



Desarrollo y Programación de
Aplicaciones Avanzadas con Angular

Formación online



Módulo 7. Angular Services

Módulo 7. Angular Services

¿Qué es un servicio?

Un servicio es una clase decorada encargada de un propósito concreto, algo específico

Un servicio se “inyecta” en un componente mediante Inyección de dependencias

Angular distingue componentes y servicios para incrementar la modularidad y la reusabilidad

Un uso común de los servicios es manejar datos y conectar con nuestra API para guardar/editar/añadir/eliminar datos de nuestra aplicación backend. El componente delega esta responsabilidad y asume que el servicio cumplirá su cometido perfectamente.

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

@Injectable({
  providedIn: 'root'
})
export class UserService {

  constructor() { }
}
```

Módulo 7. Angular Services

Partes de un Servicio

Este decorador hace el servicio inyectable por el sistema de inyección de dependencias de Angular

providedIn determina en qué módulo será registrado y si será un singleton.

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

@Injectable({
  providedIn: 'root'
})
export class PruebaService {

  constructor() { }
}
```

Vemos que es una clase TypeScript

Módulo 7. Angular Services

Inyección de dependencias

Una **dependencia significa que algo depende de otro algo**. Un componente puede tener dependencias de servicios, es decir, **necesita a los servicios** para poder llevar a cabo algunas tareas

Si no usamos un sistema de inyección de independencias, tendríamos que resolver esas dependencias nosotros mismos (creando instancias a esos servicios) y esto puede ser muy tedioso, porque un servicio puede tener dependencia de otro servicio y de otro... y sería un no terminar!

Angular incorpora un sistema de inyección de dependencias extremadamente sencillo. Creamos una clase, la decoramos con `@Injectable`, y lo añadimos al constructor del componente, voilá! Nada más!

Módulo 7. Angular Services

Inyección de dependencias

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

@Injectable({
  providedIn: 'root'
})
export class PruebaService {

  constructor() { }
}
```

```
import { Component } from '@angular/core';
import { PruebaService } from './prueba.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class PruebaComponent {

  imageUrl = 'https://google...';
  inputName = 'Valor por defecto!';

  // Servicio resuelto por inyección de dependencias!
  constructor(private pruebaService: PruebaService) {
    console.log(this.pruebaService); // OK!
    // const pruebaService = new PruebaService(); // NO!
  }
}
```

Patrón Singleton

Singleton es un patrón de diseño en la ingeniería del software, el cual garantiza que una clase solo tenga una única instancia, siendo accesible de forma global.

Un servicio Singleton es un servicio del cual sólo existe una instancia en toda la App, lo cual nos asegura que comparta su estado y lógica, pudiendo compartir datos, rehusar código...

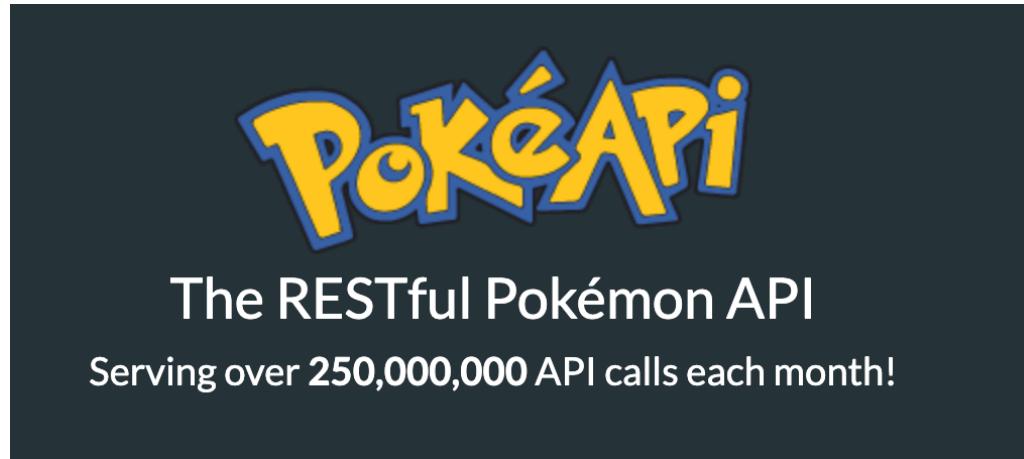
Hay dos formas de crear un servicio Singleton en Angular:

- Mediante la propiedad providedIn del decorator @Injectable con valor “root”
- Registrando el servicio en el array de providers de AppModule

* Si registras el servicio en módulos distintos al AppModule y que no se importan en el AppModule, se crearán distintas instancias del servicio.

Módulo 7. Angular Services

PokéAPI

The screenshot shows the homepage of PokeAPI. At the top, there's a dark header with the "PokeAPI" logo in yellow and blue. Below it, the text "The RESTful Pokémon API" and "Serving over 250,000,000 API calls each month!" is displayed. The main content area has a light green background. It contains the text "All the Pokémon data you'll ever need in one place, easily accessible through a modern RESTful API." followed by a blue "Check out the docs!" button.

The RESTful Pokémon API
Serving over 250,000,000 API calls each month!

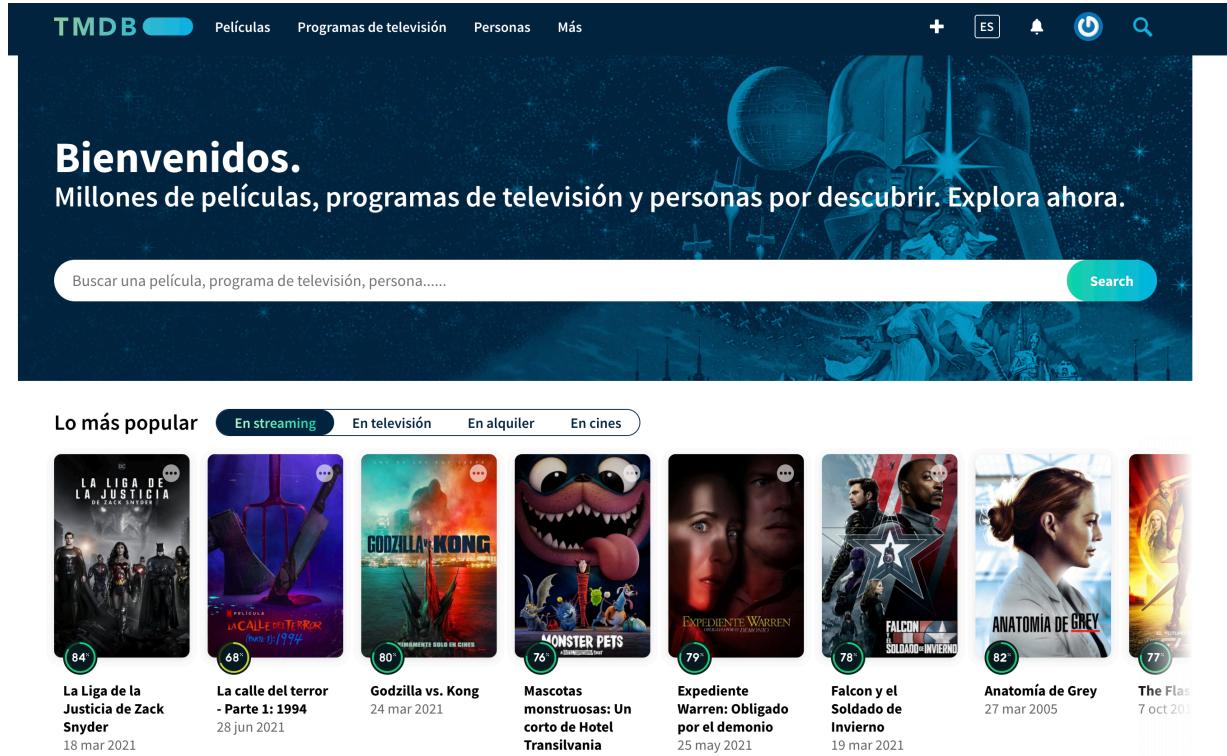
All the Pokémon data you'll ever need in one place,
easily accessible through a modern RESTful API.

[Check out the docs!](#)

- API solo de consulta de datos
 - <https://pokeapi.co/>
- *No requiere autenticación*
- *Se puede utilizar de manera complementaria, el siguiente repositorio de imágenes:*
 - [https://github.com/PokeAPI/sprites/
tree/master/sprites/pokemon](https://github.com/PokeAPI/sprites/tree/master/sprites/pokemon)

Módulo 7. Angular Services

TheMovieDB API



- Web TheMovieDB (registro):
 - <https://www.themoviedb.org>
- *Configuración API (private)*
 - <https://www.themoviedb.org/settings/api>
- *Documentación API*
 - <https://developers.themoviedb.org/3>

Importar módulo HttpClientModule

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Injectar dependencia en Servicio

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}
```

- Debemos injectar la dependencia de la clase HttpClient, en el constructor del servicio donde queremos llevar a cabo las peticiones a la API.

Módulo 7. Angular Services

Http Get Request

```
@Injectable()  
export class ConfigService {  
  constructor(private http: HttpClient) {}  
}
```

```
configUrl = 'assets/config.json';  
  
getConfig() {  
  return this.http.get<Config>(this.configUrl);  
}
```

- Para hacer referencia a la dependencia inyectada en el constructor, utilizamos la palabra reservada “this”.
- En este caso, al ser una petición “get” solo debemos facilitar la URL, con los parámetros si los hubiera concatenados.
- Debemos indicar entre <> el tipo de dato que devuelve la petición, en este ejemplo, un dato de tipo interface Config.

Suscripción a petición Http

```
config: Config | undefined;

showConfig() {
  this.configService.getConfig()
    // clone the data object, using its known Config shape
    .subscribe((data: Config) => this.config = { ...data });
}
```

LABORATORIO: Selección de Pokemon

EJERCICIO 1

Según el ejercicio visto sobre el listado de Pokémon, aplicar el uso de @Input y @Output sobre el listado de Pokemon, permitiendo seleccionar un pokémon (en el PokemonItemComponent) y que aparezca en el componente Pokémon list qué pokémon es el seleccionado. Debe conectarse con la API de Pokémon y tanto en el input como en el output debe pasarse un objeto Pokémon con la información del pokémon.

LABORATORIO: Listado de Actores

Se pide:

1. A partir del JSON de actores populares (/person/popular), generar las interfaces en la web json2ts.com.
2. Debemos crear el fichero models/person-popular.interface.ts > dentro del fichero pegamos las interfaces generadas en el paso anterior.
3. Generamos el componente padre “person-popular-list”: ng g c person-popular-list - -skip-tests
4. Generamos el componente padre “person-popular-item”: ng g c person-popular-item - -skip-tests
5. Definir en el componente padre la lista de actores, mediante: JSON.parse(`...`)
6. Incluir el componente “app-person-popular-list” dentro del fichero app.component.html: <app-person-popular-list></app-person-popular-list>
7. Incluir el componente “person-popular-item” dentro del fichero “person-popular-list.component.html”: <app-person-popular-item></app-person-popular-item>
8. Definir el @Input y el @Output dentro del componente person-popular-item.component.ts:
 - @Input() person: Person | undefined;
 - @Output() personFavEmitter = new EventEmitter<Person>();
9. Incluir el input y output en la invocación al elemento hijo (ir al fichero person-popular-list.component.html):
<app-person-popular-item [person] = “person” (personFavEmitter) = “selectedFav(\$event)”></app-person-popular-item>
10. Debemos asegurarnos que en el componente padre (fichero: person-popular-item.component.ts), tenemos el JSON con la lista de actores, cargado en una variable que es un atributo de nuestra clase, por ejemplo listadoActores: Person[];
11. Podemos por tanto aplicar la directiva ngFor para iterar sobre esa lista de actores (ir al fichero person-popular-item.component.html):
<app-person-popular-item *ngFor = “let actor of listadoActores” [person] = “actor” (personFavEmitter) = “selectedFav(\$event)”></app-person-popular-item>