Angular

> ⚠ **DOCUMENTO EN DESARROLLO** ⚠

Introducción

Angular es un framework de desarrollo de aplicaciones web desarrollado y mantenido por Google. Angular está basado en el lenguaje de programación **TypeScript** y sigue el patrón de diseño de arquitectura MVC (Modelo-Vista-Controlador), específicamente la variante MVVM (Modelo-Vista-VistaModelo).

AngularJS fue desarrollado en 2009 por Misko Hevery y Adams Abrons. Posteriormente fue liberado el proyecto como una biblioteca de código abierto. Adams Abrons abandonó el proyecto pero Misko Hevery, como empleado de Google, continuó con el desarrollo y mantenimiento del proyecto.

A partir de la versión **2.0**, anunciada en Septiembre de 2014, AngularJS fue reescrito utilizando el lenguaje TypeScript y se rediseñó por completo todo el framework. Esta nueva versión pasó a llamarse únicamente **Angular**. Ambas versiones conviven, aunque AngularJS ha quedado acotada en versiones 1.X.Y mientras que Angular continúa su evolución. Ambos proyectos tiene ciclos de vida independientes.

Angular se utiliza para construir aplicaciones web de una sola página (SPA) y facilita la creación de interfaces de usuario dinámicas y interactivas.

Las aplicaciones de una sola página, conocidas como **SPAs** (*Single Page Applications*), representan un enfoque revolucionario en el desarrollo web al ofrecer una experiencia de usuario más fluida y dinámica. A diferencia de las aplicaciones tradicionales, donde la navegación implica cargar páginas completas, las SPAs cargan una única página HTML inicial y actualizan dinámicamente el contenido según las interacciones del usuario.

Las principales **características** de Angular incluyen la vinculación bidireccional de datos, que permite la sincronización automática entre la vista y el modelo de datos; inyección de dependencias, que promueve la modularidad y la reutilización de código; y un conjunto de herramientas y bibliotecas que simplifican tareas comunes como la manipulación del DOM, la gestión de eventos y la realización de peticiones HTTP.

Angular aprovecha la tecnología de los componentes web o *'web components'* y el *'Shadow DOM'* para apoyar el desarrollo impulsado por componentes. La **arquitectura** de Angular puede resumirse en:

- **Módulos**: Angular organiza la aplicación en módulos, que son conjuntos lógicos de componentes, servicios, directivas y otros artefactos relacionados. Los módulos ayudan a estructurar y organizar la aplicación, facilitando la carga y gestión de diferentes partes de la misma.

- **Componentes**: son los bloques de construcción fundamentales de una aplicación Angular. Cada componente está asociado a una vista y un controlador (o en términos de Angular, un ViewModel). La vista define la interfaz de usuario, mientras que el controlador maneja la lógica y los datos relacionados con esa vista.

- **Templates**: son archivos que definen la estructura y el diseño de las vistas en Angular. Utilizan una sintaxis especial, que combina HTML con extensiones específicas de Angular para la vinculación de datos y la manipulación de la interfaz de usuario.

- **Directivas**: son atributos especiales en el HTML que permiten extender o modificar el comportamiento de los elementos DOM. Angular incluye directivas integradas, como `*ngIf` para controlar la visibilidad de elementos, o `*ngFor` para realizar bucles sobre conjuntos de datos.

- **Servicios**: los servicios en Angular son objetos singleton que realizan funciones específicas y que pueden ser compartidos entre componentes. Se utilizan para encapsular la lógica de negocio, la manipulación de datos, o para realizar operaciones asíncronas, como llamadas HTTP.

- **Inyección de dependencias**: Angular utiliza un sistema de inyección de dependencias para proporcionar a los componentes y servicios las dependencias que necesitan. Esto facilita la modularidad y la reutilización de código al tiempo que mejora la testabilidad.

- **Vinculación de datos (*Data Binding*)**: Angular ofrece una potente vinculación de datos bidireccional, lo que significa que los cambios en el modelo de datos se reflejan automáticamente en la vista y viceversa. Esto simplifica la actualización de la interfaz de usuario en respuesta a eventos o cambios en los datos de la aplicación.

- **Enrutamiento**: Angular proporciona un módulo de enrutamiento que permite la navegación entre las distintas vistas de la aplicación sin necesidad de recargar la página. Esto es esencial para construir aplicaciones de una sola página (SPA).

⚠️ **Sección introductoria generada por ChatGPT** ⚠️

Primeros pasos

Instalación

Angular requiere **Node.js** que se instala descargándolo de su página oficial.

Cuando se instala **Node.js** también se instala ***NPM*** (*Node Package Manager*) que nos permite, entre otras cosas, gestionar las dependencias de un proyecto.

Para desarrollar en Angular se puede utilizar como lenguaje de programación tanto **Javascript** como **TypeScript**, pero dado que **Angular está implementado con TypeScript**, se recomienda usar este lenguaje.

Para instalar **Typescript** se utiliza la herramienta ***NPM*** ya que también está publicado como paquete.

> Para funcionar correctamente, cada **versión de Angular** requiere **determinadas versiones de Node.js, TypeScript y RxJS** que pueden consultarse en la siguiente tabla de la documentación oficial.

```
// Comprobar la versión de Node.js
$ node -v

// Comprobar la versión de NPM
$ npm -v

// Instalar TypeScript
$ npm install -g typescript

// Comprobar la versión de TypeScript
$ tsc --version
```

Angular CLI

'Angular CLI' es la herramienta de línea de comandos estándar para crear, depurar construir y publicar aplicaciones Angular.

Esta herramienta está publicada en **NPM** como el paquete `@angular/cli` e incluye el binario `ng` :

```
// Instalación de 'Angular CLI' de forma global
$ npm install -g @angular/cli

// Comprobar la versión de 'Angular CLI'
$ ng version

// Ver la ayuda general
$ ng help
```

```
// Ver la ayuda de un comando en concreto como 'ng serve'
$ ng serve --help
```

Se puede visitar la documentación oficial para consultar la lista completa de comandos y párametros que ofrece esta herramienta de Angular.

Puesta en marcha del proyecto

Las aplicaciones Angular se desarrollan en el contexto de una espacio de trabajo o ***'workspace'***. Un espacio de trabajo puede contener múltiples aplicaciones y bibliotecas.

Para crear un nuevo espacio de trabajo y una aplicación inicial dentro de este nuevo espacio de trabajo, se utiliza el comando `'ng new'`:

```
// Crear una nueva aplicación en un workspace
ng new my-app  // Cambiando 'my-app' por el nombre de la nueva aplicación
```

Este comando nos solicitará **información** para configurar la aplicación inicial, pudiendo utilizar la configuración que viene por defecto.

A continuación se instalan todas las bibliotecas y dependencias necesarias. Finalmente obtiene una aplicación inicial completamente funcional y las configuraciones necesarias para su depuración, pruebas y ejecución.

Por defecto la aplicación se crea con el prefijo `app` que se usará en todos los componentes pero puede personalizarse usando el modificador `--prefix`.

Desarrollo del proyecto

'Angular CLI' incluye un servidor de desarrollo lo que permite servir la aplicación fácilmente a nivel local.

El comando `'ng serve'` inicia el servidor de desarrollo, observa los cambios en el código fuente, reconstruye automáticamente la aplicación cuando detecta algún cambio en el código y recarga la página en el navegador ("*hot-reloading*"):

```
// Navegar dentro del espacio de trabajo
cd my-app

// Arranca el servidor y abre el navegador en 'http://localhost:4200'
ng serve --open
```

Compilación del proyecto

Para compilar el proyecto se utiliza el comando `'ng build'`.

Se compilará el código TypeScript en código JavaScript, se optimizará el código, se empaquetará y se comprimirán (*'minify'*) los ficheros según sea necesario.

Este comando ejecuta el constructor o *'builder'* definido en el fichero `angular.json`.

Por defecto, cuando se inicializa la aplicación con `'ng new'` se utiliza el constructor `@angular-devkit/build-angular:application`.

Existen **4 constructores disponibles**, según el tipo de aplicación que se necesite construir.

Despliegue del proyecto

Una vez la aplicación está lista para su despliegue, se puede realizar de varias formas, tanto manual como automática.

Para realizar el despliegue de **forma automática**, se dispone del comando `'ng deploy'`.

Existen desarrolladores *'third-party'* que implementan capacidades de despliegue en diferentes plataformas. Se pueden añadir cualquiera de ellos al proyecto con `'ng add'`.

- **Firebase hosting**
- **Vercel**
- **Netlify**
- **Github Pages**
- **Amazon Cloud S3**

Cuando se agrega un paquete con capacidad de despliegue, automáticamente se actualizará la configuración del espacio de trabajo (archivo `angular.json`) con una sección de despliegue para el proyecto seleccionado. Luego, se puede usar el comando `'ng deploy'` para desplegar ese proyecto.

Por ejemplo, podemos añadir un paquete de terceros para realizar el despliegue en **Firebase**:

```
# Añade el 'package' en 'angular.json'
$ ng add @angular/fire

# Ejecuta el despliegue
$ ng deploy
```

Y esto añadirá la sección `deploy` en el fichero `angular.json`:

```
{
  "projects": {
    "my-app": {
      "architect": {
        "deploy": {
          "builder": "@angular/fire:deploy",
          "options": {
            "target": "hosting:my-app"
          }
        }
      }
    }
  }
}
```

Estructura de una aplicación Angular

Cuando se ejecuta el comando `'ng new'` se instalan las bibliotecas y dependencias necesarias en el nuevo espacio de trabajo, además del **esqueleto funcional** de una aplicación dentro de la carpeta `src/`.

Esta aplicación se considera la **aplicación principal** o **aplicación raíz**. El directorio raíz del espacio de trabajo contiene todos los ficheros de configuración, etc.. necesarios para construir y servir la aplicación Angular.

La aplicación inicial creada es la **aplicación por defecto** para todos los comandos lanzados a través de `ng`.

Para un espacio de trabajo que contiene una única aplicación, la subcarpeta `src/` del espacio de trabajo contendrá los ficheros de código (lógica de la aplicación, datos y *assets*) de la aplicación raíz.

Para espacios de trabajo de tipo *'multi-project'*, cada proyecto estará en su propia carpeta dentro de la carpeta `projects/` .

Ficheros de configuración

Todos los proyectos dentro del mismo espacio de trabajo o *'workspace'* comparten los ficheros de configuración que están en la raíz del espacio de trabajo. Estos ficheros de configuración tienen **ámbito del espacio de trabajo** y por tanto su configuración afecta a todos los proyectos.

Los archivos de configuración que están en la raíz del espacio de trabajo son los ficheros de configuración de la aplicación raíz. Para un espacio de trabajo de múltiples proyectos, los archivos de configuración específicos de cada proyecto estarán en la carpeta raíz de cada proyecto, dentro de `projects/project-name` .

El fichero `'package.json'` es el fichero estándar de ***NPM*** donde se almacenan las dependencias de terceros que se utilizará para todos los proyectos del espacio de trabajo. Contiene las bibliotecas que necesita la aplicación para ejecutarse tanto en desarrollo como producción:

```
{
  "dependencies": {
    "@angular/core": "^7.2.0"
  },
  "devDependencies": {
    "@angular/cli": "7.2.0"
  }
}
```

Además de las dependencias, el fichero `'package.json'` sirve para indicar información sobre el proyecto como el nombre, versión, nombre del autor, url del repositorio, etc.. y también como contenedor de scripts para automatizar tareas de operaciones rutinarias:

```
{
  "name": "my-app",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "devDependencies": {
    // ...
  }
}
```

El fichero `'tsconfig.json'` contiene los parámetros de configuración por defecto de TypeScript.

El fichero `angular.json` contiene los valores predeterminados de configuración para todos los proyectos en el área de trabajo, incluidas las opciones de configuración para compilar, servir y probar la aplicación.

Los cambios en los ficheros de configuración no se recargan automáticamente. Hay que parar la servidor y volver a lanzarlo para que se carguen.

Carpetas y ficheros de la aplicación

La estructura de la aplicación se distribuye en carpetas.

A nivel raíz de la estructura de la aplicación tenemos las siguientes carpetas y ficheros:

- `src/` : los archivos fuente para el proyecto de la aplicación a nivel raíz.
- `public/` : contiene imágenes y otros archivos servidos de forma estática y que se copiarán tal cual cuando se cree la aplicación.
- `node_modules/` : los paquetes **NPM** instalados para todo el espacio de trabajo
- `package.json` : configura las dependencias NPM
- `angular.json` : configuración CLI para todo el espacio de trabajo
- `tsconfig.json` : la configuración base de TypeScript para los proyectos en el espacio de trabajo.

Y otros ficheros como `package-lock.json` , `.gitignore` , `.editorconfig` o `README.md` .

Dentro de la carpeta `/src` tenemos:

- `src/app` : contiene los componentes Angular que componen la aplicación.
- `index.html` : la página HTML principal que se sirve cuando alguien visita el sitio. Los archivos JavaScript y CSS se agregan automáticamente al crear la aplicación, por lo que normalmente no se necesita agregar ninguna etiqueta `<script>` o `<link>` manualmente.
- `main.ts` : punto de entrada para la aplicación
- `styles.css` : los estilos CSS globales de la aplicación
- `favicon.ico`

Dentro de la carpeta `src/app/` se ubican la lógica y los datos de la aplicación:

- `app.config.ts` : define la configuración de la aplicación que informa a Angular sobre cómo debe construir la aplicación. Sólo se genera cuando se usa la opción `--standalone` .
- `app.component.ts` : define la lógica para el componente raíz de la aplicación, llamado `AppComponent` .
- `app.component.html` : define la plantilla HTML asociada con el componente raíz `AppComponent` . Esta vista se convierte en la raíz de la jerarquía de vistas a medida que agrega componentes y servicios a su aplicación
- `app.component.css` : define la hoja de estilo CSS básica para el componente raíz `AppComponent` .
- `app.module.ts` : define el módulo raíz, llamado `AppModule` , que le dice a Angular cómo ensamblar la aplicación. Inicialmente declara solo el `AppComponent` . A medida que agregue más componentes a la aplicación, deberá declararlos aquí. Sólo se genera cuando se utiliza la opción `--standalone false` .

Components

En el contexto de Angular, un componente es una parte fundamental de la arquitectura de este framework para el desarrollo de aplicaciones web. Angular utiliza una **arquitectura basada en componentes**, lo que significa que la interfaz de usuario se construye mediante la composición de componentes individuales.

Todo componente tiene que tener:

- Una **clase TypeScript** que contiene la lógica y el estado del componente.
- Un **decorador** `@Component` sobre esta clase TypeScript con los metadatos necesarios para la configuración del componente.
- Una **plantilla HTML** que define el marcado que se renderiza en la vista.
- Un **selector CSS** que permite usar el componente en otras plantillas HTML.

Opcionalmente puede incluir una lista de estilos CSS que se aplicarán al componente. Por defecto esos estilos **sólo aplican** al componente donde se definen.

```
@Component({
  selector: 'profile-photo',
  template: `<img src="profile-photo.jpg" alt="Your profile photo">`,
  styles: ` img { border-radius: 50%; } `,
})
export class ProfilePhoto { }
```

Como alternativa, se pueden escribir la plantilla HTML y los estilos CSS en ficheros **separados**:

```
@Component({
  selector: 'profile-photo',
  templateUrl: 'profile-photo.html',
  styleUrl: 'profile-photo.css',
})
export class ProfilePhoto { }
```

Los metadatos `templateUrl` y `styleUrl` son relativos al directorio donde reside el componente.

El objeto que se pasa al decorador `@Component` son los **metadatos** del componente.

Angular crea una instancia del componente para cada elemento HTML coincidente que encuentra. El elemento DOM que coincide con el selector de un componente se denomina elemento **anfitrión o host** de ese componente. El contenido de la plantilla de un componente se representa dentro de su elemento anfitrión.

El DOM representado por un componente, correspondiente a la plantilla de ese componente, se denomina **vista o *'view'*** de ese componente.

Los componentes se generan con:

> `'ng generate component [name] [options]'`

Importing and using components

Angular dispone de **dos formas** de hacer disponible los componentes:

- Mediante los componentes ***'standalone'***, siendo la forma recomendada o
- Mediante `@NgModule`

Un componente ***'standalone'*** es un componente que tiene `standalone: true` en los metadatos del componente.

Este tipo de componentes de pueden importar **de forma directa** en otros componentes.

```
// profilePhoto.ts
@Component({
  standalone: true,
  selector: 'profile-photo',
})
export class ProfilePhoto { }

// userProfile.ts
@Component({
  standalone: true,
  imports: [ProfilePhoto],
    template: `
      <profile-photo />
      <button>Upload a new profile photo</button>`,
})
export class UserProfile { }
```

Para crear un componente ***'standalone'*** se utiliza la opción:

> `'ng generate component [name] --standalone'`.

Selectors

Los selectores se utilizan para **identificar** los componentes en el DOM y son esenciales para el mecanismo de plantillas de Angular.

Un selector en Angular es una **cadena** que especifica el nombre que se usará para insertar un componente en una plantilla. Los selectores son ***"case-sensitive"***.

Además, un único componente puede coincidir exactamente con un selector. Si varios componentes coinciden con un selector, Angular devuelve un error.

Angular realiza esta asociación entre selectores y componentes en **tiempo de compilación**.

Angular dispone de varios tipos de selectores:

- **Selectores de etiqueta ➔** este es el tipo más común. Utiliza el nombre del selector como una etiqueta HTML.

```
@Component({
  selector: 'app-mi-componente',
  templateUrl: './mi-componente.component.html',
  styleUrls: ['./mi-componente.component.css']
})
export class MiComponenteComponent { }
```

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

- **Selectores de atributo ➔** empareja elementos basándose en la presencia de un atributo HTML y, opcionalmente, un valor exacto para ese atributo.

```
@Component({
  selector: '[app-mi-componente]',
  templateUrl: './mi-componente.component.html',
  styleUrls: ['./mi-componente.component.css']
```

```
})
export class MiComponenteComponent { }
```

```
<!-- ... -->
<div app-mi-componente></div>
<!-- ... -->
```

- **Selectores de clase ➔** coincide con elementos según la presencia de una clase CSS.

```
@Component({
  selector: '.app-mi-componente',
  templateUrl: './mi-componente.component.html',
  styleUrls: ['./mi-componente.component.css']
})
export class MiComponenteComponent { }
```

```
<!-- ... -->
<div class="app-mi-componente"></div>
<!-- ... -->
```

Los selectores se pueden **combinar** como por ejemplo un selector que seleccione todos los `<button>` de un tipo `type="reset"`:

```
@Component({
  selector: 'button[type="reset"]',
  ...
})
export class ResetButton { }
```

También se pueden definir selectores múltiples separados por comas:

```
@Component({
  selector: 'drop-zone, [dropzone]',
  ...
})
export class DropZone { }
```

Hay que evitar prefijos como `app-` que pueden generar confusión. También hay que evitar el prefijo `ng` ya que es utilizado por Angular.

Styling

Los componentes pueden incluir estilos CSS que se aplicarán a **todos los elementos** que pueda tener el *'template'* del componente:

```
@Component({
  selector: 'profile-photo',
  template: `<img src="profile-photo.jpg" alt="Your profile photo">`,
  styles: ` img { border-radius: 50%; } `,
})
export class ProfilePhoto { }
```

> Por defecto esos estilos **sólo aplican** al componente donde se definen.

Otro modo es escribir los estilos CSS en un fichero separado y referenciado en el decorador:

```
@Component({
  selector: 'profile-photo',
  templateUrl: 'profile-photo.html',
  styleUrl: 'profile-photo.css',
})
export class ProfilePhoto { }
```

Cada componente se puede configurar como el framework aplica los estilos al componente:

- ***ViewEncapsulation.Emulated*** **(por defecto) ➔** Los estilos están aislados y sólo aplican al *'template'*. Compatible con todos los navegadores
- ***ViewEncapsulation.ShadowDom*** **➔** Utiliza el Shadow DOM para aislamiento completo de estilos. Compatible con navegadores que soporten Shadow DOM.
- ***ViewEncapsulation.None*** **➔** Desactiva la encapsulación de estilos y se vuelve globales.

En la plantilla del componente se puede usar `<link>` para referenciar un fichero CSS externo. Además, dentro del CSS se puede utilizar la regla `@import` para referenciar un fichero CSS externo.

Estos ficheros externos son tratados por Angular como **estilos externos**, por lo que no se ven afectos por el ámbito de aplicación de los estilos.

Accepting data with input properties

Al crear un componente, se pueden marcar propiedades de clase específicas como **vinculables** agregando el decorador `@Input` en la propiedad:

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-card',
  standalone: true,
  template: `
    <div class="card">
      <h2>{{ title }}</h2>
      <p>{{ description }}</p>
    </div>
  `,
  styleUrls: ['./card.component.css']
})
export class CardComponent {
  @Input() title: string = '';
  @Input() description: string = '';
}
```

Este decorador permite que un componente hijo reciba datos que se le pasan desde su componente contenedor:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <h1>My Card Example</h1>
    <app-card [title]="cardTitle" [description]="cardDescription"></app-card>
```

```
  `,
  styleUrls: './card.component.css'
})
export class AppComponent {
  cardTitle = 'Card Title';
  cardDescription = 'This is a description of the card.';
}
```

Angular registra los `@Input` estáticamente en tiempo de compilación. Por lo tanto no se pueden agregar ni eliminar en tiempo de ejecución. Esto significa que una vez que un componente está compilado, sus propiedades `@Input` están fijadas y no pueden ser modificadas dinámicamente.

Al extender una clase de componente, los `@Input` definidos en la clase base son heredados por la clase derivada. Esto permite reutilizar y extender funcionalidades en componentes hijos.

Los nombres utilizados en el decorador `@Input` son *"case-sensitive"*. Por lo tanto, el nombre de la propiedad en el componente debe coincidir exactamente con el nombre utilizado en la plantilla para la vinculación de datos.

Customizing inputs

El decorador `@Input` acepta un objeto de configuración que permite **modificar su comportamiento**.

Se puede especificar la opción `'required'` para exigir que una entrada determinada siempre deba tener un valor. Si no se especifica un valor, Angular reporta un error en tiempo de compilación:

```
@Component({...})
export class CustomSlider {
  @Input({required: true}) value = 0;
}
```

Otra forma de personalización de los *'input'* es mediante **funciones transformadoras** que modifiquen el valor de la propiedad cuando sean asignados por Angular. Este ejemplo, se pone en mayúsculas el título de la tarjeta llamando automáticamente a la función `upperCaseString(...)`:

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-card',
  standalone: true,
  template: `
    <div class="card">
      <h2>{{ title }}</h2>
      <p>{{ description }}</p>
    </div>
  `,
  styleUrls: ['./card.component.css']
})
export class CardComponent {
  @Input({transform: upperCaseString}) title: string = '';
  @Input() description: string = '';
}

function upperCaseString(value: string | undefined) {
  return value?.toUpperCase() ?? '';
}
```

Type checking

Cuando se especifica una transformación, el tipo de parámetro de la función de transformación determina los tipos de valores que se pueden establecer para la entrada en una plantilla.

Por ejemplo, el decorador `@Input` acepta un `number` mientras que la propiedad de la clase es un `string`:

```
@Component({...})
export class CustomSlider {
  @Input({transform: appendPx}) widthPx: string = '';
}

function appendPx(value: number) {
  return `${value}px`;
}
```

Built-in transformations

Angular incluye dos funciones de transformación integradas para los dos escenarios más comunes: **convertir valores a booleanos y números**:

```
import {Component, Input, booleanAttribute, numberAttribute} from '@angular/core';

@Component({...})
export class CustomSlider {
  @Input({transform: booleanAttribute}) disabled = false;
  @Input({transform: numberAttribute}) number = 0;
}
```

Inputs aliases

Se puede especificar un alias opcional para cambiar el nombre de una entrada en las plantillas:

```
@Component({...})
export class CustomSlider {
  @Input({alias: 'sliderValue'}) value = 0;
}
```

```
<custom-slider [sliderValue]="50" />
```

Template syntax

En Angular, una plantilla es un **fragmento de HTML** que utiliza una sintaxis especial para aprovechar muchas de las funcionalidades del framework. Estas plantillas son secciones de HTML que se integran en la página que el navegador muestra, representando la vista o interfaz de usuario.

Casi toda la sintaxis HTML es **válida** en las plantillas de Angular. Sin embargo, Angular extiende el HTML con una sintaxis especial para funcionalidades avanzadas como por ejemplo la interpolación de cadenas con `{{}}`, el uso de directivas `ngIf` o `ngFor`, etcétera…

Cada plantilla en Angular se incluye como parte de la página que muestra el navegador, representando una vista o interfaz de usuario con funcionalidad adicional en comparación con HTML estático.

Como una plantilla de Angular es solo una parte de la página completa, no es necesario incluir elementos como `<html>`, `<body>`, o `<base>`. Solo se necesita el contenido relevante para esa sección de la página.

> ⚠ Angular no soporta la etiqueta `<script>` en las plantillas para evitar riesgos de ataques de tipo *'script injection'*. Para manejar contenido dinámico de manera segura, Angular proporciona herramientas como `DomSanitizer`.

Cuando se genera una aplicación Angular con Angular CLI, el archivo `app.component.html` es la **plantilla predeterminada** y contiene el HTML general que define la estructura inicial de la aplicación.

Text interpolation

La **interpolación** en Angular permite insertar valores y expresiones directamente en el HTML usando las llaves dobles `{{ }}` como delimitadores. Esto significa que se pueden incluir variables, realizar cálculos, e incluso llamar a métodos del componente.

Por ejemplo:

- Se puede mostrar el resultado de una expresión matemática, como `{{ a + b }}`, donde 'a' y 'b' son variables definidas en el componente.
- También se puede invocar métodos del componente para mostrar el valor que devuelven, como `{{ getVal() }}`, donde *'getVal'* es un método del componente que retorna un valor.

Esto permite construir plantillas dinámicas que reflejan los datos y la lógica del componente de manera directa y eficaz:

```
import {Component} from '@angular/core';
import {NgFor} from '@angular/common';
import {CUSTOMERS} from './customers';

@Component({
  standalone: true,
  selector: 'app-root',
  templateUrl: './app.component.html',
  imports: [NgFor],
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  customers = CUSTOMERS;
  currentCustomer = 'Maria';
  title = 'Featured product:';
  itemImageUrl = '../assets/potted-plant.svg';
  getVal(): number {
    return 2;
  }
}
```

La interpolación se utiliza para mostrar el valor de las variables en la plantilla del componente correspondiente:

```
<div>
  <h1>Interpolation and Template Expressions</h1>
  <hr />
  <div>
    <h2>Interpolation</h2>
    <h3>Current customer: {{ currentCustomer }}</h3>
    <p>{{ title }}</p>
    <div><img alt="item" src="{{ itemImageUrl }}"></div>
    <h3>Evaluating template expressions </h3>
    <h4>Simple evaluation (to a string):</h4>
    <!-- "The sum of 1 + 1 is 2" -->
    <p>The sum of 1 + 1 is {{ 1 + 1 }}.</p>
    <h4>Evaluates using a method (also evaluates to a string):</h4>
    <!-- "The sum of 1 + 1 is not 4" -->
```

```
    <p>The sum of 1 + 1 is not {{ 1 + 1 + getVal() }}.</p>
  </div>
  <hr />
<h2>Expression Context</h2>
<div>
  <h4>Template context, template input variables (let customer):</h4>
  <ul>
    @for (customer of customers; track customer) {
      <li>{{ customer.name }}</li>
    }
  </ul>
</div>
```

En Angular, las expresiones dentro de `{{ }}` se evalúan en el contexto del componente y el resultado se inserta en el DOM. Esto permite dinámicamente reflejar el estado de las propiedades del componente en la vista.

Template statements

Las declaraciones de plantilla (***"template statements"***) son métodos o propiedades que se pueden utilizar en el HTML para responder a los eventos del usuario:

```
<button type="button" (click)="deleteHero()">Delete hero</button>
```

Cuando el usuario pulsa en el botón *'Delete hero'*, Angular llama al método `deleteHero()`.

Al igual que la interpolación de texto, estas declaraciones utilizan un lenguaje que parece JavaScript. Sin embargo, hay algunas diferencias con la interpolación de texto. Concretamente, las declaraciones de plantilla soportan el uso de la asignación mediante el signo igual `=` y el encadenado de expresiones con el uso de semicolons `';'`:

```
<button (click)="mostrarMensaje(); contarClicks()">Haz clic</button>
<!-- ... -->
<button (click)="contador = contador + 1; showMessage('Contador:', contador)">Incrementar Contador</button>
```

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = "my-app";
  contador = 0;

  showMessage(message: string, contador: number) {
    console.log(message, contador);
  }
}
```

Las declaraciones tienen un **contexto**: una parte particular de la aplicación a la que pertenece la declaración.

El contexto de la declaración también puede hacer referencia a propiedades del propio contexto de la plantilla. En el siguiente ejemplo, el método de manejo de eventos del componente, `onSave()` toma el objeto `$event` de la plantilla como argumento:

```
<button type="button" (click)="onSave($event)">Save</button>
```

Property binding

La vinculación de propiedades (***"property binding"***) en Angular permite establecer valores para las propiedades de elementos o directivas HTML.

Esta vinculación mueve el valor en una dirección, desde la propiedad de un **componente** a la propiedad del elemento de destino en el **HTML**.

```
@Component({
  ...
})
export class ExampleBindingComponent {
  itemImageUrl = '../assets/phone.svg';
}
```

Para **vincular la propiedad**, se encierra el nombre de la propiedad entre corchetes `'[ ]'`. Si no se encierra entre corchetes, Angular interpreta la cadena de forma literal.

```
<img alt="item" [src]="itemImageUrl">
```

Para desactivar un botón por ejemplo, se puede vincular la propiedad `disabled` del DOM con un booleano en la clase:

```
@Component({
  ...
})
export class ExampleBindingComponent {
  isUnchanged = true;
}
```

```
<!-- Bind button disabled state to `isUnchanged` property -->
<button type="button" [disabled]="isUnchanged">Disabled Button</button>
```

A menudo **se puede elegir entre interpolación o *"property binding"***. Los siguientes ejemplos son equivalentes:

```
<p><img src="{{itemImageUrl}}"> is the <i>interpolated</i> image.</p>
<p><img [src]="itemImageUrl"> is the <i>property bound</i> image.</p>

<p><span>"{{interpolationTitle}}" is the <i>interpolated</i> title.</span></p>
<p>"<span [innerHTML]="propertyTitle"></span>" is the <i>property bound</i> title.</p>
```

Attribute binding

La vinculación de atributos (***"attribute binding"***) en Angular permite establecer valores para los atributos directamente.

La sintaxis de vinculación de atributos se parece a la vinculación de propiedades, pero en lugar de una propiedad de elemento entre corchetes, se antepone el nombre del atributo con el prefijo `attr`, seguido de un punto. Luego, se establece el valor del atributo con una expresión que se resuelve en una cadena:

```
<p [attr.attribute-you-are-targeting]="expression"></p>
```

> Cuando la expresión se resuelve con un `null` o un `undefined`, Angular elimina el atributo.

Uno de los usos más frecuentes es establecer atributos 'ARIA'.

```
<!-- create and set an aria attribute for assistive technology -->
<button type="button" [attr.aria-label]="actionName">{{ actionName }} with Aria</button>
```

Otro uso podría ser establecer el atributo `colspan` de una tabla HTML, lo que permitiría ajustar de forma dinámica una tablas:

```
<!--  expression calculates colspan=2 -->
<tr><td [attr.colspan]="1 + 1">One-Two</td></tr>
```

Class and style binding

La vinculación de clases y estilos permite agregar y eliminar nombres de clases CSS del atributo de clase de un elemento o establecer estilos dinámicamente.

Binding to a single CSS class

Para realizar la vinculación con una **única clase CSS** se utiliza la forma `[class.classname]="expression"` siendo `expression` una expresión que se evalúa como `true` o `false` para indicar si se aplica o no la clase CSS.

La expresión también puede ser una propiedad que se evalúe a `true` o `false`:

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

@Component({
  selector: 'app-other-component',
  standalone: true,
  imports: [],
  template: `
    <p [class.background]="mark" [class.larger]="true">
      other-component works!
    </p>
  `,
  styles: `
    .background {
      background-color: yellow;
    }
    .larger {
      font-size: 2em;
    }
  `
})
export class OtherComponentComponent {
  mark: boolean = true;
}
```

Binding to multiple CSS clasess

Para vincular **más de una clase CSS** se utiliza la forma `[class]="classExpression"` donde `classExpression` puede ser:

- Una cadena de nombres de clases delimitada por espacios como por ejemplo `"my-class-1 my-class-2 my-class-3"`.
- Un objeto con nombres de clases como claves y expresiones verdaderas o falsas como valores.

- Un array de nombres de clase, como por ejemplo `['my-class-1', 'my-class-2']`

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

@Component({
  selector: 'app-other-component',
  standalone: true,
  imports: [],
  template: `
    <p [class]="['background', 'larger']">
      other-component works!
    </p>
  `,
  styles: `
    .background {
      background-color: green;
    }
    .larger {
      font-size: 2em;
    }
  `
})
export class OtherComponentComponent {}
```

Binding to a single style

Para vincular un **único estilo CSS**, se utiliza la forma `[style.styleName]="expression"`. Según el estilo, la expresión será una cadena o numérico, como por ejemplo para indicar un '%' o un tamaño en 'em'.

```
import {Component} from '@angular/core';
@Component({
  standalone: true,
  selector: 'app-nav-bar',
  template: `
    <a [style.text-decoration]="activeLinkStyle">Home Page</a>
    <a [style.text-decoration]="linkStyle">Login</a>
  `,
})
export class NavBarComponent {
  linkStyle = 'underline';
  activeLinkStyle = 'overline';
  /* . . . */
}
```

Binding to multiple styles

Para vincular **múltiples estilos**, se utiliza la forma `[style]="styleExpression"`, donde `styleExpression` será:

- Una lista de cadenas que corresponden a los estilos, como por ejemplo `"width: 100px; height: 100px; background-color: cornflowerblue;"`.
- Un objeto con el nombre del estilo como clave y el valor del estilo como valor, como por ejemplo `{width: '100px', height: '100px', backgroundColor: 'cornflowerblue'}`.

```
import {Component} from '@angular/core';
@Component({
  standalone: true,
  selector: 'app-nav-bar',
  template: ` <nav [style]="navStyle">
    <a [style.text-decoration]="activeLinkStyle">Home Page</a>
    <a [style.text-decoration]="linkStyle">Login</a>
```

```
  </nav>`,
})
export class NavBarComponent {
  navStyle = 'font-size: 1.2rem; color: cornflowerblue;';
  linkStyle = 'underline';
  activeLinkStyle = 'overline';
  /* . . . */
}
```

Event binding

La vinculación de eventos (***"event binding"***) permite escuchar y responder a las acciones del usuario, como pulsaciones de teclas, movimientos del mouse, clics y toques.

Para vincularse a un evento, se utiliza la sintaxis de vinculación de eventos Angular:

```
<button (click)="onSave()">Save</button>
```

La vinculación de eventos "escucha" los eventos de `click` en el botón e invoca el método `onSave()` del componente cuando este evento se produce.

También se pueden realizar vinculaciones de eventos de teclado usando la sintaxis de Angular:

```
<input (keydown.shift.t)="onKeydown($event)" />
```

Los campos `key` y `code` son una parte nativa del objeto de evento del teclado del navegador.

Two-way binding

El enlace bidireccional (***"two-way binding"***) brinda a los componentes de una aplicación una forma de compartir datos. Se utiliza el enlace bidireccional para escuchar eventos y actualizar valores simultáneamente entre los componentes principal y secundario.

Este tipo de vinculación combina el *"property binding"* con el *"event binding"*.

Por tanto la sintaxis del *"two-way binding"* es una mezcla de ambas sintaxis:

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
```

En los formularios, se utiliza la directiva `ngModel` .

Control flow

Las plantillas en Angular admiten bloques de flujo de control que le permiten mostrar, ocultar y repetir elementos condicionalmente.

Esta característica se introdujo en Angular en la **v.17** como experimental y en la **v.18** se convirtió en estable. En versiones previas se utilizan directivas estructurales como **NgIf**, **NgFor** o **NgSwitch**.

Bloques condicionales

El bloque condicional con `@if` muestra el contenido cuando la expresión de condición es verdadera:

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

@Component({
  selector: 'app-jupiter',
  standalone: true,
  imports: [],
  template: `
    @if (showSection) {
      <p>jupiter works!</p>
    }
  `,
  styles: ``
})
export class JupiterComponent {
  showSection = true;
}
```

El bloque `@if` puede tener uno o más bloques `@else` asociados. Inmediatamente después de un bloque `@if`, puede opcionalmente especificar cualquier número de bloques `@else if` y un bloque `@else`:

```
@if (a > b) {
  {{a}} is greater than {{b}}
} @else if (b > a) {
  {{a}} is less than {{b}}
} @else {
  {{a}} is equal to {{b}}
}
```

El nuevo condicional `@if` incorporado admite la referencia de resultados de expresiones para mantener una solución para patrones de codificación comunes:

```
// El resultado del pipe
@if (users$ | async; as users) {
  {{ users.length }}
}
```

En el ejemplo, el resultado del pipe `users$ | async;` se asigna a la variable *'users'*, un patrón común en la evaluación del patrón observable.

Bloques en bucle

La anotación `@for` renderiza repetidamente el contenido de un bloque para cada elemento en una colección. La colección puede ser representada como cualquier iterable de JavaScript, pero existen ventajas de rendimiento al usar un Array regular. Un bucle básico `@for` se ve así:

```
@for (item of items; track item.id) {
  {{ item.name }}
}
```

Utilizar `track` de manera efectiva puede mejorar significativamente el rendimiento de tu aplicación, especialmente en bucles que iteran sobre colecciones de datos.

Para colecciones que no sufren modificaciones (es decir, ningún elemento se mueve, añade o elimina), usar `track $index` es una estrategia eficiente. Esto aprovecha el índice de posición en el array como clave de seguimiento.

Sin embargo, para colecciones con datos mutables o cambios frecuentes, es mejor seleccionar una propiedad que identifique de manera única cada elemento y utilizarla como expresión de seguimiento, como por ejemplo `item.id`.

Dentro de los contenidos `@for`, siempre están disponibles varias variables implícitas:

- **$count**: número de elementos de una colección iterados
- **$index**: índice de la fila actual
- **$first**: si la fila actual es la primera fila
- **$last**: si la fila actual es la última fila
- **$even**: si el índice de fila actual es par
- **$odd**: si el índice de la fila actual es impar

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

@Component({
  selector: 'app-jupiter',
  standalone: true,
  imports: [],
  template: `
    @for (item of numeros; track $index) {
      @if ($odd) {
        <p>{{item}}</p>
      }
    }
  `,
  styles: ``
})
export class JupiterComponent {
  numeros = [1, 2, 3, 4, 5];
}
```

En caso de bloques `@for` anidados, el uso de estas variables puede provocar colisiones. Para ello se puede usar un **alias** con `let`:

```
@for (item of items; track item.id; let idx = $index, e = $even) {
Item #{{ idx }}: {{ item.name }}
}
```

Opcionalmente, puede incluir una sección `@empty` inmediatamente después del contenido del bloque `@for`. El contenido de este bloque se mostrará cuando no hay elementos disponibles:

```
@for (item of items; track item.name) {
  <li> {{ item.name }}</li>
} @empty {
  <li> There are no items.</li>
}
```

Bloques de selección

La sintaxis de `@switch` es muy similar a `@if` y está inspirada en la declaración de `switch` de JavaScript:

```
@switch (condition) {
  @case (caseA) {
    Case A.
  }
  @case (caseB) {
    Case B.
  }
  @default {
    Default case.
  }
}
```

El valor de la expresión condicional se compara con la expresión de caso utilizando el operador `===`.

`@switch` no tiene falla, por lo que no necesita un equivalente a una declaración de `break` o `return`.

El bloque `@default` es opcional y puede ser omitido. Si ningún bloque `@case` coincide con la expresión evaluada y no hay un bloque `@default`, no se mostrará nada.

Pipes

Las ***"pipes"*** se utilizan para transformar cadenas, importes de moneda, fechas y otros datos para su visualización.

Estas *"pipes"* son funciones simples de usar en plantillas que aceptan un valor de entrada y devuelven un valor transformado. Son útiles porque se pueden utilizar en toda la aplicación, declarando cada *"pipe"* sólo una vez.

Angular provee de una serie de *"pipes"* para transformaciones típicas:

- DatePipe
- UpperCasePipe
- LowerCasePipe
- CurrencyPipe
- DecimalPipe
- PercentPipe
- AsyncPipe
- JsonPipe

Para usar una *"pipe"* hay que usar el operador `|` tal y como se muestra en el ejemplo. Además, se tiene que importar del paquete `@angular/common`:

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

@Component({
  standalone: true,
  template: `
    <p>The hero's birthday is {{ birthday | date }}</p>
  `,
  imports: [DatePipe],
})
export class AppComponent {
```

```
    birthday = new Date();
  }
```

Las *"pipes"* pueden tomar parámetros adicionales que permitan configurar la transformación. Estos parámetros pueden ser **obligatorios** u **opcionales**:

```
<p>The hero's birthday is in {{ birthday | date:'yyyy' }}</p>
```

Algunas *"pipes"* pueden tomar múltiples parámetros. Para ello se utiliza el operador `':'`:

```
<p>The current time is: {{ currentTime | date:'hh:mm':'UTC' }}</p>
```

Por último, las *"pipes"* se pueden **encadenar** de forma que la salida de la *"pipe"* anterior es la entrada de la siguiente:

```
<p>The hero's birthday is {{ birthday | date }}</p>
<p>The hero's birthday is in {{ birthday | date:'yyyy' | uppercase }}</p>
```

Understanding template variables

Las variables de plantilla en Angular son una forma de hacer referencia a elementos de la plantilla o a directivas en cualquier otra parte dentro de esa plantilla. Son útiles para acceder a propiedades y métodos de estos elementos desde el contexto de la plantilla.

Se puede hacer referencia a:

- **un elemento DOM dentro de la plantilla**
- **una directiva o componente**
- **un TemplateRef de una ng-template**
- **un web component**

Una variable de plantilla se define en Angular utilizando la sintaxis `#nombreVariable` en el HTML:

```
<form #myForm="ngForm">
  <input #phone placeholder="phone number" />
  <!-- ... -->

  <!-- phone refers to the input element; pass its `value` to an event handler -->
  <button type="button" (click)="callPhone(phone.value)">Call</button>

  <app-child #childComponent></app-child>
  <button (click)="childComponent.doSomething()">Do Something</button>
</form>
```

En este ejemplo, `#myForm` es una variable de plantilla que hace referencia a la instancia del formulario asociado con la directiva `NgForm`, la variable `#phone` hace referencia al campo `<input>` y la variable `#childComponent` se refiere al componente hijo, pudiendo llamar a un método de ese componente.

Por tanto, lo que permite las variables de plantilla es **acceder a cualquier propiedad o método que tenga el elemento** al que hace referencia, ya sea una etiqueta HTML estándar, otra directiva (como `NgForm`), etcétera…

Si la variable especifica un nombre en el lado derecho, como `#var="ngModel"`, la variable se refiere a la directiva o componente del elemento con un nombre `exportAs` coincidente.

Las variables de plantilla **solo están disponibles en el contexto en el que se definen**. No se puede acceder a ellas fuera de su alcance (por ejemplo, en otros componentes o en la lógica del componente TypeScript).

Directives

Las directivas son clases que agregan **comportamiento adicional** a los elementos de una aplicación Angular.

Angular dispone de tres tipos de directivas:

- ***"Components"***: los componentes en Angular son una forma especial de directiva con una plantilla asociada.
- ***"Attributes directives"***: cambiar la apariencia o el comportamiento de un elemento, componente u otra directiva.
- ***"Structural directives"***: cambiar el diseño DOM agregando y eliminando elementos DOM.

Built-in attribute directives

Las directivas de atributos más comunes son **NgClass**, **NgStyle** y **NgModel**.

NgClass

Con la directiva `NgClass` se pueden añadir o eliminar **múltiples clases CSS** de forma simultánea. Para añadir o eliminar una única clase CSS es mejor utilizar el *"class binding"*.

En el siguiente ejemplo se muestra el uso de `NgClass` y un *"class binding"* para una única clase CSS:

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

@Component({
  selector: 'app-moon',
  standalone: true,
  imports: [NgClass],
  template: `
    <p [ngClass]="isLarger ? 'larger' : 'shorter'" [class.background]="withBackground">
      moon works!
    </p>
  `,
  styles: `
    .background {
      background-color: green;
    }
    .larger {
      font-size: 2em;
    }
    .shorter {
      font-size: 1em;
    }
  `
})
export class MoonComponent {
  withBackground = true;
  isLarger = true;
}
```

NgStyle

Con la directiva `NgStyle` se pueden aplicar estilos *inline* a un elemento:

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

@Component({
  selector: 'app-mars',
  standalone: true,
  imports: [NgStyle],
  template: `
    <p [ngStyle]="currentStyles">
      mars works!
    </p>
  `,
  styles: ``
})
export class MarsComponent {
  canSave = true;
  isUnchanged = true;
  isSpecial = true;

  currentStyles = {
    'font-style': this.canSave ? 'italic' : 'normal',
    'font-weight': !this.isUnchanged ? 'bold' : 'normal',
    'font-size': this.isSpecial ? '24px' : '12px',
    'background-color': 'red'
  };
}
```

NgModel

Utilice la directiva `NgModel` para mostrar una propiedad de datos y actualizar esa propiedad cuando el usuario realice cambios:

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

@Component({
  selector: 'app-venus',
  standalone: true,
  imports: [FormsModule],
  template: `
    <input [(ngModel)]="nombre" placeholder="Ingrese su nombre">
    <p>Tu nombre ingresado es: {{ nombre }}</p>
  `,
  styles: ``
})
export class VenusComponent {
  nombre: string = ''; // Variable para almacenar el nombre ingresado
}
```

Para personalizar su configuración, escriba el formulario ampliado, que separa el enlace de propiedad para establecer el valor y evento para responder a los cambios:

```
<input [ngModel]="currentItem.name" (ngModelChange)="setUppercaseName($event)" id="example-uppercase">
```

Structural directives

Las directivas estructurales son responsables del **diseño HTML**. Dan forma o remodelan la estructura del DOM, generalmente agregando, eliminando y manipulando los elementos *host* a los que están adjuntos.

Using NgIf

Para agregar o eliminar un elemento, vincule la directiva `NgIf` a una expresión de condición, es decir, que se evalúe a verdadero o falso:

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

@Component({
  selector: 'app-ejemplo-ngif',
  template: `
    <div *ngIf="mostrarElemento">
      Este elemento se muestra si mostrarElemento es true.
    </div>
    <button (click)="toggleElemento()">Toggle Elemento</button>
  `
})
export class EjemploNgIfComponent {
  mostrarElemento: boolean = true;

  toggleElemento() {
    this.mostrarElemento = !this.mostrarElemento;
  }
}
```

Si se aplica a un componente, cuando la expresión es verdadera, Angular añade el componente al DOM. Sin embargo, cuando es falsa, suprime el componente del DOM:

```
<app-item-detail *ngIf="isActive" [item]="item"></app-item-detail>
```

De forma predeterminada, la directiva `NgIf` impide la visualización de un elemento vinculado a un **valor nulo**.

Using NgFor

La directiva `NgFor` en Angular se utiliza para **iterar sobre elementos en una colección**, como un array o una lista, y renderizar dinámicamente elementos HTML basados en cada elemento de la colección:

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

@Component({
  selector: 'app-jupiter',
  standalone: true,
  imports: [NgFor],
  template: `
    <ul>
      <li *ngFor="let num of numeros; let i = index">
        Índice: {{ i + 1 }} - Número: {{ num }}
      </li>
    </ul>
  `,
  styles: ``
})
export class JupiterComponent {
  numeros = [2, 4, 6, 8, 10, 12];
}
```

Para repetir un elemento componente, aplique `*ngFor` al selector:

```
<app-item-detail *ngFor="let item of items" [item]="item"></app-item-detail>
```

Using NgSwitch

La directiva `NgSwitch` muestra **un elemento entre varios elementos posibles**, según una condición de cambio. Angular coloca solo el elemento seleccionado en el DOM.

Esta directiva se compone de tres directivas que deberán ser importadas: `NgSwitch`, `NgSwitchCase` and `NgSwitchDefault`.

```
<div [ngSwitch]="currentItem.feature">
  <app-stout-item    *ngSwitchCase="'stout'"    [item]="currentItem"></app-stout-item>
  <app-device-item   *ngSwitchCase="'slim'"     [item]="currentItem"></app-device-item>
  <app-lost-item     *ngSwitchCase="'vintage'"  [item]="currentItem"></app-lost-item>
  <app-best-item     *ngSwitchCase="'bright'"   [item]="currentItem"></app-best-item>
...
  <app-unknown-item  *ngSwitchDefault           [item]="currentItem"></app-unknown-item>
</div>
```

Hosting a directive without a DOM element

La directiva de Angular `<ng-container>` es un elemento de agrupación que no interfiere con los estilos o el diseño porque Angular no lo añade en el DOM.

Es decir, `<ng-container>` es útil cuando se necesita aplicar directivas estructurales (`NgIf`, `NgFor`) a múltiples elementos sin introducir un nodo extra en el DOM final.

Por ejemplo, en el caso de la directiva `NgIf`, si queremos condicionar varias `<p>` con la misma variable, podríamos envolverlas con un `<div>` para no repetir `NgIf` en cada una de ellas:

```
<div *ngIf="showMessage">
  <p>This is the first message.</p>
  <p>This is the second message.</p>
</div>
```

Sin embargo, esta solución introduce un `<div>` innecesario en el DOM. Usando `<ng-container>` podemos agrupar estos elementos sin agregar etiquetas extra al DOM:

```
<ng-container *ngIf="showMessage">
  <p>This is a conditional message.</p>
  <p>This is the second message.</p>
</ng-container>
```

Otro ejemplo con la directiva `*ngFor`:

```
<ul>
  <ng-container *ngFor="let num of numbers">
    <li *ngIf="num % 2 === 0">
      {{ num }}
    </li>
  </ng-container>
</ul>
```

Aquí, iteramos sobre una lista de números y solo mostramos los pares, sin agregar un contenedor innecesario alrededor de los `<li>`.

Dependency injection in Angular

Cuando se desarrolla una parte más pequeña del sistema, como un módulo o una clase, puede ser necesario utilizar funcionalidades de otras clases. Por ejemplo, es posible que se necesite un servicio HTTP para realizar llamadas al backend. La **inyección de dependencias (DI)**, es un patrón de diseño y un mecanismo para crear y proporcionar algunas partes de una aplicación a otras partes que las necesiten. Angular soporta este patrón de diseño y se puede utilizar en las aplicaciones para aumentar la flexibilidad y la modularidad.

En Angular, las dependencias suelen ser **servicios**, aunque también pueden ser valores como cadenas o funciones. Un inyector para una aplicación (creado automáticamente durante el arranque) instancia las dependencias cuando son necesarias, utilizando un proveedor configurado del servicio o valor.

Understanding dependency injection

La inyección de dependencias (DI), es uno de los conceptos fundamentales en Angular. La DI está integrada en el marco de Angular y permite a las clases con decoradores de Angular, como **Componentes**, **Directivas**, **Pipes** e **Inyectables**, configurar las dependencias que necesitan.

En el sistema de DI existen dos roles principales: el **consumidor** de dependencias y el **proveedor** de dependencias.

Angular facilita la interacción entre los consumidores de dependencias y los proveedores de dependencias utilizando una abstracción llamada *"Injector"*. Cuando se solicita una dependencia, el injector verifica su registro para ver si ya hay una instancia disponible allí. Si no la hay, se crea una nueva instancia y se almacena en el registro.

Angular crea un injector de aplicación a nivel global (también conocido como *"root injector"*) durante el proceso de inicio de la aplicación. En la mayoría de los casos, no es necesario crear manualmente injectores, pero es importante saber que existe una capa que conecta proveedores y consumidores.

Providing a dependency

El primer paso es agregar el decorador `@Injectable` para mostrar que la clase se puede inyectar.

```
@Injectable()
class HeroService {}
```

El siguiente paso es hacerlo disponible en la inyección de dependencias (DI) proporcionándolo. Una dependencia puede ser proporcionada en varios lugares, siendo la recomendable mediante el metadato *"providedIn"* en el decorador `@Injectable`:

```
@Injectable({
  providedIn: 'root'
})
class HeroService {}
```

Proporcionar un servicio a **nivel de raíz (*"root"*)** de la aplicación utilizando *"providedIn"* permite inyectar el servicio en todas las demás clases. Esto permite a Angular y a los optimizadores de código JavaScript eliminar de manera efectiva los servicios que no se utilizan.

Cuando se proporciona el servicio a nivel de raíz, Angular crea **una única instancia compartida del servicio** y lo inyecta en cualquier clase que lo solicite.

Otra forma es a **nivel de componente** en el decorador `@Component` usando el metadato *"providers"*. En este caso, el servicio estará disponible para todas las instancias de este componente y para otros componentes y directivas utilizados en la plantilla:

```
@Component({
  standalone: true,
  selector: 'hero-list',
  template: '...',
  providers: [HeroService]
})
class HeroListComponent {}
```

Cuando se registra un proveedor a nivel de componente, se obtiene una nueva instancia del servicio con cada nueva instancia de ese componente.

El inconveniente de esta forma es que declarar un servicio de esta manera hace que siempre esté incluido en la aplicación, incluso si el servicio no se utiliza.

Injecting/consuming a dependency

La forma más común de inyectar una dependencia es declararla en el constructor de una clase. Cuando Angular crea una nueva instancia de una clase de componente, directiva o *pipe*, determina qué servicios u otras dependencias necesita esa clase observando los tipos de parámetros del constructor.

```
@Component({ … })
class HeroListComponent {
  constructor(private service: HeroService) {}
}
```

Esta es la forma recomendable ya que al inyectar dependencias en el constructor, Angular puede gestionar el ciclo de vida y las instancias de los servicios de manera más eficaz. Esto también facilita la sustitución de dependencias durante **las pruebas unitarias** mediante el uso de *mocks* o *stubs*.

Otra opción es utilizar el método `inject()`:

```
@Component({ … })
class HeroListComponent {
  private service = inject(HeroService);
}
```

Signals

Las *"Signals"* en Angular es un sistema que rastrea de forma granular cómo y dónde se usa su estado en una aplicación, lo que permite que el framework optimice las actualizaciones de renderizado.

Una señal es un envoltorio o *"wrapper"* alrededor de un valor que **notifica a los consumidores interesados cuando ese valor cambia**. Las señales pueden contener cualquier valor, desde primitivos hasta estructuras de datos complejas.

El valor de una señal se lee llamando a su función *"getter"*, que permite a Angular rastrear dónde se utiliza la señal.

Las señales pueden ser de escritura (*"writable"*) o de sólo lectura (*"read-only"*).

Writable signals

Estas señales proporcionan una API para actualizar sus valores directamente. Para crear una señal modificable (*"writable signal"*) se llama a la función con el valor inicial de la señal:

```
const count = signal(0);
// Signals are getter functions - calling them reads their value.
console.log('The count is: ' + count());
```

Para modificar el valor, se utiliza la función `.set()`:

```
count.set(3);
```

O para tener en cuenta el valor anterior de la señal, se utiliza la función `.update()`:

```
// Increment the count by 1.
count.update(value => value + 1);
```

Computed Signals

Las señales calculadas (*"computed signals"*) son señales de solo lectura que derivan su valor de otras señales. Las señales calculadas se definen utilizando la función `computed()`:

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```

En el ejemplo, la señal `doubleCount` depende de la señal `count`. Cada vez que se actualiza `count`, Angular sabe que `doubleCount` también debe actualizarse.

Las señales calculadas **no son señales modificables**, es decir, no se puede modificar su valor directamente. Un intento de modificar una señal calculada produce un **error de compilación**:

```
doubleCount.set(3); // Error de compilación
```

Effects

Las señales son útiles porque notifican a los consumidores interesados cuando cambian. Un *"effect"* es una operación que se ejecuta cada vez que uno o más valores de señal cambian. Se puede crear un *"effect"* con la función correspondiente:

```
effect(() => {
  console.log(`The current count is: ${count()}`);
});
```

Los *"effect"* siempre se ejecutan **al menos una vez**. Cuando se ejecuta, rastrea cualquier valor de señal leído. Siempre que cualquiera de estos valores de señal cambie, el *"effect"* se ejecutará nuevamente. De manera similar a las señales calculadas, los efectos realizan un seguimiento de sus dependencias dinámicamente y solo rastrean las señales que se leyeron en la ejecución más reciente.

Los *"effect"* siempre se ejecutan de **forma asincrónica**, durante el proceso de detección de cambios.

De forma predeterminada, solo se puede crear un `effect()` dentro de un contexto de inyección (donde se tiene acceso a la función de inyección). La forma más sencilla de satisfacer este requisito es dentro de un componente, directiva o constructor de servicios.

En este ejemplo, se utiliza un *"effect"* para suscribirse a una señal desde otro componente. Cómo nexo de unión se utiliza un servicio entre ambos componentes:

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

@Injectable({
  providedIn: 'root'
})
export class ContadorService {
  contador = signal(0);

  incrementar() {
    this.contador.set(this.contador() + 1);
  }
}
```

Este componente definirá la lógica para cambiar el valor de la señal:

```
import { Component } from '@angular/core';
import { ContadorService } from './contador.service';

@Component({
  selector: 'app-contador',
  template: `
    <div>
      <p>Contador: {{ contadorService.contador() }}</p>
      <button (click)="incrementar()">Incrementar</button>
    </div>
  `
})
export class ContadorComponent {
  constructor(public contadorService: ContadorService) {}

  incrementar() {
    this.contadorService.incrementar();
  }
}
```

Por último, en este componente se inyecta el servicio y se crea el *"effect"* dentro del contexto de inyeccción para suscribirse a la señal:

```
import { Component } from '@angular/core';
import { ContadorService } from './contador.service';
import { effect } from '@angular/core';

@Component({
  selector: 'app-logger',
  template: `
    <div>
      <p>Logger: Mira la consola para los cambios en el contador</p>
    </div>
  `
})
export class LoggerComponent {
  constructor(private contadorService: ContadorService) {
    effect(() => {
      console.log(`El valor del contador es: ${this.contadorService.contador()}`);
```

```
    });
  }
}
```

Un *"effect"* **se destruye automáticamente cuando se destruye el contexto que lo contiene**. Esto significa que los efectos creados dentro de los componentes se destruyen cuando se destruye el componente. Lo mismo ocurre con los efectos dentro de directivas, servicios, etcétera…

Routing

El enrutamiento ayuda a cambiar lo que ve el usuario en **una aplicación de una sola página**.

En una aplicación de una sola página, se cambia lo que ve el usuario mostrando u ocultando partes de la pantalla que corresponden a componentes particulares, en lugar de ir al servidor para obtener una nueva página.

A medida que los usuarios realizan tareas en la aplicación, deben poder moverse entre las diferentes vistas que se hayan definido.

Para manejar la navegación de una vista a la siguiente, se utiliza el `Router` en Angular. El enrutador permite la navegación interpretando la URL del navegador como una instrucción para cambiar la vista.

Defining a basic route

Para utilizar el enrutador Angular, una aplicación debe tener al menos **dos componentes** para poder navegar de uno a otro.

Los componentes que se vayan a utilizar en el enrutador, se agregan al fichero `app.routes.ts` y al array `Routes[]`. Si se ha utilizado la herramienta de línea de comandos de Angular, este fichero ya estará creado y también se habrá generado un array `Routes[]` vacío:

```
import { Routes } from '@angular/router';
import { FirstComponent } from './routerExample/first/first.component';
import { SecondComponent } from './routerExample/second/second.component';

export const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent}
];
```

El enrutador y las rutas se deben importar en `app.config.ts`. Las rutas deben añadirse en la función `provideRouter()` y a su vez esta función añadirse a `providers[...]`. Sin embargo, si se ha utilizado la CLI de Angular para generar la aplicación, por defecto ya se habrá creado este fichero y se habrá configurado con el enrutador y las rutas:

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

import { routes } from './app.routes';
import { provideAnimationsAsync } from '@angular/platform-browser/animations/async';

export const appConfig: ApplicationConfig = {
  providers: [provideZoneChangeDetection({ eventCoalescing: true }), provideRouter(routes), provideAnimationsAsync()]
};
```

Ahora que se han definido sus rutas, hay que agregarlas a la aplicación mediante etiquetas `<a>` por ejemplo.

Además, se utiliza `routerLink`. Es una directiva que se utiliza para **enlazar rutas** en aplicaciones Angular. Es similar a un enlace HTML (`<a href="...">`), pero en lugar de recargar la página, Angular maneja la navegación internamente.

También se utiliza `routerLinkActive`. Es una directiva que se utiliza para **aplicar clases CSS** a un enlace cuando la ruta asociada está activa. Esto es útil para indicar visualmente al usuario en qué página se encuentra actualmente.

A continuación, actualice la plantilla de su componente para incluir `<router-outlet>`. Este elemento informa a Angular que actualice la vista de la aplicación con el componente para la ruta seleccionada.

```
<h1>Angular Router App</h1>
<nav>
  <ul>
    <li><a routerLink="/first-component" routerLinkActive="active" ariaCurrentWhenActive="page">First Component</a></li
    <li><a routerLink="/second-component" routerLinkActive="active" ariaCurrentWhenActive="page">Second Component</a></l
  </ul>
</nav>
<!-- The routed views render in the <router-outlet>-->
<router-outlet></router-outlet>
```

También se debe agregar `RouterLink`, `RouterLinkActive` y `RouterOutlet` a la matriz de importaciones de `AppComponent`:

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, RouterLink, RouterLinkActive],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'routing-app';
}
```

El **orden de las rutas** es crucial porque el `Router` utiliza una estrategia de "la primera coincidencia gana" al hacer coincidir las rutas. Por lo tanto, las rutas más específicas deben colocarse antes que las rutas menos específicas:

- Primero las rutas con un camino estático,
- Seguidas por una ruta de camino vacío, que coincide con la ruta predeterminada.
- La ruta comodín viene al final porque coincide con cualquier URL y el `Router` la selecciona solo si ninguna otra ruta coincide primero.

```
const routes: Routes = [
  { path: 'detalle/:id', component: DetalleComponent },  // Ruta específica con parámetro
  { path: 'lista', component: ListaComponent },          // Ruta específica estática
  { path: '', redirectTo: '/home', pathMatch: 'full' },  // Ruta de camino vacío (ruta predeterminada)
  { path: '**', component: PaginaNoEncontradaComponent } // Ruta comodín (ruta de error 404)
];
```

Getting route information

A menudo, cuando un usuario navega por la aplicación, desea pasar información de un componente a otro. Por ejemplo, desde una lista de elementos cuando se accede a la edición o al detalle de un elemento concreto. En esa página de detalle se necesita saber que elemento (mediante un 'id', por ejemplo) ha seleccionado el usuario.

Se agrega la característica `withComponentInputBinding` a la función `provideRouter()`:

```
providers: [
  provideRouter(appRoutes, withComponentInputBinding()),
]
```

En el componente, se añade el decorador `@Input()` con el nombre que coincida con el parámetro:

```
@Component({
  selector: 'app-second',
  standalone: true,
  imports: [],
  templateUrl: './second.component.html',
  styleUrl: './second.component.css'
})
export class SecondComponent {
  @Input()
  set id(id: string) {
    console.log(`Identificador: ${id}`);
  }
}
```

Se añade el parámetro en las rutas del `app.routes.ts`:

```
import { Routes } from '@angular/router';
import { FirstComponent } from './routerExample/first/first.component';
import { SecondComponent } from './routerExample/second/second.component';

export const routes: Routes = [
  { path: 'first-component', component: FirstComponent, title: 'My First Component' },
  { path: 'second-component/:id', component: SecondComponent, title: 'My Second Component'}
];
```

Por último, se añade el parámetro al `routerLink` por interpolación por ejemplo o por cualquier otro sistema.

```
<nav>
  <ul>
    <li><a routerLink="/first-component" routerLinkActive="active" ariaCurrentWhenActive="page">First Component</a></li
    <li><a routerLink="/second-component/{{id}}" routerLinkActive="active" ariaCurrentWhenActive="page">Second Component
  </ul>
</nav>
```

Setting up wildcard routes

Una aplicación que funcione bien debería manejar todos los escenarios, como por ejemplo cuando los usuarios intentan navegar a una parte de la aplicación que no existe.

Para agregar esta funcionalidad a la aplicación, se configura una **ruta comodín**. El enrutador Angular selecciona esta ruta cada vez que la URL solicitada no coincide con ninguna ruta del enrutador.

```
const routes: Routes = [
  { path: 'detalle/:id', component: DetalleComponent },  // Ruta específica con parámetro
  { path: 'lista', component: ListaComponent },          // Ruta específica estática
  { path: '', redirectTo: '/home', pathMatch: 'full' },  // Ruta de camino vacío (ruta predeterminada)
  { path: '**', component: PaginaNoEncontradaComponent } // Ruta comodín (ruta de error 404)
];
```

Los dos asteriscos, `**`, indican a Angular que esta definición de ruta es una ruta comodín. La ruta comodín es la **última ruta** porque coincide con cualquier URL.

Para la propiedad del componente, se puede definir cualquier componente en la aplicación. Las opciones comunes incluyen un componente específico de la aplicación, cuyo objetivo sea mostrar **una página 404** a los usuarios; o una **redirección al componente principal** de su aplicación.

Setting up redirects

En Angular, el concepto de redirección se utiliza para redirigir automáticamente a una ruta diferente cuando se accede a una ruta específica. Para ello se utiliza `redirectTo` para indicar la redirección y `pathMatch: 'full'` para indicar que la ruta debe coincidir exactamente con el camino especificado o `pathMatch: 'prefix'` cuando cualquier prefijo del camino puede coincidir.

La redirección puede ser útil en varios escenarios:

- Cuando los usuarios acceden a la raíz de la aplicación (`/`), pueden ser redirigidos a una ruta predeterminada, como una página de inicio o un dashboard.

```
const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' }
];
```

- Redirigir usuarios no autenticados a una página de inicio de sesión cuando intentan acceder a rutas protegidas:

```
const routes: Routes = [
  { path: 'protected', canActivate: [AuthGuard], component: ProtectedComponent },
  { path: 'login', component: LoginComponent },
  { path: '', redirectTo: '/login', pathMatch: 'full' }
];
```

- Si una ruta antigua ha cambiado para redirigir a los usuarios a la nueva ruta con parámetros actualizados:

```
const routes: Routes = [
  { path: 'old-path/:id', redirectTo: '/new-path/:id', pathMatch: 'full' }
];
```

Forms in Angular

La gestión de formularios es esencial para procesar la entrada del usuario en la mayoría de las aplicaciones.

Angular proporciona dos enfoques diferentes para manejar la entrada del usuario a través de formularios: **reactivo** y **basado en plantillas**. Ambos capturan eventos de entrada del usuario desde la vista, validan la entrada del usuario, crean un modelo de formulario y un modelo de datos para actualizar y proporcionan una forma de realizar un seguimiento de los cambios.

Los **formularios reactivos** proporcionan acceso directo y explícito al modelo de objetos del formulario subyacente. Son más sólidos, más escalables, reutilizables y testeables. Si los formularios son una parte clave de la aplicación, o si ya se está utilizando patrones reactivos para crear la aplicación, los formularios reactivos son la mejor opción.

Además, utilizan un **flujo de datos síncrono** entre la vista y el modelo de datos, lo que facilita la creación de formularios a gran escala. Los formularios reactivos requieren menos configuración para las pruebas, y las pruebas no requieren una comprensión profunda de la detección de cambios para probar adecuadamente las actualizaciones y validación de los formularios.

Los **formularios basados en plantillas** son útiles para agregar un formulario simple a una aplicación, como un formulario de registro en una lista de correo electrónico. Son fáciles de agregar a una aplicación, pero no escalan tan bien como los formularios reactivos. Si tiene requisitos de formulario muy básicos y una lógica que se puede administrar únicamente en la plantilla, los formularios basados en plantillas podrían ser una buena opción.

Además, utilizan un **flujo de datos asíncrono** entre la vista y el modelo de datos. La abstracción de formularios basados en plantillas también afecta las pruebas. Las pruebas dependen en gran medida de la ejecución manual de detección de cambios para ejecutarse correctamente y requieren más configuración.

Tanto los formularios reactivos como los basados en plantillas se basan en las siguientes clases base ***FormControl***, ***FormGroup***, ***FormArray*** y ***ControlValueAccessor***.

Reactive forms

Adding a basic form control

Para usar controles de formulario reactivos, importar `ReactiveFormsModule` desde el paquete `@angular/forms` en el componente y se añade a `imports[...]` :

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

@Component({
  selector: 'app-reactive-forms',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './reactive-forms.component.html',
  styleUrl: './reactive-forms.component.css'
})
export class ReactiveFormsComponent { }
```

Importar `FormControl` desde el paquete `@angular/forms` :

```
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-reactive-forms',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './reactive-forms.component.html',
  styleUrl: './reactive-forms.component.css'
})
export class ReactiveFormsComponent {

  // Sets initial value
  name = new FormControl('');

}
```

Se utiliza el constructor de `FormControl` para establecer un valor inicial, que en el ejemplo sería una cadena vacía.

Al crear estos controles en la clase del componente, se obtiene acceso inmediato para escuchar, actualizar y validar el estado de la entrada del formulario.

Después de crear el control en la clase del componente, debes asociarlo con un elemento de control del formulario en la plantilla utilizando la vinculación `[formControl]` proporcionada por `FormControlDirective` , que también está incluida en el `ReactiveFormsModule` .

```
<label for="name">Name: </label>
<input id="name" type="text" [formControl]="name">
```

Usando la sintaxis de enlace en la plantilla, el control de formulario ahora está registrado en el elemento de entrada `name` en la plantilla. El control de formulario y el elemento DOM se comunican entre sí: la vista refleja los cambios en el modelo, y el modelo refleja los cambios en la vista.

Puede mostrar el valor de las siguientes maneras:

- A través del observable `valueChanges`, puedes escuchar los cambios en el valor del formulario en la plantilla utilizando `AsyncPipe` o en la clase del componente utilizando el método `subscribe()`.
- Con la propiedad `value`, que le brinda una instantánea del valor actual

```
<label for="name">Name: </label>
<input id="name" type="text" [formControl]="name">

<p>Value: {{ name.value }}</p>
```

Replacing a form control value

Los formularios reactivos tienen **métodos para cambiar el valor de un control programáticamente**, lo cual te proporciona la flexibilidad de actualizar el valor sin interacción del usuario.

Una instancia de control de formulario proporciona un método `setValue()` que actualiza el valor del control de formulario y valida la estructura del valor proporcionado con respecto a la estructura del control.

```
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-reactive-forms',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './reactive-forms.component.html',
  styleUrl: './reactive-forms.component.css'
})
export class ReactiveFormsComponent {

  // Sets initial value
  name = new FormControl('');

  updateName() {
    // Update value
    this.name.setValue('Nancy');
  }

}
```

En la plantilla, un botón simulará una actualización del valor. Al hacer clic en el botón, el valor ingresado en el elemento de control del formulario se refleja como su valor actual.

```
<label for="name">Name: </label>
<input id="name" type="text" [formControl]="name">

<p>Value: {{ name.value }}</p>
```

```
<button type="button" (click)="updateName()">Update Name</button>
```

Grouping form controls

Los formularios reactivos proporcionan **dos formas de agrupar múltiples controles** relacionados en un único formulario de entrada:

- **Form group**: define un formulario con un conjunto fijo de controles que pueden administrarse juntos.
- **Form array**: define un formulario dinámico, donde se pueden agregar y eliminar controles en tiempo de ejecución.

Así como una instancia de `FormControl` da control sobre un solo campo de entrada, una instancia de `FormGroup` rastrea el estado del formulario de un grupo de instancias de `FormControl` (por ejemplo, un formulario). Cada control en una instancia de `FormGroup` es **rastreado por nombre** al crear el `FormGroup`.

Lo primero es importar las clases `FormGroup` y `FormControl` de `@angular/forms`:

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

Lo siguiente es crear una propiedad en la clase del componente y establecer en la propiedad una nueva instancia de `FormGroup`. Para inicializar el `FormGroup`, se proporciona al constructor un objeto de claves con el mismo nombre asignado al control en la plantilla. Cada una de estas claves es un `FormControl` o tal vez otro `FormGroup`.

Además, en este constructor se puede pasar un `Validators` o un array de `Validators` con las validaciones a aplicar en ese control, como por ejemplo `Validators.required`.

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

@Component({
  selector: 'app-form-group-forms',
  standalone: true,
  imports: [ReactiveFormsModule, JsonPipe],
  templateUrl: './form-group-forms.component.html',
  styleUrl: './form-group-forms.component.css'
})
export class FormGroupFormsComponent {

  profileForm = new FormGroup({
    firstName: new FormControl('', Validators.required),
    lastName: new FormControl('', Validators.required),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl(''),
    })
  });

  onSubmit() {
    console.log(this.profileForm.value);
  }

}
```

En la plantilla, los controles tienen el nombre de las claves en el atributo `formControlName` y el formulario tiene el nombre de la propiedad definida en el componente como en el ejemplo `<form [formGroup]="profileForm" ...>`. Además, si hay grupos anidados, el nombre del grupo anidado se indica en `formGroupName`.

```
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">
  <label for="first-name">First Name: </label>
  <input id="first-name" type="text" formControlName="firstName">
  <label for="last-name">Last Name: </label>
  <input id="last-name" type="text" formControlName="lastName">

  <div formGroupName="address">
    <h2>Address</h2>
    <label for="street">Street: </label>
    <input id="street" type="text" formControlName="street">
    <label for="city">City: </label>
    <input id="city" type="text" formControlName="city">
    <label for="state">State: </label>
    <input id="state" type="text" formControlName="state">
    <label for="zip">Zip Code: </label>
    <input id="zip" type="text" formControlName="zip">
  </div>

  <p>Complete the form to enable button.</p>
  <button type="submit" [disabled]="!profileForm.valid">Submit</button>

  <p>Form Status: {{ profileForm.status }}</p>
  <p>Form Values: {{ profileForm.value | json }}</p>
</form>
```

Un `FormGroup` rastrea el estado y los cambios de cada uno de sus controles, por lo que si uno de los controles cambia, el control principal también emite un nuevo estado o cambio de valor.

La directiva `FormGroup` escucha el evento `submit` emitido por un elemento de formulario como un `button` y emite un evento `ngSubmit` que se puede vincular a una función *"callback"* de devolución de llamada en el componente como `onSubmit() { ... }`

```
<button type="submit" [disabled]="!profileForm.valid">Submit</button>
```

Template-driven forms

Los formularios basados en plantillas o *"Template-drive forms"* permiten utilizar directivas específicas de formulario en una plantilla Angular.

Los formularios basados en plantillas utilizan enlace de datos bidireccional o *"two-way data binding"* para actualizar el modelo de datos en el componente a medida que se realizan cambios en la plantilla y viceversa.

Los formularios basados en plantillas se basan en directivas definidas en `FormsModule`.

- **NgModel**: armoniza los cambios de valor en el elemento del formulario adjunto con los cambios en el modelo de datos, permitiéndote responder a la entrada del usuario con validación de entrada y manejo de errores.

- **NgForm**: crea una instancia de `FormGroup` de nivel superior y la vincula a un elemento `<form>` para rastrear el valor agregado del formulario y el estado de validación. Tan pronto como se importe `FormsModule`, esta directiva se activa por defecto en todas las etiquetas `<form>`. No se necesita añadir un selector especial.

- **NgModelGroup**: crea y vincula una instancia de `FormGroup` a un elemento DOM.

Los *"Template-driven forms"* se configuran de manera declarativa en la plantilla HTML. Esto convierte a este tipo de *forms* en simples y adecuados para formularios simples y de pequeña a mediana escala.

Utilizan las directivas de Angular como `ngModel`, `ngForm`, y `ngSubmit` para enlazar datos y manejar eventos del formulario.

Los formularios basados en plantillas aprovechan el enlace bidireccional o *"two-way data binding"* de Angular para sincronizar el modelo de datos con los elementos del formulario. Esto se hace principalmente mediante la directiva `[(ngModel)]`.

Las validaciones se definen en la plantilla utilizando atributos estándar de HTML5 y directivas de Angular. Angular proporciona directivas como `required`, `minlength`, `maxlength` o `pattern` para realizar validaciones.

Para trabajar con formularios basados en plantillas, se necesita importar `FormsModule` en el módulo de la aplicación Angular.

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

@Component({
  selector: 'app-template-driven-forms',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './template-driven-forms.component.html',
  styleUrl: './template-driven-forms.component.css'
})
export class TemplateDrivenFormsComponent {

  model = {
    name: ''
  };

  onSubmit(form: any) {
    console.log('Formulario enviado:', form);
  }

}
```

Esta es la plantilla HTML de este ejemplo, con la directiva `ngModel` de `FormsModule` enlazada con las propiedad del modelo de datos:

```
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <label for="name">Nombre:</label>
  <input type="text" id="name" name="name" [(ngModel)]="model.name" required>

  <button type="submit">Enviar</button>
</form>
```

Cuando se incluye la directiva usando la sintaxis para el enlace de datos bidireccional o *"two-way data binding"* `[(ngModel)]`, Angular puede rastrear el valor y la interacción del usuario con el control y mantener la vista sincronizada con el modelo.

Access the overall form status

Cuando se importa el módulo `FormsModule` en el componente, Angular crea y adjunta automáticamente una directiva `NgForm` al elemento `<form>` en la plantilla (porque `NgForm` tiene el selector `form` que coincide con los elementos `<form>`).

Para obtener acceso a `NgForm` y al estado general del formulario, declare una variable de referencia de plantilla.

```
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <!-- ... -->
</form>
```

La variable de la plantilla `#myForm` es ahora una referencia a la directiva `NgForm`.

Naming control elements

Cuando se utiliza `[(ngModel)]` en un elemento en Angular, se debe definir un atributo `name` para ese elemento. Angular utiliza el nombre asignado para registrar el elemento con la directiva `NgForm` que está adjunta al elemento `<form>` padre.

El atributo `name` actúa como un identificador único para cada control en el formulario, lo que permite que Angular mantenga el estado del control (como su valor, validez y estado de modificación) y lo gestione correctamente dentro del contexto del formulario.

Además, cuando se envía el formulario, Angular utiliza los nombres de los controles para recoger los datos del formulario y construir el objeto de datos que se envía al servidor o que se maneja en el componente.

```
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <label for="username">Username:</label>
  <input id="username" name="username" [(ngModel)]="username" required>

  <label for="email">Email:</label>
  <input id="email" name="email" [(ngModel)]="email" type="email" required>

  <button type="submit">Submit</button>
</form>
```

Cada elemento `<input>` tiene una propiedad de identificación o `id`. Esto lo utiliza el atributo `for` del elemento `<label>` para hacer coincidir la etiqueta con su control de entrada. Esta es una característica HTML estándar.

Cada elemento `<input>` también tiene la propiedad `name` requerida que Angular usa para registrar el control con el formulario.

HTTP Client

La mayoría de las aplicaciones de front-end necesitan comunicarse con un servidor a través del protocolo HTTP, para descargar o cargar datos y acceder a otros servicios de back-end.

Angular proporciona una API HTTP de cliente para aplicaciones Angular, la clase de servicio *"HttpClient"* en `@angular/common/http`.

Setting up HttpClient

Antes de poder usar *"HttpClient"* en la aplicación, se debe configurar mediante la inyección de dependencias.

"HttpClient" se proporciona mediante la función auxiliar `provideHttpClient()`, que la mayoría de las aplicaciones incluyen en los proveedores de aplicaciones `providers[]` en *"app.config.ts"*:

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter, withComponentInputBinding } from '@angular/router';
import { provideHttpClient } from '@angular/common/http';

import { routes } from './app.routes';
import { provideAnimationsAsync } from '@angular/platform-browser/animations/async';

export const appConfig: ApplicationConfig = {
  providers: [provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes, withComponentInputBinding()),
    provideAnimationsAsync(),
    provideHttpClient()]
};
```

La función `provideHttpClient()` acepta una lista de configuraciones de funciones opcionales para habilitar o configurar el comportamiento de diferentes aspectos del cliente.

withFetch()

De forma predeterminada, *"HttpClient"* utiliza la API `XMLHttpRequest` para realizar solicitudes.

La función `withFetch()` cambia el cliente para que utilice la API `fetch`. Esta es una API más moderna y está disponible en algunos entornos donde `XMLHttpRequest` no es compatible. Tiene algunas limitaciones, como no producir eventos de progreso de carga.

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(
      withFetch(),
    ),
  ]
};
```

withInterceptors(...)

Con la función `withInterceptors(...)` se configura el conjunto de funciones interceptoras que procesarán las solicitudes realizadas a través de *"HttpClient"*.

withInterceptorsFromDi(...)

Con la función `withInterceptorsFromDi(...)` incluye el estilo antiguo de interceptores basados en clases en la configuración de *"HttpClient"*.

withRequestsMadeViaParent()

De forma predeterminada, cuando configura *"HttpClient"* usando `provideHttpClient(...)` dentro de un inyector determinado, esta configuración anula cualquier configuración para *"HttpClient"* que pueda estar presente en el inyector principal.

Cuando se agrega `withRequestsMadeViaParent()`, *"HttpClient"* se configura para pasar solicitudes a la instancia de *"HttpClient"* en el inyector principal, una vez que hayan pasado por cualquier interceptor configurado en este nivel.

withJsonpSupport()

Incluir `withJsonpSupport()` habilita el método `.jsonp()` en *"HttpClient"*, que realiza una solicitud GET a través de la convención JSONP para la carga de datos entre dominios.

Sin embargo, es preferible usar CORS para realizar solicitudes entre dominios en lugar de JSONP cuando sea posible.

withXsrfConfiguration(...)

La inclusión de esta opción permite la personalización de la funcionalidad de seguridad XSRF integrada de *"HttpClient"*.

withNoXsrfProtection()

Incluir esta opción deshabilita la funcionalidad de seguridad XSRF incorporada de HttpClient.

Making requests

A partir del momento en que se haya importado y configurado `provideHttpClient()`, ya se puede inyectar el servicio *"HttpClient"* como una dependencia de componentes, servicios u otras clases:

```
@Injectable({providedIn: 'root'})
export class ConfigService {
  constructor(private http: HttpClient) {
    // This service can now make HTTP requests via `this.http`.
  }
}
```

"HttpClient" dispone de métodos correspondientes a los diferentes verbos HTTP utilizados para realizar solicitudes, tanto para cargar datos como para aplicar mutaciones en el servidor.

Cada método devuelve un `Observable` RxJS que, cuando se suscribe, envía la solicitud y luego emite los resultados cuando el servidor responde.

Los observables creados por *"HttpClient"* pueden suscribirse tantas veces como desee y realizarán una nueva solicitud de backend para cada suscripción.

Fetching JSON data

Obtener datos de un backend a menudo requiere realizar una solicitud GET utilizando el método `HttpClient.get()`. Este método toma dos argumentos: la URL del endpoint desde la cual obtener los datos y un objeto de opciones para configurar la solicitud.

```
getUserData(username: string): void {
  const url = 'https://api.github.com/users';
  this.http.get<any>(`${url}/${username}`).subscribe({
    next: (data) => {
      this.userData = data;
    },
    error: (error) => {
      console.error('Error al obtener datos del usuario:', error);
    }
  });
}
```

De forma predeterminada, *"HttpClient"* supone que los servidores devolverán **datos JSON**. Al interactuar con una API que no es JSON, puede indicarle a *"HttpClient"* qué tipo de respuesta esperar y devolver al realizar la solicitud. Esto se hace con la opción `responseType`:

- **'json' (por defecto)**: datos JSON del tipo genérico dado
- **'text'**: datos en formato string
- **'arraybuffer'**: un `ArrayBuffer` que contiene los bytes de respuesta sin procesar
- **'blob'**: una instancia de `Blob`

Por ejemplo, puede pedirle a *"HttpClient"* que descargue los bytes sin procesar de una imagen .jpeg en un *"ArrayBuffer"*:

```
http.get('/images/dog.jpg', {responseType: 'arraybuffer'}).subscribe(buffer => {
  console.log('The image is ' + buffer.byteLength + ' bytes large');
});
```

Mutating server state

Los APIs del servidor que realizan mutaciones a menudo requieren hacer solicitudes POST con un cuerpo de la solicitud que especifique el nuevo estado o el cambio que se debe realizar.

El método `HttpClient.post()` se comporta de manera similar a `HttpClient.get()`, y acepta un argumento adicional para el cuerpo (body) antes de sus opciones:

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

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

  private apiUrl = 'https://api.example.com/data';  // URL de la API

  constructor(private http: HttpClient) { }

  // Método para realizar una solicitud POST
  sendData(data: any): Observable<any> {
    const headers = new HttpHeaders({
      'Content-Type': 'application/json'
    });

    return this.http.post<any>(this.apiUrl, data, { headers: headers });
  }
}
```

Este componente utiliza el servicio:

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

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.css']
})
export class MyComponent implements OnInit {

  constructor(private apiService: ApiService) { }

  ngOnInit(): void {
    const postData = {
      name: 'John Doe',
      age: 30
    };

    this.apiService.sendData(postData).subscribe(response => {
      console.log('Response from server:', response);
    }, error => {
      console.error('Error occurred:', error);
    });
  }
}
```

Muchos tipos diferentes de valores pueden proporcionarse como el *"body"* de la solicitud, y *"HttpClient"* los **serializará** en consecuencia:

- **'string'**: serializado como texto en plano
- **'number, boolean, array or plain objet'**: serializado como JSON

- `ArrayBuffer` : datos sin procesar del buffer
- `Blob` : datos sin procesar con el tipo de contenido del Blob
- `FormData` : datos codificados *"multipart/form-data"*
- `HttpParams` : cadena formateada *"application/x-www-form-urlencoded"*
- `URLSearchParams` : cadena formateada *"application/x-www-form-urlencoded"*

Setting URL parameters

Especifique los parámetros de solicitud que deben incluirse en la URL de solicitud mediante la opción *"params"*.

Pasar un objeto es la forma más sencilla de configurar los parámetros de URL:

```
http.get('/api/config', {
  params: {filter: 'all'},
}).subscribe(config => {
  // ...
});
```

Como alternativa, se puede pasar una instancia de *"HttpParams"* si se necesita más control sobre la construcción o serialización de los parámetros.

```
const baseParams = new HttpParams()
      .set('filter', 'all')
      .set('show', 'yes');

// URL -> /api/config?filter=all&show=yes&details=enabled
http.get('/api/config', {
  params: baseParams.set('details', 'enabled'),
}).subscribe(config => {
  // ...
});
```

`HttpParams` es una clase inmutable para manejar parámetros de URL de HTTP y no puede ser modificada directamente. En vez de eso, hay que usar métodos como `.set()` o `.append()`.

Setting request headers

Se puede especificar los encabezados de solicitud que deben incluirse en la solicitud mediante la opción *"headers"*.

Pasar un objeto es la forma más sencilla de configurar encabezados de solicitud:

```
http.get('/api/config', {
  headers: {
    'X-Debug-Level': 'verbose',
  }
}).subscribe(config => {
  // ...
});
```

Como alternativa, se puede pasar una instancia de *"HttpHeaders"* si necesita más control sobre la construcción de encabezados.

`HttpHeaders` es una clase inmutable para manejar encabezados y no puede ser modificada directamente. En vez de eso, hay que usar métodos como `.set()` o `.append()`.

```
const baseHeaders = new HttpHeaders().set('X-Debug-Level', 'minimal');

http.get<Config>('/api/config', {
  headers: baseHeaders.set('X-Debug-Level', 'verbose'),
}).subscribe(config => {
  // ...
});
```

Interacting with the server response events

Para mayor comodidad, *"HttpClient"* devuelve de forma predeterminada un *"Observable"* de los datos devueltos por el servidor (el cuerpo de la respuesta). En ocasiones es conveniente examinar la respuesta real, por ejemplo, para recuperar encabezados de respuesta específicos.

Para acceder a la respuesta completa, configure la opción `observe` en 'response':

```
http.get<Config>('/api/config', {observe: 'response'}).subscribe(res => {
  console.log('Response status:', res.status);
  console.log('Body:', res.body);
});
```

Este es otro ejemplo con la API de Github:

```
getUserData2(username: string): void {
  const url = 'https://api.github.com/users';
  this.http.get<any>(`${url}/${username}`, { observe: 'response' }).subscribe({
    next: (data) => {
      this.userData = data;
      console.log('Response status:', data.status);  // Imprime el estado de la respuesta HTTP
      console.log('Body:', data.body);  // Imprime el cuerpo de la respuesta (tipo Config)
    },
    error: (error) => {
      console.error('Error al obtener datos del usuario:', error);
    }
  });
}
```

Receiving raw progress events

Además del cuerpo de la respuesta o del objeto de respuesta, *"HttpClient"* también puede devolver un flujo de eventos en bruto que corresponden a momentos específicos en el ciclo de vida de la solicitud.

Estos eventos incluyen cuándo se envía la solicitud, cuándo se devuelve el encabezado de respuesta y cuándo se completa el cuerpo. También pueden incluir eventos de progreso que informan sobre el estado de carga y descarga para cuerpos grandes de solicitud o respuesta.

Los eventos de progreso están deshabilitados de forma predeterminada (ya que tienen un costo de rendimiento), pero se pueden habilitar con la opción *"reportProgress"*.

La implementación `fetch` opcional de *"HttpClient"* no informa eventos de progreso de carga.

Handling request failure

Hay dos formas en que una solicitud HTTP puede **fallar**:

- Un error de red o de conexión puede impedir que la solicitud llegue al servidor backend.
- El backend puede recibir la solicitud pero no puede procesarla y devolver una respuesta de error.

"HttpClient" captura ambos tipos de errores en un `HttpErrorResponse`, que devuelve a través del canal de error del Observable.

Los errores de red tienen un **código de estado 0** y el error es una instancia de `ProgressEvent`.

Los errores del servidor backend tienen el **código de estado fallido devuelto por el backend**, y la respuesta de error como *"error"*.

Se puede utilizar el operador *"catchError"* para transformar una respuesta de error en un valor para la interfaz de usuario. Este valor puede indicarle a la interfaz de usuario que muestre una página o valor de error y capture la causa del error si es necesario.

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

// ...

ngOnInit(): void {
  this.apiService.getData().subscribe(
    (response) => {
      console.log('Response from server:', response);
      // Trabajar con la respuesta exitosa
    },
    (error: HttpErrorResponse) => {
      if (error.error instanceof ErrorEvent) {
        // Error del lado del cliente o de red
        console.error('An error occurred:', error.error.message);
      } else {
        // Error del lado del servidor
        console.error(
          `Backend returned code ${error.status}, ` +
          `body was: ${error.error}`);
      }
      // Realizar acciones específicas en caso de error
    }
  );
}
```

Dentro de la función de manejo de errores `(error: HttpErrorResponse) => { ... }`, ahora `error` es tipado como `HttpErrorResponse`, lo que permite acceder a propiedades específicas del error como `error.status`, `error.error`, etc.

`error.error instanceof ErrorEvent`: verifica si el error es del lado del cliente o de red (`ErrorEvent`), por ejemplo, problemas de conexión o errores de cliente.

Si no es un `ErrorEvent`, se asume que es un error del servidor, por lo que se imprime el código de estado del error y el cuerpo de la respuesta de error.

Http Observables

Los métodos de *"HttpClient"* devuelven *"Observables"* que representan el tipo de respuesta solicitada. Estos *"Observables"* son "fríos" en RxJS, lo que significa que no se realiza ninguna solicitud hasta que se suscriban.

Suscribirse a un *"Observable"* de *"HttpClient"* desencadena una solicitud al backend. Las múltiples suscripciones resultan en múltiples solicitudes, ya que cada suscripción es independiente.

Desuscribirse de un *"Observable"* suscrito aborta la solicitud en curso. Esto es beneficioso, especialmente al usar pipes asíncronas tipo `async` para cancelar automáticamente las solicitudes al navegar.

Al recibir una respuesta, los *"Observables"* de *"HttpClient"* suelen completarse (los interceptores pueden afectar este comportamiento).

Las suscripciones de *"HttpClient"* generalmente evitan fugas de memoria debido a la finalización automática. Sin embargo, se recomienda limpiar las suscripciones al destruir el componente para evitar errores con componentes destruidos.

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

@Component({
  selector: 'app-github-user',
  template: `
    <div *ngIf="user$ | async as user; else loading">
      <h2>{{ user.login }}</h2>
      <p>Nombre: {{ user.name }}</p>
      <p>Ubicación: {{ user.location }}</p>
      <p>Repositorios públicos: {{ user.public_repos }}</p>
    </div>
    <ng-template #loading>
      <p>Cargando usuario...</p>
    </ng-template>
  `,
  styleUrls: ['./github-user.component.css']
})
export class GithubUserComponent implements OnInit {

  user$: Observable<any>;

  constructor(private http: HttpClient) { }

  ngOnInit(): void {
    this.user$ = this.http.get('https://api.github.com/users/octocat');
  }
}
```

Best practices

Si bien *"HttpClient"* se puede inyectar y utilizar directamente desde los componentes, generalmente es recomendable **crear servicios inyectables y reutilizables** que aíslen y encapsule la lógica de acceso a datos:

```
@Injectable({providedIn: 'root'})
export class UserService {
  constructor(private http: HttpClient) {}
  getUser(id: string): Observable<User> {
    return this.http.get<User>(`/api/user/${id}`);
  }
}
```

Dentro de un componente, se puede combinar `@if` con la canalización asíncrona para representar la interfaz de usuario de los datos solo después de que haya terminado de cargarse:

```
import { AsyncPipe } from '@angular/common';
@Component({
  standalone: true,
  imports: [AsyncPipe],
  template: `
    @if (user$ | async; as user) {
```

```
      <p>Name: {{ user.name }}</p>
      <p>Biography: {{ user.biography }}</p>
    }
  `,
})
export class UserProfileComponent {
  @Input() userId!: string;
  user$!: Observable<User>;

  constructor(private userService: UserService) {}

  ngOnInit(): void {
    this.user$ = userService.getUser(this.userId);
  }
}
```

Interceptors

En Angular, los *"interceptors"* (interceptores) son una característica poderosa del módulo `HttpClient` que permite **interceptar y manipular las solicitudes HTTP y sus respuestas**.

Se utilizan principalmente para agregar lógica común a todas las solicitudes HTTP, como la gestión de autenticación, el manejo de errores, la configuración de encabezados, el registro de solicitudes, entre otros.

`HttpClient` admite dos tipos de interceptores: funcionales y basados en DI. La recomendación es utilizar **interceptores funcionales** porque tienen un comportamiento más predecible.

Los interceptores generalmente son funciones que se pueden ejecutar para cada solicitud y tienen amplias capacidades para afectar el contenido y el flujo general de solicitudes y respuestas. Estos interceptores se pueden **encadenar**, de forma cada interceptor procesa la solicitud o respuesta antes de reenviarla al siguiente interceptor de la cadena.

Puede utilizar interceptores para implementar una variedad de patrones comunes, como por ejemplo:

- Agregar encabezados de autenticación a solicitudes salientes a una API en particular.
- Reintentar solicitudes fallidas con retroceso exponencial.
- Almacenamiento en caché de las respuestas durante un período de tiempo o hasta que las invaliden las mutaciones.
- Personalización del análisis de respuestas.
- Medir los tiempos de respuesta del servidor y registrarlos.
- Controlar elementos de la interfaz de usuario, como un control de carga tipo *"spinner"*, mientras las operaciones de red están en curso.
- Solicitudes de recopilación y lotes realizadas dentro de un período de tiempo determinado.
- Solicitudes que fallan automáticamente después de una fecha límite o tiempo de espera configurable.
- Sondear periódicamente el servidor y actualizar los resultados.

Defining an interceptor

La forma básica de un interceptor es una función que recibe la `HttpRequest` saliente y una función `next` que representa el siguiente paso de procesamiento en la cadena del interceptor.

```
export function loggingInterceptor(req: HttpRequest<unknown>, next: HttpHandlerFn): Observable<HttpEvent<unknown>> {
  console.log(req.url);
  return next(req);
}
```

Configuring interceptors

Se declara el conjunto de interceptores en la función `withInterceptors(...)` cuando se configura `HttpClient` mediante DI:

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter, withComponentInputBinding } from '@angular/router';
import { provideHttpClient } from '@angular/common/http';

import { routes } from './app.routes';
import { provideAnimationsAsync } from '@angular/platform-browser/animations/async';

export const appConfig: ApplicationConfig = {
  providers: [provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes, withComponentInputBinding()),
    provideAnimationsAsync(),
    provideHttpClient(
      withInterceptors([loggingInterceptor, cachingInterceptor]),
    )]
};
```

Los interceptores se encadenan en el orden en que se hayan enumerado en `withInterceptors(...)`.

Modifying requests

La mayoría de los aspectos de las instancias de *"HttpRequest"* y *"HttpResponse"* son **inmutables**, y los interceptores no pueden modificarlos directamente.

En su lugar, los interceptores aplican modificaciones clonando estos objetos usando la operación `.clone()`, y especificando qué propiedades deben ser mutadas en la nueva instancia.

```
const reqWithHeader = req.clone({
  headers: req.headers.set('X-New-Header', 'new header value'),
});
```

Dependency injection in interceptors

Los interceptores se ejecutan en el contexto de inyección del inyector que los registró y pueden usar la API *"inject"* de Angular para recuperar dependencias.

Un interceptor puede inyectar y utilizar un servicio si lo necesita:

```
export function authInterceptor(req: HttpRequest<unknown>, next: HttpHandlerFn) {
  // Inject the current `AuthService` and use it to get an authentication token:
  const authToken = inject(AuthService).getAuthToken();
  // Clone the request to add the authentication header.
  const newReq = req.clone({headers: {
    req.headers.append('X-Authentication-Token', authToken),
  }});
  return next(newReq);
}
```

Request and response metadata

A menudo resulta útil incluir información en una solicitud que no se envía al backend, sino que está destinada específicamente a los interceptores.

Las *"HttpRequests"* tienen un objeto `.context` que almacena este tipo de metadatos como una instancia de `HttpContext`. Este objeto funciona como un mapa escrito, con claves de tipo `HttpContextToken`.

Por ejemplo, para almacenar si el interceptor de almacenamiento en caché debe almacenar en caché una solicitud particular en el mapa `.context` de esa solicitud, defina un nuevo `HttpContextToken` para que actúe como clave:

```
export const CACHING_ENABLED = new HttpContextToken<boolean>(() => true);
```

La función proporcionada crea el valor predeterminado para el token para solicitudes que no han establecido explícitamente un valor para él. El uso de una función garantiza que si el valor del token es un objeto o una matriz, cada solicitud obtenga su propia instancia.

Luego, un interceptor puede leer el token y elegir aplicar la lógica de almacenamiento en caché o no en función de su valor:

```
export function cachingInterceptor(req: HttpRequest<unknown>, next: HttpHandlerFn): Observable<HttpEvent<unknown>> {
  if (req.context.get(CACHING_ENABLED)) {
    // apply caching logic
    return ...;
  } else {
    // caching has been disabled for this request
    return next(req);
  }
}
```

Al realizar una solicitud a través de la API *"HttpClient"*, puede proporcionar valores para *"HttpContextTokens"*:

```
const data$ = http.get('/sensitive/data', {
  context: new HttpContext().set(CACHING_ENABLED, false),
});
```

Testing

TODO

Referencias

- https://angular.dev
- https://www.typescriptlang.org
- https://www.youtube.com/@Angular
- https://material.angular.io/

Licencia