Clase 8 @May 15, 2023 subir

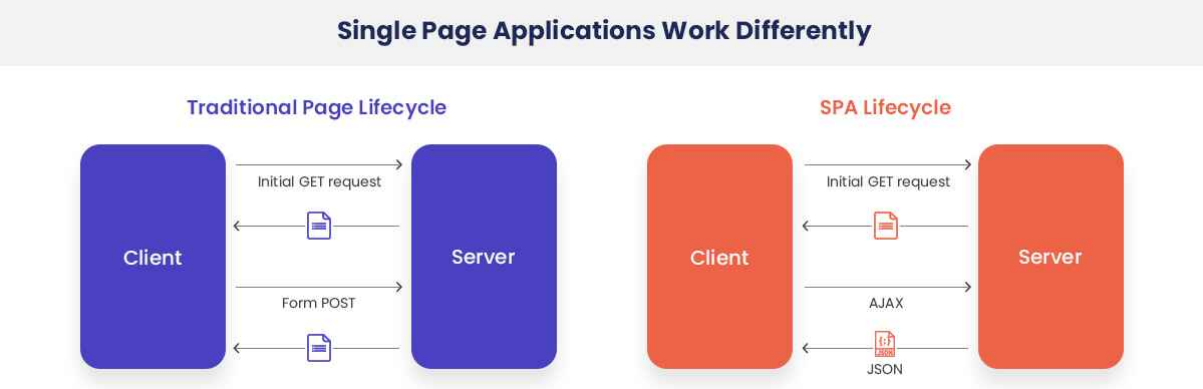
Puntos para la clase

- ¿Cómo funcionaban las web applications tradicionalmente?
- ¿Cómo funcionan las SPAs?
- Características de las SPAs
- ¿Qué es Angular?
- ¿Por qué vamos a usar Angular?
- Arquitectura de una aplicación Angular
- ¿Qué es un componente en Angular?
- ¿Qué es un Modulo en Angular?
- Data Binding e Interpolación
- Ejemplo introductorio angular (isntalamos npm, node y angular cli y creamos proyecto ejemplo).

Introducción al concepto de Single Page Applications

Esta clase tiene como objetivo introducirlos en el concepto de las SPAs y comenzar a ver Angular como una tecnología para lograrlo.

Es interesante ver cómo esta forma de contruir aplicaciones web se diferencia de las maneras “tradicionales” , es por esto que en esta calse vamos a entender qué diferencias hay entre una modalidad y otra.



¿Cómo funcionaban las web applications tradicionalmente?

En el pasado, el flujo que existia en una **Round-Trip Applications** era algo así:

1. Se tenía un **Web Server** (servidor web) que almacenaba y servía el contenido del sitio web. Este servidor estaba encargado de responder a las solicitudes HTTP que llegaban desde los navegadores de los usuarios.
2. Cuando un usuario quería acceder al contenido del sitio web, abría un **Browser** (navegador web)
3. El navegador enviaba una solicitud HTTP al servidor web, solicitando el contenido asociado.
4. El servidor web procesaba la solicitud y generaba una respuesta HTTP que contenía el contenido solicitado, como archivos HTML, CSS.
5. El servidor web enviaba la respuesta al navegador, que a su vez recibía y procesaba la respuesta.
6. El navegador interpretaba el contenido HTML recibido y construía una representación visual de la página web utilizando el motor de renderizado.
7. El usuario veía la página web en su navegador y podía interactuar con ella, por ejemplo, haciendo clic en enlaces, llenando formularios o enviando datos al servidor.
8. Si el usuario realizaba alguna acción que requería información adicional del servidor, como enviar datos a través de un formulario o solicitar una nueva página, el navegador enviaba otra solicitud HTTP al servidor web y el proceso se repetía.

Este flujo de Round-Trip Applications implicaba una comunicación continua entre el navegador y el servidor web, donde cada acción del usuario generaba una nueva solicitud y respuesta HTTP. Siempre que se realizaba una interacción, se producía un "viaje de ida y vuelta" entre el navegador y el servidor.

Este modelo, si bien se sigue usando hoy en día, tiene algunas **desventajas**. Por ejemplo:

1. El usuario debe esperar mientras el siguiente documento HTML se genera, requieren mayor infraestructura del lado del servidor para procesar todos los requests y manejar el estado de la aplicación, y ademas requieren más ancho de banda, ya que cada documento HTML debe estar autocontenido.
2. A su vez, la experiencia de usuario se degrada por factores como el efecto de refreshing (pestañeo) y el tiempo en ir a pedir los recursos al servidor.

Es por esto que ete enfoque solía ser común en el pasado, pero con el avance de las tecnologías web, han surgido enfoques como las Single-Page Applications (SPAs) y el uso de AJAX, que permiten una interacción más fluida y dinámica sin la necesidad de recargar la página completa en cada solicitud.

¿Cómo funcionan las SPAs?

Las *SPAs*, si bien siguen manteniendo la misma forma de interactuar **cliente-servidor**, toman un enfoque diferente.

1. Cuando un usuario accede a una página web, se envía una petición inicial al servidor.
2. El servidor responde enviando un archivo HTML inicial que contiene la estructura básica de la página.
3. Cuando el usuario realiza alguna acción, como hacer clic en un botón o llenar un formulario, se generan eventos en el navegador.
4. En lugar de enviar una petición completa al servidor y cargar toda la página de nuevo, se utilizan solicitudes AJAX para obtener pequeños fragmentos de HTML o datos adicionales del servidor.

5. Estas solicitudes AJAX se envían al servidor en segundo plano, sin necesidad de recargar toda la página.
6. Una vez que el navegador recibe la respuesta del servidor, actualiza dinámicamente la página mostrando el fragmento de HTML o integrando los datos recibidos en la interfaz **existente**.
7. De esta manera, el usuario experimenta una interfaz mas fluida y dinámica sin tener que esperar a que se cargue una página completa en cada interacción.

El documento HTML inicial nunca se recarga, y el usuario puede seguir interactuando con el html existente mientras las requests ajax terminan de ejecutarse asincrónicamente.

Particularmente veremos un framework que está 100% orientado a la construcción de SPAs: **Angular**.

El mismo logra sus mejores resultados cuando la aplicación a desarrollar se acerca al modelo de **Single-Page**. No quiere decir que no se pueda usar para Round-trip.

¿Qué es Angular?

- Framework de JavaScript (conjunto de herramientas y librerías predefinidas que proporcionan una base sólida para el desarrollo de aplicaciones web o de otro tipo utilizando el lenguaje de programación JavaScript)
- Orientado a construir Client-Side Apps (aplicaciones web que se ejecutan principalmente en el lado del cliente)
- Basado en HTML, CSS y JavaScript

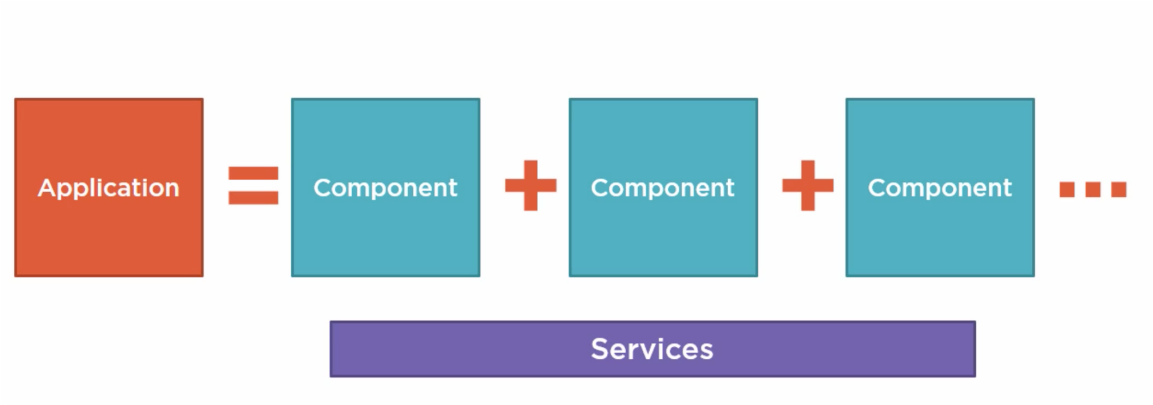
La meta de angular es traer las herramientas y capacidades que han estado disponibles para el desarrollo de back-end al cliente web, facilitando el desarrollo, test y mantenimiento de aplicaciones web complejas y ricas en contenido.

Angular funciona permitiéndonos extender HTML, expresando funcionalidad a través de elementos, atributos, clases y comentarios.

¿Por qué vamos a usar Angular?

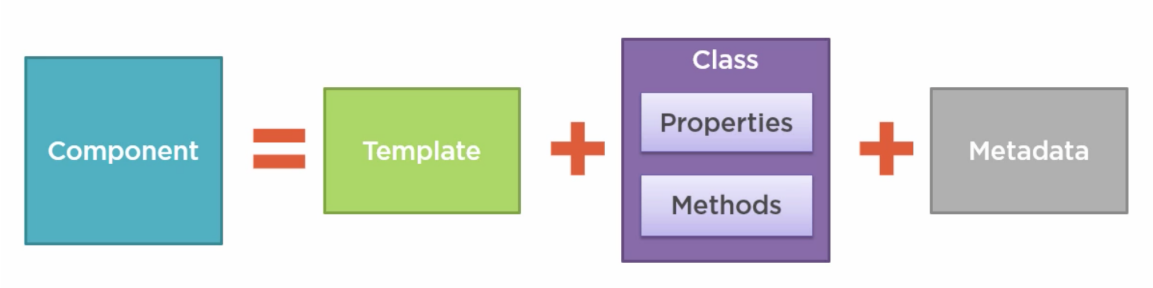
- Angular hace que nuestro HTML sea más expresivo, permitiéndole embeber/agregar features y lógica al HTML para lograr un data-binding con nuestros modelos. Esto nos permite mostrar campos que tengan valores de nuestros modelos/datos de forma sencilla, y tener un seguimiento de los mismos (actualización en tiempo real).
- Angular promueve la modularidad desde su diseño, siendo fácil crear y lograr reuso de los componentes y del contenido.
- Angular a su vez tiene soporte ya incluido para comunicación con servicios de back-end (es fácil que nuestras webs apps se conecten a nuestros backends y ejecuten leogica del lado del servidor).

Arquitectura de una aplicación Angular



En **Angular**, una aplicación **se define a partir de un conjunto de componentes**, del mismo modo que también de servicios subyacentes que son comunes a ellos y permiten el reuso de la lógica. Por ejemplo: servicios para contectarse con APIs REST, servicios que manejen la sesión desde el lado del cliente, servicios de autenticación, etc.

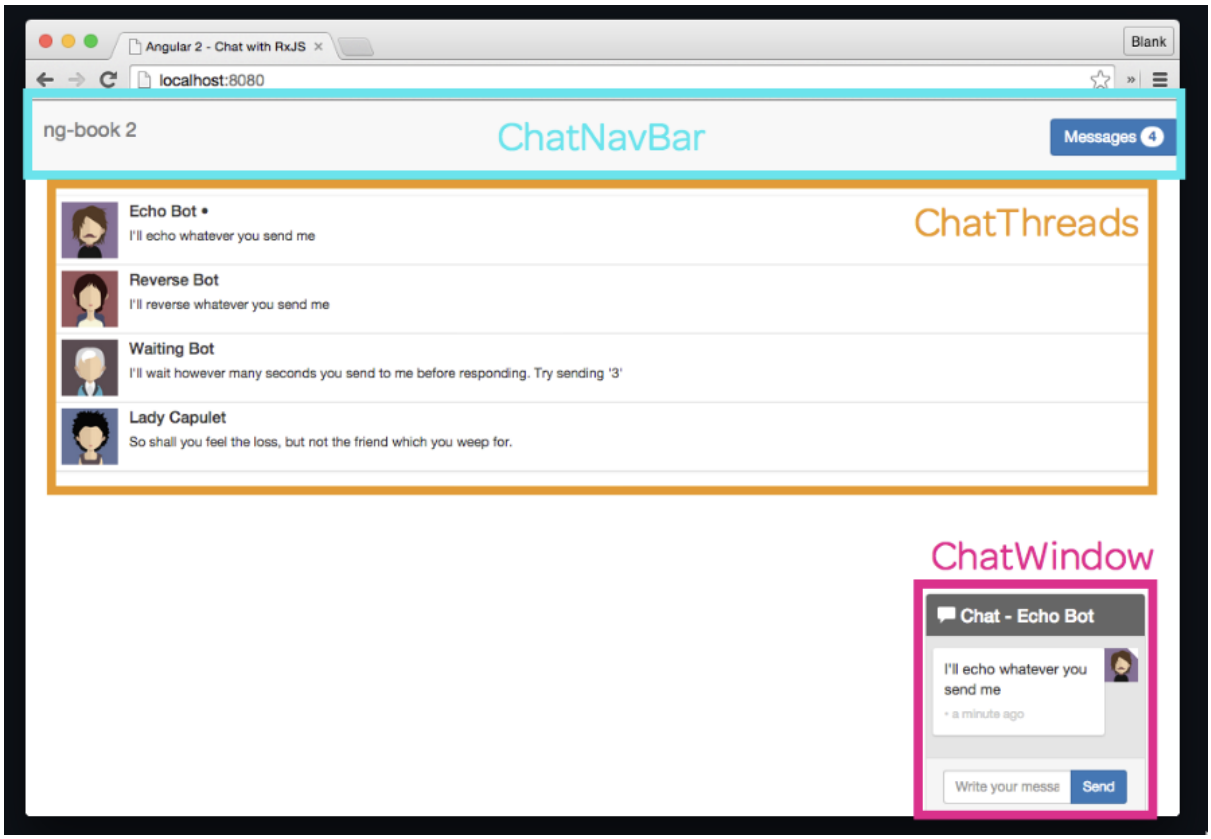
¿Qué es un componente en Angular?



Un componente es una una unidad modularizada que define la vista y la lógica para controlar una porción de una pantalla en Angular.

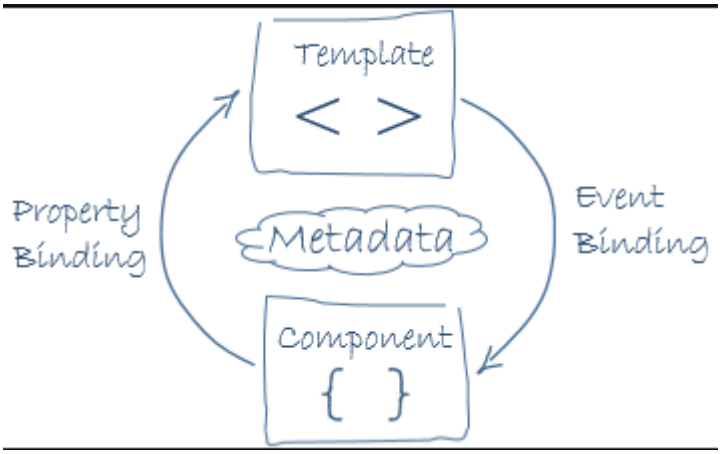
Cada componente la idea es que funcione armoniosamente y en conjunto con el resto para proveer una experiencia de usuario única. Como mencione, estos son modulares, resuelven un problema concreto y colaboran entre sí para lograr ir armando la interfaz de usuario como un puzzle donde cada pieza tiene sus diferentes responsabilidades.

Por ejemplo, una excelente forma de pensar los componentes es a través de la siguiente imagen:



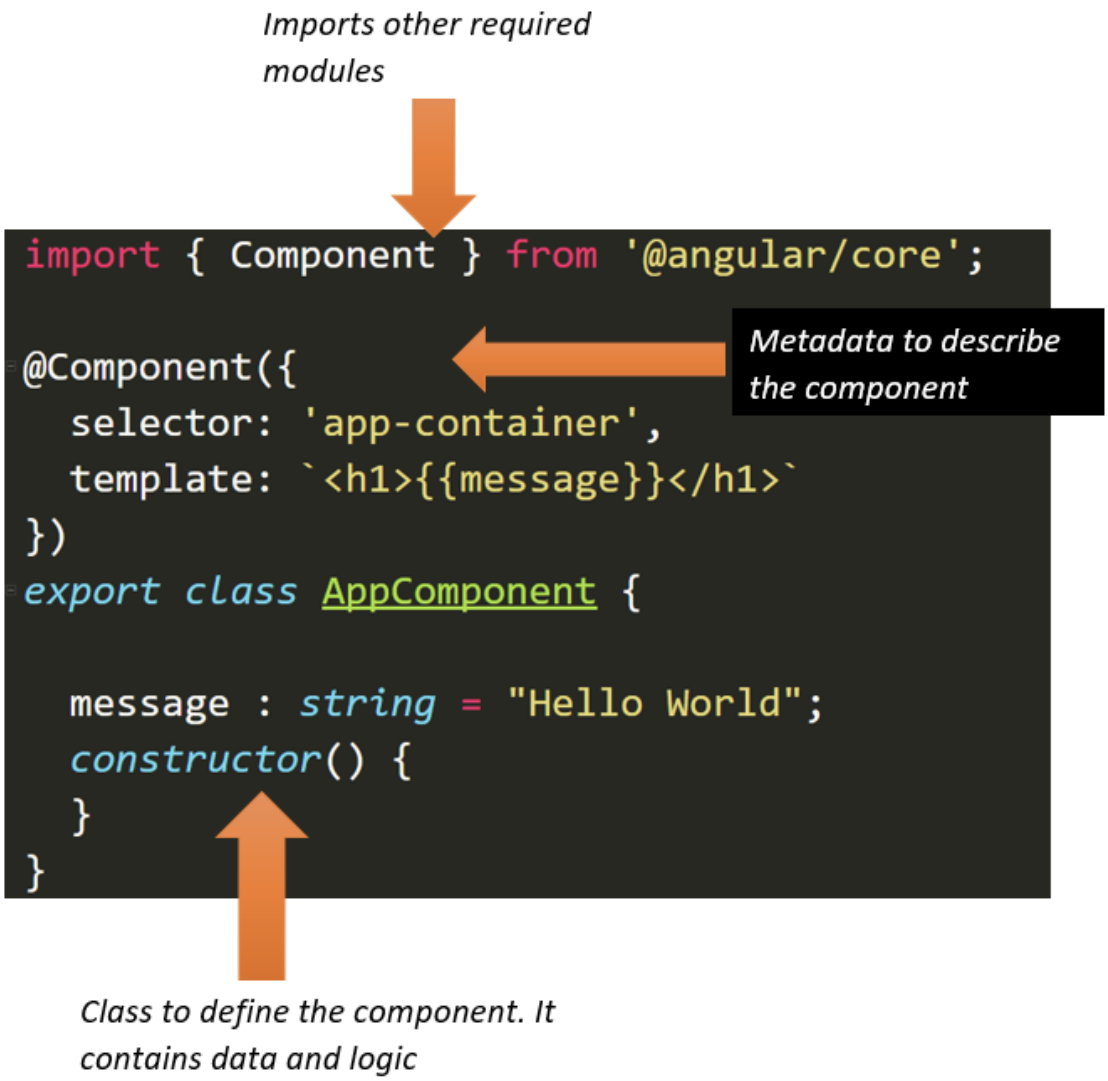
Cada componente se compone de:

- Un **template** el cual define la estructura HTML o la vista del mismo. Se crea con código html y define lo que se renderizará en a página. A su vez incluye *bindings* y *directivas* para darle poder a nuestra vista y hacerla dinámica.
- Una **clase,** esta representa el código asociado a la vista (creada con TypeScript), la cual posee los *datos*, llamadas las *properties* para que la vista los use (el nombre de una mascota por ejemplo), y a su vez posee la *lógica/funciones* que usan los mismos. Por ejemplo: la lógica para mostrar o esconder una imagen, la lógica para traer datos desde una Api, etc.
- **Metadata**, la cual provee información adicional del componente a Angular. Es lo que identifica a la clase asociada al componente.



La clase de un componente:

Para la clase de un componente, la sintaxis es como sigue a continuación:



El Template y la Metadata de un componente

Sin embargo, como ya sabemos, las clases de los componentes no son lo único necesario que precisamos para armar nuestra app en angular, precisamos darle el HTML, la vista, la UI. Todo eso y mas lo definimos a través del **metadata** del componente.

Una clase como la que definimos anteriormente se convierte en un componente de Angular cuando le definimos la metadata de componente.

En otras palabras el decorator `@Component` indica que la clase subyacente es un Componente de Angular y recibe la metadata del mismo (en forma de un objeto JSON de configuración).

Algunas de las opciones de configuración más útiles para `@Component`:

1. **selector**: Es un selector de CSS que le dice a Angular que cree e inserte una instancia de este componente en donde encuentre un tag `<nombre-component>` en su HTML padre. Por ejemplo, si el HTML de una app contiene contains `<nombre-component></nombre-component>`, entonces Angular inserta una instancia de la view asociada a `NombreComponent` entre esos dos tags.
2. **template**: Representa el código HTML asociado al componente y que debe ser mostrado cada vez que se use su selector. Es la UI para ese componente.
3. **templateUrl**: Similar al anterior pero permite referenciar a un documento HTML en lugar de tener que escribirlo directamente en el código del componente (puedo ponerlo en un archivo separado y tratarlo simplemente como un HTML.
4. **providers**: es un array de objeto de los providers de los servicios que el componente requiere para operar. Estos se inyectan a partir de inyección de dependencias; es simplemente una forma de decirle a Angular que el constructor del componente requiere algúns servicio para funcionar.

¿Qué es un Modulo en Angular?

Las aplicaciones Angular son modulares y Angular tiene su propio sistema de modularidad llamado **NgModules.**

Los NgModules son contenedores, nos permiten organizar nuestros componentes en funcionalidad cohesiva.

pueden contener componentes, proveedores de servicios y otros archivos de código cuyo alcance está definido por el NgModule que los contiene.

Pueden importar la funcionalidad que se exporta desde otros NgModules y exportar la funcionalidad seleccionada para que la utilicen otros NgModules.

```
content_copyimport {NgModule } from '@angular/core';
import {BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Cada aplicación Angular tiene al menos una clase NgModule, el *módulo raíz*, que se llama convencionalmente `AppModule` y reside en un archivo llamado `app.module.ts`.

Metadatos de NgModule

Un NgModule está definido por una clase decorada con `@NgModule()`. El decorador `@NgModule()` es una función que toma un único objeto de metadatos, cuyas propiedades describen el módulo. Las propiedades más importantes son las siguientes.

- `declarations`: Los componentes, *directivas*, y *pipes* que pertenecen a este NgModule.
- `exports`: El subconjunto de declaraciones que deberían ser visibles y utilizables en las *plantillas de componentes* de otros NgModules.
- `imports`: Otros módulos cuyas clases exportadas son necesarias para las plantillas de componentes declaradas en *este* NgModule.
- `providers`: Creadores de servicios que este NgModule aporta a la colección global de servicios; se vuelven accesibles en todas las partes de la aplicación. (También puedes especificar proveedores a nivel de componente, que a menudo es preferido).
- `bootstrap`: La vista principal de la aplicación, llamado el *componente raíz*, que aloja todas las demás vistas de la aplicación. Sólo el *NgModule raíz* debe establecer la propiedad `bootstrap`.

En angular la reutilización de componentes, servicios, etc ser realiza a nivel de módulo. **No importamos directamente un componente individual** desde otro módulo como, por ejemplo, `people.component`

```
import { PeopleComponent } from '../people/people. component'

@NgModule({
  imports:[PeopleComponent ]
})

export class ContactsModule {}
```

En vez de esto, tendremos otro módulo, digamos `PeopleModule` que ya declara y además **exporta** su propio `PeopleComponent` lo declara y además **lo exporta,** entonces puede ser utilizado por otros módulos. Consecuentemente, en nuestro `ContactsModule` simplemente importamos el `PersonModule` y entonces podremos utilizar en los componentes de nuestro `ContactsModule` el `PersonComponent` exportado.

Quedando en `PersonModule`

```
import { PeopleComponent } from '../people/people.component'

@NgModule({
  declarations:[PeopleComponent ],
  exports:[PeopleComponent ]
})

export class PeopleModule {}
```

Y en ContactsModule

```
import { PeopleModule } from '../people/people.module'

@NgModule({
  imports:[PeopleModule ]
})

export class ContactsModule {}
```

Una buena practica es tener un modulo por agrupacion logica de los componentes, es decir usar mas de un modulo ayuda a enriquecer la estructura. Por ejemplo podria tener un modulo de people que agrupo el componente de agregar people, el componente de eliminar people y el componente de editar people.

Ejemplo introductorio angular

Instalamos Node y npm

Para poder utilizar angular lo que debemos instalar es **Node.js** que es un entorno de ejecución para javascript.

¿Por qué lo usaremos en nuestras Apps de Angular?

1. Lo vamos a usar para instalar todas las librerías de Angular, es decir las dependencias.
2. También nos permitirá correr el compilador que convierta todos nuestros **.ts** en **.js**, de una forma muy simple, para que el navegador los pueda reconocer correctamente.
3. Funciona también como **WebServer**, que "servirá" nuestras Angular SPAs, en un web server liviando que levanta. Esto es mucho más cercano a un escenario real y evita problemas que suelen existir cuando accedemos directamente a los archivos a partir de su path en disco

Creamos el proyecto

1. **Configurar el entorno de desarrollo** - Instalamos Angular CLI `npm install -g @angular/cli` (este comando instala el cli de angular de manera global y solo debemos ejecutarlo cuando no se encuentra el CLI en nuestra pc)

   ```
   echo 'export PATH="$HOME/.npm-global/bin:$PATH"' >> ~/.bash_profile
   source ~/.bash_profile
   ```

2. **Creamos nuestro proyecto** - `ng new NombreDelProyecto` (si no se coloca un nombre se crea en el proyecto actual)
3. **Ejecutamos nuestro proyecto** - `ng serve --open` (--open habre una ventana en el navegador en `http://localhost:4200/`)

¿Qué sucede al levantar nuestra app?

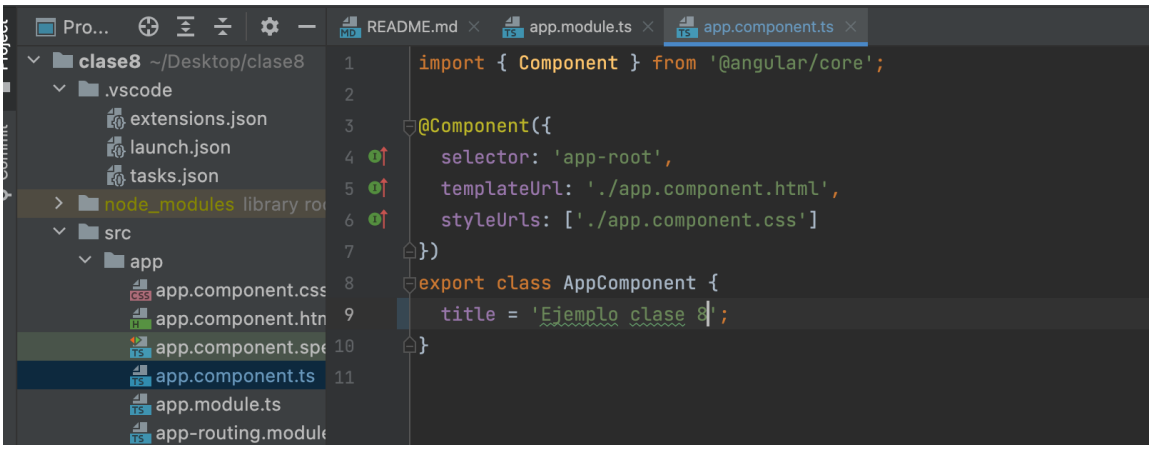
Al arrancar nuestra app usamos el comando `ng serve`.

Esto lo que hace es:

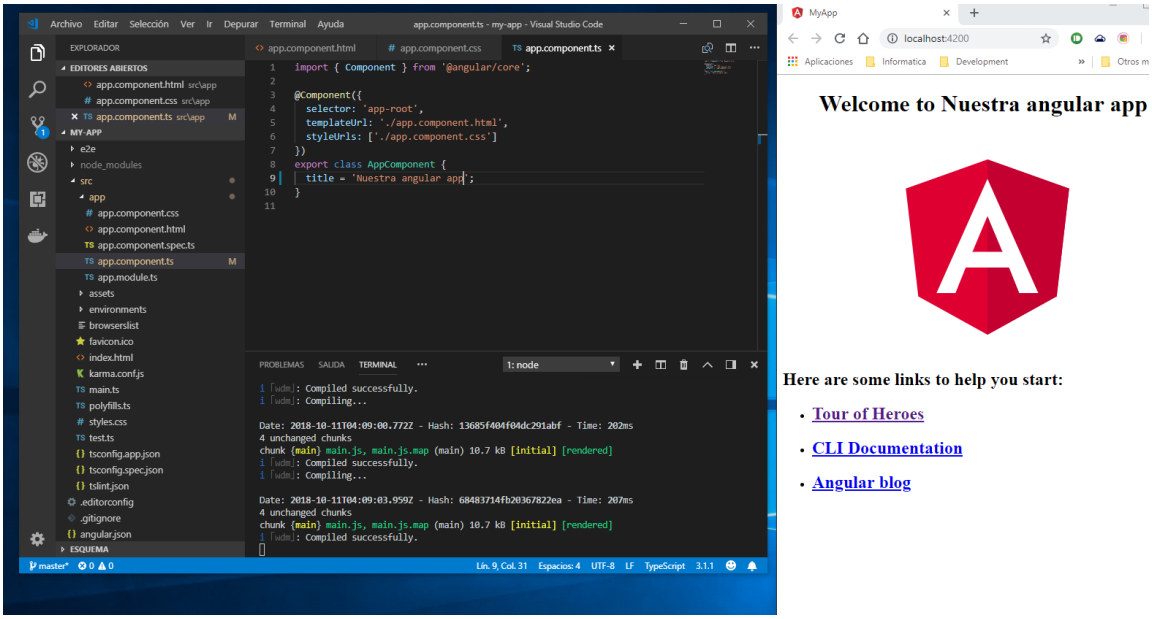
- Levantar un Web Server para que nuestro navegador pueda consumir los archivos desde ahí, es simplemente un ambiente local que funciona como un ambiente real.
- A su vez, vemos comos se ejecuta **tsc, el TypeScript Compiler**, el cual compila nuestros .ts y los transforma a .js.
- También vemos que el TypeScript compiler y el FileServer que levantamos, observa cualquier cambio a nivel de código, de manera que cada vez que hacemos un cambio en nuestro código, TypeScript lo recompila y podemos ver los cambios en el navegador.
-

Ejemplo:

- 1 Cambio el Componente de APP



- 2 Guardo y veo como el watcher se activa



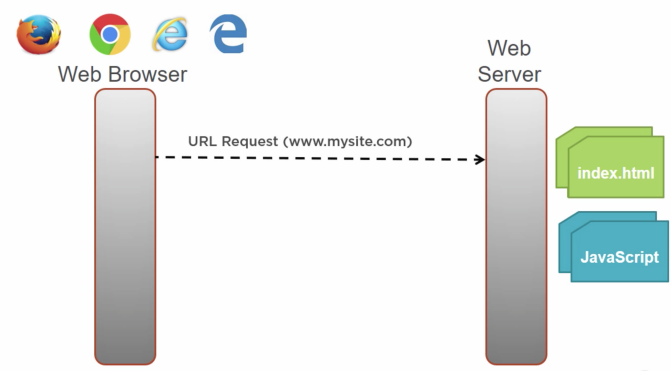
- 3 Instantáneamente mis cambios en la vista se reflejan en el navegador

4) El punto de entrada de nuestra aplicación es index.html

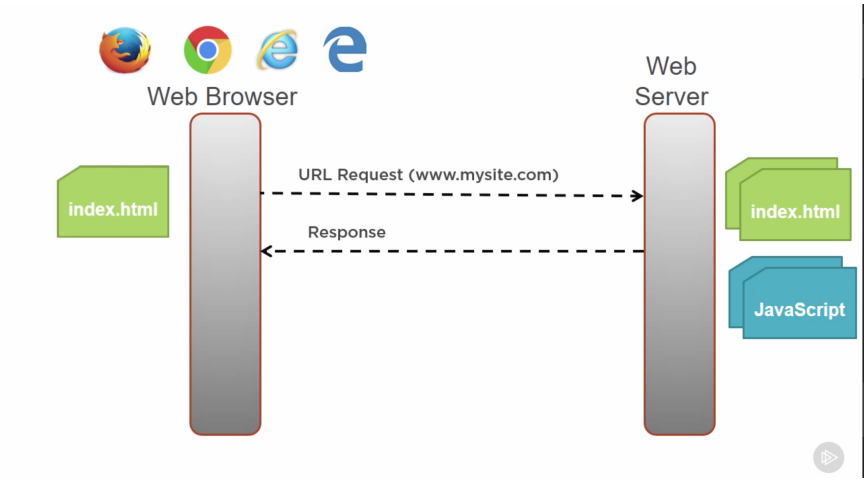
Este es el documento que es llevado desde el Web Server hasta el navegador, aquí comienza a ejecutarse toda nuestra aplicación en Angular.

El proceso es similar a cómo describimos al principio:

Se realiza una request del navegador al web Server:



Y este le contesta:



¿Qué contiene?

```
<!doctype html><html lang="en">
<head>
  <meta charset="utf-8">
  <title>MyApp</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>
```

- Lo unico que interesante que destacar que hace referencia a nuestro componente principal app.component.
- Casi nunca sera necesario editarlo, el CLI se encargara de añadir los archivos js y css al momento del building.

¿Y nuestro main.ts?

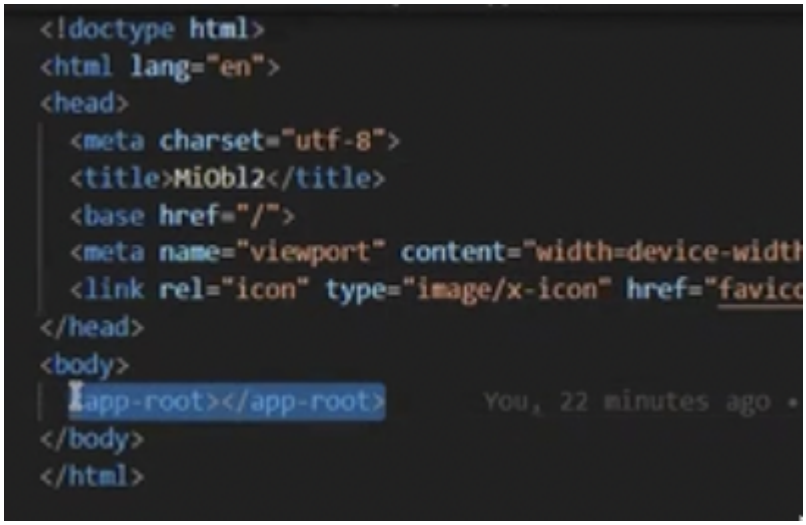
```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

Es el punto de entrada de nuestra aplicacion. Compila la aplicación con el compilador JIT y arranca el módulo raíz de la aplicación (AppModule) para ejecutarse en el navegador. También puede se puede usar el compilador AOT sin cambiar ningún código agregando el indicador - aot a los comandos ng build y ng serve.

Los componentes se “enganchan” al html mediante los selectores como se ve en el siguiente codigo, los componentes se recomiendan que empiecen con app

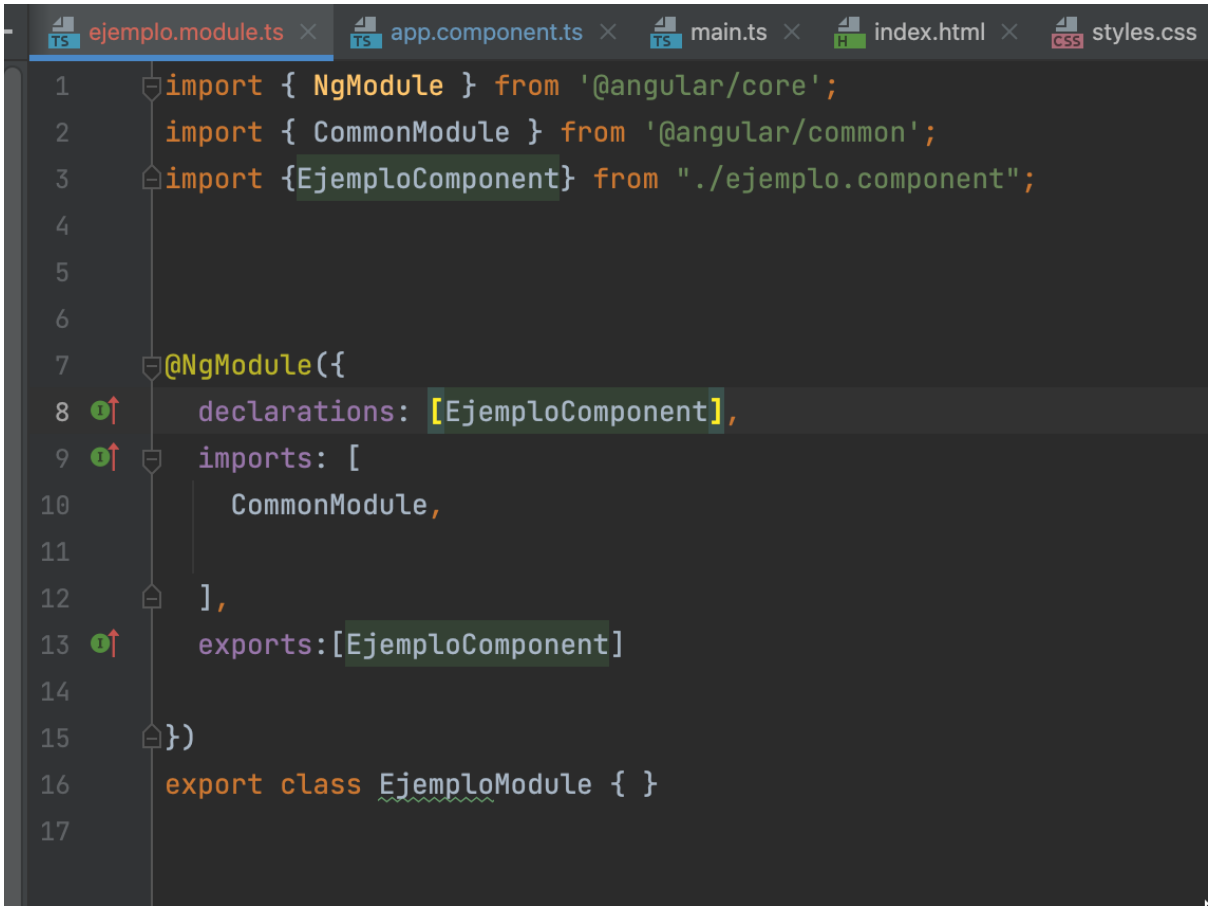
```
@component ({
selector: 'app-root'
templateurl: '/app.component.html'
styleurls: './app.component.scss'
})
export class AppComponent(
  title = 'ejemplo clase 7'
}
```

y despuyes en el index.html

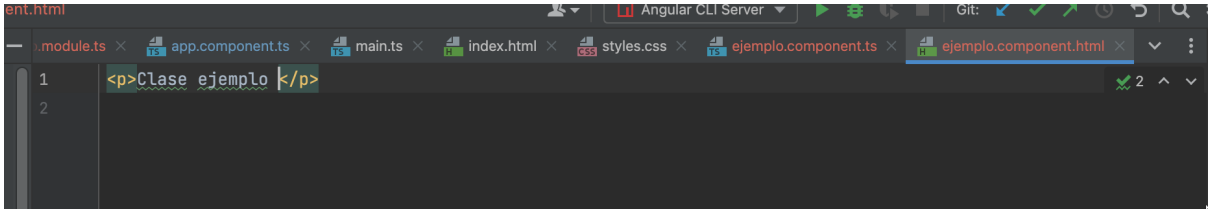


me dice podes crear todas estas cosas, nosotros vamos acrear modulos, componentes, servcicios, guardas y pipe.

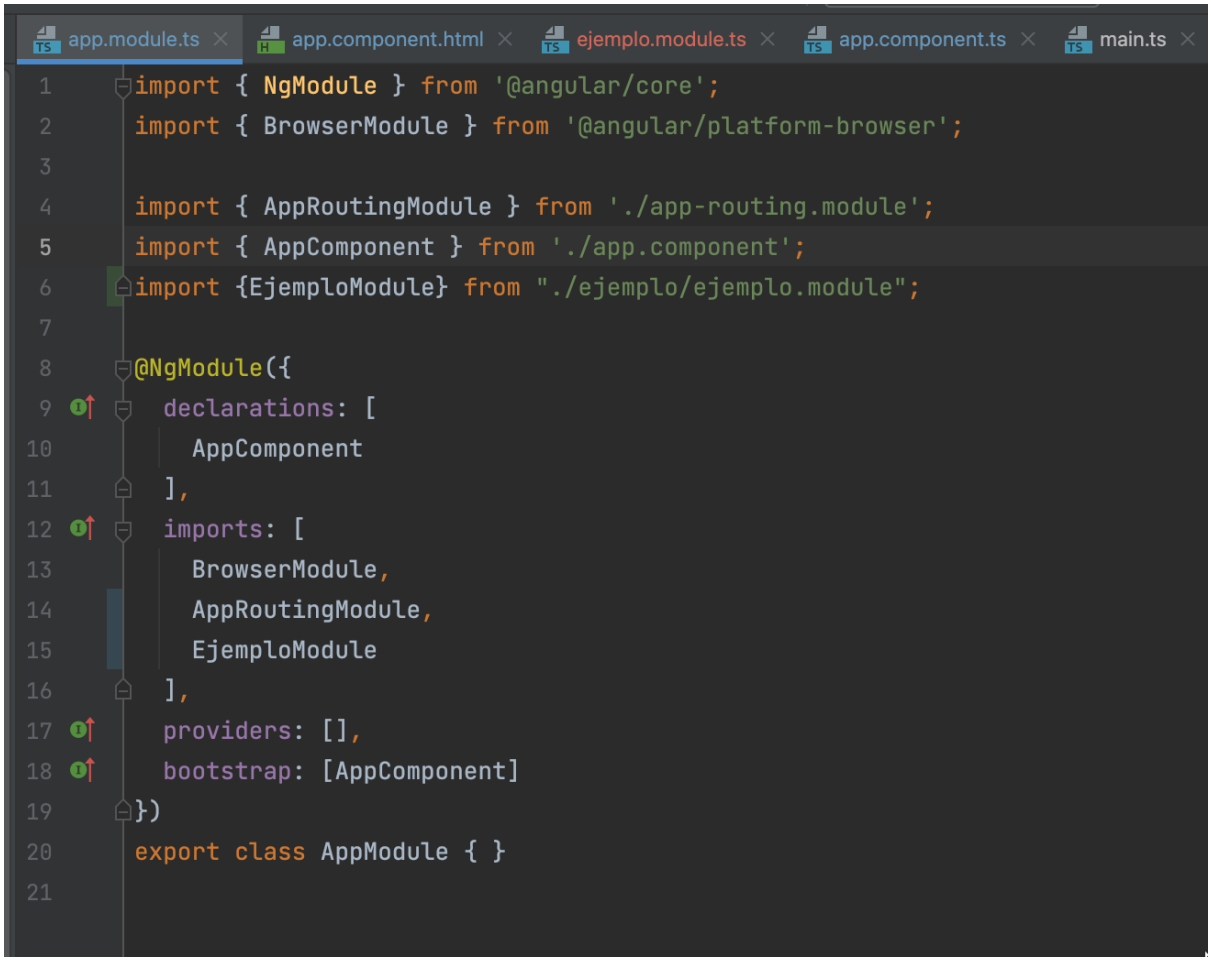
```
ng generate module ejemplo
```

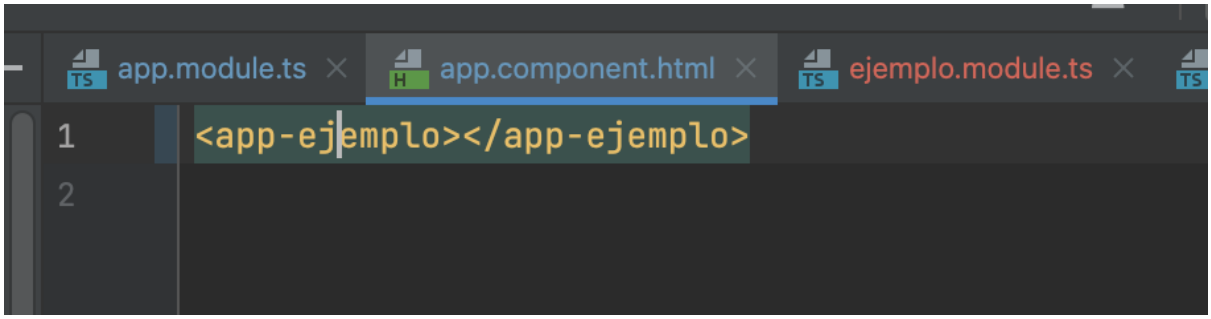


```
ng generate component ejemplo
```



Agrego el modulo al app module





Data Binding e Interpolación

Como mencionamos anteriormente, queremos que cada componente tenga asociada una cierta vista (HTML), sin embargo, queremos que los datos que se muestran en la misma sean dinámicos, y que vengan desde las properties de la clase del componente. No queremos hardcodear el HTML que representa los datos a mostrar. Por ejemplo:

No queremos:

```
<div class='panel-heading'>
    Nombre de la página que puede cambiar
</div>
```

Queremos:

En ejemplo.component.html

```
<div class='panel-heading'>
    {{pageTitle}}
</div>
```

En ejemplo.component.ts

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

@Component({
  selector: 'app-ejemplo',
  templateUrl: './ejemplo.component.html',
  styleUrls: ['./ejemplo.component.css']
})
export class EjemploComponent {
  @Input() pageTitle: string = "Título de la página";

}
```

En app.component.html

```
<app-ejemplo pageTitle="Título de la página 2"></app-ejemplo>
```

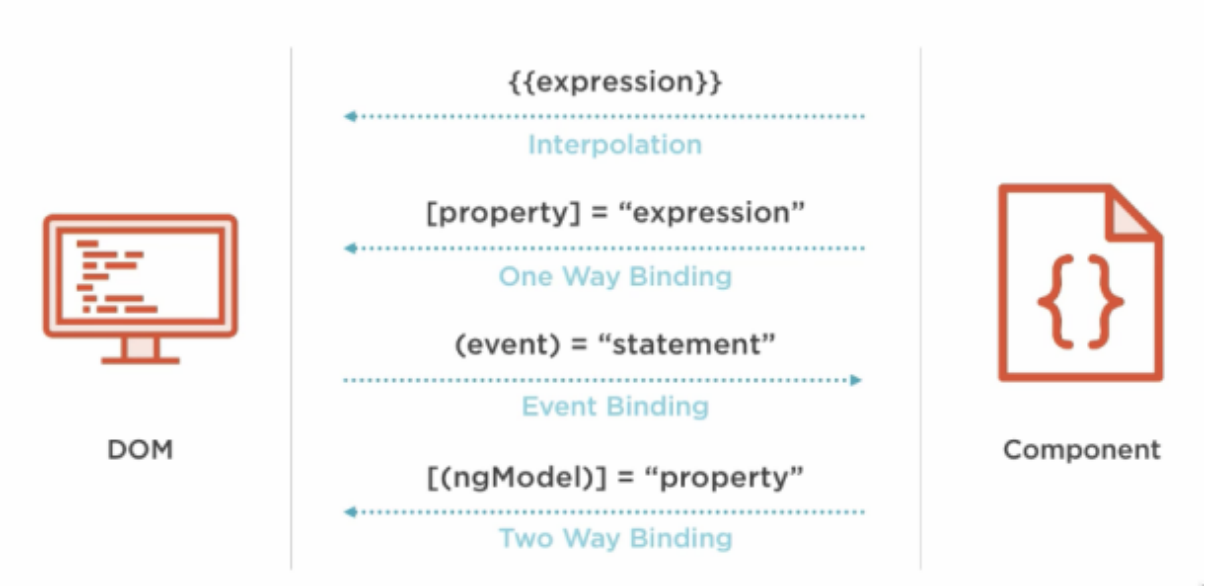
Lo que se ve en el código anterior es el concepto de **Data Binding**, es decir, "el enlace" existente entre una porción de UI y ciertos datos de una clase de un componente. En otras palabras, estamos diciendole a la UI que mire el valor de la property `pageTitle` de la clase. Si dicha property cambia, entonces el valor mostrado por la UI cambia.

De manera que cuando el HTML se renderiza, el HTML muestra el valor asociado al modelo pageTitle.

El Data Binding va de la mano del concepto de **interpolación**, la cual es la habilidad de poner datos dentro de un HTML (interpolar). Esto es lo que logramos con las

llaves dobles `{{ ... }}`.

La interpolación no es el único tipo de Data Binding, también hay otros:



- **Property Binding**: cuando el binding es hacia una property particular y no a una expresión compleja como puede ser la interpolación. Setea el valor de una property a un a expresíon en el template. Ver sintaxis en imagen de abajo.
- **Event Binding**: es binding hacia funciones o métodos que se ejecutan como consecuencia de eventos (por ejemplo: un click sobre un elemento).
- **Two-Way Binding**: Es un idea y vuelta entre el template y una property entre un component. Muestra el valor de la property en la vista, y si en la vista/template dicho valor cambia, la property también se ve reflejada (por eso es de dos caminos). Esto lo veremos con ms detalle en el tutorial de más abajo.