

Angular Cheatsheet

Joel Albornoz

Bootstrap:

Bootstrap es una biblioteca multiplataforma de estilos o un conjunto de herramientas orientados al desarrollo de estilos de una pagina web. Consiste de una hoja de estilos css y diversos métodos Javascript que determinan la apariencia visual y el comportamiento de componentes lógicos respectivamente. Dichos elementos y estilos se enlazan a los elementos de nuestro DOM a través de clases que asignamos como atributos en las etiquetas de dichos elementos.

¿Cómo se implementa?

Para utilizar Bootstrap en nuestro proyecto es posible instalarlo con nuestro gestor de paquetes desde <https://www.npmjs.com/package/bootstrap> , aunque la practica mas común es implementarlo enlazando sus CDN en el head de nuestro archivo index.html

```
<head>
```

```
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
  integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
```

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha384-
q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script><script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
  integrity="sha384-
U02eT0CpHqdSJJQ6hJty5KVphtPhzWj9W01c1HTMGa3JDZwrnQq4sF86dIHNDz0W1"
crossorigin="anonymous"></script><script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-
JjSmVgyd0p3pXB1rRibZUAYoIIy60rQ6VrjIEaFf/nJGzIxFDs4x0xIM+B07jRM"
crossorigin="anonymous"></script>
```

```
</head>
```

¿Cómo se controla el layout de elementos con Bootstrap?

- Containers:

Los contenedores son elementos más básicos para la disposición de elementos en Bootstrap, es posible anidarlos y dentro de ellos los elementos se ordenan en filas y columnas. Cada contenedor tiene un total de 12 columnas y los elementos dentro de el pueden ocupar una cantidad arbitraria de ellas según sea necesario.

```
<!-- Declaración de un contenedor -->
```

```
<div class='container'> </div>
```

-Media queries:

Las media queries de Bootstrap tienen como objetivo facilitar el desarrollo responsivo en nuestras páginas, por defecto trae definidas xs, sm, md, lg y xl que se refieren a los tamaños de pantallas muy pequeñas, pequeñas, medianas, grandes, y muy grandes respectivamente

```
@include media-breakpoint-up(sm) { estilos aplicables a dispositivos cuya pantalla sea pequeña}
```

- Margin and padding:

Bootstrap Incluye utilidades para manejar los márgenes y el padding de los elementos del DOM, basta con especificar “mr-3” como clase a un elemento para aplicar un margen de 1 rem al mismo, estas propiedades también tienen sus propias variantes responsivas, por ejemplo 'pr-md-3' aplicaría un padding de 1 rem solo a los elementos cuyas pantallas entren en la categoría mediana.

-¿Como funciona el sistema de grillas de Bootstrap?

Bootstrap divide sus elementos contenedores en filas cuyas columnas ocupan siempre un total de 12 celdas, la disposición de los elementos en estas filas y la cantidad de columnas que cada uno ocupa son altamente personalizables, pudiendo estos mismos elementos, ser a su vez contenedores de otros.

El siguiente código crea un contenedor de una fila, cuyos hijos ocupan una cantidad de columnas iguales.

```
<div class="container">
  <div class="row">
    <div class="col-sm">One of three columns</div>
    <div class="col-sm">One of three columns</div>
    <div class="col-sm">One of three columns</div>
  </div>
</div>
```

Componentes:

¿Qué es un componente?

Un componente es el elemento más básico del que se construyen las aplicaciones web en Angular, esta compuesto por una clase, con sus atributos y métodos ligado a una plantilla o template html que se encarga de definir como se muestra el componente de manera visual (junto con una hoja de estilos css). Una aplicación web puede estar formada por uno o más componentes y la relación entre ellos se da con las distintas directivas y elementos (Módulos y Servicios), que Angular pone a nuestra disposición.

¿Como diseño una clase como componente?

El decorador de Angular `@Component` identifica la clase inmediatamente debajo suyo como un componente. Como argumentos, recibe un objeto con la metadata que define nuestro componente; se deben especificar el selector (la etiqueta con la que se accede a nuestro componente), y un template (que puede ser una ruta al archivo, o escrito en nuestro componente como una string), además se pueden especificar providers (que inyectan dependencias de servicios) entre otras directivas.

Definición típica de un componente

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:   [ HeroService ]
})
export class HeroListComponent implements OnInit {
  /* . . . */
}
```

¿Cómo es la sintaxis del template de un componente?

La sintaxis del template de un componente es similar a cualquier código HTML excepto por el hecho de que también se incluyen en el código directivas propias de Angular, referencias a variables y métodos del componente (con enlace de datos) y pipes. Todos estos elementos, alteran el comportamiento del HTML basados en la lógica de nuestra aplicación.

Servicios

¿Qué es un servicio?

En Angular, un servicio es un elemento que sirve para compartir funcionalidades entre componentes. El patrón de diseño que se utiliza en los servicios es Singleton lo que quiere decir que solo hay una instancia de nuestro servicio que se consume en distintas partes de nuestra aplicación, esto permite mantener un buen nivel de mantenibilidad y orden además de aislar estas funcionalidades del resto de nuestra aplicación.

¿Cómo se crea un servicio?

Usando el siguiente comando en Angular CLI

```
ng generate service hero
```

Esto genera el esqueleto de un servicio en el directorio especificado.

¿Qué hay que saber sobre el decorador “@Injectable()”?

Este decorador señala nuestro servicio como parte del *Sistema de Inyección de dependencias de Angular*. Lo que nos permite utilizarlo en otros componentes y módulos. Este decorador acepta un objeto con la metadata de nuestro servicio de igual manera que el decorador “@Component()”.

¿Cómo inyecto un servicio a un componente?

El inyector de Angular se encarga de inyectar las dependencias especificadas por el desarrollador, para ello es necesario importar y definir nuestro servicio como proveedor del componente a inyectar.

```
@Component({  
  ...  
  providers: [MyService]})
```

Y luego instanciamos nuestro servicio en el constructor del componente

```
constructor(private myService: MyService){  
  console.log('myService', myService);  
}
```

Conexión con una API

¿Por qué conectar con una API?

Conectar con una API nos permite consumir información en general que otros desarrolladores (o nosotros mismos) ponen a nuestra disposición con la ayuda de un servidor y (por lo general) una base de datos. La posibilidad de utilizar APIs en nuestra aplicación no solo nos abre un gran abanico de posibilidades para nuestro desarrollo sino que nos permite acceder a una cantidad innumerable de recursos muy útiles y de muy variada aplicación.

¿Cómo se conecta con una API en Angular?

En Angular una conexión a una API por lo general se hace a través de un servicio que provee de dicha conexión a los componentes en los que necesitemos consumirla. El módulo HTTPClient de Angular nos permite hacer solicitudes HTTP a servidores y nos devuelve un observable que recibirá

```
getHeroes (): Observable<Hero[]> {  
  return this.http.get<Hero[]>(this.heroesUrl)  
}
```

La información que solicitamos. A diferencia de Async/Await típicamente usado en Javascript o el formato promise.then() también usado comúnmente, la clase Observable en Angular nos permite subscribirnos a ella con el método Observable.subscribe() y pasar como argumento una callback que se ejecutará una vez nuestra información llegue desde el servidor.

```
observable.subscribe(x => console.log(x));
```

La clase Observable es parte de las librerías de RxJS que vienen incluidas con Angular, esta librería tiene muchas más rutinas, la mayoría orientadas al control de flujo de datos, por lo general utilizadas en comunicaciones cliente-servidor.



Comunicación entre componentes

¿Qué utilidad tiene la comunicación entre componentes?

La comunicación entre componentes en Angular tiene una infinita cantidad de aplicaciones entre las más notables se pueden encontrar:

- Compartir Variables.
- Compartir Funciones.
- Compartir Instancias de un objeto (Incluidos los servicios).
- Crear subrutinas cuya implementación esté dividida en distintos componentes.

¿Qué opciones existen para comunicar dos componentes?

*Pasar una referencia de un componente a otro:

Desde el componente padre se crea una referencia de uno de nuestros componentes, y se pasa como atributo dentro de la etiqueta en otro.

```
<div class="container">  
  app component  
  <br>  
  <br>  
  <app-side-bar-toggle [sideBar]="sideBar"></app-side-bar-toggle>  
  <app-side-bar #sideBar></app-side-bar>  
</div>
```

Se recibe la referencia con el decorador @Input() en el componente hijo y se guarda en una variable.

```
@Input() sideBar: SideBarComponent;
```

*Realizar la comunicación a través de un componente Padre:

En este caso tenemos un estado que controlamos en el componente padre

```
sideBarIsOpened = false;
```

```
toggleSideBar(shouldOpen: boolean) {  
  this.sideBarIsOpened = !this.sideBarIsOpened;  
}
```

Este estado lo pasamos al componente hijo que cambia su comportamiento según el mismo, una vez más lo recibimos con el decorador @Input()

```
<app-side-bar [isOpen]="sideBarIsOpened"></app-side-bar>
```

```
@Input() isOpen = false;
```

***Utilizar un Servicio:**

En este caso tanto el estado como los métodos para cambiarlo se encuentran en un servicio, que se comparte en los distintos componentes, cabe remarcar que es importante importar el servicio en el app.module y especificarlo como proveedor, para que el inyector de Angular lo reconozca.

Input/ outputs

Los decoradores @Input() y @Output sirven para compartir información entre componentes.

@Input():

@Input() nos permite enviarle información a un componente hijo a través de un atributo en su etiqueta, y este, a su vez lo recibe en una variable.

En el HTML del componente padre:

```
<bank-account bankName="RBC" account-id="4747"></bank-account>
```

Dentro de la clase del componente hijo:

```
@Input() bankName: string;
```

```
@Input('account-id') id: string;
```

@Output():

Al usar @Output() el componente hijo utiliza un EventEmitter para producir eventos a los que reacciona el componente padre.

```
@Output() voted = new EventEmitter<boolean>();
```

En el componente padre:

HTML:

```
<app-voter *ngFor="let voter of voters"
  [name]="voter"
  (voted)="onVoted($event)">
```

Método de clase:

```
onVoted(agreed: boolean) {
  agreed ? this.agreed++ : this.disagreed++;
}
```

Unsubscribe:

¿Para que usar unsubscribe()?

Es importante que al usar Observables utilicemos unsubscribe() para evitar la memory leaks. Los observables de Angular tienen la capacidad para devolver cero, uno o infinitos datos dependiendo de la situación cuando nos subscribimos a ellos. Por eso es importante especificar cuando queremos cerrar nuestra suscripción para hacer un uso óptimo de la memoria en nuestra aplicación.

¿Cuándo es necesario?

Por lo general, cuando ya no necesitamos utilizar el observable; cuando el componente que lo está consumiendo es destruido es posible utilizar ngOnDestroy() para hacer una limpieza previa, cerrando suscripciones a observables justo antes de que esto suceda.

¿Cómo?

-Es posible guardar la suscripción al momento de crearla y luego utilizar unsubscribe() en ella cuando el componente se destruye.

-Otra opción es establecer un condicional con takeWhile() para cerrar la suscripción automáticamente.

-Por último, se puede utilizar takeUntil() para establecer un límite como condicional, haciendo uso de un segundo observable.

AsyncPipe:

Para concluir, es necesario recalcar que al utilizar nuestros Observables a través de un AsyncPipe Angular se encarga de realizar las suscripciones y el manejo de memoria pertinentes, por lo que no es necesario implementar unsubscribe() en estos casos.

Un poco más sobre AsyncPipe:

```
{{ obj_expression | async }}
```

Al igual que las demás pipes AsyncPipe sirve para dar un formato a nuestros datos, lo que la hace un caso especial, sin embargo, es que esta pipe es utilizada sobre Observables y Promesas y nos devuelve el último valor retornado por estos, como se mencionó antes, AsyncPipe cierra automáticamente toda subscripción al ser destruido el componente en el que se encuentra.

Ciclos de vida en Angular:

Angular nos ofrece Hooks de Ciclo de vida que nos proveen acceso a los distintos momentos en la vida de nuestros componentes y la habilidad de actuar cuando estos suceden.

¿Qué ciclos de vida tiene un componente?

| Hook | Proósito y momento en que ocurren |
|-----------------------------|--|
| ngOnChanges() | Ocurre cuando la información que almacena un componente o que recibe como parámetro cambia, recibe un objeto SimpleChanges como argumento y ocurre antes de OnInit() |
| ngOnInit() | Inicializa el componente y fija las propiedades recibidas por Input |
| ngDoCheck() | Detecta y actua sobre cambios que Angular no actua por si mismo. |
| <u>ngAfterContentInit()</u> | Responde despues de que Angular Proyecta el contenido externo en la vista del componente. |
| ngAfterContentChecked() | Responde luego de que angular confirma la vista del componente. |
| <u>ngAfterViewInit()</u> | Responde luego de que Angular inicializa la vista del componente y sus hijos. |
| ngAfterViewChecked() | Responde luego de que Angular confirma la vista del componente y sus hijos. |
| ngOnDestroy() | Se encarga de tareas de limpieza justo antes de que Angular destruya el componente. Cierra manejadores de eventos y subscripciones. |

@ViewChild/ @ViewChildren

@ViewChild y @ViewChildren son utilizados para acceder a los elementos hijos del host que coincidan con el la llamada realizada. Los datos son devueltos en una QueryList.

@ViewChild devuelve el primer elemento que coincida con la directiva o variable que se le pasa como argumento mientras que @ViewChildren devuelve la QueryList con todos los elementos coincidentes.

Por ejemplo, creamos un componente:

```
1  @Component({
2    selector: 'alert',
3    template: `
4      <h1 (click)="alert()">{{type}}</h1>
5    `,
6  })
7  export class AlertComponent {
8    @Input() type: string = "success";
9
10   alert() {
11     console.log("alert");
12   }
13 }
```

Luego generamos varias instancias de nuestro componente y accedemos a ellas con @ViewChildren:

```
1  @Component({
2    selector: 'my-app',
3    template: `
4      <alert></alert>
5      <alert type="danger"></alert>
6      <alert type="info"></alert>
7    `,
8  })
9  export class App {
10    @ViewChildren(AlertComponent) alerts: QueryList<AlertComponent>
11
12    ngAfterViewInit() {
13      this.alerts.forEach(alertInstance => console.log(alertInstance));
14    }
15  }
```

Esto nos devuelve instancias de nuestro componente.

```
► AlertComponent {type: "success"}
► AlertComponent {type: "danger"}
► AlertComponent {type: "info"}
```

FormBuilder:

La clase FormBuilder de Angular nos provee de una azucar sintáctica para la generación de Formularios reactivos, cuenta con varios métodos además de traer la posibilidad de implementar validaciones ya creadas previamente.

Para utilizarlo basta con importarlo en nuestro modulo

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

import { AppComponent } from './app.component';
import { ControlMessagesComponent } from './control-messages.component';
import { ValidationService } from './validation.service';

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

Y luego servirlo en nuestro componente:

```
import { Component } from '@angular/core';
import { FormBuilder, Validators } from '@angular/forms';
import { ValidationService } from 'app/validation.service';

@Component({
  selector: 'demo-app',
  templateUrl: 'app/app.component.html'
})
export class AppComponent {
  userForm: any;

  constructor(private formBuilder: FormBuilder) {
    

this.userForm = this.formBuilder.group({
        name: ['', Validators.required],
        email: ['', [Validators.required, ValidationService.emailValidator]],
        profile: ['', [Validators.required, Validators.minLength(10)]]
      });


  }
}
```

Por último utilizamos nuestro formulario generado en el template:

```
<form [formGroup]="userForm" (submit)="saveUser()">
  <label for="name">Name</label>
  <input formControlName="name" id="name" />
  <control-messages [control]="userForm.get('name')"></control-messages>

  <label for="email">Email</label>
  <input formControlName="email" id="email" />
  <control-messages [control]="userForm.get('email')"></control-messages>

  <label for="profile">Profile Description</label>
  <textarea formControlName="profile" id="profile"></textarea>
  <control-messages [control]="userForm.get('profile')"></control-messages>

  <button type="submit" [disabled]="!userForm.valid">Submit</button>
</form>
```

Ruteo en Angular:

En Angular, el manejo de las rutas está delegado a un Router, en el que podemos definir que componentes se deben renderizar al entrar a una ruta en específico de nuestra aplicación.

Angular genera nuestro componente Router al momento de la creación de nuestro proyecto por nosotros si así se lo especificamos, luego esta en nosotros el configurarlo e implementarlo:

Configurar las rutas:

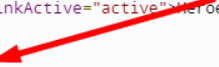
Nuestro componente Router utiliza un array de rutas (especificadas en forma de objetos), para dirigir la navegación en nuestra app, la declaración del array Routes, típicamente se ve así:

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id', component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];
```

Cada uno de estos objetos en el array especifica una ruta de nuestra aplicación y establece un componente que se debe renderizar cuando se está en ella, por último define una ruta ‘**’ que es la que se ejecutará en caso de que el usuario intente ingresar a una ruta inexistente.

Con nuestro componente listo, queda insertar su etiqueta en nuestro app-component.html para que pueda comenzar a hacer su trabajo tal que así:

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```



Las directivas “routerLink” en los anchor tags nos sirven para guiar la navegación dentro de nuestra página y las rutas especificadas allí se corresponden a las que definimos en el router component.

Parámetros en las Rutas:

También es posible insertar parámetros en nuestras rutas para afectar el comportamiento de los componentes que los reciben

```
{ path: 'hero/:id', component: HeroDetailComponent }
```

En este caso ‘:id’ es el parámetro que recibirá el componente a la hora de ser llamado, se puede establecer el valor a enviar en la llamada al componente, al momento de realizar una redirección hacia este:

```
<a [routerLink]="['/hero', hero.id]">
```

Luego, en nuestro componente receptor, utilizamos el parámetro recibido de la siguiente forma:

```
this.service.getHero(params.get('id'))
```

Guardas:

Existen muchas razones por las que querríamos proteger diversas rutas de nuestra página, esto es posible con la implementación de guardas en Angular en la configuración de nuestras rutas.

El router de Angular soporta múltiples interfaces de guarda:

- CanActivate para intermediar la navegación hacia una ruta.
- CanActivateChild para mediar la navegación hacia una ruta hija.
- CanDeactivate para mediar la navegación hacia fuera de la ruta actual.
- Resolve para realizar la recepción de datos de una ruta antes de su activación.
- CanLoad para intermediar la navegación hacia una página que se carga de manera asíncrona.

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { CanActivate } from '@angular/router';
import { LoginService } from '../login/login.service';

@Injectable()
export class CanActivateViaAuthGuard implements CanActivate {

  constructor(private authService: LoginService, private router: Router) { }

  canActivate() {
    // If the user is not logged in we'll send them back to the home page
    if (!this.authService.isLoggedIn()) {
      console.log('No estás logueado');
      this.router.navigate(['/']);
      return false;
    }

    return true;
  }
}
```

En este ejemplo implementamos CanActivate en nuestro componente guarda y definimos una condición para la guarda.














Luego importamos nuestra directiva guarda en nuestro componente router y lo aplicamos en las rutas que nos interesen.

```
{ path: 'films', component: FilmListComponent, canActivate: [CanActivateViaAuthGuard] },
```

Pipes:

Las pipes en Angular son directivas que sirven para dar un formato a la información que mostramos en nuestro HTML de acuerdo a nuestras necesidades.

Las Pipes definidas actualmente en la documentación de Angular son:

| | | |
|---|--|--|
|  AsyncPipe |  CurrencyPipe |  DatePipe |
|  DecimalPipe |  I18nPluralPipe |  I18nSelectPipe |
|  JsonPipe |  KeyValuePipe |  LowerCasePipe |
|  PercentPipe |  SlicePipe |  TitleCasePipe |
|  UpperCasePipe | | |

Pipes puras e impuras:

Una pipe pura solo es llamada cuando Angular detecta un cambio en los valores que esta mostrando, mientras que una pipe impura se llama por cada ciclo de detección sin importar si los valores o parámetros cambiaron.

