergepara transformar los datos.

### **Ejemplo Práctico de Observable:**

import { Observable } from 'rxjs';

const observable = new Observable(subscriber => {

subscriber.next('Primer valor');

subscriber.next('Segundo valor');

subscriber.complete();

});

observable.subscribe({

next: (valor) => console.log(valor), // 'Primer valor', 'Segundo valor'

complete: () => console.log('Completado')

});

En este ejemplo:

* El observable emite dos valores y luego se completa.
* El método subscribegestiona cómo se consumen los datos.

## **¿Qué son los temas?**

Un **sujeto** es una extensión de los observables que puede actuar tanto como productor (observable) como consumidor. Esto lo hace ideal para manejar eventos compartidos entre múltiples partes de la aplicación.

### **Características de los sujetos:**

1. **Multicast:** Permite que múltiples suscriptores reciban los mismos datos.
2. **Bidireccionalidad:** Puede emitir y recibir valores.
3. **Flexibilidad:** Útil para compartir estados o eventos globales.

### **Ejemplo práctico de tema:**

import { Subject } from 'rxjs';

const subject = new Subject();

subject.subscribe(valor => console.log('Suscriptor 1:', valor));

subject.subscribe(valor => console.log('Suscriptor 2:', valor));

subject.next('Mensaje para todos los suscriptores');

// Ambos suscriptores reciben el mismo valor

En este ejemplo:

* El sujeto emite un mensaje que es recibido por todos los suscriptores simultáneamente.

## **Diferencias entre Observables y Sujetos**

| **Característica** | **Observables** | **Temas** |
| --- | --- | --- |
| **Rol** | Solo productor de datos | Productor y consumidor de datos |
| **Suscriptores** | Cada uno recibe valores de forma independiente | Todos reciben el mismo flujo de datos. |
| **Creación** | Generalmente usados ​​en tareas específicas | Ideal para compartir estados globales |

**Conclusión:** Los **observables** son fundamentales para manejar flujos de datos asincrónicos en Angular, mientras que los **sujetos** agregan flexibilidad al permitir la comunicación entre múltiples partes de la aplicación. Su combinación permite crear aplicaciones reactivas, escalables y organizadas.

## ¿Qué es la Inyección de Dependencias (DI) en Angular?

La **Inyección de Dependencias (DI)** permite que los servicios sean inyectados automáticamente en los componentes o módulos donde se necesiten.

**¿Cómo se usa?**

* Se define el servicio con @Injectable({ providedIn: 'root' }).
* Se usa el constructor del componente para inyectar el servicio.

**Ejemplo de código:**

import { Component, OnInit } from '@angular/core';

import { MiServicio } from './services/mi-servicio.service';

@Component({

selector: 'app-componente',

templateUrl: './componente.component.html',

styleUrls: ['./componente.component.css']

})

export class ComponenteComponent implements OnInit {

constructor(private miServicio: MiServicio) {}

ngOnInit() {

console.log(this.miServicio.getData());

}

}

## Tipos de Inyección en Angular

Angular permite controlar cómo se inyecta el servicio con las siguientes opciones:

| **Tipo** | **Descripción** | **Ejemplo** |
| --- | --- | --- |
| **useClass** | Crea una nueva instancia del servicio. | useClass: OtraClase |
| **useExisting** | Usa un servicio existente como alias. | useExisting: MiServicio |
| **useValue** | Usa un valor literal en lugar de un servicio. | useValue: { apiUrl: 'https://api.com' } |
| **useFactory** | Llama a una función para crear la instancia. | useFactory: () => new MiServicio() |

# **Introducción al Operador de Tuberías**

El operador **pipe**en RxJS es una herramienta fundamental que permite combinar múltiples operadores para transformar y manejar flujos de datos de manera eficiente. Es útil especialmente para procesar datos de observables en Angular, ya que facilita la composición y la lectura del código.

## **¿Qué es el Operador Pipe?**

El operador **pipe**actúa como un contenedor que conecta varios operadores de transformación o filtrado en un flujo continuo. Esto simplifica la gestión de flujos de datos al aplicar múltiples operadores de forma secuencial.

**Ventajas del operador pipe:**

* Combina operadores de manera clara y legible.
* Facilita la reutilización y composición de operadores.
* Optimiza la gestión de flujos de datos en proyectos complejos.

## **Ejemplo Práctico: Uso de mapy filteren un Pipe**

Supongamos que tenemos un observable que emite una lista de números. Queremos transformar estos números (multiplicándolos por 2) y filtrar aquellos que sean mayores de 5.

**Código de ejemplo:**

import { of } from 'rxjs';

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

const numeros$ = of(1, 2, 3, 4, 5, 6);

numeros$

.pipe(

map(valor => valor \* 2), // Multiplica cada valor por 2

filter(valor => valor > 5) // Filtra valores mayores a 5

)

.subscribe(resultado => console.log('Resultado:', resultado));

**Explicación del Flujo:**

1. **map:** Transforma los números emitidos por el observable multiplicándolos por 2.
2. **filter:** Filtra los números resultantes, dejando pasar solo aquellos mayores a 5.

**Salida esperada:**

* Resultado: 6
* Resultado: 8
* Resultado: 10
* Resultado: 12

## **Aplicación en Angular**

En Angular, el operador pipese utiliza frecuentemente en:

* Transformaciones de datos obtenidos de APIs.
* Filtrado de resultados en listas.
* Modificación de eventos antes de manejarlos.

**Ejemplo Angular:**

this.miServicio.getDatos()

.pipe(

map(data => data.items), // Extraer la lista de elementos

filter(items => items.length > 0) // Filtrar listas no vacías

)

.subscribe(items => this.lista = items);

## **Conclusión**

El operador **pipe**es una herramienta poderosa para manejar y transformar flujos de datos en RxJS. Su capacidad de combinar múltiples operadores como mapy filterlo hace esencial en el desarrollo con Angular, permitiendo construir flujos de datos eficientes y fáciles de leer.

## ¿Qué es el Pipe Async?

El **Pipe Async** permite mostrar los resultados de un **Observable** o una **Promesa** directamente en la vista.

**Ejemplo de uso:**

<p>Resultado: {{ observable$ | async }}</p>

**Ventajas:**

* Simplifica la visualización de datos en la vista.
* No se necesita subscribe(), ya que el Pipe Async gestiona la suscripción automáticamente.

**Operadores Básicos**

En RxJS, los **operadores** son herramientas poderosas para transformar y gestionar flujos de datos. Los operadores **map** , **tap** y **forkJoin** son esenciales en Angular para manipular observables de manera efectiva. A continuación, se explican sus funcionalidades, aplicaciones prácticas y resultados esperados.

### **Mapa del operador**

El operador **map** transforma cada valor emitido por un observable aplicando una función definida. Es útil para modificar datos antes de procesarlos o enviarlos a otro flujo.

**Ejemplos prácticos:**

import { of } from 'rxjs';

import { map } from 'rxjs/operators';

of(1, 2, 3)

.pipe(map(valor => valor \* 2))

.subscribe(resultado => console.log(resultado));

**Resultado esperado:**

* El flujo original ( 1, 2, 3) se transforma en 2, 4, 6.

**Aplicación en Angular:**

* Transformar datos obtenidos de una API, como convertir unidades o calcular totales.

### **Operador Tap**

El operador **tap** permite realizar acciones secundarias (efectos secundarios) en los valores emitidos por un observable, sin alterar el flujo de datos. Es ideal para realizar registros o depuración.

**Ejemplos prácticos:**

import { of } from 'rxjs';

import { tap } from 'rxjs/operators';

of('Angular', 'RxJS', 'Operators')

.pipe(tap(valor => console.log(`Procesando: ${valor}`)))

.subscribe(resultado => console.log(`Emitido: ${resultado}`));

**Resultado esperado:**

* Procesando: Angular
* Emitido: Angular
* Repite el proceso para cada valor.

**Aplicación en Angular:**

* Registrar datos en la consola para depuración o realizar acciones sin modificar los valores originales.

### **Operador ForkJoin**

El operador **forkJoin** combina múltiples observables, esperando a que todos completen para emitir sus últimos valores como un array o un objeto. Es útil para procesar operaciones que deben finalizar juntas.

**Ejemplos prácticos:**

import { of, forkJoin } from 'rxjs';

const observable1 = of('Primera respuesta');

const observable2 = of('Segunda respuesta');

forkJoin([observable1, observable2])

.subscribe(resultados => console.log(resultados));

**Resultado esperado:**

* ['Primera respuesta', 'Segunda respuesta']
* Los valores se emiten solo cuando todos los observables completan su flujo.

**Aplicación en Angular:**

* Ejecutar múltiples solicitudes HTTP y procesar sus resultados una vez que todas se completen.

**Conclusión:** Los operadores **map** , **tap** y **forkJoin** son fundamentales en Angular para transformar, depurar y combinar flujos de datos de forma eficiente. Su uso adecuado ayuda a crear aplicaciones más dinámicas y organizadas.

## ¿Qué son los Operadores de Alto Orden?

Los **operadores de alto orden** permiten combinar observables de forma más avanzada.

| **Operador** | **Descripción** | **Ejemplo** |
| --- | --- | --- |
| **mergeMap** | Convierte cada valor en un nuevo Observable. | mergeMap(valor => Observable) |
| **switchMap** | Cancela la suscripción al observable anterior. | switchMap(valor => Observable) |

**Ejemplo de código con mergeMap:**

import { fromEvent } from 'rxjs';

import { mergeMap } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');

clicks.pipe(

mergeMap(() => fetch('https://api.com'))

);

## Métodos next, error y complete en Observables

Los **observables** en RxJS tienen un ciclo de vida que se controla a través de los métodos **next** , **error** y **complete** . Estos métodos permiten manejar los datos emitidos, los errores y la finalización de un flujo de datos. A continuación, exploramos su funcionalidad con ejemplos prácticos.

### **Ciclo de Vida de un Observable**

1. **next():** Se llama para emitir un valor del observable al suscriptor.
2. **error():** Se ejecuta cuando ocurre un error durante el flujo. Detiene el observable.
3. **complete():** Indica que el observable ha terminado de emitir valores. No se emiten más datos después de este punto.

### **Ejemplo práctico**

A continuación, creamos un observable que emite una serie de valores, simula un error y luego finaliza correctamente:

import { Observable } from 'rxjs';

const observable = new Observable(observer => {

observer.next('Valor 1'); // Emitir primer valor

observer.next('Valor 2'); // Emitir segundo valor

if (Math.random() > 0.5) {

observer.error('Ocurrió un error'); // Emitir error aleatorio

} else {

observer.complete(); // Completar el flujo

}

});

observable.subscribe({

next: valor => console.log('Emitido:', valor),

error: err => console.error('Error:', err),

complete: () => console.log('Flujo completado'),

});

### **Flujo de eventos**

#### **1. Cuando no hay errores:**

1. next('Valor 1') → next('Valor 2') → complete()  
   **Salida esperada:**
   * Emitido: Valor 1
   * Emitido: Valor 2
   * Flujo completado

#### **2. Cuando ocurre un error:**

1. next('Valor 1') → next('Valor 2') → error('Ocurrió un error')  
   **Salida esperada:**
   * Emitido: Valor 1
   * Emitido: Valor 2
   * Error: Se produjo un error

### **Aplicaciones en Angular**

* **next():** Utilizado para manejar datos emitidos por solicitudes HTTP o eventos del usuario.
* **error():** Útil para gestionar errores en llamadas a APIs o flujos complejos.
* **complete():** Garantiza que los recursos del observable se liberen correctamente tras su uso.

**Conclusión:** Los métodos **next** , **error** , y **complete** son fundamentales para gestionar el ciclo de vida de un observable. Comprender su uso es clave para manejar flujos de datos dinámicos y asincrónicos en Angular de manera eficiente.

# **Gestión Avanzada de Suscripciones**

En Angular, gestionar correctamente las suscripciones a observables es fundamental para evitar problemas de rendimiento, como **fugas de memoria** . Esta guía presenta estrategias avanzadas utilizando el **patrón unsubscribe** y el operador **takeUntil** para manejar suscripciones de manera eficiente.

## **Problemas Comunes con las Suscripciones**

Cuando los componentes no cierran sus suscripciones al destruirse, los observables continúan activos, consumiendo recursos innecesariamente. Esto puede generar:

* **Pérdidas de memoria:** Acumulación de recursos no utilizados.
* **Baja eficiencia:** Incremento del consumo de memoria y procesamiento.

## **Estrategias para Gestionar Suscripciones**

### **1. Patrón darse de baja**

El método unsubscribe()permite cerrar manualmente las suscripciones en el ciclo de vida del componente, específicamente en el método ngOnDestroy.

**Ejemplo práctico:**

import { Component, OnDestroy } from '@angular/core';

import { Subscription, interval } from 'rxjs';

@Component({

selector: 'app-example',

template: `<p>Ejemplo con unsubscribe...</p>`,

})

export class ExampleComponent implements OnDestroy {

private subscription: Subscription;

constructor() {

this.subscription = interval(1000).subscribe(val => console.log(val));

}

ngOnDestroy(): void {

this.subscription.unsubscribe(); // Cerrar manualmente la suscripción

console.log('Suscripción cerrada');

}

}

**Ventajas:**

* Fácil de implementar para una o pocas suscripciones.

**Desventajas:**

* Propenso a errores si se olvida llamar a unsubscribe().

### **2. Operador takeUntil**

El operador **takeUntil** utiliza un **Asunto** para automatizar el cierre de suscripciones. Cuando se emite un valor desde el Subject, todas las suscripciones asociadas finalizan automáticamente.

**Ejemplo práctico:**

import { Component, OnDestroy } from '@angular/core';

import { Subject, interval } from 'rxjs';

import { takeUntil } from 'rxjs/operators';

@Component({

selector: 'app-example',

template: `<p>Ejemplo con takeUntil...</p>`,

})

export class ExampleComponent implements OnDestroy {

private destroy$ = new Subject<void>();

constructor() {

interval(1000)

.pipe(takeUntil(this.destroy$))

.subscribe(val => console.log(val));

}

ngOnDestroy(): void {

this.destroy$.next(); // Emitir señal para cerrar suscripciones

this.destroy$.complete(); // Completar el Subject

console.log('Suscripciones gestionadas con takeUntil');

}

}

**Ventajas:**

* Automatiza el cierre de múltiples suscripciones.
* Ideal para componentes con varios observables.

**Desventajas:**

* Requiere un Asunto adicional.

## **Comparación de métodos**

| **Método** | **Ventajas** | **Desventajas** |
| --- | --- | --- |
| **Darse de baja** | Control directo sobre cada suscripción. | Riesgo de olvidar llamar a unsubscribe(). |
| **tomarHasta** | Automatiza el cierre de múltiples suscripciones. | Requiere un Asunto adicional. |

## **Diagrama sugerido**

1. **Darse de baja:**
   * Línea continua de eventos conectada a un nodo donde se llama manualmente a unsubscribe()para detener el flujo.
2. **tomarHasta:**
   * Varias líneas de observables que convergen en un Subject, el cual emite una señal para detener automáticamente todas las suscripciones.

## **Conclusión**

La gestión adecuada de suscripciones es esencial para mantener el rendimiento y evitar problemas de memoria en aplicaciones Angular. Para escenarios simples, el patrón **darse de baja** es suficiente. Sin embargo, en aplicaciones más complejas, el uso del operador **toma** hasta mejorar la escalabilidad y automatiza la limpieza de suscripciones, asegurando un manejo eficiente de los recursos.

**Posibles soluciones a las actividades prácticas:**

## **1.Actividad: Crear un Servicio con providedIn: 'root' y consumirlo desde un Componente**

**Ubicación:** Diapositiva "Utilización de Servicios"  
**Duración:** 20 minutos

### **Consigna**

1. Crear un servicio que permita recuperar una lista de artículos.
2. Registrar el servicio usando providedIn: 'root' en el decorador @Injectable.
3. Consumir el servicio desde un componente e imprimir la lista de artículos en la consola.

### **Posible Solución**

**Archivo: articulos.service.ts**

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root'

})

export class ArticulosService {

retornar() {

return [

{ codigo: 1, descripcion: 'patatas', precio: 12.33 },

{ codigo: 2, descripcion: 'manzanas', precio: 54 }

];

}

}

**Archivo: home.component.ts**

import { Component, OnInit } from '@angular/core';

import { ArticulosService } from './articulos.service';

@Component({

selector: 'app-home',

templateUrl: './home.component.html',

styleUrls: ['./home.component.css']

})

export class HomeComponent implements OnInit {

constructor(private articulosService: ArticulosService) {}

ngOnInit(): void {

console.log(this.articulosService.retornar());

}

}

## **2. Actividad: Inyección de Dependencias**

**Ubicación:** Diapositiva "Inyección de Dependencias (DI)"  
**Duración:** 15 minutos

### **Consigna**

1. Crear un servicio.
2. Inyectar el servicio en un componente utilizando el constructor de la clase.
3. Verificar que el servicio funciona correctamente en el componente consumiéndolo.

### **Posible Solución**

**Archivo: mi-servicio.service.ts**

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root'

})

export class MiServicioService {

getMensaje(): string {

return 'Este es un mensaje del servicio.';

}

}

**Archivo: app.component.ts**

import { Component } from '@angular/core';

import { MiServicioService } from './mi-servicio.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

constructor(private miServicio: MiServicioService) {}

ngOnInit(): void {

console.log(this.miServicio.getMensaje());

}

}

## **3. Actividad: Find the Bug - Error en la Provisión de Servicios**

**Ubicación:** Después de la diapositiva 9  
**Duración:** 10 minutos

### **Consigna**

1. Se proporciona un archivo con errores en la provisión de servicios.
2. Corregir el archivo del servicio y su registro en el módulo.
3. Verificar que el componente puede acceder al servicio y consumir los datos.

### **Posible Solución**

**Archivo original con errores:**

export class ArticulosService {

retornarArticulos() {

return [

{ codigo: 1, descripcion: 'patatas', precio: 12.33 },

{ codigo: 2, descripcion: 'manzanas', precio: 54 }

];

}

}

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule],

bootstrap: [AppComponent]

})

export class AppModule {}

export class HomeComponent {

constructor(private articulosService: ArticulosService) {}

ngOnInit() {

console.log(this.articulosService.obtenerArticulos());

}

}

**Correcciones necesarias:**

1. Agregar el decorador @Injectable() al servicio.
2. Registrar el servicio en el array de providers del módulo.
3. Corregir el nombre del método en el componente.

**Archivo corregido:**

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root'

})

export class ArticulosService {

retornarArticulos() {

return [

{ codigo: 1, descripcion: 'patatas', precio: 12.33 },

{ codigo: 2, descripcion: 'manzanas', precio: 54 }

];

}

}

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule],

providers: [ArticulosService],

bootstrap: [AppComponent]

})

export class AppModule {}

export class HomeComponent {

constructor(private articulosService: ArticulosService) {}

ngOnInit() {

console.log(this.articulosService.retornarArticulos());

}

}

## **4. Actividad: Aplicar Operadores RxJS (filter, map, tap, mergeMap)**

**Ubicación:** Diapositiva "Operadores RxJS"  
**Duración:** 10 minutos

### **Consigna**

1. Crear un observable con una lista de usuarios.
2. Usar operadores como filter, map y tap para procesar los datos y filtrar usuarios mayores de 25 años.
3. Mostrar los resultados en la consola.

### **Posible Solución**

import { of } from 'rxjs';

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

const usuarios = of(

{ id: 1, nombre: 'Juan', edad: 40 },

{ id: 2, nombre: 'Sandra', edad: 20 },

{ id: 3, nombre: 'Marta', edad: 30 }

);

usuarios.pipe(

filter(usuario => usuario.edad > 25),

map(usuario => usuario.nombre),

tap(nombre => console.log('Usuario:', nombre))

).subscribe();

## **5. Actividad: Implementación de Pipe Async**

**Ubicación:** Diapositiva "Pipe Async"  
**Duración:** 10 minutos

### **Consigna**

1. Crear un observable que devuelva un valor después de un retraso de 2 segundos.
2. Usar el pipe async en una vista para mostrar los datos.

### **Posible Solución**

**Archivo: app.component.ts**

import { Component } from '@angular/core';

import { Observable, of } from 'rxjs';

import { delay } from 'rxjs/operators';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

observable: Observable<string>;

ngOnInit() {

this.observable = of('Hola, Pipe Async').pipe(delay(2000));

}

}

**Archivo: app.component.html**

<h1>{{ observable | async }}</h1>