Clase 8 @October 13, 2022

Esta clase veremos

- Repaso de concepto de componentes
- Introduccion a las diferentes formas de realizar estilos (Angular Material, Bootstrap,Templates de otras organizaciones)
- Introduccion a los modulos
  - Que son
  - Analogia con Backend
  - Para que los necesito
  - Como los uso
  - Como los creo
- Proceso de inicio de nuestra aplicacion o Bootstrapping
  - Data Binding o Interpolacion
  - Directivas
  - Creacion del primer componente
  - Introduccion a bootstrap
  - Introduccion a Two-Way binding
  - Introduccion a los pipes de angular
  - Introduccion a Event Binding

Modulos

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

Los NgModules son contenedores, 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`. Inicia tu aplicación *cargando* el NgModule raíz.

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({
  declarations:[PeopleComponent ]
})

export class ContactsModule {}
```

En vez de esto, tendremos otro módulo, digamos `PeopleModule` que ya importa su propio `PersonComponent` 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({
```

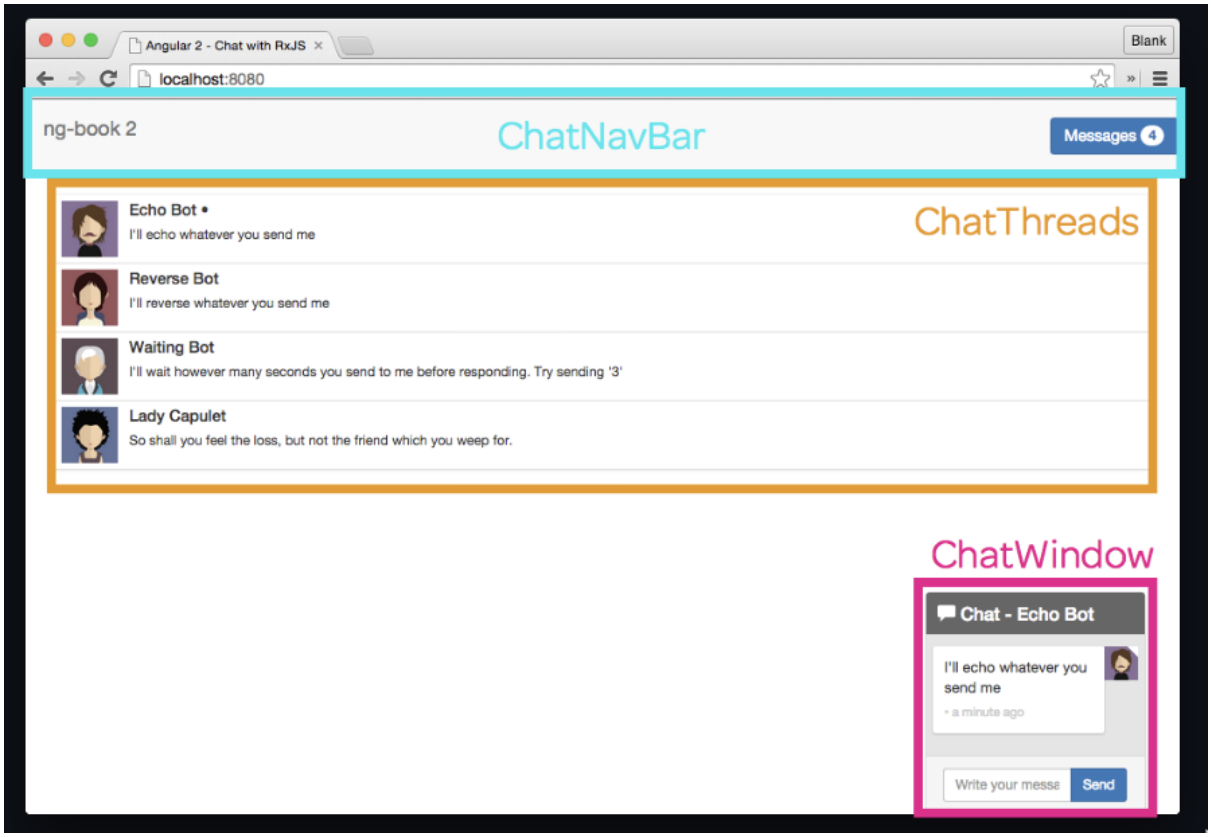
```
  imports:[PeopleModule ]
})

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 enrquecer la estructura. Por ejemplo podria tener un modulo de medicamento que agrupo el componente de agregar medicamento, el componente de eliminar medicamento y el componente de editar medicamento.

Repaso de concepto de componentes

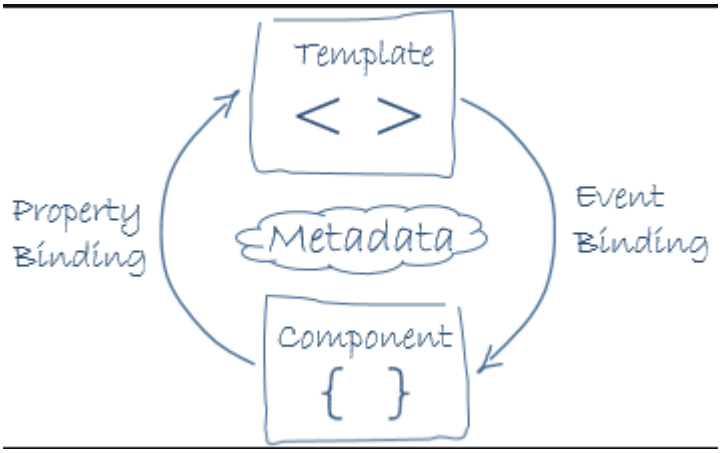
Cada componente la idea es que funcione armoniosamente y en conjunto con el resto para proveer una experiencia de usuario única. Como dijimos, estos son modulares, resuelven un problema concreto y colaboran entre sí para lograr ir armando la interfaz de usuario como un puzzle donde cada pieza tiene sus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

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



A su vez, es interesante recordar cómo se comporta internamente cada componente. Como habíamos dicho, los componentes se componen de tres cosas:

1. **Template:** El **template** del componente, 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. Estos dos conceptos los veremos más adelante en este módulo
2. **Clase:** A su vez la view tiene un código asociado, el cual llamamos la **clase** de un componente. 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.
3. **Metadata:** Finalmente, el componente también tiene **metadata**, que es información adicional para Angular, siendo esta definida con un *decorator* (los que arrancan con @). Un decorador es una función que agrega metadata a una clase, sus miembros o los argumentos de sus métodos.



La clase de un componente:

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

```
export class NombreComponent {
  property1: tipo = valor,
  property2: tipo2 = valor2,
  property3: tipo2 = valor3
  ...
}
```

Pero a su vez necesitamos utilizar el decorador `Component`, el cual lo debemos importar desde `@angular/core`:

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

@Component({
  selector: 'nombre-component',
  template: `<h1>Hello {{property1}}</h1>`
})
export class NombreComponent {
  property1: tipo = valor,
  property2: tipo2 = valor2,
  property3: tipo2 = valor3
  ...
}
```

También podemos definir un constructor:

```
export class NombreComponent {
  property1: string;
  property2: number;
```

```
  constructor(property1:string, property2: number) {
    this.property1 = property1
    this.property2 = property2;
  }
}
```

Algunos otros conceptos a tener en cuenta:

- Recordemos que por convención, el componente fundamental de una app de angular se llama AppComponent (el root component).
- La palabra reservada “export” simplemente hace que el componente se exporte y pueda ser visto por otros componentes de nuestra app.
- La sintaxis de definición del archivo es `nombre.component.ts`.
- El valor por defecto en las properties de nuestros componentes es opcional.
- Los métodos vienen luego de las properties, en lowerCamelCase.

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 ms 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.

Angular precisa de esa metadata para saber como instanciar el componente, estructurar la view, y ver la interacción:

- Usamos un decorator, siendo el scope de tal decorator la feature que decora. Siempre son prefijos con un `@`.
- Angular tiene un conjunto predefinido de decoradores que podemos usar, y hasta podemos crearnos los propios.
- El decorator va siempre antes de la definición de la clase, como las DataAnnotations en .NET (no va ;)
- Es una función y recibe un objeto que define el template.

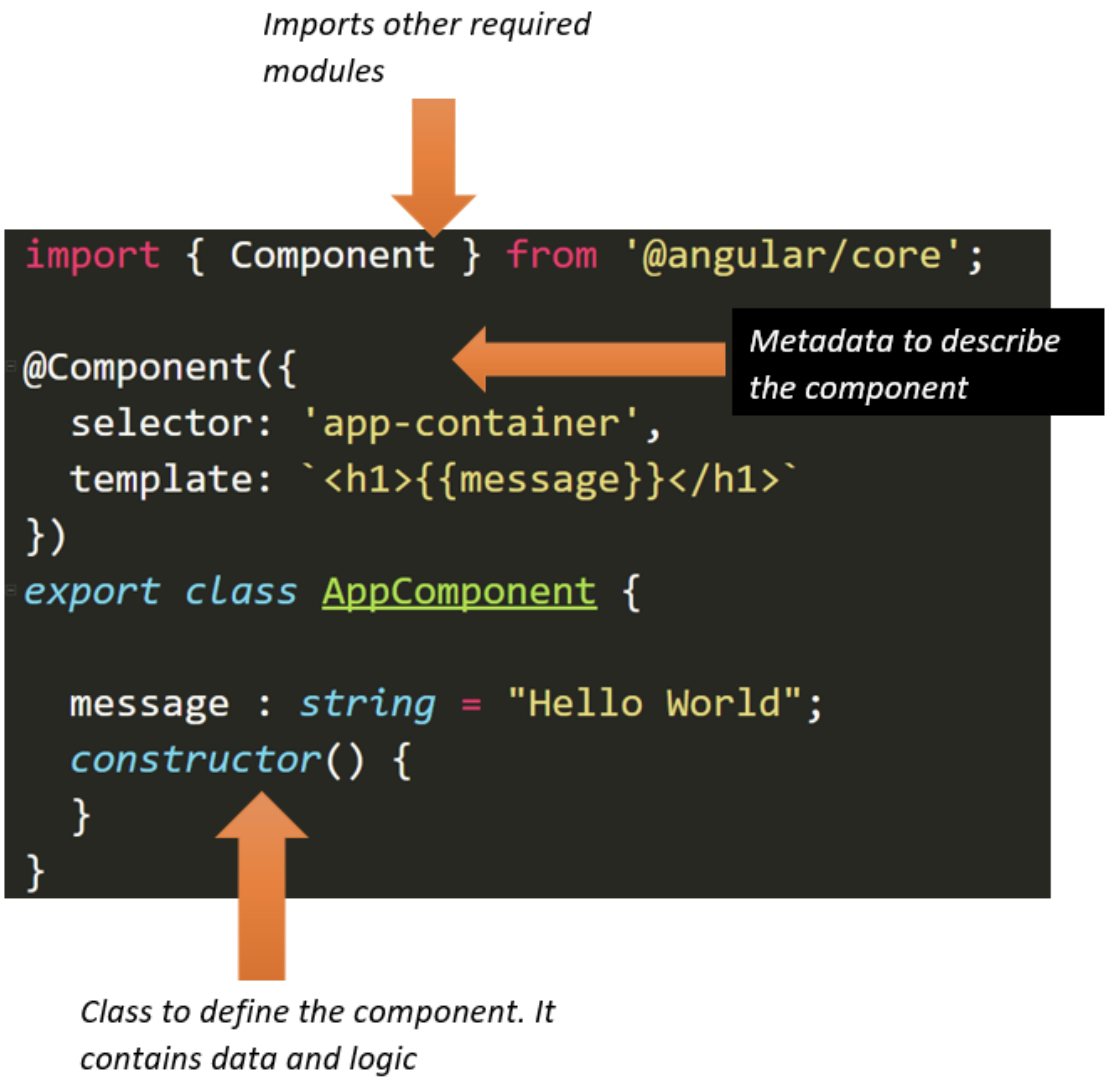
En otras palabras el decorator `@Component` indica que la clase subyacente es un Componente de Angular y recibe la metadata del mismo (en forma de un objeto

JSON de configuración). Aquí van 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.

Especifica el nombre de la directiva que vamos a usar con el componente. Es simplemente un tag HTML custom que vamos a poder usar.

1. **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.
2. **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.
3. **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. Ejemplo:



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:

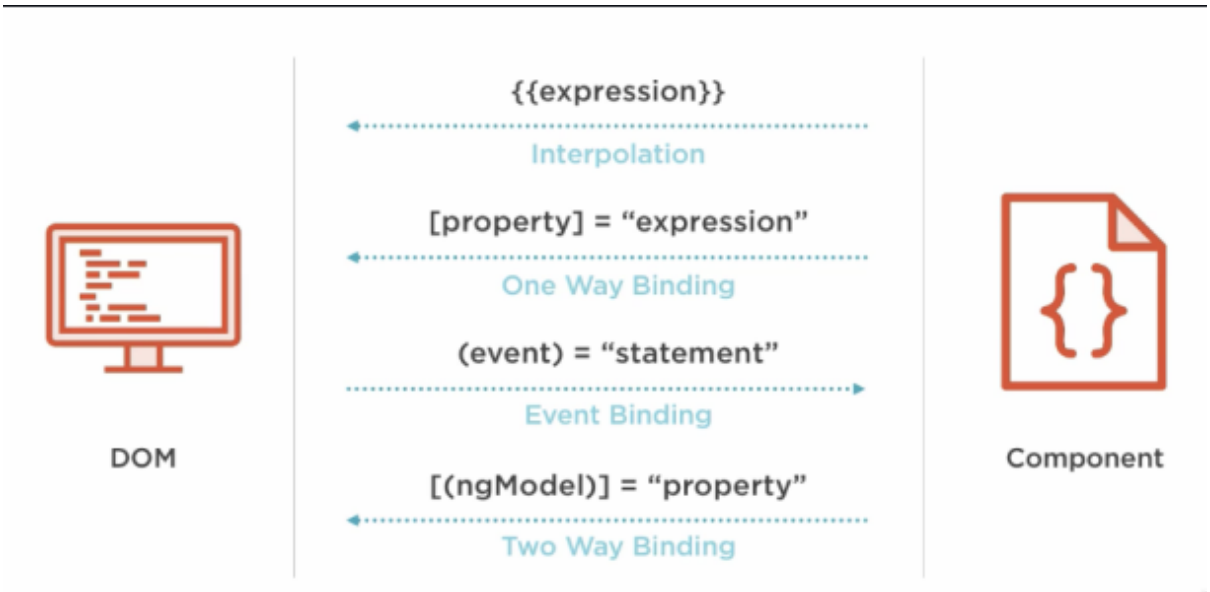
```
<div class='panel-heading'>
    {{pageTitle}}
</div>
```

```
export class MyComponent {
  pageTitle: string = "Nombre de la página que puede cambiar"
  constructor(pageTitle : string)
  {
    this.pageTitle = pageTitle;
  }
}
```

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.

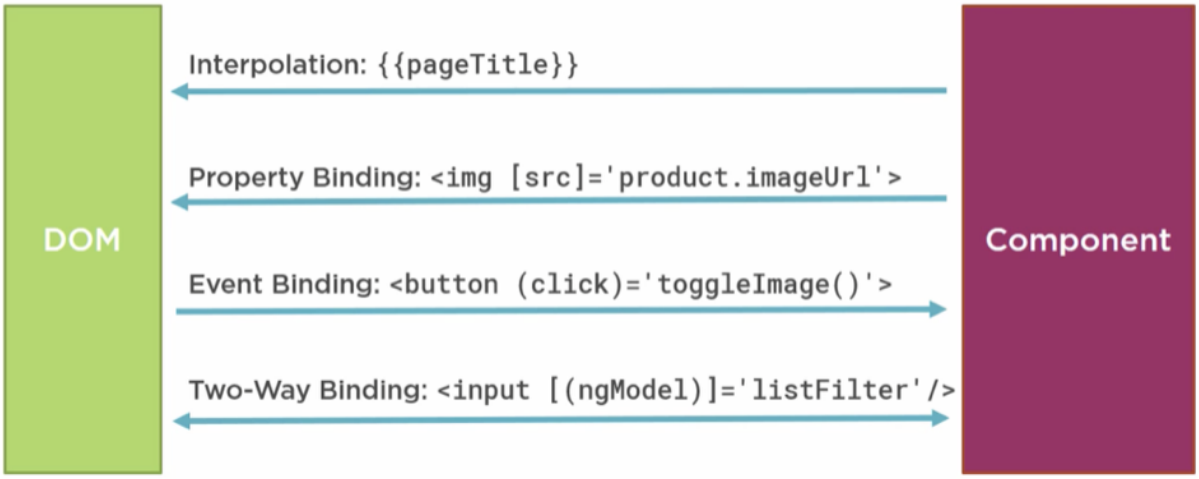
De manera que cuando el HTML se renderiza, el HTML muestra el valor asociado al modelo pageTitle.

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.



Mas sobre interpolación

La interpolación soporta mucho más que el mostrado properties simples, también permite realizar operaciones complejas o cálculos, o llamar métodos!

Template

```
<h1>{{pageTitle}}</h1>
{{'Title: ' + pageTitle}}
{{2*20+1}}
{{'Title: ' + getTitle()}}
<h1 innerText={{pageTitle}}></h1>
```

Class

```
export class AppComponent {
 pageTitle: string =
     'Acme Product Management';
 getTitle(): string {...};
}
```

Agregando Event Binding para los Exercises

Usaremos **Event Binding** para mandar información al revés de los tipos de Data Binding como la Interpolación o el Property Binding. En lugar de que el las properties de nuestra clase manden datos al template (o vista), esta vez será el template o vista quien se comunicará con la clase. Esto lo hace a partir de responder a eventos del usuario, por ejemplo un click, un mouse over, un copy o paste, un scroll, un tecleo, etc. La información de eventos disponibles que podemos usar se encuentra bien documentada en: https://developer.mozilla.org/en-US/docs/Web/Events La idea es que nuestros componentes estén funcionando como "listeners" a las acciones del usuario, usando Event Binding, y pudiendo ejecutar cierta lógica particular.

La sintaxis es por ejemplo:

```
<button (click)='hacerAlgoCuandoOcurreClick()'> </button>
```

- El nombre del evento va entre paréntesis.
- La lógica a ejecutar va entre comillas simples luego del igual.

Directivas

A su vez, también podemos enriquecer nuestro HTML a partir de lo que se llaman **directivas**, pudiendo agregar **ifs** o **loops** (estilo for), sobre datos en nuestro HTML y generar contenido dinámicamente.

Una directiva es un elemento custom del HTML que usamos para extender o mejorar nuestro HTML. Cada vez que creamos un componente y queremos renderizar su template, lo hacemos a través de su *selector* asociado, el cual define la directiva del componente.

Pero a su vez angular también tiene algunas directivas built-in, sobre todo las *structural directives*. Por ejemplo: ***ngIf** o ***ngFor** (los asteriscos marcan a las directivas como que son estructurales).

Pipes

Cuando los datos no están en el formato apropiado que queremos para mostrarlos, usamos Pipes. Estos permiten aplicar cierta lógica sobre las properties de nuestra clase antes de que estas sean mostradas (por ejemplo para que sean más user friendly). Angular ya provee varios Pipes built-in para diferentes tipos de datos (date, number, decimal,json, etc), e incluso permite crear pipes propios para realizar lógica particular como lo es manejar el filtrado.Los pipes en general se denotan con el caracter `|` (pipe), expresión.

```
{{ product.productCode | lowercase }}

<img [src]='product.imageUrl'
     [title]='product.productName | uppercase'>

{{ product.price | currency | lowercase }}

{{ product.price | currency:'USD':true:'1.2-2' }}
```

Ejemplo integrador con lo visto anteriormente y agregando Bootstrap

Ahora veremos la creación de un componente, agregarlo a nuestro módulo principal, trabajaremos con templates, data binding, interpolación y directivas.

1. Instalamos Bootstrap

Instalamos la librería de Twitter Bootstrap (nos da estilos y nos permite lograr diseños responsive de forma simple). Para ello, parados sobre nuestro proyecto usamos npm para descargarla (recordemos que npm es como Nuget pero para librerías o módulos de JavaScript):

```
npm install bootstrap@3 --save
```

El --save lo que hace es guardar la referencia a este módulo en el package.json

Vemos como se impacta el package.json

```
"dependencies": {
    "@angular/animations": "^6.1.0",
    "@angular/common": "^6.1.0",
    "@angular/compiler": "^6.1.0",
    "@angular/core": "^6.1.0",
    "@angular/forms": "^6.1.0",
    "@angular/http": "^6.1.0",
    "@angular/platform-browser": "^6.1.0",
    "@angular/platform-browser-dynamic": "^6.1.0",
    "@angular/router": "^6.1.0",
    "bootstrap": "^3.3.7", //aca aparecio boostrap
    "core-js": "^2.5.4",
    "rxjs": "~6.2.0",
    "zone.js": "~0.8.26"
  },
```

Y en el `angular.json` agregamos (en projects/:NOMBRE DEL PROJECTO:/styles) para poder usarlo en nuestra aplicacion.

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
],
```

2. Creamos nuestro componente

Para eso lanzaremos el siguiente commando `ng generate component HomeworksList` Una carpeta llamada homeworks-list, con 4 archivos:

- homeworks-list.component.spec.ts (Archivo con pruebas) (Se puede tanto eleminar como mover a la carpeta e2e)
- homeworks-list.component.ts (Archivo con la clase del componente y la metadata)
- homeworks-list.component.html (Archivo que contiene la vista)
- homeworks-list.component.css (Archivo que contiene el css del componente)

Este commando nos agrega el componente automaticamnte al `app.module.ts` dentro del array declarations

```
{
  declarations: [
    AppComponent,
    HomeworksListComponent,
    HomeworksFilterPipe,
```

```
  ],
....
```

3. Agregamos el html.

Agregamos en nuestro archivo de vista (`homeworks-list.component.html`) nuestro template basico.

Particularmente utilizaremos la propiedad `templateUrl` luego en nuestro componente:

```
<div class='panel panel-primary'>
    <div class='panel-heading'>
        Homeworks List
    </div>

    <div class='panel-body'>
        <!-- Aca filtramos las tareas  -->
        <!-- Selector de filtro:  -->
        <div class='row'>
            <div class='col-md-2'>Filter by:</div>
            <div class='col-md-4'>
                <input type='text' />
            </div>
        </div>
        <!-- Muestra filtro:  -->
        <div class='row'>
            <div class='col-md-6'>
                <h3>Filtered by: </h3>
            </div>
        </div>

        <!-- Mensaje de error -->
        <div class='has-error'> </div>

        <!--Tabla de tareas -->
        <div class='table-responsive'>
            <table class='table'>
                <!--Cabezal de la tabla -->
                <thead>
                    <tr>
                        <th>Id</th>
                        <th>Description</th>
                        <th>DueDate</th>
                        <th>Score</th>
                        <th>
                            <button class='btn btn-primary'>
                               Show Exercises
                            </button>
                        </th>
                    </tr>
                </thead>
```

```
                <!--Cuerpo de la tabla-->
                <tbody>
                    <!-- Aca va todo el contenido de la tabla  -->
                </tbody>
            </table>
        </div>
    </div>
</div>
```

4. El código del componente

Modificamos el archivo `homeworks-list.component.ts` y le agregamos el siguiente código:

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

@Component({
  selector: 'app-homeworks-list',
  templateUrl: './homeworks-list.component.html',
  styleUrls: ['./homeworks-list.component.css']
})
export class HomeworksListComponent implements OnInit {
    pageTitle:string = "Homeworks List"

    constructor() { }

    ngOnInit() {
    }
}
```

5. Agregamos el componente nuevo a través de su selector.

Lo que haremos aquí es usar el selector `app-homeworks-list` en el root component, es decir el AppComponent.

De manera que en `app.component.ts` quedaría algo como:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
    title:string = 'Homeworks Angular';
    name:string = "Santiago Mnedez";
    email : string = "santi17mendez@hotmail.com";
    address = {
        street: "la dirección del profe",
```

```
        city: "Montevideo",
        number: "1234"
    }
}
```

Y nuestro `app.component.html`:

```
<app-homeworks-list></app-homeworks-list>
```

Sin embargo, con esto no basta, ya que para que un componente pueda usar a otro componente (a través de su selector), estos deben pertenecer al mismo módulo, o el módulo del componente que importa debe importar al mdulo del otro componente.

En consecuencia, vamos a `app.module.ts`, y aseguraros que se encuentre el import:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { HomeworksListComponent } from './homeworks-list/homeworks-list.component';

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

¿Como hace el componente para saber a dónde buscar el component? Cómo ya dijimos, ahora lo encuentra porque pertenecen al mismo modulo. El módulo que sea dueño de este component es examinado para encontrar todas las directivas que pertenecen al mismo.

6. Usando Data Binding para mostrar datos dinmicos

Hacer cambio en el `homeworks-list.component.html` y poner:

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

Veamos que pasa.

7. Utilizando *ngIf para elegir hacer algo o no

En el template, cambiamos `<table clas="table">` por lo siguiente:

```
<table class='table' *ngIf='homeworks && homeworks.length'>
```

Esto todava no va a tener resultado hasta que en el paso siguiente agreguemos la property 'homeworks'.

8. Utilizando *ngFor para iterar sobre elementos de forma dinamica

Ahora crearemos la clase Exercise y Homeworks (en una carpeta llamada models), y agregaremos la propiedad 'homeworks' a nuestro componente.

Clase Exercise en la carpeta models:

```
export class Exercise {
    id: string;
    problem: string;
    score: number;

    constructor(id:string = "", problem:string = "", score:number = 0) {
        this.id = id;
        this.problem = problem;
        this.score = score;
    }
}
```

Clase Homework en la carpeta models:

```
import { Exercise } from './Exercise';

export class Homework {
    id: string;
    description: string;
    dueDate: Date;
    score: number;
    exercises: Array<Exercise>;

    constructor(id:string, description:string, score:number, dueDate:Date, exercises:
 Array<Exercise>){
        this.id = id;
        this.description = description;
```

```
        this.score = score;
        this.dueDate = dueDate;
        this.exercises = exercises;
    }
}
```

Nuestro componente:

```
import { Component, OnInit } from '@angular/core';
import { Homework } from '../models/Homework';
import { Exercise } from '../models/Exercise';

@Component({
  selector: 'app-homeworks-list',
  templateUrl: './homeworks-list.component.html',
  styleUrls: ['./homeworks-list.component.css']
})
export class HomeworksListComponent implements OnInit {
    pageTitle:string = "Homeworks List";
    homeworks:Array<Homework> = [
        new Homework("1", "Una tarea", 0, new Date(), [new Exercise("1", "Un Problem
a", 0)]),
        new Homework("2", "Otra tarea", 0, new Date(), [])
    ];

    constructor() { }

    ngOnInit() {
    }
}
```

Y en el template cambiamos el `<tbody>` por lo siguiente:

```
<tbody>
  <tr *ngFor='let aHomework of homeworks'>
      <td>{{aHomework.id}}</td>
      <td>{{aHomework.description }}</td>
      <td>{{aHomework.dueDate}}</td>
      <td>{{aHomework.score}}</td>
      <td>
          <div>
              <table>
                  <thead>
                      <tr>
                          <th>Problem</th>
                          <th>Score</th>
                      </tr>
                  </thead>
                  <tbody>
                      <tr *ngFor='let aExercise of aHomework.exercises'>
                          <td>{{aExercise.problem}}</td>
                          <td>{{aExercise.score}}</td>
```

```
                    </tr>
                </tbody>
            </table>
        </div>
    </td>
  </tr>
</tbody>
```

9. Agregando Two-Way Binding:

En nuestro HomeworksListComponent, agregamos la property listFilter:

```
text: string='';
```

En el template asociado, reemplazamos los dos primeros divs de class "row" que aparecen:

```
<div class='row'>
    <div class='col-md-2'>Text:</div>
    <div class='col-md-4'>
        <input type='text' [(ngModel)]='text' />
    </div>
</div>
<div class='row' *ngIf='text'>
    <div class='col-md-6'>
        <h3>Text: {{text}} </h3>
    </div>
</div>
```

Vemos que no nos anda.

Para ello vamos al `app.module.ts` y agregamos el import a FormsModule:

```
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { HomeworksListComponent } from './homeworks-list/homeworks-list.component';

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

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

10. Usando Pipes en Angular

Para ello, simplemente cambiamos:

```
<td>{{aHomework.description  | uppercase}}</td>
... // o
<td>{{aHomework.description  | lowercase}}</td>
```

11. Agregando Event Binding para los Exercises

Lo que haremos ahora es la lógica del mostrado de imagenes con Event Binding, para ello: En `homeworks-list.component.ts`, agregamos la siguiente property a la clase:

```
showExercises:boolean = false;
```

A su vez agregamos la siguiente función:

```
toggleExercises(): void {
    this.showExercises = !this.showExercises;
}
```

Quedando:

```
import { Component, OnInit } from '@angular/core';
import { Homework } from '../models/Homework';
import { Exercise } from '../models/Exercise';

@Component({
  selector: 'app-homeworks-list',
  templateUrl: './homeworks-list.component.html',
  styleUrls: ['./homeworks-list.component.css']
})
export class HomeworksListComponent implements OnInit {
    pageTitle:string = "Homeworks List";
    listFilter:string = "";
```

```
    showExercises:boolean = false;
    homeworks:Array<Homework> = [
        new Homework("1", "Una tarea", 0, new Date(), [new Exercise("1", "Un Problem
a", 0)]),
        new Homework("2", "Otra tarea", 0, new Date(), [])
    ];

    constructor() { }

    ngOnInit() {
    }

    toggleExercises(): void {
        this.showExercises = !this.showExercises;
    }

}
```

Y en el template hacemos estos dos cambios:

1. En cada click al botón que tenemos en el header de la tabla, llamamos a la función `toggleExercises()`:

```
<button (click)='toggleExercises()'class='btn btn-primary'>
    {{showExercises ? 'Hide' : 'Show'}} Exercises
</button>
```

1. En el mostrado de los Exercises, agregamos la condición de que solo se muestre si la property lo indica.

```
<div *ngIf='showExercises'>
    <table>
        <thead>
            <tr>
                <th>Problem</th>
                <th>Score</th>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor='let aExercise of aHomework.exercises'>
                <td>{{aExercise.problem}}</td>
                <td>{{aExercise.score}}</td>
            </tr>
        </tbody>
    </table>
</div>
```