|  |  |
| --- | --- |
| Guía Introductoria: Angular 7 + Bootstrap 4  **Cristhian Leonardo Arrieta Botello**  Desarrollador JavaScript  Cristhian280196@gmail.com |  |
|  |  |

**Documento preparado por Azurian - Chile**

Tabla de Contenido

[1. Introducción 2](#_Toc18062422)

[2. Angular 7 3](#_Toc18062423)

[3.1. Definición 3](#_Toc18062424)

[3.2. Ventajas 3](#_Toc18062425)

[3.3. Estructura 4](#_Toc18062426)

[3. Boostrap 4 5](#_Toc18062427)

[3.1. Definición 5](#_Toc18062428)

[3.2. Fundamentos 5](#_Toc18062429)

[4. Ejercicio Práctico: Desarrollo aplicación “my-app” 7](#_Toc18062430)

[4.1. Configuración de ambiente y creación de proyecto 7](#_Toc18062431)

[4.2. Creación de módulos 8](#_Toc18062432)

[4.3. Configuración de appModule, enrutador, layoutModule, viewsModule y sharedModule 8](#_Toc18062433)

[4.4. Desarrollo de module1 14](#_Toc18062434)

[4.5. Desarrollo de module2 20](#_Toc18062435)

[4.6. Variables de ambiente 27](#_Toc18062436)

[4.7. Deployment en tomcat 27](#_Toc18062437)

# 1. Introducción

El presente documento tiene como finalidad, presentar el contenido que será abordado en el Taller de Angular 7 + Boostrap 4 impartido en Azurian como parte del entrenamiento del personal de desarrollo y demás áreas relacionadas.

# 2. Angular 7

## 3.1. Definición

Angular es un framework usado para desarrollar single-page web apps (SPA) unificando HTML y Javascript de una forma mucho más cómoda y eficiente al programar. Google es el encargado de su desarrollo y mantenimiento desde su inicio en 2012, para ese entonces llamado AngularJs. En 2016 se lanzó su versión más popular y recurrida, cambiando su nombre a Angular 4.

Este framework está programado en TypeScript, siendo éste un superconjunto de Javascript. Nos resulta de gran utilidad ya que extiende el poder nativo de Javascript permitiendo usar interfaces y clases, haciendo aún más robusta nuestra aplicación.

Angular está basado en la arquitectura modelo-vista-controlador (MVC):

* **Modelo**: estructura de data que maneja la información que recibe del controlador.
* **Controlador**: hace de intermediario entre la vista el modelo.
* **Vista**: presenta al modelo en un formato amigable para el usuario.

En el contexto de Angular:

* Angular enlaza el Javascript y el HTML.
* Javascript toma la entrada de data que llega desde Angular, la manipula, y la envía de regreso a Angular.
* Angular usa la entrada de datos y modifica los elementos HTML.

## 3.2. Ventajas

* **Two-way data binding**: nos permite modificar el valor de una propiedad de forma bidireccional, de forma que cuando modificamos el valor de una propiedad en la vista el valor también se actualiza en el modelo de datos.
* **Directivas**: extiende las directivas nativas de HTML.
* **Estructura de código**: Angular provee plantillas, dando como resultado un código más limpio y homogéneo.
* **Testeo**: permite pruebas unitarias y de integración.
* **Compatibilidad**: Angular funciona en casi todos los browsers, incluyendo móviles.

## 3.3. Estructura

* **Módulos**: las apps son modulares; Angular tiene su propio sistema llamado NgModule. Éstos son contenedores de bloques de código (componentes, servicios y otros).
  + **AppModule**: es el módulo principal y toda app de Angular posee uno. Aquí se unen los módulos secundarios y se hacen las configuraciones generales de la app.
  + Módulo **core**:guarda los servicios globales. Éste lo usaremos de modo singleton; se llamará al módulo una sola vez en el AppModule.
  + Módulo **shared**: almacena los componentes y servicios compartidos. Se debe importar en cada módulo UI que requiera utilizarlo. No se debe importar en el AppModule.
  + Módulo **UI:** guarda los componentes que poseen el UI (interfaz de usuario) de la aplicación.
  + Módulo **views**: guarda los módulos UI.
  + Módulo **layout** (preferencia personal): guarda la plantilla HTML que tendrá la aplicación (tema).
* **Componentes**: es la forma más básica para crear UI que brinda Angular. Los componentes son elementos o subconjuntos de elementos asociados a un HTML. Se instancian mediante un selector.
  + Contenedor: por buena práctica, todos los componentes de interacción directa con el usuario van dentro de un wrapper padre llamado Container.
  + Componente hijo: generalmente son los componentes que poseen el UI, por ejemplo, una barra de búsqueda.
* **Router**: permite mostrar los componentes de la app dependiendo de dónde se encuentre el usuario (URL). El router controla la navegación de una vista a otra. Angular nos permite emplear lazy loading, el cual consiste en cargar los módulos a medida que va navegando el usuario, obteniendo mejor rendimiento.
* **Service**: hace posible la comunicación entre dos clases que no se conocen entre ellas. También nos permite consumir servicios REST.

# 3. Boostrap 4

## 3.1. Definición

Es un framework open-source que facilita el diseño web orientado a la programación responsiva. Está hecho en SCSS y JQuery, brindándonos todo tipo de componentes personalizados, desde tipografías, hasta botones y formularios.

Bootstrap funciona mediante clases de CSS previamente definidas, facilitando la personalización de los elementos HTML.

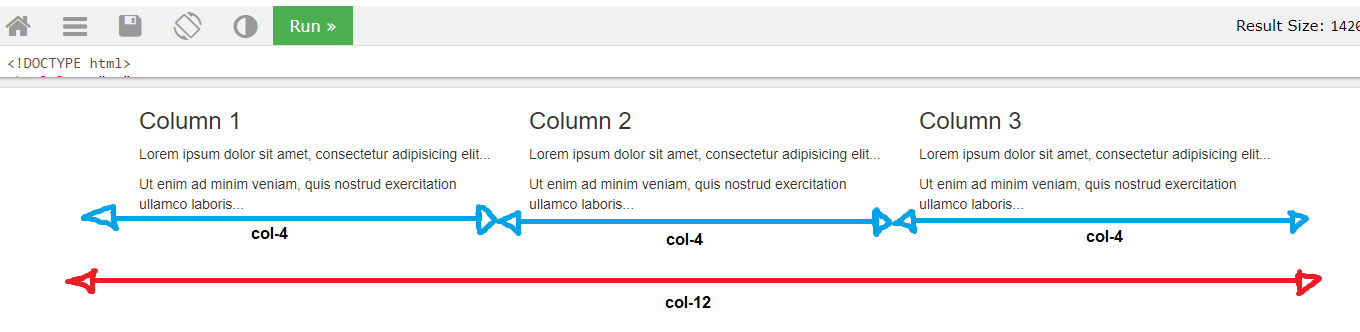
## 3.2. Fundamentos

Bootstrap está basado para actuar dependiendo del tamaño del viewport (resolución del navegador), esperando cuatro posibilidades:

* Sm: < 768 px (small).
* Md: 768 – 992 px (medium).
* Lg: 992 – 1200 px (large).
* Xl: 1200 px > (extra large).

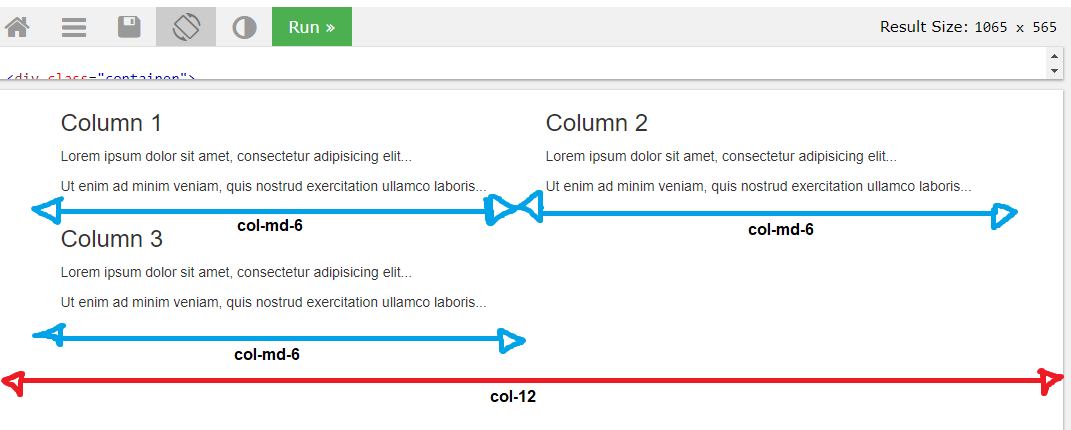
Podemos manipular los elementos HTML (generalmente tamaño y posición) a partir de estos tamaños. Bootstrap implementa cuadrículas (grids) en sus patrones de diseño.

**Lógica de Bootstrap:** los grid se dividen en doce columnas (col) de igual tamaño, por lo que podemos asignar un máximo de doce espacios (col-12) a un elemento, siendo el número de columnas que tome dicho elemento modificables según el tamaño del viewport. Por ejemplo, tenemos tres elementos que queremos alinear en la misma fila, entonces tendríamos que dividir las doce columnas entre tres (12 / 3), por lo que cada elemento debe ocupar cuatro (col-4) espacios de columna para que los tres se encuentren en la misma fila.

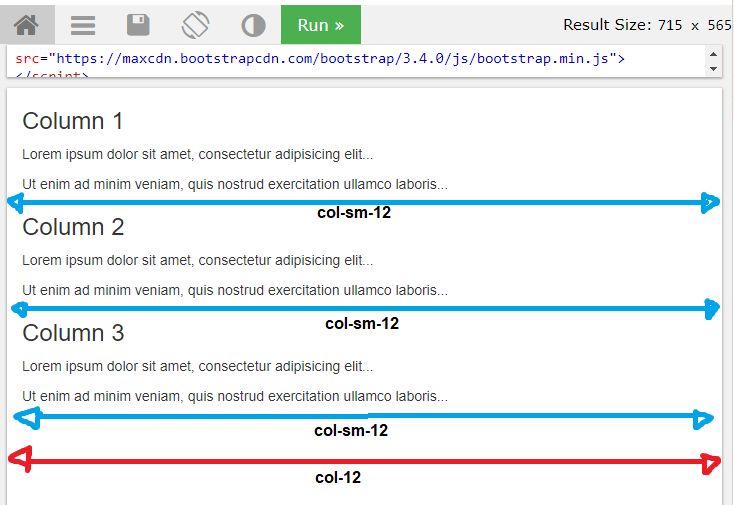


Como previamente mencionaba, el tamaño de los elementos y posicionamiento se da mediante clases, en este caso col-4. Para indicarle a Bootstrap el tamaño que queremos que tome el elemento dependiendo del tamaño del viewport, debemos utilizar un prefijo en la clase col, indicando el tamaño del viewport, por ejemplo:

* Viewport tamaño médium (md):



* Viewport tamaño small (sm):



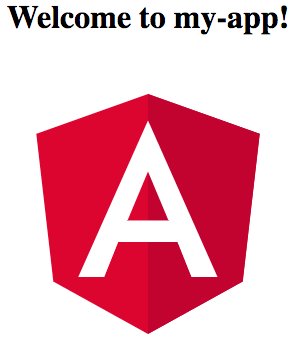
# 4. Ejercicio Práctico: Desarrollo aplicación “my-app”

## 4.1. Configuración de ambiente y creación de proyecto

Pre-requisitos:

* Conocimientos básicos-intermedios en HTML y Javascript.
* Node.js versión 10.9.0 o posterior.
* Manejador de paquetes Npm (Node.js lo trae por defecto).
* **Paso 1**: instalar el Angular CLI mediante el comando:
* ***npm install -g @angular/cli***
* **Paso 2:** crear el ambiente de trabajo y una aplicación inicial mediante el comando:
  + ***ng new my-app --routing***
    - ***ng new*** nos genera una aplicación inicial. En la consola se nos pedirán los datos informativos acerca de la app.
    - ***my-app*** es el nombre que elegimos para la app.
    - ***--routing*** esta opción genera un enrutador para el módulo principal de la app.
* **Paso 3:** moverse a la carpeta creada en el paso anterior (my-app) y hacerle launch a la app.
  + ***cd my-app***
  + ***ng serve --open***
    - Este comando construye la app, inicia el servidor de desarrollo, chequea los archivos fuente y reconstruye la app al detectar un cambio.
    - La opción **--open** nos abrirá automáticamente el navegador con la URL del servidor de desarrollo (<http://localhost:4200/>).

Deberíamos ver lo siguiente en el navegador:



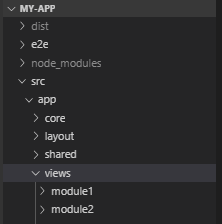
* **Paso 4:** instalar ngx-bootstrap mediante el siguiente comando:
  + ***npm install ngx-bootstrap --save***

## 4.2. Creación de módulos

El comando ***ng generate module rutadelmodulo*** genera los módulos. Una vez situados en la carpeta root de la aplicación, ejecutamos la siguiente línea de comandos:

* ***ng generate module core***
* ***ng generate module layout***
* ***ng generate module shared***
* ***ng generate module views --routing***

Obteniendo la siguiente estructura:



## 4.3. Configuración de appModule, enrutador, layoutModule, viewsModule y sharedModule

* El **appModule** debe quedar de la siguiente forma:

import { BrowserModule } from '@angular/platform-browser';

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

import { AppRoutingModule } from './app-routing.module';

import { LayoutModule } from './layout/layout.module';

import { HttpClientModule } from '@angular/common/http';

import { CoreModule } from './core/core.module';

import { AppComponent } from './app.component';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule,

LayoutModule,

HttpClientModule,

CoreModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

* **BrowserModule** y **AppRoutingModule** vienen por defecto al crear la app; el primero importa toda la infraestructura de la aplicación y el segundo es nuestro enrutador principal.
* **LayoutModule** es el módulo que posee la plantilla HTML de la app, específicamente en el componente **Theme.** Lo debemos llamar en el appModule para poder usar **Theme** en el enrutador.
* **HttpClientModule** nos permitirá consumir servicios REST.
* El router principal **app-routing.module.ts**:

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

import { Routes, RouterModule } from '@angular/router';

import { ThemeComponent } from './layout/theme/theme.component';

import { NotFoundComponent } from './layout/not-found/not-found.component';

const routes: Routes = [

{

path: '',

component: ThemeComponent,

loadChildren: './views/views.module#ViewsModule'

},

{

path: '\*\*',

component: NotFoundComponent

}

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

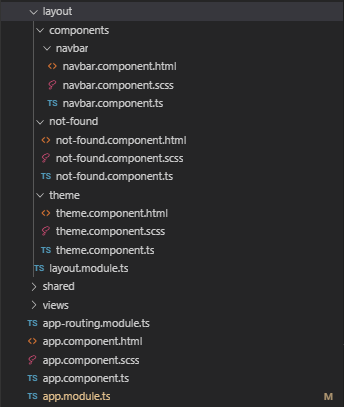
export class AppRoutingModule { }

En la variable **routes** definimos las rutas, las cuales se definen mediante JSONs. Con la propiedad **path** definimos el URL, con **component** el componente al que queremos llamar cuando se llegue al URL, **loadChildren** cargamos un módulo, el cual tiene su propio enrutador. Los componentes que el módulo hijo cargue los podremos referenciar mediante el **<router-outlet>** que veremos más adelante).

Con **path: ‘\*\*’** manejamos las URL que no están definidas.

* Para **layoutModule** ejecutamos los siguientes comandos:
  + ***ng generate component layout/components/navbar***: creamos la barra de navegación que usará el tema.
  + ***ng generate component layout/not-found***: creamos el componente que usaremos para **path: ‘\*\*’**.
  + ***ng generate component layout/theme*:** creamos la estructura HTML (tema) de la app en general.

Obteniendo la siguiente estructura:



* + En **layout.module.ts:**

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

import { RouterModule } from '@angular/router';

import { CommonModule } from '@angular/common';

import { CollapseModule } from 'ngx-bootstrap/collapse';

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ThemeComponent } from './theme/theme.component';

import { NavbarComponent } from './components/navbar/navbar.component';

import { NotFoundComponent } from './not-found/not-found.component';

@NgModule({

declarations: [

ThemeComponent,

NotFoundComponent,

NavbarComponent

],

imports: [

CommonModule,

RouterModule,

CollapseModule.forRoot(),

BrowserAnimationsModule

]

})

export class LayoutModule { }

Los módulos se importan y se definen en **imports**. Los componentes se importan y se definen en **declarations**.

* + En **navbar.componente.html:**

<nav class="navbar navbar-expand-lg navbar-light bg-light mb-2 border">

<a class="navbar-brand"><b>My</b> App!</a>

<button class="navbar-toggler" type="button" (click)="isCollapsed = !isCollapsed">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse ml-3 ml-md-5" id="navbarNav" [collapse]="isCollapsed">

<ul class="navbar-nav">

<li class="nav-item">

<a class="nav-link" routerLink="/modulo1" routerLinkActive="text-success">Módulo 1</a>

</li>

<li class="nav-item">

<a class="nav-link" routerLink="/modulo2" routerLinkActive="text-success">Módulo 2</a>

</li>

</ul>

</div>

</nav>

Se usa la directiva **routerLink** para cambiar la URL del enrutador y la directiva **routerLinkActive** para añadir o eliminar una clase dependiendo si el actual URL hace match con el **routerLink**.

* + En **not-found.component.html**:

<h5 class="text-center font-weight-bold">

URL inexistente.

</h5>

* + En **theme.component.html**:

<div class="col-12 col-md-8 mx-md-auto mt-3">

<h4 class="text-center font-weight-bold">

<span class="font-weight-light">My </span>App!

</h4>

<app-navbar></app-navbar>

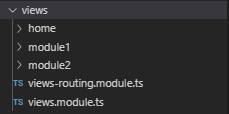
<router-outlet></router-outlet>

</div>

Donde **<app-navbar>** es el selector de nuestra barra de búsqueda que creamos previamente, **<router-outlet>** nos cargará los componentes hijos que el enrutador nos pase; en nuestro caso cargamos un módulo como hijo, por lo que se cargarán los componentes que el enrutador del módulo hijo nos indique (lazy loading).

* Para **viewsModule** ejecutamos los siguientes comandos:
* ***ng generate module views/module1 --routing***
* ***ng generate module views/module2 --routing***
* ***ng generate component views/home***

Obteniendo la siguiente estructura:



* + En **views-routing.module.ts**:

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

import { Routes, RouterModule } from '@angular/router';

import { HomeComponent } from './home/home.component';

const routes: Routes = [

{

path: '',

redirectTo: 'home'

},

{

path: 'home',

component: HomeComponent

},

{

path: 'modulo1',

loadChildren: './module1/module1.module#Module1Module'

},

{

path: 'modulo2',

loadChildren: './module2/module2.module#Module2Module'

}

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule]

})

export class ViewsRoutingModule { }

En el primer objeto le indicamos al enrutador redireccionar a la ruta ‘home’ cuando el URL esté vacío. En el segundo objeto cargamos el componente **HomeComponent** para la ruta ‘home’.

También podemos observar que, de nuevo, cargamos un módulo hijo para la ruta. El enrutador se puede encadenar a otro enrutador, y el componente final que se nos muestre será accesible desde cualquier lado mediante **<router-outlet>**.

* + En **home.component.html**:

<div class="card">

<div class="card-header">

<h5 class="py-3 text-center">

Guía introductoria a <b>Angular 7 + Bootstrap 4.</b>

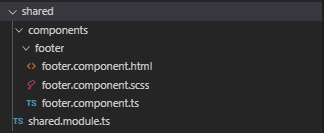
</h5>

</div>

</div>

* Para **sharedModule** ejecutamos el siguiente comando:
  + ***ng generate component shared/components/footer***

Obteniendo la siguiente estructura:



* + En **shared.module.ts**:

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

import { CommonModule } from '@angular/common';

import { FooterComponent } from './components/footer/footer.component';

@NgModule({

declarations: [FooterComponent],

imports: [

CommonModule

],

exports: [FooterComponent]

})

export class SharedModule { }

Declaramos el componente Footer en **declarations**, pero también en **exports** para que éste sea accesible desde módulos externos.

* + En **footer.component.html**:

<footer class="text-center rounded text-white bg-secondary py-3">

Componente compartido mediante el Shared Module.

</footer>

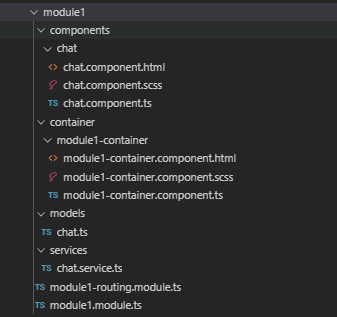
## 4.4. Desarrollo de module1

En el **module1** crearemos un chat local. Los componentes, al ser ambos padres, se comunicarán por medio de un servicio de Angular.

Ejecutamos los siguientes comandos:

* ***ng generate component views/module1/container/module1-container****:* creamos el componente contenedor de la vista.
* ***ng generate component views/module1/components/chat****: creamos el componente que lleva la UI del chat.*
* ***ng generate interface views/module1/models/chat****:* creamos la interfaz para el intercambio de datos del chat.
* ***ng generate service views/module1/services/chat****:* creamos el servicio que permitirá la comunicación entre las dos instancias de nuestro componente chat.

Deberíamos tener la siguiente estructura:



* En **module1.module.ts**:

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

import { CommonModule } from '@angular/common';

import { Module1RoutingModule } from './module1-routing.module';

import { FormsModule } from '@angular/forms';

import { SharedModule } from '../../shared/shared.module';

import { Module1ContainerComponent } from './container/module1-container/module1-container.component';

import { ChatComponent } from './components/chat/chat.component';

@NgModule({

declarations: [

Module1ContainerComponent,

ChatComponent

],

imports: [

CommonModule,

Module1RoutingModule,

FormsModule,

SharedModule

]

})

export class Module1Module { }

Importamos el módulo **FormsModule** para poder usar la directiva **[(ngModel)]** (la explicaré luego). También importamos el módulo **sharedModule** para hacer uso del componente compartido (Footer) creado previamente.

* En **module1-routing.module.ts**:

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

import { Routes, RouterModule } from '@angular/router';

import { Module1ContainerComponent } from './container/module1-container/module1-container.component';

const routes: Routes = [

{

path:'',

component: Module1ContainerComponent

}

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule]

})

export class Module1RoutingModule { }

Indicamos al enrutador cargar el container cuando se llegue a la ruta que llama al módulo (‘/module1’).

* En **chat.component.html**:

<h5 class="font-weigth-light text-center">{{chat | titlecase}}</h5>

<div class="chatBox mb-1">

<div class="row p-1">

<div class="col-10 mb-1 text-light rounded d-flex flex-column" \*ngFor="let message of messages" [ngClass]="{

'bg-secondary': message.chat == chat,

'ml-auto': message.chat == chat,

'bg-success': message.chat != chat}">

<p [class.ml-auto]="message.chat != chat">{{message.message}}</p>

<small [class.ml-auto]="message.chat == chat">

{{message.time | date: 'short':'UTC'}}

</small>

</div>

</div>

</div>

<div class="d-flex">

<input [(ngModel)]="message" class="form-control" type="text" placeholder="Mensaje...">

<button class="btn btn-success ml-1" (click)="sendMessage()" [disabled]="!message">Enviar</button>

</div>

Mediante el uso de llaves {{ }} podemos hacer referencia a una propiedad dentro del .ts que esté llamando a la plantilla HTML, por ejemplo, **{{chat | titlecase}}**, dónde **chat** es la propiedad, y **titlecase** es un **pipe**. Los **pipe** nos permiten formatear la data para mostrarla como mejor convenga.

Podemos observar el uso de las siguientes directivas:

* **\*ngFor**: presenta una lista de elementos, ahorrándonos tener que reescribir el mismo HTML una y otra vez.
* **[ngClass]**: añade o elimina una clase del elemento dependiendo de una condición.
* **[(ngModel)]**: nos permite modificar el valor de una propiedad de forma bidireccional; si el valor en la vista cambia, también lo hará en el modelo de datos.
* En **chat.component.ts**:

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

import { ChatService } from '../../services/chat.service';

import { Chat } from '../../models/chat';

import { Subscription } from 'rxjs';

@Component({

selector: 'app-chat',

templateUrl: './chat.component.html',

styleUrls: ['./chat.component.scss']

})

export class ChatComponent implements OnDestroy {

constructor(private chatService: ChatService) {

this.chatSubscription = this.chatService.getMessage()

.subscribe(message => {

this.messages.push(message);

});

}

@Input() chat: string;

messages: Array<Chat> = [];

message: string = "";

chatSubscription: Subscription;

sendMessage(): void {

this.chatService.sendMessage(

{

chat: this.chat,

message: this.message,

time: new Date()

}

);

this.message = "";

}

ngOnDestroy() {

if (this.chatSubscription) {

this.chatSubscription.unsubscribe();

}

}

}

En el constructor instanciamos nuestro servicio y nos suscribimos a una propiedad de tipo **subject** que éste posee (lo explico en la creación del servicio). Cada vez que nos llega un nuevo mensaje le hacemos **push** al array **messages**, y éste actualiza la vista automáticamente gracias al uso de la directiva **\*ngFor**.

El decorador **@Input** nos permite declarar un dato de entrada para el componente desde el selector (etiqueta HTML). En este caso, **chat** nos servirá de ayuda para diferenciar los mensajes recibidos de los enviados.

Con el método **sendMessage()** le enviamos un nuevo mensaje al subject del servicio.

Con el método **ngOnDestroy()** lo usamos para cancelar la suscripción del observable a la hora de salir del componente y evitar problemas de memoria.

* En **chat.component.scss**:

.chatBox{

height: 300px;

overflow-y: auto;

overflow-x: hidden;

border: 1px solid lightgray;

border-radius: 5px;

}

* En **module1-container.component.html**:

<div class="card">

<div class="card-header text-center h5">

<b>Módulo 1</b>: Chat

<br>

<small>(Comunicación entre componentes padres)</small>

</div>

<div class="card-body">

<div class="container-fluid">

<div class="row justify-content-center">

<app-chat class="col-12 col-md-5 mr-1 mb-3 md-md-0" [chat]="'chat1'"></app-chat>

<app-chat class="col-12 col-md-5" [chat]="'chat2'"></app-chat>

</div>

</div>

</div>

</div>

La etiqueta **<app-chat>** es el selector del componente **chat** que creamos previamente. Lo instanciamos dos veces y los diferenciamos mediante la propiedad [chat] (el @Input que mencionaba).

* En **chat.ts**:

export interface Chat {

chat: string,

message: string,

time: Date

}

* En **chat.service.ts**:

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

import { Observable, Subject } from 'rxjs';

import { Chat } from '../models/chat';

@Injectable({

providedIn: 'root'

})

export class ChatService {

private messageEmitter = new Subject<any>();

sendMessage(chat: Chat) {

this.messageEmitter.next(chat);

}

getMessage(): Observable<any> {

return this.messageEmitter.asObservable();

}

}

La propiedad **messageEmitter** es un **subject**. Los subject actúan como observable y observador. Los observables emiten mensajes a sus suscriptores mediante un evento, y los observadores escuchan esperando mensajes. La característica importante del subject es que todos sus observadores recibirán la misma data, a diferencia de los observables, que crean una instancia nueva para cada observador.

En el método **sendMessage()** usamos el método **next** del **subject**, que es el método que usan los observables para enviar mensajes a sus observadores.

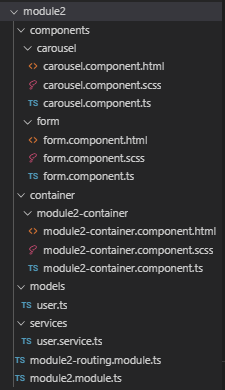
## 4.5. Desarrollo de module2

En el **module2** consumiremos servicios REST; nos traeremos una lista de usuarios mediante un formulario y mostraremos los resultados en un carousel de bootstrap. Se empleará la comunicación entre el componente padre y los componentes hijos mediante los decoradores **@Input** y **@Output**.

Ejecutamos los siguientes comandos:

* ***ng generate component views/module2/container/module2-container****:* creamos el componente contenedor de la vista.
* ***ng generate component views/module1/components/carousel****:* creamos el componente que contiene la UI del carousel.
* ***ng generate component views/module1/components/form***: creamos el componente que contiene el formulario de búsqueda.
* ***ng generate interface views/module1/models/user****:* creamos la interfaz para la estructura de los usuarios.
* ***ng generate service views/module1/services/user****:* creamos el servicio que consumirá los servicios REST.

Deberíamos tener la siguiente estructura:



* En **module2.module.ts**:

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

import { CommonModule } from '@angular/common';

import { Module2RoutingModule } from './module2-routing.module';

import { CarouselModule } from 'ngx-bootstrap/carousel';

import { SharedModule } from '../../shared/shared.module';

import { Module2ContainerComponent } from './container/module2-container/module2-container.component';

import { CarouselComponent } from './components/carousel/carousel.component';

@NgModule({

declarations: [

Module2ContainerComponent,

CarouselComponent

],

imports: [

CommonModule,

Module2RoutingModule,

CarouselModule,

SharedModule

]

})

export class Module2Module { }

* En **module2-routing.module.ts**:

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

import { Routes, RouterModule } from '@angular/router';

import { Module2ContainerComponent } from './container/module2-container/module2-container.component';

const routes: Routes = [

{

path:'',

component: Module2ContainerComponent

}

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule]

})

export class Module2RoutingModule { }

* En **form.component.ts**:

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

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({

selector: 'app-form',

templateUrl: './form.component.html',

styleUrls: ['./form.component.scss']

})

export class FormComponent implements OnInit {

constructor(private formBuilder: FormBuilder) { }

@Output() emitter = new EventEmitter();

pageForm: FormGroup;

ngOnInit() {

this.pageForm = this.formBuilder.group({

pageNumber: ["", Validators.required],

});

}

submit() {

this.emitter.emit(this.pageForm.get("pageNumber").value);

this.pageForm.reset();

}

}

El método **ngOnInit()** se ejecuta cuando toda la data asociada al componente se carga.

Los formularios en Angular constan de dos partes, el objeto que vive dentro del componente que almacena y valida el formulario, y la parte visual que se encuentra en la plantilla HTML. Para hacer un formulario declaramos una propiedad de tipo **FormGroup**, en nuestro caso será **pageForm**, luego lo construimos en el **ngOnInit()** mediante la propiedad **formBuilder** que es de tipo **FormBuilder**, la cual crea los controladores (los veremos en el .html del componente), les da un valor inicial y les asocia una validación.

Con el decorador **@Output()** emitimos data mediante un evento (**EventEmitter**) al componente padre, como podemos ver en el método **itemClicked()**, donde se envía data mediante el método **emit** del **EventEmitter**.

* En **form.component.html**

<div class="border rounded p-3">

<form [formGroup]="pageForm" (ngSubmit)="submit()">

<label>Número de página:</label>

<div class="d-flex">

<input class="form-control" type="number" formControlName="pageNumber" min="1">

<button class="btn btn-success ml-1" type="submit" [disabled]="!pageForm.valid">Enviar</button>

</div>

</form>

</div>

Mediante **[formGroup]** le asignamos a la etiqueta **form** la propiedad **pageForm** que definimos previamente en el componente. El evento **(ngSubmit)** le asocia un método al botón encargado de enviar el formulario. Definimos un controlador llamado **pageNumber** mediante el atributo **formControlName**.

El botón de tipo **submit** ejecutará el método que le indiquemos en **(ngSubmit)**. También podemos observar que el botón tiene un atributo llamado **[disabled]**, donde le decimos al botón que se deshabilite si el formulario es inválido. El formulario será válido si todos los controladores cumplen con la validación que le pasamos en el **formBuilder** del .ts. Para el controlador **pageNumber** de este ejemplo usamos **Validators.required**, que nos indica que nuestro controlador, el **input,** no puede estar vacío (ser nulo).

* En **carousel.component.ts**:

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

@Component({

selector: 'app-carousel',

templateUrl: './carousel.component.html',

styleUrls: ['./carousel.component.scss']

})

export class CarouselComponent {

@Input() items = [];

@Output() emitter = new EventEmitter();

itemClicked(item:JSON){

this.emitter.emit(item);

}

}

* En **carousel.component.html**:

<carousel [itemsPerSlide]="3" [singleSlideOffset]="false" [noWrap]="false" [interval]="false">

<slide class="mx-3" \*ngFor="let item of items" style="cursor: pointer" (click)="itemClicked(item)">

<img [src]="item.avatar" alt="image slide" class="w-100 d-block">

<div class="carousel-caption">

<p>Slide</p>

</div>

</slide>

</carousel>

* En **module2-container.component.html**:

<div class="card">

<div class="card-header text-center h5">

<b>Módulo 2</b>: Lista de Usuarios

<br>

<small>(Consumo de servicio REST y comunicación entre componente padre y componente hijo)</small>

</div>

<div class="card-body">

<app-form (emitter)="searchUsers($event)"></app-form>

<div \*ngIf="users" class="border rounded mx-auto mt-3 p-3">

<div class="row justify-content-center">

<div class="col-12 h5 mb-2 text-center">

<b>Email: {{user.email}}</b>

</div>

<div class="col-12 mb-2 text-center my-5">

<img class="mw-100" alt="Avatar de usuario" src="{{user.avatar}}">

</div>

<div class="col-12 col-md-8 d-flex flex-column text-center">

<small>Nombre: <b>{{user.first\_name}}</b></small>

<small>Apellido: <b>{{user.last\_name}}</b></small>

</div>

</div>

<app-carousel [items]="users" (emitter)="user = $event"></app-carousel>

</div>

</div>

</div>

<app-footer></app-footer>

La directiva **\*ngIf** renderiza o no un elemento HTML dependiendo de una condición. Si la condición deja de cumplirse, el elemento es destruido.

Los selectores **<app-form>** y **<app-carousel>** hacen referencia a los componentes hijos y los instancian. Podemos observar que en ambas etiquetas hacemos uso del evento **(emitter)** que creamos en cada .ts respectivamente; con **$event** manejamos la data que se emita. Dentro del selector **<app-form>** le pasamos al método **searchUsers()** del componente padre el número de página proveniente del selector mediante **$event**.Dentro del selector **<app-carousel>** le asignamos **$event** a la propiedad **user** del componente padre, como también le pasamos la propiedad **users** del componente padre a la propiedad de entrada **[ítems]** (**@Input** que explicamos previamente) del selector.

* En **module2-container.component.ts:**

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

import { UserService } from '../../services/user.service';

import { User } from '../../models/user';

@Component({

selector: 'app-module2-container',

templateUrl: './module2-container.component.html',

styleUrls: ['./module2-container.component.scss']

})

export class Module2ContainerComponent {

constructor(private service: UserService) { }

users: Array<User>;

user: User;

searchUsers(page: number): void {

this.users = null;

this.service.getBooks(page).subscribe((response: any) => {

if (response.data.length != 0) {

this.users = response.data;

this.user = response.data[0];

} else {

alert("Número de página inválido. El servicio sólo funciona con 1 y 2.");

}

}, (error: any) => {

console.log(error);

});

}

}

Por medio de **searchUsers()** le pasaremos el número de página al método **getBooks()** que se encuentra en el servicio y éste nos retornará un observable con la respuesta del servidor.

* En **user.service.ts**:

import { HttpClient, HttpHeaders } from '@angular/common/http';

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

import { Observable } from 'rxjs';

import { environment } from 'src/environments/environment';

const httpOptions = {

headers: new HttpHeaders({

'Content-Type': 'application/json'

})

};

const url = environment.USER\_URL;

@Injectable({

providedIn: 'root'

})

export class UserService {

constructor(private http: HttpClient) { }

getBooks(page: number): Observable<any> {

return this.http.get(`${url}?page=${page}`, httpOptions);

}

}

Mediante **httpOptions** y la interfaz **HttpHeaders** definimos la cabecera del request.

Instanciamos la clase HttpClient en el constructor. Ésta posee los métodos tradicionales de consumo de servicios (get, post, put, delete…). El método **get(URL, cabecera)**, lo usaremos para hacer peticiones; retorna un observable, por lo que nos debemos suscribir desde el componente.

* En **user.ts**:

export interface User {

id: number,

email: string,

first\_name: string,

last\_name: string,

avatar: string

}

## 4.6. Variables de ambiente

Angular por defecto nos genera una carpeta **enviroments** con dos archivos para configurar las variables de ambiente: **environment.ts** y **environment.prod.ts**, siendo éstos seleccionables a la hora de compilar el proyecto.



* En **environment.ts**:

export const environment = {

production: false,

USER\_URL:'https://reqres.in/api/users'

};

Definimos la URL que consumirá el servicio como una propiedad del JSON **environment**.

## 4.7. Deployment en tomcat

En la carpeta root del proyecto tenemos un archivo *index.html*, el cual tiene una etiqueta **<base>**:

<base href="/">

Que debemos cambiar a la ruta que tendrá el proyecto en el servidor:

<base href="/my-app/">

Luego, compilamos el proyecto por medio del comando ***ng build****,*generando *dist/my-app* en la carpeta root. Podemos cambiar las variables de ambiente por medio de la opción ***--configuration****,* por ejemplo, ***ng build --configuration=prod***.

Para hacer deploy en Tomcat debemos añadir un archivo .xml en la carpeta root para manejar el acceso a una URL que no esté definida dentro de nuestro enrutador:

<?xml version="1.0"?>

<web-app>

<error-page>

<error-code>404</error-code>

<location>/index.html</location>

</error-page>

</web-app>