# Primeros Pasos con Angular 8

## Introducción

La palabra inglesa "framework" (marco de trabajo) define, en términos generales, un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.
En el desarrollo de software, un framework o infraestructura digital, es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto. Representa una arquitectura de software que modela las relaciones generales de las entidades del dominio, y provee una estructura y una especial metodología de trabajo, la cual extiende o utiliza las aplicaciones del dominio.

### ¿Entonces, qué es Angular?

Angular es un framework que facilita la construcción de aplicaciones web. Angular combina templates declarativos, inyección de dependencias y herramientas de comunicación entre Frontend y Backend.
Está pensado para dividir un proyecto en componentes, lo que permite una mejor reutilización de recursos.
El proyecto Angular es propiedad de la empresa Google y tiene una versión previa no compatible llamada AngularJS.

## Herramientas para trabajar con Angular.

Para poder trabajar con Angular vamos a tener que instalar las siguientes dependencias:

- NodeJs.
- NPM (En Windows se instala junto con NodeJs)
- GIT (No es obligatorio, pero deseable)
- Angular CLI (Interfaz de línea de comandos para Angular)
- Visual Studio Code (Editor de código fuente desarrollado por Microsoft para Windows, Linux, MacOs)

## Typescript.

### ¿Que es Typescript?

Es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft.
Es un superconjunto de JavaScript, que añade tipado estático y objetos basados en clases.
extiende la sintaxis de JavaScript.

### Instalación

Para poder empezar a trabajar con Typescript primero vamos a necesitar instalarlo, para ello vamos a abrir una terminal del sistema operativo y ponemos:
*npm install -g typescript.*

## Compilación a Javascript

TypeScript se escribe en archivos .ts, los cuales no pueden ser usados directamente en el navegador y necesitan ser traducidos a JavaScript primero.

Para ello vamos a usar la herramienta de línea de comando tsc.

El siguiente comando toma un archivo Typescript llamado main.ts y lo traduce a un archivo JavaScript llamado main.js. Sí main.js ya existe, lo sobreescribe.

*tsc main.ts*

## Tipado Estático

Un rasgo distintivo de TypeScript es que soporta tipado estático. Esto significa que se puede declarar los tipos de las variables, y el compilador se asegurará de que no esten asignados los tipos de las variables en forma errónea. Si fueron omitidas las declaraciones, el compilador deducirá automáticamente el tipo en base al dato almacenado.

Estos son algunos de los tipos más comunes usados:

- Number: Todos los valores numéricos son representados por este tipo, no hay separación entre enteros,de coma flotante, o otros.
- String: Tipo de dato de texto, los strings pueden ser rodeados por 'comillas simples' o "comillas dobles".
- Boolean: **true** o **false**. Usando 0 y 1 causará un error de compilación.
- Any: Variable cuyo tipo puede ser un valor string,number,o ***any****thing else*.
- Arrays: Tiene dos posibles sintaxis: *my_arr: number[]*; o *my_arr:Array<number>*
- Void: Usado para las funciones que no retornan valor.

Para ver la lista completa de tipos disponibles, ir a la página oficial de documentación de TypeScript.

## Interfaces

Las interfaces se utilizan para verificar si un objeto se ajusta a una estructura determinada. Por definición una interfaz puede estar compuesta por una combinación específica de variables, asegurándose de que siempre vayan juntas. Cuando se traduce a JavaScript las interfaces desaparecen, su único propósito es ayudar al desarrollo de la aplicación.

Clases.

TypeScript ofrece un sistema de clases muy similar a Java o C#, incluyendo herencia, clases abstractas, implementación de interfaces , setters/getters, etc.

Para obtener más información sobre clases en TypeScript se puede consultar la documentación oficial.

## ANGULAR CLI

Angular CLI es una herramienta de línea de comandos que nos permite crear un proyecto, agregar archivos y realizar una serie de tareas de desarrollo tales como testear y deployar.

Para instalarlo vamos a tener que abrir una terminal del sistema operativo y poner el siguiente comando:

*npm install -g @angular/cli*

Para generar un nuevo proyecto Angular y correrlo en modo desarrollo tenemos que hacer lo siguiente:

*ng new nuevo-proyecto*

*cd nuevo-proyecto*

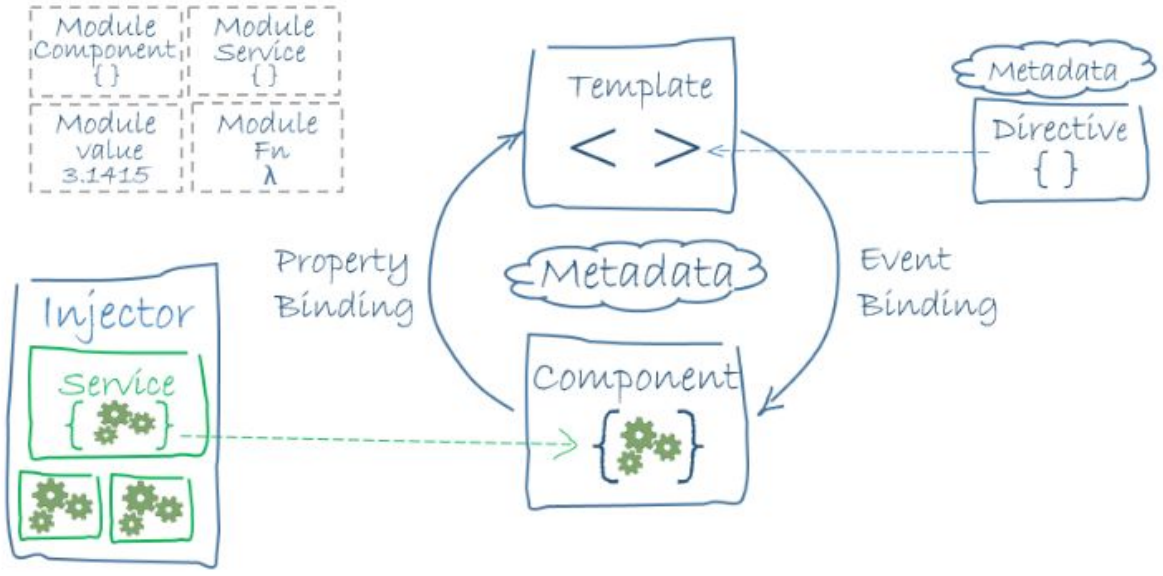
*ng serve*

Comandos más comunes de Angular CLI:

- ng new
- ng serve
- ng generate
- ng lint
- ng test
- ng build

Si entramos a la wiki de Angular CLI vamos a tener un listado completo de todos los comandos.

# Arquitectura de una aplicación Angular.




Las aplicaciones Angular son modularizadas y Angular tiene su propio sistema de modularización llamado Angular Modules o NgModules.

Toda aplicación Angular tiene al menos un módulo Angular, el "root module", convencionalmente llamado "AppModule".

Un módulo Angular es una clase con el decorador @NgModule.

Los decoradores son funciones que modifican las clases JavaScript.

NgModule es una función decorador que toma un simple objeto metadata cuyas propiedades describen el módulo.

Las propiedades más importantes son:

- Declaraciones: son las clases vistas que pertenecen al módulo. Angular tiene tres tipos de clases vistas: componentes,directivas y pipes.
- Exports: Subconjunto de declaraciones que deben ser visibles y usables en los componentes de los templates de otros módulos.

- Imports: otros módulos cuyas clases que se exportan son necesarias por los componentes de los templates de otros módulos.
- Providers: creadores de servicios de ese módulo que contribuye con la colección global de servicios. Se vuelven accesibles en todas las partes de la aplicación.
- Bootstrap: vista de la aplicación principal, llamado el "root component" que contiene todas las otras vistas de la aplicación. Solo el root module debe tener la propiedad bootstrap.

## Componentes

Un componente controla parches de pantallas llamadas vistas.
Se definen componentes de lógica de aplicación dentro de las clases. Las clases interactúan con las vistas a través de una API de propiedades y métodos.
Angular crea, actualiza y destruye componentes cada vez que el usuario se mueve a través de la aplicación.

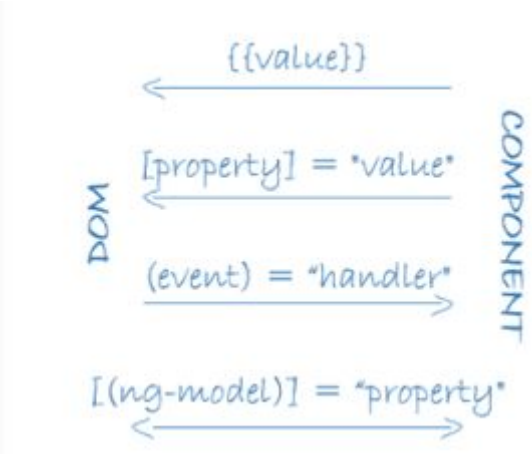
## Metadata

La metadata le dice a Angular como procesar la clase.
En TypeScript se agrega metadata a través de los decoradores.
El decorador @Component toma el objeto de configuración requerido con la información que Angular necesita para crear y presentar el componente y su vista.

## Data Binding

Angular soporta **data binding**, un mecanismo que sirve para coordinar partes de un template con partes de un componente. Agregando binding markups en el documento HTML le decimos a Angular como conectar ambos lados.
Hay cuatro formas de sintaxis de data binding:

- Desde el Componente hacia el DOM.
- Desde el DOM hacia el Componente.
- En ambas direcciones.




Angular admite el enlace de datos bidireccional (Two way Data Binding), un mecanismo para coordinar las partes de una plantilla con las partes de un componente

## Estructuras de carpetas

Una vez creada la aplicación Angular a través del comando ng new nos vamos a encontrar con las siguientes carpetas:

- **e2e**: Es usada para testear y ayudarnos a que la aplicación funciona correctamente.
- **node_modules**: Carpeta donde se van a guardar todos los paquetes Node.
- **src**:  En esta carpeta es donde vamos a trabajar en el proyecto usando Angular.
- **.angular-cli.json**: Contiene el nombre del proyecto, la versión del CLI, etc.
- **.editorconfig**: Archivo de configuración del editor.
- **.gitignore**: Establecemos que archivos y carpetas van a ser ignoradas a la hora de subir nuestro código al repositorio.
- **.karma.conf.js**: Este archivo es usado para crear pruebas unitarias.
- **package.json**: Establece que librerías van a ser instaladas dentro de la carpeta node_modules cuando corramos el comando npm install.
- **protractor.conf.js**: Este es el archivo de configuración requerido para testear la aplicación.
- **tsconfig.json**: Contiene las opciones a la hora de compilar el proyecto.
- **tslint.json**: Archivo que contiene las reglas que deben ser consideradas mientras se compila.

# Componentes Angular

Los componentes son clases que interactúan con los archivos .html

Dentro de la carpeta src, nos vamos a encontrar con otra carpeta llamada app.

Dentro de esta carpeta tenemos los siguientes archivos:

- app.component.css
- app.component.html
- app.component.spect.ts
- app.component.ts
- app.module.ts

Dentro del archivo **app.module.ts** encontramos algunas librerías que ya fueron importadas.

También vamos a ver como ya fue importado el componente *AppComponent*. A partir de él vamos a comenzar nuestro desarrollo, lo que lo transforma en el componente "padre".

Una de las mejoras cosas que tiene Angular, es que "componible", esto significa que podemos construir grandes componentes a partir de componentes más pequeños. "*Una aplicación es simplemente un componente que renderiza a otros componentes".*

Debido a que los componentes están estructurados en una forma de árbol "*padre/hijo"*, cada vez que se renderiza un Componente, este lo hace a sus Componentes hijos.

## Estructura de los Componentes Angular

Cada componente está compuesto por tres partes:

- Un Decorador de Componente.
- Una vista
- Un controlador.

```
3   @Component({
4     selector: 'app-hola-mundo',
5     template: `<h1>Hola Mundo!</h1>`,
6     styleUrls: ['./hola-mundo.component.css']
7   })
8   export class HolaMundoComponent implements OnInit {
9
10    constructor() { }
11
12    ngOnInit() {
13    }
14
15  }
```

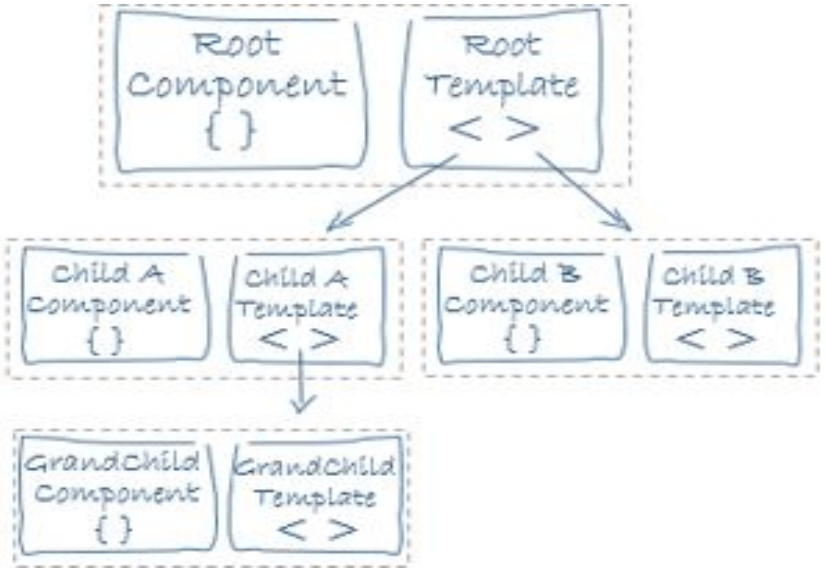
El *@Component* se denomina **decorador**. Este agrega metadata a la clase que le sigue (HolaMundoComponent).

El decorador @Component especifica lo siguiente:

- un *selector*, el cual establece que elemento Angular va a coincidir.
- un *template*: el cual define el template del componente.

El Componente **controlador** es definido por una clase, HolaMundoComponent, en este caso.

Además el decorador *@Component* asocia un componente a un template. Juntos, el componente y su respectivo template, definen una **vista**.




## Component selector

El *selector* tiene la tarea de indicar cómo el componente va a ser reconocido cuando sea usado en un template. Es una forma de definir qué elementos en el HTML van a coincidir con el componente. Por ejemplo a través del selector *'app-hola-mundo'* estamos definiendo un nuevo tag html llamado *app-hola-mundo*, el cual es definido como un nuevo tag que tiene su respectiva funcionalidad se lo utilice donde se lo utilice.

*<app-hola-mundo></app-hola-mundo>*

Angular va a usar el componente *HolaMundoComponent* para implementar la funcionalidad definida por este selector.

## Component template

El template es la parte visual del componente. Usando la opción *template* en el *@Component*, declaramos el template HTML que ese component va a estar usando.

*template: `<h1>Hola Mundo!</h1>`*

En este caso se está utilizando la sintaxis de tipo string llamada backtick, propia de Typescript.

También podemos establecer el template en un archivo separado, en ese caso en vez de usar *template* vamos a estar usando *templateUrl*

```
3   @Component({
4     selector: 'app-hola-mundo',
5     templateUrl: 'hola-mundo.component.html',
6     styleUrls: ['./hola-mundo.component.css']
7   })
8   export class HolaMundoComponent implements OnInit {
9
10    constructor() { }
11
12    ngOnInit() {
13    }
14
15  }
```

# Sintaxis de los Templates

El lenguaje utilizado en los templates de Angular es HTML, pero a su vez vamos a poder extender su markup a través de nuevos elementos y atributos.

## Interpolación

Usamos la interpolación para enlazar el valor de una propiedad de una clase Component a su respectivo template.

*<h1>{{titulo}}</h1>*

El texto que se encuentra entre la doble llave es el nombre de la propiedad de la clase. Angular reemplaza ese nombre por el valor que corresponde a la propiedad.

El texto encerrado entre la doble llave es una expresión que Angular primero evalúa y luego lo convierte a un string.

## Template Expressions

Una expresión de plantilla produce un valor. Angular ejecuta la expresión y la asigna a una propiedad de un objeto de enlace; el destino puede ser un elemento HTML, un componente o una directiva

*{{ 1 + 1}} // En la vista se verá reflejado el número 2*

## Template statements

Un *template statement* responderá a un evento unido a un objeto como un elemento, componente o directiva.

*(event)=”statement”*

*<button (click)=”saludar()”>Saludar</button>*

Responder a eventos es otro de los lados del “flujo unidireccional de datos” de Angular.

En relación a su contexto, los statements solo pueden ser manejados por métodos de la instancia del componente.

## Data Binding

Data binding es el mecanismo para coordinar lo que el usuario ve, en relación a los valores de los datos de la aplicación. Mientras que podamos establecer valores y obtener valores desde el HTML, la aplicación va a ser más facil de escribir, leer y mantener. Solo debemos declarar las uniones entre la fuente (el componente) y el objeto HTML, el resto queda a cargo del Framework.

Los tipos de uniones pueden ser agrupados en tres categorías, dependiendo de la dirección del flujo de datos:

| Data direction | Syntax | Type |
|---|---|---|
| One-way<br>from data source<br>to view target | `{{expression}}`<br>`[target]="expression"`<br>`bind-target="expression"` | Interpolation<br>Property<br>Attribute<br>Class<br>Style |
| One-way<br>from view target<br>to data source | `(target)="statement"`<br>`on-target="statement"` | Event |
| Two-way | `[(target)]="expression"`<br>`bindon-target="expression"` | Two-way |

# Directivas de Atributos

Una directiva de atributo cambia la apariencia o el comportamiento de un elemento del DOM.
Hay tres tipos de directivas en Angular:

- Componentes: directivas con una plantilla.
- Directivas Estructurales: cambia el diseño del DOM agregando y eliminando elementos.
- Directivas de Atributos: cambia la apariencia o el comportamiento de un elemento, componente, u otra directiva.

Las directivas estructurales cambian la estructura de la vista. Dos ejemplos son NgFor y NgIf
Las directivas de atributos se utilizan como atributos de elementos.
Para construir una directiva de atributo necesitamos construir un controlador con el decorador *@Directive*, el cual especifica el selector que identifica al atributo.
La clase de controlador implementa el comportamiento directivo deseado.
Para generar una clase del tipo directiva con CLI debemos escribir en la terminal el siguiente comando:
*ng generate directive nombreDirectiva*

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

El decorador *@Directive* especifica la propiedad **selector**, en la cual establecemos el atributo CSS relacionado con la directiva que estamos creando.
En el caso anterior, Angular identifica cada elemento en el template que tenga el atributo *appHighLight* y aplica la lógica establecida en la directiva.
Después de la metadata *@Directive* se define la clase controladora, la cual contiene la logica de la directiva.

## Veamos un ejemplo de Directiva de Atributo

Primero creamos la clase controladora de la directiva:

*ng g directive resaltado*

Luego definimos el comportamiento de la directiva en la clase controladora. Para ello, vamos a utilizar la directiva *ElementRef* del core de Angular:

```
1   import { Directive, ElementRef } from '@angular/core';
2
3   @Directive({
4     selector: '[appResaltado]'
5   })
6   export class ResaltadoDirective {
7
8     constructor(el: ElementRef) {
9       el.nativeElement.style.backgroundColor = 'yellow';
10    }
11
12  }
```

Luego aplicamos la directiva a un elemento del Template:

```
4   <p appResaltado>Estoy resaltado</p>
```

Lo que vamos a obtener es lo siguiente:

Estoy resaltado

# Directivas Estructurales

Las directivas estructurales son aquellas que forman o modifican la estructura del DOM, por lo general añadiendo, removiendo, o manipulando elementos.

Las directivas estructurales son facilmente reconocibles debido a que llevan un asterisco (*) por delante.

## Directivas estructurales más comunes

## NgIF

Esta directiva recibe un valor booleano y de acuerdo a ello muestra u oculta un elemento del DOM.

```
<p *ngIf="true">
  Expression is true and ngIf is true.
  This paragraph is in the DOM.
</p>
<p *ngIf="false">
  Expression is false and ngIf is false.
  This paragraph is not in the DOM.
</p>
```

Cabe destacar que la directiva ngIf no oculta los elementos con CSS. Directamente lo agrega o remueve físicamente del DOM.

## NgFor

Veamos un ejemplo del funcionamiento de la directiva *ngFor:

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

@Component({
  selector: 'app-hola-mundo',
  templateUrl: 'hola-mundo.component.html',
  styleUrls: ['./hola-mundo.component.css']
})
export class HolaMundoComponent {
  nombre = 'Pablo Rodriguez';
  edad = 40;
  sueldos = [40000, 50000, 60000];
}

```

En el componente HolaMundoComponent definimos un nombre, edad y un array de sueldos.

```
<p>Nombre del empleado: {{nombre}}</p>
<p>Edad: {{edad}}</p>
<p *ngIf="edad >= 18">Es mayor de edad.</p>
<table border="1">
  <tr>
    <th>Sueldos</th>
  </tr>
  <tr *ngFor="let sueldo of sueldos">
    <td>{{sueldo}}</td>
  </tr>
</table>
```

El valor de la propiedad *nombre* va a ser siempre mostrado por *interpolación*, si la propiedad *edad* tiene un valor mayor a 18 se va a mostrar el párrafo donde está definida la directiva **ngIf*, y luego mostramos el valor de cada uno de los elementos del array *sueldos*, para ello hacemos uso de la directiva **ngFor*.