

Angular

Servicios para agrupar la l-gica de negocio

CertiDevs

 ndice de contenidos

1. Servicios	1
2. Creaci�n y estructura de un servicio	1
3. Inyecci�n de dependencias	2
4. �mbito y ciclo de vida de un servicio	2
5. Comunicaci�n entre componentes utilizando servicios	3
6. Ejemplo 1	5
6.1. Paso 1: Crear un servicio	5
6.2. Paso 2: Utilizar el servicio en un componente	6
6.3. Paso 3: Mostrar el mensaje en otro componente	6
6.4. Paso 4: Agregar los componentes al m�dulo principal	7
6.5. Paso 5: Utilizar los componentes en 'app.component.html'	8
6.6. Paso 6: Ejecutar la aplicaci�n	8
7. Ejemplo 2	8
7.1. Paso 1: Crear un servicio CRUD	8
7.2. Paso 2: Crear un componente para utilizar el servicio CRUD	9
7.3. Paso 3: Agregar el componente y FormsModule al m�dulo principal	11
7.4. Paso 4: Utilizar el componente en 'app.component.html'	11
7.5. Paso 5: Ejecutar la aplicaci�n	11
8. Ejemplo 3: leer datos de un API REST	12
8.1. Paso 1: Crear un servicio para obtener datos de la API	12
9. APIs p�blicas	13

1. Servicios

Los servicios en Angular son clases que encapsulan la l gica y los datos espec ficos de una aplicaci n. Se utilizan para compartir informaci n y funcionalidades entre componentes a trav s de la inyecci n de dependencias.

Los servicios se utilizan para organizar y reutilizar c digo, as  como para separar las preocupaciones en una aplicaci n Angular.

Los servicios tambi n son fundamentales para la inyecci n de dependencias en Angular, un patr n de dise o que permite desacoplar las dependencias y mejorar la modularidad y la capacidad de prueba de una aplicaci n.

2. Creaci n y estructura de un servicio

Para crear un servicio en Angular, se utiliza la clase `@Injectable` como decorador de una clase TypeScript.

El decorador `@Injectable` marca la clase como elegible para la inyecci n de dependencias y acepta un objeto de metadatos como argumento.

A continuaci n, se muestra un ejemplo b sico de un servicio en Angular:

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

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

  private users: string[] = [];

  addUser(user: string): void {
    this.users.push(user);
  }

  getUsers(): string[] {
    return this.users;
  }
}
```

En este ejemplo, el servicio `UserService` encapsula la l gica y el estado para agregar y recuperar usuarios.

El objeto de metadatos en el decorador `@Injectable` contiene la propiedad `providedIn`, que indica d nde se proporciona el servicio y c mo se administran las instancias del servicio.

3. Inyección de dependencias

La inyección de dependencias (DI) es un patrón de diseño que permite desacoplar las dependencias entre los objetos y mejorar la modularidad y la capacidad de prueba de una aplicación.

Angular utiliza la inyección de dependencias para proporcionar instancias de servicios a los componentes y otras clases que los requieren.

Para utilizar un servicio en un componente u otra clase, se debe agregar el servicio como parámetro en el constructor de la clase. Angular automáticamente proporcionará una instancia del servicio al crear la clase.

A continuación se muestra un ejemplo de cómo inyectar el servicio `UserService` en un componente:

```
import { Component } from '@angular/core';
import { UserService } from './user.service';

@Component({
  selector: 'app-user',
  templateUrl: './user.component.html'
})
export class UserComponent {
  constructor(private userService: UserService) {}

  addUser(user: string): void {
    this.userService.addUser(user);
  }
  getUsers(): string[] {
    return this.userService.getUsers();
  }
}
```

En este ejemplo, el servicio `UserService` se inyecta en el componente `UserComponent` utilizando el constructor de la clase.

El servicio se asigna a una propiedad privada de la clase y se utiliza para agregar y recuperar usuarios en los métodos del componente.

4. Alcance y ciclo de vida de un servicio

El alcance y el ciclo de vida de un servicio en Angular están determinados por la propiedad `providedIn` en el objeto de metadatos del decorador `@Injectable`.

Esta propiedad indica dónde se proporciona el servicio y cómo se administran las instancias del servicio.

providedIn: 'root': Esta opción indica que el servicio se proporciona en el nivel de la aplicación y se crea una instancia única del servicio (singleton) para toda la aplicación. Esta opción es la predeterminada y se recomienda para la mayoría de los servicios que requieren un estado compartido a nivel de aplicación.

providedIn: SomeModule: Esta opción indica que el servicio se proporciona en el nivel del módulo especificado (en este caso, SomeModule). Angular creará una instancia única del servicio para cada módulo que lo requiera. Cuando se utiliza esta opción, es necesario importar y agregar el servicio a la lista de providers en el módulo correspondiente.

```
import { NgModule } from '@angular/core';
import { UserService } from './user.service';

@NgModule({
  providers: [UserService]
})
export class SomeModule { }
```

Proporcionar el servicio en un componente: Se puede proporcionar un servicio a nivel de componente agregándolo a la lista de providers en el decorador **@Component**.

Esto creará una nueva instancia del servicio para cada instancia del componente.

```
import { Component } from '@angular/core';
import { UserService } from './user.service';

@Component({
  selector: 'app-users',
  templateUrl: './users.component.html',
  providers: [UserService]
})
export class UsersComponent {

}
```

Tener en cuenta el ámbito y el ciclo de vida del servicio es importante para administrar correctamente las instancias del servicio y garantizar un comportamiento adecuado y un rendimiento óptimo en la aplicación.

5. Comunicación entre componentes utilizando servicios

Los servicios pueden ser utilizados para facilitar la comunicación entre componentes, especialmente aquellos que no están relacionados jerárquicamente.

Un enfoque común es utilizar un servicio para almacenar y compartir datos entre componentes, o

emitir eventos utilizando la clase `EventEmitter`.

Por ejemplo, supongamos que queremos que dos componentes no relacionados compartan una lista de usuarios. Podemos utilizar un servicio para almacenar la lista de usuarios y proporcionar métodos para agregar usuarios y suscribirse a cambios en la lista:

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

@Injectable({
  providedIn: 'root'
})
export class UserService {
  private usersSubject = new BehaviorSubject<string[]>([]);

  addUser(user: string): void {
    const users = this.usersSubject.getValue();
    users.push(user);
    this.usersSubject.next(users);
  }

  getUsersObservable() {
    return this.usersSubject.asObservable();
  }
}
```

En este ejemplo, el servicio `UserService` utiliza la clase `BehaviorSubject` de la biblioteca RxJS para mantener un estado compartido de la lista de usuarios y notificar a los componentes suscritos cuando la lista cambia.

Los componentes pueden suscribirse al `Observable` devuelto por `getUsersObservable()` para recibir actualizaciones de la lista de usuarios:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { UserService } from '../user.service';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-user-list',
  templateUrl: '../user-list.component.html'
})
export class UserListComponent implements OnInit, OnDestroy {
  users: string[] = [];
  private subscription: Subscription;

  constructor(private userService: UserService) {}

  ngOnInit() {
    this.subscription = this.userService.getUsersObservable().subscribe(users => {
      this.users = users;
    });
  }

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

```

    });
  }
  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
}

```

En resumen, los servicios en Angular desempeñan un papel crucial en la organización, reutilización de código y separación de preocupaciones en las aplicaciones. Los servicios también son fundamentales para la inyección de dependencias, lo que permite desacoplar las dependencias y mejorar la modularidad y capacidad de prueba de una aplicación.

6. Ejemplo 1

En este ejercicio, aprenderemos cómo crear y utilizar servicios en Angular.

Los servicios son una forma efectiva de compartir datos y funciones entre componentes y mantener la lógica de la aplicación separada y reutilizable.

6.1. Paso 1: Crear un servicio

Vamos a crear un servicio llamado 'example-service' utilizando el siguiente comando:

```
ng generate service example-service
```

Esto creará dos archivos en la carpeta 'src/app', 'example-service.service.ts' y 'example-service.service.spec.ts'.

Abre el archivo 'example-service.service.ts' y reemplaza su contenido con el siguiente código:

```

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

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

  private messageSource = new BehaviorSubject<string>('Mensaje inicial');
  currentMessage = this.messageSource.asObservable();

  constructor() { }
  changeMessage(message: string) {
    this.messageSource.next(message);
  }
}

```

En el c digo anterior, hemos creado un servicio con un `BehaviorSubject` para manejar el flujo de datos.

Tambi n hemos creado una funci n `changeMessage` para actualizar el mensaje.

6.2. Paso 2: Utilizar el servicio en un componente

Ahora vamos a utilizar este servicio en un componente llamado `'message-sender'`.

Ejecuta el siguiente comando para crear el componente:

```
ng generate component message-sender
```

Abre el archivo `'message-sender.component.ts'` y reemplaza su contenido con el siguiente c digo:

```
import { Component } from '@angular/core';
import { ExampleService } from '../example.service';

@Component({
  selector: 'app-message-sender',
  templateUrl: './message-sender.component.html',
  styleUrls: ['./message-sender.component.css']
})
export class MessageSenderComponent {
  newMessage: string;

  constructor(private exampleService: ExampleService) {}

  sendMessage() {
    this.exampleService.changeMessage(this.newMessage);
  }
}
```

En el c digo anterior, hemos inyectado `ExampleService` en el componente y hemos creado una funci n `sendMessage` para enviar el mensaje al servicio.

Abre el archivo `message-sender.component.html` y reemplaza su contenido con el siguiente c digo:

```
<input [(ngModel)]="newMessage" placeholder="Escribe un mensaje">
<button (click)="sendMessage()">Enviar mensaje</button>
```

6.3. Paso 3: Mostrar el mensaje en otro componente

Vamos a crear otro componente llamado `message-receiver` para mostrar el mensaje enviado a trav s del servicio.

Ejecuta el siguiente comando para crear el componente:

```
ng generate component message-receiver
```

Abre el archivo `message-receiver.component.ts` y reemplaza su contenido con el siguiente código:

```
import { Component, OnInit } from '@angular/core';
import { ExampleService } from '../example-service.service';

@Component({
  selector: 'app-message-receiver',
  templateUrl: './message-receiver.component.html',
  styleUrls: ['./message-receiver.component.css']
})
export class MessageReceiverComponent implements OnInit {
  message: string;

  constructor(private exampleService: ExampleService) {}

  ngOnInit() {
    this.exampleService.currentMessage.subscribe(message => this.message = message);
  }
}
```

En el código anterior, hemos inyectado `ExampleService` y nos hemos suscrito al `BehaviorSubject` para recibir actualizaciones del mensaje.

Abre el archivo `message-receiver.component.html` y reemplaza su contenido con el siguiente código:

```
<h3>Mensaje recibido:</h3>
<p>{{ message }}</p>
```

6.4. Paso 4: Agregar los componentes al módulo principal

Abre `src/app/app.module.ts` e importa los nuevos componentes y `FormsModule`:

```
import { MessageSenderComponent } from '../message-sender/message-sender.component';
import { MessageReceiverComponent } from '../message-receiver/message-receiver.component';
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
```

```

    AppComponent,
    MessageSenderComponent,
    MessageReceiverComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  // ...
})
export class AppModule { }

```

6.5. Paso 5: Utilizar los componentes en 'app.component.html'

Abre el archivo `app.component.html` y reemplaza su contenido con el siguiente código:

```

<app-message-sender></app-message-sender>
<app-message-receiver></app-message-receiver>

```

6.6. Paso 6: Ejecutar la aplicación

Ejecuta el siguiente comando en la terminal para iniciar la aplicación:

```
ng serve
```

Abre tu navegador y visita 'http://localhost:4200'. Verás un campo de texto y un botón para enviar un mensaje, así como un componente que muestra el mensaje enviado. Al enviar un mensaje, el componente 'message-receiver' mostrará el mensaje enviado a través del servicio.

7. Ejemplo 2

En este ejercicio, vamos a crear un servicio Angular que realizará operaciones CRUD (Crear, Leer, Actualizar y Eliminar) en una estructura de datos simple.

Utilizaremos este servicio en un componente para mostrar, agregar, editar y eliminar elementos.

7.1. Paso 1: Crear un servicio CRUD

Ejecuta el siguiente comando para crear un servicio llamado 'crud-service':

```
ng generate service crud-service
```

Abre el archivo 'crud-service.service.ts' y reemplaza su contenido con el siguiente c digo:

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

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

  private items = [
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' },
    { id: 3, name: 'Item 3' },
  ];

  constructor() {}

  getAllItems() {
    return this.items;
  }

  addItem(item) {
    this.items.push(item);
  }

  updateItem(item) {
    const index = this.items.findIndex(i => i.id === item.id);
    if (index > -1) {
      this.items[index] = item;
    }
  }

  deleteItem(itemId) {
    const index = this.items.findIndex(i => i.id === itemId);
    if (index > -1) {
      this.items.splice(index, 1);
    }
  }
}
```

En el c digo anterior, hemos creado un servicio con operaciones CRUD b sicas para manipular una estructura de datos de elementos.

7.2. Paso 2: Crear un componente para utilizar el servicio CRUD

Ejecuta el siguiente comando para crear un componente llamado 'crud-example':

```
ng generate component crud-example
```

Abre el archivo 'crud-example.component.ts' y reemplaza su contenido con el siguiente código:

```
import { Component } from '@angular/core';
import { CrudServiceService } from '../crud-service.service';

@Component({
  selector: 'app-crud-example',
  templateUrl: './crud-example.component.html',
  styleUrls: ['./crud-example.component.css']
})
export class CrudExampleComponent {
  items;
  selectedItem = { id: null, name: '' };
  editMode = false;

  constructor(private crudService: CrudServiceService) {
    this.items = this.crudService.getAllItems();
  }

  addItem() {
    if (!this.editMode) {
      const newItem = { id: Date.now(), name: this.selectedItem.name };
      this.crudService.addItem(newItem);
      this.selectedItem.name = '';
    } else {
      this.crudService.updateItem(this.selectedItem);
      this.selectedItem = { id: null, name: '' };
      this.editMode = false;
    }
  }

  editItem(item) {
    this.selectedItem = { ...item };
    this.editMode = true;
  }

  deleteItem(itemId) {
    this.crudService.deleteItem(itemId);
  }
}
```

En el código anterior, hemos inyectado el servicio 'CrudServiceService' en el componente y hemos creado funciones para realizar operaciones CRUD utilizando el servicio.

Abre el archivo 'crud-example.component.html' y reemplaza su contenido con el siguiente código:

```
<div>
```

```

    <input [(ngModel)]="selectedItem.name" placeholder="Nombre del elemento">
    <button (click)="addItem()">{{ editMode ? 'Actualizar' : 'Agregar' }}</button>
  </div>
  <ul>
    <li *ngFor="let item of items">
      {{ item.name }}
      <button (click)="editItem(item)">Editar</button>
      <button (click)="deleteItem(item.id)">Eliminar</button>
    </li>
  </ul>

```

7.3. Paso 3: Agregar el componente y FormsModule al módulo principal

Abre 'src/app/app.module.ts' e importa el nuevo componente y FormsModule:

```

import { CrudExampleComponent } from './crud-example/crud-example.component';
import { FormsModule } from '@angular/forms';
@NgModule({
  declarations: [
    AppComponent,
    CrudExampleComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  // ...
})
export class AppModule { }

```

7.4. Paso 4: Utilizar el componente en 'app.component.html'

Abre el archivo 'app.component.html' y reemplaza su contenido con el siguiente código:

```
<app-crud-example></app-crud-example>
```

7.5. Paso 5: Ejecutar la aplicación

Ejecuta el siguiente comando en la terminal para iniciar la aplicación:

```
ng serve
```

Abre tu navegador y visita 'http://localhost:4200'. Verás una lista de elementos con la capacidad de agregar, editar y eliminar elementos usando el servicio CRUD.

8. Ejemplo 3: leer datos de un API REST

Vamos a crear un ejemplo de cómo utilizar RxJS en Angular para consumir datos desde una API utilizando `HttpClient` en un servicio Angular.

Para este ejemplo, utilizaremos la API pública JSONPlaceholder para obtener datos de publicaciones (posts).

8.1. Paso 1: Crear un servicio para obtener datos de la API

En un nuevo proyecto, comienza generando un servicio utilizando Angular CLI:

```
ng generate service posts
```

Importa `HttpClient` y `Observable` en el servicio y crea un método para obtener las publicaciones:

```
// src/app/posts.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class PostsService {
  private readonly API_URL = 'https://jsonplaceholder.typicode.com/posts';

  constructor(private http: HttpClient) {}

  getPosts(): Observable<any> {
    return this.http.get<any>(this.API_URL);
  }
}
```

En este ejemplo, hemos creado un servicio `PostsService` que utiliza `HttpClient` para obtener datos de la API.

El método `getPosts()` devuelve un `Observable` que emite los datos de las publicaciones cuando se completa la solicitud HTTP.

Genera un componente para mostrar las publicaciones:

```
ng generate component posts
```

Importa `PostsService` en el componente y suscríbete al `Observable` devuelto por `getPosts()`:

```
// src/app/posts/posts.component.ts
import { Component, OnInit } from '@angular/core';
import { PostsService } from '../posts.service';

@Component({
  selector: 'app-posts',
  templateUrl: './posts.component.html',
  styleUrls: ['./posts.component.css'],
})
export class PostsComponent implements OnInit {
  posts: any[] = [];

  constructor(private postsService: PostsService) {}

  ngOnInit(): void {
    this.postsService.getPosts().subscribe((data) => {
      this.posts = data;
    });
  }
}
```

En este ejemplo, hemos inyectado `PostsService` en el componente `PostsComponent` y nos hemos suscrito al `Observable` devuelto por `getPosts()` en el método `ngOnInit()`.

Cuando el `Observable` emite los datos de las publicaciones, asignamos esos datos al arreglo `this.posts`.

Muestra las publicaciones en la plantilla del componente:

```
<!-- src/app/posts/posts.component.html -->
<ul>
  <li *ngFor="let post of posts">{{ post.title }}</li>
</ul>
```

En este ejemplo, utilizamos la directiva `*ngFor` para iterar sobre el arreglo `posts` y mostrar el título de cada publicación en la lista.

9. APIs públicas

¥ JSONPlaceholder: Como mencionaste, JSONPlaceholder es una API REST sencilla que proporciona datos de prueba en forma de usuarios, publicaciones, comentarios, álbumes, fotos y más. Es útil para aprender y probar Angular sin preocuparse por la lógica de backend.

! URL: <https://jsonplaceholder.typicode.com>

¥ OpenWeatherMap: OpenWeatherMap es una API que proporciona informaci3n del tiempo, incluido el pron3stico del tiempo, las condiciones actuales y los datos hist3ricos.

! URL: <https://openweathermap.org/api>

¥ The Movie Database (TMDB): TMDB es una base de datos de pel3culas y programas de televisi3n que ofrece una API para obtener informaci3n detallada sobre pel3culas, programas de televisi3n, actores y m3s.

! URL: <https://www.themoviedb.org/documentation/api>

¥ PokeAPI: PokeAPI es una API que proporciona datos sobre Pok3mon, incluidas sus estad3sticas, habilidades, evoluciones y m3s. Es ideal para crear aplicaciones relacionadas con Pok3mon y juegos.

! URL: <https://pokeapi.co>

¥ REST Countries: REST Countries es una API que proporciona informaci3n sobre pa3ses, como la capital, la poblaci3n, la moneda y m3s. Es 3til para crear aplicaciones relacionadas con la geograf3a y la demograf3a.

! URL: <https://restcountries.com>

¥ Unsplash: Unsplash es una API que proporciona acceso a millones de fotos gratuitas y de alta calidad. Puedes utilizar esta API para agregar im3genes a tu aplicaci3n Angular.

! URL: <https://unsplash.com/developers>

¥ News API: News API es una API que proporciona noticias de diferentes fuentes y en diferentes categor3as. Puedes utilizar esta API para crear una aplicaci3n de noticias o agregar noticias a tu aplicaci3n existente.

! URL: <https://newsapi.org>

¥ JokeAPI: JokeAPI es una API que proporciona chistes en diferentes categor3as y formatos. Puedes utilizar esta API para agregar humor a tu aplicaci3n Angular.

! URL: <https://jokeapi.dev>

¥ GitHub REST API: GitHub REST API proporciona acceso a datos sobre repositorios, usuarios y eventos de GitHub. Puedes utilizar esta API para crear aplicaciones relacionadas con el desarrollo de software y la colaboraci3n en proyectos.

! URL: <https://docs.github.com/en/rest>

Se pueden encontrar m3s en <https://apistlist.fun>