Angular S4: Services

Después de esta lección podrás:

1. Entender y crear servicios en Angular.
2. Acceder a datos compartidos en tu aplicación.
3. Comunicar componentes sin relación padre e hijo.
4. Usar el modulo HttpClient para lanzar peticiones contra una API .

Ahora que ya sabemos pasar información de unos componentes a otros, vamos a aprender como compartir información de otra manera. ¿Qué ocurre si la información que hay que compartir tiene que estar disponible para muchos componentes? ¿Nos lo vamos pasando de padres a hijos, en cadena?

Pues para esta función existen los **Servicios de Angular**, son artefactos diseñados para compartir información entre varios componentes. La información pueden ser: variables, funciones, etc.

Angular Service

Un servicio de angular, se instancia **una única vez** (*singleton*). Esto es lo que permite que se pueda compartir información entre varios componentes, ya que la **instancia** que consultan y comparten, siempre es la misma.

Un servicio no es un componente, y funcionan de manera transversal en nuestra aplicación, por eso es importante sobre qué modulo declararlos a la hora de proveer datos, ya que en caso de declararlos en varios módulos estaríamos cometiendo el error de crear varias varias el servicio (varias instancias), esto haría imposible compartir información entre los componentes.

A continuación plantearemos un pequeño proyecto, en el que crearemos un componente para listar mensajes, y otro componente para crearlos. Como ya sabemos comunicarlos mediante **input/output**, ahora vamos a ver como se comunicarían a través de un servicio.

Servicios para compartir datos

Vamos a crearnos un proyecto base sobre el que trabajar:

```
ng new message-list-app
cd message-list-app
cd src/app
```

Y a continuación, crearemos los dos componentes: `message-list` y `new-message` además del servicio `messages`:

```
ng generate component message-list
ng generate component new-message
ng generate service messages
```

Cómo en otras ocasiones, limpiamos el **app.component.html**:

```
<app-message-list></app-message-list>
<app-new-message></app-message>
```

Y preparamos nuestro **app.module.ts** para poder declarar dichos componentes, así como generar el **provider** de nuestro servicio `messages`:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { MessageListComponent } from './message-list/message-list.component';
import { NewMessageComponent } from './new-message/new-message.component';

import { MessagesService } from './messages.service';

@NgModule({
  declarations: [
    AppComponent,
    MessageListComponent,
    NewMessageComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [MessagesService], // Aquí proveemos nuestro servicio al módulo
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Dentro de nuestro servicio, debemos declarar las **variables** y **funciones compartidas**. Si os fijáis, por defecto los servicios se generan de manera que el proveedor es el '*root*'. Para mejorar la escalabilidad de nuestra aplicación, seremos nosotros los que decidimos en qué módulo proveerlo. Para ello modificamos `@Injectable`, como se muestra a continuación:

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

// @Injectable({
//   providedIn: 'root'
// })
@Injectable()
export class MessagesService {
  messageList: string[];

  constructor() {
    this.messageList = ['Hola!', 'Soy Pedro', 'Cómo estás?'];
  }

  getMessageList() {
    return this.messageList;
  }

  pushNewMessage(message: string) {
    this.messageList.push(message);
  }
}
```

Ya tenemos un servicio para compartir una variable con los mensajes, y dos funciones para listarlos y crear uno nuevo.

Por parte del componente listado, expondremos en su vista el listado compartido, en el **message-list.component.html**:

```
<h3>Lista de mensajes</h3>
<div *ngIf="list.length">
  <p *ngFor="let message of list">{{ message }}</p>
</div>
```

Y a nivel de lógica en **message-list.component.ts**, tendremos que inyectar el servicio en el constructor, para poder tener acceso a sus variables compartidas en el componente:

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

```
import { MessagesService } from '../messages.service';

@Component({
  selector: 'app-message-list',
  templateUrl: './message-list.component.html',
  styleUrls: ['./message-list.component.scss']
})
export class MessageListComponent implements OnInit {
  list: string[]; // Listado local del componente

  // Inyección de dependencia del servicio
  constructor(messagesService: MessagesService) {
    this.list = messagesService.getMessageList();
  }

  ngOnInit() {

  }

}
```

Repasemos hasta este punto, hemos declarado un listado en un servicio, y lo hemos visualizado en un componente. Ahora vamos a meter nuevos mensajes en el listado, desde otro componente. ¡Y todo esto ahorrándonos los input/output!

Para el componente de creación de mensajes, vamos a crear un pequeño formulario en **new-message.component.html**:

```
<h3>Nuevo mensaje</h3>
<form novalidate (ngSubmit)="onSubmit()" [formGroup]="messageForm">
  <label>Mensaje: <input type="text" formControlName="message" /></label>
  <button class="button" type="submit">Crear</button>
</form>
```

Y en su fichero de lógica, vamos a crear mensajes en el servicio, lo que provocará que el componente del listado se entere de los nuevos mensajes.

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

import { MessagesService } from '../messages.service';
```

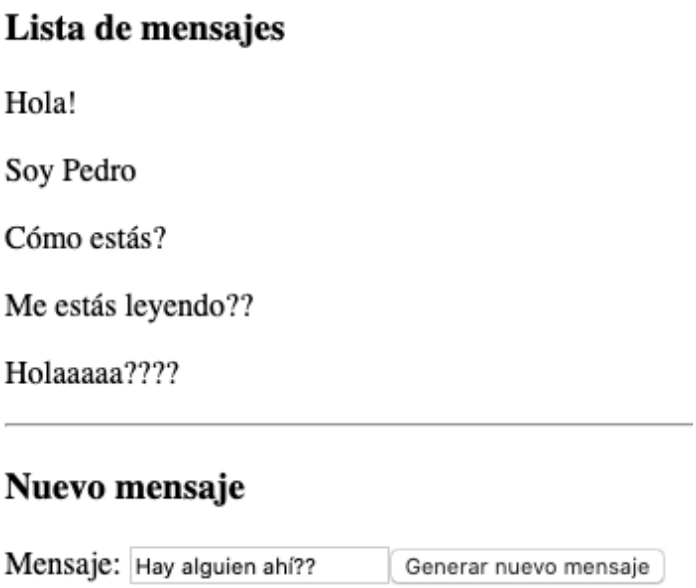
```
@Component({
  selector: 'app-new-message',
  templateUrl: './new-message.component.html',
  styleUrls: ['./new-message.component.scss']
})
export class NewMessageComponent implements OnInit {
  messageForm: FormGroup;

  constructor(private messagesService: MessagesService) {
    this.messageForm = new FormGroup({
      message: new FormControl('', Validators.minLength(2)),
    });
  }

  ngOnInit() {}

  onSubmit(): void {
    this.messagesService.pushNewMessage(this.messageForm.value.message);
    this.messageForm.reset();
  }
}
```

Finalmente, nos quedará una aplicación tal que así:



Y ya hemos aprendido otra manera de **comunicarnos** entre componentes. Por lo general, usaremos siempre el patrón **input/output** entre componentes dentro de un mismo módulo.

Los servicios son útiles para información muy general que necesita ser compartida por varios componentes.

Es por eso que en Angular los servicios se deben usar para **acceder a los datos del API**. La idea consiste en crear funciones que atacarán al servidor y devolverán los datos.

Lo bueno de crear un **data-service**, es que puede ser usado por varios componentes del mismo módulo, para extraer la información del servidor. En la siguiente lección veremos como usar el **HttpClient** para montar un **servicio de datos** que recupere información del *backend*.

Servicios para hacer peticiones a APIS

Lo primero como siempre nos crearemos un proyecto para poder seguir la clase con un ejemplo:

```
ng new htttp-app

cd htttp-app

cd src/app

ng generate component request-example
```

Por fin hemos llegado a una de las partes más importantes de Angular, la comunicación con el servidor. Para ello tenemos que trabajar algunos conceptos como el HttpClient.

Angular HttpClient

Hagamos algunas peticones rápidas para que conozcamos los conceptos de antemano y podamos avanzar sin preocupaciones.

En primer lugar, importamos el **HttpClientModule** en nuestro **app.module.ts**:

```
import { HttpClientModule } from '@angular/common/http';
```

Y lo importamos:

```
imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
```

El HttpClientModule es un modulo de Angular que nos ayudará a realizar peticiones contra una API, lo bueno de utilizar Frameworks como angular es que parte del trabajo nos lo da hecho.

¡Ahora podemos crear servicios llenos de peticiones! Para ello vamos a utilizar una API pública, y en este caso utilizaremos la API de Rick y Morty. Porque... ¿a quién no le gusta Rick y Morty?

Nuestro EndPoint sobre el que atacaremos es:

```
https://rickandmortyapi.com/api/character/
```

Y en caso de que queramos solicitar datos "paginados" para que no vengan muchos resultados, podemos hacer:

```
https://rickandmortyapi.com/api/character/?page=2
```

Después de tener nuestra estructura de proyecto crearemos un Servicio dentro de nuestro componente ya que de momento soló lo consumiremos desde ahí:

```
cd request-example
ng generate service services/request-example
```

Y comenzamos a trabajar con nuestro **request-example.service.ts** para manejar una sola solicitud a los personajes:

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

//EndPoint base sobre el que atacaremos
const baseUrl = 'https://rickandmortyapi.com/api/';

//La petición character
const characterUrl = this.baseUrl + 'character';

@Injectable()
export class RequestExampleService {

  constructor(private http: HttpClient) { }

  getCharacters() {
    return this.http.get(characterUrl);
  }
}
```

Y recordad lo importamos en nuestro app.module.ts en los providers:

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { RequestExampleComponent } from './request-example/request-example.component';
import { HttpClientModule } from '@angular/common/http';
```

```
import { RequestExampleService } from './request-example/services/request-example.service';

@NgModule({
  declarations: [
    AppComponent,
    RequestExampleComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [RequestExampleService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Fácil eh? Importamos el módulo **HttpClient** allí y solo buscamos usando **http.get** desde el módulo inyectado.

¿Qué nos devolverá esta petición? Un **observable**! Esta es una buena descripción de ellos:

> Observables provide support for passing messages between publishers and subscribers in your application. Observables offer significant benefits over other techniques for event handling, asynchronous programming, and handling multiple values.

Es prácticamente una forma súper genial y declarativa de codificar reactivamente. Pero no os preocupéis que en la próxima lección nos adentraremos dentro del mundo de los observables, por ahora nos centraremos en como recibir esa info.

Entonces, ¿cómo podemos obtener el resultado de esa búsqueda? Usando el método **.subscribe**, que permite indicar una función a ejecutar en caso de éxito en la llamada, y otra función en caso de error.

La respuesta en el caso de la petición realizada tiene este formato:

```
{
  "info": {},
  "results": [{
    "id": 1,
    "name": "Rick Sanchez",
    "image": "https://rickandmortyapi.com/api/character/avatar/1.jpeg"
  }]
}
```

Por lo tanto, podemos crear interfaces para la respuesta esperada y el objeto de carácter recibido. Y es lo primero que vamos a hacer antes de trabajar en nuestro componente. Siempre es bueno tipar la entrada y salida de datos para controlar el flujo de información de nuestra aplicación.

Por ello lo primero será crear una carpeta dentro de nuestro request-example con el nombre de models, y dentro de esta definir nuestra interfaz. Algo como:

```
export interface CharacterInterface {
  id: number;
  name: string;
  image: string;
}

export interface CharacterResponseInterface {
  info: {
    count: number;
    next: string;
    pages: number;
    prev: string;
  };
  results: CharacterInterface[];
}
```

Ahora cambiemos nuestro **request-example.component.ts**:

```
import { Component, OnInit } from '@angular/core';
import { CharacterInterface, CharacterResponseInterface } from './models/character.interface'
import { RequestExampleService } from './services/request-example.service';

@Component({
  selector: 'app-request-example',
  templateUrl: './request-example.component.html',
  styleUrls: ['./request-example.component.scss']
})
export class RequestExampleComponent implements OnInit {
  // declaramos la variable donde almacenamos nuestro resultado
  characterList: CharacterInterface[] = [];

  // Llamamos a nuestro servicio o inicializamos servicio
  constructor(private requestExampleService: RequestExampleService) {}

  // Al arrancar nuestra aplicación:
  ngOnInit() {
    // Utilizamos la función getCharacters para guardar nuestros resultados:
    this.requestExampleService.getCharacters()
    .subscribe((data: CharacterResponseInterface) => {
      const results: CharacterInterface[] = data.results;

      const formattedResults = results.map(({ id, name, image }) => ({
        id,
        name,
        image,
      }));
      this.characterList = formattedResults;
    });
  }
}
```

Vamos a explicar qué ha sucedido aquí:

1. Hemos importado el **RequestExampleService para** que podamos acceder a él más adelante desde nuestras funciones.
2. En el método **ngOnInit**, llamamos al método **requestExampleService.getCharacters()** y nos suscribimos al observable devuelto. Para ello le indicamos la función a ejecutar en caso de que la respuesta del servidor sea correcta, en ese caso lo que haremos es guardar el Array de resultados dentro de nuestra variable characterList.

Ahora cambiemos el archivo HTML para mostrar los caracteres:

```
<div *ngIf="characterList">
  <h3>Rick and Morty Character List!</h3>
  <div *ngFor="let character of characterList">
    <h4>{{ character.id }} - {{ character.name }}</h4>
    <img [src]="character.image" [alt]="character.name">
  </div>
</div>
```

Pintamos contenido solo cuando se define **characterList** , y luego iteramos a través de la lista para crear divs de caracteres con la id, el nombre y la imagen.

Obteniendo el siguiente resultado:

¡Esto esta encendido! 🔥😎

Podríamos seguir usando solicitudes POST o paginación, pero mantengamos eso en espera. Una vez aprendamos bien a usar el módulo de router completaremos una aplicación desde cero.