Desarrollo Web en Entorno Cliente

Angular 17.

Contenidos

1 Primera Aplicación con Angular 17.

Angular es un framework de desarrollo de aplicaciones web desarrollado por *Google*. Ofrece una arquitectura robusta y un conjunto completo de herramientas para construir aplicaciones web de una sola página (SPA).

Para comenzar a trabajar con **Angular**, es necesario realizar algunos pasos de **instalación** y **configuración** previos.

1.1 Herramientas necesarias.

Para trabajar con Angular 17, necesitamos instalar las siguientes herramientas:

1. **Node.js**: Nos proporciona el entorno de ejecución para ejecutar JavaScript fuera del navegador. Descargar desde https://nodejs.org/en/ la versión LTS (Long Term Support) e instalar.

2. **npm (*Node Package Manager*):** El gestor de paquetes de Node.js para instalar bibliotecas y herramientas. (instalar la última versión compatible con Angular

   Verificar instalación:
   ```
   node -v
   npm -v
   ```

   Actualizar npm:
   ```
   npm install -g npm
   ```

3. **Angular CLI**: (*Interfaz de línea de comandos de Angular*) En el desarrollo de aplicaciones con Angular, el uso de comandos es una práctica esencial. Estos comandos son proporcionados por Angular Cli, herramienta que automatiza tareas relacionadas con el ciclo de vida de una aplicación Angular.

   Instalar la CLI:
   ```
   npm install -g @angular/cli@17
   ```

   Ver las opciones:
   ```
   ng help
   ```

Algunos comandos básicos:

Crear un nuevo proyecto:
```
ng new nombre-proyecto
```

Iniciar el servidor de desarrollo:
```
ng serve
```
Generar componentes:
```
ng generate component mi-componente
```
ó

ng g c mi-componente

Generar servicios:

ng generate service nombre-servicio

Generar módulos:

ng generate module nombre-modulo

Generar pipes:

ng generate pipe nombre-pipe

Ver comandos de generación:

ng generate --help

Otros comandos:

Compilación para la producción:

ng build – configuration production

Ejecución de pruebas unitarias:

ng test

Agregar dependencias externas:

ng add ngx-bootstrap

Actualizar Angular CLI:

ng update @angular/cli

Ver versión:

ng versión

4. **Visual Studio Code**: Editor de código con extensiones útiles:

   - **Angular Language Service**: Ayuda con sugerencias de código y autocompletado.
   - **Prettier**: Para formatear el código.

1.2 Estructura de un proyecto.

Cuando generamos un proyecto Angular 17 con la CLI, la estructura que nos crea incluye:

Raíz del proyecto:

- **angular.json**: Configura cómo se compila y ejecuta la aplicación. Por ejemplo:
  - `projects.architect.build.options.styles`: Lista de estilos globales.
  - `projects.architect.build.options.scripts`: Lista de scripts globales.

- **package.json**: Información del proyecto y dependencias. Por ejemplo:
  - `start`: Arranca el servidor de desarrollo.
  - `build`: Construye la aplicación para producción.
  - `test`: Ejecuta pruebas unitarias.

- **tsconfig.json**: Configura el compilador TypeScript. Por ejemplo:
  - `strict`: Habilita comprobaciones estrictas de TypeScript.
  - `paths`: Define alias para rutas.

- **README.md**: Documentación básica.
- **node_modules/**: Dependencias instaladas.
- **src/**:
  - **app/**: Contiene los **componentes**, **módulos** y **servicios** de la aplicación.
  - **assets/**: Recursos estáticos como **imágenes** y **fuentes**.
  - **environments/**: Archivos para manejar configuraciones específicas de entorno:
    - `environment.ts`: Desarrollo.
    - `environment.prod.ts`: Producción.

1.3 Arranque de una aplicación

- Crear un nuevo proyecto: `ng new miapp`
- Navegar al directorio del proyecto: `cd miapp`
- Ejecutar el servidor de desarrollo: `ng serve`
- Abrir el navegador en: `http://localhost:4200` *para ver la aplicación*.

1.4 Componentes

Los **componentes** son los bloques básicos reutilizables que combinan lógica y vista, para la construcción de aplicaciones Angular modulares y mantenibles.

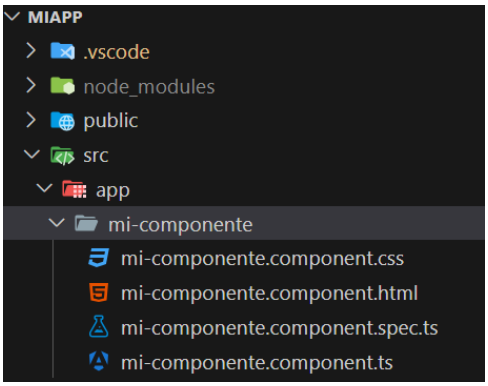
En cada componente está formado por:

- **HTML** (*Define la vista (UI)*).
- **CSS** (*Define los estilos*).
- **TypeScript** (*Maneja la lógica*).

Crear un componente:

`ng generate component mi-componente` o bien `ng g c mi-componente`

Tras ejecutar el comando, CLI generará automáticamente una nueva carpeta con el nombre del componente en el directorio **src/app** y los siguientes archivos:

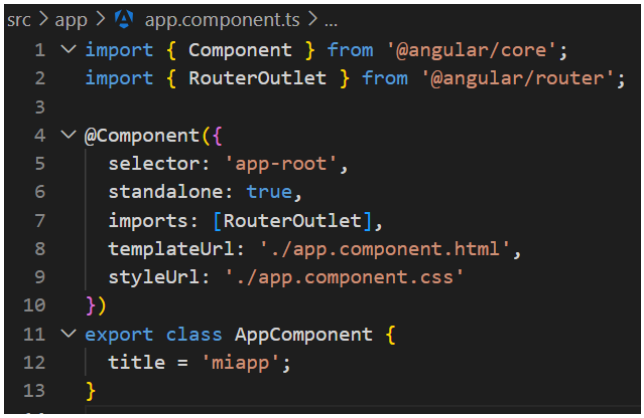


- `mi-componente.component.css`: El archivo CSS que contiene los estilos específicos del componente.
- `mi-componente.component.html`: La plantilla HTML que define la estructura visual del componente.
- `mi-componente.component.ts`: La clase TypeScript que controla la lógica del componente.
- `mi-componente.component.spec.ts`: El archivo de prueba para realizar pruebas unitarias.

Estructura de un componente:

Un **componente** en Angular se define utilizando la clase `@Component`. Esta clase es un decorador que acepta un objeto de metadatos como argumento.

El objeto de metadatos define las propiedades clave del componente, como el selector, la plantillas, los estilos.



Propiedades del decorador `@Component`:

- `Selector`**:** Define cómo se utilizará el componente en una plantilla HTML. Puede ser un nombre de elemento, un nombre de atributo, o una clase CSS.
- `Template/TemplateUrl`: Define la estructura HTML del componente. Puede ser una cadena de HTML directamente en el objeto de metadatos usando template, o referenciar un archivo HTML externo usando templateUrl.
- `Styles/StyleUrls`: Define los estilos CSS para el componente. Puede ser una cadena CSS directamente en el objeto de metadatos usando styles, o referenciar un archivo CSS externo usando styleUrls.
- `Standalone`: Indica que el componente es **independiente**, lo que significa que no necesita ser declarado en ningún módulo. Esta es una característica introducida en Angular 14 para simplificar la gestión de módulos.
- `Imports`: Permite importar otros componentes, directivas o servicios directamente en el componente. En este ejemplo, `imports: [RouterOutlet]` se utiliza para incluir el RouterOutlet, para aplicaciones que utilizan enrutamiento.

Ejemplo: Vamos a crear un componente llamado `info-usuario` que muestra información sobre un usuario:

Creamos el componente: `ng generate component info-usuario`

Como hemos visto al crear el nuevo componente se nos crea una carpeta con su nombre dentro de `app/src` y los siguientes archivos:

- `info-usuario.component.css`: El archivo CSS que contiene los estilos específicos del componente.
- `Info-usuario.component.html`: La plantilla HTML que define la estructura visual del componente.
- `Info-usuario.component.ts`: La clase TypeScript que controla la lógica del componente.
- `Info-usuario.component.spec.ts`: El archivo de prueba para realizar pruebas unitarias.

En el archivo TypeScript `info-usuario.component.ts`, añadimos a la clase `InfoUsuarioComponent` los datos de un usuario de ejemplo, también podríamos añadir métodos para manejar eventos o realizar operaciones relacionadas con este componente.

src > app > info-usuario > info-usuario.component.ts > ...

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

@Component({
  selector: 'app-info-usuario',
  standalone: true,
  imports: [],
  templateUrl: './info-usuario.component.html',
  styleUrl: './info-usuario.component.css'
})
export class InfoUsuarioComponent {
  usuario = {
    idUsuario: 2,
    nomUsuario: 'Ada',
    apeUsuario: 'Lovelace',
    emailUsuario: 'ada.lovelace@gmail.com'
  }
}
```

src > app > info-usuario > info-usuario.component.html > ...

```html
<div class="info-usuario">
  <h2>{{ usuario.idUsuario }}</h2>
  <p>{{ usuario.nomUsuario }} {{ usuario.apeUsuario }}</p>
  <p>{{ usuario.emailUsuario }}</p>
</div>
```

En la plantilla (archivo HTML) `info-usuario.component.html`, mostramos los datos de usuario mediante interpolación.

En el archivo `Info-usuario.component.css`, podemos incluir estilos propios al componente.

src > app > info-usuario > info-usuario.component.css

```css
.info-usuario{
  border: 7px solid #009a79;
  padding: 20px;
  margin: 5% 30%;
  background-color: #ccc;
  align-self: center;
  text-align: left;
}
```

src > app > app.component.html > ...

```html
<h1>{{ title }}</h1>
<app-info-usuario></app-info-usuario>
<router-outlet />
```

Una vez creado el componente, lo podemos **utilizar en otras partes de la aplicación** mediante su selector: `<app-info-usuario>`

Para ejecutar la aplicación, se utiliza el comando de Angular CLI: ng serve En un navegador web, navegar hacia http://localhost:4200 para observar el componente `info-usuario` en funcionamiento.

2 Introducción a TypeScript

TypeScript es un **superconjunto de JavaScript** que incorpora tipado estático y características avanzadas para facilitar el desarrollo de aplicaciones robustas y mantenibles.

2.1 Conceptos generales del lenguaje

TypeScript es un lenguaje de programación de código abierto desarrollado por Microsoft. Compila a JavaScript, lo que permite ejecutarse en cualquier entorno que soporte JavaScript (navegadores, Node.js, etc.).

Algunas características:

- Tipado estático.
- Soporte para clases.
- Interfaces y módulos.
- Detección temprana de errores gracias al tipado.
- Mejora en el autocompletado y refactorización.
- Compatibilidad con bibliotecas JavaScript existentes.
- Utiliza tsc (TypeScript Compiler) para compilar código .ts a .js. (tsc archivo.ts)

2.2 Clases, campos, interfaces, herencia

- **Clases:** TypeScript soporta **clases** con sintaxis similar a otros lenguajes orientados a objetos.

```
class Persona {
  nombre: string;
  edad: number;

  constructor(nombre: string, edad: number) {
    this.nombre = nombre;
    this.edad = edad;
  }

  saludar(): void {
    console.log(`Hola, soy ${this.nombre} y tengo ${this.edad} años.`);
  }
}

const persona = new Persona('María', 32);
persona.saludar();
```

- **Campos y modificadores de acceso**
  - **Públicos** (`public`): Accesibles desde cualquier lugar.

- **Privados** (`private`): Accesibles solo dentro de la clase.
- **Protegidos** (`protected`): Accesibles dentro de la clase y subclases.

```
class Coche {
  private marca: string;
  protected modelo: string;

  constructor(marca: string, modelo: string) {
    this.marca = marca;
    this.modelo = modelo;
  }

  getMarca(): string {
    return this.marca;
  }
}
```

- **Interfaces:** Las interfaces definen contratos para estructuras de datos y clases.

```
interface Animal {
  nombre: string;
  hacerSonido(): void;
}

class Perro implements Animal {
  nombre: string;
  constructor(nombre: string) {
    this.nombre = nombre;
  }

  hacerSonido(): void {
    console.log('Guau guau');
  }
}
```

- **Herencia:** Las clases pueden heredar de otras clases.

```
class Animal {
  constructor(public nombre: string) { }
  mover(): void {
    console.log(`${this.nombre} está en movimiento.`);
  }
}

class Perro extends Animal {
  ladrar(): void {
    console.log(`${this.nombre} está ladrando.`);
  }
```

```
}

const perro = new Perro('Max');
perro.mover();
perro.ladrar();
```

2.3 Tipos de datos y operadores especiales

- **Tipos de datos primitivos:**
  - `number`: Números (enteros y flotantes).
  - `string`: Cadenas de texto.
  - `boolean`: Verdadero o falso.
  - `null` y `undefined`: Valores especiales.
  - `any`: Desactiva el tipado estático.

- **Tipos de datos compuestos:**
  - `Arrays:`
    ```
    const numeros: number[] = [1, 2, 3];
    ```
  - `Tuplas:`
    ```
    let tupla: [string, number];
    tupla = ['Hola', 17];
    ```
  - `Enums:`
    ```
    enum Color {Rojo, Verde, Azul,}
    const miColor: Color = Color.Rojo;
    ```

- **Operadores especiales:**
  - `? (Opcional)`: Indica que un atributo o parámetro es opcional.

    ```
    function saludar(nombre?: string): void {
      console.log(`Hola, ${nombre || 'visitante'}`);
    }
    ```
  - `! (Aserción no nula):` Afirma que un valor no es `null` o `undefined`.

    ```
    let valor: string | undefined;
    console.log(valor!.toUpperCase());
    ```

2.4 Genéricos y conversión de tipos

- **Genéricos:** Permiten trabajar con tipos que se especifican al momento de la ejecución.

```
function identidad<T>(valor: T): T {
  return valor;
}
const numero = identidad<number>(42);
const texto = identidad<string>('Hola');
```

- **Conversión de tipos (Casting):**

```
let valor: any = 'Texto';
let longitud: number = (valor as string).length;
console.log(longitud);
```

2.5 Decoradores

Los **decoradores** permiten añadir metadatos o modificar el comportamiento de clases, métodos, propiedades o parámetros. Hay diferentes tipos de decoradores:

- **De clase**:

```
function Log(constructor: Function) {
  console.log('Clase creada:', constructor.name);
}

@Log
class Persona {
  constructor(public nombre: string) { }
}
```

- **De método**:

```
function Deprecated(target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
  console.warn(`El método ${propertyKey} está en desuso.`);
}

class Producto {
  @Deprecated
  precio(): void {
    console.log('Calculando precio...');
  }
}
```

- **De propiedad**:

```
function Propiedad(target: any, key: string) {
  console.log(`Propiedad decorada: ${key}`);
}

class Vehiculo {
  @Propiedad
  marca: string = 'Toyota';
}
```

3 Arquitectura de Angular

Angular utiliza una **arquitectura basada en componentes** y **servicios** que permite el desarrollo de aplicaciones dinámicas, modulares y reutilizables.

Componentes, **plantillas**, **directivas**, **pipes**, **servicios** y **módulos** (en versiones anteriores) trabajan juntos para crear un sistema estable y fácil de mantener.

3.1 Componentes y plantillas

Los **componentes** son las unidades básicas de la interfaz de usuario en Angular. **Encapsulan una lógica (TypeScript), una vista (HTML) y estilos (CSS o SCSS)**.

Las **plantillas** son archivos HTML que **definen la interfaz del componente**. Entre sus principales características nos encontramos:

- **Interpolación**: Mostrar valores dinámicos dentro de una plantilla.
- **Data Binding**: Sincronización de datos entre el modelo (TS) y la vista (HTML).

3.2 Interpolación y Data Binding

El **Data Binding** o **enlace de datos** es un mecanismo que **permite la sincronización** entre la **vista** (representación visual en el navegador) y el **modelo** (los datos TypeScript del componente) en una aplicación.

Angular ofrece diferentes formas de implementar el enlace de datos, unidireccional y bidireccional.

- **Interpolación:** Utiliza las llaves dobles `{{ }}` para mostrar valores dinámicos dentro de las plantillas, por ejemplo, el valor de una propiedad del componente directamente en el HTML.
- **Data Binding unidireccional (One-way binding):**
  - **Enlace de propiedad**: Utiliza corchetes `[ ]` para asignar el valor de una propiedad del componente a un atributo del DOM. **Propiedad → a la vista**:
  - **Enlace de evento:** Permite responder a eventos del usuario, como clics, pulsaciones de teclas, inputs, etc. Utiliza paréntesis `( )` para vincular estos eventos del DOM a un método del componente **. Vista → al modelo.**

- **Data Binding Bidireccional (Two-way binding):**
  - **Enlace bidireccional** Utiliza la combinación de corchetes y parantesis `[( )]` para vincular una propiedad del componente y un evento del DOM, permitiendo una comunicación bidireccional, para esto necesitamos importar `FormsModule`. Combina la entrada y salida de datos con `[(ngModel)]` (requiere `FormsModule`).

3.3 Directivas.

Las **directivas** son instrucciones que se aplican al DOM para modificar su estructura o comportamiento.

- **Directivas estructurales**: Alteran la estructura del DOM (añadir o eliminar elementos).

  Ejemplo: *ngIf, *ngFor.

```
<div *ngIf="mostrar">Se muestra si la condición es verdadera. </div>

<ul>
  <li *ngFor="let item of lista">{{ item }}</li>
</ul>
```

- **Directivas de atributo**: Cambian el aspecto o comportamiento de un elemento.

  Ejemplo: [ngClass], [ngStyle].

```
      <p [ngClass]="{'activo': esActivo}">Texto con clases dinámicas.</p>
```

- **Directivas personalizadas**: Se crean para añadir funcionalidades específicas.

  Ejemplo:

```
import { Directive, ElementRef, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appResaltar]'
})
export class ResaltarDirective {
  constructor(el: ElementRef, renderer: Renderer2) {
    renderer.setStyle(el.nativeElement, 'color', 'blue');
  }
}
```

3.4 Pipes

Los **pipes** transforman datos para mostrarlos en una plantilla. Angular incluye pipes integrados, pero también se pueden crear pipes personalizados.

Ejemplo: `<p>{{ fecha | date }}</p>`

- **Creación de pipes personalizados**:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'mayusculas'
})
export class MayusculasPipe implements PipeTransform {
  transform(valor: string): string {
    return valor.toUpperCase();
  }
}
```

Ejemplo de uso: `<p>{{ 'texto' | mayusculas }}</p>`

3.5 Servicios.

Los **servicios** proveen **lógica compartida** y manejan datos entre diferentes componentes.

- **Creación de un servicio**:

```
ng generate service nombre-servicio
```

Ejemplo:

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

@Injectable({
  providedIn: 'root'
})
export class DatosService {
  obtenerDatos() {
    return ['Dato 1', 'Dato 2', 'Dato 3'];
  }
}
```

- **Uso en un componente**:

```
import { Component } from '@angular/core';
import { DatosService } from './datos.service';

@Component({
  selector: 'app-lista',
  template: `<ul><li *ngFor="let dato of datos">{{ dato }}</li></ul>`
})
export class ListaComponent {
  datos: string[] = [];

  constructor(private datosService: DatosService) {
    this.datos = this.datosService.obtenerDatos();
  }
}
```

3.6 Módulos (Angular tradicional)

Los módulos son contenedores que agrupan componentes, directivas, pipes y servicios. El módulo raíz es AppModule, definido en app.module.ts.

Ejemplo:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent, // Declarar componentes, directivas y pipes aquí
  ],
  imports: [
    BrowserModule,
    FormsModule, // Importar otros módulos necesarios
  ],
  providers: [], // Servicios globales
  bootstrap: [AppComponent] // Componente principal
})
export class AppModule { }
```

4 Plantillas y Data Binding

Las **plantillas** (*archivos HTML que definen la interfaz del componente*) y el **Data Binding** (*enlace de datos entre la vista y el modelo*) nos permiten **construir la interfaz de usuario** y **vincular datos entre el modelo y la vista** de manera efectiva.

Las **directivas estructurales** (`*ngIf`, `*ngFor`, `*ngSwitch`) y las **variables de plantilla** permiten construir aplicaciones dinámicas, eficientes y fáciles de mantener.

Los diferentes tipos de **Data Binding** proporcionan flexibilidad para sincronizar datos de forma eficiente.

4.1 Directivas estructurales: `*ngIf`, `*ngFor`, `*ngSwitch`

Las **directivas estructurales** manipulan la estructura del DOM, añadiendo o eliminando elementos en función de ciertas condiciones o iteraciones. **Las directivas estructurales añaden o eliminan por completo secciones del DOM**. Se las reconoce porque utilizan el **asterisco (*)** antes de su nombre.

Suelen colocarse en elementos (como `div`, `li`, `p`, etc.) y manipulan la plantilla del HTML de acuerdo con la lógica definida.

4.1.1 *ngif

`*ngIf` se utiliza para mostrar u ocultar un elemento (*o sección completa de una plantilla*) en función de una expresión booleana.

Cuando la expresión que acompaña al `*ngIf` es `true`, el elemento se agrega al DOM. Si es `false`, el elemento se elimina del DOM (*no sólo se oculta con CSS, sino que no está presente en la estructura del documento*).

Sintaxis:

```
<div *ngIf="condicion">
      Se muestra sólo si la condición es verdadera
</div>
```

Donde:

- ✓ Si "`condición`" es una variable booleana en el componente, al cambiar su valor de `false` a `true`, Angular agregará dinámicamente el elemento y su contenido al DOM.
- ✓ Si "`condición`" cambia su valor de `true` a `false`, Angular removerá completamente el elemento del DOM, liberando recursos.

Con else:

```
<div *ngIf="condicion; else noMostrar">
      Se muestra si la condición es verdadera
</div>
<ng-template #noMostrar>
  <p>Se muestra si la condición es falsa.</p>
</ng-template>
```

Ejemplo: Al presionar el botón, el texto se mostrará u ocultará alterando directamente la estructura del DOM.

En typescript:

```
// mi-componente.component.ts
export class MiComponenteComponent {
  public mostrarMensaje: boolean = true;
}
```

En la plantilla (HTML):

```
<!-- mi-componente.component.html -->
<p *ngIf="mostrarMensaje">¡Este es un mensaje que se muestra
condicionalmente!</p>
<button (click)="mostrarMensaje = !mostrarMensaje">
  Alternar Mensaje
</button>
```

4.1.2 *ngFor

`*ngFor` se utiliza para iterar sobre colecciones (arrays) y así renderizar un bloque de código para cada elemento de la lista. Se usa para crear listados dinámicos, tablas, menús, etc.

Sintaxis:

```
<ul>
  <li *ngFor="let item of items">
    {{ item }}
  </li>
</ul>
```

Donde:

- ✓ Por cada elemento en el array `items`, Angular creará un `<li>` correspondiente con el contenido especificado.
- ✓ Si `items` cambia (por ejemplo, se agrega un elemento al array), Angular actualizará dinámicamente la lista en el DOM.

- ✓ `*ngFor` también soporta variables locales adicionales como `index` (el índice del elemento), `first`, `last`, `even`, `odd` para controlar la lógica interna.

```
<li *ngFor="let item of items; index as i; first as esPrimero">
      {{ i }} - {{ item }} - ¿Primero?: {{ esPrimero }}
</li>
```

Ejemplo:

En typescript:

```
// mi-componente.component.ts
export class MiComponenteComponent {
  public colores: string[] = ['Rojo', 'Azul', 'Verde', 'Blanco'];
}
```

En la plantilla (HTML):

```
<!-- mi-componente.component.html -->
<ul>
  <li *ngFor="let color of colores; let i = index">
    {{ i }} - {{ color }}
  </li>
</ul>
```

Este código generará cuatro elementos <li>:

- `0 - Rojo`
- `1 - Azul`
- `2 - Verde`
- `3 – Blanco`

Si posteriormente agregamos `this.colores.push('Negro');` en el componente, Angular añadirá automáticamente un nuevo `<li>` con `4 - Negro` sin necesidad de cambiar el HTML manualmente.

4.1.3 *ngSwitch

`*ngSwitch` se utiliza para mostrar una sección del DOM entre múltiples opciones mutuamente excluyentes, similar a la estructura switch en lenguajes de programación. Se emplean las directivas `*ngSwitchCase` para cada caso y `*ngSwitchDefault` en caso de que ninguno de los casos coincida.

Sintaxis:

```
<div [ngSwitch]="valor">
  <p *ngSwitchCase="'opcion1'">Opción 1 seleccionada</p>
  <p *ngSwitchCase="'opcion2'">Opción 2 seleccionada</p>
  <p *ngSwitchDefault>Opción por defecto</p>
</div>
```

- ✓ Angular evaluará la expresión en [ngSwitch] y mostrará solo el bloque *ngSwitchCase correspondiente al valor resultante.
- ✓ Si no hay una coincidencia, mostrará el bloque *ngSwitchDefault.
- ✓ Sólo un bloque será visible a la vez, los demás serán removidos del DOM.

Ejemplo:

En typescript:

```
// mi-componente.component.ts
export class MiComponenteComponent {
  public opcionSeleccionada: string = 'opcion1';
}
```

En la plantilla (HTML):

```
<!-- mi-componente.component.html -->
<div [ngSwitch]="opcionSeleccionada">
  <p *ngSwitchCase="'opc1'">Has seleccionado la Opción 1</p>
  <p *ngSwitchCase="'opc2'">Has seleccionado la Opción 2</p>
  <p *ngSwitchDefault>Opción no reconocida</p>
</div>

<button (click)="opcionSeleccionada = 'opc1'">Opción 1</button>
<button (click)="opcionSeleccionada = 'opc2'">Opción 2</button>
<button (click)="opcionSeleccionada = 'otra'">Opción Desconocida</button>
```

Además de las tres directivas estructurales vistas (*ngIf, *ngFor y *ngSwitch), Angular proporciona otras directivas que también pueden considerarse estructurales, ya sea porque alteran la forma en la que el DOM se presenta o insertan contenido dinámicamente. Entre ellas se encuentran:

4.1.4 NgTemplateOutlet

NgTemplateOutlet no suele usarse con el asterisco directamente, pero se considera estructural en el sentido de que permite inyectar vistas (plantillas) en el DOM dinámicamente. Se utiliza frecuentemente junto con **ng-template**.

Ejemplo:

```
<!-- En la plantilla: mi-componente.component.html -->
<ng-template #miPlantilla>
  <p>Contenido de la plantilla</p>
</ng-template>

<ng-container *ngTemplateOutlet="miPlantilla"></ng-container>
```

- ✓ El contenido definido dentro de `#miPlantilla` se inserta en el `ng-container` cuando se evalúa la directiva.

4.1.5 NgComponentOutlet

NgComponentOutlet permite cargar componentes de forma dinámica en una vista. Cumple una función similar a `*ngIf`, `*ngFor` y `*ngSwitch`, ya que modifica la estructura del DOM al insertar vistas de componentes.

Ejemplo:

```
<!-- En la plantilla: mi-componente.component.html -->
<ng-container *ngComponentOutlet="miComponenteDinamico"></ng-container>
```

- ✓ Donde `miComponenteDinamico` podría ser una referencia a un componente importado dinámicamente.

4.1.6 NgPlural y NgPluralCase (en i18n)

Dentro de las herramientas de internacionalización, **NgPlural** y **NgPluralCase** funcionan de manera similar a `*ngSwitch`, pero enfocadas a la pluralización de texto. Dependiendo del conteo, se renderizará una u otra vista.

Ejemplo:

```
<!-- En la plantilla: mi-componente.component.html -->
<ng-container [ngPlural]="cantidad">
  <ng-template ngPluralCase="=0">No tienes mensajes</ng-template>
  <ng-template ngPluralCase="=1">Tienes un mensaje</ng-template>
  <ng-template ngPluralCase="other">Tienes {{ cantidad }} mensajes</ng-
template>
</ng-container>
```

- ✓ Dependiendo del valor de `cantidad`, se mostrará una de las plantillas correspondientes.

¿Qué son i18n y l10n?

- **i18n** es la abreviatura utilizada para referirse a "internacionalización". En inglés "*internationalization*", que tiene 20 letras. Al tomar la primera letra "*i*", la última "*n*" y contar las 18 letras intermedias, se forma la **notación i18n**.
    - **Internacionalización** (**i18n**) es el proceso de diseñar y preparar una aplicación para que pueda adaptarse fácilmente a diferentes idiomas, formatos de fecha, monedas, reglas de pluralización y otras convenciones regionales y culturales sin requerir cambios significativos en el código.

- **Localización** (**l10n**), es el proceso de adaptar una aplicación ya "internacionalizada" a un idioma o región específicos, traduciendo textos, ajustando el formato de fechas, números, monedas y atendiendo a las particularidades culturales de cada región.
  - **l10n (localización)**: El acto de traducir y adaptar la aplicación a un idioma o cultura en particular.

4.1.7 Directivas estructurales personalizadas

Además de las directivas estructurales antes mencionadas, podemos crear nuestras propias directivas estructurales utilizando **@Directive()** y el **ViewContainerRef** junto con **TemplateRef** para manipular directamente la inserción y eliminación de vistas en el DOM. Son útiles si necesitamos patrones específicos que no se cubren con las directivas existentes.

4.2 Tipos de Data Binding

Angular permite la sincronización de datos entre el modelo y la vista utilizando diferentes tipos de Data Binding.

Interpolación (One-way Binding: Modelo → Vista):

Utiliza `{{ }}` para insertar valores en la plantilla.

En el modelo (TS):

```
nombre: string = 'Ada';
```

En la vista (HTML):

```
<p>Mi nombre es: {{ nombre }}</p>
```

Property Binding (One-way Binding: Modelo → Vista):

Vincula una propiedad del componente a un atributo del DOM.

En el modelo (TS):

```
desactivado: boolean = true;
```

En la vista (HTML):

```
<button [disabled]="desactivado">Pulsa</button>

<img [src]="imagenUrl" />
```

Event Binding (One-way Binding: Vista → Modelo):

Captura eventos del DOM y los vincula a métodos del componente.

En la vista (TS):

```
<button (click)="saludar()">Saludo</button>
```

En el modelo (TS);

```
saludar() {
  alert('Un saludo desde Angular');
}
```

Two-way Binding (Modelo ↔ Vista):

Vincula datos de forma bidireccional entre el modelo y la vista usando [(ngModel)].

Requiere FormsModule:

```
import { FormsModule } from '@angular/forms';
```

Ejemplo:

```
<input [(ngModel)]="nombre" />
<p>Tu nombre es: {{ nombre }}</p>
```

4.3 Variables de plantilla

Las **variables de plantilla** son identificadores declarados en las plantillas que permiten acceder a elementos del DOM o datos del componente. Se declara una variable con #.

Ejemplo:

```
<input #miInput />
<button (click)="mostrarValor(miInput.value)">Mostrar Valor</button>
```

Usos:

- **Acceso a propiedades del DOM:** Usar una variable de plantilla para manipular directamente un elemento del DOM:

```
<input #nombreInput />
<button (click)="nombreInput.focus()">Enfocar</button>
```

- **Acceso a directivas o componentes:** Si un elemento tiene una directiva asociada, se puede acceder a sus propiedades.

```
<app-hijo #hijo></app-hijo>
<button (click)="hijo.metodoHijo()">Llamar método del hijo</button>
```

5 Formularios en Angular.

Los **formularios** son una parte importante de cualquier aplicación web, para manejar la interacción del usuario con la interfaz.

Angular ofrece dos enfoques para trabajar con formularios:

- **formularios basados en plantilla** (Template-Driven Forms)
- **formularios reactivos** (Reactive Forms), ambos con capacidades avanzadas de validación.

5.1 Formularios de plantilla

Los **formularios de plantilla** (Template-Driven Forms) se caracterizan por:

- Basarse principalmente en directivas en el **HTML** (plantilla).
- Usar la directiva **ngModel** para enlazar (binding) los campos del formulario con las propiedades del componente TypeScript.
- Delegar a Angular la mayor parte de la gestión del estado del formulario, validaciones y control de errores a través de la plantilla.
- Ser adecuados para formularios simples o de tamaño moderado.

5.1.1 Configuración inicial: importar `FormsModule`

Para poder usar los formularios de plantilla, es **imprescindible** importar el `FormsModule` en el componente standalone donde está el formulario.

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css'],
})
export class LoginComponent {
  // ……
}
```

5.1.2 Data Binding con `[(ngModel)]`

Dentro del HTML del formulario, la directiva `ngModel` nos permite enlazar las propiedades del componente (por ejemplo, `email` y `password`) con los campos de entrada (`<input>`) del formulario.

- La sintaxis de doble enlace `[(ngModel)]="propiedad"` indica que el valor del input (vista) y la propiedad del componente (modelo) se sincronizan automáticamente.
- **El atributo** `name` en el input es **obligatorio** al usar `[(ngModel)]`; Angular necesita ese nombre para poder gestionar internamente el control de formulario.

Ejemplo en la plantilla:

```
<input
  type="email"
  id="email"
  class="form-control"
  [(ngModel)]="email"
  name="email"
  required
  #emailModel="ngModel"
/>
```

Donde:

- ✓ `[(ngModel)]="email"`: Enlaza el valor del input con la variable `email` del componente.
- ✓ `name="email"`: Hace que Angular asocie internamente este campo con el nombre "email".
- ✓ `required`: Directiva de validación simple; Indica que este campo es obligatorio.
- ✓ `#emailModel="ngModel"`: Esta **variable de referencia** local nos permite acceder a propiedades de validación y estado del control, por ejemplo, `emailModel.invalid`, `emailModel.touched`, etc. (útil para mostrar mensajes de error condicionales en el HTML).

5.1.3 Estructura del formulario en HTML

Para declarar el <form>:

```
<form (ngSubmit)="login()">
  <!—Campos del formulario -->
    <button type="submit" class="btn btn-primary btn-lg">Enviar</button>
</form>
```

Donde:

- ✓ Utilizamos `(ngSubmit)="login()"` para **escuchar** el evento **submit** del formulario. Cuando el usuario hace clic en "Enviar" (o presiona Enter), se llama automáticamente al método `login()` del componente.
- ✓ Con `#loginForm="ngForm"` podríamos acceder a las propiedades del formulario en la plantilla, permitiéndonos usar en el HTML expresiones como `loginForm.valid` o `loginForm.invalid`.

```
<form #loginForm="ngForm" (ngSubmit)="login()">
</form>
```

5.1.4 Validaciones sencillas con directivas

En los formularios de plantilla, Angular ofrece directivas **básicas** de validación, como:

- `required`
- `minlength`
- `maxlength`
- `pattern`

Por ejemplo, en el campo de email usamos `required`, lo que obligará a que el campo no quede vacío. En conjunto, Angular controla el estado (`valid`, `invalid`, `touched`, `pristine`, etc.) automáticamente, y podríamos mostrar mensajes de error usando `*ngIf="emailModel.invalid && emailModel.touched"` o similar.

También podemos añadir en TypeScript, validaciones personalizadas, por ejemplo:

```
if (!this.validarEmail(this.email)) {
      this.error = 'Formato de email incorrecto';
      return;
    }
    if (!this.validarPassword(this.password)) {
      this.error =
        'La contraseña debe tener más de 6 caracteres y al menos un
número y una letra';
      return;
    }
```

5.1.5 Manejo del submit con `(ngSubmit)`

El método `login()` en el componente se llama cuando se dispara el evento `(ngSubmit)` del formulario:

```
login(): void {
    // 1. Validaciones de email y password
    // 2. Llamar al servicio sociosService.getSocioLogin(...)
    // 3. Manejo de la respuesta con éxito o error
}
```

```
  validarEmail(email: string): boolean {
    const re = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return re.test(email);
  }

  validarPassword(password: string): boolean {
    const re = /^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{6,}$/;
    return re.test(password);
  }
```

El ejemplo seguido componente `login` de la app `angcoop25`:

1. Valida que el formato de `email` sea correcto (por `regex`).
2. Valida la `contraseña` usando otra `regex` que exige un mínimo de 6 caracteres, combinando letras y números.
3. Si pasa las validaciones, construye un `FormData` con `opcion`, `email` y `password`.
4. Llama al servicio `sociosService` para verificar credenciales.
5. Si la respuesta tiene éxito, se inicia sesión en el `sessionService` y se redirige a '`/galeria`'.
6. Si hay error (usuario no registrado o problema de conexión), se maneja estableciendo `this.error = ...` y usando el `alertService` para mostrar el error.

5.1.6 Visualización de errores en la plantilla

Utilizamos:

```
  <div *ngIf="error" class="alert alert-danger mt-3">
    {{ error }}
  </div>
```

Para mostrar un mensaje de error (error) proveniente del componente. Este mensaje se actualiza cuando:

- No pasan las validaciones de `email` o `password`.
- El servicio `socioService` retorna un error.
- O hay error de conexión.

Cuando `this.error` tiene contenido, se renderiza dinámicamente este `<div>` con la clase `alert alert-danger`.

5.1.7 El formulario login

1. **Carga de la vista**: El formulario se muestra con dos inputs: `email` y `password`, ambos enlazados a las variables del componente.
2. **El usuario introduce datos**: Cada vez que el usuario teclea, `[(ngModel)]` sincroniza los valores con `this.email` y `this.password`.
3. **El usuario hace Submit** (por clic en botón `type="submit"` o presionando Enter):
   - Se llama a la función `login()`.
   - Dentro de `login()`, se ejecutan las validaciones personalizadas en TypeScript.
   - Si las validaciones pasan, se hace la llamada al servicio (`socioService.getSocioLogin`).
4. **Se procesa la respuesta**:
   - Éxito: Se inicia la sesión (`sessionService.iniciarSesion`), se muestra el `SweetAlert` y se redirige a '`/galeria`'.
   - Error: Se asigna un mensaje a `this.error`, se llama al `alertService` y (en algunos casos) se redirige a '`/registro`'.

Con el ejemplo de `login` del proyecto `angcoop25,` formulario con dos campos enlazados a variables, validaciones básicas en la plantilla + validaciones más detalladas en el componente TypeScript, y luego la llamada a un servicio para procesar el login en el backend.

Hemos visto cómo **Angular** maneja los **formularios de plantilla**, cómo se realiza la **validación** y cómo se efectúa el **data binding** entre la vista y la lógica en TypeScript.

El **modelo de formularios de plantilla** se centra en:

1. **Importar** `FormsModule`.
2. **Crear** propiedades en el componente para cada dato que maneje el formulario (`email`, `password`, etc.).
3. **Usar** directivas como `[(ngModel)]`, `name`, y validaciones como `required`, `pattern` en la plantilla HTML.
4. **Escuchar** el evento `(ngSubmit)` para disparar la lógica de envío (método `login()`).

5.2 Formularios Reactivos

Los **formularios reactivos** se caracterizan por:

- **Definir** y **administrar** la estructura del formulario (campos, validaciones, etc.) principalmente desde el **TypeScript** (en el componente).
- Utilizar **clases y APIs** como `FormGroup`, `FormControl`, `FormBuilder` y `Validators`.
- Proporcionar un mayor **control** y **flexibilidad** para validaciones avanzadas, flujos de datos complejos y pruebas unitarias.

En este modelo, cada campo del formulario se convierte en un control que se define con validadores y que se agrupa dentro de un `FormGroup` que representa al formulario completo.

5.2.1 Configuración inicial: `ReactiveFormsModule`

Para utilizar formularios reactivos, debemos:

1. **Importar** el módulo `ReactiveFormsModule` en el componente:

```
import { ReactiveFormsModule } from '@angular/forms';
```

2. En componentes **standalone**, en la declaración del decorador `@Component` se agregan los imports:

```
@Component({
  selector: 'app-registro-socio',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  . . .
})
export class RegistroSocioComponent implements OnInit {. . .}
```

5.2.2 Estructura de un formulario reactivo

En el ejemplo registrar un socio de la app angcoop25, el componente `RegistroSocioComponent` define y maneja un formulario para registrar un nuevo socio:

```
export class RegistroSocioComponent implements OnInit {
  // Declaramos un FormGroup
  registroForm: FormGroup;

  constructor(
    private fb: FormBuilder,    // Inyectamos el FormBuilder
    private sociosService: SociosService,
    private router: Router,
    private sanitizer: DomSanitizer,
    private alertService: AlertService
```

```
) {
  // Se crea e inicializa el FormGroup usando FormBuilder
  this.registroForm = this.fb.group({
    nombre: ['', [Validators.required,
                  Validators.pattern(/^[A-Za-zÁÉÍÓÚáéíóúñÑ0-9 ]{2,20}$/), ]],
    apellidos: ['', [Validators.required,
                  Validators.pattern(/^[A-Za-zÁÉÍÓÚáéíóúñÑ0-9 ]{2,30}$/), ]],
    email: ['', [Validators.required, Validators.email]],
    foto: [null], // valor inicial sin validación
    password: ['', [Validators.required,
                  Validators.pattern(/^(?=.*[A-Za-z])(?=.*\d).{5,10}$/),]],
    repitePassword: ['', [Validators.required,
                  Validators.pattern(/^(?=.*[A-Za-z])(?=.*\d).{5,10}$/),]],
  });
}

ngOnInit(): void {}

// . . .
}
```

5.2.3 FormBuilder, FormGroup, FormControl y Validators

- **`FormBuilder`**: Servicio que simplifica la creación de formularios, en vez de instanciar manualmente `FormControl` y `FormGroup`.
- **`FormGroup`**: Agrupa un conjunto de controles (campos) y sus validaciones. Cada `key` en el objeto corresponde a un `FormControl`.
- **`FormControl`**: Representa un campo único del formulario. Se asigna un valor inicial y un conjunto de validadores (opcional).
- **`Validators`**: Conjunto de validadores provistos por Angular (por ejemplo, `required`, `email`, `pattern`, `minLength`, etc.). También podemos crearnos **validadores personalizados**.

En el ejemplo:

```
  this.registroForm = this.fb.group({
    nombre: [
          '',    // valor inicial
          [      // validadores
                 Validators.required,
                 Validators.pattern(/^[A-Za-zÁÉÍÓÚáéíóúñÑ0-9 ]{2,20}$/),
          ]
     ],
     // resto de campos
```

Cada campo (`nombre`, `apellidos`, etc.) está definido como un **array**:

`[nombreInicial, [listaDeValidadores]]`

5.2.4 Vinculación con la Plantilla: `[formGroup]` y `formControlName`

En el HTML, se **asocia** el `FormGroup` del componente con el `<form>` mediante `[formGroup]="registroForm"` y cada input se enlaza a su correspondiente control con `formControlName="nombreDelCampo"`:

```
<form
    [formGroup]="registroForm"
    (ngSubmit)="registrarSocio()"
    enctype="multipart/form-data"
  >
    <!-- campo 'nombre' -->
    <div class="mb-3">
      <label for="nombre" class="form-label">Nombre</label>
      <input
        type="text"
        id="nombre"
        class="form-control"
        formControlName="nombre"   <!-- vincula con el FormControl 'nombre'-->

      />
          <!--   Mensajes de error   -->
      <div *ngIf="mostrarError('nombre', 'required')" class="text-danger">
        El nombre es obligatorio.
      </div>
      <div *ngIf="mostrarError('nombre', 'pattern')" class="text-danger">
        El nombre debe tener entre 2 y 20 caracteres . . .
      </div>
    </div>

    <!—resto de campos -->

</form>
```

`formControlName`

- `formControlName="nombre"` indica que este `<input>` se enlaza al control definido en `this.registroForm` con la clave **nombre**.
- En tiempo de ejecución, Angular sincroniza **el valor** del input con el `FormControl` (y sus validaciones).

`[formGroup]="registroForm"`

- El `<form>` sabe que debe manejar su conjunto de inputs como un solo **grupo** de controles, representado por `registroForm` en TypeScript.
- `(ngSubmit)` escucha el envío del formulario y ejecuta `registrarSocio()` (o el método que definamos).

Los componentes (`FormControl`, `FormGroup`, `FormArray`) tiene propiedades y métodos similares para manejar sus estados y valores.

Propiedades:

- **value:** El valor actual del control o grupo.
- **status:** El estado de validación actual, `VALID`, `INVALID`, `PENDING` o `DISABLED`.
- **errors:** Los errores actuales si el control es inválido.
- **touched:** Indicador de si el control ha sido tocado por el usuario.
- **untouched:** Indicador de si el control no ha sido tocado por el usuario.
- **pristine:** Indicador de si el control no ha sido modificado.
- **dirty:** Indicador de si el control ha sido modificado.

Métodos:

- **setValue:** Establece un nuevo valor para el control, grupo o array.
- **patchValue:** Similar a setValue, pero permite establecer valores de forma parcial.
- **reset:** Reinicia el control al estado inicial.
- **disable:** Desactiva el control.
- **enable:** Activa el control.

5.3 Validaciones en formularios reactivos

En Reactive Forms, las validaciones se definen en el **TypeScript** (al crear los `FormControl`). Angular se encarga de verificar dichas reglas automáticamente. En el ejemplo:

```
    nombre: [
          '',    // valor inicial
          [      // validadores
                 Validators.required,
                 Validators.pattern(/^[A-Za-zÁÉÍÓÚáéíóúñÑ0-9 ]{2,20}$/),
          ]
     ],
```

- `Validators.required` exige que el campo no esté vacío.
- `Validators.pattern(...)` aplica una expresión regular para restringir los caracteres permitidos y la longitud.

También se pueden combinar validadores como `Validators.minLength`, `Validators.maxLength`, `Validators.email`, etc. Si se requiere lógica personalizada, se pueden crear **validadores propios**.

5.4 Mostrar mensajes de error en la plantilla

Para mostrar u ocultar mensajes de error en HTML, utilizaremos:

```
    <div *ngIf="mostrarError('nombre', 'required')" class="text-danger">
      El nombre es obligatorio.
    </div>
```

El método `mostrarError` en el componente evalúa si un control específico (`nombre`) tiene un error concreto (`required`) y, además, si el control fue "tocado" (`touched`):

```
mostrarError(controlName: string, errorName: string): boolean {
  const control = this.registroForm.get(controlName);
  return control!.hasError(errorName) && control!.touched;
}
```

Esto evita mostrar los mensajes de error hasta que el usuario haya interactuado con el campo.

5.5 Lógica de envío (Submit)

Cuando el usuario hace clic en el botón `type="submit"`, se activa `(ngSubmit)="registrarSocio()"`.

Dentro de `registrarSocio()`:

1. Se verifica si `this.registroForm.invalid` (el formulario es inválido).
2. Si es inválido, se marcan todos los campos como tocados con: `control?.markAsTouched()` para que se muestren los mensajes de error.
3. Se comprueba que las contraseñas coincidan.
4. Se construye un `FormData`.
5. Se manda la información al servicio (`sociosService.registrarSocio(formData)`) que hace la llamada HTTP al backend.
6. Se maneja la respuesta (caso de éxito o error) y se realizan acciones (mostrar alertas, redirigir).

```
registrarSocio(): void {
    if (this.registroForm.invalid) {
      // Marca los campos como tocados
      Object.keys(this.registroForm.controls).forEach((field) => {
        const control = this.registroForm.get(field);
        control?.markAsTouched();
      });
      this.alertService.error(
        'Error',
        'Por favor, completa todos los campos correctamente.'
      );
      return;
    }

    // Verificar que las contraseñas coincidan
    const password = this.registroForm.value.password;
    const repitePassword = this.registroForm.value.repitePassword;
    if (password !== repitePassword) {
      this.alertService.error('Error', 'Las contraseñas no coinciden.');
      return;
    }

    // Crear FormData y enviar la petición
     ...
}
```

5.6 Subida de archivos y previsualización

El formulario de registro, permite la subida por parte del usuario de su foto de perfil, al cambiar el input `type="file"`, se llama a `previsualizarImagen($event)`:

```
previsualizarImagen(event: Event): void {
   const input = event.target as HTMLInputElement;
   if (input.files && input.files[0]) {
     const file = input.files[0];
     // Actualizar el campo foto en el formulario
     this.registroForm.patchValue({ foto: file });

     // Crea un objeto URL temporal para mostralo el <img>
     const objectUrl = URL.createObjectURL(file);
     this.imagenPreview = this.sanitizer.bypassSecurityTrustUrl(objectUrl);
   }
 }
```

Donde:

- ✓ Se lee el archivo seleccionado por el usuario (un `File`).
- ✓ Usa `patchValue` para asignar ese archivo al campo `foto` del `FormGroup`.
- ✓ Genera una URL temporal con `URL.createObjectURL(file)` para previsualizarlo.
- ✓ Se **sanitiza** la URL con `DomSanitizer` para evitar advertencias de seguridad en Angular.

En el HTML, el `<img [src]="imagenPreview" …>` se actualiza dinámicamente con la imagen seleccionada.

5.7 Diferencias con los formularios de plantilla

- **Ubicación de la lógica**: En formularios reactivos, la mayor parte de la definición y validación se hace en **TypeScript**; en formularios de plantilla, se hace en el **HTML** (con directivas como `ngModel`, `required`, etc.).
- **Mayor control y testabilidad**: Los formularios reactivos son más fáciles de probar y de escalar, especialmente con validaciones o flujos de datos complejos.
- **Estructura declarativa en TS**: El `FormGroup` se comporta como una "fuente única de la verdad" para el estado del formulario (valores, validez, etc.). En cambio, en Template-Driven Forms, la lógica del formulario se dispersa entre la plantilla y el componente.

7 Servicios en Angular

7.1 ¿Qué es un servicio?

Un **servicio** en Angular es una clase que encapsula lógica de negocio o lógica específica de la aplicación (cálculos, llamadas HTTP, almacenamiento de datos, etc.) y está pensada para ser **reutilizada** y **compartida** entre componentes. Los servicios suelen encargarse de:

- Proveer datos a los componentes (por ejemplo, a través de peticiones HTTP).
- Realizar operaciones de negocio o procesar datos de manera centralizada.
- Comunicarse con APIs externas (servidores, servicios en la nube, etc.).
- Mantener un estado común que puede ser compartido entre varios componentes de la aplicación.

Los servicios permiten mantener el código organizado, favoreciendo la separación de responsabilidades: el **componente se encarga de la interacción con la vista**, mientras que el **servicio gestiona la lógica y la obtención de datos**. Esto hace que la aplicación sea más fácil de mantener, probar, reutilizar y escalar.

7.2 Creación de un servicio en Angular.

Para crear un servicio en Angular usaremos el CLI con el comando:

```
ng generate service nombre-del-servicio
```

Por ejemplo:

```
ng generate service services/articulo
```

Esto generará dos archivos en la carpeta services:

- `articulo.service.ts` (la clase del servicio).
- `articulo.service.spec.ts` (para pruebas unitarias).

El archivo: `articulo.service.ts`

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

@Injectable({
  providedIn: 'root',
})
export class ArticuloService {
  constructor() {}
```

```
  // Método para obtener datos de los artículos
  obtenerArticulos() {
    // lógica para retornar o gestionar datos de los artículos
  }
}
```

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

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

  constructor() { }
}
```

Donde:
`@Injectable({  providedIn: 'root'})`

- Indica a Angular que este servicio se registre a nivel **root**.
- Es decir, Angular creará una sola instancia de este servicio para toda la aplicación (Singleton).

7.3 Inyección de dependencias

Cuando necesitamos utilizar un servicio dentro de un componente (o dentro de otro servicio), lo podemos inyectar a través del **constructor**:

Por ejemplo, si creamos el componente `list-articulos` en la capeta `components`:

```
ng generate component components/list-articulos
```

El archivo: `list-articulos.component.ts`

```
import { Component, OnInit } from '@angular/core';
// Importamos el sercivio
import { ArticuloService } from '../../services/articulo.service';

@Component({
  selector: 'app-list-articulos',
  standalone: true,
  imports: [],
  templateUrl: './list-articulos.component.html',
```

```
  styleUrl: './list-articulos.component.css',
})
export class ListArticulosComponent implements OnInit {

  constructor(private articuloService: ArticuloService) { }

  ngOnInit(): void {
    // Podemos acceder a los métodos del servicio
    const articulos = this.articuloService.obtenerArticulos();
  }
}
```

Aquí, `ListArticulosComponent` no crea el servicio por sí mismo, sino que declara una dependencia (`private articuloService: ArticuloService`) en su constructor. Angular utiliza el sistema DI (Inyección de Dependencias) para proveer `ArticuloService`.

7.4 Uso de HttpClient en servicios

Normalmente, los servicios se emplean para **comunicar** la aplicación con un servidor mediante HTTP. Angular ofrece el **HttpClient** para realizar peticiones de forma sencilla.

Para trabajar con `HttpClient` debemos:

1. **Importar** `provideHttpClient()`. Angular provee el helper `provideHttpClient()` para registrar los servicios asociados a `HttpClient`. Habilita el uso de HttpClient para hacer solicitudes HTTP (GET, POST, etc.) hacia APIs o servicios externos.

   Usar `provideHttpClient()` **en el array de providers** de **ApplicationConfig** (**en** `app.config.ts`),lo que:

   - **Registra los servicios y proveedores internos de HTTP**: Esto incluye las dependencias internas que el HttpClient requiere (como HttpHandler, interceptores por defecto, etc.).

   - **Habilitar la inyección del HttpClient**: Una vez configurado el injector global con `provideHttpClient()`, cualquier componente o servicio que solicite `HttpClient` en su constructor lo recibirá sin necesidad de una configuración adicional.

En el archivo `app.config.ts`:

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';
```

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

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes),
    // usar provideHttpClient en el array de providers
    provideHttpClient()
  ],
};
```

¿Qué es un "helper" en Angular? Un helper en Angular es una función utilitaria que encapsula la configuración de proveedores y dependencias.

- `provideHttpClient()` registra todos los providers necesarios para HttpClient.
- `provideRouter(routes)` hace algo similar para configurar el enrutador.

Los helpers simplifican la configuración inicial de la aplicación, evitando tener que declarar manualmente cada dependencia o importar módulos enteros.

2. **Inyectar HttpClient en el servicio** y usarlo en métodos para llamar a APIs:

En el archivo: `articulo.service.ts`

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

@Injectable({
  providedIn: 'root',
})
export class ArticuloService {
  private apiUrl = 'http://localhost/pruebaServidor/php/prueba.php'; // la URL

  constructor(private http: HttpClient) {}

  // Método para obtener datos de los artículos
  obtenerArticulos(): Observable<any> {
    return this.http.get(`${this.apiUrl}?opcion=AV`);
  }
}
```

3. Consumir el servicio desde un componente:

En el archivo: `list-articulos.component.ts`

```
import { Component, OnInit } from '@angular/core';
// Importamos el sercivio
import { ArticuloService } from '../../services/articulo.service';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-list-articulos',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './list-articulos.component.html',
  styleUrl: './list-articulos.component.css',
})
export class ListArticulosComponent implements OnInit {
  articulos: any[] = [];
  constructor(private articuloService: ArticuloService) {}

  ngOnInit(): void {
    this.articuloService.obtenerArticulos().subscribe({
      next: (data) => {
        console.log('Respuesta de servidor ---> ', data);
        this.articulos = data;
      },
      error: (error) =>
        console.error('Error al obtener los artículos ---> ', error),
    });
  }
}
```

En el ejemplo, `obtenerArticulos()` devuelve un **Observable** al cual nos subscribimos para leer la respuesta de la petición.

Nota : Una de las ventajas del `HttpClient` es su soporte para el tipado estricto. En lugar de **any**, podemos definir **interfaces o modelos** con la estructura de los datos recibidos así el editor y el compilador TypeScript podrán detectar errores y ofrecer autocompletado, mejorando el desarrollo del código.

7.5 Observables y RxJS

RxJS (Reactive Extensions for JavaScript) es una biblioteca que implementa el patrón Observer, utilizada en Angular para manejar flujos de datos asíncronos.

¿Qué son los Observables? Los **Observables** son una manera de manejar flujos asíncronos de datos que pueden emitir valores en diferentes momentos en el tiempo. En Angular, los Observables son la base para muchas funcionalidades reactivas, como:

- Peticiones HTTP (a través de `HttpClient`).
- Eventos de usuario (por ejemplo, desde librerías como Reactive Forms).
- Control de estados y notificaciones entre componentes.

La **principal diferencia** entre un Observable y otras formas de manejar asincronía (como Promises) **es que un Observable puede emitir múltiples valores en el tiempo**, mientras que una Promesa normalmente resuelve un único valor.

Subscribe y Unsubscribe

Para **consumir** un Observable, necesitamos **subscribirnos**:

```
miObservable.subscribe({
  next: (valor) => console.log('Nuevo valor: ', valor),
  error: (err) => console.error('Error: ', err),
  complete: () => console.log('Finalizado'),
});
```

- **next**: Se llama cada vez que el observable emite un valor.
- **error**: Se llama si ocurre algún error.
- **complete**: Se llama cuando el observable ya no emitirá más valores.

Podemos **cancelar** la suscripción cuando el componente se destruye, para evitar fugas de memoria y comportamientos inesperados:

```
export class MiComponente implements OnInit, OnDestroy {
  private miSuscripcion: Subscription;

  ngOnInit() {
    this.miSuscripcion = miObservable.subscribe(/* ... */);
  }

  ngOnDestroy() {
    if (this.miSuscripcion) {
```

```
      this.miSuscripcion.unsubscribe();
    }
  }
}
```

Operadores de RxJS más comunes

RxJS dispone de un conjunto de **operadores** para transformar, combinar y filtrar Observables. Algunos de los más utilizados son:

- **map**: Transforma cada valor emitido por el observable a otro valor.

```
import { map } from 'rxjs/operators';

this.http
  .get('api/usuarios')
  .pipe(map((usuarios) => usuarios.filter((u) => u.activo)))
  .subscribe((usuariosActivos) => {
    console.log(usuariosActivos);
  });
```

- **filter**: Filtra emisiones que no cumplen cierta condición.

```
import { filter } from 'rxjs/operators';

this.miObservable
  .pipe(
    filter((valor) => valor % 2 === 0) // solo valores pares
  )
  .subscribe((valor) => console.log(valor));
```

- **mergeMap / switchMap / concatMap**: Operadores para manejar Observables que retornan otros Observables (manejo de peticiones encadenadas, por ejemplo).
  - **switchMap** cancela la suscripción anterior al recibir una nueva emisión.
  - **mergeMap** ejecuta todos los Observables en paralelo (sin cancelar los anteriores).
  - **concatMap** encola las peticiones, esperando a que termine una antes de comenzar la siguiente.
- **tap**: Permite ejecutar efectos secundarios sin transformar el valor.
- **take** / **takeUntil**: Controla cuántas emisiones se reciben o hasta cuándo subscribirse.

- **Subject/BehaviorSubject**: Si necesitamos **emitir** valores manualmente y permitir que otros se suscriban, podemos usar un `Subject`.
- **BehaviorSubject** retiene el último valor emitido y lo reenvía a nuevos suscriptores.

7.6 Algunas características de HttpClient

- `HttpClient` ofrece varios **métodos** para cubrir diferentes necesidades HTTP:
  - `http.get<T>(url, options?)`: Recuperar datos.
  - `http.post<T>(url, body, options?)`: Enviar nuevos datos.
  - `http.put<T>(url, body, options?)`: Actualizar datos existentes.
  - `http.delete<T>(url, options?)`: Eliminar datos.
  - `http.patch<T>(url, body, options?)`: Actualizar parcialmente un recurso.
  - `http.head(url, options?)`: Obtener los encabezados de un recurso.

- Todos estos métodos retornan por defecto `Observable<T>`, lo que facilita la suscripción y la combinación con otras secuencias reactivas.
- Podemos adjuntar **opciones adicionales** a las peticiones, como **encabezados**, **parámetros de consulta** o **manejo de la respuesta**. Por ejemplo, para enviar un token de autenticación o indicar el tipo de contenido:
- El **manejo de errores** se realiza usando la infraestructura de Observable. Al suscribirse a una petición, se puede definir un callback de error: usando el operador `catchError`, `throwError` de RxJS para capturar errores.
- Los interceptores (`HttpInterceptor`) son una característica clave del `HttpClient`. Permiten interceptar todas las peticiones salientes y respuestas entrantes para modificarlas, agregar tokens de autenticación u otro tipo de lógica.

- **HttpClient** por defecto asume respuestas **JSON** e intenta parsearlas automáticamente. Simplificando el trabajo, ya que no necesitamos convertir la respuesta. No obstante, si se necesitan otros formatos, se pueden ajustar las opciones de la petición para recibir datos en texto plano u otros formatos.

Direcciones de interés:

https://angular.dev/api

https://rxjs.dev/