UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN III
FRONT END

Unidad Temática N° 4:
Angular

Teórico

2° Año – 3° Cuatrimestre

Desarrollo web actual

Introducción al Desarrollo Front-End Moderno y las Single Page Applications (SPA)

En la última década, el desarrollo web ha evolucionado significativamente, impulsado por la creciente demanda de aplicaciones más interactivas, rápidas y con una experiencia de usuario fluida. En el pasado, la mayoría de las aplicaciones web seguían un modelo de servidor tradicional, donde cada interacción del usuario requería una nueva carga de página completa desde el servidor. Este enfoque, aunque funcional, tenía limitaciones notables en términos de rendimiento y experiencia de usuario.

Hoy en día, el desarrollo front-end se centra en ofrecer aplicaciones ricas e interactivas que se comportan de manera más parecida a las aplicaciones de escritorio, brindando una experiencia más dinámica y reactiva. Este cambio ha sido posible gracias al surgimiento de las **Single Page Applications (SPA)**.

¿Por qué SPA?

Las **SPA** son aplicaciones web que cargan una única página HTML y actualizan dinámicamente su contenido a medida que el usuario interactúa con la aplicación. A diferencia de las aplicaciones tradicionales, las SPA no requieren recargar toda la página para mostrar nuevo contenido, lo que resulta en una experiencia más rápida y fluida.

Este enfoque responde a varias necesidades clave del desarrollo moderno:

- **Rendimiento y Velocidad**: Al reducir las solicitudes al servidor y minimizar la recarga de páginas completas, las SPA ofrecen tiempos de respuesta más rápidos, mejorando la percepción del rendimiento por parte del usuario.
- **Experiencia de Usuario**: Las SPA permiten una navegación más fluida y continua, similar a la de una aplicación de escritorio. Esto mejora significativamente la experiencia del usuario, haciéndola más intuitiva y atractiva.
- **Modularidad y Escalabilidad**: Las SPA promueven una arquitectura más modular, donde diferentes partes de la aplicación pueden ser desarrolladas, mantenidas y actualizadas de manera independiente.

En este contexto, Angular se destaca como uno de los frameworks más potentes y populares para desarrollar SPA, proporcionando un conjunto completo de herramientas y prácticas que permiten a los desarrolladores construir aplicaciones escalables, mantenibles y de alto rendimiento. A lo largo de este curso, explicaremos cómo Angular aborda estos desafíos y facilita la creación de aplicaciones modernas y robustas.

Angular

Angular es un framework de desarrollo de aplicaciones web de código abierto, mantenido por Google y la comunidad de Angular. Diseñado para facilitar la construcción de aplicaciones web dinámicas y de una sola página (SPA), Angular se destaca por su enfoque integral y su capacidad para manejar aplicaciones complejas de manera eficiente. A continuación, explicaremos qué es Angular, sus principales características y los beneficios que ofrece a los desarrolladores.

Historia y Evolución

Angular comenzó como una librería llamada "AngularJS" (también conocida como Angular 1) en 2010. Su objetivo era proporcionar un marco más estructurado para el desarrollo de aplicaciones web en comparación con las técnicas tradicionales basadas en JavaScript. Con el tiempo, AngularJS evolucionó a una versión completamente nueva y rediseñada, conocida como "Angular" (a partir de la versión 2 en adelante), que se lanzó en 2016. Esta nueva versión fue construida desde cero utilizando TypeScript, un superconjunto de JavaScript que añade tipado estático y otras características avanzadas.

Características Clave de Angular

Arquitectura Basada en Componentes: Angular organiza la interfaz de usuario en componentes modulares y reutilizables, cada uno con su propia lógica, vista y estilos. Esta arquitectura facilita la gestión del código y el desarrollo de aplicaciones complejas.

- **Inyección de Dependencias**: El sistema de inyección de dependencias de Angular permite gestionar y proporcionar servicios y objetos a los componentes de manera eficiente, mejorando la modularidad y simplificando las pruebas y el mantenimiento del código.

- **Sistema de Ruteo**: Angular incluye un sistema de ruteo que gestiona la navegación entre diferentes vistas sin recargar la página completa. Esto permite una experiencia de usuario fluida y continua, esencial para las SPA.
- **Data Binding Bidireccional**: El enlace bidireccional de datos en Angular sincroniza automáticamente el modelo y la vista, reflejando cualquier cambio en uno de ellos en el otro, lo que reduce la necesidad de código adicional para manejar estos cambios.
- **Directivas**: Las directivas en Angular extienden el HTML con nuevas funcionalidades y comportamientos, permitiendo modificar la estructura del DOM o ajustar el comportamiento y estilo de los elementos.
- **Pipes**: Los pipes transforman datos en las plantillas, permitiendo formatear información como fechas y números directamente en la vista, sin necesidad de lógica adicional en el componente.
- **Formularios**: Angular ofrece dos enfoques para manejar formularios: los formularios basados en plantillas (Template-driven Forms) y los formularios reactivos (Reactive Forms), proporcionando herramientas para la validación y el manejo de datos de entrada del usuario.
- **Angular CLI**: La Angular CLI (Command Line Interface) facilita la creación, desarrollo y mantenimiento de aplicaciones Angular. Permite generar componentes, servicios y módulos, y realizar tareas de construcción y despliegue de manera eficiente.

Herramientas necesarias para el desarrollo con Angular

En esta unidad, profundizaremos en las herramientas y procesos necesarios para trabajar con Angular, comenzando con una introducción a Node.js, Angular CLI y Typescript. También aprenderemos cómo iniciar un proyecto Angular y cómo se estructura este proyecto.

Node.js

Node.js es un entorno de ejecución de JavaScript basado en el motor V8 de Google Chrome. Su principal objetivo es permitir la ejecución de JavaScript del lado del servidor, lo que abre nuevas posibilidades para el desarrollo de aplicaciones web y servidores.

En el contexto de Angular, Node.js es fundamental por las siguientes razones:

Gestión de Dependencias: Node.js utiliza npm (Node Package Manager) para gestionar las dependencias del proyecto. Cuando trabajamos con Angular, npm se encarga de instalar y actualizar Angular y sus bibliotecas, así como otras herramientas necesarias para el desarrollo.

Angular CLI: Angular CLI es una herramienta construida con Node.js. Utiliza Node.js para ejecutar comandos de línea de comandos que generan, construyen y mantienen aplicaciones Angular, facilitando el flujo de trabajo del desarrollo.

Automatización de Tareas: Node.js y npm permiten automatizar tareas de desarrollo como la construcción del proyecto, el linting del código, y la ejecución de pruebas. Esto mejora la eficiencia y asegura que el código siga las mejores prácticas.

Servidor de Desarrollo: Node.js es utilizado por Angular CLI para servir la aplicación en un servidor de desarrollo local, lo que permite ver los cambios en tiempo real mientras desarrollamos.

Documentación

A continuación, te proporcionamos un enlace que engloba toda la información y documentación oficial necesaria para trabajar con Node.js. Puedes acceder a esta valiosa fuente de conocimiento a través del siguiente enlace:

https://nodejs.org/es/docs.

Angular CLI

Angular CLI (Command Line Interface) es una herramienta de línea de comandos desarrollada por el equipo de Angular que facilita la creación, el desarrollo y la administración de proyectos basados en Angular. Angular CLI automatiza muchas de las tareas repetitivas y configuraciones iniciales, lo que agiliza el proceso de desarrollo y garantiza que los proyectos sigan las mejores prácticas recomendadas por el equipo de Angular.

Permite realizar, entre muchas más, las siguientes acciones:

- **Generación de Código**: Angular CLI permite crear componentes, servicios, módulos y otros artefactos de la aplicación con comandos simples
- **Configuración del Proyecto**: CLI gestiona automáticamente la configuración de tu proyecto, incluyendo el archivo `angular.json` que define las opciones de construcción y configuración de la aplicación.
- **Gestión de Dependencias**: Angular CLI utiliza npm para gestionar las dependencias del proyecto. El archivo `package.json` lista todas las dependencias necesarias, y puedes instalar o actualizar estas dependencias utilizando comandos npm. Angular CLI automáticamente actualiza `package.json` cuando instalas nuevas dependencias o creas un nuevo proyecto
- **Construcción y Despliegue**: Con comandos como `ng build`, Angular CLI construye y optimiza la aplicación para producción, manejando la minificación y la optimización de los recursos.
- **Servidor de Desarrollo**: El comando `ng serve` inicia un servidor de desarrollo que permite ejecutar y probar la aplicación localmente, con recarga automática de cambios durante el desarrollo.

Documentación

https://angular.dev/tools/cli

Typescript

TypeScript es un lenguaje de programación desarrollado por Microsoft que se basa en JavaScript, pero añade características adicionales que mejoran el desarrollo y mantenimiento del código. Al ser un superset de JavaScript, TypeScript se compila a JavaScript, lo que significa que el código TypeScript se convierte en código JavaScript estándar que puede ejecutarse en cualquier entorno compatible con JavaScript.

Características:

Las principales características de TypeScript son:

- **Tipado Estático**: TypeScript permite definir tipos para variables, parámetros, funciones y objetos. Este tipado estático ayuda a detectar errores durante el proceso de compilación en lugar de en tiempo de ejecución. Esto mejora la calidad del código y la detección temprana de errores.
- **Clases y Decoradores**: TypeScript soporta la sintaxis de clases, que facilita la creación de estructuras orientadas a objetos. Los decoradores, como @Component o @Injectable, permiten añadir metadatos a las clases y configurar su comportamiento. Aunque estos son más utilizados en frameworks como Angular, los decoradores pueden ser útiles en otros contextos.
- **Interfaces y Tipos Avanzados**: Las interfaces en TypeScript definen contratos para estructuras de datos y funciones, lo que facilita la creación de código más robusto y coherente. TypeScript también soporta tipos avanzados, como tipos literales, uniones y tuplas, que permiten definir estructuras de datos más complejas y precisas.
- **Modularidad**: TypeScript soporta módulos, lo que permite organizar el código en archivos separados y gestionar dependencias de manera efectiva. Esto mejora la mantenibilidad y la escalabilidad del código, permitiendo una mejor estructura del proyecto.

Documentación

A continuación, te proporcionamos el enlace a la documentación oficial de TypeScript:

https://www.typescriptlang.org/docs/

Creando nuestro primer proyecto de Angular

Configuración Angular CLI

Antes de comenzar a ejecutar los comandos del **Angular CLI** es necesario contar **NodeJS** y **NPM** en el sistema operativo. Siempre se debe descargar desde la página oficial para evitar descargar software malicioso.

La página oficial de node es la siguiente: https://nodejs.org/ aquí podrá encontrar varias versiones y para varios sistemas operativos, a la hora de elegir una versión se recomienda siempre trabajar con las versiones LTS las cuales nos dan soporte a largo plazo a diferencia de las STS.

Comandos

Todos los comandos serán ejecutados en una terminal (cmd, bash, etc), incluso es posible utilizar la terminal integrada de VSCode. Lo primero que es necesario realizar es instalar Angular CLI lo que nos va a permitir utilizar comandos para la creación de los distintos elementos que proporciona angular. Para ello recurriremos al administrador de paquetes de node (NPM) ejecutando el siguiente comando:

```
npm install -g @angular/cli
```

Descomposición de comando:

- **npm**: es la librería que se está llamando, como se nombró anteriormente esta hace referencia a Node Package Manager (Administrador de Paquetes de Node).
- **install**: hace referencia a la operación que se realizará, en este caso instalar.
- **-g**: es un parámetro opcional que indica que dicho paquete va a ser instalado de manera global en el equipo para que esté disponible en cualquier lugar que se necesite, por ejemplo, en caso de crear un nuevo proyecto al hacerlo de esta manera no será necesario instalarlo nuevamente.
- **@angular/cli**: es el nombre del paquete que se desea instalar, en este caso es el Comand Line Interface de Angular.

Creación de proyecto

Una vez que se tiene el Angular CLI ya es posible crear un nuevo proyecto en Angular. Para ello hay que ejecutar el siguiente comando:

```
ng new nombre-del-proyecto
```

Dicho comando realizará algunas preguntas sobre el routing (por ahora podemos poner que no) y los estilos que desea aplicar (por el momento podemos utilizar el ofrecido por defecto “css”).

Descomposición de comando:

- **ng**: hace referencia a Angular.
- **new**: operación que se va a realizar, en este caso crear un nuevo proyecto.
- **nombre-del-proyecto**: este es el nombre que le desea poner al proyecto. En caso de tener separaciones en el nombre del proyecto es necesario indicarlas con un guión medio.

Una vez finalizado el proceso de creación del proyecto lo siguiente es correr el mismo. Para ello será necesario utilizar un comando para iniciar el servidor de desarrollo integrado. Este servidor se encarga de compilar la aplicación Angular en el entorno local, facilitando la visualización y probar la aplicación mientras se desarrolla. El comando que se usará es:

```
ng serve
```

Luego de ejecutar este comando se mostrará un mensaje indicando el servidor de desarrollo que se creó. Internamente este comando compila el código TypeScript de la aplicación en código JavaScript creando un conjunto de archivos que el navegador pueda entender. Además, monitorea los cambios en los archivos de la aplicación y recarga automáticamente la página del navegador cada vez que detecta un cambio en el código fuente. Esto facilita el proceso de desarrollo, ya que permite ver inmediatamente los resultados de los cambios realizados en la aplicación.

Estructura proyecto

Cada una de las carpetas generadas en un proyecto de Angular tienen un propósito específico. Se presenta una estructura típica de un proyecto:

```
nombre-del-proyecto/
├── node_modules/   # Dependencias de Node.js
├── src/            # Código fuente de la aplicación
|   ├── app/        # Código de la aplicación
|   |   ├── components/    # Componentes de la aplicación
|   |   ├── services/      # Servicios de la aplicación
|   |   ├── app.module.ts  # Módulo principal de la aplicación
|   |   ├── app.component.ts   # Componente principal de la aplicación
|   |   └── ...            # Otros archivos de la aplicación
|   ├── assets/     # Recursos estáticos (imágenes, archivos, etc.)
|   ├── environments/   # Configuración de entornos
|   ├── index.html    # Punto de entrada HTML
|   ├── main.ts       # Punto de entrada de la aplicación
|   ├── styles.scss   # Archivo global de estilos
|   └── ...
├── angular.json    # Configuración del proyecto Angular
├── package.json    # Información y dependencias del proyecto
├── tsconfig.json   # Configuración del compilador TypeScript
├── README.md       # Documentación del proyecto
└── ...
```

Dicha estructura puede variar dependiendo los equipos y definiciones que puedan tener los distintos equipos aunque los archivos creados por angular al crear el proyecto es recomendable no modificarlos.

La carpeta **node_modules**, que contiene todas las dependencias, no debe ser incluida en el repositorio. Por lo tanto, cuando se trabaja en proyectos que ya han sido creados, es necesario regenerar esta carpeta en el entorno local. Para lograrlo, simplemente ejecuta el siguiente comando de instalación de paquetes en la misma ubicación donde se encuentra el archivo **package.json**.

```
npm install
```

Este comando restablecerá las dependencias necesarias según las especificaciones del archivo **package.json**.

Componentes de Angular

¿Qué son los Componentes en Angular?

En Angular, un componente es una unidad fundamental de la interfaz de usuario (UI). Cada componente es una pieza de la interfaz que encapsula su lógica, datos y presentación. Los componentes permiten construir aplicaciones modulares y reutilizables, dividiendo la interfaz en partes manejables y organizadas.

¿Para qué sirven los componentes?

- **Encapsulamiento**: Los componentes encapsulan la lógica de la interfaz, mejorando la organización del código.
- **Reusabilidad**: Pueden reutilizarse en diferentes partes de la aplicación, evitando duplicación.
- **Mantenimiento**: Facilitan actualizaciones y cambios en partes específicas sin afectar al resto.

Creación de un Componente

Componentes Tradicionales (con Módulos)

Para crear un componente tradicional (antes de Angular 17), se usa el CLI:

```
ng generate component nombre-del-componente
```

Componentes Autónomos (Standalone Components)

A partir de Angular 17/18, se introdujeron los Standalone Components, que no requieren módulos. Para crearlos, usa:

```
ng generate component nombre-del-componente --standalone
ng g c nombre-del-componente --standalone
```

Descomposición del Comando

- **ng**: Referencia a Angular.
- **generate (o g)**: Operación para generar un elemento.
- **component**: Tipo de elemento a crear.
- **nombre-del-componente**: Nombre del componente (usando guiones para separar palabras).
- **--standalone**: Opción para crear un componente autónomo (Angular 17+).

Estructura de un Componente

Componente Tradicional

El CLI genera:

- **nombre-del-componente.component.ts:** Lógica en TypeScript.
- **nombre-del-componente.component.html:** Plantilla HTML.
- **nombre-del-componente.component.css:** Estilos encapsulados.
- **nombre-del-componente.component.spec.ts:** Pruebas unitarias.

Ejemplo del archivo **.ts** tradicional:

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

no usages
@Component({
    selector: 'app-mi-componente',
    templateUrl: './mi-componente.component.html',
    styleUrls: ['./mi-componente.component.css']
})
export class MiComponente {
    // Lógica del componente
}
```

Componente Autónomo (Standalone)

En Angular 17+, el componente incluye **standalone: true** y gestiona sus propias dependencias:

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

no usages
@Component({
    selector: 'app-mi-componente',
    templateUrl: './mi-componente.component.html',
    styleUrls: ['./mi-componente.component.css'],
    standalone: true, // Indica que es autónomo
    imports: [/* Componentes o directivas usadas en la plantilla */]
})
export class MiComponente {
    // Lógica del componente
}
```

Uso de un Componente

Componentes Tradicionales

Deben declararse en un módulo. Ejemplo en **app.module.ts**:

```
import { MiComponente } from './mi-componente/mi-componente.component';

no usages
@NgModule({
    declarations: [MiComponente],
    // ...
})
export class AppModule { }
```

Luego, se usa en HTML con su selector:

```
<app-mi-componente></app-mi-componente>
```

Componentes Autónomos (Standalone)

No requieren módulos. Se importan directamente donde se necesitan:

```
import { Component } from '@angular/core';
import { MiComponente } from './mi-componente/mi-componente.component';

no usages
@Component({
    selector: 'app-otro-componente',
    templateUrl: './otro-componente.component.html',
    styleUrls: ['./otro-componente.component.css'],
    standalone: true,
    imports: [MiComponente] // Importa el componente autónomo aqui
})
export class OtroComponente {
    // Lógica del componente
}
```

Y en HTML:

```
<app-mi-componente></app-mi-componente>
```

Ventajas de los Standalone Components (Angular 17+)

- **Sin módulos**: Elimina la necesidad de declarar componentes en módulos.
- **Código más limpio**: Reduce archivos de configuración y boilerplate.
- **Lazy Loading simplificado**: Carga componentes directamente en rutas sin módulos.
- **Reusabilidad mejorada**: Los componentes son autocontenidos y fáciles de compartir.

Configuración de Rutas con Standalone Components

En Angular 17+, puedes definir rutas usando componentes autónomos:

```
import { Routes } from '@angular/router';
import { MiComponente } from './mi-componente/mi-componente.component';

no usages
export const routes: Routes = [
    {
        path: 'ruta',
        component: MiComponente, // Componente autónomo
    }
];
```

Inyección de Dependencias en Standalone Components

Funciona igual que en componentes tradicionales. Ejemplo con un servicio:

```
import { Component } from '@angular/core';
import { MiServicio } from './mi-servicio.service';

no usages
@Component({
    selector: 'app-mi-componente',
    templateUrl: './mi-componente.component.html',
    styleUrls: ['./mi-componente.component.css'],
    standalone: true
})
export class MiComponente {
    no usages
    constructor(private miServicio: MiServicio) {
        // Usa el servicio aqui
    }
}
```

¿Cuándo usar Standalone Components?

- **Aplicaciones nuevas**: Angular 17+ recomienda usar componentes autónomos como estándar.
- **Refactorización**: Migrar gradualmente componentes antiguos a autónomos.
- **Librerías**: Para crear componentes reutilizables sin depender de módulos.

Conclusión

Los **Standalone Components** son una evolución clave en Angular (versiones 17/18) que simplifican la arquitectura de las aplicaciones, eliminando la necesidad de módulos en la mayoría de los casos. Esta característica permite un desarrollo más ágil, código más limpio y una mejor escalabilidad. Se recomienda adoptarlos en proyectos nuevos y migrar progresivamente los existentes.

Binding

One way binding

Se refiere a la forma en que los datos fluyen desde el componente de Angular hacia la vista (la plantilla). Permite que los valores de las propiedades del componente se reflejen en la vista, pero no permite que los cambios en la vista afecten directamente las propiedades del componente. Esto garantiza una dirección única de flujo de datos, lo que simplifica el seguimiento de cómo los datos se actualizan y evita posibles problemas de sincronización.

Hay dos tipos posibles:

- **Interpolación**: la interpolación se utiliza para mostrar valores de propiedades del componente en la vista. Se realiza utilizando dobles llaves. Por ejemplo:

```
<p>{{ mensaje }}</p>
```

Donde mensaje es una propiedad del componente (clase typescript), por ende dicho valor será mostrado dentro de las etiquetas de párrafo:

```
export class SaludoComponent {
    mensaje: string = '¡Hola, mundo!';
}
```

- **Enlace de propiedades**: conocido como **"property binding"**, permite enlazar una propiedad del DOM a una propiedad del componente. Esto se realiza utilizando corchetes []. Por ejemplo:

Componente:

```
export class EjemploComponent {
    imagenUrl: string = 'https://example.com/imagen.jpg';
}
```

Template:

```
<img [src]="imagenUrl" alt="Imagen de ejemplo">
```

En este caso nombre es una propiedad del componente, por lo que el valor de la imagen será establecido con el valor que tenga la propiedad imageUrl del componente. En el caso del enlace unidireccional, los cambios en el componente se reflejarán en la vista automáticamente. Sin embargo, si los valores en la vista cambian esos cambios no se reflejarán en el componente de forma directa.

Two way binding

Es una característica que permite mantener sincronizados automáticamente los valores entre una propiedad del componente y un elemento en la vista, como un campo de entrada. Esto significa que los cambios realizados en la vista se reflejarán en el componente y viceversa, lo que simplifica la gestión de los datos entre ambas direcciones.

Para ello debemos agregar un evento que modifique el valor de la propiedad que se haya asociado a un elemento de la vista. A continuación, se plantea un ejemplo:

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

no usages
@Component({
    selector: 'app-root',
    template: `
     <input [value]="nombre" (input)="onNombreChange($event)">
     <p>Nombre: {{ nombre }}</p>
    `,
})
export class AppComponent {
    nombre: string = '';

    1+ usages
    onNombreChange(event: any) : void  {
        this.nombre = event.target.value;
    }
}
```

El evento **(input)** se activa cada vez que el contenido del campo de entrada cambia. El método **onNombreChange()** actualiza la propiedad nombre del componente con el valor del campo de entrada.

Existen otras alternativas más sencillas como el uso de la directiva **ngModel** que ***será visto en otra unidad***.

Servicios

Son clases que se utilizan para organizar y compartir lógica, datos y funcionalidad comunes entre diferentes componentes de una aplicación. Los servicios desempeñan un papel fundamental en la arquitectura de una aplicación Angular, ya que permiten la separación de preocupaciones y promueven la reutilización de código.

Características:

- **Reutilización de código:** Puedes crear servicios para encapsular la lógica o la funcionalidad que se usa en varios lugares de tu aplicación. Esto promueve la reutilización del código y evita la duplicación.
- **Inyección de dependencias:** Angular tiene un sistema de inyección de dependencias incorporado que se utiliza para proporcionar instancias de servicios a componentes u otros servicios que los necesiten. Esto facilita la gestión de dependencias y asegura que los servicios se compartan de manera eficiente.
- **Singleton por defecto:** Cuando inyectas un servicio en un componente o en otro servicio, Angular crea una única instancia de ese servicio y la comparte en toda la aplicación. Esto asegura que los datos y la funcionalidad compartidos sean coherentes en toda la aplicación.
- **Separación de preocupaciones:** Los servicios ayudan a separar la lógica de negocios y la manipulación de datos de la presentación. Los componentes se centran en la interfaz de usuario y delegan tareas específicas a los servicios.
- **Interacción con datos externos:** Los servicios son comunes para interactuar con servidores, bases de datos o servicios web externos para recuperar o enviar datos. Pueden manejar solicitudes HTTP, WebSocket, almacenamiento local y otras operaciones de E/S.

Los servicios pueden actuar como intermediarios para la comunicación entre componentes. Pueden emitir eventos o utilizar Observables (a menudo con RxJS)

para notificar a otros componentes sobre cambios o eventos importantes en la aplicación. Esto facilita la creación de aplicaciones interactivas y dinámicas.

Son útiles para gestionar la autenticación y la autorización en una aplicación. Puedes crear un servicio de autenticación que maneje la lógica de inicio de sesión, verificación de tokens y control de acceso, lo que permite que varios componentes accedan a esta funcionalidad de manera consistente.

En resumen, facilitan la organización del código, la separación de preocupaciones, la reutilización y la comunicación eficiente entre componentes. Además, permiten gestionar la lógica de la aplicación, la configuración global y la interacción con recursos externos de manera eficiente y coherente.

Ejemplo:

Para crear un servicio se puede utilizar Angular CLI por medio del siguiente comando:

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

Descomposición de comando:

- ng: hace referencia a Angular.
- generate: operación que desea realizar, en este caso generar o crear un elemento en angular, también puede escribirse con la letra “g” únicamente.
- service: elemento que desea generar, en este caso se está solicitando la creación de un servicio.
- nombre-del-servicio: como lo dice su nombre es el nombre que se le va a asignar al servicio en este caso. Al igual que en la creación de un proyecto los guiones indican la separación entre las palabras, Angular resuelve esto haciendo mayúscula la primera letra después de cada guion en los archivos de clase (se verá más adelante).

Suponiendo que se creó un servicio **persona** este es el código generado:

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

no usages
@Injectable({
    provideIn: root,
})
export class PersonaService {
    private personas: string[] = [];

    no usages
    agregarPersona(nombre: string) : void  {
        this.personas.push(nombre);
    }

    no usages
    obtenerPersonas() {
        return this.personas;
    }
}
```

Una clase la cual contiene el decorador **@Injectable** que indica que dicho servicio va a poder ser inyectado como una dependencia en otros lugares de mi aplicación (por ejemplo: un componente, otro servicio, etc). Por defecto Angular añade la propiedad **provideIn: 'root'** que permite trabajar como singleton por lo que estará disponible en toda la aplicación sin necesidad de una configuración adicional.

Luego para utilizar dicho servicio es necesario inyectar en donde se necesite utilizarlo, en este caso se verá un ejemplo con un componente:

```
import { Component } from '@angular/core';
import { PersonaService } from './persona.service';

no usages
@Component({
    selector: 'app-persona-listado',
    template: `
    <div>  <h2>Personas</h2>
      <ul>
    @for(persona of personas; track persona.id){
       <li >{{ persona }}</li>
       }
      </ul>
      <input [(ngModel)]="nombre" placeholder="Nueva persona" />
      <button (click)="agregarPersona()">Agregar persona</button>
    </div>
  `,
})
export class PersonaComponent {
    personas: string[] = [];
    nombre: string = '';


    no usages
    constructor(private personaService: PersonaService) {
        this.personas = this.personaService.obtenerPersonas();
    }
    1+ usages
    agregarPersona() : void  {
        this.personaService.agregarPersona(this.nombre);
        this.nombre = '';
    }
}
```

Para ello Angular utiliza un mecanismo de inyección desde el constructor por lo que es necesario declarar aquellas dependencias en el constructor junto con su modo de acceso. En caso de utilizarlo unicamente en el archivo typescript se puede declarar como privado. Si es necesario utilizarlo en el template tiene que ser declarado como protected o public, siempre es recomendable utilizar el modo de acceso más restringido o la mayor restricción posible para cumplir con uno de los principios **SOLID**.

En caso de no utilizar la propiedad **provideIn: 'root'** que ofrece el decorador **@Injectable** es necesario realizar una configuración adicional en el archivo de módulos. Donde se debe agregar en un arreglo llamado **providers** el servicio que se desea utilizar. En caso de no hacerlo Angular arrojará un mensaje de error diciendo que no se encuentra una dependencia para el servicio solicitado.

Ejemplo del error arrojado en consola: *No provider for ServiceName*.

A continuación, se muestra un ejemplo del código de ejemplo del servicio sin la propiedad **provideIn** y la configuración necesaria para que dicho servicio pueda ser utilizado a posterior.

- **persona.service.ts**

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

no usages
@Injectable()
export class PersonaService {
    private personas: string[] = [];

    no usages
    agregarPersona(nombre: string) : void  {
        this.personas.push(nombre);
    }

    no usages
    obtenerPersonas() {
        return this.personas;
    }
}
```

- **persona-component.ts**

```
import { Component, OnInit } from '@angular/core';
import { PersonaService } from './persona.service';

no usages
@Component({
    selector: 'app-persona-componente',

    providers: [PersonaService], // Solo accede el componente de persona
    standalone: true,
    template:
        `
    <div>  <h2>Personas</h2>
      <ul>
    @for(persona of personas; track persona.id){
       <li >{{ persona }}</li>
       }
      </ul>
      <input [(ngModel)]="nombre" placeholder="Nueva persona" />
      <button (click)="agregarPersona()">Agregar persona</button>
    </div>
  `,
})
export class PersonaComponent {
  personas: string[] = [];
  nombre: string = '';

  no usages
  constructor(private personaService: PersonaService) {
    this.tasks = this.personaService.obtenerPersonas();
  }
  1+ usages
  agregarPersona() : void  {
    this.personaService.agregarPersona(this.newTask);
    this.newTask = '';
  }
}
```

Al especificar providers: [PersonaService] en el decorador del componente, estás creando una instancia de PersonaService solo para PersonaComponente. Esto significa que no puedes usar PersonaService en otros componentes a menos que lo proporciones nuevamente en sus providers.

Esta es una técnica útil cuando quieres limitar el alcance de un servicio a un solo componente, lo que puede ayudar a mantener la separación de responsabilidades y mejorar la gestión de recursos en tu aplicación Angular.

HTTP

Petición HTTP

Una petición HTTP (Hypertext Transfer Protocol) es un mensaje que un cliente (como un navegador web) envía a un servidor web para solicitar algún tipo de recurso o realizar una acción en el servidor. Las peticiones HTTP son el fundamento de la comunicación en la World Wide Web y se utilizan para acceder a sitios web, obtener datos de servidores, enviar datos a servidores y realizar otras operaciones en línea.

Una solicitud HTTP generalmente consta de los siguientes elementos:

- **Verbo HTTP**: Este es un método que indica la acción que se debe realizar en el servidor. Los verbos HTTP más comunes son:
  - GET: Para solicitar datos del servidor.
  - POST: Para enviar datos al servidor, como al enviar formularios.
  - PUT: Para actualizar un recurso en el servidor.
  - DELETE: Para eliminar un recurso en el servidor.
  - Otros, como HEAD, OPTIONS, PATCH, etc.
- **URL (Uniform Resource Locator)**: La URL indica la dirección del recurso que se desea acceder en el servidor. Esto puede ser una página web, un archivo, una API, etc.
- **Encabezados HTTP**: Los encabezados son metadatos adicionales que acompañan a la solicitud y proporcionan información adicional sobre la solicitud. Los encabezados pueden incluir información como el tipo de contenido que se acepta, cookies, información de autenticación, y más.
- **Cuerpo de la solicitud (opcional):** En algunas solicitudes, como las solicitudes POST o PUT, se puede incluir un cuerpo que contiene los datos que se envían al servidor. Por ejemplo, al enviar datos de un formulario.

Una vez que el servidor recibe una solicitud HTTP, procesa la solicitud, toma la acción apropiada según el verbo y la URL, y luego devuelve una respuesta HTTP al cliente. La respuesta también incluye un código de estado que indica el resultado de la solicitud (por ejemplo, 200 para éxito, 404 para recurso no encontrado, 500 para error interno del servidor, etc.), encabezados de respuesta y, en muchas ocasiones, un cuerpo de respuesta que contiene los datos solicitados o información adicional.

Asincronismo

Se refiere a un estilo de programación en el cual las tareas no se ejecutan de manera secuencial, una después de la otra, sino que pueden ejecutarse de manera concurrente y en un orden no determinista. Esto significa que las tareas pueden comenzar y finalizar en momentos diferentes y en un orden que puede variar.

El asincronismo es especialmente útil en situaciones en las que se pueden producir operaciones lentas o bloqueantes, como lecturas/escrituras de archivos, solicitudes a servidores remotos o cualquier operación que pueda llevar tiempo. En lugar de esperar a que una tarea se complete antes de continuar con la siguiente, el programa puede continuar ejecutando otras tareas y responder a eventos a medida que se produzcan.

En muchos lenguajes de programación, se utilizan conceptos y mecanismos para implementar la asincronía, como:

- **Hilos (Threads):** Los hilos son subprocesos de ejecución dentro de un programa que pueden ejecutarse de manera concurrente. Los programas que utilizan hilos pueden realizar múltiples tareas al mismo tiempo.
- **Callbacks:** Los callbacks son funciones que se pasan como argumentos a otras funciones y se ejecutan cuando se completa una operación asincrónica. Son comunes en JavaScript y se utilizan en la programación basada en eventos.
- **Promesas (Promises):** Las promesas son objetos que representan un valor que puede estar disponible ahora, en el futuro o nunca. Se utilizan para manejar operaciones asincrónicas en un estilo más estructurado y legible.
- **Async/Await:** Este es un enfoque más moderno y legible para trabajar con código asincrónico en muchos lenguajes, incluyendo JavaScript y Python. Permite escribir código asincrónico de manera similar a las operaciones síncronas, lo que facilita su comprensión.

Observables and Subscriptions

Los observables son una parte fundamental de la programación reactiva y se utilizan comúnmente en aplicaciones Angular para gestionar flujos de datos asíncronos. Los observables proporcionan una forma elegante de trabajar con secuencias de datos que pueden cambiar con el tiempo.

¿Qué es?

Un observable es un objeto que representa una secuencia de datos asíncronos. Estos datos pueden ser eventos, resultados de solicitudes HTTP, cambios en propiedades, o cualquier otra fuente de información que cambie con el tiempo. Los observables permiten observar (escuchar) estos cambios y reaccionar a ellos de manera asincrónica.

Características claves:

- **Asincronía**: Los observables se utilizan para manejar operaciones asíncronas, lo que significa que no bloquean la ejecución del programa mientras esperan resultados.
- **Emisión de datos**: Los observables emiten datos a lo largo del tiempo. Pueden emitir varios valores en una secuencia.
- **Manejo de errores**: Los observables permiten manejar errores que pueden ocurrir durante la ejecución de una operación asíncrona.
- **Completado**: Un observable puede completarse, lo que significa que no emitirá más datos y se dará por terminado.

Usos más comunes:

- **Solicitudes HTTP**: Cuando haces una solicitud HTTP utilizando el módulo HttpClient, obtienes un observable como respuesta. Puedes suscribirte a este observable para recibir los datos de respuesta.
- **Manejo de eventos**: Puedes usar observables para manejar eventos del usuario, como clics de botones o entradas de formularios, utilizando la API de RxJS en Angular.
- **Actualizaciones en tiempo real**: Si trabajas con aplicaciones en tiempo real, como chat en vivo o paneles de control, los observables son útiles para recibir actualizaciones automáticas de los datos del servidor.

- **Manejo de datos reactivos**: Los observables son una parte fundamental del enfoque reactivo en Angular, que permite construir interfaces de usuario altamente responsivas y dinámicas.

En Angular, los observables se implementan utilizando RxJS (Reactive Extensions for JavaScript), una biblioteca que proporciona una amplia gama de operadores para transformar y combinar observables.

En Angular, las suscripciones se utilizan para observar (escuchar) los valores emitidos por un observable. Un observable es una secuencia de eventos o datos que pueden cambiar con el tiempo. Para interactuar con los valores emitidos por un observable, debes suscribirte a él. Ejemplo:

- **Crear un Observable:**

```
import { Observable, interval } from 'rxjs';

const intervalo : Observable<number>  = interval( period: 1000);
// Emite un valor cada segundo
```

- **Suscribirse al Observable:**

Una vez que tienes un observable, puedes suscribirte a él utilizando el método **subscribe()**. Este método acepta uno o más argumentos de devolución de llamada (funciones) que se ejecutarán cuando el observable emita valores, errores o se complete. Por lo general, se proporcionan tres devoluciones de llamada: next, error y complete. Sin embargo, sólo next es obligatorio.

```
import { Observable, interval } from 'rxjs';

const suscripcion = intervalo.subscribe(
    (valor) : void  => {
        console.log('Valor emitido', valor);
        // Realiza acciones con el valor emitido
    },
    (error) : void  => {
        console.error('Error', error);
        // Realiza acciones para manejar el error
    },
    () : void  => {
        console.log('Observable completado');
        // Realiza acciones cuando el observable se completa
    },
);
```

La función **next** se ejecutará cada vez que el observable emita un valor. La función error se ejecutará si ocurre un error en el observable. La función **complete** se ejecutará cuando el observable se complete y ya no emita más valores.

- **Desuscribirse del Observable:**

Es importante desuscribirse de un observable cuando ya no necesitas observar sus valores para evitar pérdidas de memoria y problemas de rendimiento. Para hacerlo, puedes almacenar la suscripción en una variable y luego llamar al método **unsubscribe()** cuando desees cancelar la suscripción.

```
suscripcion.unsubscribe();
// Cancela la suscripción al observable
```

También puedes utilizar el operador takeUntil o el método pipe para gestionar automáticamente la desuscripción en ciertas condiciones.

HttpClient

Antes de utilizar **HttpClient** en su aplicación Angular, es imprescindible configurarlo correctamente mediante **inyección de dependencia**. Esta configuración asegura que los servicios que requieren acceso a la red puedan comunicarse con API externas de manera efectiva y segura.

Se ofrece a través de la función auxiliar **provideHttpClient()**, que permite a Angular gestionar las solicitudes HTTP de manera eficiente. Esta función generalmente se incluye en la configuración de los proveedores de la aplicación, que se define en un archivo de configuración como `app.config.ts`.

```
export const appConfig: ApplicationConfig = {
    providers: [
        provideHttpClient(),
        // Aquí puedes agregar otros proveedores que necesite tu aplicación
    ]
};
```

- **Inyectar HttpClient en tu servicio o componente:**

Para utilizar HttpClient en un servicio o componente, es necesario inyectar y en angular se puede hacer mediante el constructor.

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

no usages
@Injectable({
    provideIn: root
})
export class NuevoService {
    no usages
    constructor(private http: HttpClient) {  }

    no usages
    obtenerDatos(): Observable<Product> {
        return this.http.get<Product>(
            url: 'https://api.example.com/data'
        );
    }
}
```

- **Manejar la respuesta:**

Para obtener la respuesta de la solicitud HTTP desde el componente es necesario suscribirse a dicho método el cual una vez que la API responda emitirá la respuesta. Ejemplo:

Inyectar servicio en componente y realizar la llamada:

```
import { NuevoService } from './nuevo.service.ts';

no usages
@Component({
    selector: 'app-prueba',
    templateUrl: './app-prueba.component.html'
})
export class PruebaComponent implements ngOnInit {
    listado: Producto[] = [];
    no usages
    constructor(private servicio: NuevoService) {  }

    no usages
    ngOnInit(): void {
        this.servicio.obtenerDatos().subscribe({
            next: (response: Producto[]) : void  => {
                this.listado = response;
                // Aqui puedes realizar cualquier otra acción con los datos recibidos
            },
            error: (error: Error) : void  => {
                console.error(error);
                // Aqui puedes realizar el manejo del error como se desee,
                // en este caso se imprime un mensaje en la consola del navegador
            }
        });
    }
}
```

```
import { NuevoService } from './nuevo.service.ts';

no usages
@Component({
    selector: 'app-prueba',
    templateUrl: './app-prueba.component.html'
})
export class PruebaComponent implements ngOnInit {
    listado: Producto[] = [];
    no usages
    constructor(private servicio: NuevoService) {  }

    no usages
    ngOnInit(): void {
        this.servicio.obtenerDatos().subscribe(
            (response: Producto[]) : void  => {
                this.listado = response;
                // Aquí puedes realizar cualquier otra acción con los datos recibidos
            },
            (error) : void  => {
                console.error(error);
                // Aquí puedes realizar el manejo del error como se desee,
                // en este caso se imprime un mensaje en la consola del navegador
            }
        );
    }
}
```

Como se puede ver existen dos alternativas para el manejo de las respuestas al suscribirse a un Observable. La primera es la utilizada en el ejemplo de observables utilizando un intervalo y la segunda es una forma simplificada que no requiere la declaración de las funciones **callback** (next, error, complete).

Otro punto interesante es que no es obligatorio implementar el manejo de los errores o la completitud de la notificación emitida por el Observable, aunque es una buena práctica manejar los escenarios de errores para ofrecer un buen feedback a los usuarios de nuestras aplicaciones.

Una vez que se llama al servicio que hace la llamada a la API en caso de obtener una respuesta satisfactoria se carga el dicho resultado en una propiedad existente en el componente, en el ejemplo se asigna el arreglo de Productos proveniente de la API a la propiedad listado dentro del componente.

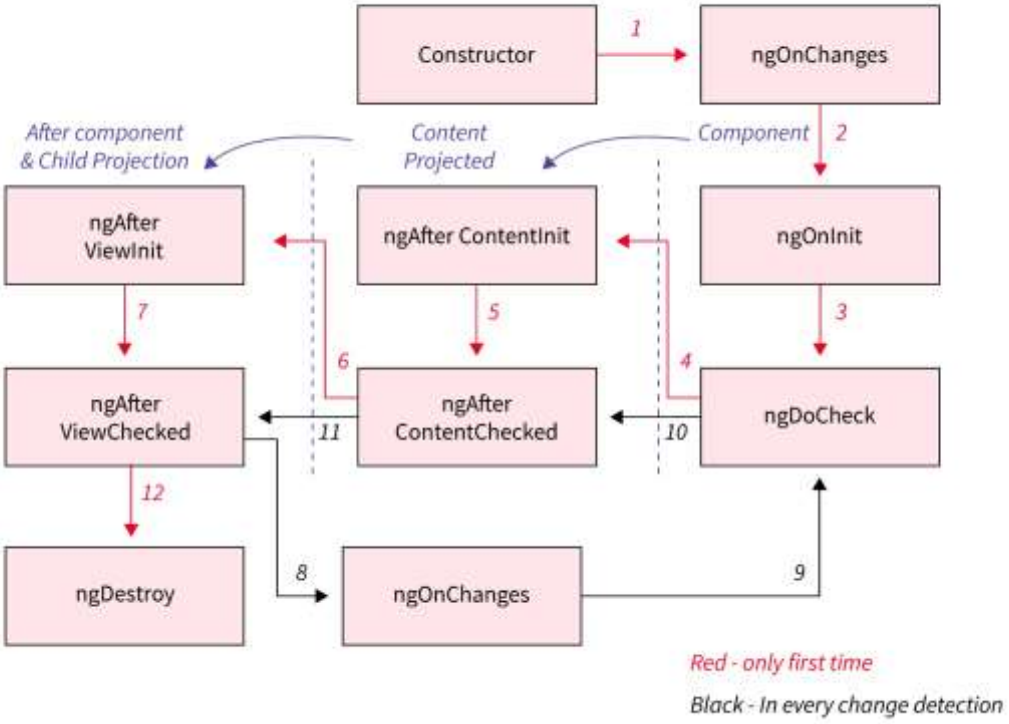
Ciclos de vida de componentes

Los componentes en Angular tienen un ciclo de vida que consta de una serie de etapas o eventos que ocurren desde la creación hasta la destrucción del componente. Estos eventos permiten que el componente responda y realice acciones en diferentes momentos durante su existencia.

Permiten a los desarrolladores controlar y responder a diferentes momentos en la vida de un componente Angular. Para aprovechar al máximo el ciclo de vida de un componente, es importante entender cuándo se dispara cada evento y cómo se pueden usar para realizar acciones específicas.

Para utilizarlos se puede hacer implementando la interfaz de cada uno de ellos o escribiendo el método directamente.

- **ngOnChanges**: es disparado cuando los valores de las propiedades de entrada (@Input) cambian. Es útil para realizar acciones en respuesta a cambios en las propiedades.
- **ngOnInit**: ocurre una vez después de que las propiedades de entrada se hayan inicializado y antes de que el componente se muestre en la vista. Es un buen lugar para realizar inicializaciones iniciales y llamadas a servicios.
- **ngDoCheck**: se dispara en cada ciclo de detección de cambios, lo que sucede cada vez que se realiza una verificación de cambios en la aplicación. Puede usarse para realizar acciones de detección de cambios personalizadas.
- **ngAfterContentInit**: ocurre después de que el contenido proyectado (contenido transcluido) del componente se haya inicializado.
- **ngAfterContentChecked**: se ejecuta después de cada verificación de cambios en el contenido proyectado.
- **ngAfterViewInit**: se dispara después de que los elementos de la vista del componente se hayan inicializado.
- **ngAfterViewChecked**: sucede después de cada verificación de cambios en los elementos de la vista.
- **ngOnDestroy**: se ejecuta justo antes de que el componente se destruya. Es un buen lugar para liberar recursos, cancelar suscripciones y realizar limpieza.


Constructor
1
ngOnChanges
After component & Child Projection
Content Projected
Component
2
ngAfter ViewInit
ngAfter ContentInit
ngOnInit
7
5
3
6
4
ngAfter ViewChecked
11
ngAfter ContentChecked
10
ngDoCheck
12
ngDestroy
8
ngOnChanges
9
Red - only first time
Black - In every change detection

Directivas

Son una parte fundamental que te permiten manipular el DOM, aplicar lógica y cambiar la apariencia de los elementos en la vista. Las directivas son instrucciones que se agregan a los elementos HTML para modificar su comportamiento o apariencia. Hay tres tipos principales de directivas en Angular:

Directivas estructurales

Modifican la estructura del DOM, agregando, eliminando o reemplazando elementos HTML en función de condiciones. Las directivas estructurales más comunes son **ngIf**, **ngFor** y **ngSwitch**.

- **ngIf**: agrega o elimina elementos del DOM basados en una condición.

```
<div *ngIf="condicion">Este elemento se mostrará si condicion es verdadero</div>
```

- **ngFor**: itera sobre una colección y genera elementos en función de los elementos de la colección.

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

no usages
@Component({
    selector: 'app-root',
    template: `
     <ul>
          <li *ngFor="let item of items">{{ item }}</li>
     </ul>
    `,
})
export class AppComponent {
    items: string[] = [
        'uno', 'dos', 'tres'
    ];
};
```

- **ngSwitch**: muestra un bloque de elementos HTML basado en múltiples condiciones. Misma función que el switch de los lenguajes de programación más utilizados.

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

no usages
@Component({
    selector: 'app-root',
    template: `
    <div [ngSwitch]="opcion">
        <p *ngSwitchCase="'opcion1'">Opción 1 seleccionada</p>
        <p *ngSwitchCase="'opcion2'">Opción 2 seleccionada</p>
        <p *ngSwitchDefault>Opción por defecto</p>
    </div>`
    ,})
export class AppComponent {
    opcion: string = 'opcion1';
}
```

Directivas de atributos

Se aplican como atributos en elementos HTML para proporcionar comportamientos específicos. Las directivas de atributo más conocidas son **ngClass**, **ngStyle**.

- **ngClass**: Permite cambiar las clases CSS de un elemento en función de condiciones.

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

no usages
@Component({
    selector: 'app-root',
    template: `
        <div [ngClass]="{'clase-activa': estaActiva, 'clase-inactiva': !estaActiva}">
            Este elemento tendrá una clase diferente basada en estaActiva
        </div>  `,
})
export class AppComponent {
    estaActiva: boolean = true;
}
```

- **ngStyle**: Permite aplicar estilos CSS a un elemento en función de condiciones.

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

no usages
@Component({
    selector: 'app-root',
    template: `
        <div [ngStyle]="{'color': estaActiva ? 'green' : 'red', 'font-weight': estaActiva ? 'bold' : 'normal'}">
            Este elemento tendrá una clase diferente basada en estaActiva
        </div>  `,
})
export class AppComponent {
    estaActiva: boolean = true;
}
```

Estos ejemplos demuestran cómo puedes usar directivas de atributos para cambiar la apariencia, el comportamiento y las interacciones de los elementos en función de condiciones dinámicas.

Directivas personalizadas

Son directivas creadas por los desarrolladores para aplicar comportamientos y lógica personalizados a los elementos del DOM. Las directivas personalizadas pueden ser directivas estructurales o de atributo.

Para crear una directiva personalizada en Angular, puedes utilizar el comando **ng generate directive <nombre-de-la-directiva>**, también es posible creando los archivos manualmente.

Código directiva:

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

no usages
@Component({
    selector: 'appColorFondo'
})
export class ColorFondoDirective implements ngOnInit {
    @Input() appColorFondo: string;

    no usages
    constructor(private el: ElementRef) {}

    no usages
    ngOnInit() : void  {
        this.el.nativeElement.style.backgroundColor = this.appColorFondo || 'green';
    }
}
```

En este caso se muestra un ejemplo de una directiva que modifica el color de fondo del elemento donde se la utilice.

Utilización directiva:

```
<div appColorFondo="red">Este elemento tendrá color rojo</div>
<div appColorFondo>
  Este elemento tendrá color de fondo verde (por defecto) ya que en caso de que
  la primera expresión sea falsa (en este caso lo será cuando sea nula o indefinida)
  tomará la segunda en su lugar.
</div>
```

El valor del color de fondo es tomado desde el valor proporcionado en el atributo declarado con el mismo nombre que la directiva. Es posible declarar otros atributos y los mismos van a ser llamados como una propiedad de entrada vista anteriormente en la comunicación de componentes.

Otras directivas

- **ng-content**

Es una directiva en Angular que se utiliza para transmitir contenido desde un componente padre a un componente hijo. Permite que el componente padre inserte contenido dentro del componente hijo en la ubicación marcada por <ng-content>. Esto es especialmente útil cuando deseas crear componentes reutilizables y flexibles que puedan adaptarse al contenido específico proporcionado por el componente padre.

La directiva <ng-content> se coloca en la plantilla del componente hijo donde se desea que se inserte el contenido transmitido. El contenido transmitido puede ser cualquier cosa, desde texto simple hasta componentes más complejos.

```
<!-- card.component.html -->
<div class="card">
  <ng-content></ng-content>
</div>

<!-- parent.component.html -->
<app-card>
  <h2>Título</h2>
  <p>Contenido del parrafo.</p>
</app-card>
```

El contenido proporcionado entre las etiquetas **<app-card></app-card>** en **ParentComponent** se insertará en el lugar donde se encuentra **<ng-content>** en la plantilla de **CardComponent**.

- **ng-template**

Es una estructura en Angular que te permite definir contenido que no se mostrará en la página de manera predeterminada, pero que puede ser referenciado y renderizado cuando sea necesario. Se utiliza en conjunto con otras directivas, como ***ngIf**, ***ngFor** y ***ngSwitch**, para controlar cuándo y dónde se muestra el contenido.

Definición y uso del ng-template

```
<!-- card.component.html -->
<ng-template #bloqueElse>
  <ng-content></ng-content>
</ng-template>

<!-- parent.component.html -->
<div *ngIf="mostrarContenido; else bloqueElse">Contenido condicional</div>
```

En este caso **mostrarContenido** es una variable en el componente typescript la cual tiene un valor booleano. En caso de ser positiva va a mostrar el contenido del párrafo. De lo contrario el sistema mostrará lo que se haya definido en el template que está renombrado como **bloqueElse**.

- **ng-container**

Es una estructura en Angular que actúa como un contenedor temporal y no se renderiza en el DOM final. Se utiliza para agrupar elementos y aplicar lógica sin agregar elementos adicionales al DOM. Esta estructura es especialmente útil cuando deseas manipular o aplicar directivas a múltiples elementos sin afectar la estructura del DOM resultante.

- **Agrupacion de elementos:** para agrupar elementos sin añadir un elemento adicional al DOM. Esto es útil cuando necesitas aplicar directivas o lógica a varios elementos sin agregar un contenedor adicional en el DOM.

```
@for (item of items) {
  <ng-container>
    <div>{{ item }}</div>
  </ng-container>
}
```

- **En directivas estructurales:** permite aplicar estas directivas a un grupo de elementos sin agregar un elemento extra al DOM.

```
@if (mostrarElemento) {
  <ng-container>
    <div>Este elemento se mostrará si mostrarElemento es verdadero</div>
  </ng-container>
}
```

- **Evitar Elementos Adicionales en el DOM:** una ventaja es que no agrega elementos adicionales al DOM final. Esto puede ser útil cuando deseas aplicar lógica o estructuras a tu plantilla sin afectar la estructura de salida.

Nuevas Directivas en Angular 18: @if, @else, @for, y @switch

Angular 18 introduce un conjunto de nuevas directivas que simplifican y mejoran la legibilidad del código en los templates. Estas directivas (`@if`, `@else`, `@for`, y `@switch`) reemplazan y modernizan algunas de las directivas estructurales tradicionales como `*ngIf`, `*ngFor`, y `*ngSwitch`. A continuación, se detalla cómo se utilizan estas nuevas directivas.

@if

La directiva `@if` se utiliza para condicionar la renderización de un elemento en función de una expresión booleana. Es una versión más limpia y legible de la tradicional `*ngIf`.

Sintaxis:

```
@if (condicion) {
  <div>Este contenido se mostrará si 'condicion' es verdadera.</div>
}
```

Ejemplo con `@else`:

`@else` se utiliza junto con `@if` para manejar la renderización condicional cuando la expresión booleana es falsa.

```
@if (condicion) {
  <div>Este contenido se mostrará si 'condicion' es verdadera.</div>
} @else {
  <div>Este contenido se mostrará si 'condicion' es falsa.</div>
}
```

Este enfoque permite un código más claro y anidado dentro del propio template, eliminando la necesidad de `ng-template` y manteniendo la lógica en un solo lugar.

@for

La directiva `@for` es la versión modernizada de `*ngFor`, utilizada para iterar sobre listas o arreglos en el template y generar un bloque de código HTML para cada ítem.

Sintaxis:

```
@for (item of items) {
  <div>{{ item.nombre }}</div>
}
```

Explicación:

`@for` se utiliza para iterar sobre una colección de `items`. En cada iteración, `item` representa el elemento actual de la colección.

Este nuevo enfoque es más natural para los desarrolladores, ya que sigue la sintaxis típica de los bucles en TypeScript.

@switch y @case

La directiva `@switch` es una alternativa más expresiva y clara a `*ngSwitch`, permitiendo la selección entre diferentes bloques de código basados en una expresión evaluada.

Sintaxis:

```
@switch (valor) {
  @case ('opcion1') {
    <div>Este contenido se muestra si 'valor' es 'opcion1'.</div>
  }
  @case ('opcion2') {
    <div>Este contenido se muestra si 'valor' es 'opcion2'.</div>
  }
  @default {
    <div>Este contenido se muestra si 'valor' no coincide con ninguna opción.</div>
  }
}
```

Explicación:

`@switch` evalúa la expresión `valor` y selecciona el bloque correspondiente a la coincidencia dentro de `@case`.

El bloque `@default` se ejecuta si no se encuentra ninguna coincidencia con las opciones dadas.

Comunicación entre componentes

La comunicación entre componentes es esencial para construir aplicaciones más complejas y modulares. Puedes lograr esto utilizando propiedades de entrada (**@Input**) y eventos de salida (**@Output**) para compartir información entre componentes.

Propiedades de Entrada (@Input):

Las propiedades de entrada permiten que un componente padre pase datos a un componente hijo. Para hacer esto:

1. **Definir la Propiedad de Entrada en el Componente Hijo:**

En el componente hijo (**HijoComponent**), define una propiedad de entrada utilizando el decorador **@Input()**:

```
@Component({
    selector: 'app-hijo',
    template: 'Hijo Component: {{ inputValue }}',
})
export class OtroComponenteComponent {
    @Input() inputValue: string = "";
}
```

2. **Usar el componente hijo con la Propiedad de Entrada**

En el componente padre (PadreComponent), utiliza el componente hijo (app-hijo) y pasa datos utilizando la propiedad de entrada (inputValue):

```
<app-hijo [inputValue]="datosDelPadre"></app-hijo>
```

Eventos de Salida (@Output):

Los eventos de salida permiten que un componente hijo emita eventos que pueden ser capturados por el componente padre. Para hacer esto:

1. **Definir el Evento de Salida en el Componente Hijo:**

En el componente hijo (HijoComponent), define un evento de salida utilizando el decorador @Output() y un EventEmitter.

```typescript
import { Component, Output, EventEmitter } from '@angular/core';

no usages
@Component({
    selector: 'app-hijo',
    template: '<button (click)="emitirEvento()">Emitir Evento</button>',
})
export class HijoComponent {
    @Output() eventoEmitido : EventEmitter<void>  = new EventEmitter<void>();

    1+ usages
    emitirEvento() : void  {
        this.eventoEmitido.emit();
    }
}
```

2. **Capturar el Evento en el Componente Padre:**

En el componente padre (PadreComponent), captura el evento del componente hijo (eventoEmitido) y realiza una acción:

```typescript
<app-hijo (eventoEmitido)="accionEnPadre()"></app-hijo>
```

Servicios (Alternativa para Comunicación)

Otra forma de comunicación entre componentes es utilizando servicios para compartir datos y estado entre ellos. Los servicios actúan como contenedores de datos y lógica que se pueden inyectar en diferentes componentes. Esta es una forma efectiva de comunicar componentes que no tienen una relación directa de padre-hijo.

Recuerda que, en Angular, los cambios en las propiedades de entrada o en los eventos de salida en los componentes hijo se reflejarán automáticamente en el componente padre. La comunicación bidireccional es una de las características poderosas de Angular que permite crear aplicaciones interactivas y reactivas.

Pipes

Son una característica que te permite transformar y formatear datos de manera fácil y legible en las plantillas HTML. Los pipes toman un valor de entrada y lo transforman en un formato deseado antes de mostrarlo en la vista. Angular proporciona varios pipes incorporados, y también permite crear pipes personalizados para realizar transformaciones específicas.

Nativos de Angular:

- **UpperCasePipe y LowerCasePipe:** transformación de texto a mayúsculas o minúsculas.

```
<p>{{ 'este texto será transformado a mayúsculas' | uppercase }}</p>
<!-- Resultado: ESTE TEXTO SERÁ TRANSFORMADO A MAYÚSCULAS -->

<p>{{ 'ESTE TEXTO SERÁ TRANSFORMADO A MINÚSCULAS' | lowercase }}</p>
<!-- Resultado: este texto será transformado a minúsculas -->
```

- **DatePipe:** utilizado para darle formato a aquellas variables de tipo fecha.

```
<p>{{ today | date }}</p>
<!-- Resultado: la fecha actual formateada -->

<p>{{ today | date: format: 'yyyy-MM-dd' }}</p>
<!-- Resultado: 2023-08-14 -->
```

- **CurrencyPipe:** sirve para darle formato a valores numéricos como monedas.

```
<p>{{ 10.30 | currency }}</p>
<!-- Resultado: $10.30 -->

<p>{{ precio | currency }}</p>
<!-- Resultado: $valor_variable_precio -->
```

- **DecimalPipe:** permite formatear números decimales.

```
<p>{{ 10.00 | number: digitsInfo: 2.2-2 }}</p>
<!-- Resultado: 10.00 -->
```

{{ valor | number[:digitosEnteros:digitosFraccionarios] }}

Descomposición decimal pipe:

- **valor**: la variable de tipo numérico que se va a formatear.
- **number**: pipe propiamente dicho
- **digitosEnteros**: la cantidad de dígitos enteros que se mostrarán. Puede ser un número o un rango (mínimo y máximo separados por un guión), como '1-5'.
- **digitosFraccionarios**: cantidad de dígitos fraccionarios que se mostrarán.
- **SlicePipe:** similar al método disponible para las variables de tipo string para recortar cadenas de texto.

```
<p>{{ 'Hola este es mi texto que va a ser cortado' | slice: start: 0: end: 4 }}</p>
<!-- Resultado: Hola  -->
```

Personalizadas:

Para crear un pipe personalizado es necesario correr el comando ng generate pipe <nombre-del-pipe> de está manera se podrá modificar el valor recibido y devolver uno nuevo a partir del mismo.

Código:

```
import { Pipe, PipeTransform } from '@angular/core';
1+ usages  new *
@Pipe({
    standalone: true,
    name: 'saludo'
})
export class SaludoPipe implements PipeTransform {
    1+ usages  new *
    transform(value: string): string {
        return '!Hola, ' + value + '!';
    }
}
```

Utilización pipe:

```
<p>{{ 'Pepe Luis' | saludo }}</p>
<!-- Resultado: !Hola, Pepe Luis! -->
```

Formulario

Un formulario es una estructura en la que los usuarios pueden ingresar y enviar información a través de campos y controles específicos. En el contexto de desarrollo web, un formulario es una parte fundamental de la interacción entre los usuarios y las aplicaciones. Los formularios se utilizan para recopilar datos, realizar acciones y enviar información a través de una página web.

Un formulario web suele estar compuesto por varios elementos, como:

- **Campos de entrada:** son campos donde los usuarios pueden ingresar datos. Pueden incluir campos de texto, casillas de verificación, botones de radio, áreas de texto y campos de selección (menús desplegables).
- **Etiquetas:** son descripciones que indican qué tipo de información se debe ingresar en un campo determinado. Proporcionan orientación a los usuarios sobre qué se espera en cada campo.
- **Botones:** permiten a los usuarios enviar el formulario o realizar otras acciones, como restablecer los valores del formulario o cancelar una operación.
- **Validación:** los formularios pueden incluir reglas de validación para garantizar que los datos ingresados sean válidos. Esto puede incluir comprobar si se han completado campos obligatorios, si los datos son del tipo correcto (números, texto, fechas, etc.) y si cumplen con ciertos criterios (por ejemplo, un correo electrónico válido).
- **Acciones:** una vez que los usuarios han completado el formulario, generalmente tienen la opción de enviarlo. La acción de envío suele llevar a cabo una operación en el servidor, como guardar la información en una base de datos, enviar correos electrónicos o realizar algún otro procesamiento.
- **Estilo y diseño:** pueden tener diferentes diseños y estilos para que se integren con el diseño general de la página web y sean fáciles de entender y usar por parte de los usuarios.

Los formularios son esenciales para interactuar con los usuarios en aplicaciones web y recopilar información. Pueden encontrarse en diversas aplicaciones y sitios web, desde el registro de usuarios, inicio de sesión y envío de comentarios hasta la compra de productos en línea.

Template Driven Forms

Los "Template-Driven Forms" (Formularios basados en plantillas) son una forma de crear y manejar formularios utilizando plantillas y directivas en el HTML, haciendo que la creación y validación de formularios sea más simple y rápida en comparación con otros enfoques.

En este enfoque, la idea es que en lugar de definir el formulario y sus elementos de entrada completamente en el componente TypeScript, aquí, gran parte de la estructura y la lógica del formulario se define directamente en la plantilla HTML utilizando directivas como **ngForm**, **ngModel**, **ngSubmit**, entre otras, para establecer la estructura del formulario y la vinculación de datos.

Estos tipos de formularios son útiles para formularios más simples y rápidos de implementar, pero pueden volverse menos prácticos para formularios más complejos que requieran lógica de validación avanzada y control granular.

Directiva ngForm

Se coloca en la etiqueta **<form>** del HTML para indicar que es un formulario gestionado por Angular. Esto habilita la funcionalidad proporcionada por Angular para rastrear el estado y comportamiento del formulario y sus controles.

Cada campo de entrada que se coloca dentro de un formulario gestionado por ngForm se convierte en un "control" que puede ser accedido a través de la instancia de ngForm. Esto permite acceder a las propiedades y métodos específicos de ese control.

Al trabajar con ngForm, Angular añadirá automáticamente clases CSS que reflejarán el estado del formulario y sus controles. Esto permite proporcionar retroalimentación visual a los usuarios sobre la validez y el estado del formulario.

Ejemplo de validación simple:

1. Crea un componente llamado **FormularioEjemplo**:
2. En el archivo **formulario-ejemplo.component.html** insertar el siguiente formulario:

```html
<form #myForm="formulario" (ngSubmit)="enviarFormulario(formulario)">
  <label for="name">Nombre:</label>
  <input type="text" id="name" name="name" ngModel required>

  <label for="email">Correo Electrónico:</label>
  <input type="email" id="email" name="email" ngModel required>

  <button type="submit" [disabled]="myForm.invalid">Enviar</button>
</form>
```

Es importante aclarar que es necesario cargar el atributo “name” para que pueda hacer el bindeo (?).

3. En el archivo typescript formulario-ejemplo.component.ts se define la siguiente lógica:

```typescript
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';

no usages
@Component({
    selector: 'app-formulario-ejemplo',
    templateUrl: './app-formulario-ejemplo',
})
export class FormularioEjemploComponent {
    no usages
    enviarFormulario(form: NgForm) : void  {
        if (form.valid) {
            console.log('Formulario válido. Datos enviados:', form.value);
        } else {
            console.log('Formulario inválido. Por favor, complete todos los campos.');
        }
    }
}
```

Directiva ngModel

Se aplica a los elementos de entrada como y establece una vinculación bidireccional entre el valor del elemento y una propiedad en el componente TypeScript. Esto significa que los cambios en el elemento de entrada actualizan automáticamente la propiedad en el componente y viceversa.

Es una directiva que se utiliza para establecer enlaces bidireccionales (**two-way binding**) entre elementos de entrada de formularios HTML y propiedades de componentes en Angular. Esta directiva es parte del módulo **FormsModule** dentro del paquete **@angular/forms**, por lo que es necesario importar este módulo en tu componente para poder utilizarla.

Algunos de los elementos de entrada que se pueden utilizar junto con esta directiva son **<input>, <select>**, **<textarea>,** entre otros.

La característica principal de ngModel es que permite sincronizar automáticamente los datos entre un elemento de formulario en la vista y una propiedad en el componente. Cuando un valor cambia en el elemento de formulario, se refleja en la propiedad del componente y viceversa. Esto simplifica la manipulación de datos en formularios y evita la necesidad de escribir mucho código para mantener la sincronización manualmente.

Primero se declara la propiedad en el componente typescript.

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

no usages
@Component({
    selector: 'app-formulario',
    templateUrl:  './app-formulario.html',
    styleUrls: ['./app-formulario.css'],
    standalone : true,
    imports : [FormsModule] ,

})
export class FormularioComponent {
    valor: string = '';
}
```

Luego en el template, es decir, en el archivo HTML se puede utilizar la directiva ngModel para para enlazar el elemento del formulario con una propiedad del componente.

```
<label>El valor de la variable es: {{ valor }}</label>
<input type="text" [(ngModel)]="valor" />
```

En este ejemplo, cada vez que el usuario escriba en el campo de entrada, la propiedad valor en el componente se actualizará automáticamente y viceversa.

Validaciones

Proporcionan validación automática en función de los atributos HTML5, como **required, min, max, pattern**, etc. También permite establecer clases CSS específicas para la presentación visual del estado de validación.

Existen herramientas que permiten a los desarrolladores mostrar las validaciones en tiempo real de distintas formas.

Las validaciones disponibles son:

- **required**: garantiza que el campo no esté vacío.
- **minlength**: garantiza que el campo tenga una cantidad mínima de caracteres permitida.
- **maxlength**: asegura que el campo tenga una cantidad máxima de caracteres permitida.
- **pattern**: garantiza que el campo respete la expresión regular especificada.

Componente typescript el cual solo contiene la declaración de la propiedad categoría de tipo string.

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

no usages
@Component({
    selector: 'app-formulario',
    templateUrl:  './app-formulario.html',
    styleUrls: ['./app-formulario.css'],
    standalone : true,
    imports : [FormsModule] ,
})
export class FormularioComponent {
    categoria: string = '';
}
```

Plantilla HTML que contiene las directivas **ngForm** y **ngModel** para realizar la asociación bidireccional.

```
<form #form="ngForm">
  <div class="form-group">
    <label for="categoria">Categoria:</label>
    <input #controlCategoria="ngModel" type="text" id="categoria" name="categoria" [(ngModel)]="categoria" required minlength="3" />

    @if (controlCategoria.invalid && (controlCategoria.dirty || controlCategoria.touched))
    {
      @if (controlCategoria.getError('required'))
      {
        <div>La categoria es obligatoria.</div>
      }
      @else if (controlCategoria.getError('minlength'))
      {
        <div>La categoria debe tener al menos 3 caracteres.</div>
      }
    }
  </div>
  <button type="submit" [disabled]="!form.valid">Enviar</button>
</form>
```

Para poder acceder a las distintas propiedades que ofrece la directiva ngForm y ngModel se hace por medio de la siguiente notación **#nombreControl="ngModel"**, la cual se descompone de la siguiente manera: la parte izquierda luego del numeral corresponde al nombre que se le asignará a dicho control y con el cual se podrá acceder a los valores proporcionados en la directiva dentro del mismo template. La parte derecha corresponde a la directiva con la que se está trabajando.

Directiva ngSubmit

Utilizada para en el formulario para capturar el evento de envío. Puedes asociar una función en el componente que se ejecutará cuando se envíe el formulario. Esta función puede acceder a los valores ingresados en los campos de entrada a través de las propiedades vinculadas con **ngModel**.

Manejo de estados

Angular proporciona automáticamente el estado del formulario (válido, inválido, sucio, limpio, etc.) y el estado de los controles individuales. Estas propiedades se pueden usar para habilitar o deshabilitar botones de envío y proporcionar retroalimentación visual al usuario.

Descripción de las propiedades disponibles para el manejo de estados de los formularios:

- **valid**: indica si el control o el formulario no contiene errores.
- **invalid**: indica si el control o formulario contiene al menos un error.
- **pristine**: indica cuando el usuario no modificó un control o formulario.
- **dirty**: indica cuando el usuario realizó alguna modificación a un control o formulario.

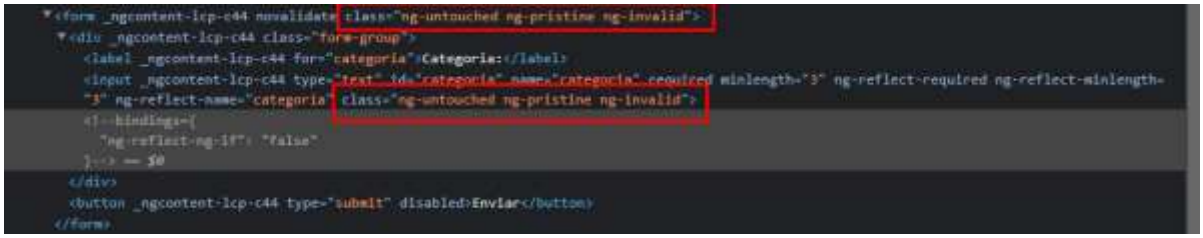
- **touched**: indica si el usuario interactuó con el control o formulario por medio de eventos como click, foco, perdiendo el foco, etc.
- **untouched**: indica si el usuario no realizó ninguna interacción con el control o formulario por medio de eventos como click, foco, perdiendo el foco, etc.

Por cada uno de ellos Angular marca las etiquetas utilizando clases CSS cada vez que el control sufre alguna modificación, de esta forma los desarrolladores pueden aplicar estilos específicos por cada uno de los estados disponibles. Dichas clases contienen el prefijo seguido del nombre del estado en el que se encuentra **ng-estado.**

Listado completo:

- **ng-valid**
- **ng-invalid**
- **ng-pristine**
- **ng-dirty**
- **ng-touched**
- **ng-untouched**

A continuación, se muestra un HTML de ejemplo del navegador con las clases marcando algunos de los estados de un control.



Ejemplo formulario con un objeto más complejo:

A la hora de trabajar con formularios Angular utiliza el concepto de **ModelGroup**, lo cual abre la posibilidad de trabajar con estructura de objetos hijos. Un ejemplo es el de una persona que tiene que registrar sus datos personales en los cuales es posible tener su nombre, apellido (entre otros datos) y luego los datos de contacto que pueden ser manejados como un formulario anidado del formulario persona, dicho formulario va a ser un **ModelGroup** que si lo vemos en forma de clase será transformado a una propiedad de tipo clase que tendrá los valores declarados en la otra clase a la que hace referencia.

A continuación, se plantea el ejemplo mencionado con la salida que tendría el objeto en formato JSON.

- **persona-alta.component.html**

```
<form #form="ngForm" (ngSubmit)="enviar(form)">
  <h1>Persona</h1>
  <div class="form-group">
    <label for="nombre">Nombre:</label>
    <input type="text" id="nombre" name="nombre" [(ngModel)]="persona.nombre" required/>
  </div>

  <div class="form-group">
    <label for="apellido">Apellido:</label>
    <input type="text" id="apellido" name="apellido" [(ngModel)]="persona.apellido" required />
  </div>
  <div class="form-group contact-container"  ngModelGroup="contacto">
    <h3>Datos contacto: </h3>
    <div class="form-group">
      <label for="email">Apellido:</label>
      <input type="email" id="email" name="email" [(ngModel)]="persona.contacto.email" required />
    </div>
    <div class="form-group">
      <label for="telefono">Telefono:</label>
      <input type="text" id="telefono" name="telefono" [(ngModel)]="persona.contacto.telefono" required />
    </div>
    <div class="form-group">
      <label for="redsocial">Red Social:</label>
      <input type="text" id="redsocial" name="redsocial" [(ngModel)]="persona.contacto.redsocial" required />
    </div>
  </div>
  <button type="submit" [disabled]="!form.valid">Enviar</button>
</form>
```

- **persona-alta.component.ts**

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

import { Persona } from './persona.model';
import { Contacto } from './contacto.model';

no usages
@Component({
    selector: 'app-persona-alta',
    templateUrl:  './app-persona-alta.component.html',
    styleUrls: ['./app-persona-alta.component.css'],
})
export class PersonaAltaComponent {
    persona: Persona = new Persona();

    no usages
    enviar(form: NgForm) : void  {
        console.log(form);
    }
}
```

- **persona.model.ts**

```
import { Contacto } from './contacto.model';

no usages
export class Persona {
    nombre: string;
    apellido: string;
    contacto: Contacto;

    no usages
    constructor() {
        this.contacto = new Contacto();
    }
}
```

- **contacto.model.ts**

```
export class Contacto {
    email: string;
    telefono: string;
    redsocial: string;
}
```

De esta forma con la directiva **ngModelGroup** se le indica que dicho contenido debe ser bindeado con la propiedad que le especifiquemos dentro de las comillas dobles.

Bibliografía

- **Componentes**

https://angular.dev/guide/components

https://angular.dev/guide/components/lifecycle

- **Pipes**

https://angular.dev/guide/pipes

https://angular.dev/api/common/DatePipe

https://angular.dev/api/common/UpperCasePipe

https://angular.dev/api/common/CurrencyPipe

https://angular.dev/api/common/DecimalPipe

- **Directivas**

https://angular.dev/guide/directives

https://angular.dev/guide/directives#built-in-attribute-directives

https://angular.dev/guide/directives#built-in-structural-directives

https://angular.dev/guide/directives#built-in-attribute-directives

https://angular.dev/guide/components/content-projection#

https://angular.dev/api/core/ng-container#

- **Inyección de dependencia**

https://angular.dev/guide/di

https://angular.dev/guide/di/dependency-injection