
ANGULAR 15

CONTENIDO

- ~~Angular~~.
- ~~Instalación~~.
- ~~Commandos~~.
- ~~Estructura (Scaffolding)~~.
- ~~Components~~.
- ~~Routing~~.
- ~~Directives~~.
- ~~Interfaces~~.
- ~~Pipes~~.
- ~~Guards~~.
- ~~Data Binding~~.
- ~~Comunicación entre componentes~~.
- ~~Manipulación del DOM~~.

CONTENIDO

- ~~Angular~~.
- ~~Instalación~~.
- ~~Commandos~~.
- ~~Estructura (Scaffolding)~~.
- ~~Components~~.
- ~~Routing~~.
- ~~Directives~~.
- ~~Interfaces~~.
- ~~Pipes~~.
- ~~Guards~~.
- ~~Data Binding~~.
- ~~Comunicación entre componentes~~.
- ~~Manipulación del DOM~~.
- Services.
- Interceptors.
- async-await.
- RxJS.

Services

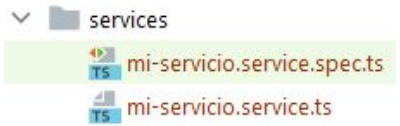
Los Services son una categoría amplia que abarca cualquier valor, función o característica que necesite una aplicación. Un servicio es típicamente una clase con un propósito limitado y bien definido. Debe hacer algo específico y hacerlo bien.

Comando

```
ng generate service service-name [options]
```

Resultado

```
├── .../
├──────── mi-servicio.service.spect.ts
└──────── mi-servicio.service.ts
```




```
...
export class MiServicioService {

    constructor() { }
}
```

Services

Uso

- HTTP verbs.
  - GET.
  - POST.
  - PUT.
  - DELETE.
- Comunicación entre componentes.
- Libraries.
- Servicios de uso común.
- Etc.

Services

HTTP verbs

- GET.
- POST.
- PUT.
- DELETE.

Services

HTTP verbs - GET

```
...
export class MiServicioService {

      constructor(private http: HttpClient) { }

      getFunction(): Observable<any[]> {
            const headers = new HttpHeaders({
               'Content-Type': 'application/json',
               'x-access-token': String(sessionStorage.getItem('TOKEN'))
            });

            let apiUrl = 'https://jsonplaceholder.typicode.com/comments?postId=1';
            let apiUrl = 'https://jsonplaceholder.typicode.com/posts';
            return this.http.get<any[]>(apiUrl, {headers});
      }
}
```

Services

HTTP verbs - POST

```
...
export class MiServicioService {

      constructor(private http: HttpClient) { }

      postFunction(body: Object): Observable<any> {
            const headers = new HttpHeaders({
               'Content-Type': 'application/json',
               'x-access-token': String(sessionStorage.getItem('TOKEN'))
            });

            let apiUrl = 'https://jsonplaceholder.typicode.com/posts;
            return this.http.post(apiUrl, JSON.stringify(body), {headers});
      }
}
```

Services

HTTP verbs - PUT

```
...
export class MiServicioService {

      constructor(private http: HttpClient) { }

      putFunction(id: String, body: Object): Observable<any> {
            const headers = new HttpHeaders({
               'Content-Type': 'application/json',
               'x-access-token': String(sessionStorage.getItem('TOKEN'))
            });

            let apiUrl = `https://jsonplaceholder.typicode.com/posts/${id}`;
            return this.http.put(apiUrl, JSON.stringify(body), {headers});
      }
}
```

Services

HTTP verbs - DELETE

```
...
export class MiServicioService {

      constructor(private http: HttpClient) { }

      deleteFunction(id: String): Observable<any> {
            const headers = new HttpHeaders({
               'Content-Type': 'application/json',
               'x-access-token': String(sessionStorage.getItem('TOKEN'))
            });

            let apiUrl = `https://jsonplaceholder.typicode.com/posts/${id}`;
            return this.http.delete(apiUrl, {headers});
      }
}
```

Services

HTTP verbs - Uso

```
...
export class MiComponenteComponent {

	constructor(private _miServicio: MiServicioService) {
		this.doGet();
	}

	doGet(){
		this._miServicio.getFunction().subscribe({
			next: (data) => {
				console.warn(data)
			},
			error: (e) => {
				console.warn(e)
			}
		});
	}
}
```

Services

HTTP verbs - Uso

```
...
export class MiComponenteComponent {

	constructor(private _miServicio: MiServicioService) {
		this.doDelete();
	}

	doDelete(){
		this._miServicio.deleteFunction('1').subscribe({
			next: (data) => {
				console.warn(data)
			},
			error: (e) => {
				console.warn(e)
			}
		});
	}
}
```

Services

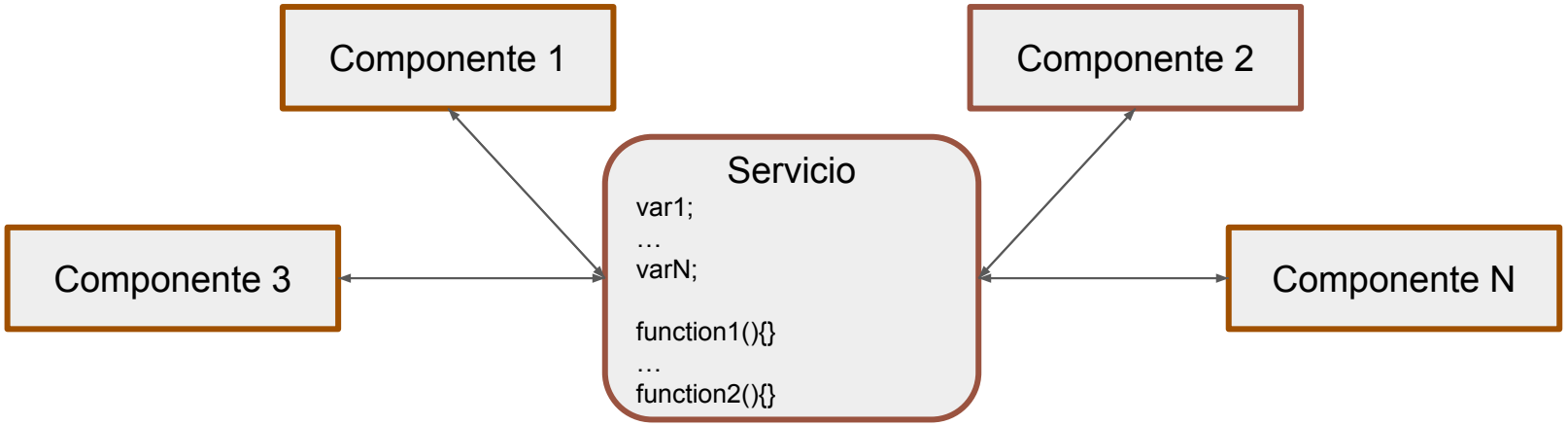
Comunicación entre componentes

Respecto a los otros métodos de comunicación entre componentes, las limitaciones que poseen es que la relación debe ser directa, es decir, componente a componente.
Si los componentes no están directamente relacionados, una opción es utilizar un servicio compartido para la comunicación. El servicio actúa como un intermediario entre los componentes y puede almacenar y proporcionar datos y métodos compartidos.

Services

Comunicación entre componentes

Respecto a los otros métodos de comunicación entre componentes, las limitaciones que poseen es que la relación debe ser directa, es decir, componente a componente.
Si los componentes no están directamente relacionados, una opción es utilizar un servicio compartido para la comunicación. El servicio actúa como un intermediario entre los componentes y puede almacenar y proporcionar datos y métodos compartidos.



Interceptors

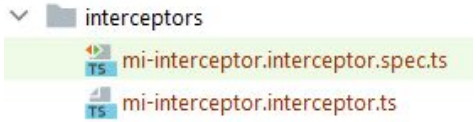
Los Interceptors son, por lo general, funciones que se pueden ejecutar para cada request y tienen amplias capacidades para afectar el contenido y el flujo general de las requests y los responses.

Comando

```
ng generate interceptor interceptor-name [options]
```

Resultado

```
├── .../
├──────── mi-interceptor.interceptor.spect.ts
└──────── mi-interceptor.interceptor.ts
```



Interceptors

Resultado

```
...
@Injectable()
export class MiInterceptorInterceptor implements HttpInterceptor {

    constructor() {}

    intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
                return next.handle(request);
    }
}
```

Interceptors

Uso

```
intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
      const basename = request.url;
      const params = request.body;
      const start = performance.now();
      let end = basename.split('/').pop()?.search('json');

      console.debug('%c REST call: ', 'background: #000; color: #fff', basename, params);

      return next.handle(request).pipe(
tap((event: HttpEvent<any>) => {
            if (event instanceof HttpResponse) {
                  const end = performance.now();
                  const data: any = event.body;

                  console.debug('%c REST time: ', 'background: #000; color: #1e90ff', basename, 'tiempo: ',
                  end - start, 'ms');
                  console.debug('%c REST response: ', 'background: #000; color: #ff0', basename, data);
            }
})
      );
}
```

Async-Await

La programación asíncrona permite que ciertas operaciones, como la lectura de archivos o las **solicitudes de red**, se realicen sin bloquear la ejecución del programa. Esto significa que mientras se espera la respuesta de una operación, el programa puede continuar ejecutando otras tareas.
El async-await nos permite ejecutar código asincrónico como sincronico.

Async-Await

Uso

```
// Asincrónico
getFunction() {
      console.log('A')
      this._miServicio.getFunction().subscribe({
            next: (data) => { console.log('B') },
            error: (e) => { … }
      });
      console.log('C')
}
// Sincrónico
async getFunction() {
      console.log('A')
      const resp: any= await firstValueFrom(this.miServicioService.getFunction(1));
      const resp: any= await lastValueFrom(this.miServicioService.getFunction(1));
      console.log(await firstValueFrom(this.miServicioService.getFunction(1)))
      console.log('C')

}
```

RxJS

RxJS es una biblioteca para programación reactiva que utiliza Observables, para facilitar la composición de código asincrónico o basado en devoluciones de llamadas. Este proyecto es una reescritura de Reactive-Extensions/RxJS con mejor rendimiento, mejor modularidad, mejores pilas de llamadas depurables, mientras que se mantiene en gran medida compatible con versiones anteriores, con algunos cambios importantes que reducen la superficie de la API.

¿Que vamos a ver?

- Observables.
- Operadores.
    - Operadores de creación.
    - Operadores de transformación.
    - Operadores de manejo de errores.
    - Operadores de filtro.

RxJS

Observables

Permiten trabajar con datos que se emiten de forma asincrónica (como eventos de usuario, peticiones HTTP, etc.). Se suscriben a los datos, y cada vez que estos se emiten, el observable notifica a sus suscriptores.

```
getData(): Observable<string[]> {
  return new Observable((observer) => {
    observer.next(['Dato 1', 'Dato 2', 'Dato 3']);
    observer.complete();
  });
}

this.getData().subscribe((data) => {
  console.log(data);  // Salida: ['Dato 1', 'Dato 2', 'Dato 3']
});
```

RxJS

Pipe

Es una forma de combinar múltiples operadores de RxJS en una sola secuencia. Permite hacer que el flujo de datos pase por distintas transformaciones de manera sencilla.

```
getData(): Observable<string[]> {
  return new Observable((observer) => {
    observer.next(['Dato 1', 'Dato 2', 'Dato 3']);
    observer.complete();
  }).pipe();
}

this.getData().subscribe((data) => {
  console.log(data);  // Salida: ['Dato 1', 'Dato 2', 'Dato 3']
});
```

RxJS

Operadores de creación - of

Es una función de creación que permite crear un observable a partir de valores específicos. Útil para pruebas y ejemplos simples.

```
getData(): Observable<string[]> {
  return of(['Dato 1', 'Dato 2', 'Dato 3']);
}

this.getData().subscribe((data) => {
  console.log(data);  // Salida: ['Dato 1', 'Dato 2', 'Dato 3']
});
```

RxJS

Operadores de transformación - map

Operador que transforma cada valor emitido por el observable aplicando una función, similar a .map de los arrays.

```
getData(): Observable<string[]> {
  return of(['Dato 1', 'Dato 2', 'Dato 3']).pipe(
    map((data) => data.map(d => d.toUpperCase()))
  );
}

this.getData().subscribe((data) => {
  console.log(data);  // Salida: ['DATO 1', 'DATO 2', 'DATO 3']
});
```

RxJS

Operadores de transformación - concatMap

Opera secuencialmente, ejecutando una función que retorna un observable para cada valor emitido. Espera a que cada observable finalice antes de ejecutar el siguiente.

```
  getData(): Observable<string[]> {
  return of(['Dato 1', 'Dato 2', 'Dato 3']).pipe(
    concatMap((dato) => {
      return of(`${dato} procesado`);  // Simula una operación asincrónica
    })
  );
}

this.getData().subscribe((data) => {
  console.log(data);  // Salida: 'Dato 1 procesado', 'Dato 2 procesado', 'Dato 3 procesado'
});
```

RxJS

Operadores de manejo de errores - catchError

Permite manejar errores en una cadena de operadores sin detener la secuencia completa. Es común en observables que manejan peticiones HTTP.

```
getData(): Observable<string[]> {
  return of(['Dato 1', 'Dato 2', 'Dato 3']).pipe(
    map((data) => {
      if (data.includes('Dato 2')) {
        throw new Error('Error en Dato 2');
      }
      return data;
    }),
```

```
catchError((error) => {
      console.error(error);
      return of(['Error']);  // Devuelve un valor alternativo
    })
  );
}

this.getData().subscribe((data) => {
  console.log(data);  // Salida: ['Error']
});
```

RxJS

Operadores de filtro - filter

Filtra los valores emitidos por el observable en función de una condición dada, emitiendo solo aquellos valores que cumplen dicha condición.

```
getData(): Observable<string[]> {
  return of(['Dato 1', 'Dato 2', 'Dato 3']).pipe(
    filter((dato) => dato.includes('Dato 1'))
  );
}

this.getData().subscribe((data) => {
  console.log(data);  // Salida: ['Dato 1']
});
```

RxJS

Operadores de filtro - debounce

Es útil para controlar la frecuencia con la que se emiten valores, especialmente en entradas de usuario, ignorando los valores emitidos si ocurre otra emisión en el intervalo de tiempo dado.

```
getData(): Observable<string[]> {
  return of('Dato 1', 'Dato 2', 'Dato 3').pipe(
    debounceTime(500),  // Espera 500ms después de la última emisión
    map((dato) => dato.toUpperCase())
  );
}

this.getData().subscribe((data) => {
  console.log(data);  // Salida: 'DATO 1' (después de 500ms)
});
```