Profesor: Borja Rey Seoane

TEMA 5

INTRODUCIÓN A ANGULAR

Módulo: **Desenvolvemento Web en Contorna Cliente**

Ciclo: **DAW**

Introdución a Angular

Neste tema aprenderemos os conceptos básicos sobre o *framework* Angular e centrarémonos tamén na creación de aplicacións que sigan o patrón MVC e a funcionalidade Cross-Browser[1], para o cal pretendemos acadar os seguintes obxectivos:

- Identificar as diferenzas que presenta o modelo en función dos navegadores.
- Programar aplicacións Web de xeito que funcionen en navegadores con diferentes implementacións do modelo.
- Independizar o contido, o aspecto e o comportamento en aplicacións Web.

Angular 2+

A primeira versión de Angular, lanzada en 2010 (e que difería en grande medida da versión actual), foi Angular JS. Tratábase dun *framework* Web de *front-end* baseado, coma é obvio, en JavaScript e desenvolto e mantido por Google. Podía empregarse nunha páxina HTML engadindo a correspondente etiqueta `<script>`.

En 2016 aparece Angular 2, supoñendo un cambio radical de enfoque e unha reescritura completa do *framework* (incluíndo mesmo o cambio da linguaxe de base cara TypeScript). A partires dese punto xurdiron as versións 4 a 19, a través das cales a funcionalidade medrou exponencialmente (a versión que aquí tratamos, a 18, foi lanzada en agosto de 2024).

Angular 2+ é un *framework* de código aberto baseado en TypeScript que permite crear aplicacións Web dinámicas, constituíndo actualmente un dos marcos de traballo máis populares de JavaScript para o desenvolvemento de aplicacións Web tanto de escritorio como móbiles no lado cliente. Podemos sinalar unha serie de vantaxes no seu uso (fronte ao estándar de JS):

- Permite detectar erros en tempo de compilación.

[1] Denominamos Cross Browser a aquelas aplicacións Web que foron desenvoltas para verse e funcionar exactamente igual en calquera navegador.

- Produce un código máis lexible e estruturado.
- Fornece ferramentas avanzadas de autocompletado e refactorización.

Podemos tamén identificar unha serie de características de TypeScript:

- É un dialecto de JavaScript (máis concretamente un superconxunto) desenvolvido por Microsoft que actúa coma unha capa adicional de JS ES6.
- Engade tipado estático e funcionalidade orientada á obxectos.

Como características xerais de Angular 2+ podemos salientar:

- É un *framework* que permite crear SPA[2], o que ten tamén unha serie de vantaxes:
    - Todas as pantallas son amosadas na mesma páxina, sen necesidade de recargar o navegador cun novo ficheiro HTML ao mudar de ruta.
    - Permite dispoñer de múltiples vistas[3] (entendendo por vista o que sería unha pantalla nunha aplicación de escritorio).
- É orientado a compoñentes, o que permite:
    - Modularizar as funcionalidades da aplicación.
    - Encapsular lóxica, datos e vista.
- Emprega carga diferencial: unha tecnoloxía que permite compilar dous *bundles*[4] e servir un ou outro dependendo da versión do navegador que esteamos a empregar.
- Importación dinámica de rutas.
- Soporta Angular Ivy: motor de renderizado desenvolto por Google que fora introducido con Angular 8.

MVC con Angular

O **MVC** ou **Modelo-Vista-Controlador** é un patrón de arquitectura de *software* que separa a lóxica dunha aplicación da súa vista. Trátase dun estilo de arquitectura empregado tanto en compoñentes básicas como en amplos sistemas empresariais. Así, temos:

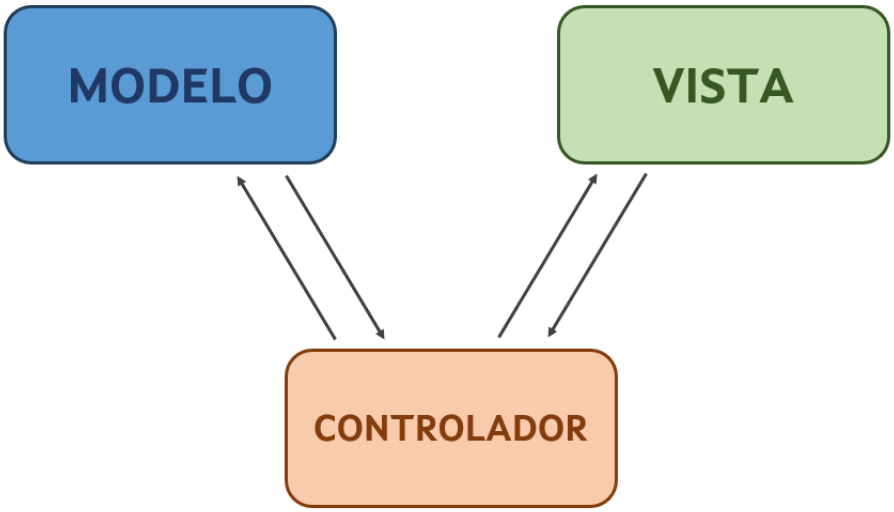
- **Modelo:**
    - É a parte encargada da xestión dos datos e a lóxica de negocio.
    - Comunícase co controlador para fornecer información á vista.
    - Polo xeral traballa sobre algún tipo de base de datos.

[2] *Single Page Application*.

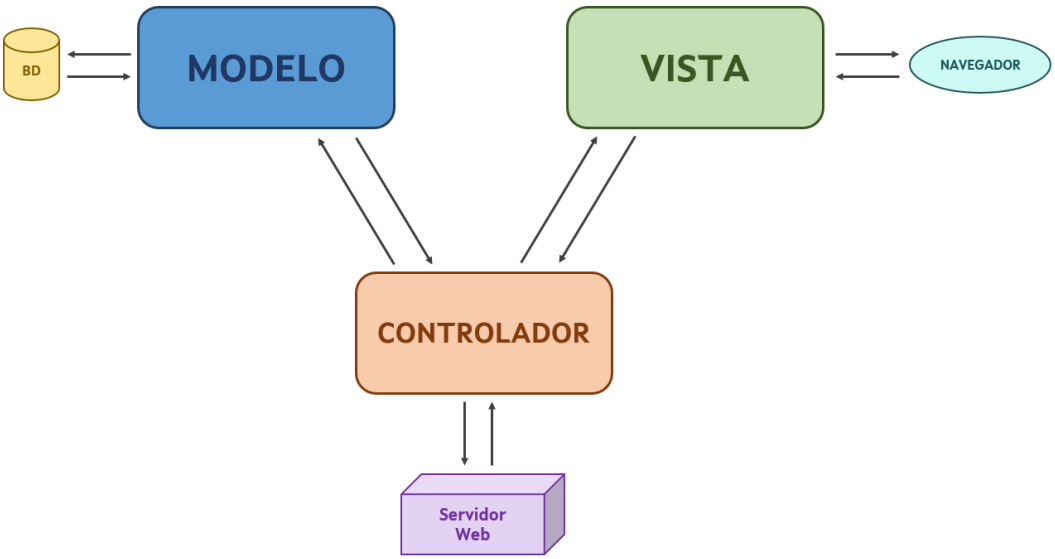
[3] Nunha SPA vanse intercambiando vistas producindo o efecto de que estivesen a navegar páxinas diferentes, cando realmente a páxina é única.

[4] Paquetes.

- **Vista:**
  - É a representación visual dos datos, todo o que teña que ver coa interface de usuario e as interaccións.
  - Recibe do controlador a información procesada procedente do modelo.
  - Tanto o modelo como o controlador non se preocupan na forma en que os datos serán amosados, recaendo esta responsabilidade unicamente na vista.
- **Controlador:**
  - É a parte encargada de xestionar e recibir as ordes do usuario, actualizar o modelo e comunicar os cambios á vista.
  - Actúa de intermediario entre o modelo e a vista.



O diagrama típico do patrón MVC.



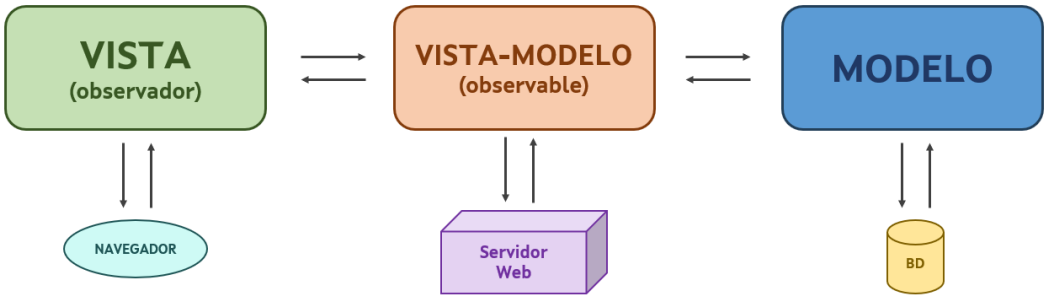
O diagrama do patrón MVC aplicado á programación Web.

Angular non emprega realmente o patrón MVC clásico dado que está baseado en compoñentes. Este *framework* presenta moita relación coa vista debido ao enlace de datos bidireccional (*two-way*

data binding) que permite sincronizar os datos entre a vista e o modelo, é dicir, na vista podemos modificar o modelo e no modelo podemos modificar a vista.

Isto fai que a independencia que existe no patrón clásico de modelo-vista-controlador desapareza en Angular, e polo tanto adoita chamarse **modelo-vista vista-modelo** (MVVM) ou ben **modelo-vista-whatever** (MVW). Así atopamos:

- **Modelo:** representado por clases ou servizos que xestionan os datos e a lóxica de negocio.
- **Vista:** definida polo código HTML de cada compoñente (inclúe a interface de usuario e o enlazado de datos).
- **Controlador:**
    - Encapsula a lóxica precisa para conectar o modelo coa vista.
    - En Angular, está representado principalmente polo código TypeScript de cada compoñente.



O diagrama do patrón MVVM aplicado á programación Web.

Non obstante, tamén existe o modelo como lóxica de negocio, como poden ser os servizos ou todo aquelo que se inxecte e que si está totalmente independizado da vista.

Arquitectura de Angular

Angular organiza o desenvolvemento de aplicacións Web empregando unha estrutura modular baseada en compoñentes, módulos, servizos e plantillas. Esta arquitectura permite estruturar o código de forma clara e escalable, facilitando o traballo con proxectos de distinta complexidade.

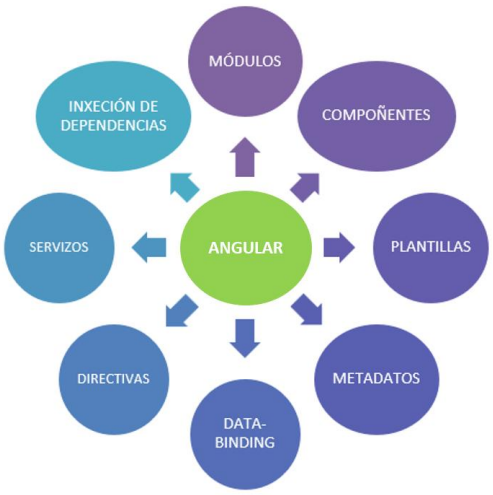
Elementos principais da arquitectura

A continuación vemos os distintos elementos dos que consta a arquitectura de Angular, cunha breve explicación de cada un:

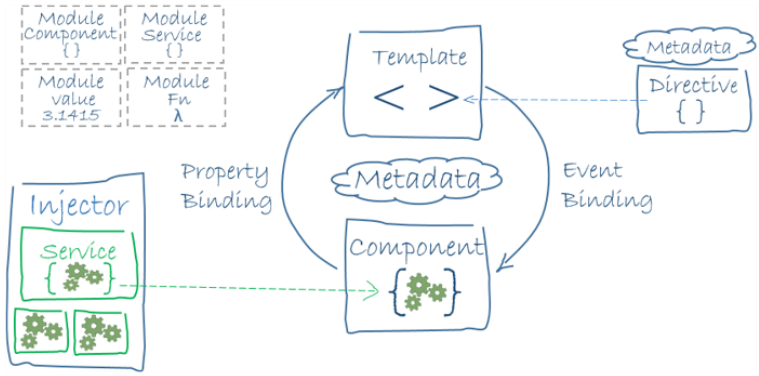
- **Módulos (`@NgModule`):**
    - Un módulo en Angular é unha unidade lóxica que agrupa compoñentes, servizos e outros elementos relacionados.

    - As aplicacións Angular poden ser *standalone* (non modulares) ou modulares (neste caso, cada aplicación Angular contará cun módulo raíz -habitualmente chamado **AppModule-**, encargado de inicializar a aplicación).
    - Outros módulos poden ser creados para organizar funcionalidade compartida ou específica.
- **Compoñentes (`@Component`):**
    - Os compoñentes son a unidade básica da interface de usuario en Angular.
    - Cada compoñente define unha parte da interface e a lóxica asociada.
- **Plantillas:**
    - As plantillas en Angular son arquivos ou cadeas que definen a estrutura HTML dun compoñente.
    - Os templates poden conter directivas, interpolación de datos e elementos HTML.
- **Directivas:**
    - As directivas permiten modificar o comportamento ou a estrutura dun elemento HTML na plantilla.
- **Servizos (`@Injectable`):**
    - Os servizos encapsulan funcionalidades reutilizables como chamadas a API, manipulación de datos ou lóxica de negocio.
    - Poden ser compartidos entre compoñentes a través do sistema de inxección de dependencias.
- **Inxección de dependencias:**
    - A inxección de dependencias permite a unha compoñente ou servizo recibir automaticamente instancias doutros servizos.
    - O sistema de inxección de dependencias xestiona a creación e distribución destes servizos.

Na figura seguinte aprécianse os elementos clave que dan a súa potencia a Angular:



Na figura seguinte podemos observar un esquema da lóxica de traballo da arquitectura.



Instalación

Neste capítulo trataremos os procedementos de instalación das ferramentas necesarias para desenvolver baixo Angular en Ubuntu 24.04 LTS:

- **Node.js:** contorna de execución JS de código aberto que permite executar o código fora de navegadores Web. Adoita empregarse no desenvolvemento en lado servidor.
- **Angular CLI:** ferramenta de liña de comandos que nos permite crear, desenvolver, estruturar e manter aplicacións Angular.

Instalación de Node.js

Para poder instalar a versión máis recente de **Node.js** en Ubuntu 24.04 primeiramente temos que seguir os pasos do sitio oficial da tecnoloxía:

https://deb.nodesource.com/

Con ese fin, dende a terminal actualizamos a listaxe de paquetes:

```
sudo apt update
```

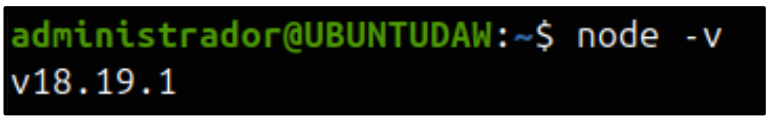
Seguidamente, instalamos Node.js dende os repositorios oficiais de Ubuntu:

```
sudo apt install nodejs -y
```

Se quixeramos comprobar que a versión instalada é a correcta, faremos:

```
node -v
```

Velaquí o resultado no noso caso:



Finalmente, instalamos o xestor de paquetes de NodeJS (npm):

```
sudo apt install npm -y
```

Podemos tamén comprobar a versión que acaba de instalar con:

```
npm -v
```

No noso caso:

```
administrador@UBUNTUDAW:~$ npm -v
9.2.0
```

Instalación de Angular CLI

Unha vez instalado **Node.js**, temos xa dispoñible o Node Package Manager (ou simplemente, **npm**), un xestor de paquetes que vai permitirnos engadir funcionalidade ao noso servidor.

O primeiro paso é limpar a caché de npm:

```
sudo npm cache verify
```

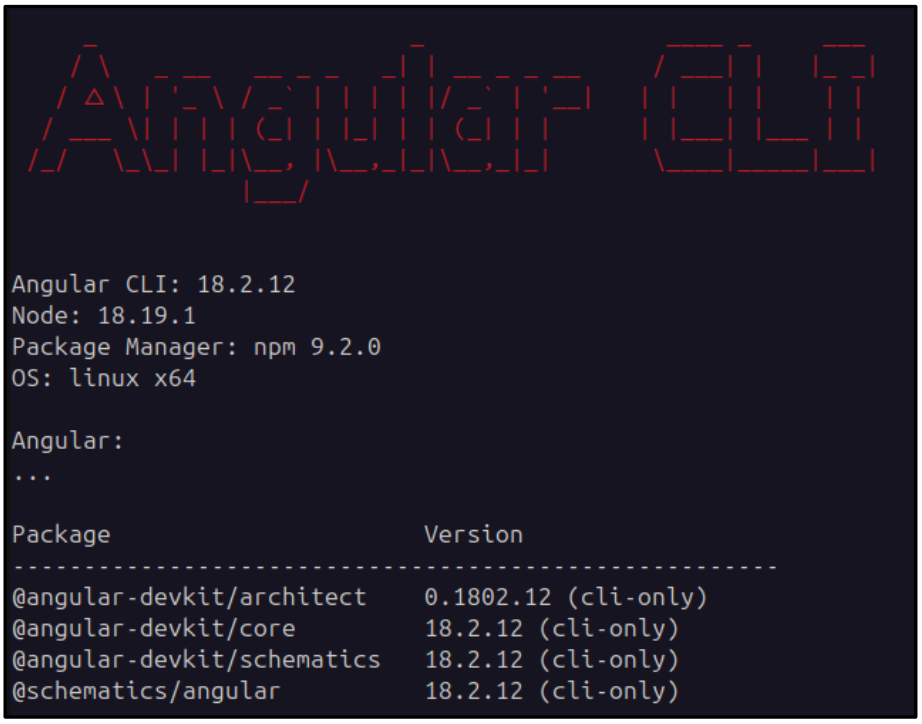
Agora instalamos a versión de Angular CLI que desexamos (neste caso a 18.2.12):

```
sudo npm install -g @angular/cli@18.2.12
```

Igualmente ca no caso anterior, podemos comprobar a versión unha vez instalada para asegurar que é a correcta:

```
ng version
```

Obtendo algo semellante a isto:



Estrutura dunha aplicación Angular

Coma xa mencionamos no apartado referido á arquitectura, existen diversos tipos de elementos que constitúen a estrutura dunha aplicación Angular, sendo:

- **Módulos.**
- **Compoñentes.**
- **Modelos[5].**
- **Enlaces de datos.**
- **Directivas.**
- **Servizos.**
- **Inxección de dependencias.**

Módulos

Os **módulos** en Angular son unidades fundamentais para a organización, encapsulamento e reemprega de funcionalidade dentro dun proxecto. Represéntanse mediante clases que empregan o decorador **@NgModule** para definir as súas compoñentes, servizos e dependencias.

Estrutura dun módulo

Un módulo típico posúe unha serie de metadatos que especifican o xeito en que interactúan os seus elementos con outras compoñentes da aplicación, sendo estes:

- **Declaracións (*declarations*):**
    - Neste apartado defínense as vistas (compoñentes, directivas e *pipes*) que pertencen ao módulo.
    - As vistas dun módulo só poderán ser empregadas dentro do propio módulo non sendo que sexan exportadas.
    - Velaquí un exemplo dunha sección `declarations` na cabeceira dun módulo:

```
declarations: [
  AppComponent,
  HeaderComponent,
  FooterComponent
]
```

[5] En castelán «plantillas» (no orixinal, en inglés, *«templates»*).

- **Exportacións (*exports*):**
    - Define os elementos que estarán dispoñibles para outros módulos que importen o módulo actual.
    - Emprégase para compartir as compoñentes, directivas e *pipes* propios do módulo.
    - Velaquí un exemplo dunha sección `exports` na cabeceira dun módulo:

```
exports: [
  HeaderComponent,
  FooterComponent
]
```

- **Importacións (*imports*):**
    - Especifica os módulos que o módulo actual empregará para obter funcionalidades adicionais.
    - Pódense importar tanto módulos predefinidos de Angular (como, por exemplo, `BrowserModule` ou `FormsModule`) coma módulos personalizados.
    - Velaquí un exemplo dunha sección `imports` na cabeceira dun módulo:

```
imports: [
  BrowserModule,
  ReactiveFormsModule
]
```

- **Provedores (*providers*):**
    - Indica os servizos que estarán dispoñibles para as compoñentes do módulo (ou para toda a aplicación se foron definidos no módulo raíz).
    - Velaquí un exemplo dunha sección `imports` na cabeceira dun módulo:

```
providers: [
  AuthService,
  ApiService
]
```

- **Arrinque (*bootstrap*):**
    - Especifica a compoñente raíz que Angular debe cargar ao iniciar a aplicación.
    - Só debe figurar no módulo raíz (`AppModule`).
    - Velaquí un exemplo da sección `bootstrap` do módulo raíz:

```
bootstrap: [
  AppComponent
]
```

Tipos de módulos

- **Módulo Raíz:**
  - É o núcleo da aplicación, xeralmente denominado `AppModule`.
  - Contén a configuración global da aplicación e o compoñente raíz.
- **Módulos de Funcionalidade:**
  - Encapsulan funcionalidade específica, permitindo unha mellor organización e modularidade.
  - Exemplo: un módulo para xestionar usuarios (`UserModule`) ou un módulo para manexar pedidos (`OrderModule`).
- **Módulos Compartidos:**
  - Contén compoñentes, directivas e *pipes* que son usados en múltiples módulos.
  - Exemplo: un `SharedModule` que exporta botóns, cadros de diálogo, ou directivas comúns.
- **Módulos de Servizo:**
  - Usados para agrupar servizos específicos que poden ser compartidos en distintos módulos.
  - Normalmente incluídos en `providers`.

Exemplo dun módulo

Veremos a continuación un exemplo de código dun módulo raíz básico:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { HeaderComponent } from './header/header.component';
import { FooterComponent } from './footer/footer.component';

@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent,
    FooterComponent
  ],
  imports: [
    BrowserModule
  ],
  exports: [
    HeaderComponent,
    FooterComponent
  ],
  providers: [
    AuthService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Compoñentes

As **compoñentes** son a peza clave de Angular, encargadas de definir e controlar as diferentes «vistas» ou seccións visuais da aplicación.

Cada compoñente encapsula a súa propia estrutura (HTML), estilos (CSS) e comportamento (TS), promovendo unha separación clara de responsabilidades e funcionalidades.

Estrutura dunha compoñente

Unha compoñente represéntase mediante unha clase TypeScript que emprega o decorador `@Component` para engadir os seus metadatos. Estes metadatos definen a funcionalidade, apariencia e o xeito de integración da compoñente co resto da aplicación, sendo:

- **Selector (`selector`):**
  - Define o nome da etiqueta HTML que representa á compoñente dentro do proxecto.
  - Permite inserir a compoñente noutras vistas.
  - Velaquí un exemplo dunha sección `selector` na cabeceira dunha compoñente:

```
selector: 'app-cabeceira'
```

O que permitiría empregar a compoñente mediante a etiqueta:

```
<app-cabeceira></app-cabeceira>
```

- **Modelo interno (`template`):**
  - Contén o código HTML da vista da compoñente directamente escrito na propia cabeceira (*inline*).
  - Velaquí un exemplo dunha sección `template` na cabeceira dunha compoñente:

```
template: '<h1>Benvid@s a Angular!</h1>'
```

- **Modelo externo (`templateUrl`):**
  - Alternativo ao anterior, permite especificar o ficheiro externo onde está almacenado o código HTML da compoñente.
  - Velaquí un exemplo dunha sección `templateUrl` na cabeceira dunha compoñente:

```
templateUrl: './cabeceira.component.html'
```

- **Estilos internos (`styles`):**
    - Contén o código CSS da vista da compoñente directamente escrito na propia cabeceira (*inline*).
    - Velaquí un exemplo dunha sección `styles` na cabeceira dunha compoñente:

```
styles: [
  'h1 { color: blue; }
']
```

- **Estilos externos (`stylesUrls`):**
    - Contén a listaxe de ficheiros CSS que a compoñente emprega na súa vista.
    - Velaquí un exemplo dunha sección `stylesUrls` na cabeceira dunha compoñente:

```
styleUrls: [
'./header.component.css'
]
```

- **Encapsulamento (`encapsulation`):**
    - Angular aplica por defecto a encapsulamento de estilos (Shadow DOM-like) para que o CSS dunha compoñente non interfira cos doutras.
    - Velaquí un exemplo dunha sección `encapsulation` na cabeceira dunha compoñente:

```
encapsulation: ViewEncapsulation.Emulated
```

- **Anfitrión (`host`):**
    - Define propiedades e eventos directamente no elemento que representa a compoñente.
    - Velaquí un exemplo dunha sección `host` na cabeceira dunha compoñente:

```
host: {
  '(click)': 'onClic()',
  '[class.active]': 'isActive'
}
```

Ademais dos metadatos anteriores, as compoñentes poden incorporar propiedades de entrada e saída para facilitar o intercambio de datos dende e cara á súa compoñente nai (que serán tratados en máis detalle nunha sección posterior):

- **Propiedades de entrada (`@Input`):** permiten á compoñente recibir datos dende a compoñente nai.
- **Propiedades de saída (`@Output`):** permiten á compoñente enviar datos cara á compoñente nai.

Exemplo dunha compoñente

A continuación preséntase o ficheiro de comportamento (`cabeceira.component.ts`) unha compoñente chamada `CabeceiraComponent`:

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

@Component({
  selector: 'app-cabeceira,
  templateUrl: './cabeceira.component.html',
  styleUrls: ['./cabeceira.component.css']
})
export class CabeceiraComponent {
  @Input() titulo: string;
  @Output() onLogout = new EventEmitter<void>();

  logout(): void {
    this.onLogout.emit();
  }
}
```

E de seguido podemos ver o seu código HTML (`cabeceira.component.html`):

```
<header>
  <h1>{{ titulo }}</h1>
  <button (click)="logout()">Logout</button>
</header>
```

E o seu código CSS (`cabeceira.component.css`):

```
header {
  background-color: #f4f4f4;
  padding: 10px;
  text-align: center;
}

button {
  margin-top: 10px;
}
```

A súa principal característica é que permiten estender funcións a partires doutras sen modificar a orixinal.

Entrada de datos

Angular pon á nosa disposición un decorador, **`@Input`**, que nos permite pasar información dende compoñentes nai cara compoñentes fillas (de xeito descendente na xerarquía de compoñentes).

Co gallo diso, márcase unha propiedade/atributo da compoñente nai co devandito decorador pasando a mesma a ser accesible polas compoñentes fillas.

Compoñente nai

Para o seguinte exemplo empregaremos coma compoñente nai á propia compoñente raíz de Angular (**`app-root`**). Ficando o seu código coma segue:

- **`app.component.ts`:** engadimos unha propiedade nova á cal chamamos **`compartida`**, neste caso de tipo numérico e iniciada co valor «0».

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FillaComponent } from './filla/filla.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FillaComponent ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'input';
  compartida : number = 0;
}
```

- **`app.component.html`:** neste ficheiro engadimos a etiqueta correspondente á compoñente filla (**`app-filla`**) cun *binding* de propiedades que asocia a súa propiedade **`recibida`** coa propiedade **`compartida`** da compoñente nai. A maiores incorporamos un botón que permite incrementar o valor da variable **`compartida`**, para comprobar como efectivamente a información flúe dende a compoñente nai cara á filla.

```
<app-filla [recibida]="compartida"></app-filla>
<button (click)="compartida=compartida+1">INCREMENTAR</button>
```

Compoñente filla

Supoñamos pois que temos unha compoñente filla denominada **`nova-componhente`** co seguinte código:

- **`filla.component.ts`:** neste caso engadimos unha propiedade (**`recibida`**) á clase e acompañámola do decorador `@Input` para indicar que vai asociada á información recibida pola etiqueta HTML da compoñente. O valor asignado á propiedade unicamente serve para confirmar que efectivamente o valor inicial que aparecerá na páxina non será o que aquí figura, senón o que se reciba dende a compoñente nai.

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

```
@Component({
  selector: 'app-filla',
  standalone: true,
  imports: [],
  templateUrl: './filla.component.html',
  styleUrl: './filla.component.css'
})

export class FillaComponent {
  @Input() recibida : number = 10; // Poñemos un valor por defecto á propiedade
para ver que efectivamente é sobreescrito dende a compoñente nai
}
```

- **`filla.component.html:`** o contido deste ficheiro é relativamente sinxelo, e unicamente hai unha interpolación de *strings* que dirixe á propiedade **`recibida`**.

```
<p>O valor da propiedade recibida por input é {{ recibida }}</p>
```

Saída de datos

O decorador **`@Output`** de Angular posibilita o paso de información dende as compoñentes fillas cara ás compoñentes nai, isto é, de xeito ascendente na xerarquía de compoñentes.

Co gallo diso, márcase unha propiedade/atributo da compoñente filla co devandito decorador pasando a mesma a ser accesible pola compoñente nai.

Compoñente nai

No exemplo que figura a continuación empregaremos coma compoñente nai á propia compoñente raíz de Angular (**`app-root`**). O código será o seguinte:

- **`app.component.ts:`** engadimos unha propiedade nova á cal chamamos **`recibida`**, neste caso de tipo numérico e iniciada co valor «0; e un método **`amosarMensaxe(texto)`** que escribe o texto recibido sobre a variable **`recibida`**.

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FillaComponent } from './filla/filla.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FillaComponent ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'output';
  recibida : string = ''; // Poñemos un valor por defecto á propiedade para que se
vexa que efectivamente é sobreescrito dende a compoñente filla
```

```
  amosarMensaxe(texto: string): void {
    this.recibida = texto;
  }
}
```

- **app.component.html:** engadimos a etiqueta correspondente á compoñente filla (**nova-componhente**) incorporando un *binding* de eventos que chama ao método **amosarMensaxe(texto)** pasándolle coma parámetro o evento desencadeado.

```
<app-filla (mensaxe)="amosarMensaxe($event)"></app-filla>
<p>A compoñente filla enviou a seguinte mensaxe: <b>{{ recibida }}</b></p>
```

Compoñente filla

O código da nosa compoñente filla (**filla**) é como figura a continuación:

- **filla.component.ts:** neste caso engadimos unha propiedade (**mensaxe**) á clase e acompañámola do decorador `@Output` de tipo `EventEmitter` e con tipo de datos `string`, isto implica que **mensaxe** será un evento emisible que pode ser desencadeado dende HTML. Ademais diso, engadimos tamén un método `enviar()` no cal emítese o evento **mensaxe** cunha mensaxe asociada.

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

@Component({
  selector: 'app-filla',
  standalone: true,
  imports: [],
  templateUrl: './filla.component.html',
  styleUrl: './filla.component.css'
})

export class FillaComponent {
  @Output() mensaxe = new EventEmitter<string>(); // Esta propiedade actuará como
vehículo da mensaxe

  enviar() {
    this.mensaxe.emit('Ola compoñente nai!');
  }
}
```

- **filla.component.html:** unicamente contén un botón HTML que chama ao método `enviar()`.

```
<button (click)="enviar()">ENVIAR</button>
```

Enlaces de datos

O mecanismo de Angular que permite intercambiar datos entre modelo e clase da compoñente denomínase **enlace de datos** ou ***data binding***.

Angular distingue catro tipos de enlace de datos, os cales trataremos en detalle máis adiante.

Directivas

As **directivas** permiten engadir comportamento dinámico ao código HTML. Isto é posible mediante o uso duns selectores especiais que poden ser de tipo estrutural ou representar atributos HTML.

Son unha das ferramentas máis poderosas para manipular o comportamento e a presentación do DOM. Estas permiten implementar funcionalidades dinámicas e interactuar co DOM de xeito eficiente, favorecendo unha arquitectura modular e escalable.

As directivas serán tamén tratadas con máis detalle en adiante.

Servizos

Os **servizos** de Angular son clases que permiten encapsular e centralizar funcionalidade que posteriormente pode ser compartida entre distintas partes dunha aplicación, como compoñentes ou outros servizos. Os servizos constitúen o alicerce dunha arquitectura limpa, modular e reempregable.

Dun xeito máis xeral podemos atopar no uso de servizos as seguintes vantaxes:

- **Manteñen a lóxica compartida fóra das compoñentes:** actuando coma intermediarios entre compoñentes.
- **Reemprega de código:** un mesmo servizo pode ser usado por múltiples compoñentes, reducindo a replicación innecesaria de código.
- **Comunicación entre compoñentes:** facilitan a transferencia de datos entre compoñentes sen relación directa no DOM.
- **Acceso a recursos externos:** son ideais para xestionar chamadas HTTP ou para interactuar con APIs.

Estrutura dun servizo

Os servizos empregan o decorador `@Injectable` para engadir os seus metadatos. Estes metadatos definen a configuración do servizo, sendo:

- **Ámbito (`providedIn`):**
    - Coma o seu nome indica, determina o ámbito do servizo, isto é, onde estará dispoñible na aplicación.

- Velaquí un exemplo dunha sección `providedIn` na cabeceira dun servizo:

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

Tipos de servizos

En Angular podemos clasificar os servizos segundo o ámbito no que sexan accesibles, así temos:

- **Servizos globais:**
  - Accesibles dende calquera parte da aplicación.
  - Son aqueles que levan o valor `root` no metadato `providedIn`.
- **Servizos locais a un módulo:**
  - Definidos no metadato `providers` dun módulo específico:
  - Vexamos un pequeno exemplo:

```
import { NgModule } from '@angular/core';
import { UsuarioServizo } from './usuario.service';

@NgModule({
  providers: [UsuarioServizo]
})
export class UsuarioModulo {}
```

- **Servizos locais a unha compoñente:**
  - Definidos no metadato `providers` dunha compoñente.
  - Xeran unha nova instancia do servizo para cada compoñente.
  - Vexamos un exemplo de uso desta opción:

```
import { Component } from '@angular/core';
import { UsuarioServizo } from './usuario.service';

@Component({
  selector: 'app-usuario',
  providers: [UsuarioServizo],
  template: '<p>Compoñente Usuario</p>'
})
export class UsuarioComponent {}
```

Exemplo dun servizo

No código seguinte podemos ver o código dun servizo chamado `ExemploServizo`:

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

@Injectable({
  providedIn: 'root' // O servizo estará dispoñible en toda a aplicación.
})
```

```
export class ExemploServizo {
  constructor() {}

  obterDatos(): string {
    return 'Datos do servizo';
  }
}
```

Inxección de dependencias

As **inxeccións de dependencias** son un patrón de deseño fundamental en Angular, xa que facilitan a creación, xestión e provisión de obxectos e instancias de clases, chamadas dependencias, ás distintas partes da aplicación (compoñentes, servizos e outros elementos).

Dito doutro xeito, a inxección de dependencias permite que as clases non se encarguen de crear directamente as súas dependencias, senón que estas sonlles fornecidas a través dun mecanismo externo (o contedor de inxección de dependencias). Con isto conséguese unha arquitectura máis modular e doada de manter.

Este mecanismo funciona en dúas fases:

- **Provisión:** as dependencias (servizos, valores ou outros obxectos) son rexistradas nun contedor (`Injector`).
- **Inxección:** cando unha compoñente ou servizo precisa dunha dependencia, o `Injector` fornécella automaticamente.

Estrutura dunha inxección de dependencias

Angular emprega diversos metadatos e configuracións para xestionar a inxección de dependencias, sendo estes:

- **O decorador `@Injectable`:**
  - Marca unha clase como susceptible de ser inxectada noutras partes da aplicación.
  - Inclúe a configuración de onde estará dispoñible a dependencia.
  - Vexamos un pequeno exemplo na declaración dun servizo:

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

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

- **O metadato `providers`:**
  - Define onde e como deben de se crear as dependencias.

- Pode especificarse no módulo (@`NgModule`), na compoñente (@`Component`) ou mesmo nun servizo (`@Injectable`).
- Vexamos un exemplo de aplicación a un módulo:

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

@NgModule({
  providers: [UsuarioServizo]
})
export class AppModule {}
```

- **O decorador `@Inject`:**
  - Emprégase para indicar explicitamente que unha dependencia debe ser inxectada, especialmente en casos onde a inxección de dependencias non é quen de resolver automaticamente a mesma.
  - Vexamos un exemplo:

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

constructor(@Inject('API_URL') private apiUrl: string) {}
```

Tipos de inxeccións de dependencias

Do mesmo xeito que ocorre cos servizos, as inxeccións de dependencias adoitan clasificarse segundo o ámbito no que son de aplicación, tendo:

- **Ámbito global (`providedIn: 'root'`):**
  - A dependencia é compartida en toda a aplicación.
  - É a configuración por defecto para os servizos.
- **Ámbito local a un módulo:**
  - A dependencia é única para cada módulo onde estea rexistrada nos `providers`.
  - Velaquí un pequeno exemplo:

```
@NgModule({
  providers: [UsuarioServizo]
})
export class UsuarioModulo {}
```

- **Ámbito local a unha compoñente:**
  - Cada instancia da compoñente recibe unha nova instancia da dependencia.

- Vexamos un exemplo deste xeito de operar:

```
@Component({
  selector: 'app-usuario',
  providers: [UsuarioService],
  template: '<p>Compoñente Usuario</p>'
})
export class UsuarioComponent {}
```

Exemplo dunha inxección de dependencias

O exemplo seguinte ilustra como un servizo é inxectado nunha compoñente. Primeiro temos o código do servizo:

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

@Injectable({
  providedIn: 'root'
})
export class UsuarioServizo {
  obterUsuarios(): string[] {
    return ['Xoán', 'Ana', 'María'];
  }
}
```

E agora o código da compoñente que usa o tal servizo a través dunha inxección:

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

@Component({
  selector: 'app-usuario',
  template: `
    <ul>
      <li *ngFor="let usuario of usuarios">{{ usuario }}</li>
    </ul>
  `
})
export class UsuarioComponent implements OnInit {
  usuarios: string[] = [];

  constructor(private usuarioServizo: UsuarioServizo) {}

  ngOnInit() {
    this.usuarios = this.usuarioServizo.obterUsuarios();
  }
}
```

Inxección personalizada

A **inxección personalizada** permite configurar como Angular crea ou resolve unha dependencia específica. Isto é útil cando queremos usar clases alternativas, valores estáticos, instancias creadas dinamicamente ou mesmo personalizar o comportamento dunha dependencia en función de certas condicións.

Tipos de Inxección Personalizada

- **Substitución de clases (`useClass`):**
  - Permite indicar unha clase alternativa que Angular debe usar en lugar da especificada orixinalmente.
  - Cando se usa:
    - Cando queremos substituír unha implementación por outra (por exemplo, en función do contorno -produción vs. Desenvolvemento-).
    - Cando queremos empregar unha versión estendida dunha clase.
  - Supoñamos, co seguinte exemplo, que temos dúas clases de servizo para autenticar usuarios (unha real e outra simulada):

```
export class AuthServizo {
  autenticar() {
    return 'Autenticación por defecto';
  }
}

export class AuthSimuladaServizo {
  autenticar() {
    return 'Autenticación simulada';
  }
}
```

Ao configurar o módulo do xeito que segue estaremos a usar, en toda a aplicación, `AuthSimuladaServizo` no canto de `AuthServizo`.

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

@NgModule({
  providers: [
    { provide: AuthServizo, useClass: AuthSimuladaServizo }
  ]
})
export class AppModule {}
```

- **Provisión dun valor estático (`useValue`):**
  - Permite inxectar un valor constante como dependencia. Este valor non será modificado durante o ciclo de vida da aplicación.
  - Cando se usa:
    - Cando precisamos un valor constante ou configuración global.
    - Cando queremos evitar a creación dunha clase para dependencias simples.

- No seguinte exemplo vemos como fornecer a URL base dunha API. Comezamos polo módulo:

```
@NgModule({
  providers: [
    { provide: 'API_URL', useValue: 'https://api.example.com' }
  ]
})
export class AppModule {}
```

E continuamos coa compoñente `ApiComponent`, a cal poderá acceder ao valor `'https://api.example.com'` como dependencia:

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

@Component({
  selector: 'app-api',
  template: `<p>API URL: {{ apiUrl }}</p>`
})
export class ApiComponent {
  constructor(@Inject('API_URL') public apiUrl: string) {}
}
```

- **Creación dinámica de dependencias (`useFactory`):**
  - Usa unha función para crear a dependencia, permitindo configurar ou inicializala dinamicamente.
  - Cando se usa:
    - Cando a dependencia require configuración baseada en parámetros ou estado.
    - Cando a creación da dependencia depende doutros servizos.
  - No seguinte exemplo créase un servizo de configuración que depende da contorna:

```
export function configFactory(environment: string) {
  return environment === 'producción'
    ? 'https://api.prod.example.com'
    : 'https://api.dev.example.com';
}
```

A configuración no módulo sería a seguinte:

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

@NgModule({
  providers: [
    { provide: 'API_URL', useFactory: () => configFactory('producción') }
  ]
})
export class AppModule {}
```

O uso é análogo a `useValue`. O resultado é que o valor da dependencia cambiará en función da contorna especificada na función `configFactory`.

- **Reutilización de instancias (`useExisting`):**
  - Permite que un *token* (identificador) reutilice a mesma instancia doutro servizo ou clase.
  - Cando se usa:
    - Cando queremos que varias dependencias compartan a mesma instancia.
    - Cando desexamos evitar a duplicación de obxectos.
  - O seguinte exemplo ilustra como compartir a mesma instancia entre dous *tokens*. Primeiro vexamos o código do servizo:

```
export class LoggerService {
  log(mensaxe: string) {
    console.log(mensaxe);
  }
}
```

Agora o módulo:

```
@NgModule({
  providers: [
    LoggerService,
    { provide: 'CONSOLE_LOGGER', useExisting: LoggerService }
  ]
})
export class AppModule {}
```

E finalmente a compoñente:

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

@Component({
  selector: 'app-log',
  template: `<p>Ver consola para os logs.</p>`
})
export class LogComponent {
  constructor(@Inject('CONSOLE_LOGGER') private logger: LoggerService) {
    this.logger.log('Isto é un log desde CONSOLE_LOGGER.');
  }
}
```

Comparación dos métodos de inxección

Método	Funcionalidade	Uso
`useClass`	Substitúe unha clase por outra	Para usar implementacións alternativas ou simulacións (*mocking*).

Método	Funcionalidade	Uso
`useValue`	Un valor constante	Para configuración global ou valores simples.
`useFactory`	Unha dependencia configurada dinamicamente	Cando a dependencia require parámetros ou configuración dinámica.
`useExisting`	Reemprega unha instancia existente	Para compartir obxectos entre diferentes *tokens*.

Primeiros pasos

Neste capítulo veremos os pasos a seguir para crear un primeiro proxecto de Angular ao que, posteriormente, engadiremos novas compoñentes.

No noso caso, empregaremos un novo cartafol chamado `Proxectos`, dentro do cal crearemos todos os nosos proxectos (se ben isto non é obrigatorio, e os mesmos poden ser situados en calquera lugar que desexemos).

Creación dun proxecto Angular

Situados no cartafol de proxectos, procedemos a crear un novo proxecto `ola-mundo` mediante a seguinte orde:

```
ng new OlaMundo
```

Se quixeramos crear unha aplicación modular, no canto de *standalone*, é dicir, tal e coma se traballaba nas versións anteriores de Angular, teriamos que empregar o seguinte comando:

```
ng new ola-mundo --no-standalone
```

Se quixeramos que non se crearan os ficheiros de probas unitarias para cada novo elemento TS da aplicación engadiriamos:

```
ng new ola-mundo –skip-tests
```

A continuación vánsenos presentar un par de preguntas:

- **Queres habilitar o autocompletado?**
    - Respostar SI a esta opción implica que os comandos de Angular CLI se autocompletarán ao premer TAB na terminal (coma calquera comando ordinario de GNU/Linux).

```
administrador@UBUNTUDAW:~/Proxectos$ ng new OlaMundo
 Would you like to enable autocompletion? This will set up your terminal so pressing TAB
while typing Angular CLI commands will show possible options and autocomplete arguments.
(Enabling autocompletion will modify configuration files in your home directory.) (Y/n)
```

- **Queres compartir datos de uso do proxecto con Google?**
    - Respostar SI implicará que eses datos usaranse coma *feedback* para a resolución de *bugs* e o desenvolvemento de novas versións do *framework* por parte da equipa de Angular.

```
Would you like to share pseudonymous usage data about this project with the Angular Team
at Google under Google's Privacy Policy at https://policies.google.com/privacy. For more
details and how to change this setting, see https://angular.dev/cli/analytics.

  (y/N)
```

- **Que tipo de estilos queremos empregar?**
    - Isto vai depender da tecnoloxía visual escollida (CSS, SCSS, Sass Indentado ou Less) aínda que, no noso caso, ficaremos con CSS estándar.

```
? Which stylesheet format would you like to use? (Use arrow keys)
❯ CSS             [ https://developer.mozilla.org/docs/Web/CSS                ]
  Sass (SCSS)     [ https://sass-lang.com/documentation/syntax#scss           ]
  Sass (Indented) [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less            [ http://lesscss.org                                        ]
```

- **Queres habilitar o prerrenderizado *Server-Side Rendering* (SSR) e *Static Side Generation* (SSG)?**
    - SSR (*Server-Side Rendering*) é o proceso de renderizar unha aplicación de lado cliente completamente no servidor, no canto de no navegador. Só o resultado final do renderizado é enviado ao navegador do cliente. Isto pode mellorar o SEO da nosa páxina, pero tamén incrementa a carga do servidor.
    - SSG (*Static Side Generation*) é o proceso de crear un sitio Web como conxunto de ficheiros HTML estáticos, no canto de como unha aplicación dinámica. Isto é útil se o noso sitio non require actualizacións frecuentes nin implica moita interacción co usuario.

```
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation
(SSG/Prerendering)? (y/N)
```

Estrutura dun proxecto Angular

A resultas da creación do proxecto mediante a orde vista no punto anterior (`ng new`) aparecerá un directorio co nome do proxecto dentro do cal distinguiremos os seguintes elementos:

- **Cartafol `node_modules`:**
    - Contén todas as dependencias de terceiros instalados mediante `npm`.
    - Non se edita directamente, senón que as súas dependencias son especificadas a través do ficheiro package.json.

- **Cartafol `public`:**
    - Contén imaxes e outros elementos que servirán como ficheiros estáticos para o servidor de desenvolvemento.
    - Os elementos contidos nel serán copiados á aplicación definitiva durante a compilación.
- **Cartafol `src`:**
    - Constitúe o directorio principal do noso proxecto no que poderemos atopar os ficheiros de código fonte.
- **Ficheiro `angular.json`:**
    - É o ficheiro de configuración principal do proxecto.
    - Controla aspectos como a compilación, o *linting*[6], as probas e a configuración dos estilos e *scripts*.
- **Ficheiro `package.json`:**
    - Contén a listaxe de dependencias do proxecto e *scripts* de comando.
    - Consta de tres seccións clave:
        - `dependencies`: dependencias precisas para executar a aplicación.
        - `devDependencies`: dependencias precisas para o desenvolvemento.
        - `scripts`: comandos para xestionar o proxecto.
- **Ficheiro `server.ts`:**
    - Punto de entrada para o servidor que executa a aplicación Angular no lado do servidor.
    - Funcións:
        - Configura o servidor HTTP.
        - Configura o renderizado SSR usando o `renderModule` de Angular.
        - Serve recursos estáticos como imaxes ou ficheiros CSS.
        - Define rutas para servir as vistas da aplicación.
- **Ficheiro `tsconfig.json`:**
    - Define as opcións do compilador, como o soporte a ECMAScript, a configuración dos módulos e as rutas aos distintos ficheiros.

O cartafol src

No interior deste cartafol atoparemos:

- **Cartafol `app`:**
    - Raíz de todos os módulos, compoñentes e servizos que iremos creando dentro do noso proxecto.

[6] O *linting* é o chequeo automático do código na procura de erros programáticos e estilísticos.

- Dentro deste último cartafol atoparemos unha serie de ficheiros que nos serán de moita relevancia:
  - **`app.component.html`:**
    - Representa o contido HTML da compoñente principal da aplicación, sendo o elemento base empregado por Angular para realizar a ligazón de datos (*data binding*).
  - **`app.component.css`:**
    - Representa os estilos CSS da nosa compoñente principal.
  - **`app.component.ts`:**
    - É o arquivo de TypeScript fundamental da compoñente principal, xa que inclúe a lóxica do mesmo. A páxina pódese entender coma unha unha árbore de compoñentes, onde a principal actúa coma raíz.
  - **`app.component.spec.ts`:**
    - Trátase do ficheiro que permite configurar os test de proba unitarios na compoñente principal. Nesta actividade non imos empregalo pero é importante saber que fariamos uso deste tipo de arquivos mediante Angular CLI a través do comando:

```
ng test
```

  - **`app.module.ts`:**
    - Este ficheiro TypeScript inclúe as dependencias da nosa Web. Ten coma obxecto a importación dos módulos necesarios, a declaración das compoñentes a empregar e a compoñente principal que será arrincada (este ficheiro non aparece en proxectos *standalone*, que é o xeito por defecto de traballo de Angular dende a versión 17).
  - **`app.config.server.ts`:**
    - Contén a configuración específica para o lado servidor (define o comportamento da aplicación cando se executa no servidor).
    - Almacena:
      - URLs de APIs específicas para o servidor.
      - Garda a configuración de seguridade.
      - Garda configuracións de módulos e servizos que só son precisos no SSR.
  - **`app.config.ts`:**
    - Contén a configuración xeral da aplicación, independentemente de se executa no lado do cliente ou do servidor.

- Actúa coma punto central para a xestión de parámetros, proveedores e outras configuracións globais.
- Almacena:
    - URLs de APIs xerais.
    - Configuración de temas ou estilos.
    - Proveedores globais e módulos comúns.

▪ **`app.routes.ts`:**

- Define as rutas da aplicación Angular, as cales especifican como se navega entre as distintas vistas e compoñentes.
- Almacena:
    - As definicións das rutas principais.
    - A configuración das rutas con parámetros ou rutas dinámicas.
    - A configuración de *lazy loading* para cargar módulos de xeito diferido (en aplicacións modulares).

- **Cartafol `assets`:**
    - Almacena recursos estáticos coma imaxes, fontes, vídeos, ficheiros de configuración etc.
    - Os ficheiros contidos no mesmo serán incluídos na compilación e accesibles dende a URL base.
    - Dende Angular 18 este cartafol non é creado por defecto (aínda que pode crearse manualmente), podéndose empregar para tal función o cartafol `public` do directorio raíz do proxecto.
- **Ficheiro `index.html`:**
    - Ficheiro HTML principal da aplicación, é o punto de entrada onde Angular insire a súa vista a través da carga da compoñente raíz (`app.component`).
- **Ficheiro `main.server.ts`:**
    - Actúa coma punto de entrada da aplicación Angular no lado do servidor.
    - En aplicacións modulares inicializa o módulo raíz da aplicación para SSR.
    - Integra a aplicación Angular co servidor HTTP que executará a mesma.
- **Ficheiro `main.ts`:**
    - Ficheiro TS principal dende o que Angular arrinca a aplicación e arrinca o módulo raíz (en aplicacións modulares).
- **Ficheiro `styles.css`:**
    - Ficheiro de estilos globais da aplicación.
    - Todas as compoñentes herdan por defecto estes estilos.

Despregamento de aplicacións Angular

O despregamento dunha aplicación Angular implica poñela en funcionamento nun servidor ou contorna específica para ser accesible por usuarios ou para probas internas. Este proceso, coma o de calquera outro proxecto de *software* profesional, inclúe dúas fases principais:

1. **Contorna de desenvolvemento:** proba e visualización da aplicación localmente durante o desenvolvemento.
2. **Contorna de produción:** creación dunha versión optimizada da aplicación para a súa distribución final e explotación.

A estrutura modular de Angular facilita o despregamento tanto en fase de desenvolvemento como en produción, aproveitando ferramentas integradas en Angular CLI.

Contorna de desenvolvemento

O **despregamento en contorna de desenvolvemento** permite probar a aplicación en tempo real namentres se realizan cambios no código. Angular fornece un servidor de desenvolvemento a través do comando `ng serve`, que recompila automaticamente os cambios e os aplica á aplicación en tempo de execución.

A continuación veremos os comandos básicos relacionados con esta fase:

- Para **executar a aplicación no servidor de desenvolvemento** lanzaremos, dende o cartafol raíz do proxecto, o seguinte comando:

```
ng serve
```

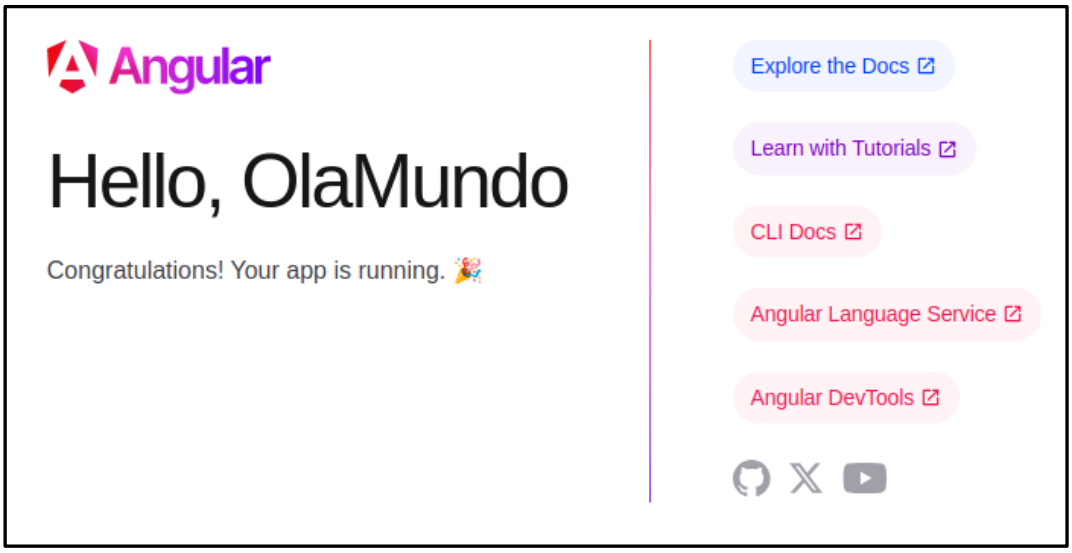
- Se además queremos **abrir automáticamente a aplicación no navegador por defecto** lanzaremos:

```
ng serve -o
```

- Se quixeramos **indicar un porto distinto do porto por defecto** (4200) usaríamos:

```
ng serve --port 4300
```

Por defecto, a pantalla de benvida dun novo proxecto en Angular 18 é a seguinte:



Contorna de produción

O **despregamento en contorna de produción** implica xerar unha versión optimizada da aplicación lista para ser servida a través dun servidor Web definitivo. Isto conséguese empregando o comando `ng build`, que produce una aplicación HTML5, CSS3 e JS contida no cartafol `dist`.

Estes son os comandos fundamentais empregados nesta fase:

- Para **compilar[7] a aplicación** executamos, no directorio raíz do proxecto:

```
ng build
```

- Para **producir unha versión de produción** executamos:

```
ng build –configuration production
```

- Tamén podemos **especificar o cartafol base da aplicación** (útil para subdirectorios):

```
ng build --base-href="/mi-aplicacion/"
```

Os ficheiros xerados no cartafol `dist` pódense subir a calquera servidor Webque soporte ficheiros estáticos, como por exemplo Apache, Nginx, Firebase Hosting ou GitHub Pages.

O despregamento de aplicacións Angular require coñecer as diferenzas entre as contornas de desenvolvemento e produción. A través de comandos sinxelos e ferramentas integradas, Angular facilita a creación e distribución de aplicacións optimizadas para distintas necesidades. Adaptar a

[7] En realidade «transpilar», xa que o que faremos con esta acción será traducir a aplicación TypeScript ao seu equivalente JavaScript.

configuración ás particularidades do proxecto pode garantir mellores resultados tanto en probas como en produción.

Inicio e carga dun proxecto en Angular

Unha vez creado o proxecto (con `ng new`) e lanzado no servidor (con `ng serve`) procedemos a entender como está cargando esa páxina de inicio por defecto. Con ese fin tómanse os seguintes pasos:

1. **Carga do ficheiro `index.html`:**
    - O navegador carga o ficheiro raíz `index.html`, que contén:
        - A referencia ao script principal (`main.ts`) que inicializa a aplicación.
        - O punto de montaxe onde Angular renderiza a interface (`<app-root>` por defecto).
2. ***Bootstrap* do módulo raíz (`AppModule`):**
    - Angular inicializa o módulo raíz (`AppModule`) definido no ficheiro `main.ts` e carga o compoñente raíz (`AppComponent`).
3. **Renderización e visualización inicial.**
    - Angular usa o motor Ivy para compilar e renderizar a compoñente raíz na etiqueta `<app-root>` do `index.html`.

Como está a funcionar isto?

Se observamos o contido do ficheiro **`index.html`**, situado no directorio `src` do proxecto veremos o seguinte código HTML:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>OlaMundo</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Destacamos no documento anterior a etiqueta `app-root`, xa que fai referencia á compoñente principal da aplicación. Esa etiqueta defínese no ficheiro **`app-component.ts`**, cuxo contido é o seguinte:

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'Ola-Mundo';
}
```

Podemos ver como dentro do «decorador» `@Component` defínese unha propiedade `selector`, a cal leva coma valor o nome da etiqueta que se usará en HTML para facer referencia a esta compoñente.

Cando se lanza unha aplicación de Angular mediante o comando `ng serve -o` o primeiro que se fai é crear os *bundles* (que podemos traducir como paquetes) en tempo de execución. Dende eses *bundles* o primeiro código en executarse é o TypeScript contido no ficheiro **`main.ts`**. É neste ficheiro onde comeza a execución de calquera proxecto en Angular. Velaquí o contido do noso:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

Dado que dende Angular 17 o modo *standalone* (é dicir, de aplicación sen módulos) é o módulo por defect, non atoparemos o arquivo **`app.module.ts`** que podiamos ver en versións anteriores, e que adoitaba ter unha forma semellante á seguinte:

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
```

```
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

En resumo, a execución da nosa aplicación de Angular 18 vai accedendo aos ficheiros que figuran a continuación, en orde, dende que é lanzada lanza até que chega a visualizarse a vista inicial:

1. main.ts
2. ~~app.module.ts~~
3. app.component.ts
4. index.html
5. app.component.html

Optimizacións no proceso de carga

- ***Lazy loading*:** permite cargar módulos de forma diferida, reducindo o tempo de inicio inicial.
- ***Preloading*:** mellora a experiencia cargando módulos en segundo plano.

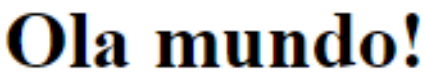
O noso primeiro proxecto en Angular

Para introducirnos ao desenvolvemento con Angular imos crear un primeiro proxecto e, como non pode ser doutro xeito, faremos o clásico **Ola Mundo!**:

1. Accedemos ao arquivo **`app.component.html`**:
2. Borramos todo o seu contido (podemos deixar unicamente a etiqueta router-outlet se queremos conservar o enrutado dinámico de Angular –do que falaremos posteriormente-).
3. Incorporamos simplemente unha cabeceira de exemplo:

```
<h1>Ola mundo!</h1>
<router-outlet></router-outlet>
```

4. Visualizando a nosa aplicación no navegador veremos o resultado da modificación anterior:



Compoñentes

En Angular, as **compoñentes** son a unidade básica para o desenvolvemento da interface de usuario. Cada compoñente encapsula unha parte da lóxica, a estrutura e os estilos da aplicación, facilitando a modularidade, a reutilización do código e a sustentabilidade.

Neste capítulo veremos dúas formas de crear compoñentes en Angular: manualmente e mediante un comando específico.

Creación de compoñentes

É habitual crear un directorio chamado **`components`** dentro de **`src/app`** para organizar as compoñentes do noso proxecto. Dentro dese directorio, recomendamos separar cada compoñente no seu propio cartafol. Esta práctica axuda a manter o proxecto limpo e ben estruturado.

Creación manual de compoñentes

Para crear unha compoñente manualmente tomamos os seguintes pasos:

1. Crea un novo cartafol en `src/app/components` chamado `nova-componhente`.
2. Dentro dese cartafol, crea un ficheiro chamado **`nova-componhente.component.ts`**.
3. Engade o seguinte código fonte no ficheiro:

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

@Component({
    selector: 'nova-componhente',
    standalone: true,
    imports: [],
    template: '<p>O contido da nova compoñente simplemente é un parágrafo.</p>',
    styles: ['p { font-weight: bold; color: green; }']
})
export class NovaComponhenteComponent {}
```

En proxectos máis grandes, separar o HTML e CSS en ficheiros independentes é unha boa práctica. Para facelo tomariamos estes pasos:

4. Crea os ficheiros **`nova-componhente.component.html`** e **`nova-componhente.component.css`** no cartafol da compoñente.
5. Actualiza o decorador para referenciar eses ficheiros:

```
@Component({
    selector: 'nova-componhente',
    standalone: true,
    templateUrl: './nova-componhente.component.html',
    styleUrls: ['./nova-componhente.component.css']
})
```

Finalmente, para poder usar a compoñente no proxecto, precisamos incluíla no decorador `imports` doutra compoñente ou módulo. No noso caso haberá un único paso:

6. Modificamos o *script* da compoñente raíz (**`app.component.ts`**):

```
import { NovaComponhenteComponent } from './components/nova-componhente/nova-
componhente.component';

@Component({
    selector: 'app-root',
    standalone: true,
    imports: [NovaComponhenteComponent],
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {}
```

Creación automática de compoñentes

Angular CLI permite crear compoñentes automaticamente empregando comandos específicos, o que simplifica o proceso. Velaquí os pasos:

1. Para crear unha compoñente chamada nova-componhente2 no cartafol **components**, executa:

```
ng generate component components/nova-componhente2
```

Ou, de forma abreviada:

```
ng g c components/nova-componhente2
```

O comando anterior creará automaticamente os seguintes ficheiros no cartafol correspondente:

- **`nova-componhente2.component.ts`**
- **`nova-componhente2.component.html`**
- **`nova-componhente2.component.css`**
- **`nova-componhente2.component.spec.ts`** (ficheiro de probas unitarias)

Ademais, a compoñente queda rexistrada automaticamente no proxecto, polo que podes usar o seu selector sen pasos adicionais.

Erros e problemas habituais

A continuación algún dos erros e problemas que atoparemos habitualmente ao empregar compoñentes en Angular:

- **Erro: "Ficheiro ou estrutura de compoñente incompleta".**
  - **Causa:** o comando Angular CLI para crear a compoñente non se executou correctamente ou non se indicou o camiño desexado.
  - **Solución:** asegúrate de usar o comando correcto (e verifica que todos os ficheiros da componente foron creados correctamente):

```
ng generate component nome_da_compoñente
```

- **Erro: "Erro: 'nome-compoñente' is not a known element".**
  - Causa: a compoñente non foi declarada no módulo correspondente.
  - Solución: asegúrate de engadir a compoñente no vector `declarations` do módulo ou compoñente onde será empregada.

```
@NgModule({
  declarations: [NomeCompoñente],
  imports: [...],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

- **Erro: "Error: Cannot declare 'NomeCompoñente' in multiple modules".**
  - Causa: a compoñente foi declarada en máis dun módulo.
  - Solución: declara o compoñente nun único módulo ou utiliza un módulo compartido para exportalo.
- **Erro: "O compoñente non se renderiza no HTML".**
  - Causa: `selector` mal escrito ou non usado correctamente no HTML.
  - Solución: comproba o valor do `selector` no decorador `@Component` e usa o mesmo no ficheiro HTML.

```
@Component({
  selector: 'app-nome-compoñente',
  templateUrl: './nome-compoñente.component.html',
  styleUrls: ['./nome-compoñente.component.css']
})
```

  - O uso correcto no HTML sería:

```
<app-nome-compoñente></app-nome-compoñente>
```

- **Erro: "Datos non dispoñibles na vista ao cargar o compoñente".**
    - Causa: os datos necesarios non se cargaron antes de renderizar o compoñente.
    - Solución: usa o *hook* `ngOnInit` para realizar a inicialización necesaria.

```
ngOnInit() {
  this.cargarDatos();
}
```

- **Erro: "Os estilos definidos non se aplican ao compoñente".**
    - Causa: o estilo foi definido de maneira global ou non está ben configurado no `styleUrls`.
    - Solución: asegúrate de que os estilos estean definidos no ficheiro CSS asociado e que `styleUrls` apunte correctamente a ese ficheiro.

```
styleUrls: ['./nome-compoñente.component.css']
```

- **Erro: "Problemas co cambio dinámico de contido na compoñente".**
    - Causa: cambios de estado mal xestionados ou non detectados.
    - Solución: usa `ChangeDetectorRef` para forzar a detección de cambios se é necesario.

Directivas

As **directivas** de Angular son un dos elementos máis poderosos do *framework*, permitindo manipular o DOM de xeito modular e eficiente. Con elas, podemos modificar a estrutura do DOM, alterar a aparencia dos elementos ou engadir lóxica personalizada ao noso código baixo un nome predefinido.

Creación de Directivas

Para crear unha directiva personalizada empregamos o seguinte comando:

```
ng generate directive nome_da_directiva
```

Ou, empregando un *shorthand* propio de Angular CLI:

```
ng g d nome_da_directiva
```

Calquera dos comandos anteriores xera un ficheiro TypeScript cuxa finalidade é a de implementar a lóxica da directiva asegurando a súa integración no proxecto.

Tipos de directivas

En función do seu obxectivo destacamos tres tipos de directivas en Angular:

- **Directivas de compoñentes:** atópanse na clase principal e conteñen os detalles de como se procesa, instancia ou emprega a compoñente en tempo real.
- **Directivas estruturais:** empréganse para manipular e cambiar a estrutura da árbore DOM. Destacamos tres das máis usuais:
    - `*ngIf`: permite engadir ou eliminar un elemento DOM en función dunha condición lóxica.
    - `*ngFor`: repite un determinado elemento HTML por cada elemento existente nunha lista iterable.
    - `*ngSwitch`: permite engadir ou eliminar un elemento DOM en función dunha serie de valores que pode tomar unha variable.
    - `ngPlural`: permite mostrar ou ocultar elementos en función dun valor numérico.
    - `ngTemplate`: permite definir bloques de contido reutilizables sen renderizalos inmediatamente no DOM.

    - `ngComponentOutlet`: permite renderizar compoñentes dinámicamente no DOM. Vincúlase a unha compoñente ou clase que debe ser instanciada e inserido no lugar onde se aplique a directiva.
- **Directivas de atributos:** cambian a vista e o comportamento dos elementos DOM. As dúas máis habituais son:
    - `ngClass`: engade ou elimina clases CSS dun elemento HTML
    - `ngStyle`: modifica o estilo dun elemento HTML empregando unha expresión que inclúe todos os cambios a realizar. Desta forma podemos cambiar dinamicamente o estilo dun elemento.
    - `ngModel`: implementa o enlace bidireccional de datos entre un compoñente e un formulario.

Directivas de Compoñentes

Estas directivas inclúen *templates* (modelos) e encapsulan a lóxica dunha compoñente.

Vexamos un exemplo deste tipo de directivas:

- Nunha banda tempos o *script* da compoñente raíz (**`app.component.ts`**):

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Aplicación Angular';
}
```

- Noutra tempos o *template* da compoñente raíz (**`app.component.html`**):

```
<h1>Benvido a {{ title }}</h1>
```

Directivas estruturais

Estas directivas alteran a estrutura do DOM, engadindo e eliminando elementos na mesma, así como manexando coleccións para producir elementos DOM iterativamente.

A directiva `*ngIf`

Esta directiva é, xunto con `*ngFor`, posiblemente a máis empregada en Angular. Coma xa mencionamos con anterioridade, permite engadir ou eliminar un elemento HTML do DOM en base ao valor dunha expresión lóxica.

Para probar esta directiva imos aproveitar e usar a compoñente raíz **app** sobre a cal faremos as probas correspondentes.

Dentro do ficheiro **app.component.ts** podemos ver o nome que lle da ao selector que vai permitir empregar esa compoñente dende calquera vista en Angular. O selector correspondente (tamén coñecido coma «etiqueta»), neste caso, é a `app-root`.

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'ngIf';
}
```

Imos modificar a vista HTML **app.component.html** para probar a directiva `*ngIf` de xeito que permita amosar un elemento HTML en base a se o usuario fai clic nun botón ou non. Para iso imos reemprazar o seu contido co seguinte código:

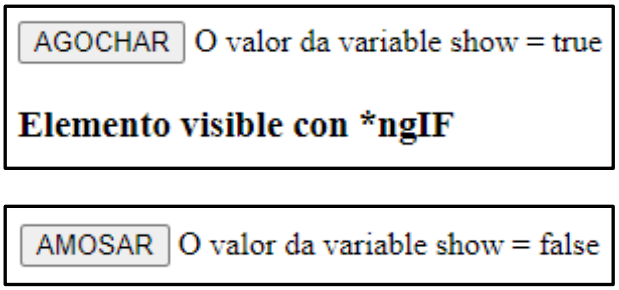
```
<button (click)="amosar = !amosar  ">
    {{amosar ? 'AGOCHAR' : 'AMOSAR'}}
</button>

O valor da variable amosar = {amosar}}<br>

<h3 *ngIf="amosar">Elemento visible con *ngIF</h3>
```

Coma pode verse, o que fixemos foi incorporar ao modelo anterior un botón no que se captura o evento de clic do usuario para, seguidamente, executar un código asociado ao mesmo. O enlace de eventos verémolo en profundidade máis adiante. O código o único que fai é gardar o valor da variable lóxica `show` negado (variable que estará na lóxica da compoñente ou, o que é o mesmo, no seu TypeScript asociado). É dicir, se `show` é verdadeiro ponse a falso e senón ao revés cando se fai un clic. No texto do botón temos as opcións de etiqueta `AGOCHAR` ou `AMOSAR`, escollidas en función do valor desa variable. A maiores amósase o valor da variable mediante o enlace de datos coas chaves `{{}}` (este enlace de datos tamén será analizado máis tarde en profundidade). Finalmente na etiqueta `<h3>` é onde facemos uso da nosa directiva `*ngIf`, de xeito que se amose ou non o propio elemento en función do valor de `amosar`.

Retornamos ao ficheiro **app.component.ts** para incluír a declaración da variable `amosar`, o cal realizaremos dentro da clase que lle corresponde, producíndose a inicialización dentro do construtor (neste caso co valor de verdadeiro).

```
…
export class ApComponent {
  title = 'ngIf';
  amosar: boolean;

  constructor() {
    this.amosar=true;
  }
…
```

Agora podemos visualizar no noso navegador o resultado nas súas dúas formas:



A directiva `*ngFor`

Esta directiva permite repetir unha serie de elementos HTML tantas iteracións como teña o percorrido dunha lista iterable (`Collection`). Existen unha serie de variables locais tales como `Index`, `First`, `Last`, `odd` e `even` que son exportadas por esta directiva e que posibilitan e asinxelan notablemente o seu manexo.

Facendo uso da compoñente raíz (**app**), imos crear un exemplo cun *array* definido no arquivo TypeScript no cal introduciremos catro elementos. Estes elementos vanse amosar no modelo HTML facendo uso de `*ngFor`. A maiores incluiremos no noso arquivo CSS algúns nomes de clases para introducir tamén o uso de `ngClass`, o cal vai posibilitar incorporar ou eliminar novas clases CSS nos elementos HTML.

No arquivo **app.component.ts** definimos a listaxe de elementos, que será inicializada no construtor:

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
```

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'ngFor';
  listaxe: string[];

  constructor() {
    this.listaxe = ["DAW", "DAM", "ASIR", "SMIR"];
  }
}
```

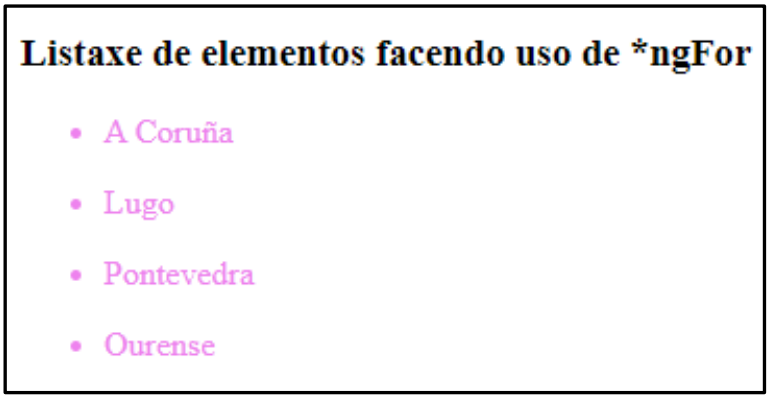
Incluímos agora algunha clase no ficheiro CSS **`app.component.css`** para modificar a cor de texto dalgún dos elementos:

```
.textoVioleta {
    color: violet;
}

.textoVerde {
    color: green;
}
```

Pola súa parte, no arquivo **`app.component.html`** empregamos `*ngFor` indicando un nome de variable `item` que en cada iteración acollerá un elemento e permitirá introducilo como un novo `<li>` usando a clase `textoVioleta` (a clase `textoVerde`) será empregada en exemplos posteriores.

```
<h1>Listaxe de elementos facendo uso de *ngFor</h1>
<ul>
  <li *ngFor="let provincia of listaxe" ngClass="textoVioleta">
    {{provincia}}
  </li>
</ul>
```

Ao lanzarmos a aplicación, o resultado será algo así:



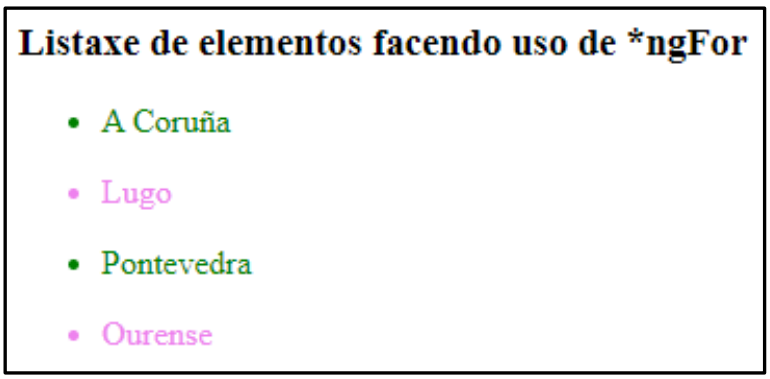
Podemos levar o noso exemplo un bocadiño alén, empregando algunha das variables propias que exporta esta directiva. Por exemplo, se queremos que os elementos pares teñan unha cor e os

impares outra, teremos que empregar as variables `odd` e `even` para impares e pares respectivamente.

Modificando o noso código **`probasngfor.component.html`** por:

```
<h3>Listaxe de elementos facendo uso de *ngFor</h3>
<ul>
  <li
*ngFor="let provincia of listaxe; let impar=odd; let par=even;" [ngClass]="{textoV
ioleta: impar, textoVerde: par}">
    {{provincia}}
  </li>
</ul>
```

Veremos un resultado semellante ao seguinte:



En cada iteración gárdanse, polo tanto, dúas variables (par e impar) de tipo booleano que son avaliadas por `ngClass`, que indicará que clase aplicará en función das mesmas. Se nos fixamos, aparece un elemento imprescindible que son os corchetes, os cales conteñen a directiva `ngClass`, e encárganse de facer o *binding*, indicando se existe algún elemento debe ser substituído polo seu valor (isto será analizado con máis detalle no apartado de enlace de datos).

A directiva `*ngSwitch`

Trátase tamén dunha directiva estrutural, que neste caso permite eliminar e engadir múltiples elementos do DOM en función dos valores que poida tomar unha determinada expresión ou variable.

Imos crear, coma exemplo, unha nova compoñente na cal disporemos dunha variable numérica, de xeito que se o número ten un valor entre 1 e 6, vai amosarse o elemento correspondente dunha listaxe. No caso de que o número sexa outro, indicarase que non se atopa ningún elemento en correspondencia. Ademais do anterior, empregaremos un botón, de xeito que cada vez que se faga clic no mesmo aumente o valor numérico da variable nunha unidade un elemento, coa condición de que se o valor é maior que 6 entón volverá ser 1.

Os elementos principais desta directiva son tres:

- **`[ngSwitch]:`** onde avaliamos o valor dunha expresión ou variable.

- **`*ngSwitchCase:`** que corresponde co caso concreto no que a variable teña ese valor indicado.
- **`*ngSwitchDefault:`** que corresponde co caso por defecto cando o valor da variable non emparellou con ningún dos casos definidos anteriormente.

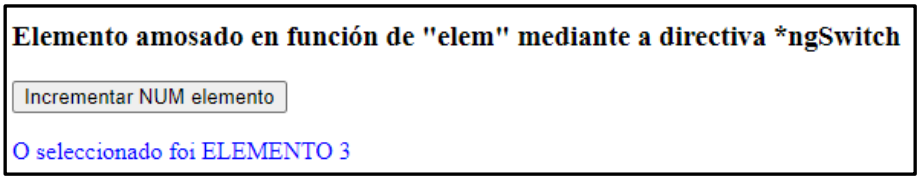
O código da vista HTML (**`app.component.html`**) será o seguinte:

```
<h3>Elemento amosado en función de "elem" mediante a directiva *ngSwitch</h3>

<button (click)="elem>6 ? elem=1 : elem=elem+1">Incrementar NUM elemento</button>

<div [ngSwitch]="elem">
  <p *ngSwitchCase="1" [ngStyle]="{'color': 'green'}">O seleccionado
foi {{listaxe[0]}}</p>
  <p *ngSwitchCase="2" [ngStyle]="{'color': 'red'}">O seleccionado
foi {{listaxe[1]}}</p>
  <p *ngSwitchCase="3" [ngStyle]="{'color': 'blue'}">O seleccionado
foi {{listaxe[2]}}</p>
  <p *ngSwitchCase="4" [ngStyle]="{'color': 'orange'}">O seleccionado
foi {{listaxe[3]}}</p>
  <p *ngSwitchCase="5" [ngStyle]="{'color': 'yellow'}">O seleccionado
foi {{listaxe[4]}}</p>
  <p *ngSwitchCase="6" [ngStyle]="{'color': 'violet'}">O seleccionado
foi {{listaxe[5]}}</p>
  <p *ngSwitchDefault [ngStyle]="{'color': 'gray'}">Non é un elemento da
listaxe</p>
</div>
```

O código do TypeScript (**`app.component.ts`**) é o seguinte:

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'ngSwitch';
  numero : number;
  listaxe : string[];

  constructor() {
    this.numero = 0;
    this.listaxe = ["DAW", "DAM", "ASIR", "SMIR"];
  }
}
```

Un dos resultados intermedios de operar o proxecto anterior será algo semellante a isto:



A directiva `ngPlural`

Renderiza elementos DOM segundo valores numéricos. O anterior acádase avaliando o valor dunha variable ou expresión programática.

Neste exemplo podemos ver como se avalía o valor da variable `numero` para cargar distintos modelos no HTML:

```
<div [ngPlural]="numero">
  <ng-template ngPluralCase="=1">Un elemento</ng-template>
  <ng-template ngPluralCase="other">{{ numero }} elementos</ng-template>
</div>
```

A directiva `ngTemplate`

Esta directiva permite definir modelos que formarán código reutilizable.

Vexamos un exemplo de definición dun modelo e a súa posterior aplicación condicionada a través dunha directiva `*ngIf`.

```
<ng-template #modelo>
  <p>Este contido pode ser reutilizado.</p>
</ng-template>

<div *ngIf="true; then modelo"></div>
```

A directiva `*ngComponentOutlet`

Esta directiva permite renderizar compoñentes dinamicamente.

De seguido vemos un exemplo práctico:

```
<ng-container *ngComponentOutlet="componente"></ng-container>
```

Directivas de atributos

Estas directivas permiten modificar propiedades e estilos dun elemento HTML.

A directiva `ngClass`

Aplica ou elimina clases CSS dinamicamente.

Agora vexamos un exemplo práctico de uso da mesma:

```
<p [ngClass]="{ 'activo': estadoActivo, 'inactivo': !estadoActivo }">
  Estado: {{ estadoActivo ? 'Activo' : 'Inactivo' }}
</p>
```

A directiva `ngStyle`

Modifica estilos *inline* de maneira condicional.

Vexamos tamén neste caso un exemplo de aplicación:

```
<p [ngStyle]="{ 'color': estado ? 'green' : 'red' }">
  Estado: {{ estado ? 'Activo' : 'Inactivo' }}
</p>
```

A directiva `ngModel`

Implementa o enlace bidireccional entre datos e formularios.

Esta directiva será explicada en detalle no capítulo dedicado ao enlazado de datos.

Directivas Personalizadas

As directivas personalizadas amplían a funcionalidade dos elementos HTML para adaptarse ás necesidades específicas do desenvolvemento.

No seguinte exemplo podemos ver a aplicación dunha directiva personalizada que permite ocultar elementos en base a un valor booleano:

```
@Directive({
  selector: '[appOcultar]'
})
export class OcultarDirective {
  @Input('appOcultar') set mostrar(valor: boolean) {
    this.el.nativeElement.style.display = valor ? 'block' : 'none';
  }

  constructor(private el: ElementRef) {}
}
```

E a súa aplicación a través do *template* HTML:

```
<div [appOcultar]="true">Este elemento está visible.</div>
```

A continuación podemos ver un segundo exemplo de creación de directivas personalizadas. Neste caso creamos unha directiva que resalta un texto ao pasar o rato por riba do mesmo:

- Implementamos a lóxica no ficheiro **`destacar.directive.ts`**:

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

@Directive({
  selector: '[appDestacar]'
})
export class DestacarDirective {
  constructor(private el: ElementRef) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.cambiarCor('yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.cambiarCor('');
  }

  private cambiarCor(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

- Aplicación da directiva:

```
<p appDestacar>Pasa o rato por riba deste texto para resaltalo.</p>
```

Erros e problemas habituais

A continuación algún dos erros e problemas que atoparemos habitualmente ao empregar directivas en Angular:

- **Erro: "Directiva personalizada non funciona".**
  - Causa: `selector` mal definido no decorador `@Directive`.
  - Solución: verificar o nome do `selector` e asegurar que está usado correctamente no HTML.
- **Erro: "Non se renderiza unha compoñente con `ngComponentOutlet`".**
  - Causa: compoñente non importado ou mal configurado.
  - Solución: revisar as importacións no módulo ou compoñente raíz.

Enlazado de datos

O **enlazado de datos**, **enlace de datos** ou ***data binding*** é un dos conceptos fundamentais de Angular, e permite definir a comunicación entre as compoñentes e o DOM, asinxelando a definición de aplicacións interactivas sen necesidade de nos preocupar da carga e recepción de datos.

Hai diversos xeitos de enlazar datos en Angular (algúns mesmo posúen variantes), diferenciándose segundo o xeito no que se produce o fluxo de datos:

- **Enlazado unidireccional:** aquel no que o fluxo de datos prodúcese, coma o seu nome indica, nun único sentido.
- **Enlazado bidireccional:** aquel no que o fluxo de datos prodúcese en ambos sentidos entre extremos.

Enlazado unidireccional

O **enlazado unidireccional** ou ***one-way binding*** posibilita o fluxo de datos entre compoñentes e DOM nun único sentido. Distinguimos, segundo ese sentido sexa, dous tipos:

- **Dende a compoñente ao DOM,** velaquí as dúas formas de traballo que hai:
    - Interpolación de *strings*: engade o valor dunha propiedade dende a compoñente, para o cal emprégase a estrutura `{{valor}}`. Por exemplo:

```
<li>Nome: {{ usuario.nome }}</li>
<li>E-mail: {{ usuario.email }}</li>
```

    - Enlazado de propiedades: o valor pásase dende a compoñente á propiedade especificada, que pode ser simplemente un atributo HTML. Neste caso emprégase a estrutura `[propiedade]="valor"`. Vexamos un exemplo:

```
<input type="email" [value]="usuario.email">
```

- **Dende o DOM á compoñente,** conta cun único mecanismo:
    - Enlazado de eventos: chama a un método da compoñente cando se produce un evento DOM (clic, cambio, soltar unha tecla, etc.). Emprega a estrutura `(evento)="método"`. Por exemplo:

```
<button (click)="prepararAlmorzo()"></button>
```

Exemplo de interpolación de *strings*

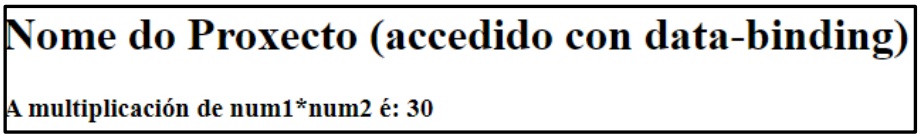
Supoñamos un exemplo de TypeScript (podemos facer uso de **`app.component.ts`**) no que declaramos unha variable de tipo *String* contendo unha certa cadea e outras variables numéricas.

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'interpolacion-strings';
  numero1: number = 12;
  numero2: number = 20;
}
```

Agora, facendo uso da dobre chave podemos substituír o nome desas variables polo seu valor e mesmo facer operacións con elas (como é o caso da operación de multiplicación que observamos no código). Así quedaría o noso código HTML (**`app.component.html`**).

```
<h1> {{ title }} </h1>
<p>O resultado de multiplicar {{ numero1 }} por {{ numero2 }} é {{ numero1*numero2
}} </p>
```

Obténdose un resultado semellante ao seguinte:



Exemplo de enlazado de propiedades

Propoñemos agora un exemplo no que declaramos tres variables (`title`, que contén o título do noso proxecto, `documentacion` que é a URL da documentación de Angular, e `logo` que é a URL do logo de Angular) no noso TypeScript (por exemplo no **`app.component.ts`**):

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
```

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'enlazado-propiedades';
  documentacion: string = 'https://angular.io/docs';
  logo: string = 'https://angular.io/assets/images/logos/angular/logo-nav@2x.png';
}
```

Agora modificamos o arquivo da nosa vista (**`app.component.html`**) para visualizar o título con interpolación de *strings* coma fixemos anteriormente, incorporamos tanto a ligazón á documentación de Angular coma o logo mediante enlazado de propiedades.

```
<h1> {{ title }} </h1>

<a [href]="documentacion">Documentación de Angular</a>
<br>
<img [src]="logo" />
```

Finalmente visualizaremos algo semellante a isto:

Exemplo de enlazado de eventos

Neste exemplo pretendemos capturar o `clic` dun botón para mudar a cor da cabeceira `<h1>`. Os cambios realizados sobre o modelo HTML (**`app.component.html`**) foron os seguintes:

```
<h1 [ngStyle]="{ 'color': getCor() }">
  EXEMPLO DE ENLAZADO DE EVENTOS
</h1>
<button (click)="cambiarCor()">Cambiar cor do título</button>
```

Observamos que conectamos a cabeceira `<h1>` mediante enlazado de propiedades cun método `getCor()` implementado no TypeScript. Ao seu tempo o evento de `clic` do botón asóciase ao método `cambiarCor()`. O arquivo TypeScript **`app.component.ts`** ficará como segue:

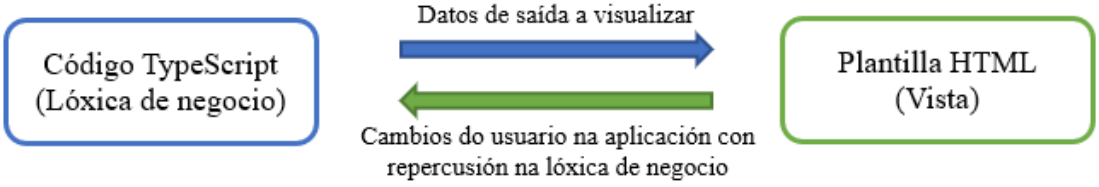
```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'enlazado-eventos';
  corPrincipal: string = "blue";

  // Este método permite mudar a cor principal definida
  cambiarCor(): void {
    if (this.corPrincipal == "blue") {
      this.corPrincipal = "violet";
    } else {
      this.corPrincipal = "blue";
    }
  }

  // Este método devolve a cor principal definida
  getCor(): string {
    return this.corPrincipal;
  }
}
```

O único que temos é unha variable definida co nome `corPrincipal` que muda a cor da cadea que contén en función de se o usuario fai `clic` no botón. Como se pode observar no capturador do evento, o método `cambiarCor()` muda o valor desa variable e amósaa por consola. O resultado será algo semellante a isto (inclúese a saída da consola na captura para entender mellor o funcionamento do exemplo):

Enlazado bidireccional

Todos os métodos que vimos no apartado anterior de enlazado de datos unidireccional permiten facer cambios no modelo (TypeScript) e automaticamente están dispoñibles na vista (HTML). Porén, en ocasións podemos necesitar o contrario, é dicir, que os cambios na vista se

reflexen no TypeScript, o que Angular posibilita a través do **enlazado bidireccional** ou ***two-way binding***.



O enlazado bidireccional de Angular emprega a estrutura `[(ngModel)]="value"`, o que se coñece coma a «sintaxe da banana nunha caixa». No exemplo que sigue podemos ver como os datos da propiedade `usuario.email` son empregados coma valor de entrada pero, se o usuario muda ese valor, a propiedade da compoñente é actualizada en consonancia automaticamente:

```
<input type="email" [(ngModel)]="usuario.email">
```

Para poder facer uso desta funcionalidade é preciso activar a directiva `ngModel` que depende do módulo `FormsModule` (para a interacción co usuario), e que podemos atopar no paquete `angular/forms`. Os cambios que temos que facer no noso código van depender de se traballamos en modo *standalone* (o modo por defecto dende Angular 17) ou se estamos a crear unha app por módulos.

Traballando en modo *standalone*

Unicamente teremos que incorporar `FormsModule` ao vector de importacións de cada compoñente que vaia empregar `[(ngModel)]`.

Traballando por módulos

Os cambios que hai que facer sobre o arquivo **`app.module.ts`** antes de poder empregar `ngModel` son os seguintes:

1. Facer as dúas importacións de `ngModule` e `FormsModule`.
2. Engadir no *array* de `imports` o módulo `FormsModule`.

O resultado será algo semellante ao seguinte:

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

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule // NOVA LIÑA
  ],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule {}
```

Exemplo de enlazado bidireccional

Facemos agora un pequeno cambio con respecto ao exemplo empregado anteriormente. O que queremos é engadir un `<input>` para que o usuario poida cambiar o título da nosa páxina, da nosa cabeceira `<h1>`. Polo tanto, cando se cargue a páxina vaise amosar o valor que teña a cadea de titulo que definimos no TypeScript e, no caso de que o usuario introduza unha nova cadea no correspondente `<input>`, o título modificarase no instante. Este enlace bidireccional é posible grazas a `ngModel`.

A vista HTML (**`app.component.html`**) quedaría como segue:

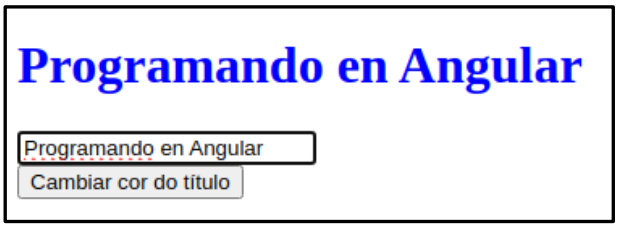
```
<h1 [ngStyle]="{ 'color': getCor() }">
  {{ titulo }}
</h1>
<input type="text" [(ngModel)]="titulo" />
<br>
<button (click)="cambiarCor()">Cambiar cor do título</button>
```

No ficheiro **`app.component.ts`** simplemente engadimos unha variable co título inicial da cabeceira `<h1>`.

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'enlazado-bidireccional';
  titulo: string = "EXEMPLO DE ENLAZADO BIDIRECCIONAL";
  corPrincipal: string = "blue";
```

```
  // Este método permite mudar a cor principal definida
  cambiarCor(): void {
    if (this.corPrincipal == "blue") {
      this.corPrincipal = "violet";
    } else {
      this.corPrincipal = "blue";
    }
  }

  // Este método devolve a cor principal definida
  getCor(): string {
    return this.corPrincipal;
  }
}
```

Tras facer o cambio do título, escribindo no correspondente `<input>` xa se observa que en tempo real é modificada a vista no `<h1>` previo.



Erros e problemas habituais

A continuación algún dos erros e problemas que atoparemos habitualmente ao empregar enlazado de datos en Angular:

- **Erro: "Erro ao empregar `[(ngModel)]`".**
  - Causa: non se importou o módulo `FormsModule`.
  - Solución: engadir `FormsModule` no vector de importacións.
- **Erro: "Erro ao mostrar datos na vista"**
  - Causa: propiedade definida no TypeScript pero non inicializada.
  - Solución: asegúrate de inicializar a propiedade no constructor ou na declaración da clase.

```
export class NomeCompoñente {
  texto: string = 'Texto de exemplo';
}
```

- **Erro: "Uncaught Error: Template parse errors"**
  - Causa: uso incorrecto de *binding* no HTML.
  - Solución: verifica que as expresións empreguen correctamente as chaves `{{ }}` ou os corchetes `[ ]`.

Decoradores

Un **decorador** é un patrón de deseño empregado para modificar o comportamento dun elemento (clase, propiedade, método ou parámetro) sen necesidade de modificar o seu código fonte. Dito doutro xeito, un decorador engade funcionalidade dinamicamente a un obxecto evitando ter que crear subclases.

Mediante o uso de decoradores, as compoñentes son rexistradas, de xeito que poden ser empregadas noutras partes da aplicación.

Tipos de decoradores

Angular contempla catro tipos principais de decoradores:

- **De clase:** unha función que recibe a clase coma parámetro devolvendo seguidamente algo que facer coa mesma. P.ex.: `@Component` e `@NgModule`.
- **De propiedades:** unha función que recibe coma parámetros a clase e o nome da propiedade. P.ex.: `@Input` e `@Output`.
- **De métodos:** unha función que recibe como parámetros a clase, o nome do método e un descritor da propiedade do obxecto[8]. P.ex.: `@HostListener`.
- **De parámetros:** unha función que recibe coma parámetros a clase, o nome do parámetro e a súa posición. P.ex.: `@Inject`.

Coma podemos ver polos exemplos anteriores, os nomes dos decoradores van sempre precedidos dunha arroba (@) seguida do nome do elemento ao cal decoran.

Decoradores de clases

O decorador `@Injectable`

O decorador **`@Injectable`** en Angular úsase para marcar unha clase como dispoñible para a inxección de dependencias no sistema de Angular. Este decorador indica que a clase pode ser inxectada noutros compoñentes, servizos ou módulos.

A continuación o código TS dun exemplo:

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

[8] Un descritor da propiedade do obxecto é un método estático que define unha propiedade directamente sobre un obxecto ou ben modifica directamente unha propiedade existente do obxecto para, a continuación, devolver o devandito obxecto.

```
@Injectable({
  providedIn: 'root'
})
export class ExemploServico {
  obterDatos() {
    return 'Datos desde o servizo.';
  }
}
```

Decoradores de propiedades

O decorador `@Input`

Angular pon á nosa disposición un decorador, **`@Input`**, que nos permite pasar información dende compoñentes nai cara compoñentes fillas (de xeito descendente na xerarquía de compoñentes).

Co gallo diso, márcase unha propiedade/atributo da compoñente nai co devandito decorador pasando a mesma a ser accesible polas compoñentes fillas.

Compoñente nai

Para o seguinte exemplo empregaremos coma compoñente nai á propia compoñente raíz de Angular (**`app-root`**). Ficando o seu código coma segue:

- **`app.component.ts`:** engadimos unha propiedade nova á cal chamamos **`compartida`**, neste caso de tipo numérico e iniciada co valor «0».

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FillaComponent } from './filla/filla.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FillaComponent ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'input';
  compartida : number = 0;
}
```

- **`app.component.html`:** neste ficheiro engadimos a etiqueta correspondente á compoñente filla (**`app-filla`**) cun *binding* de propiedades que asocia a súa propiedade **`recibida`** coa propiedade **`compartida`** da compoñente nai. A maiores incorporamos

un botón que permite incrementar o valor da variable **compartida**, para comprobar como efectivamente a información flúe dende a compoñente nai cara á filla.

```
<app-filla [recibida]="compartida"></app-filla>
<button (click)="compartida=compartida+1">INCREMENTAR</button>
```

Compoñente filla

Supoñamos pois que temos unha compoñente filla denominada **nova-componhente** co seguinte código:

- **filla.component.ts:** neste caso engadimos unha propiedade (**recibida**) á clase e acompañámola do decorador `@Input` para indicar que vai asociada á información recibida pola etiqueta HTML da compoñente. O valor asignado á propiedade unicamente serve para confirmar que efectivamente o valor inicial que aparecerá na páxina non será o que aquí figura, senón o que se reciba dende a compoñente nai.

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

@Component({
  selector: 'app-filla',
  standalone: true,
  imports: [],
  templateUrl: './filla.component.html',
  styleUrl: './filla.component.css'
})

export class FillaComponent {
  @Input() recibida : number = 10; // Poñemos un valor por defecto á propiedade
para ver que efectivamente é sobreescrito dende a compoñente nai
}
```

- **filla.component.html:** o contido deste ficheiro é relativamente sinxelo, e unicamente hai unha interpolación de *strings* que dirixe á propiedade **recibida**.

```
<p>O valor da propiedade recibida por input é {{ recibida }}</p>
```

O decorador `@Output`

O decorador **`@Output`** de Angular posibilita o paso de información dende as compoñentes fillas cara ás compoñentes nai, isto é, de xeito ascendente na xerarquía de compoñentes.

Co gallo diso, márcase unha propiedade/atributo da compoñente filla co devandito decorador pasando a mesma a ser accesible pola compoñente nai.

Compoñente nai

No exemplo que figura a continuación empregaremos coma compoñente nai á propia compoñente raíz de Angular (**`app-root`**). O código será o seguinte:

- **`app.component.ts`:** engadimos unha propiedade nova á cal chamamos **`recibida`**, neste caso de tipo numérico e iniciada co valor «0; e un método **`amosarMensaxe(texto)`** que escribe o texto recibido sobre a variable **`recibida`**.

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { FillaComponent } from './filla/filla.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet, FillaComponent ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'output';
  recibida : string = ''; // Poñemos un valor por defecto á propiedade para que se
vexa que efectivamente é sobreescrito dende a compoñente filla

  amosarMensaxe(texto: string): void {
    this.recibida = texto;
  }
}
```

- **`app.component.html`:** engadimos a etiqueta correspondente á compoñente filla (**`nova-componhente`**) incorporando un *binding* de eventos que chama ao método **`amosarMensaxe(texto)`** pasándolle coma parámetro o evento desencadeado.

```
<app-filla (mensaxe)="amosarMensaxe($event)"></app-filla>
<p>A compoñente filla enviou a seguinte mensaxe: <b>{{ recibida }}</b></p>
```

Compoñente filla

O código da nosa compoñente filla (**`filla`**) é como figura a continuación:

- **`filla.component.ts`:** neste caso engadimos unha propiedade (**`mensaxe`**) á clase e acompañámola do decorador `@Output` de tipo `EventEmitter` e con tipo de datos `string`, isto implica que **`mensaxe`** será un evento emisible que pode ser desencadeado

dende HTML. Ademais diso, engadimos tamén un método `enviar()` no cal emítese o evento **`mensaxe`** cunha mensaxe asociada.

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

@Component({
  selector: 'app-filla',
  standalone: true,
  imports: [],
  templateUrl: './filla.component.html',
  styleUrl: './filla.component.css'
})

export class FillaComponent {
  @Output() mensaxe = new EventEmitter<string>(); // Esta propiedade actuará como
vehículo da mensaxe

  enviar() {
    this.mensaxe.emit('Ola compoñente nai!');
  }
}
```

- **`filla.component.html`:** unicamente contén un botón HTML que chama ao método `enviar()`.

```
<button (click)="enviar()">ENVIAR</button>
```

O decorador `@Hostbinding`

O decorador **`@HostBinding`** en Angular úsase para ligar propiedades dunha directiva ou compoñente a atributos ou estilos do elemento anfitrión (*host*). Permite modificar directamente o comportamento ou aparencia dun elemento HTML desde a lóxica da clase.

Imos agora co código TS dun exemplo de aplicación desta directiva:

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

@Directive({
  selector: '[appLigado]'
})
export class LigadoDirective {
  @HostBinding('style.color') corTexto = 'red';
}
```

E o código do *template* HTML:

```
<p appLigado>Este texto está ligado ao atributo de estilo.</p>
```

Decoradores de métodos

O decorador `@HostListener`

O decorador **`@HostListener`** en Angular úsase para escoitar eventos do DOM directamente no elemento anfitrión (*host*) dunha directiva ou compoñente. Permite asociar métodos de clase a eventos específicos, como `click`, `mouseover`, ou `keyup`, de forma que se executen automaticamente cando o evento ocorra.

Velaquí un exemplo práctico de uso desta directiva. Comezamos polo *script* TS:

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

@Directive({
  selector: '[appClic]'
})
export class ClicDirective {
  @HostListener('click', ['$evento'])
  aoClicar(evento: Event) {
    console.log('Elemento clicado:', evento);
  }
}
```

E no *template* HTML engadiriamos:

```
<p appClic>Fai clic neste parágrafo para activar o evento.</p>
```

Decoradores de parámetros

O decorador `@Inject`

O decorador **`@Inject`** en Angular úsase para realizar a inxección de dependencias manualmente. Permite especificar un *token* ou un provedor específico que debe ser inxectado nun compoñente, servizo, ou outro elemento, mesmo en situacións nas que Angular non pode deducilo automaticamente.

Vexamos un exemplo disto:

```
import { Component, Inject } from '@angular/core';
import { DOCUMENT } from '@angular/common';

@Component({
  selector: 'app-inxeccion',
  template: '<p>Compoñente con inxección de dependencias.</p>'
})
export class InxeccionComponent {
  constructor(@Inject(DOCUMENT) private documento: Document) {
    console.log('Documento inxectado:', this.documento);
  }
}
```

O decorador `@Directive`

O decorador **`@Directive`** en Angular úsase para definir unha directiva personalizada. Permite engadir comportamento específico aos elementos do DOM, modificando o seu aspecto, estilo ou funcionalidade.

E aquí un exemplo de aplicación da mesma, comezando polo código TS:

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

@Directive({
  selector: '[appDestacar]'
})
export class DestacarDirective {
  constructor(private el: ElementRef) {
    this.el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

E o código HTML:

```
<p appDestacar>Este texto ten unha cor de fondo destacada.</p>
```

Erros e problemas habituais

A continuación algún dos erros e problemas que atoparemos habitualmente ao empregar decoradores en Angular:

- **Erro: "Erro ao empregar `@Injectable`".**
  - Causa: configuración incorrecta do ámbito no decorador.
  - Solución: comprobar se `providedIn` está correctamente definido.
- **Erro: "Non se reciben datos no compoñente".**
  - Causa: o decorador `@Input` non está configurado correctamente.
  - Solución: comproba que o compoñente pai está enviando datos coa sintaxe axeitada.

```
@Input() dato: string = '';
```

- **Erro: "Os eventos do compoñente non son capturados".**
  - Causa: o decorador `@Output` non está configurado ou conectado correctamente.
  - Solución: osa `EventEmitter` para emitir eventos desde o compoñente fillo e asegúrate de que o compoñente pai os captura.

```
@Output() evento = new EventEmitter<string>();
```

```
emitirEvento() {
  this.evento.emit('Mensaxe enviada');
}
```

Bibliografía

- «El gran libro de Angular». (2018). Boada, M. e Gómez, J.A. Marcombo (Barcelona).
- Decoradores en TypeScript: https://rafaelneto.dev/blog/decoradores-typescript/
- Documentación oficial de Angular: https://angular.io/docs
- Documentación oficial de **Node.js**: https://nodejs.org/es/docs/
- Instalar **Angular CLI** en Ubuntu 20.04 LTS: https://noviello.it/es/como-instalar-angular-cli-en-ubuntu-20-04-lts/
- Instalar **Node.js** en Ubuntu 20.04 LTS: https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-20-04-es
- Páxina oficial de **Angular**: https://angular.io/
- Páxina oficial de **Node.js**: https://nodejs.org/es/

ÍNDICE