

Curso de Angular

Plataforma: Udemy | Instructor: Fernando Herrera | Fecha: Diciembre 2023

Nota del Autor:

El siguiente documento constituye una exhaustiva recopilación de los conocimientos y prácticas derivados del curso de Angular ofrecido en la plataforma Udemy. Su propósito fundamental radica en servir como un recurso detallado y de fácil acceso para futuras referencias en mi trayectoria profesional.

Además de presentar el código proporcionado por el instructor, este documento incluye explicaciones detalladas de ciertos conceptos que, aunque se abordaron en el curso de Angular, no fueron explorados en profundidad. Se ha procurado enriquecer la comprensión de estos temas mediante análisis más detallados. Esta extensión va más allá de la enseñanza estándar del curso, proporcionando una perspectiva más completa y facilitando la asimilación de conceptos clave. Así, este material no solo actúa como una recopilación de lo aprendido, sino también como un recurso complementario que busca ofrecer una comprensión más holística de los temas tratados en el curso.

Quisiera enfatizar que este material no tiene como finalidad generar lucro alguno. En lugar de ello, busca únicamente consolidar y mantener frescos los conocimientos adquiridos durante el curso. Es importante señalar que la mayor parte del código presente en este documento ha sido proporcionado por el instructor, **Fernando Herrera**. Solo en casos excepcionales se han incorporado modificaciones o funcionalidades adicionales como resultado de prácticas complementarias.

Esta recopilación se presenta como una herramienta personal, creada con el objetivo de fortalecer y consolidar los conceptos aprendidos en el curso de Angular. Agradezco profundamente al instructor por compartir su experiencia y conocimientos, los cuales han sido fundamentales para mi desarrollo en esta tecnología.

Espero que este documento no solo sirva como recordatorio para mí, sino también como una fuente de conocimiento para otros estudiantes interesados en profundizar en Angular. Cabe destacar que cualquier beneficio derivado de este material debe ser atribuido principalmente al esfuerzo y dedicación del instructor y la plataforma Udemy, a quienes agradezco por facilitar este valioso aprendizaje.

Mas información Aquí: <https://www.udemy.com/course/angular-fernando-herrera>

Diciembre 2023 - En proceso...

CONTENIDO

Instalación

Se requiere:

- Google Chrome
- Visual Studio Code
- Postman o extensión REST CLIENT de Code.
- Mongo Compass
- Docker
- Docker Desktop
- Git
- Tener una cuenta en github o bitbucket, etc.

Para **Mongo Compass** ir al sitio <https://www.mongodb.com/try/download/compass> descargar y luego instalar el rpm (o el ejecutable dependiendo de su sistema operativo)

Extensiones recomendadas para VSCode

- Angular Snippets
- Angular Language Service
- Angular Inline
- Auto Close Tag
- TypeScript importer
- Rest Client

Installing Angular CLI

NOTA: Ver comentarios al final de esta sección para instalar la versión LTS compatible con Angular Cli.

Como usuario normal, Instalar o actualiza a la última versión de Node

```
$ nvm install node
```

Si la versión instalada es > 18.0.0 debe instalar la 16.10.x

```
$ nvm install 16.10.0
$ nvm use 16.10.0
$ nvm alias default 16.10.0 (Esto hará que los editores como code usen la
versión or default)
```

Instalar angular-CLI

```
$ npm install -g @angular/cli
```

Probar Versión:

```
$ ng --version  
Angular CLI: 13.3.4  
Node: 16.10.0  
Package Manager: npm 7.24.0  
OS: linux x64
```

NOTA: Luego de retomar el curso debí instalar la última versión estable soportada por Angular con este comando:

```
nvm install --lts  
vm alias default 20.9.0  
npm uninstall -g @angular/cli  
npm install -g @angular/cli
```

Introducción a Angular

Angular es un framework, mantenido por google, viene con todo lo que se necesite para trabajar las aplicaciones. Las aplicaciones Angular son Modulares, es decir, todo el código se escribe en módulos independientes. (Antes de la V17 cuando se incuyen los standalone)

Existe 5 bloques fundamentales

- Componentes
- Rutas
- Directivas
- Servicios
- Módulos

Los componentes:

- Es un bloque de código con segmento de HTML más una clase de Typescript. El objetivo es que cada componente sean bloques simples de código.

Servicio

- Son Singleton que permiten trabajar la aplicación con información centralizada. Los componentes hacen uso de los servicios, Los servicios exponen funcionalidades a otros servicios o componentes.

Directivas:

- Existen 3 tipos de Directivas de Componentes, de estructuras y de atributos.

Directivas de Atributo

Alteran la apariencia o comportamiento de un elemento del DOM y son usados como atributos de los elementos. (ngModel, ngClass, ngStyle)

Directivas Estructurales

Alteran la estructura del DOM, agregando, eliminando y manipulando los elementos a los que están unidos. (*ngIf, *ngFor, ngSwitch, ngPlural)

Directivas de Componente

Las Directivas de Componente son directivas con un Template. Los componentes tienen decoradores "@Component", el componente es un decorador @Directive que es extendido con características propias de los templates.

Rutas:

Inidican que componentes renderizar dependiendo el URL ingresado por el usuario. Una Ruta típica de Angular tiene dos propiedades: path : una cadena que coincide con la URL en la barra de direcciones del navegador. component : el componente que el enrutador debe crear al navegar a esta ruta.

Módulos: Permiten agrupar componentes, incluso otros módulos. Se pueden descargar módulos existentes para usarlo dentro de la APP. Por ejemplo un módulo de Calendario.

Instalar el CLI

Usaremos el Angular CLI (Command Line) de la siguiente manera

```
$ npm install -g @angular/cli
```

Crear Nuevo proyecto

para crear un nuevo proyecto con el CLI usamos el siguiente comando:

```
$ ng new 01-bases --standalone false
```

--standalone false Las últimas versiones de Angular (2023) ya no usan por default NgModule, pero durante el curso se usa la versión con NgModule, por lo tanto, este parámetro en false permite crear los proyectos tal como se hacia en las versiones anteriores, es decir de forma tradicional.

Esto genera el proyecto de pruebas y puede ejecutarse con el siguiente comando

```
$ ng serve -o
```

NOTA: si el comando anterior genera algún error, realizar nuevamente el `npm install`

Archivos creados por el ng cli.

El proyecto cuenta con varios directorios y archivos

Archivos

.editconfig archivo de configuración agregado por el plugin que permite agregar configuraciones para el editor.

.gitignore

angular.json Es un archivo importante, muy pocas configuraciones se hacen aquí, en este archivo le decimos a angular ciertas configuraciones para nuestras app.

karma.conf.js Karma - a test runner

tsconfig un grupo de archivos para configuraciones de typescript.

Directories

.angular archivo oculto que pocas veces se modifican, Ayuda a angular para detectar cambios, maneja el cache del proyecto, etc.

.vscode Archivo del editor

dist Directorio de distribución

node_modules Módulos de Node, dependencias.

src Código fuente

src/app Código fuente de nuestro proyecto

src/assets Los Assets

src/environments Archivos de configuración

src/main.ts punto de entrada de nuestra App.

Primera aproximación de un componente

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = 'Contador App';
}
```

selector: indica el elemento html que contendrá el componente una vez renderizado, en este caso es app-root ya que es la aplicación principal.

templateUrl permite incluir un template existente escrito en HTML.

Podemos pasar variables al template al momento de crear la clase del componente, como en el siguiente caso:

```
export class AppComponent {
  title = 'Contador App';
}
```

Title puede luego usarse en el template de la siguiente forma:

```
<h1>Hola Mundo</h1>
<span>Titulo desde componente: {{title}}</span>
```

Esto generará:

```
Hola Mundo  
Titulo desde componente: Contador App
```

Métodos en los componentes

Podemos agregar métodos en nuestra clase y usarlos desde el template de la siguiente forma:

```
export class AppComponent {  
  title: string = 'Contador App';  
  numero: number = 10;  
  
  acumular(valor: number) {  
    this.numero += valor;  
  }  
}
```

Y consumirlo de la siguiente forma en el template:

```
<button (click)="acumular(+1)">+1</button>  
<button (click)="acumular(-1)">-1</button>
```

Creando nuevos componentes.

Cuando la aplicación comienza a crecer, es necesario crear una estructura de directorios para organizar nuestro código. Cada componente con todos sus elementos se crea en sus propios directorios

Supongamos que vamos a crear un componente llamado `Heroe` pero este componente tiene una vista para renderizar información de un Heroe específico y tambien otro componente para mostrar información de los heroes en forma de lista, en este caso podriamos crear la siguiente estructura

```
└── heroes
    ├── heroe
    │   ├── heroe.component.html
    │   └── heroe.component.ts
    └── listado
```

El directorio `Heroe` contiene el componente que muestra información de un heroe específico, mientras que el directorio `listado`, va a contener el componente para mostrar un listado de heroes.

De esta forma si necesitamos crear otro componente relacionado con `heroes`, creariamos un nuevo directorio dentro de `heroes`.

Veamos el componente `Heroe.component.ts`.

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

@Component({
  selector: 'app-hero',
  templateUrl: 'hero.component.html',
})
export class HeroeComponent {
  nombre: string = 'IronMan';
  edad: number = 35;

  get nombreCapitalizado() {
    return this.nombre.toUpperCase();
  }

  obtenerNombre(): string {
    return `${this.nombre} - ${this.edad}`;
  }

  cambiarNombre(): void {
    this.nombre = 'Spiderman';
  }

  cambiarEdad(): void {
    this.edad = 43;
  }
}
```

Y su template:

```
<h1>Heroe Component</h1>
<dl>
  <td>Nombre:</td>
  <dd>{{nombre}}</dd>

  <td>Edad:</td>
  <dd>{{edad}}</dd>

  <td>Función:</td>
  <dd>{{obtenerNombre()}}</dd>

  <td>Capitalizado:</td>
  <dd>{{nombreCapitalizado}}</dd>
</dl>

<button (click)="cambiarNombre()">Cambiar Nombre</button>
<button (click)="cambiarEdad()">Cambiar Edad</button>
```

Notar como se usan las propiedades **nombre** y **edad** así como una propiedad especial **get** llamada **nombreCapitalizado** y un método **obtenerNombre**. Observe como se consumen esos elementos en el template.

Los métodos **cambiarNombre()** y **cambiarEdad()** cambian las propiedades nombre y edad del componente respectivamente. Al cambiar una propiedad, el componente volverá a renderizar todas aquellas partes (elementos del DOM) que dependen de la propiedad modificada.

Crear Componente de forma automática.

Para crear un componente con el ngCLI, usaremos el generador de componentes:

```
$ ng generate component heroes/listado
```

Podemos usar la forma simplificada, por ejemplo

```
$ ng g c heroes/listado
```

g: Generate c: Component

El último valor se refiere al path y nombre del componente. Si soloamente se especifica un valor, el componente será creado en el directorio app/src.

Las salida del comando anterior debe ser:

```
$ ng generate component heroes/listado
CREATE src/app/heroes/listado/listado.component.css (0 bytes)
CREATE src/app/heroes/listado/listado.component.html (23 bytes)
CREATE src/app/heroes/listado/listado.component.spec.ts (640 bytes)
CREATE src/app/heroes/listado/listado.component.ts (283 bytes)
UPDATE src/app/app.module.ts (586 bytes)
```

El componente creado se verá de la siguiente forma:

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

@Component({
  selector: 'app-listado',
  templateUrl: './listado.component.html',
})
export class ListadoComponent implements OnInit {
  constructor() {}

  ngOnInit(): void {}
}
```

ngOnInit es parte del ciclo de vida de los componentes de angular, es un método que se dispara cuando el componente se ha inicializado, veremos estos ciclos de vida más adelante.

Por el momento podemos borrar la referencia al OnInit, el archivo CSS y el archivo de pruebas.

Directiva *ngFor

Podemos recorrer arreglos o elementos iterables y a la vez imprimir su contenido en el html, veamos este ejemplo:

Actualicemos nuestro componente:

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

@Component({
  selector: 'app-listado',
  templateUrl: './listado.component.html',
})
export class ListadoComponent {
  heroeBorrado: string = '';

  heroes: string[] = ['Spiderman', 'Superman', 'Hulk', 'Ironman'];

  borrarElemento() {
    this.heroeBorrado = this.heroes.shift() || '';
  }
}
```

Y nuestro Template:

```
<p>Listado de Heroes</p>

<button (click)="borrarElemento()">Borrar Último</button>

<h2>Héroe Borrado: {{heroeBorrado}}</h2>

<ul>
  <li *ngFor="let heroe of heroes; let i = index">{{ i + 1 }} - {{heroe}}
</li>
</ul>
```

Recorremos el arreglo Heroes y luego imprimimos el contenido en un *ul*

Directivas *ngIf y else

Actualicemos nuestra plantilla con el siguiente código:

```
<p>Listado de Heroes</p>

<button *ngIf="this.heroes.length" (click)="borrarElemento()">
  Borrar Último
</button>
<button *ngIf="this.heroes.length == 0" (click)="reiniciarLista()">
  Reiniciar
</button>

<h2 *ngIf="heroeBorrado.length; else noBorrado">
  Héroe Borrado: {{heroeBorrado}}
</h2>

<ng-template #noBorrado>
  <h2>No Borrado</h2>
</ng-template>

<ul>
  <li *ngFor="let heroe of heroes; let i = index">{{ i + 1 }} - {{heroe}}
</li>
</ul>
```

La directiva **ng-if="expresión"** permite mostrar un elemento dependiendo del valor retornado al evaluar la expresión.

En el caso de los botones, se mostrará uno u otro en dependencia de la expresión "this.heroes.length" y "this.heroes.length == 0" respectivamente.

También podemos incorporar un **Else** con una referencia local (hacia una plantilla local) tal como se hace con el mensaje de borrado:

```
<h2 *ngIf="heroBorrado.length; else noBorrado" >Héroe Borrado:  
{{heroBorrado}}</h2>  
  
<ng-template #noBorrado>  
  <h2>No Borrado</h2>  
</ng-template>
```

Si **heroBorrado.length** es true ejecutará el bloque inicial H2, caso contrario renderizará la plantilla con nombre **noBorrado** la cual imprimir un H2 con texto No Borrado.

Modulos

Implementaremos esta pantalla que consta de varios componentes, un listado y un formulario para agregar nuevos elementos al listado.

El objetivo de los módulos es ayudarnos a agrupar piezas de código que tienen relación entre sí. Permite tener un código mejor organizado, por ejemplo, actualmente tenemos los componentes:

- contador
- Heroes
 - Heroe
 - Listado

Todos estos componentes son manejados en el módulo principal de la APP:

```
@NgModule({  
  declarations: [  
    AppComponent,  
    ContadorComponent,  
    HeroeComponent,  
    ListadoComponent
```

```
],
imports: [
  BrowserModule
],
providers: [],
bootstrap: [AppComponent]
})
```

Notar como los componentes **ContadorComponent**, **HeroeComponent**, **ListadoComponent** son agregados al arreglo de las **declarations**, que pasa si nuestra app tiene 100 componentes? no podemos manejar 100 Componentes desde el APP Module, de modo que aca es donde entra la importancia de organizar el código en módulos.

Vamos a crear un nuevo Módulo para los componentes de Heroes:

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
import { HeroeComponent } from './heroe/heroe.component';
import { ListadoComponent } from './listado/listdado.component';

@NgModule({
  declarations: [HeroeComponent, ListadoComponent],
  exports: [HeroeComponent, ListadoComponent],
  imports: [CommonModule],
})
export class HeroesModule {}
```

Este archivo se crea en la raíz del directorio Heroes, con el nombre: **heroes.module.ts**

```
└── heroes
    ├── heroes.module.ts
    ├── heroe
    │   ├── heroe.component.html
    │   └── heroe.component.ts
    └── listado
        ├── listado.component.html
        └── listdado.component.ts
```

La sección **declarations** indica que componentes y declarativas pertenecen al módulo.

La sección de **exports** indica que componentes serán públicos y podrán ser usado por otros módulos que importen el módulo actual.

Mientras tanto la sección **import** indica que otros módulos serán usados por el módulo actual.

NOTA: el **CommonModule** se importa en aquellos componentes que utilizan directivas como `*ngFor`, `*ngIf` (entre otras) o pipes como currency pipe, Date pipe etc.

Crear Módulos con el Generador del CLI

Ejecute el comando:

```
$ ng generate module dbz
CREATE src/app/dbz/dbz.module.ts (189 bytes)
```

El módulo autogenerado es:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  declarations: [],
  imports: [CommonModule],
})
export class DbzModule {}
```

A continuación agregamos un componente al módulo recién creado:

```
$ ng generate component dbz/mainPage
CREATE src/app/dbz/main-page/main-page.component.css (0 bytes)
CREATE src/app/dbz/main-page/main-page.component.html (24 bytes)
CREATE src/app/dbz/main-page/main-page.component.spec.ts (641 bytes)
CREATE src/app/dbz/main-page/main-page.component.ts (286 bytes)
UPDATE src/app/dbz/dbz.module.ts (283 bytes)
```

Ahora en la sección de **declarations** de nuestro módulo tenemos el nuevo componente.

```
declarations: [MainPageComponent];
```

Angular CLI no ingresa el nuevo componente en la lista de Exports del Módulo, ya que puede ser un componente privado.

Form Module

El siguiente bloque HTML genera un formulario el cual al hacer click en el botón Agregar refresca toda la página. en SPA esto no es un comportamiento aceptado.

```
<form>
  <input type="text" placeholder="Nombre" />
  <input type="number" placeholder="Poder" />
```

```
<button type="submit">Agregar</button>
</form>
```

Para evitar el evento de recargar la página durante el submit del Form podemos "prevenir" ese default event:

```
<form (submit)="agregar($event)">
  <input type="text" placeholder="Nombre" />
  <input type="number" placeholder="Poder" />
  <button type="submit">Agregar</button>
</form>
```

Primero a nivel del form agregamos un eventListener **submit** y lo asignamos a la función agregar que recibe como parametro el evento que invoca el submit, en este caso el mismo submit del formulario.

```
<form (submit)="agregar($event)"></form>
```

Luego agregamos la función en el componente:

```
agregar(event: Event) {
  event.preventDefault();
  console.log("Agregar");
}
```

La instrucción **event.preventDefault()**; previene que el evento por default del Submit se ejecute, en este caso recargar la página por completo. Con estos cambios, logramos que al hacer click en el botón agregar (submit) no se recargue la página.

Angular al ser un Framework dispone de una serie de herramientas que facilitan el desarrollo. En este caso específico, podemos evitarnos el trabajo de enviar el evento y llamar el preventDefault. Para ellos usaremos el módulo **FormsModule**

Dado que el componente donde vamos a usarlo está contenido en el módulo dbz, vamos a proceder a importar el *FormsModule* en dicho módulo, de la siguiente forma:

```
imports: [
  CommonModule,
  FormsModule
],
```

Ahora podemos usar el evento **ngSubmit**

```
<form (ngSubmit)="agregar()">
  <input type="text" placeholder="Nombre" />
  <input type="number" placeholder="Poder" />
  <button type="submit">Agregar</button>
</form>
```

Notar como se ha quitado el parametrono **event** y en la función agregar ya no es necesario agregar el *preventDefault*. Con estos cambios logramos el mismo efecto, no recargar la página, pero en este caso lo está manejando Angular.

Ngmodel

Creemos ahora un objeto en nuestro componente:

```
nuevo: Personaje = {
  nombre: 'Trunks',
  poder: 14000,
};
```

Modifiquemos nuestro input en el template para enlazar un campo del formulario con una propiedad del componente

```
<input type="text" placeholder="Nombre" [value]="nuevo.nombre" />
```

Notar el uso de [] en value, esto hace que la expresión del otro lado del signo igual sea evaluada como una propiedad o función que retorna un dato desde el componente y será enlazado al control input.

Esto se denomina Enlace de datos unidireccional (**One Way Data Binding**) ya que el componente envía los datos al input, pero si cambiamos el input, el objeto en el componente NO cambia.

Para lograr que cualquier cambio en el input afecte el valor del objeto a nivel de componente podríamos agregar un *evenListener* y llamar un metodo para actualizar el valor:

```
<input
  type="text"
  placeholder="Nombre"
  [value]="nuevo.nombre"
  (input)="cambiarNombre()"
/>
```

Y del lado del componente podríamos leer el **event.target.value** y actualizar el objeto.

```
cambiarNombre(event: any) {
  var nombre = event.target.value;
  //cambiar el valor nombre en el objeto...
}
```

Nuevamente angular viene al rescate y nos hace la vida mas simple, podemos usar el **ngModel**, este crea una instancia de **FormControl** a partir de un modelo de dominio y la vincula a un elemento de control de formulario.

Cambiemos nuestro input a esto:

```
<input
  type="text"
  placeholder="Nombre"
  name="nombre"
  [(ngModel)]="nuevo.nombre"
/>
```

**** [(ngModel)]="nuevo.nombre"****

Primero, el uso de [] indica que haremos el binding y el uso de () indica que emitiremos un evento, al igual que antes, pero, ngModel se encargará de hacer todo el trabajo, no necesitamos crear un método adicional que acepte el valor y lo guarde en el componente, lo cual aún podría ser necesario si hay que realizar algún tratamiento especial del dato, pero en este caso el código es suficiente para hacer el binding en los dos sentidos.

Agreguemos un arreglo en nuestro componente:

```
personajes: Personaje[] =
[
  {
    nombre: 'Goku',
    poder: 15000
  },
  {
    nombre: 'Vegeta',
    poder: 7500
  },
];
```

Y el siguiente método:

```
agregar() {
  if (this.nuevo.nombre.trim().length == 0) {
    return
  }
```

```

    this.personajes.push(this.nuevo);
    this.nuevo = {
      nombre: '',
      poder: 0
    }
}

```

Cambiemos nuestra template para renderizar la lista de personajes desde un arreglo con ngFor

```

<h3>Personajes</h3>
<ul class="list-group">
  <li *ngFor="let personaje of personajes; let pos = index" class="list-group-item">
    <span class="text-primary">{{pos + 1}}.</span>
    {{personaje.nombre}} - {{personaje.poder | number}}
  </li>
</ul>

```

Notar que luego de personajes poder estamos usando un pipe, number, el cual hace que el valor **poder** se imprima como un número (separador de miles, decimales, etc.)

Otro elemento a notar es el `let pos = index` dentro del ngFor, hay otras variables que se pueden usar como el `isFirst`, `isLast`, etc.

Listo, con esto podemos agregar elementos a la lista al momento de presionar "agregar".

nGClass

Podemos aplicar una clase basada en una condición, por ejemplo si queremos agregar al

- un color de fondo, podríamos usar las variables del for y el ngClass.

```

<h3>Personajes</h3>
<ul class="list-group">
  <li *ngFor="let personaje of personajes; let pos = index; let isEven =
even;" class="list-group-item"
    [ngClass] = "'list-group-item-success': isEven">
      <span class="text-primary">{{pos + 1}}.</span>
      {{personaje.nombre}} - {{personaje.poder | number}}
    </li>
</ul>

```

`[ngClass] = "'list-group-item-success': isEven">` agrega la clase *list-group-item-success* únicamente a los elementos Pares de la lista.

NOTA: dado que el índice del ciclo inicia en 0, se toma como par realmente los impares.

Angular APP!

Drangon Ball Z

Personajes

1. Goku - 15,000
2. Vegeta - 7,500
3. Maestro Roshi - 1,000
4. Otro - 100

Crear Componentes Hijo

Cuando un componente crece en complejidad, una señal puede ser muchos métodos en el componente o mucho código html en el template se aconseja crear componentes hijos para reducir dicha complejidad, por ejemplo dado el siguiente template:

```
<h1>Drangon Ball Z</h1>
<hr />

<div class="row">
  <div class="col">
    <h3>Personajes</h3>
    <ul class="list-group">
      <li *ngFor="let personaje of personajes; let pos = index; let isEven = even;">
        <span class="text-primary">{{pos + 1}}</span>
        {{personaje.nombre}} - {{personaje.poder | number}}
      </li>
    </ul>
  </div>

  <div class="col">
    <h3>Agregar</h3>
    <hr />
    <form (ngSubmit)="agregar()">
      <input
        type="text"
        placeholder="Nombre"
        name="nombre"
        [(ngModel)]="nuevo.nombre" />
    </form>
  </div>
</div>
```

```

    />
<input
  type="number"
  placeholder="Poder"
  name="poder"
  [(ngModel)]="nuevo.poder"
/>
<button type="submit">Agregar</button>
</form>
</div>
</div>

```

Es probable que lo ideal sería separar el listado y el formulario en componentes separados. Vamos a proceder a relazar dicha separación de componentes. En la línea de comandos generamos el nuevo componente

```
$ ng generate component dbz/personajes --skipTests
```

Luego en el template del nuevo componente moveremos el HTML de la lista

```

<h3>Personajes</h3>
<ul class="list-group">
  <li *ngFor="let personaje of personajes; let pos = index; let isEven =
even;" 
    class="list-group-item"
    [ngClass] = "{'list-group-item-success': isEven}">
    <span class="text-primary">{{pos + 1}}.</span>
    {{personaje.nombre}} - {{personaje.poder | number}}
  </li>
</ul>

```

Luego necesitamos pasar la lista de personajes que es manejado por el componente main-page para el nuevo componente pueda renderizarlo.

En el nuevo componente usaremos @Input para indicar que el objeto personajes será entregado por el componente padre.

```

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

@Component({
  selector: 'app-personajes',
  templateUrl: './personajes.component.html',
})
export class PersonajesComponent {
  @Input() personajes: any[] = [];
}

```

Y finalmente el componente padre debe pasar el valor de personajes de la siguiente forma>

```
<app-personajes [personajes]="personajes"> </app-personajes>
```

Recordemos que el uso de [propiedad] = "valor" indica que en el componente destino existe una propiedad de nombre **propiedad** la cual va a recibir el valor externo de **valor**

Podríamos crear un alias en el input para renombrar la propiedad:

```
@Input('data') personajes: any[] = [];
```

Luego tendríamos que cambiar el template del componente hijo para que use la variable Data:

```
<li *ngFor="let personaje of data">
  {{personaje.nombre}} - {{personaje.poder | number}}
</li>
```

Usando la Interfaz Personaje

En La declaración siguiente sería idela usar la interfaz personaje que habiamos definido en el componente main-page:

```
@Input('data') personajes: any[] = [];
```

Asi que en lugar de any[] deberíamos de tener el tipo **Personaje[]** para ello vamos a crear nuestra interfaz en un nivel superior, de modo que ambos componentes pueda usarla.

Crear directorio **interfaces** en el directorio **dbz** y luego crear el archivo **personaje.interface.ts** con el siguiente contenido:

```
export interface Personaje {
  nombre: string;
  poder: number;
}
```

Finalmente eliminamos el any y colocamos el tipo correcto.

```
@Input() personajes: Personaje[] = []
```

Se tiene que corregir el main-page ya que allí estabamos usando la misma interfaz pero fue removida,

```
import { Personaje } from './interfaces/dbz.interface';

personajes: Personaje[] =
[
  {
    nombre: 'Goku',
    poder: 15000
  },
  {
    nombre: 'Vegeta',
    poder: 7500
  },
];
```

Listo todo debería de funcionar exactamente igual.

Crear Otro Componente Hijo

Vamos a extraer el formulario y la acción agregar en un componente hijo,

```
$ ng generate component dbz/agregar
```

Vamos a mover parte del código html del componente main-page al template del agregar, quedaría de la siguiente forma:

```
<h3>Agregar</h3>
<hr />
<form (ngSubmit)="agregar()">
  <input
    type="text"
    placeholder="Nombre"
    name="nombre"
    [(ngModel)]="nuevo.nombre"
  />
  <input
    type="number"
    placeholder="Poder"
    name="poder"
    [(ngModel)]="nuevo.poder"
  />
  <button type="submit">Agregar</button>
</form>
```

De igual manera, vamos a mover código del componente main-page (archivo typescript) al nuevo componente:

```

import { Component, Input, OnInit } from '@angular/core';
import { Personaje } from '../interfaces/dbz.interface';

@Component({
  selector: 'app-agregar',
  templateUrl: './agregar.component.html',
})
export class AgregarComponent {
  @Input() personajes: Personaje[] = [];

  @Input() nuevo: Personaje = {
    nombre: '',
    poder: 0,
  };

  agregar() {
    if (this.nuevo.nombre.trim().length == 0) {
      return;
    }

    this.personajes.push(this.nuevo);
    this.nuevo = {
      nombre: '',
      poder: 0,
    };
  }
}

```

personajes y **nuevo** son propiedades que se pasarán desde el componente padre.

Y nuestras plantilla html de nuestro main-page component quedará así:

```

<h1>Drangon Ball Z</h1>
<hr />

<div class="row">
  <div class="col">
    <app-personajes [personajes]="personajes"> </app-personajes>
  </div>

  <div class="col">
    <app-agregar [personajes]="personajes" [nuevo]="nuevo"></app-agregar>
  </div>
</div>

```

Todo debe de funcionar igual, pero tenemos un problema o un error de diseño, resulta que el método agregar del componente hijo recibe como referencia el arreglo de personajes y el componente hijo es el que se encarga de hacer el push al arreglo que se le pasa por medio del @Input, esto no es correcto, y lo

que debería de suceder es que el componente hijo emita un evento a su Padre y sea el quien modifique el arreglo, el cual esta definido en su scope.

En otras palabras, si un objeto es manejado por un componente, y un segundo componente intenta modificarlos, la forma correcta es que el segundo componente emita algún tipo de evento que sea escuchado por el primer componente (el dueño del objeto) y sea éste el que al final actualice sus propiedades.

Outputs y EventEmiter

Haremos cambios en el componente *agregar*, primero no vamos a pasar el arreglo *personajes*, únicamente vamos a pasar el objeto nuevo, para inicializar los campos del formulario. Recordemos que estamos usando *ngModel* para habilitar el **two way binding** entre el formulario y el objeto *nuevo*

En el template del main-page dejemos el app-agregar de la siguiente forma:

```
<div class="col">
  <app-agregar [nuevo]="nuevo"></app-agregar>
</div>
```

Y en el componente *agregar*, vamos a crear una nueva propiedad, usaremos **@Output()** y será de tipo *EventEmitter*.

```
@Output() onNewPersonaje: EventEmitter<Personaje> = new EventEmitter();
```

Cuando hacemos click en *agregar*, el *ngModule* mapea los valores del formulario al objeto **nuevo** definido en el componente *agregar* y por lo tanto necesitamos emitir un evento con este nuevo personaje para que el evento padre pueda obtener ese valor y proceder con la actualización del objeto *Personajes*.

Así quedaría parte de nuestro *agregarComponent*:

```
@Output() onNewPersonaje: EventEmitter<Personaje> = new EventEmitter();

agregar() {
  if (this.nuevo.nombre.trim().length == 0) {
    return
  }

  this.onNewPersonaje.emit(this.nuevo)

  this.nuevo = {
    nombre: '',
    poder: 0
  }
}
```

Ahora necesitamos capturar ese evento en nuestro *main-page-component*, es decir el componente padre y crear la lógica necesaria para agregar el elemento nuevo al arreglo de personajes:

onNewPersonaje es el nombre del *customEvent* que hemos creado en el componente hijo, entonces con () vamos a especificar el nombre de ese evento y al recibirla, vamos a mandar a llamar un nuevo método, que será el responsable de agregar el elemento al arreglo.

```
<div class="col">
  <app-agregar [nuevo]="nuevo"
    (onNewPersonaje)="agregarNuevoPersonaje($event)">
    </app-agregar>
</div>
```

Finalmente creamos el método en el componente Padre:

```
agregarNuevoPersonaje( arg: Personaje ) {
  this.personajes.push(arg);
}
```

Y listo, ahora el componente Padre, dueño del arreglo personaje, es quien actualiza dicho arreglo y no otro componente.

Debugger con VS Code

Presione F5, en el archivo que se muestra cambiar el puerto al puerto donde corre la app de Angular-

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "pwa-chrome",
      "request": "launch",
      "name": "Launch Chrome against localhost",
      "url": "http://localhost:4200",
      "webRoot": "${workspaceFolder}"
    }
  ]
}
```

Cerrar Archivo y presionar nuevamente F5, para esto necesita que se esté ejecutando:

```
$ ng server -o
```

Servicios

Los servicios es uno de los conceptos más fuertes de Angular, permite implementar patrones de diseño sin la necesidad de Redux por ejemplo.

Los servicios es otra forma de organizar el código, a medida que nuestra APP crece en número de componentes, es posible que muchos de estos componentes dupliquen cierto código, cierta funcionalidad. Podemos extraer dicha funcionalidad a un servicio.

Dado que los servicios son compartidos por varios componentes, suelen crearse dentro de módulos compartidos, que pueden llamarse common, shared, etc.

Otra de las utilidades de los servicios parte del hecho de que los componentes no deberían buscar o guardar datos directamente, el acceso a los datos es una tarea que debe delegarse a un servicio.

También los servicios son una forma para compartir funcionalidades entre clases que no se conocen entre sí.

Creando un servicio de Manera Manual

En el modulo de *dbz*, creamos una carpeta llamada services, en este caso, los servicios que se guarden en esta carpeta serán utilizado por todos los componentes del módulo *dbz*. Si necesitaramos uno a nivel global se puede crear la carpeta services en el directorio *app/*

Dentro del nuevo directorio crear el archivo **dbz.service.ts**

Crearmos un servicio con un constructor sencillo

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

@Injectable()
export class DbzService {
  constructor() {
    console.log('Servicio Inicializado');
  }
}
```

Luego en el *dbz.module.ts* agregaremos una nueva sección:

```
providers: [DbzService];
```

Los *providers* es un arreglo de Servicios.

Hasta el momento el servicio no ha sido utilizado por ningún modulo, clase, servicio, etc, por lo tanto el constructor no se ha ejecutado aún.

Vamos a refactorizar un poco el código del main-page component, recordemos este componente como ha sido desarrollado:

```
import { Component } from '@angular/core';
import { Personaje } from '../interfaces/dbz.interface';

@Component({
  selector: 'app-main-page',
  templateUrl: './main-page.component.html',
})
export class MainPageComponent {
  nuevo: Personaje = {
    nombre: 'Maestro Roshi',
    poder: 1000,
  };

  personajes: Personaje[] = [
    {
      nombre: 'Goku',
      poder: 15000,
    },
    {
      nombre: 'Vegeta',
      poder: 7500,
    },
  ];
}

agregarNuevoPersonaje(arg: Personaje) {
  this.personajes.push(arg);
}
```

Tal como se indicó anteriormente, el componente no debe ser responsable de buscar o guardar la data, eso se tiene que delegar a un servicio.

Vamos a usar la Inyección de dependencias. El servicio *dbzService* ha sido definido como *@Injectable* con lo cual podemos injectarlo en el constructor de cualquier componente. Obviamente porque también hemos dicho que este servicio es un provider.

Si agregamos el constructor al main-page componente

```
constructor(private dbzService: DbzService) {
```

Esto inyecta el servicio y ejecutará el constructor del servicio. Si adicionamos el mismo contructor a otro componente, Angular solo creará una única vez el servicio y podrá ser usado por el segundo componente. Es decir, nuestro constructor solo se ejecuta una sola vez.

Podríamos hacer estos cambios:

- El primer cambio es mover el arreglo de personajes del main-page componente al servicio.
- El siguiente cambio es crear una propiedad *personajes* en el componente y luego inicializarlo desde el Servicio. Por el momento lo haremos desde el contructor, pero la idea es usar los ciclos de vida del componente para inicializar datos.

Eso funcionaría, Pero podríamos aplicar otro enfoque, el siguiente:

- Establecer un método *get personajes()* que retorne la lista de personajes que viene del servicio, este sería como la operación de búsqueda de la Data que queremos tener en el componente. Recordemos que este componente usa un componente hijo *personaje* el cual crea la lista de personajes

Pero ciertamente el *main-page* no necesita manejar la lista de personajes, esto podría hacerlo el componente *personaje* quien es al final el que imprime la lista, entonces podemos inyectar el servicio directamente al el componente hijo.

Métodos en el Servicio

Tal como lo hemos indicado, el servicio es el responsable de actualizar los datos, en este caso necesitamos un método *AgregarPersonaje* a nivel del servicio y éste debe ser llamado por el componente *AgregarPersonaje*.

Este componente anteriormente emitía un evento, y el *main-page* capturaba ese evento y actualizaba los datos, pues ahora no es necesario emitir evento puesto que el servicio es el que se encarga de agregar los datos.

Esto debe de eliminarse

```
@Output() onNewPersonaje: EventEmitter<Personaje> = new EventEmitter();
```

Al final el código quedaría de la siguiente forma:

Servicio:

```
import { Injectable } from '@angular/core';
import { Personaje } from '../interfaces/dbz.interface';

@Injectable()
export class DbzService {
  private _personajes: Personaje[] = [
    {
      nombre: 'Goku',
      poder: 15000,
```

```

    },
    {
      nombre: 'Vegeta',
      poder: 7500,
    },
  ];

  get personajes(): Personaje[] {
    return [...this._personajes];
  }

  agregarPersonaje(personaje: Personaje) {
    this._personajes.push(personaje);
  }
}

```

Main-Page Component:

```

import { Component } from '@angular/core';
import { Personaje } from '../interfaces/dbz.interface';
import { DbzService } from '../services/dbz.service';

@Component({
  selector: 'app-main-page',
  templateUrl: './main-page.component.html',
})
export class MainPageComponent {
  constructor() {}

  nuevo: Personaje = {
    nombre: 'Maestro Roshi',
    poder: 1000,
  };
}

```

Main-Page template

```

<h1>Drangon Ball Z</h1>
<hr />

<div class="row">
  <div class="col">
    <app-personajes></app-personajes>
  </div>

  <div class="col">
    <app-agregar [nuevo]="nuevo"> </app-agregar>
  </div>
</div>

```

Personaje Component:

```
import { Component, Input } from '@angular/core';
import { Personaje } from '../interfaces/dbz.interface';
import { DbzService } from '../services/dbz.service';

@Component({
  selector: 'app-personajes',
  templateUrl: './personajes.component.html',
})
export class PersonajesComponent {
  constructor(private dbzService: DbzService) {}

  get personajes() {
    return this.dbzService.personajes;
  }
}
```

Personaje Template:

```
<h3>Personajes</h3>
<ul>
  <li *ngFor="let personaje of personajes">
    {{personaje.nombre}} - {{personaje.poder | number}}
  </li>
</ul>
```

Agregar Component:

```
import { Component, Input } from '@angular/core';
import { Personaje } from '../interfaces/dbz.interface';
import { DbzService } from '../services/dbz.service';

@Component({
  selector: 'app-agregar',
  templateUrl: './agregar.component.html',
})
export class AgregarComponent {
  constructor(private dbzService: DbzService) {}

  @Input() nuevo: Personaje = {
    nombre: '',
    poder: 0,
  };

  agregar() {
    if (this.nuevo.nombre.trim().length == 0) {
```

```

        return;
    }

    this.dbzService.agregarPersonaje(this.nuevo);

    this.nuevo = {
        nombre: '',
        poder: 0,
    };
}
}

```

Agregar Template:

```

<h3>Agregar</h3>
<hr />
<form (ngSubmit)="agregar()">
    <input
        type="text"
        placeholder="Nombre"
        name="nombre"
        [(ngModel)]="nuevo.nombre"
    />
    <input
        type="number"
        placeholder="Poder"
        name="poder"
        [(ngModel)]="nuevo.poder"
    />
    <button type="submit">Agregar</button>
</form>

```

Un par de cambios más

Agregamos un método eliminar personaje para permitir remover de la lista un personaje:

En el servicio agregamos el nuevo método:

```

eliminarPersonaje( index: number ) {
    this._personajes.splice(index, 1);
}

```

En el template de la lista de personajes, agregamos un botón al final y adicionalmente reordenamos para mejorar la interfaz, incluimos algunas clases de bootstrap también

```

<h3>Personajes</h3>
<ul class="list-group">

```

```

<li *ngFor="let personaje of personajes; let pos = index; let isEven =
even;" class="list-group-item d-flex justify-content-between align-items-center" [ngClass] = "{'list-group-item-success': isEven}">

    <div>
        <span class="text-primary">{{pos + 1}}.</span>
        <span>{{personaje.nombre}}</span>
        <span> - </span>
        <strong>Power: </strong>
        <span>{{personaje.poder | number}}</span>
    </div>

    <button (click)="onEliminarPersonaje(pos)" class="btn btn-danger">X</button>

</li>
</ul>

```

Finalmente enlazamos la vista con el servicio, esto lo logramos agregando el siguiente método al componente de personajes

```

public onEliminarPersonaje(index: number): void {
    this.dbzService.eliminarPersonaje(index);
}

```

Implementar identificador único para cada personaje

En lugar de eliminar por index, vamos a eliminar por ID, y agregaremos una libreria que nos permita generar códigos UUID.

Instalamos:

```
$ npm install uuid
```

NOTA: algunas librerías están escritas en Javascript, para poder usarlas en nuestro ambiente de desarrollo con typescript, debemos importar su archivo de definición de tipos.

```
$ npm install --save-dev @types/uuid
```

Luego modificamos la interfaz para incluir el nuevo campo id

```
export interface Personaje {  
  + id: string;  
  nombre: string;  
  poder: number;  
}
```

En es servicio haremos estos cambios

```
import { Injectable } from "@angular/core";  
+import { v4 as uuid } from "uuid";  
import { Personaje } from "../interfaces/dbz.interface";  
  
@Injectable()  
@@ -7,18 +8,22 @@ export class DbzService {  
  private _personajes: Personaje[] =  
  [  
    {  
      + id: uuid(),  
      nombre: 'Goku',  
      poder: 15000  
    },
```

Y en nuestro agregar Component, justo antes de guardar en la lista el nuevo personaje, generamos el UUID()

```
agregar() {  
  if (this.nuevo.nombre.trim().length == 0) {  
    return  
  }  
  
  //Generar un id unico  
  this.nuevo.id = uuid();  
  
  this.dbzService.agregarPersonaje(this.nuevo);  
  
  this.nuevo = {  
    id: '',  
    nombre: '',  
    poder: 0  
  }  
}
```

Generar Build de la Aplicación

```
$ ng build --prod
```

En versiones recientes, no es necesario el parametro --prod

```
$ ng build --prod
Option "--prod" is deprecated: No need to use this option as this builder
defaults to configuration "production".
✓ Browser application bundle generation complete.
✓ Copying assets complete.
✓ Index html generation complete.

Initial Chunk Files          | Names           | Raw Size | Estimated
Transfer Size
main.6b9c85c0379b42ab.js    | main            | 147.28 kB |
41.18 kB
polyfills.f21860001d995b00.js | polyfills       | 33.01 kB |
10.64 kB
runtime.d15e317f76fa3172.js  | runtime         | 1.04 kB |
600 bytes
styles.dc3e7ac02ccd86e8.css  | styles          | 464 bytes |
218 bytes

                                         | Initial Total | 181.78 kB |
52.62 kB

Build at: 2022-05-13T00:17:29.695Z - Hash: 9ca419420a6538bb - Time: 10607ms
```

Se generan los siguientes archivos:

- main.js: Contiene nuestro código
- polyfills: contiene funciones para asegurar que nuestra app se pueda ejecutar en diferentes browsers.
- runtime: código utilizado por Webpack para cargar código en tiempo de ejecución
- style.js: Nuestros estilos css.

El hash en el nombre del archivo garantiza que se utilice la última versión del archivo en lugar del archivo en cache.

Esta información es almacenada en el directorio /dist

```
$ tree dist/
dist/
└── bases
    ├── 3rdpartylicenses.txt
    ├── favicon.ico
    ├── index.html
    ├── main.6b9c85c0379b42ab.js
    ├── polyfills.f21860001d995b00.js
    ├── runtime.d15e317f76fa3172.js
    └── styles.dc3e7ac02ccd86e8.css
```

```
1 directory, 7 files
```

Desplegando nuestra aplicación

una vez construido el proyecto, podemos copiar el directorio bases y servirlo en un servidor web.

Eso genera los archivos en el directorio dist.

Podemos agregar esta configuración en apache:

```
<VirtualHost *:80>
    ServerName localhost
    ServerAlias udemy-angular.com 127.0.0.1

    DocumentRoot /home/fcruz/Cursos/udemy/angular/code/udemy-angular/01-
bases/dist/bases

    <Directory /home/fcruz/Cursos/udemy/angular/code/udemy-angular/01-
bases/dist/bases>
        Options Indexes FollowSymLinks
        AllowOverride All
        Require all granted
    </Directory>

    ErrorLog /var/log/httpd/angular-app_error.log
    CustomLog /var/log/httpd/angular-app_access.log combined
</VirtualHost>
```

Luego debemos cambiar permisos para que el httpd pueda ejecutar nuestros archivos

```
sudo chcon -R -t httpd_sys_content_t /home/fcruz/Cursos

sudo chmod +x /home/fcruz /home/fcruz/Cursos /home/fcruz/Cursos/udemy
/home/fcruz/Cursos/udemy/angular /home/fcruz/Cursos/udemy/angular/code
/home/fcruz/Cursos/udemy/angular/code/udemy-angular
/home/fcruz/Cursos/udemy/angular/code/udemy-angular/01-bases

sudo chmod -R +r /home/fcruz/Cursos/udemy/angular/code/udemy-angular/01-
bases

sudo systemctl restart httpd
```

Agregamos en el /etc/hosts

```
127.0.0.1 udemy-angular.com
```

probar link:

<http://udemy-angular.com>

Nueva Sección: Nueva Aplicación:

¿Qué veremos en esta sección?

La sección contendrá nuestra primera aplicación real de Angular, este es un breve listado de los temas fundamentales:

Modularización de la aplicación

Estructura de la aplicación de media a gran escala

Componentes

ViewChild

Consumir servicios

Historial de búsquedas

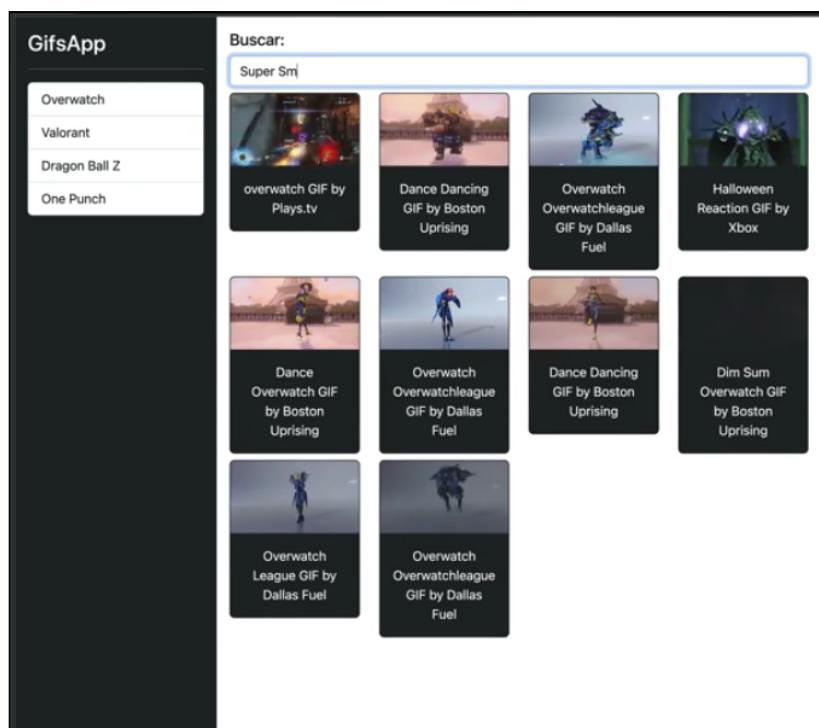
Uso de Api Keys

Guardar información en local storage.

Peticiones HTTP

Animaciones mediante css

Vamos a construir una nueva aplicación que consiste en un buscador de GIF, contiene un historial y un grid de resultados.



Creamos el nuevo proyecto

```
$ ng new 02-gifsApp
```

Vamos a trabajar con bootstrap, para ellos iremos al sitio oficial y copiaremos el CDN link para el CSS únicamente

```
<!-- CSS only -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6D0itfPri4tjfHxaWutUpFmBp4vmVor" crossorigin="anonymous">
```

Pegamos este link en la sección *head* del archivo index.html

```
./02-gifsApp/src/index.html
```

Luego borraremos toda la información en el archivo *code/udemy-angular/02-gifsApp/src/app/app.component.html* y crearemos nuestro diseño base HTML:

```
<div class="d-flex">
  <!-- Este es el sidebar -->
  <div class="bg-dark border-right p-2" id="sidebar">
    <h3 class="text-light">Gifs-App</h3>
    <hr class="text-white" />
```

```
<div class="list-group list-reset">
  <a href="#" class="list-group-item list-group-item-action"> DashBoard
</a>
</div>
</div>
<!-- Fin del sidebar -->

<div class="container">
  <div class="row p-3">
    <div class="col">
      <h5>Buscar:</h5>
      <input type="text" class="form-control" placeholder="Buscar Gifs"
/>
      </div>
    </div>
    <hr />
    <div class="row">
      <div class="col">Estos es un texto temporal</div>
    </div>
  </div>
</div>
```

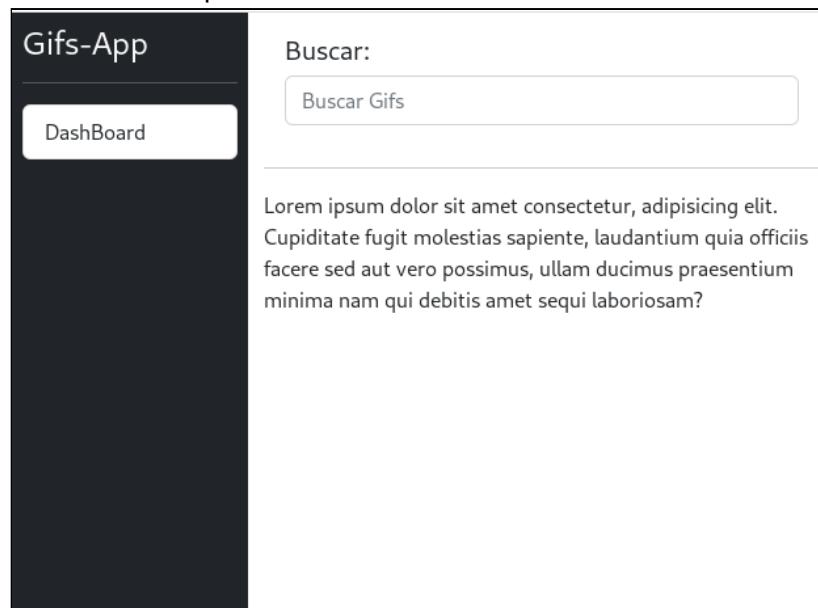
Agregamos el siguiente estilo al archivo principal *style.css*

```
html,
body {
  height: 100%;
}

#sidebar {
  height: 100%;
  min-height: 100vh;
  min-width: 180px;
}
```

Y listo, tenemos un diseño básico de lo que será nuestras APP. En contenedor (en el que aparece el texto temporal) mostrará el resultado de las búsqueda de GIF, y la barra lateral (sideBar) contendrá la lista de las

últimas 10 búsquedas.



Crear Módulo y Componente Sidebar

Creamos el Modulo y el Componente:

```
$ ng g m shared
CREATE src/app/shared/shared.module.ts (192 bytes)

$ ng g c shared/sidebar
CREATE src/app/shared/sidebar/sidebar.component.css (0 bytes)
CREATE src/app/shared/sidebar/sidebar.component.html (22 bytes)
CREATE src/app/shared/sidebar/sidebar.component.spec.ts (633 bytes)
CREATE src/app/shared/sidebar/sidebar.component.ts (279 bytes)
UPDATE src/app/shared/shared.module.ts (280 bytes)
```

Movemos el HTML del main template hacia */shared/sidebar/sidebar.component.html*

Hacemos la exportación en el modulo shared del componente creado

Importamos el Módulo Share en el app.module.ts

Inluimos el en el template principal y listo.

Crar nuevo Módulo gifs

Vamos a incluir todas las utilidades referente a los gifs en este módulo.

Generamos el módulo

```
$ ng g m gifs
CREATE src/app/gifs/gifs.module.ts (190 bytes)
```

Y seguidamente lo importamos en el app.module.ts

Nuevo Complemento gifs-page

Creamo el componente:

```
$ ng g c gifs/gifsPage --skip-tests -is
CREATE src/app/gifs/gifs-page/gifs-page.component.html (24 bytes)
CREATE src/app/gifs/gifs-page/gifs-page.component.ts (259 bytes)
UPDATE src/app/gifs/gifs.module.ts (284 bytes)
```

NOTA: -is (internal style) omite la creación del archivo css:

Exportamos el componente en el gifs.module.ts

Importamos el modulo gifs en el app.module.ts

Creamos nuestra etiqueta app-gifs-page en nuestro main template

Crear search Component

La barra de búsquedas será un componente aparte, dentro del módulo de gifs, procedemos:

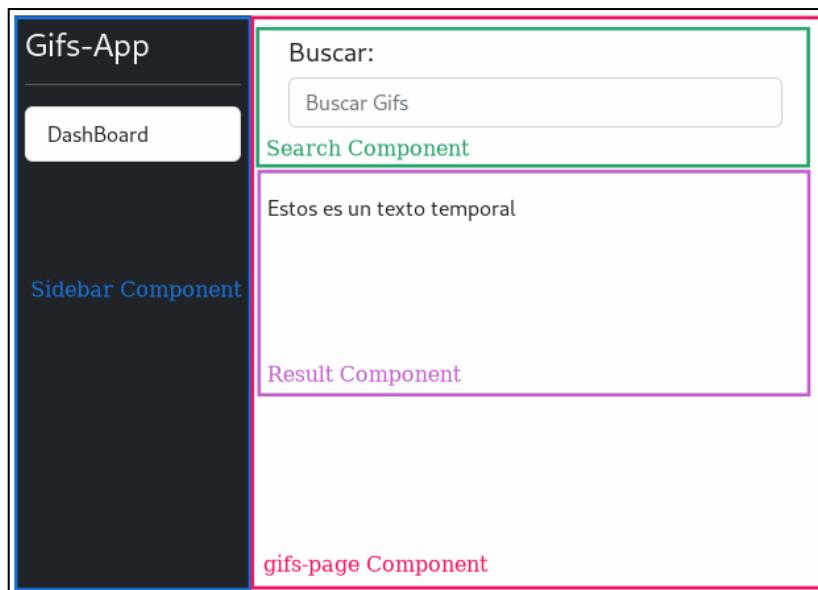
```
$ ng g c gifs/search --skip-tests -is
CREATE src/app/gifs/search/search.component.html (21 bytes)
CREATE src/app/gifs/search/search.component.ts (275 bytes)
UPDATE src/app/gifs/gifs.module.ts (406 bytes)
```

Este es un componente interno del módulo gifs por lo tanto no lo exportaremos, es de consumo interno.

Creamos el componente results

Este contendrá los resultados de las búsquedas

Al final hemos separado el diseño original en componentes separados.



Hasta este punto los templates quedan de la siguiente forma:

El Main template debería de verse mas limpio de HTML:

app.template.html:

```
<div class="d-flex">
  <!-- Este es el sidebar -->
  <app-sidebar></app-sidebar>
  <!-- Fin del sidebar -->

  <!-- Este es el gifs-page component -->
  <app-gifs-page></app-gifs-page>
  <!-- Fin del gifs-page -->
</div>
```

sidebar.template.html:

```
<div class="bg-dark border-right p-2" id="sidebar">
  <h3 class="text-light">Gifs-App</h3>
  <hr class="text-white" />
  <div class="list-group list-reset">
    <a href="#" class="list-group-item list-group-item-action"> Dashboard
  </a>
  </div>
</div>
```

gifs-page.template.html

```
<div class="container">
  <div class="row p-3">
```

```

<div class="col">
  <app-search></app-search>
</div>
</div>

<hr />
<div class="row">
  <div class="col">
    <app-results></app-results>
  </div>
</div>
</div>

```

search.template.html

```

<h5>Buscar :</h5>
<input type="text" class="form-control" placeholder="Buscar Gifs" />

```

results.template.html

Esto es un texto temporal

Por el momento solo nos hemos enfocado en trabajar con el HTML, el código a nivel de componentes.ts es el código por default del CLI.

Capturar el texto de búsqueda

Para efectos prácticos y dado que solo tenemos un input, no consideramos necesario en este punto usar el FormModule para implementar el two-data-binding, vamos a usar otra técnica: el **@ViewChild**

Tanto ViewChild como ViewChildren son dos decoradores que, utilizados sobre una propiedad de la clase que representa a un componente, permiten obtener las instancias de elementos nativos, directivas y componentes que estén en el template del mismo.

El primer cambio que haremos es en el componente:

```

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

@Component({
  selector: 'app-search',
  templateUrl: './search.component.html',
})
export class SearchComponent {
  @ViewChild('txtBuscar') txtBuscar: any;

  constructor() {}
}

```

```
buscar() {
  console.log(this.txtBuscar);
}
```

Como podemos observar, hemos usado el decorador `@ViewChild` para enlazar un elemento del DOM con una propiedad del Componente

```
@ViewChild('txtBuscar') txtBuscar:any;
```

Con eso indicamos que en el template, vamos a tener un elemento que es identificado por medio de `txtBuscar` y que vamos a enlazar con una propiedad del componente llamada `txtBuscar`.

Ademas agregamos un método que imprime en consola un texto.

En la plantilla agregamos dos cosas:

- 1- Un identificador interno al control input: `#txtBuscar`
- 2- Un evento (`keyup.enter`) que llama al método `buscar` con el texto del elemento referenciado `txtBuscar.value`

```
<input
  type="text"
  class="form-control"
  placeholder="Buscar Gifs..."
  (keyup.enter)="buscar(txtBuscar.value)"
  #txtBuscar
/>
```

Pero que imprime ese `console.log`? Imprime un objeto de tipo `ElementRef` y no el valor en si, porque lo que estamos enlazando es el Elemento Completo. Entonces podemos cambiar el `any` por este tipo de datos:

```
@ViewChild('txtBuscar') txtBuscar:ElementRef;
```

Pero ahora tenemos un error, Typescript nos exige que inicialicemos la propiedad, pero como en este caso sabemos que vamos a enlazar dicha propiedad a un elemento del DOM existente haremos uso del operador !

```
@ViewChild('txtBuscar') txtBuscar!:ElementRef;
```

Este operador se denomina **Operador de aserción no vacío ** Si el verificador de tipo no puede determinar si una variable es nula o no está definida durante el tiempo de ejecución, también arrojará un

error. Podemos saber por nosotros mismos que no estará vacío, pero el verificador de tipos no. Por lo tanto, tenemos que decirle al verificador de tipos que no será nulo y utilizaremos el operador de aserción no nulo.

Listo, intentemos imprimir ahora el valor, si nos vamos al código y escribimos

`console.log(this.txtBuscar.nativeElement);` vemos que luego de `nativeElement` no nos da permite listar más propiedades, para resolver esto tenemos que agregar el tipo de elemento al `ElementRef` ya que es de tipo genérico, en este caso como estamos trabajando con un elemento HTML, más específicamente el Input, entonces, finalmente la propiedad debe de quedar definida de la siguiente forma:

```
@ViewChild('txtBuscar') txtBuscar!: ElementRef<HTMLInputElement>;
```

Ahora si podemos decifrar esta línea como: **Estamos definiendo una propiedad denominada txtBuscar que es del tipo ElementRef y que estamos seguros que no es NULL (porque está definida en el DOM) y que el decorador @ViewChild creará un binding con un elemento en el DOM con un identificador intencionado #txtBuscar**

Y ahora si podemos buscar la propiedad `value` de esta forma y imprimir su valor-

```
this.txtBuscar.nativeElement.value;
```

En este punto tenemos acceso al control Input, y podríamos manipular dicho control según lo necesitemos, por ejemplo si luego de usar el valor queremos limpiarlo, entonces haremos esto:

```
buscar() {
  console.log(this.txtBuscar.nativeElement.value);
  this.txtBuscar.nativeElement.value = '';
}
```

Cambios en la APP.

Haremos algunos cambios importantes en la configuraciones de los componentes y sus plantillas, se hará una reorganización más limpia, al final el proyecto queda de esta forma:

Los directorios quedan de esta forma:

```
— app.component.css
|— app.component.html
|— app.component.spec.ts
|— app.component.ts
|— app.module.ts
|— app-routing.module.ts
|— gifs
|   |— components
```

```

    └── results
        ├── results.component.html
        └── results.component.ts
    └── search-box
        ├── search-box.component.html
        └── search-box.component.ts
    ├── gifs.module.ts
    ├── helpers
    ├── pages
    └── home
        ├── home-page.component.html
        └── home-page.component.ts
    └── services
    shared
        ├── components
        │   └── sidebar
        │       ├── sidebar.component.html
        │       └── sidebar.component.ts
        ├── helpers
        ├── pages
        ├── services
        └── shared.module.ts

```

La aplicación se ha dividido en dos secciones

- gifs
- shared

Gifs contiene los elementos relacionados con renderizar las búsqueda de gifs, componentes propiamente requeridos por la aplicación para buscar y mostrar los gifs, en cambio shared, son elementos que pueden ser compartidos por otros elementos o secciones dentro de la app, por ejemplo una barra lateral de menú, está podría mostrarse no solo para una página que renderiza los gifs, pero también para cualquier otra página que renderice otro tipo de información dentro de la App. Por ejemplo, supongamos que agregaremos un panel para permitir configurar las búsquedas de los gifs, podemos usar otro directorio gifSetUp y dentro de eso crear todos los elementos como componentes, helpers, servicios que muestren las herramientas para configuraciones. Entonces Shared contiene los elementos que se pueden compartir entre todos los componentes (vistas o secciones) de nuestra app

Dentro de cada sección crearemos una estructura de subdirectorios

components Aca se incluyen todos los componentes de la sección, para Gifs tendremos dos componentes, el SearchBox y el Result

helpers Usaremos este directorio para colocar cualquier rutina, función o método que proporcione cierta funcionalidad de ayuda para los demás elementos de esta sección

pages Las páginas son plantillas que agrupan a los componentes dentro de la sección

services Servicios que otorgan utilidad a la sección

Componentes principales

El template principal *app.component.html*

```
<div class="d-flex">
  <!-- Este es el sidebar -->
  <shared-sidebar></shared-sidebar>
  <!-- Fin del sidebar -->

  <!-- Este es el gifs-page component -->
  <gifs-home-page></gifs-home-page>
  <!-- Fin del gifs-page -->

</div>
```

El Componente, *app.component.ts*

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = '02-gifsApp';
}
```

El enrrutador: *app-routing.module.ts*

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

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

y finalmente el módulo

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

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

```

import { AppComponent } from './app.component';
import { SharedModule } from './shared/shared.module';
import { GifsModule } from './gifs/gifs.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    SharedModule,
    GifsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Componente Shared

El template

```

<div class="bg-dark border-right p-2" id="sidebar">
  <h3 class="text-light">Gifs-App</h3>
  <hr class="text-white">
  <div class="list-group list-reset">
    <a href="#" class="list-group-item list-group-item-action">
      DashBoard
    </a>
  </div>
</div>

```

El componente

```

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

@Component({
  selector: 'shared-sidebar',
  templateUrl: './sidebar.component.html'
})
export class SahredSidebarComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}

```

No tenemos por el momento helpers ni services ni páginas para esta sección, solamente el Modulo

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { SahredSidebarComponent } from
'./components/sidebar/sidebar.component';

@NgModule({
  declarations: [
    SahredSidebarComponent
  ],
  imports: [
    CommonModule
  ],
  exports: [
    SahredSidebarComponent
  ]
})
export class SharedModule { }
```

Sección Gifs

Cuenta con dos componentes y una página que los agrupa,

Componente search-box

el Template

```
<h5>Buscar:</h5>
<input
  type="text"
  class="form-control"
  placeholder="Buscar Gifs..."
  (keyup.enter)="buscar()"
  #txtBuscar>
```

El componente

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

@Component({
  selector: 'gifs-search-box',
  templateUrl: './search-box.component.html'
})
```

```
export class SearchBoxComponent {  
  
  @ViewChild('txtBuscar') txtBuscar!: ElementRef<HTMLInputElement>;  
  
  constructor() { }  
  
  buscar() {  
    console.log(this.txtBuscar.nativeElement.value);  
    this.txtBuscar.nativeElement.value = '';  
  }  
  
}
```

Ver más sobre el `@ViewChild` adelante en el documento.

Componente Result

El template

Por el momento solo tenemos un texto

```
 Lorem ipsum dolor sit amet consectetur, adipisicing elit. Cupiditate fugit  
molestias sapiente, laudantium quia officiis facere sed aut vero possimus,  
ullam ducimus praesentium minima nam qui debitis amet sequi laboriosam?
```

El componente

```
import { Component, OnInit } from '@angular/core';  
  
@Component({  
  selector: 'app-results',  
  templateUrl: './results.component.html'  
})  
export class ResultsComponent implements OnInit {  
  
  constructor() { }  
  
  ngOnInit(): void {  
  }  
  
}
```

Pages: Componente home-page

Este es un ejemplo de un componente que va en la carpeta `pages` es un componente que agrupa otros componentes y los presenta como una sola página.

el Template

```
<div class="container">
  <div class="row p-3">
    <div class="col-12">
      <gifs-search-box></gifs-search-box>
    </div>
  </div>

  <hr>
  <div class="row">
    <div class="col-12 pt-2">
      <app-results></app-results>
    </div>
  </div>
</div>
```

Vemos como se muestran los dos componentes principales de la sección

El Componente

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

@Component({
  selector: 'gifs-home-page',
  templateUrl: './home-page.component.html',
  styles: [
  ]
})
export class HomePageComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

El Modulo de Gifs

Este es el módulo de la sección Gifs

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HomeComponent } from './pages/home/home-page.component';
import { SearchBoxComponent } from './components/search-box/search-
box.component';
import { ResultsComponent } from './components/results/results.component';
```

```
@NgModule({
  declarations: [
    HomePageComponent,
    SearchBoxComponent,
    ResultsComponent
  ],
  imports: [
    CommonModule
  ],
  exports: [
    HomePageComponent,
  ]
})
export class GifsModule { }
```

Y con eso se completa el rediseño de los componentes.

@ViewChild

Documentación de internet:

En Angular, @ViewChild es un decorador que se utiliza para obtener una referencia a un elemento hijo dentro de un componente. Permite a un componente padre acceder a propiedades o métodos de un componente hijo directamente en el código TypeScript, en lugar de hacerlo a través del DOM.

La sintaxis básica de @ViewChild es la siguiente:

```
@ViewChild(TipoDeElemento) nombreDeVariable: TipoDeElemento;
```

Donde:

TipoDeElemento es el tipo de elemento que se está buscando. Puede ser un componente, una directiva, o un elemento nativo del DOM.
nombreDeVariable es el nombre de la variable que se utilizará para referenciar el elemento.

Aquí hay un ejemplo simple para ilustrar cómo se usa @ViewChild:

Supongamos que tienes un componente hijo llamado ChildComponent:

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

@Component({
  selector: 'app-child',
  template: <p>Este es el Child Component</p>
})
```

```

    template: '<p>Hola, soy un componente hijo.</p>',
  })
export class ChildComponent {
  // ... lógica del componente hijo, si es necesario
}

```

Y ahora, en el componente padre, puedes usar `@ViewChild` para obtener una referencia a ese componente hijo:

```

import { Component, ViewChild, AfterViewInit } from '@angular/core';
import { ChildComponent } from './child.component';

@Component({
  selector: 'app-parent',
  template: `
    <p>Componente Padre</p>
    <app-child></app-child>
  `,
})
export class ParentComponent implements AfterViewInit {
  @ViewChild(ChildComponent) childComponentRef!: ChildComponent;

  ngAfterViewInit() {
    // Ahora puedes acceder a las propiedades y métodos del componente hijo
    console.log(this.childComponentRef);
  }
}

```

En este ejemplo, `@ViewChild` se utiliza para obtener una referencia al componente hijo `ChildComponent` en el componente padre `ParentComponent`. Después del ciclo de vida `ngAfterViewInit`, la variable `childComponentRef` contendrá una referencia al componente hijo, y puedes interactuar con él directamente en el código del componente padre.

Fin de la documentación de internet.

Nuestro componente Search-Box usa este decorador, recordemos el template

```

<h5>Buscar :</h5>
<input
  type="text"
  class="form-control"
  placeholder="Buscar Gifs..."
  (keyup.enter)="buscar()"
  #txtBuscar>

```

El input tiene una referencia `#txtBuscar` la cual se usará como identificado único para enlazar el text con nuestro código en TypeScript.

En nuestro componente, definimos el enlace de esta forma

```
@ViewChild('txtBuscar') txtBuscar!: ElementRef<HTMLInputElement>;
```

@ViewChild acepta como parámetro el identificador único del componente, a la vez necesita un nombre para la propiedad txtBuscar! (puede ser nullable) y se define el tipo del elemento, en este caso es un HTMLInputElement

Luego, podemos acceder a nuestro elemento de la siguiente forma:

```
console.log(this.txtBuscar.nativeElement.value);
```

Servicio Gifs

Vamos a implementar un servicio, lo haremos desde la línea de comando

```
ng g service gifs/services/gifs
```

Esto crea un par de archivo, el más importante *src/app/gifs/services/gifs.service.ts*

Agregaremos una propiedad privada *_tagsHistory* para almacenar la lista de Tags con los que hemos realizado búsquedas.

```
private _tagHistory: string[] = [];
```

A la vez, crearemos un metodo get para obtener la lista de tags (Arreglo)

```
get tagHistory(): string[] {
    return [...this._tagHistory];
}
```

Dado que en Js los arreglos se pasan como referencia, y no queremos pasar esa referencia a cualquiera que lo solicite, usamos el operador spread ... para pasar una copia del arreglo en lugar de la referencia.

Procedemos a crear un nuevo método *searchTag()* que recibe como parametro el tag de búsqueda y agregara el TAG a nuestro tags history, de paso mantendremos unicamente 10 elementos en nuestro arreglo.

El servicio por el momento es el siguiente:

```

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

@Injectable({
  providedIn: 'root'
})
export class GifsService {

  private _tagHistory: string[] = [];

  constructor() { }

  get tagHistory(): string[] {
    return [...this._tagHistory];
  }

  public searchGifs(tag: string): void {
    tag = tag.trim().toLowerCase();
    if (!this._tagHistory.includes(tag)) {
      this._tagHistory.unshift(tag);
      this._tagHistory = this._tagHistory.splice(0, 10);
    }
  }
}

```

Algunas cosas a notar:

Primero, el decorador `@Injectable` por medio del configurador `providedIn: 'root'` indicamos que cualquier módulo que importe este servicio, automáticamente tiene a su disposición usar dicho servicio.

De Internet:

Cuando se usa `provideIn: 'root'` en tu servicio, le indicamos a Angular que proporcione el servicio en el inyector raíz. El inyector raíz es el inyector superior en su aplicación del cual heredan todos los demás inyectores. Esto significa que el servicio estará disponible en toda la aplicación. Es una forma de crear un servicio singleton, es decir, de garantizar que una clase de servicio tenga solo una instancia en toda la aplicación.

Inyectando el nuevo servicio

Regresemos a nuestro search-box Component, en el constructor debemos inyectar el nuevo servicio de la siguiente forma:

```
constructor(private gifsService: GifsService) { }
```

Dado que el servicio fue agregado al Inyector Raiz, no es necesario hacer Importaciones o exportaciones en los módulos, simplemente lo inyectamos.

Seguidamente podemos usar los métodos del servicio:

```
const newTag = this.txtBuscar.nativeElement.value.trim().toLowerCase();
this.gifsService.searchGifs(newTag);
```

Consumiendo el servicios desde el SideBar

Vamos a imprimir la lista de elementos que tenemos en nuestro `_tagHistory` en el SideBar Component, para ello primero inyectamos el servicio en el componente y luego creamos un metodo publico GET para retornar la lista actual de elementos guardados.

```
export class SahredSidebarComponent {

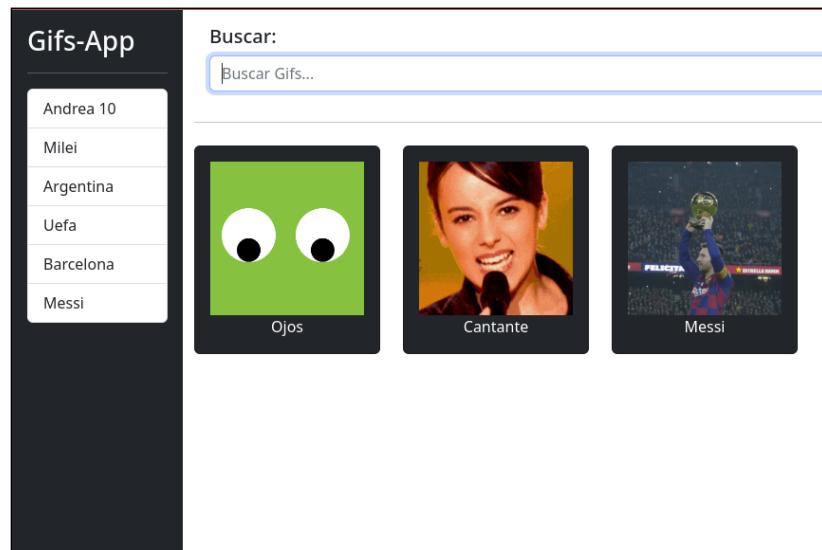
  constructor(private gifsService: GifsService) { }

  public get tags(): string[] {
    return this.gifsService.tagHistory;
  }
}
```

Seguidamente podemos usar el método Get en nuestro template, y con esto cada vez que agregamos un TAG se mostrará en la Barra lateral.

```
<button *ngFor="let tag of tags" class="list-group-item list-group-item-action">
  {{ tag | titlecase}}
</button>
```

El resultado: Veremos como un botón los elementos guardados.



NOTA: Angular está constantemente observando los cambios en los datos y actualizando la vista en consecuencia. Cuando el array `_tagHistory` en el servicio cambia (debido a la adición de un nuevo tag),

Angular detecta este cambio. Angular actualiza automáticamente todas las referencias en las vistas que dependen de estos datos.

Giphy API key

Vamos a usar un API publico para relizar busquedas de Gifs

Primero creamos una cuenta en <https://developers.giphy.com/> luego en el Dashboard (<https://developers.giphy.com/dashboard/>) creamos una nueva APP y seleccionamos el API. Esto genera un API key, el cual vamos a usar en nuestra app.

Por el momento usaremos una constante en el servicio para almacenar el API Key, mas adelante usaremos variables de Entorno.

```
/**
 * The GIPHY API key.
 * TODO: Move to environment variables.
 */
const GIPPHY_API_KEY = '*****';
```

```
@Injectable({
  providedIn: 'root'
})
export class GifsService {}
```

Para probar el API podemos instalar la extensión Rest Client, creamos un archivo con extensión http en nuestra carpeta root del proyecto y agregamos request como este:

```
GET https://api.giphy.com/v1/gifs/search?
api_key=KTsjcCoEj7zWZC0vEgmQDtFd6Fegr8ta&q=funny+cat
```

Luego Ctrl + Click en el URL.

Obtener Datos

Podemos implementar un Fetch para realizar búsquedas usando el API,

```
fetch(`${GIPHY_API_URL}/search?
api_key=${GIPPHY_API_KEY}&q=${tag}&limit=10`)
  .then( response => response.json())
  .then( response => {
    console.log(response.data);
  });
}
```

Pero existe otra forma en angular que permite implementar interceptors, y realizar otras operaciones con la llamada al API, usaremos HttpClientModule

HttpClientModule

HttpClientModule permite nativamente desde angular implementar, entre otras cosas, llamadas a API en las que podamos aplicar interceptor, programación reactiva, etc. Es un client muy poderoso y se recomienda usarlo en Angular.

Primero en el `app.module.ts` importamos el paquete:

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

@NgModule({
  imports: [
    HttpClientModule
  ]
})
export class AppModule { }
```

Y luego en nuestro servicio lo inyectamos en el constructor:

```
constructor(private http: HttpClient) { }
```

Y luego implementamos el GET de la siguiente forma:

```
this.http.get(` ${GIPHY_API_URL}/search?
api_key=${GIPHY_API_KEY}&q=${tag}&limit=10 `).subscribe( (response: any) =>
{
  console.log(response.data);
});
```

El `this.http.get` regresa un `Observable` el cual es un objeto que a lo largo del tiempo puede estar emitiendo diferentes valores

Un objeto `Observable` permite subscribirse por medio del método `subscribe` eso significa que vamos a estar escuchando las emisiones de ese objeto. Cada vez que emita un valor, vamos a escucharlo y reaccionar a dicho evento. En nuestro caso, es un objeto que emite eventos una sola vez pero igual, debemos de suscribirnos. Esto quiere decir que hay otros Observables que emiten eventos constantemente en el tiempo.

HttpParams

Continuamos optimizando nuestro código, esta vez lo haremos con los parámetros de la petición. Vamos a implementar un objeto `HttpParams` (propio de JS, no necesitamos importar nada) y agregaremos nuestros parámetros a dicho objeto, posteriormente lo agregaremos a la petición, el código queda de esta forma:

```

const params = new HttpParams()
  .set('api_key', GIPHY_API_KEY)
  .set('q', tag)
  .set('limit', GIPHY_SEARCH_LIMIT.toString());

this.http.get(` ${GIPHY_API_URL}/search`, { params }).subscribe(
  (response: any) => {
    console.log(response);
  });

```

NOTA: `GIPHY_SEARCH_LIMIT` es una constante agregada en el servicio.

Implementando Tipo de Datos para el API response

El API retorna un objeto con muchos datos, mapear eso a las interfaces manualmente puede ser una tarea complicada, pero podemos usar herramientas que transforman un JSON en las interfaces necesarios, por ejemplo podemos usar <https://app.quicktype.io/>

Pegamos el JSON, seleccionamos el tipo de lenguaje (typescript) y marcamos Solo interfaces, cambiamos el nombre de la interfaz principal a `SearchResponse`, creamos un nuevo archivo:

`src/app/gifs/interfaces/gifs.interface.ts` y agregamos todas las interfaces creadas por quicktype.

La interfaz principal `SearchResponse` cuenta con tres campos

```

export interface SearchResponse {
  data: Gifs[];
  pagination: Pagination;
  meta: Meta;
}

```

`Data` es un arreglo de `gifs`, y es el objeto que contiene los datos mas relevantes. Los almacenaremos en una variable pública `public gifsList: Gifs[] = [];`

NOTA: en este caso usamos una propiedad pública y no una privada con su Get (usando desestructuración) porque esta lista es volátil, cada vez que hacemos una búsqueda, esta lista va a ser reemplazada. Por eso no tiene mucha importancia si un proceso externo al servicio la modifica. Aunque lo ideal es siempre usare el metodo GET con su propiedad privada tal como se hizo con el `private _tagHistory: string[] = [];`

A continuación, vamos a tipar nuestro response, el código siguiente imprime únicamente el título de los gifs en la respuesta:

```

this.http.get<SearchResponse>(` ${GIPHY_API_URL}/search`, { params })
  .subscribe( (response) => {
    this.gifsList = response.data;
  });

```

Mostrar Gifs en Pantalla

Tenemos esta estructura de componentes:

```
app_root
  |
  |-shared-sidebar
  |
  |-gifs-home-page
    |
    |-gifs-search-box
    |
    |-gifs-card-list
```

Tenemos dos opciones:

1. Inyectar el servicio directamente en el *gifs-card-list*
2. Inyectar el servicio en el *gifs-home-page* y que sea este el que pase los gifs al componente hijo *gifs-card-list*

Con la opción 1, inyectamos directamente el servicio al card list, de modo que únicamente este componente podrá usar la información de los gifs.

La ventaja del item 2 es que podemos posteriormente agregar otro componente al home page que use la información de los gifs, por ejemplo una sección de resumen.

Implementaremos la opción #2.

Inyectamos el servicio y creamos un método *get*

```
export class HomePageComponent {

  constructor(private gifsService: GifsService) { }

  get gifs(): Gifs[] {
    return this.gifsService.gifsList;
  }
}
```

Ahora necesitamos pasar esa información al componente hijo *gifs-card-list* y lo haremos por medio de un *@Input*. En el componente Card list agregamos este input:

```
export class CardListComponent {

  @Input() gifs: Gif[] = [];
```

```
constructor() { }

}
```

Nuestro componente hijo está listo para recibir la lista de Gifs desde el padre, para pasar esa información, modificaremos el template del componente *gifs-home-page* y al momento de definir el componente *gifs-card-list* pasamos el dato como un parámetro

```
<div class="row">
  <div class="col-12 pt-2">
    <gifs-card-list [gifs]="gifs"></gifs-card-list>
  </div>
</div>
```

En este momento el card list puede renderizar en su template la lista de gifs.

Actualizar el SideBar

En nuestro sideBar Component, ya tenemos el servicio injectado en el constructor, lo usamos para crear la lista de tags (últimos 10 tags buscados), vamos a agregar un método que llame nuevamente el servicio para buscar por un tag específico, en este caso, al hacer click en el botón

```
export class SahredSidebarComponent {

  constructor(private gifsService: GifsService) { }

  public get tags(): string[] {
    return this.gifsService.tagHistory;
  }

  public searchTag(tag: string): void {
    this.gifsService.searchGifs(tag);
  }
}
```

Ahora, en nuestro template, usaremos el método *searchTag*

```
<button
  (click) = "searchTag(tag)"
  *ngFor="let tag of tags"
  class="list-group-item list-group-item-action">
  {{ tag | titlecase}}
</button>
```

Con este cambio, al hacer click en el botón, se dispara nuevamente una búsqueda con el Tag seleccionado.

Veremos como se actualiza el cardListComponent sin hacer mayor cambio, porque la llamada al método `this.gifsService.searchGifs(tag)`; actualizará la propiedad `this.gifsList = response.data`; esto hace que en cualquier sitio de nuestro código donde hemos usado esta propiedad, se refresquen los componentes asociados, por ejemplo, en el homePageComponente hicimos esto

```
get gifs(): Gif[] {
    return this.gifsService.gifsList;
}
```

El método gifs se usa en la renderización del cardList

```
<gifs-card-list [gifs]="gifs"></gifs-card-list>
```

Y esto hace que el componente actualice su información en Pantalla.

LocalStorage

La lisa de TAGS y sus resultados actualmente se pierde cuando recargamos el navegador, vamos a implementar un localStorage, para ello podemos guardar la información en el browser usando localStorage, sessionStorage o las Cookies.

El Localstorage permite un mayor cantidad de datos de almacenamiento y es persistente aunque apaguemos el computador.

El SessionStorage también permite guardar grandes cantidades de datos pero se pierde cuando cerramos el navegador.

El cookies tambien persisten datos pero a diferencia de los métodos anteriores, el contenido es enviado al backend cuando interactuamos con el.

Implementemos un metodo sencillo en el GigsService

```
/**
 * Save the tag history to the local storage.
 */
private saveTagHistory(): void {
    localStorage.setItem('history', JSON.stringify(this._tagHistory));
}
```

Este método debemos llamarlo tan pronto el servicio actualice la lista de Tags.

Cargar datos guardados

Ahora es momento de cargar los datos guardados en el localStorage la carga nuestra App.

Creamos el método correspondiente para cargar el history desde el localStorage

```
/**  
 * Load the tag history from the local storage.  
 */  
private loadTagHistory(): void {  
  const history = localStorage.getItem('history');  
  if (history) {  
    this._tagHistory = JSON.parse(history);  
    this.searchGifs(this._tagHistory[0]);  
  }  
}
```

Necesitamos llamar este método en algún lugar, y lo haremos cuando inyectemos el servicio por primera vez, es decir en el constructor del servicio.

```
export class GifsService {  
  constructor(private http: HttpClient) {  
    this.loadTagHistory();  
  }  
}
```

NOTA: al recargar el browser, si tenemos tags almacenados, veremos que la app siempre carga los gifs del primer TAG, esto es posible porque hemos agregado la siguiente línea de código en nuestro loadTagHistory()

```
this.searchGifs(this._tagHistory[0]);
```

Y con esto ya podemos persistir en el localStorage la lista de TAGS.

Tarea Adicional 1

Implementaremos una funcionalidad que elimine un Tag específico, para ello agregaremos un ícono de eliminar a la par del TAG en el sideBarComponent y al presionar, eliminaremos ese TAG de la lista.

Tarea Adicional 2

Permitir cargar GIFS desde el localStorage, cuando se cargue desde el localStorage, agregar un ícono a la par del TAG que indique que ha sido cargado desde Memoria.

Solución ambas tareas

Usando mi cuenta personal, se obtiene un kit the Fontawesome y se agrega en el index.html

```
<script src="https://kit.fontawesome.com/8de2637909.js"
crossorigin="anonymous"></script>
```

Se agrega el método eliminar TAG en el *GifsService*:

```
/**
 * Delete a tag from the tag history.
 * @param tag The tag.
 */
public deleteTag(tag: string): void {
    this._tagHistory = this._tagHistory.filter( oldTag => OldTag !== tag);
    this.saveTagHistory();
}
```

Se agrega el método de eliminar en el *sideBarComponent*

```
public deleteTag(tag: string): void {
    this.gifsService.deleteTag(tag);
}
```

Se agregan los iconos al template del *sideBarComponent*

```
<div *ngFor="let tag of tags">
    <div class="d-flex justify-content-between align-items-center">
        <button
            (click)="searchTag(tag)"
            class="list-group-item list-group-item-action">
            {{ tag | titlecase }}
        </button>
        <span (click)="deleteTag(tag)" class="sb-icon text-white">
            <i class="fa-solid fa-trash-can"></i>
        </span>
        <span (click)="searchTag(tag)" class="sb-icon text-white">
            <i class="fa-solid fa-rotate"></i>
        </span>
    </div>
</div>
```

Nueva Sección - Image Loader

¿Qué veremos en esta sección?

El objetivo principal de la sección es poder aprender a realizar un loading mientras la imagen se carga, pero indirectamente haremos:

- @Inputs
- @Outputs
- Custom Components
- Validaciones
- Importación de módulos personalizados

Crear un nuevo componente

Antes de comenzar con esta sección, vamos a crear un componente para mostrar un Gif. Actualmente tenemos un *gif-card-list* y este, recorre la lista de gifs[] y renderiza los gifs en una especie de cartilla. Pues crearemos un componente que renderiza un gif específico (recibirá como @Input el gif) y dicho componente lo llamaremos en el *gif-card-list*

Usando el ng Client:

```
ng g c gif-card
```

En el template del nuevo componente moveremos el código html que teníamos en el ciclo del *gifCardListComponent*

```
<div class="card mb-2 text-center bg-dark">
  <div class="card-body text-white">
    <img [src]="gif.images.downsized_medium.url" [alt]="gif.title"
    class="card-img-top">
      <p class="card-text">
        {{ gif.title }}
      </p>
    </div>
  </div>
```

El código de typescript para este nuevo componente deberá:

- 1- Aceptar un gif como parámetro de entrada
- 2- Validar que le gif ha sido proporcionado, osea, que no sea null.

```

@Component({
  selector: 'gifs-gif-card',
  templateUrl: './gif-card.component.html',
  styleUrls: ['./gif-card.component.css']
})
export class GifCardComponent implements OnInit {

  constructor() { }

  @Input() gif!: Gif;

  ngOnInit(): void {
    if (!this.gif) {
      throw new Error('GifCardComponent: gif is required');
    }
  }
}

```

Ahora necesitamos cambiar el template del *gifCardListComponent* para llamar este componente y proporcionar el gif específico a renderizar

```

<div class="row">
  <gifs-gif-card
    [gif]="gif"
    *ngFor="let gif of gifs"
    class="col-md-2 col-sm-6">
  </gifs-gif-card>
</div>

```

Y listo, tenemos nuevamente la aplicación funcionando como antes, solo que esta vez, tenemos un nuevo componente para renderizar un único gif. Esto nos ayudara a reutilizar este componente, supongamos que al hacer click abrimos una nueva sección, y parte de esa sección es mostrar el gif seleccionado en un card, con este componente podremos hacer eso facilmente.

Lazy Image

Implementaremos la carga perezosa de imágenes por medio de un componente, dado que podemos reutilizar esta funcionalidad con cualquier imagen, vamos a colocar el componente en nuestro Shared.

Shared puede ser luego copiado íntegramente a otros proyecto, de modo que todos estos componentes estarán disponibles para su uso inmediato en otra aplicación.

Primero agrearemos el componente:

```

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

@Component({
  selector: 'shared-lazy-image',

```

```

    templateUrl: './lazy-image.component.html',
    styleUrls: ['./lazy-image.component.css'],
})
export class LazyImageComponent {

}

```

Luego lo declaramos y exportamos en el *SharedModule*

```

@NgModule({
  declarations: [
    SahredSidebarComponent,
    LazyImageComponent
  ],
  imports: [
    CommonModule
  ],
  exports: [
    SahredSidebarComponent,
    LazyImageComponent
  ]
})
export class SharedModule { }

```

Hasta aca, el componente puede ser usado por todo aquel módulo que importe el *SharedModule*

Ahora, importamos el módulo donde queremos usarlo, en el *GifsModule* Solamente lo incluimos en la sección *imports*

```

@NgModule({
  imports: [
    SharedModule
  ]
})
export class GifsModule { }

```

Con este cambio ya podemos usar el *LazyImageComponent* dentro de nuestro *GifsModule*

Ahora implementaremos la funcionalidad del *LazyImageComponent* agregaremos una propiedad *@Input src* que será el URL del recurso a cargar (Imagen o Gif), validaremos que el *OnInit* que el URL ha sido proporcionado. Y tambien una propiedad nullable para mostrar el ALT Text. Tambien implementaremos un flag, *hasLoaded*, para determinar cuando ha sido cargada la imagen. El objetivo es que mientras la imagen no ha sido cargada, mostremos el Spinner, una vez cargada ocultamos el spinner y se muestra la imagen.

El componente será:

```

@Component({
  selector: 'shared-lazy-image',
  templateUrl: './lazy-image.component.html',
  styleUrls: ['./lazy-image.component.css'],
})
export class LazyImageComponent implements OnInit {

  constructor() { }

  @Input() src!: string;
  @Input() alt: string = '';

  public hasLoaded: boolean = false;

  ngOnInit(): void {
    if (!this.src) {
      throw new Error('Attribute src is required');
    }
  }

  onLoad() {
    this.hasLoaded = true;
  }
}

```

El template de nuestro LazyLoader será

```

<div class="d-flex justify-content-center">
  <img
    [src]="src"
    [alt]="alt"
    class="card-img-top"
    (load)="onLoad()"
    [ngStyle]="{ display: hasLoaded ? '' : 'none' }">

  
</div>

```

Inicialmente el componente tiene el flag *hasLoaded* en *false* este es el estado inicial y mostrará el spinner.

En la primer imagen, tenemos un *(load)="onLoad()* esto significa que una vez cargada la imagen, se llama el método *onLoad()*

El método *onLoad()* cambia el flag *hasLoaded* a *true*

A continuación esto oculta el spinner `*ngIf="!hasLoaded"` y muestra la imagen `[ngStyle]={`${display: hasLoaded ? 'none'}`}`

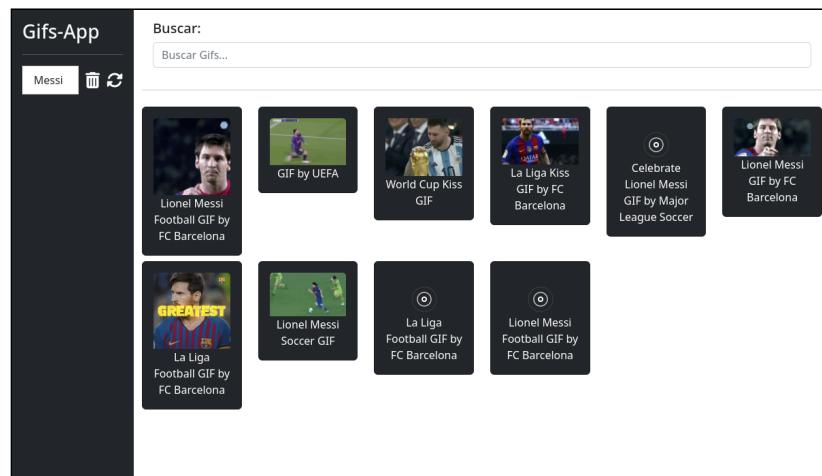
Ahora implementemos el `LazyImageComponent` en nuestro template del `gif-card-component`, el cual luce de esta forma antes de este cambio:

```
<div class="card mb-2 text-center bg-dark">
    <div class="card-body text-white">
        <img [src]="gif.images.downsized_medium.url" [alt]="gif.title"
    class="card-img-top">
        <p class="card-text">
            {{ gif.title || 'No title' }}
        </p>
    </div>
</div>
```

En ese template, vamos a implementar nuestro `shared-lazy-image` es decir reemplazamos el `img` por esta línea:

```
<shared-lazy-image [src]="gif.images.downsized_medium.url"
[alt]="gif.title"></shared-lazy-image>
```

Y listo, al cargar la página vemos un spinner y cuando la imagen ha sido cargada es mostrada.



Animaciones con CSS

Sitio: <https://animate.style/>

Colocamos en nuestro index.html

```
<link
    rel="stylesheet"
    href="https://cdnjs.cloudflare.com/ajax/libs/animate.css/4.1.1/animate.min.
```

```
css"  
/>>
```

Y luego podemos usar las classes para agregar efectos, por ejemplo, agregar un FadeIn de 400ms cuando se carga la imagen:

animate_animated animate_fadeIn animate_delay-400ms

```
<img  
  [src]="src"  
  [alt]="alt"  
  class="card-img-top animate_animated animate_fadeIn  
animate_delay-400ms"  
  (load)="onLoad()"  
  [ngStyle]="{ display: hasLoaded ? '' : 'none' }">
```

Nueva Sección: SPA

Single Page Application

¿Qué veremos en esta sección?

Este es un breve listado de los temas fundamentales:

Rutas

RouterLink y RouterLinkActive

Componentes especializados

DebounceTime

Inputs

SwitchMaps

Consumo de APIs

Tipado de datos

Menú de aplicación

Y más

En esta sección daremos los fundamentos de una aplicación SPA (Single Page Application), y luego, las seguiremos haciendo pero mediante carga perezosa (lazyload), esta es otra aplicación real que nos enseñara mucho de cómo funciona Angular y como la información fluye en ella.

Muestra de la APP:

Países

Por Capital

#	Icon	Bandera	Nombre	Capital	Población	Info
1			El Salvador	San Salvador	6,486,201	Info
2			Costa Rica	San José	5,094,114	Info
3			Dominican Republic	Santo Domingo	10,847,904	Info
4			San Marino	City of San Marino	33,938	Info
5			Puerto Rico	San Juan	3,194,034	Info
6			Chile	Santiago	19,116,209	Info
7			Yemen	Sana'a	29,825,968	Info

Mostrando información de un País en específico:

Países

País: Costa Rica

Bandera	Información
	Población: 5,094,114 Código: CRI

Traducciones

Costa Rica Kostarika Costa Rica Costa Rica Costa Rica Costa Rica Costa Rica Costa Rica
 코스타리카 كاستاريكا

Crear nuevo proyecto:

Configuraciones iniciales:

```
$ ng new 03-countryApp --standalone=false  
? Which stylesheet format would you like to use? CSS  
? Do you want to enable Server-Side Rendering (SSR) and Static Site  
Generation (SSG/Prerendering)? No
```

Agregar Bootstrap, Animate y fontawesome

```
<!-- CSS only -->  
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-  
beta1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-  
0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6D0itfPri4tjfHxaWutUpFmBp4vmVor"  
crossorigin="anonymous">  
  <link  
    rel="stylesheet"  
  
  href="https://cdnjs.cloudflare.com/ajax/libs/animate.css/4.1.1/animate.min.  
css"  
  />  
  
  <script src="https://kit.fontawesome.com/8de2637909.js"  
crossorigin="anonymous"></script>
```

Agregar la Clase Container al body

```
<body class="container">
```

Borramos el Template del APP y listo, podemos iniciar.

Modulos

Vamos a crear un SPA (Single Page Application) y nos vamos a basar en dos módulos con varias páginas cada uno. Por el momento creamos ambos módulos:

```
$ ng g m countries  
CREATE src/app/countries/countries.module.ts (195 bytes)  
$ ng g m shared  
CREATE src/app/shared/shared.module.ts (192 bytes)
```

La carga del *countries.module* se hará de modo LazyLoading, es decir, no se va a cargar inicialmente, en cambio el módulo *shared.module* se será cargado desde un inicio, por lo tanto, agregaremos el

shared.module en la sección de *imports* de nuestro *app.module*

Router Module y Páginas SPA

En la carpeta *app/shared* crearemos dos directorios: *components* y *pages* y dentro de *pages* vamos a crear dos componentes: *shared-home-page* y *shared-about-page*

```
[fcruz@fedora app]$ tree
.
└── app
    ├── countries
    │   └── countries.module.ts
    └── shared
        ├── components
        └── pages
            ├── aboutPage
            │   ├── aboutPage.component.css
            │   ├── aboutPage.component.html
            │   └── aboutPage.component.ts
            └── HomePage
                ├── HomePage.component.css
                ├── HomePage.component.html
                └── HomePage.component.ts
        └── shared.module.ts
```

LA idea es implementar un Router que permita cargar uno de los dos componentes dependiendo del URL

/home cargará el *HomePageComponent* y */about* el *aboutPageComponent*

Archivo: *src/app/app-routing.module.ts*

```
const routes: Routes = [
  {
    path: 'home',
    component: HomePageComponent
  },
  {
    path: 'about',
    component: AboutPageComponent
  },
  {
    path: '***',
    redirectTo: 'home'
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
```

```

    })
export class AppRoutingModule { }

```

Analicemos el Router, se crea un arreglo de tipo *Routes*, este objetos {} el cual, por el momento contiene un *path* el URL section, y un *component*, el que se debe cargar si el URL hace match.

Luego, como este es el Routing principal usaremos *forRoot*, en el import: *RouterModule.forRoot(routes)*

Finalmente exportamos el módulo ya que luego debemos importarlo en el App.module.

Y luego, en el template principal *src/app/app.component.html* agregamos un *router-outlet* el cual actúa como un marcador de posición que Angular llena dinámicamente según el estado actual del enrutador.

```

<div class="col">
  <router-outlet></router-outlet>
</div>

```

Listo, si cargamos el URL *http://localhost:4200/home* mostrará el *homeComponent* y si por el contrario cargamos el URL *http://localhost:4200/about* entonces mostrará el *aboutComponent*

Router Link

Construiremos una barra de navegación para que al hacer click podamos cambiar entre componentes (home o about) sin necesidad de recargar el URL.

Crearemos un *SidebarComponent* en la carpeta *Shared* que servirá de barra de navegación, el template es el siguiente:

```

<h2>Países</h2>
<hr>

<ul class="list-group">
  <li
    routerLink="home"
    routerLinkActive="active"
    class="list-group-item">
    Home
  </li>
  <li
    routerLink="about"
    routerLinkActive="active"
    class="list-group-item">
    About
  </li>
</ul>

```

Notar que cada elemento del menú /i tiene una propiedad [routerLink] con un valor del *path* que agregamos anteriormente a nuestro Router. Pero para que Angular reconozca el [routerLink] como una

propiedad de elemento `/i` debemos importar el `AppRoutingModule` en nuestro `SharedModule`. Si no importamos el Routing Module, nos dará un error.

Con respecto a `routerLinkActive` lo que permite es agregar la clase `active` al elemento seleccionado, se mostrará en azul.

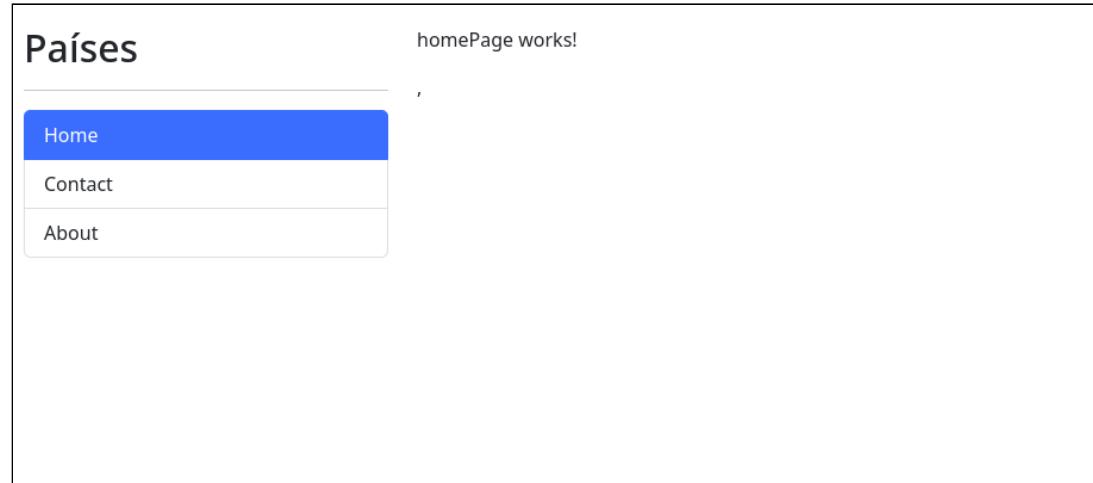
Con estos cambios, al hacer click en cada elementos, veremos como se carga el componente indicado.

NOTA: También podemos usar el `RouterLink` de la siguiente forma

```
<li
  routerLink="home"
  class="list-group-item">
  Home
</li>
```

La versión con los `//` permite asignar dinámicamente los path a cada elemento, la segunda forma es mas estática.

Aplicamos los mismos pasos para agregar un `ContactPageComponent` y la aplicación por el momento debe de verse asi



Crear estructura de Countries

Vamos a crear la estructura de directorios y a la vez agregaremos componentes y muy importante, Crearemos un `CountriesRoutingModule`, es decir un sistema de rutas específico para nuestra sección de Countries. Este será agregado como un `ChildRoutingModule`, porque la APP solo permite tener un `RootRoutingModule` que ya lo hemos creado.

Comenzamos creando algunas carpetas `components`, `services`, `pages`, `interfaces` y agregamos componentes:

```
$ ng g c countries/pages/byCapitalPage --inline-style --skip-tests
CREATE src/app/countries/pages/by-capital-page/by-capital-
page.component.html (30 bytes)
CREATE src/app/countries/pages/by-capital-page/by-capital-page.component.ts
```

```
(200 bytes)
UPDATE src/app/countries/countries.module.ts (317 bytes)
```

De la misma forma crearemos los siguientes componentes:

```
$ ng g c countries/pages/byCountryPage --inline-style --skip-tests
$ ng g c countries/pages/byRegionPage --inline-style --skip-tests
$ ng g c countries/pages/countryPage --inline-style --skip-tests
```

NOTA: Todos los componentes se crean con el prefijo *app* en el Selector, cambiemos ese prefijo, en este caso por *countries*

```
selector: 'app-country-page',
```

Cambiar por

```
selector: 'countries-country-page',
```

Repetimos el cambio para los componentes que acabamos de crear.

Crear sistema de rutas para Countries

Como se ha mencionado, vamos a crear un módulo de rutas específico para la sección *Countries* este contendrá las rutas apuntando a las páginas que acabamos de crear.

```
const routes: Routes = [
  {
    path: 'by-capital',
    component: ByCapitalPageComponent
  },
  {
    path: 'by-country',
    component: ByCountryPageComponent
  },
  {
    path: 'by-region',
    component: ByRegionPageComponent
  },
  {
    path: 'by/:id',
    component: CountryPageComponent
  },
];
@NgModule({
```

```

    imports: [
      RouterModule.forChild(routes)
    ],
    exports: []
  })
export class CountriesRoutingModule { }

```

Luego tenemos que agregar en la lista de *imports* del *CountriesModule*

```

@NgModule({
  declarations: [
    ByCapitalPageComponent,
    ByCountryPageComponent,
    ByRegionPageComponent,
    CountryPageComponent
  ],
  imports: [
    CommonModule,
    CountriesRoutingModule,
  ]
})
export class CountriesModule { }

```

La estructura final de Countries luce de esta forma:

```

.
├── components
├── countires-routing.module.ts
└── countries.module.ts
├── interfaces
└── pages
  ├── by-capital-page
  │   ├── by-capital-page.component.html
  │   └── by-capital-page.component.ts
  ├── by-country-page
  │   ├── by-country-page.component.html
  │   └── by-country-page.component.ts
  ├── by-region-page
  │   ├── by-region-page.component.html
  │   └── by-region-page.component.ts
  └── country-page
      ├── country-page.component.html
      └── country-page.component.ts
└── services

```

Hasta el momento, las rutas no están siendo "comprendidas" por la APP porque no hemos usado el *countries.module.ts* quien tiene todas las declaraciones, imports y exports. Lo veremos en la próxima

sección, ya que vamos a implementar una carga Lazy. Es decir, únicamente cuando el usuario lo demande.

LazyLoad

No vamos a agregar el *CountriesRoutingModule* en el *AppModule* porque eso significa cargar el módulo completo desde que iniciamos la APP, lo que buscamos es cargar por demanda el Módulo. Para ello primero necesitamos un punto de entrada, es decir una ruta que nos permita cargar el Countries. Agregamos una nueva ruta en nuestro Root.

```
const routes: Routes = [
  {
    path: 'home',
    component: HomePageComponent
  },
  {
    path: 'about',
    component: AboutPageComponent
  },
  {
    path: 'contact',
    component: ContactPageComponent
  },
  {
    path: 'countries',
    loadChildren: () => import('./countries/countries.module').then(m =>
      m.CountriesModule)
  },
  {
    path: '**',
    redirectTo: 'home'
  }
];
```

Notar como el path de *countries* no carga un componente directamente, carga un módulo usando *loadChildren* el cual es un objeto que especifica rutas secundarias con carga diferida.

Por el momento, si cargamos el url *http://localhost:4200/countries* notaremos que no se carga ningun componente, únicamente el sideBar

Países

Home
Contact
About



Si cargamos los URL hijos:

<http://localhost:4200/countries/by-country>

Países

by-country-page works!

Home
Contact
About

Ahora, podemos usar esta configuración para agregar más opciones al menú, por ejemplo, en nuestro sidebar podemos agregar las búsquedas por capital, region, país

```
<li
  routerLink="countries/by-capital"
  routerLinkActive="active"
  class="list-group-item">
  By Capital
</li>
<li
  routerLink="countries/by-region"
  routerLinkActive="active"
  class="list-group-item">
  By Region
</li>
<li
  routerLink="countries/by-country"
  routerLinkActive="active"
  class="list-group-item">
```

```
    By Country
</li>
```

Ahora podemos ver las opciones en el SideBar.

NOTA: Las opciones del HOME | CONTACT | ABOUT pueden ser eliminadas, solamente fueron usadas para propósito de explicar el Routing. La Aplicación únicamente mostrará las opciones para filtrar por CAPITAL | REGION | PAIS

The screenshot shows a web application titled "Países". On the left, there is a sidebar with the following menu items:

- Home
- By Capital
- By Region** (highlighted with a blue background)
- By Country
- Contact
- About

To the right of the sidebar, the main content area displays the text "by-region-page works!".

Build y Lazy Files

Los cambios realizados permiten cargar de manera perezosa el módulo *CountriesModule* de modo que hasta que hacemos click en uno de los links nuevos, el browser solicitará un archivo adicional (chunk) el cual contiene el código JS del módulo *CountriesModule*

Esto lo podemos ver también en el *build*, podemos notar que existe una sección *Lazy Chunk Files*, este se carga tan pronto hacemos click por primera vez en uno de los links nuevo.

Initial Chunk Files	Names	Raw Size	Estimated Transfer
Initial Chunk Files			
Size			
chunk-RZ5F67RC.js	-	200.00 kB	53.86
kB			
polyfills-LZBJRJJE.js	polyfills	32.69 kB	10.59
kB			
main-AE5F073B.js	main	2.90 kB	820
bytes			
styles-5INURTS0.css	styles	0 bytes	0
bytes			
	Initial Total	235.59 kB	65.25
kB			
Lazy Chunk Files			
Size			

```
chunk-60YRYQCF.js      | countries-module | 1.49 kB | 363
bytes

Application bundle generation complete. [3.172 seconds]
```

Cambios en RedirectTo

Haremos un cambio adicional con las rutas, actualmente al cargar la aplicación despliega el URL `http://localhost:4200/home` y muestra el `HomePageComponent`. Queremos que por defecto cargue el `byCapital`

Lo único que necesitamos cambiar es el `redirectTo` del comodín `**` en nuestro Root Router, en lugar de `'home'` usar el `'countries/by-capital'`

```
{
  path: '**',
  redirectTo: 'countries/by-capital'
}
```

Otra forma de hacerlo y quizás mejor es definir rutas por default en cada router, en este caso en el router ROOT podemos definir únicamente el `redirectTo` a `'countries'`

```
{
  path: '**',
  redirectTo: 'countries'
}
```

Y, en el `CountriesRoutingModule` definimos también un default path

```
{
  path: '**',
  redirectTo: 'by-capital'
}
```

Y recordemos que el URL `by-capital` navega al componente `ByCapitalPageComponent`, regla que ha sido definida en el mismo `CountriesRoutingModule`

```
{
  path: 'by-capital',
  component: ByCapitalPageComponent
}
```

Nuevo Componente SearchBox

Para realizar las búsquedas vamos a crear un nuevo componente, la idea es crear un componente que pueda ser reutilizado por cualquier tipo de búsqueda, de modo que vamos a tener que enviar algunos parámetros, y el componente tendrá que emitir el valor a buscar, para que sea el componente padre, el que llame el servicio correspondiente. El componente estará ubicado en el directorio *Shared*

Agregamos el *ShareSearchBoxComponent*

Template:

```
<input
  type="text"
  [placeholder]="placeholder"
  class="form-control"
  (keyup.enter)="searchBy()"
  #txtSearchInput>
```

TypeScript:

```
export class SearchBoxComponent {

  @Input() public placeholder: string = 'Search...';

  @ViewChild('txtSearchInput') txtSearchInput!: ElementRef<HTMLInputElement>;

  @Output() onNewSearch: EventEmitter<string> = new EventEmitter();

  public searchBy() {
    if (this.txtSearchInput.nativeElement.value.length == 0)
      return;

    this.onNewSearch.emit(this.txtSearchInput.nativeElement.value);
  }
}
```

Aca podemos observar:

Input placeholder Este control necesita que quien lo implemente le envíe el texto a mostrar en el placeholder.

@ViewChild('txtSearchInput') El template tiene un elemento `#txtSearchInput` el cual lo hacemos visible en el typescript para poder acceder directamente a su valor.

@Output() onNewSearch: EventEmitter Esto indica que el componente EMITE un valor string, el valor ingresado por el usuario, y el componente padre debe capturarla para realizar la búsqueda correspondiente.

Implementando el SearchBoxComponent

Vamos a implementarlo en el *ByCapitalPageComponent*

Primero debemos agregarlo en el template la referencia al control

```
<shared-search-box
    placeholder="Search by Capital"
    (onNewSearch)="searchByCapital($event)">
</shared-search-box>
```

Tal como lo mencionamos, el que implemente este control debe de enviar un valor para el *PlaceHolder* y debe esperar un evento (*onNewSearch*)

El evento *onNewSearch* es capturado y a la vez llama el método *searchByCapital*

Finalmente el código del control, tiene el método para iniciar la búsqueda.

```
export class ByCapitalPageComponent {

    public searchByCapital(term: string) {
        //TODO call the service
    }
}
```

De la misma forma podemos implementar el mismo control *SearchBoxComponent* en los otros componente:

- *ByCountryPageComponent*
- *ByRegionPageComponent*

Countries API

Usaremos el API de <https://restcountries.com/>

Usaremos este ENDPOINT y le enviaremos una capital para obtener una muestra de los datos.

<https://restcountries.com/v3.1/capital/{capital}>

Coparemos el JSON de respuesta y lo vamos a transformar en una serie de interfaces que las guardaremos en el archivo *country.interface.ts*

Podemos usar la extensión: *Paste JSON as Code*, abrimos el archivo *country.interface.ts*, luego en la linea de comandos de Code (ctrl + p) escribimos *Paste json as code* y le damos el nombre de la interfaz raiz *countries* y listo, solo necesitamos tener en el clipboard el JSON respuesta.

Tambien podemos usar el sitio web <https://app.quicktype.io/> que es del mismo proveedor de la extensión.

Verificamos las interfaces creadas.

Nuevo Servicio

Con el Angular CLI Creamos un nuevo servicio

```
$ ng g s countries/services/countries.service.ts
CREATE src/app/countries/services/countries.service.ts.service.spec.ts (419
bytes)
CREATE src/app/countries/services/countries.service.ts.service.ts (147
bytes)
```

Y agregamos el siguiente código

```
@Injectable({
  providedIn: 'root'
})
export class CountriesServiceTsService {

  private apiUrl: string = 'https://restcountries.com/v3.1';

  constructor(private httpClient: HttpClient) { }

  /**
   * Search countries by Capital
   * @param term string
   * @returns Observable<Country>
   */
  searchByCapital(term: string): Observable<Country[]> {
    const url = `${this.apiUrl}/capital/${term}`;
    return this.httpClient.get<Country[]>(url);
  }
}
```

NOTA Importante: Aca estamos usando el *HttpClient*

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

No solo basta con el *import* en este caso tenemos que Importar el *HttpClientModule*. Dónde lo haremos? tiene que ser en un Módulo de la APP, si las peticiones REST las haremos en todos los componentes, lo ideal es agregarlo al archivo *app.module.ts*

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    HttpClientModule
  ],
  providers: []
})
```

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

Si omitimos este paso, vamos a tener un error:

```
ERROR NullInjectorError: R3InjectorError(_CountriesModule)
[_CountriesService -> _CountriesService -> _HttpClient -> _HttpClient]:
NullInjectorError: No provider for _HttpClient!
```

La razón por la cual necesitas importar HttpClientModule es porque HttpClient es un servicio proporcionado por Angular, y estos servicios deben ser registrados en un módulo para que Angular pueda inyectarlos correctamente en los componentes y servicios que los utilizan.

Regresando al Servicio, el código es bastante simple, el método principal recibe un término de búsqueda, puede ser una capital o simplemente cualquier palabra, el API puede retornar un arreglo de objetos o países, por ejemplo si envíamos el término *San*, podría retornar

- San Marino
- Yemen
- Chile
- El Salvador
- Puerto Rico
- Dominican Republic
- Costa Rica

Por lo tanto debemos esperar un arreglo de países *Country[]*

El *httpClient.get* retorna un *Observable* del mismo tipo *Country[]*

NOTA: *Country* es la interfaz que creamos en el paso anterior.

Implementando el Servicio

La búsqueda por Capital va a ser usada por el componente *ByCapitalPageComponent* Inyectaremos el Nuevo servicio. El código del componente queda de la siguiente forma:

```
@Component({
  selector: 'countries-by-capital-page',
  templateUrl: './by-capital-page.component.html',
  styles: ``
})
export class ByCapitalPageComponent {

  private countries: Country[] = [];

  constructor(private countryService: CountriesService) { }
```

```

public get getCountries(): Country[] {
  return this.countries;
}

public searchByCapital(term: string) {
  this.countryService.searchByCapital(term)
    .subscribe(
      (countries) => {
        this.countries = countries;
      });
}
}

```

Posteriormente podemos mostrar rápidamente en una lista el nombre de los países, en el template colocamos esto:

```

<div class="row">
  <div class="col">
    <ul>
      <li *ngFor="let country of getCountry">
        {{ country.name.common }} - {{ country.capital}}
      </li>
    </ul>
  </div>
</div>

```

El resultado para el término *San* es:

Países	By Capital
By Capital	San • San Marino - City of San Marino • Yemen - Sana'a • Chile - Santiago • El Salvador - San Salvador • Puerto Rico - San Juan • Dominican Republic - Santo Domingo • Costa Rica - San José
By Region	
By Country	

Country Table

Implementaremos un nuevo control, para mostrar los países en una tabla.

```
$ ng g c countries/countryTable
```

En el template del nuevo componente creamos por el momento una tabla HTML

```

<div *ngIf="countries.length === 0; else table" class="alert alert-warning text-center">
    No countries to list!
</div>

<ng-template #table>
    <table class="table table-hover">
        <thead>
            <tr>
                <th>#</th>
                <th>Icon</th>
                <th>Name</th>
                <th>Capital</th>
                <th>Population</th>
                <th>  </th>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let country of countries; index as rnum">
                <td>{{rnum + 1}}</td>
                <td>
                    {{country.flag}}
                </td>
                <td>{{country.name.common}}</td>
                <td>{{country.capital}}</td>
                <td>{{country.population | number}}</td>
                <td></td>
            </tr>
        </tbody>
    </table>
</ng-template>

```

En código del componente:

```

@Component({
    selector: 'countries-country-table',
    templateUrl: './country-table.component.html',
    styleUrls: ['./country-table.component.css']
})
export class CountryTableComponent {

    @Input() public countries: Country[] = [];
}

```

En la sección de *declarations* del *CountriesModule* agregamos el nuevo componente *CountryTableComponent*. Aunque el componente esta definido dentro de la misma sección, debemos de

hacerlo vivible en el mismo módulo.

Luego en el template del componente ** borramos el *UL* temporal que habíamos creado y agregamos el nuevo componente:

```
<div class="row">
  <div class="col">
    <countries-country-table [countries]="getCountries"> </countries-
country-table>
  </div>
</div>
```

Recordemos que el *getCountries* retorna la lista de Países que generó la última búsqueda.

CatchErorr: Manejo de Errores en Observables

Actualmente no tenemos ningún manejador de errores, cuando hacemos la búsqueda de con un término que no genera resultados, obtenemos, del lado del servicio, un *404*. Si la tabla ya tiene resultados, da la impresión al usuario que el nuevo término generó los mismos resultados.

Lo que haremos es implementar un *catchError* del lado del servicio y si ocurre un error, entonces retornamos un arreglo vacío.

El código actualizado es el siguiente:

```
/**
 * Search countries by Capital
 * @param term string
 * @returns Observable<Country>
 */
searchByCapital(term: string): Observable<Country[]> {
  const url = `${this.apiUrl}/capital/${term}`;
  return this.httpClient.get<Country[]>(url).pipe( catchError( (err) =>
of([]) ) );
}
```

Usamos un *pipe* y capturamos cualquier error, si hay un error, entonces usamos el operador *of* para generar un nuevo observable pero con un arreglo vacío.

Con esto resolvemos el problema mencionado.

Agregando más criterios de búsquedas

Todos los pasos que se han realizado para mostrar los países por Capital, se deben repetir para las búsquedas por Región y por Nombre de país.

Agregaremos los nuevos métodos al Servicio y de paso refactorizamos

```
@Injectable({
providedIn: 'root'
})
export class CountriesService {

private apiUrl: string = 'https://restcountries.com/v3.1';

constructor(private httpClient: HttpClient) { }

/**
 * Search countries by Capital
 * @param term string
 * @returns Observable<Country>
 */
searchByCapital(term: string): Observable<Country[]> {
  const url = `${this.apiUrl}/capital/${term}`;
  return this.searchCountries(term, url);
}

/**
 * Search countries by Name
 * @param term string
 * @returns Observable<Country>
 */
searchByName(term: string): Observable<Country[]> {
  const url = `${this.apiUrl}/name/${term}`;
  return this.searchCountries(term, url);
}

/**
 * Search countries by Region
 * @param term string
 * @returns Observable<Country>
 */
searchByRegion(term: string): Observable<Country[]> {
  const url = `${this.apiUrl}/region/${term}`;
  return this.searchCountries(term, url);
}

/**
 * Search countries by Term and URL
 * @param term string
 * @param url string
 * @returns Observable<Country>
 */
private searchCountries(term: string, url: string): Observable<Country[]> {
  return this.httpClient.get<Country[]>(url).pipe( catchError( (err) =>
of([]) ) );
}
}
```

El resto de los cambios se aplican igual, en los templates y el código javascript de los componentes:

- ByCountryPageComponent
- ByRegionPageComponent

Argumentos por URL

Cuando navegamos al siguiente URL `http://localhost:4200/countries/by/CRC` el router hace que se cargue el `CountryPageComponent` debido a esta regla de ruteo:

```
{
  path: 'by/:id',
  component: CountryPageComponent
}
```

Para acceder a la lista de parámetros usaremos un servicio: `ActivatedRoute`

Inyectamos el servicio y obtenemos el Params:

```
export class CountryPageComponent implements OnInit{

  constructor(private activatedRoute: ActivatedRoute ) { }

  ngOnInit(): void {
    this.activatedRoute.params
      .subscribe( (params) => {
        console.log({ params });
      });
  }
}
```

El console log imprime:

```
{
  "id": "CRC"
}
```

Para extraer el `id` podemos usar `params["id"]` o en el suscribe deestructura el params de esta forma:

```
this.activatedRoute.params
  .subscribe( ({ id }) => {
    console.log({ params: id });
  });

```

Este último código imprime:

```
{params: 'CRC'}
```

Una vez que tenemos acceso al parámetro, que indica el código del país, necesitamos extender nuestro *CountriesService* para buscar países por códigos.

Búsqueda por Código de País

Usaremos el endpoint `/alpha/:id` y crearemos un nuevo método en nuestro servicio

```
/**
 * Get country by Alpha code
 * @param id string
 * @returns Observable<Country>
 */
getCountryByAlpha(id: string): Observable<Country[]> {
  const url = `${this.apiUrl}/alpha/${id}`;
  return this.searchCountries(id, url);
}
```

Este método utiliza nuestro método único de búsquedas

```
/**
 * Search countries by Term and URL
 * @param term string
 * @param url string
 * @returns Observable<Country>
 */
private searchCountries(term: string, url: string): Observable<Country[]>
{
  return this.httpClient.get<Country[]>(url).pipe( catchError( (err) =>
of([]) ) );
}
```

Siguiente paso: injectar el servicio en el *CountryPageComponent* y luego consumirlo:

```
export class CountryPageComponent implements OnInit{

  constructor(
    private activatedRoute: ActivatedRoute,
    private countriesService: CountriesService ) { }

  ngOnInit(): void {
    this.activatedRoute.params
    .subscribe( ({ id } ) => {
      this.countriesService.getCountryByAlpha(id).subscribe( country => {
        console.log( { country } );
      }
    });
  }
}
```

```
    } );
  });
}
}
```

Esto imprime un arreglo, normalmente con un solo elemento, un País. Pero realmente lo que necesitamos es un único elemento, y hay otro problema mayor, tenemos un llamado a un API dentro de otro llamado a otro API (endpoint), es decir nos suscribimos a un Observable pero dentro del bloque del primer Observable, tenemos otro, esto se llama CallbackHell y se debe evitar.

Para evitar este escenarios, debemos usar operadores de alto nivel de rxJS, en este caso usaremos el *switchMap*

```
ngOnInit(): void {
  this.activatedRoute.params
    .pipe(switchMap( ({ id }) =>
  this.countriesService.getCountryByAlpha(id)))
    .subscribe( country => {console.log(country);});
}
```

Explicación del Código

this.activatedRoute.params nos da un observable, el resultado es un objeto **params : Param**, pero en lugar de suscribirnos, lo que implementamos es un **pipe**, es como colocar un "tubo" en la salida de un Observable, cualquier cosa que genera el primer Observable pasará por esa tubería.

En Angular, los observables son fundamentales para trabajar con flujos asíncronos de datos, como eventos del usuario, peticiones HTTP y otros eventos que pueden ocurrir en el tiempo. El método *switchMap* es una de las numerosas funciones de alto nivel proporcionadas por la biblioteca RxJS que ayuda a gestionar estos flujos de datos de manera más efectiva.

Vamos a desglosar el código para entenderlo paso a paso:

this.activatedRoute.params Este es un observable que emite eventos cada vez que los parámetros de la ruta cambian. En Angular, el objeto *ActivatedRoute* representa la ruta activa, y *params* es un observable que emite un evento cada vez que los parámetros de la ruta cambian.

.pipe(switchMap({ id }) => this.countriesService.getCountryByAlpha(id)): Aquí estamos utilizando el método *pipe* para encadenar operadores en el observable. **switchMap** es uno de esos operadores. Este operador se utiliza para transformar cada valor emitido por el observable (**activatedRoute.params**) en otro observable (**this.countriesService.getCountryByAlpha(id)**). Además, el uso de *switchMap* implica que solo te interesa el observable más reciente generado por *getCountryByAlpha(id)* y descartas cualquier observación anterior si aún no se ha completado.

{ id } => this.countriesService.getCountryByAlpha(id): Esta es una función de proyección que toma el valor emitido por **activatedRoute.params** (en este caso, un objeto con una propiedad *id*) y devuelve un nuevo observable (**this.countriesService.getCountryByAlpha(id)**).

switchMap: Este operador toma cada valor emitido por **activatedRoute.params**, aplica la función de proyección y "cambia" al nuevo observable retornado por la función de proyección. Si un nuevo evento llega antes de que el observable retornado anteriormente por **getCountryByAlpha(id)** se complete, **el antiguo observable se cancela y se sustituye por el nuevo.**

.subscribe(country => {console.log(country);}): Finalmente, te suscribes al observable resultante después de aplicar switchMap. En este punto, estás recibiendo los datos finales emitidos por `this.countriesService.getCountryByAlpha(id)` después de que ha sido procesado por el operador switchMap. En este caso, el código dentro de subscribe simplemente imprime el país en la consola.

Para finalizar, a parte del **SwitchMap** existen otras funciones, el **ConcatMap** y el **MergeMap** la diferencia radica en como actúan con respecto al momento en el que el observable genera un nuevo evento (Observer)

SwitchMap Cancela el antiguo observable y los sustituye por el nuevo

ConcatMap Espera que termien el observable actual para procesar el siguiente. Los valores emitidos por los observables internos se concatenan en lugar de superponerse.

MergeMap combina los valores emitidos por todos los observables internos, incluso si no han completado. Puede superponerse en el tiempo.

Cambios en el CountriesServices

Ahora necesitamos una forma de retornar un solo país, en lugar de un arreglo, podemos usar el mismo concepto, aplicar un PIPE con una función reductora para retornar un único elemento.

El método actual es el siguiente:

```
getCountryByAlpha(id: string): Observable<Country[]> {
  const url = `${this.apiUrl}/alpha/${id}`;
  return this.searchCountries(id, url);
}
```

Recordemos que **searchCountries** regresa siempre un **Observable<Country[]>** Lo que necesitamos es que al final **getCountryByAlpha** regrese un **Observable<Country | null>** el null es porque si la búsqueda no retorna nada, genera un error, y necesitamos regresar un null a la capa superior.

El nuevo método es como sigue:

```
getCountryByAlpha(id: string): Observable<Country | null> {
  const url = `${this.apiUrl}/alpha/${id}`;
  return this.searchCountries(id, url)
    .pipe(map( (countries:
      Country[]) => countries.length > 0 ? countries[0] : null ))
}
```

Estamos usando nuevamente un PIPE, y esta vez usamos una función **map**, si el arreglo tiene al menos un elemento, regresaremos siempre el primero, dado que enviamos una petición por código de país, esperamos que siempre retorne null o un país.

Con este cambio, el método **getCountryByAlpha** ya retorna un Observable<Country | null>

Validacion del Null

Si la consulta por código de país retorna un **null** debemos sacar al usuario de esa página, no tiene sentido mantener ese URL porque tiene un código incorrecto,

Para ello, en el **CountryPageComponent** inyectaremos un nuevo servicio **private router: Router** y luego validamos el null. Si la respuesta es null, entonces enviamos al usuario a otra pantalla con **this.router.navigateByUrl('/countries');** Caso contrario, tendremos la información del país en la variable **public country?: Country;**

```
export class CountryPageComponent implements OnInit{

  constructor(
    private activatedRoute: ActivatedRoute,
    private countriesService: CountriesService,
    private router: Router ) { }

  public country?: Country;

  ngOnInit(): void {
    this.activatedRoute.params
      .pipe(switchMap( ({ id }) =>
    this.countriesService.getCountryByAlpha(id)))
      .subscribe( country => {
        if (!country) {
          this.router.navigateByUrl('/countries');
          return;
        }

        this.country = country;
      });
  }
}
```

Mostrar información por País

CountryPageComponent ya tiene la información del país seleccionado, en la variable **country** ahora trabajaremos en el template para mostrar información:

Tenemos un **ng-template**, el cual se muestra brevemente mientras no se ha cargado la info del país, y luego, trabajaremos en el HTML para mostrar toda la data. Por el momento solo mostramos el nombre:

```

<ng-template #loading>
  <div class="alert alert-info text-center">
    Loading country information! Wait a moment please...
  </div>
</ng-template>

<div *ngIf="country; else loading">
  <div class="row">
    <div class="col-12">
      <h2>Country: {{country.name.common}}</h2>
    </div>
  </div>
</div>

```

Podemos agregar más información, algunos de los datos necesitan un tratamiento especial, por ejemplo el **translations**

En la interfaz tenemos:

```
translations: { [key: string]: Translation };
```

Podemos usar un **ngFor** y obtener el KEY del Mapa.

```

<span class="badge bg-secondary m-1"
  *ngFor="let currency of country.currencies| keyvalue" >
  {{currency.key | uppercase}} {{ country.currencies[currency.key].name
}}
</span>

```

El resultado de navegar a este url: <http://localhost:4200/countries/by/SMR> será:

Países	Country: San Marino	
By Capital By Region By Country	 <h3>Translations</h3> <p> ARA آرجنٹینا BRE San Marino CES San Marino CYM San Marino DEU San Marino EST San Marino FIN San Marino FRA Saint-Marin HRV San Marino HUN San Marino ITA San Marino JPN サンマリノ KOR 산마리노 NLD San Marino PER سان مارينو POL San Marino POR San Marino RUS Сан-Марино SLK San Marino SPA San Marino SRP САН МАРИНО SWE San Marino TUR San Marino URD سان مارینو ZHO 圣马力诺 </p>	<h3>Flag</h3>  <h3>Information</h3> <p> Population: 33,938 people Code: SMR Currencies: EUR Euro </p>

Navegar desde el CountryTableComponent

Al final de la tabla agreguemos un link:

```
<td>
  <a [routerLink]=[ '/countries/by', country.cca3]" class="btn btn-
outline-primary btn-sm">
    View
  </a>
</td>
```

Dado que estamos usando el **routerLink** necesitamos importar el **RouterModule** en el módulo al que pertenece el **CountryTableComponent** es decir en el **CountriesModule**:

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

@NgModule({
  imports: [
    RouterModule
  ]
})
export class CountriesModule { }
```

Listo, con esto tenemos un link al final de la tabla que abre el componente **CountryPageComponent** con el país seleccionado

Países	By Capital					
By Capital	Buenos Aires					
By Region	#	Icon	Name (code)	Capital	Population	Region
By Country	1		Argentina (ARG)	Buenos Aires	45,376,763	Americas

Nueva Sección: Mejoras y Funcionalidades Extras

¿Qué veremos en esta sección?

Aquí continuaremos la aplicación de países, pero enfocados en la parte de las sugerencias, debounce y mantener el estado de las consultas, este es un breve listado de los temas fundamentales:

- ngClass y diferentes formas de manipular clases
- CSS condicionales
- Optimizaciones a peticiones HTTP
- Debounce manual mediante RxJs
- LocalStorage
- Ideas de Store
- Re-utilización de componentes
- Varios operadores de RxJS aplicados

Nuevo componente Loading

Vamos agregar un componente que muestre un Spinner mientras cargamos un componente, en este caso la tabla de países.

```
$ ng g c shared/components/loadingSpinner
CREATE src/app/shared/components/loading-spinner/loading-
spinner.component.css (0 bytes)
CREATE src/app/shared/components/loading-spinner/loading-
spinner.component.html (30 bytes)
CREATE src/app/shared/components/loading-spinner/loading-
spinner.component.spec.ts (658 bytes)
CREATE src/app/shared/components/loading-spinner/loading-
spinner.component.ts (234 bytes)
UPDATE src/app/shared/shared.module.ts (1033 bytes)
```

Agregamos en el template:

```
<div class="spinner-container bg-dark">
  <span>Searching...</span>
  
</div>
```

Los Estilos

```
.spinner-container {
  align-items: center;
  background-color: black;
```

```

border-radius: 20px;
bottom: 15px;
box-shadow: 0px 0px 10px 0px rgba(0, 0, 0, 0.75);
color: white;
display: flex;
padding: 5px 10px;
position: fixed;
right: 15px;
}

span {
  margin-left: 10px;
  font-weight: 800;
}

```

El componente: (no hemos agregado nada)

```

@Component({
  selector: 'shared-loading-spinner',
  templateUrl: './loading-spinner.component.html',
  styleUrls: ['./loading-spinner.component.css'
})
export class LoadingSpinnerComponent {
}

```

Lo declaramos y lo exportamos en el **SharedModule**

Y luego lo implementamos, primero en el **ByCapitalPageComponent** en el template, en cualquier parte, agregamos el componente

```
<shared-loading-spinner *ngIf="isLoading"></shared-loading-spinner>
```

Tal como se puede ver depende de una propiedad **isLoading** la cual debemos definir en nuestro componente **ByCapitalPageComponent**

```

export class ByCapitalPageComponent {

  private countries: Country[] = [];

  public isLoading: boolean = false;

  constructor(private countryService: CountriesService) { }

  public get getCountries(): Country[] {
    return this.countries;
  }
}

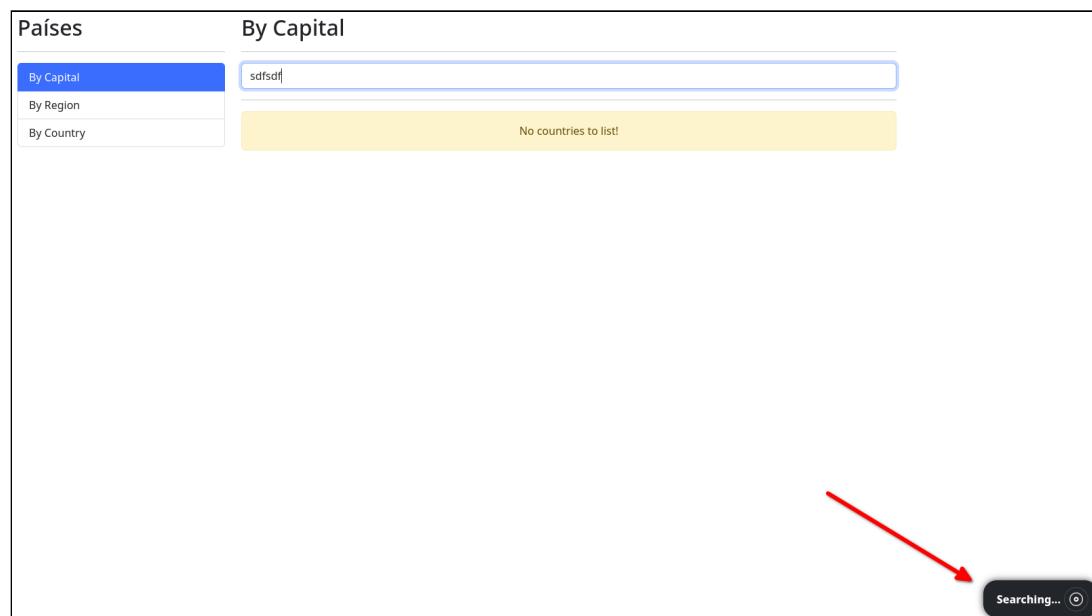
```

```

public searchByCapital(term: string) {
  this.isLoading = true;
  this.countryService.searchByCapital(term)
    .subscribe(
      (countries) => {
        this.countries = countries;
        this.isLoading = false;
      });
}
}

```

isLoading es *False* y justo cuando iniciamos la petición **searchByCapital** se cambia a *True* esto hace que se muestre el Spinner, y cuando se completa la carga se regresa a *False*



Debounce

Vamos a escribir esta funcionalidad que permite al usuario realizar la búsqueda sin la necesidad de presionar ENTER, luego de que el usuario deja de escribir por cierto tiempo, vamos a iniciar la búsqueda automáticamente.

Lo primero es agregar un nuevo evento en el textBox del Search, `*SearchBoxComponent`

```

<input
  type="text"
  [placeholder]="placeholder"
  class="form-control"
  (keyup.enter)="searchBy()"
  (keyup)="onKeyPress($event)"
  #txtSearchInput>

```

Luego debemos implementar el evento **onKeyPress(\$event)** para ello antes debemos crear un debouncer, y usaremos el **Subject** de rxJS,

```
/**
 * A Subject is a special type of Observable that allows values to be
 * multicasted to many Observers. Subjects are like EventEmitters.
 *
 * Every Subject is an Observable and an Observer. You can subscribe to a
 * Subject, and you can call next to feed values as well as error and
 * complete.
 */
export class Subject<T> extends Observable<T> implements SubscriptionLike
{}
```

Implementamos un debouncer que emitira **strings** (texto de búsqueda del usuario)

```
private debouncer: Subject<string> = new Subject();
```

Luego debemos suscribirnos a ese OBservable, lo hacemos en el **ngOnInit**

```
ngOnInit(): void {
  this.debouncer
    .pipe(
      debounceTime(300)
    )
    .subscribe( value => {
      this.searchBy();
    });
}
```

Aca vemos que se implementa un **PIPE** para agregar un delay de 300 ms cuando se emite un evento. Luego de ese delay, se dispara el **subscribe** que al final es el que llama a la búsqueda.

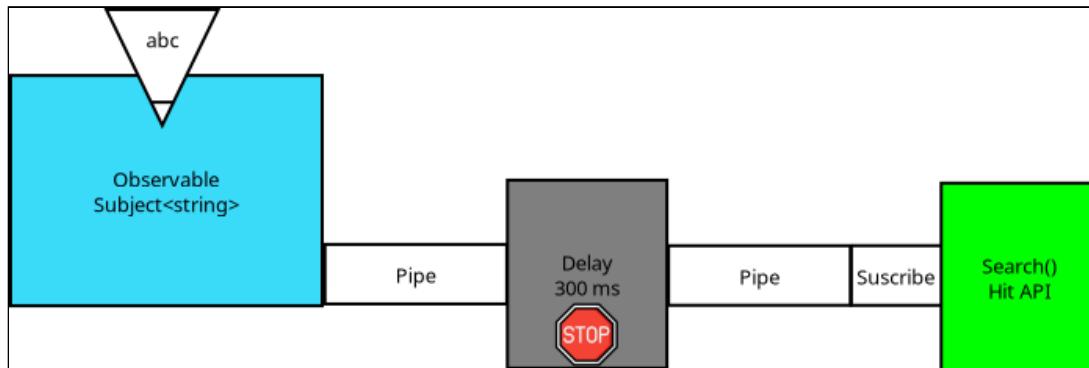
Y finalmente, debemos escribir el código que hace la emisión, eso lo hacemos en el método que se llama desde el componente. Es decir desde nuestro **onKeyPress**

```
public onKeyPress(event: any) {
  this.debouncer.next(this.txtSearchInput.nativeElement.value);
}
```

Entonces, resumiendo:

- El usuario escribe en la caja de texto
- Esto llama al **onKeyPress** quien emite un evento con el **this.debouncer.next**

- El `.pipe(debounceTime(300))` detiene por 300 MS el evento
- Si el usuario continúa escribiendo, se genera un nuevo evento y se cancela el anterior
- Si el usuario deja de escribir durante 300 MS, se ejecuta la suscripción `.subscribe(value => {this.searchBy();});`
- Eso dispara la búsqueda `this.searchBy()`;
- El usuario ve datos en pantalla



Limpieza de suscripciones

Generalmente, a nivel de componentes, cuando estos se destruyen, las suscripciones siguen activas, y aunque no se disparen más, (hasta que el componente se cree nuevamente) es mejor limpiarlas. Para ello se implementa el `ngOnDestroy`

```
ngOnDestroy(): void {
  this.debouncer.unsubscribe();
}
```

Optimizar búsqueda por regiones

La búsqueda por región solo acepta ciertas regiones, lo mejor es usar un dropdown o algo similar para que el usuario seleccione una región específica en lugar de usar el `SearchBox`.

Primero en el componente **ByRegionPageComponent** agregamos un tipo y un arreglo de ese tipo

```
type regions = 'africa' | 'americas' | 'asia' | 'europe' | 'oceania';
public regions: regions[] = ['africa', 'americas', 'asia', 'europe',
  'oceania'];
```

Luego, en el template del mismo componente, vamos a eliminar la referencia al componente **shared-search-box** y lo reemplazaremos por una lista de botones, uno por cada región.

```
<button
  *ngFor="let region of regions"
  [ngClass]="{{ 'btn-outline-primary': region === activeRegion }}"
  />
```

```

    class="btn btn-block m-2 text-center"
    (click)="searchByRegion(region)"
  >
    {{ region | titlecase }}
</button>

```

Se agrega un [ngClass] para permitir agregar una clase que muestre que botón ha sido seleccionado, el resultado es:

Países	By Region						
	Africa	Americas	Asia	Europe	Oceania		
#	Icon	Name (code)	Capital	Population	Region		
1	FR	France (FRA)	Paris	67,391,582	Europe	View	
2	VAT	Vatican City (VAT)	Vatican City	451	Europe	View	
3	NLD	Netherlands (NLD)	Amsterdam	16,655,799	Europe	View	

Mantener Data en Pantalla

La versión actual de la app, no permite persistir datos cuando nos movemos entre pantallas, si realizamos una búsqueda y nevemos al detalle de un país, al regresar hemos perdido el resultado de la búsqueda realizada anteriormente.

Esto sucede porque cuando cambiamos de ruta, el componente presentado se destruye para mostrar uno nuevo. Y varias propiedades, como la lista de países, se inicializan con un arreglo vacío, al inicializar el componente, por lo tanto la tabla de países siempre aparece vacía.

Se puede implementar LocalStorage, para que cada componente guarde la lista de países, o podemos usar algún tipo de estructura en memoria para guardar los datos.

usaremos un Objeto en memoria, y el mejor lugar para colocarlo es el servicio, ya que este se inyecta una sola vez y permanece en memoria durante la ejecución de la app.

En primer lugar, el tipo *Regions* que habíamos agregado a nivel del servicio, lo vamos a crear en un archivo independiente dentro de un directorio *Types*

```

export type Regions = 'africa' | 'americas' | 'asia' | 'europe' |
'oceania';

```

Luego vamos a crear una interfaz, dentro del directorio *interfaces*

```

import { Regions } from "../types/regions.types";
import { Country } from "./country.interface";

export interface cacheStorage {
  byCapital: cacheData;
  byCountry: cacheData;
}

```

```

        byRegion : cacheRegionData;
    }

export interface cacheData {
    term      : string;
    countries: Country[];
}

export interface cacheRegionData {
    region?   : Regions;
    countries : Country[];
}

```

Y luego, en nuestro servicio, creamos la estructura de datos:

```

public cacheCountries: cacheStorage = {
    byCapital: { term: '', countries: [] },
    byCountry: { term: '', countries: [] },
    byRegion : { region: undefined, countries: [] },
}

```

Notar que para Region hemos definido como nullable, porque no podemos inicializarla con una cadena vacía, porque no es aceptado por nuestro tipo de datos **Regions**

Por lo tanto la inicializamos en *undefined*

Guardando datos en el cacheStorage

El punto ideal para guardar es en el servicio, tan pronto realizamos la búsqueda debemos de guardar en el cacheStorage.

Por ejemplo, en la búsqueda por Capital

```

searchByCapital(term: string): Observable<Country[]> {
    const url = `${this.apiUrl}/capital/${term}`;
    return this.searchCountries(term, url);
}

```

Pero recordemos que el **this.searchCountries(term, url)**; retorna un *Observable*, y como necesitamos seguir regresando el mismo tipo de datos no podemos *Suscribirnos* para acceder a los *Countries*, pero, podemos usar un **PIPE** con una función RXJS para acceder a los Countries, llenar el Cache y retornar siempre un Observable, lo haremos con el **TAP**

```

searchByCapital(term: string): Observable<Country[]> {
    const url = `${this.apiUrl}/capital/${term}`;
    return this.searchCountries(term, url)
}

```

```

    .pipe(
      tap( (countries) => {
        this.cacheCountries.byCapital = { term, countries };
      })
    );
}

```

El **TAP** únicamente extrae el dato de la tubería, y no es necesario especificar un **return**, el flujo continua siendo un Observable.

A diferencia del **MAP** u otras funciones, estas interceptan el valor, lo procesan y debemos retornar algo a la tubería explicitamente.

Aplicamos el mismo cambio en las otras búsquedas dentro del mismo servicio y listo, ahora falta usar esa información en memoria.

Usando el cacheStorage

En este caso, debemos implementar esto a nivel de componente, todo aquel componente que use la lista de países, podemos ir al **cacheStorage** y tomar los datos iniciales de allí, en lugar de inicializarlo con un arreglo vacío.

También necesitamos restaurar el último término de búsqueda.

Haremos en cambio en el **ByCapitalPageComponent**, ya tenemos una propiedad para la lista de países, agreguemos una para el termino de búsqueda:

```

private countries: Country[] = [];
public sotoredSearchValue: string = '';

```

En el `ngOnInit` necesitamos cargar estos valores desde el Caché:

```

ngOnInit(): void {
  this.sotoredSearchValue =
  this.countryService.cacheCountries.byCapital.term;
  this.countries =
  this.countryService.cacheCountries.byCapital.countries;
}

```

Con este cambio, ya estamos listos con el asunto de los países, veremos que la tabla de Países mantiene el último resultado cuando nos movemos entre componentes, pero no así con el término de búsqueda. Para el caso de la búsqueda por Capital y País usamos el **SearchBoxComponent**, debemos pasarle el valor guardado en memoria, y para ello necesitamos agregar un `@Input` en dicho componente:

```

@Input() public sotoredSearchValue: string = '';

```

y luego lo usamos en su template:

```
<input
  type="text"
  [value]="sotoredSearchValue"
  [placeholder]="placeholder"
  class="form-control"
  (keyup.enter)="searchBy( )"
  (keyup)="onKeyPress($event)"
  #txtSearchInput>
```

Y finalmente necesitamos pasarle ese valor desde el componente que lo implementa, en este caso desde **ByCapitalPageComponent**

```
<div class="col">
  <shared-search-box
    [sotoredSearchValue]="sotoredSearchValue"
    placeholder="Search by Capital"
    (onNewSearch)="searchByCapital($event)">
  </shared-search-box>
</div>
```

Con ese cambio, podemos mantener tanto el último término de búsqueda como la lista de países cuando nos movemos entre componentes.

La implementación es exactamente igual para el componente **ByCountryPageComponent** pero en el caso de la región debemos usar otra forma para indicar que región había sido seleccionada, porque en ese caso usamos botones en lugar del SearchBOX.

```
ngOnInit(): void {
  this.countries = this.countryService.cacheCountries.byRegion.countries;
  this.activeRegion = this.countryService.cacheCountries.byRegion.region;
}
```

Agregando Mapas (No incluido en el curso)

Usaremos la librería **leaflet**

```
npm install leaflet @asymmetrik/ngx-leaflet leaflet-minimap
```

Agregamos un componente básico:

```
$ ng g c shared/components/map
CREATE src/app/shared/components/map/map.component.css (0 bytes)
CREATE src/app/shared/components/map/map.component.html (18 bytes)
CREATE src/app/shared/components/map/map.component.spec.ts (580 bytes)
CREATE src/app/shared/components/map/map.component.ts (187 bytes)
UPDATE src/app/shared/shared.module.ts (1146 bytes)
```

Buscamos la sección de Styles en el archivo **angular.json** y agregamos:

```
"styles": [
  "node_modules/leaflet/dist/leaflet.css",
  "node_modules/leaflet-minimap/dist/Control.MiniMap.css"
  "src/styles.css"
```

Tambien agregamos los JS

```
"scripts": [
  "node_modules/leaflet/dist/leaflet.js",
  "node_modules/leaflet-minimap/dist/Control.MiniMap.js"
]
```

El componente se define de la siguiente forma, el template es muy simple:

```
<div class="map-container">
  <div class="map-frame">
    <div id="map"></div>
  </div>
</div>
```

Y el componente en sí contiene el siguiente código:

```
import { AfterViewInit, Component, Input } from '@angular/core';
import * as leaflet from 'leaflet';
import { CountryMap } from
'../../../../countries/interfaces/country.interface';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'shared-map',
  templateUrl: './map.component.html',
  styleUrls: ['./map.component.css']
})
export class MapComponent implements AfterViewInit{
```

```
@Input() public countries!: CountryMap[];  
  
private map!: leaflet.Map;  
  
private initMap(): void {  
  
    if (!this.countries) {  
        return;  
    }  
  
    var lat = this.countries[0].latlong[0];  
    var lng = this.countries[0].latlong[1];  
  
    this.map = leaflet.map('map', {  
        center: [ lat, lng ],  
        zoom: 5  
    });  
  
    const tiles =  
        leaflet.tileLayer('https://s.tile.openstreetmap.org/{z}/{x}/{y}.png', {  
            maxZoom: 18,  
            minZoom: 3,  
            attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'  
        });  
  
    tiles.addTo(this.map);  
  
    this.loadGeoJson(this.countries[0].cca3);  
}  
  
constructor(private httpClient: HttpClient) {}  
  
ngAfterViewInit(): void {  
    this.initMap();  
}  
  
private loadGeoJson(countryCode: string): void {  
    const geoJsonUrl = `../../assets/geojsonfiles/${countryCode}.geojson`;  
  
    this.httpClient.get(geoJsonUrl).subscribe((geoJson: any) => {  
        const geoJsonLayer = leaflet.geoJSON(geoJson);  
        geoJsonLayer.addTo(this.map);  
    });  
}
```

El componente acepta un arreglo de países, pero por el momento solo estamos mostrando el primer país

```
var lat = this.countries[0].latlng[0];
var lng = this.countries[0].latlng[1];
```

La idea es que cuando abrimos la información de un país en específico debemos colorear el área de dicho país en el mapa, para ello necesitamos unos archivos especiales, los geojson, el cual contiene toda la información para crear la capa necesaria en el mapa.

Estos archivos fueron descargados del sitio <https://www.geoboundaries.org> se descargó el archivo global, y mediante un script en PHP se crearon archivos individuales usando el código de país

```
<?php

ini_set('memory_limit', '8912M');

// Read the geoBoundariesCGAZ_ADM0.geojson file
$geoBoundariesCGAZ_ADM0 =
file_get_contents('geoBoundariesCGAZ_ADM0.geojson');
$geoBoundariesCGAZ_ADM0 = json_decode($geoBoundariesCGAZ_ADM0, true);

// Loop through the features
foreach ($geoBoundariesCGAZ_ADM0['features'] as $feature) {
    // Get the country code
    $countryCode = $feature['properties']['shapeGroup'];

    // Create the file name
    $fileName = 'byCountry/' . $countryCode . '.geojson';

    // Create the FeatureCollection structure
    $featureCollection = [
        'type' => 'FeatureCollection',
        'crs' => [
            'type' => 'name',
            'properties' => [
                'name' => 'urn:ogc:def:crs:OGC:1.3:CRS84'
            ]
        ],
        'features' => [$feature] // Contiene solo la característica actual
    ];

    // Convertir la estructura a JSON
    $featureCollectionJson = json_encode($featureCollection,
JSON_PRETTY_PRINT);

    // Crear el archivo con la estructura FeatureCollection
    file_put_contents($fileName, $featureCollectionJson);
}
```

Esto generó un archivo por país el cual es leído desde el método **loadGeoJson**

```
private loadGeoJson(countryCode: string): void {
  const geoJsonUrl = `../../assets/geojsonfiles/${countryCode}.geojson`;

  this.httpClient.get(geoJsonUrl).subscribe((geoJson: any) => {
    const geoJsonLayer = leaflet.geoJSON(geoJson);
    geoJsonLayer.addTo(this.map);
  });
}
```

Se agregó una interfaz para enviar datos reducidos al mapa y no todo el objeto de Country:

```
export interface CountryMap {
  name: string;
  cca2: string;
  ccn3: string;
  cca3: string;
  cioc: string;
  latlng: number[];
  maps: Maps;
}
```

Ese es el tipo de dato que recibe el componente Mapa

```
@Input() public countries!: CountryMap[];
```

Luego usamos el nuevo componente, lo agregaremos al **CountryPageComponent** primero definimos una propiedad:

```
public countryMap: CountryMap[] = [];
```

Esta propiedad se llena en el ngOnInit

```
this.countryMap = [this.CountryMapper(country)];
```

Recordemos que **country** contiene toda la información que proviene del servicio **this.countriesService.getCountryByAlpha(id)** adicionalmente, mapeamos toda la información del país a un objeto más simple.

```

private CountryMapMapper(country: Country): CountryMap {
  return {
    name: country.name.common,
    cca2: country.cca2,
    ccn3: country.ccn3,
    cca3: country.cca3,
    cioc: country.cioc,
    latlng: country.latlng,
    maps: country.maps,
  }
}

```

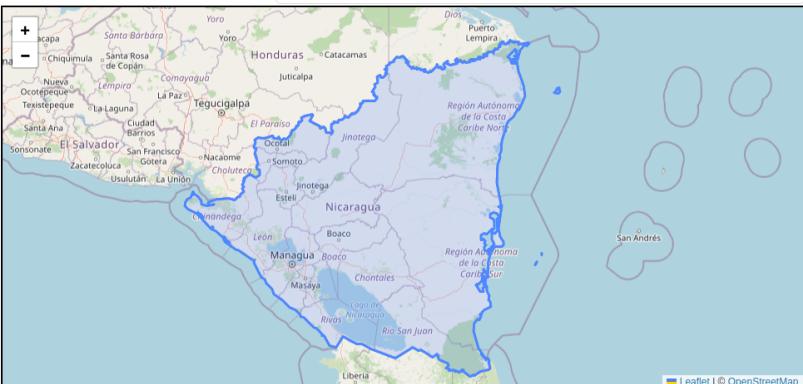
Finalmente agregamos a nuestro template la referencia al mapa y le pasamos un arreglo de países, en este caso un único país, el seleccionado para mostrar

```

<div class="map-component-container">
  <shared-map [countries]="countryMap"></shared-map>
</div>

```

Los resultados:

Países	Country: Nicaragua	
By Capital By Region By Country	Capital Managua 	Information Population: 6,624,554 people Code: NIC Currencies: NIO Nicaraguan córdoba Translations ARA BRE CES CYM DEU Nicaragua EST FIN FRA HRV HUN Nicaragua ITA JPN ニカラグア KOR 나ic라과 NLD Nicaragua PER پر POL POR Nicaragua RUS Никарагуа SLK Nicaragua SPA Nicaragua SRP Nikaragua SWE Nicaragua TUR Nikaragua URD ڀڌارڳا ZHO 尼加拉瓜
		

Nueva Sección: PIPES en Angular

¿Qué veremos en esta sección?

Esta sección es de mis favoritas del curso, veremos temas sobre Pipes y transformación visual de la data, pero también decidí mezclarlo con otro tema muy solicitado que es PrimeNG (Más adelante hay también una sección con muchos componentes de Angular Material), pero esta sección pretende enseñarles muchas cosas interesantes, aparte de los Pipes de Angular

Este es un breve listado de los temas fundamentales:

Todos los Pipes de Angular a la fecha

Uppercase, Lowercase, TitleCase, Date, Decimal, Currency, Percent, i18nSelect, i18nPlural, KeyValue, Json, Async, PrimeNG.

Módulo especializado para módulos de PrimeNG

PrimeNG es un paquete de componentes estilizados que contienen funcionalidades basadas en Angular para crear aplicaciones elegantes, hermosas y funcionales rápidamente.

Nueva APP

Crearemos una nueva APP para implementar los PIPES

```
ng new 04-pipesApp --standalone false
```

Borrar el template del HTML y dejar el

```
<router-outlet></router-outlet>
```

Que son los Pipes?

Los **pipes** (tuberías o filtros) son herramientas que te permiten transformar y dar formato a los datos que se muestran en la interfaz de usuario. Los pipes se utilizan principalmente en las plantillas de Angular para realizar operaciones de manipulación y presentación de datos de manera declarativa.

Angular proporciona varios pipes incorporados que cubren tareas comunes, como formatear fechas, números, cadenas de texto, entre otras. Algunos ejemplos de pipes integrados en Angular incluyen DatePipe, DecimalPipe, y UpperCasePipe.

Aquí hay un ejemplo básico de cómo se utiliza un pipe en una plantilla de Angular:

```
<!-- Supongamos que 'nombre' es una variable en el componente -->
<p>{{ nombre | uppercase }}</p>
```

En este ejemplo, **uppercase** es un pipe que convierte el valor de la variable nombre a mayúsculas antes de mostrarlo en la plantilla.

Además de los pipes integrados, también puedes crear tus propios **pipes personalizados** para realizar transformaciones específicas de datos según tus necesidades. Los pipes personalizados te permiten encapsular lógica de formato y reutilizarla en diferentes partes de tu aplicación.

Aquí hay un ejemplo simple de cómo crear un pipe personalizado en Angular:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'miPipePersonalizado'
})
export class MiPipePersonalizado implements PipeTransform {
  transform(valor: any): any {
    // Realizar la lógica de transformación aquí
    // Devolver el valor transformado
    return valor;
  }
}
```

Luego, en tu plantilla, puedes usar el pipe personalizado de la siguiente manera:

```
<p>{{ miDato | miPipePersonalizado }}</p>
```

PrimeNG

PrimeNG es una biblioteca de componentes de interfaz de usuario (UI) para Angular. Proporciona una amplia variedad de componentes UI que pueden ser utilizados en el desarrollo de aplicaciones web con Angular.

Instalación:

```
npm install primeng primeicons --save
```

También vamos a requerir instalar el Angular Animations, algunos controles, como el Menu requieren estas animaciones.

```
npm install @angular/animations --save
```

Agregamos al **Angular.json** los siguientes estilos

```
"styles": [
  "src/styles.css",
  "node_modules/primeng/resources/themes/lara-light-indigo/theme.css",
  "node_modules/primeng/resources/primeng.min.css",
  "node_modules/primeicons/primeicons.css"
],
```

NOTA: Podemos cambiar el theme, en lugar de lara-light-indigo podemos ubicar otro theme en el directorio y aplicarlo.

NOTA También podemos incluir los archivos en Assets para cambiarlos de forma dinámica. En el caso de agregarlos al **angular.json** debemos construir la APP.

Probaremos un botón, en el **app.module.ts**

```
import { ButtonModule } from 'primeng/button';
```

y lo colocamos en la sección **imports:** del mismo archivo, luego en el template:

```
<p-button label="Submit"></p-button>
```

También podemos usar algunos iconos

```
<p-button icon="pi pi-check"></p-button>
<p-button label="Submit" icon="pi pi-check"></p-button>
<p-button label="Submit" icon="pi pi-check" iconPos="right"></p-button>
```

Consultar el sitio <https://primeng.org/button> para obtener más información sobre como implementar los botones.

Módulo personalizado primeNg

Dado que estaremos importando muchos módulos de PrimeNg, una idea es crear un módulo personalizado y agregar los imports de primeNg en un solo lugar, esto es una idea, pueden existir otras implementaciones:

```
$ ng g m primeNg
CREATE src/app/prime-ng/prime-ng.module.ts (193 bytes)
```

El inconveniente es que centralizamos la carga de todos los módulos en un solo archivo, y los más seguro es que muchos componentes usaran unos cuantos módulos nada mas.

Agregando Menu

Antes crearemos la estructura básica de nuestra APP, por el momento agregamos el módulo Shared,

```
ng g m shared/shared  
CREATE src/app/shared/shared.module.ts (192 bytes)
```

Verificamo que el sharedModule este importado en el App.module.ts

Luego, dentro de la carpeta shared creamos la carpeta **components** y agregamos el componente Menu

```
$ ng g c shared/components/menu  
CREATE src/app/shared/components/menu/menu.component.css (0 bytes)  
CREATE src/app/shared/components/menu/menu.component.html (19 bytes)  
CREATE src/app/shared/components/menu/menu.component.spec.ts (587 bytes)  
CREATE src/app/shared/components/menu/menu.component.ts (191 bytes)
```

Dado que usaremos el menú, importamos el **SharedModule** en el **app.module.ts**

Vamos ahora al template del menu:

```
<p-menu [model]="items"></p-menu>
```

En el código del Menu Component agregamos:

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

@Component({
  selector: 'shared-menu',
  templateUrl: './menu.component.html',
  styleUrls: ['./menu.component.css']
})
export class MenuComponent implements OnInit{

  public items: MenuItem[] = [];

  ngOnInit(): void {
    this.items = [
      {label: 'Home', icon: 'pi pi-fw pi-home'},
      {label: 'Products', icon: 'pi pi-fw pi-shopping-cart'},
      {label: 'About', icon: 'pi pi-fw pi-info-circle'},
    ]
  }
}
```

Esto mostrará la página en blanco con un error:

```
RROR Error: NG05105: Unexpected synthetic listener @overlayAnimation.start found. Please make sure that:  
- Either `BrowserAnimationsModule` or `NoopAnimationsModule` are imported in your application.
```

Básicamente el componente de Menu de PrimeNg requiere un módulo de animación, ya sea el `BrowserAnimationsModule` o el `NoopAnimationsModule`. Al inicio de la sección hemos instalado el **BrowserAnimationsModule** de **@Angular** así que vamos a usarlo, primero debemos importarlo en el `app.module.ts` y luego lo agregamos a la sección **imports** del mismo archivo:

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

@NgModule({
  imports: [
    BrowserAnimationsModule,
  ]
})
export class AppModule { }
```

Con este cambio, el error desaparece y el menú se muestra en la pantalla:



Dado que ocupamos un menu horizontal, en el header de la APP, vamos a cambiar el **menu** por el **menubar**

1. En el **PrimeNgModule** importamos el menubar, y lo agregamos al **Exports[]**

```
import { MenubarModule } from 'primeng/menubar';

@NgModule({
  declarations: [],
  imports: [],
  exports: [
    MenubarModule,
  ]
})
```

```
})
export class PrimeNgModule { }
```

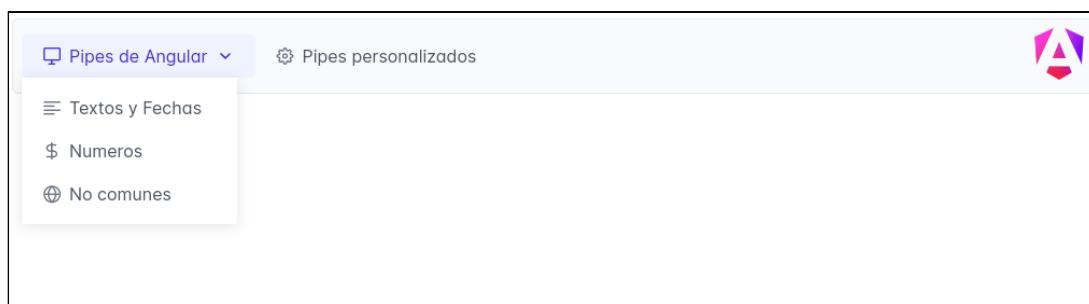
2. En nuestro **MenuComponent** creamos un menú con más opciones:

```
this.items = [
  {
    label: 'Pipes de Angular',
    icon: 'pi pi-desktop',
    items: [
      {
        label: 'Textos y Fechas',
        icon: 'pi pi-align-left',
        routerLink: '/'
      },
      {
        label: 'Numeros',
        icon: 'pi pi-dollar',
        routerLink: 'numeros'
      },
      {
        label: 'No comunes',
        icon: 'pi pi-globe',
        routerLink: 'no-comunes'
      }
    ]
  },
  {
    label: 'Pipes personalizados',
    icon: 'pi pi-cog',
    routerLink: 'ordenar'
  }
]
```

3. Cambiamos el template

```
<p-menubar [model]="items">
  
</p-menubar>
```

Verificamos el resultado:



Módulos y Rutas de la Application

Crearemos un módulo para relizar la carga perezosa,

```
$ ng g m products --routing
CREATE src/app/products/products-routing.module.ts (251 bytes)
CREATE src/app/products/products.module.ts (288 bytes)
```

--routing En Angular, un "routing module" es un módulo especializado que se utiliza para gestionar la navegación y la configuración de las rutas en una aplicación. Este módulo se encarga de definir las rutas y las asociaciones entre las URL y los componentes que se deben cargar cuando el usuario navega a una determinada URL.

Dentro del directorio **products** crearemos el directorio **pages** y dentro del mismo agregaremos nuestros componentes

```
ng g c products/pages/basicPage --skip-tests
ng g c products/pages/numbersPage --skip-tests
ng g c products/pages/uncommonPage --skip-tests
```

Resultado

```
$ tree src/app/products/pages/
src/app/products/pages/
├── basics-page
│   ├── basics-page.component.css
│   ├── basics-page.component.html
│   └── basics-page.component.ts
├── numbers-page
│   ├── numbers-page.component.css
│   ├── numbers-page.component.html
│   └── numbers-page.component.ts
└── uncommon-page
    ├── uncommon-page.component.css
    ├── uncommon-page.component.html
    └── uncommon-page.component.ts
```

Ya hemos creado el routingModule de nuestra página de productos, hemos agregado los componentes tipo Page, y ahora necesitamos agregar las rutas para navegar a dichas páginas.

En el **ProductsRoutingModule** (nuestro routing module para productos) agregamos las rutas:

```
const routes: Routes = [
  {
    path: '',
    component: BasicsPageComponent
  },
  {
    path: 'numbers',
    component: NumbersPageComponent
  },
  {
    path: 'uncommon',
    component: UncommonPageComponent
  },
  {
    path: '**',
    redirectTo: ''
  }
];
```

Recordemos que estas son rutas hijas, debemos agregarlas en el mismo **ProductsRoutingModule** como **Childs**

```
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ProductsRoutingModule { }
```

Ahora vamos a nuestro Router principal, **AppRoutingModule** y hacemos la carga perezosa de nuestro products Module

```
const routes: Routes = [
  {
    path: '',
    loadChildren: ()=> import('./products/products.module').then(m =>
      m.ProductsModule)
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
```

```
})
export class AppRoutingModule { }
```

Reordemos esta sintaxis:

```
loadChildren: ()=> import('./products/products.module').then(m =>
m.ProductsModule)
```

loadChildren: Es una propiedad de configuración de ruta en Angular. Se utiliza para cargar dinámicamente un módulo cuando se navega a una ruta específica.

()=> import('./products/products.module'): Aquí se utiliza una función de flecha para definir un punto de entrada dinámico al módulo que se va a cargar. La función import() es una característica de JavaScript llamada "import dinámico". Este enfoque permite cargar módulos de manera asíncrona, lo que es fundamental para la carga perezosa.

import('./products/products.module'): Esta parte del código indica que el módulo products.module se cargará de manera dinámica. El camino relativo ('./products/products.module') indica la ubicación del archivo del módulo.

.then(m => m.ProductsModule): Después de que el módulo es importado dinámicamente, la función then() se utiliza para especificar lo que debe suceder después de que la importación sea exitosa. En este caso, m representa el módulo importado, y m.ProductsModule es la referencia al módulo de productos que será cargado de forma perezosa.

Conectando el MenuBar con las Rutas.

Para Links internos, usamos el **routerLink** del **MenuItem** para enlazar a una ruta interna. Si fuese externa podemos usar el **URL**

```
this.items = [
{
  label: 'Pipes de Angular',
  icon: 'pi pi-desktop',
  items: [
    {
      label: 'Textos y Fechas',
      icon: 'pi pi-align-left',
      routerLink: '/'
    },
    {
      label: 'Numeros',
      icon: 'pi pi-dollar',
      routerLink: 'numbers'
    },
    {
      label: 'No comunes',
      icon: 'pi pi-globe',
    }
  ]
}
```

```

        routerLink: 'uncommon'
    }
]
}, {
    label: 'Pipes personalizados',
    icon: 'pi pi-cog',
    routerLink: 'ordenar'
}
]
```

Construir Basics Page

Vamos a agregar primeramente una serie de Imports en el **PrimeNgModule** los cuales los usaremos en varios de los componentes que estamos por desarrollar.

```

import { ButtonModule } from 'primeng/button';
import { CardModule } from 'primeng/card';
import { FieldsetModule } from 'primeng/fieldset';
import { MenubarModule } from 'primeng-menubar';
import { PanelModule } from 'primeng/panel';

@NgModule({
    declarations: [],
    imports: [],
    exports: [
        ButtonModule,
        CardModule,
        FieldsetModule,
        MenubarModule,
        PanelModule
    ]
})
export class PrimeNgModule {}
```

Si queremos agregar algún Module de PrimeNG recien importado, dentro de cualquie página o componente del nuevo módulo **Products** debemos importarlo en dicho módulo, recordemos que cada módulo requiere importar lo que necesita usar.

Entonces agregamos en nuestro **ProductsModule** en la sección de imports el **PrimeNgModule**

Y luego iniciemos con la construcción del Compomente, agregemos una propiedad pública en el código del compomente **BasicsPageComponent**

```
export class BasicsPageComponent {
```

```
public name: string = 'fReddy cRuz'  
}
```

Y agregemos esto en el template:

```
<p-panel [toggleable]="true">  
  <p-header>  
    <h2>Basics Angular Pipes</h2>  
  </p-header>  
  
  <p>  
    Basics Angular Pipes, included in CommonModule.  
  </p>  
  <p>  
    Consider having the following public property in your component:  
  </p>  
  <pre>  
    <code style="white-space: nowrap;">  
      <span style="color: blue;">public</span>  
      <span style="color: #333;"> name</span>  
      <span style="color: #333;">: string = </span>  
      <span style="color: green;">'fReddy cRuz'</span>  
    </code>  
  </pre>  
  
  <div class="grid">  
    <div class="col-12 md:col-4">  
      <p-card header="Uppercase">  
        <pre class="wrapper">{{ {{ name | uppercase }} }}  
produces {{ name | uppercase }}</pre>  
      </p-card>  
    </div>  
    <div class="col-12 md:col-4">  
      <p-card header="Lowecase">  
        <pre class="wrapper">{{ {{ name | lowercase }} }}  
produces {{ name | lowercase }}</pre>  
      </p-card>  
    </div>  
    <div class="col-12 md:col-4">  
      <p-card header="TitleCase">  
        <pre class="wrapper">{{ {{ name | titlecase }} }}  
produces {{ name | titlecase }}</pre>  
      </p-card>  
    </div>  
  </div>  
</p-panel>
```

PrimeFlex es una biblioteca de utilidades CSS que presenta varios helpers, como un sistema de cuadrícula, flexbox, espaciado, elevación y más. Las clases que agregamos al template anterior no tienen efecto aún, necesitamos instalar el PrimeFlex

```
npm install primeflex --save
```

Y luego agregamos el estilo en el **Angular.json**

```
"styles": [
  "src/styles.css",
  "node_modules/primeng/resources/themes/lara-light-indigo/theme.css",
  "node_modules/primeng/resources/primeng.min.css",
  "node_modules/primeicons/primeicons.css",
  "node_modules/primeflex/primeflex.css"
],
```

Con eso cambios se genera esta vista:

The screenshot shows a card titled 'Basics Angular Pipes'. It includes code snippets for a component's properties:

```
public name : string = 'fReddy cRuz'
public customDate : date = 'new Date();'
```

Three examples demonstrate pipes:

- Uppercase**: `{{ name | uppercase }}` produces `FREDDY CRUZ`
- Lowecase**: `{{ name | lowercase }}` produces `freddy cruz`
- TitleCase**: `{{ name | titlecase }}` produces `Freddy Cruz`

Agregar este otro Card

```
<div class="col-12 md:col-6">
  <p-card header="Date">
    <pre class="wrapper">{{ "{{ customDate | date }}" }} produces {{ customDate | date }}</pre>
    <pre class="wrapper">{{ "{{ customDate | date:'short' }}" }} produces {{ customDate | date: 'short' }}</pre>
    <pre class="wrapper">{{ "{{ customDate | date:'medium' }}" }} produces {{ customDate | date: 'medium' }}</pre>
    <pre class="wrapper">{{ "{{ customDate | date:'long' }}" }} produces {{ customDate | date: 'long' }}</pre>
    <pre class="wrapper">{{ "{{ customDate | date:'full' }}" }} produces {{ customDate | date: 'full' }}</pre>
    <pre class="wrapper">{{ "{{ customDate | date:'shortDate' }}" }} produces {{ customDate | date: 'shortDate' }}</pre>
    <pre class="wrapper">{{ "{{ customDate | date:'mediumDate' }}" }} produces {{ customDate | date: 'mediumDate' }}</pre>
```

```
<pre class="wrapper">{{ "{{ customDate | date:'shortTime' }}" }}  
produces {{ customDate | date: 'shortTime' }}</pre>  
</p-card>  
</div>
```

Produce esta salida:

Date

```
 {{ customDate | date }} produces Dec 31, 2023  
 {{ customDate | date:'short' }} produces 12/31/23, 10:46 PM  
 {{ customDate | date:'medium' }} produces Dec 31, 2023, 10:46:49 PM  
 {{ customDate | date:'long' }} produces December 31, 2023 at 10:46:49 PM GMT-6  
 {{ customDate | date:'full' }} produces Sunday, December 31, 2023 at 10:46:49 PM GMT-06:00  
 {{ customDate | date:'shortDate' }} produces 12/31/23  
 {{ customDate | date:'mediumDate' }} produces Dec 31, 2023  
 {{ customDate | date:'shortTime' }} produces 10:46 PM
```

Configuración del Locale

Para usar el PIPE Date con Locale, es necesario agregar un par de configuraciones, necesitamos importar el Locale que nuestra app va a usar y luego usar la función **registerLocaleData** para registrarlo.

```
import localeEsNI from '@angular/common/locales/es-NI';
import { registerLocaleData } from '@angular/common';

registerLocaleData(localeEsNI);

@NgModule({})
```

Ahora ya podemos usar el Locale ES-Ni en nuestra app.

```
 {{ customDate | date: 'full':'es-NI' }} produces lunes, 1 de enero de  
2024, 17:04:54 GMT-06:00
```

Definir globalmente el Locale

Podemos definir un Locale de forma global, para evitar tener que definirlo en cada lugar, por defecto está configurado el en-US, si queremos cambiarlo por otro, debemos cambiar nuestro app.module.ts,

Debemos mantener los cambios anteriores, es decir importar el Locales y registrarlos, pero para hacerlo que sea el default, debemos agregar un **providers** en el mismo app.module.ts

```
providers: [
  { provide: LOCALE_ID, useValue: 'es-NI' }
],
```

Ahora podemos ver las fechas con el locale es-NI.

Date

```
 {{ customDate | date }} produces 1 ene 2024
 {{ customDate | date:'short' }} produces 1/1/24, 17:13
 {{ customDate | date:'medium' }} produces 1 ene 2024 17:13:37
 {{ customDate | date:'long' }} produces 1 de enero de 2024, 17:13:37 GMT-6
 {{ customDate | date:'full' }} produces lunes, 1 de enero de 2024, 17:13:37 GMT-06:00
 {{ customDate | date:'shortDate' }} produces 1/1/24
 {{ customDate | date:'mediumDate' }} produces 1 ene 2024
 {{ customDate | date:'shortTime' }} produces 17:13
 {{ customDate | date: 'full':'::es-NI' }} produces lunes, 1 de enero de 2024, 17:13:37
GMT-06:00
```

NOTA: El LOCALE_ID se importa del **@angular/core**

```
import { LOCALE_ID, NgModule } from '@angular/core';
```

Pipes para Números

Creamos un nuevo componente para mostrar la funcionalidad de los Pipes de Números. (Son los mismos pasos y el mismo template, varían únicamente los Pipes a mostrar.)

En el caso de Number, si tenemos esta variable:

```
public amount: number = 123456.789;
```

Y aplicamos el pipe **number: '1.0-2'** obtendremos el valor **123,456.79**

El DecimalPipe se define de esta forma

```
 {{ value_expression | number [ : digitsInfo [ : locale ] ] }}
```

digitsInfo se define como:

```
{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}
```

minIntegerDigits: el número mínimo de dígitos enteros antes del punto decimal. El valor predeterminado es 1.

minFractionDigits: el número mínimo de dígitos después del punto decimal. El valor predeterminado es 0.

maxFractionDigits: el número máximo de dígitos después del punto decimal. El valor predeterminado es 3.

De modo que '**1.0-2**' va a generar un número con:

- Al menos un dígito antes del punto decimal
- Un mínimo de 0 decimales
- Un máximo de 2 decimales

Dado que el número original contiene 3 decimales **123456.789** el valor transformado será **123,456.79**

Pipes de Moneda

Consultar la documentación oficial para obtener más información sobre los parámetros de monedas. Estos son un par de ejemplos:

```
 {{ amount | currency: 'NIO' }} produces C$123,456.79  
 {{ amount | currency: 'EUR': 'symbol-narrow' | currency }} produces  
 €123,456.79  
 {{ amount | currency: 'USD': 'symbol-narrow': '1.4-4' }} produces  
 $123,456.7890
```

NOTA: la configuración **1.4-4** indica, al menos 4 decimales (puede agregar ceros) y máximo 4 decimales (puede aplicar redondeos)

Pipe de Porcentaje

Definida la siguiente propiedad:

```
public percent: number = 0.712;
```

Estos son un par de ejemplos:

```
 {{ percent | percent }} produces 71 %  
 {{ percent | percent: '1.0-2' }} produces 71.2 %  
 {{ percent | percent: '1.4-4' }} produces 71.2000 %
```

Uncommon Pipes

Iniciemos con el **i18nSelect**: Selector genérico que muestra la cadena que coincide con el valor actual. Por ejemplo, si vamos a enviar correos de invitación a personas, podemos cambiar algunos textos dependiendo

del género.

Creamos este componente:

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

type gender = 'male' | 'female';
export interface Client {
  name: string;
  gender: gender;
}

@Component({
  selector: 'app-uncommon-page',
  templateUrl: './uncommon-page.component.html',
  styleUrls: ['./uncommon-page.component.css']
})
export class UncommonPageComponent {

  //i18nSelect
  public clients: Client[] = [
    { name: 'Freddy', gender: 'male'},
    { name: 'Adrea', gender: 'female'},
    { name: 'Martin', gender: 'male'},
    { name: 'Ana', gender: 'female'}
  ];

  public selectedClient: Client = this.clients[0];

  public changeClient(): void {
    this.selectedClient = this.clients[Math.floor(Math.random() *
this.clients.length)];
  }
}
```

En resumen un arreglo de Clientes (nombre y género), un método que cambia aleatoriamente el cliente seleccionado.

Ahora veamos el Template:

```
<p-fieldset legend="i18nSelect Pipe" [toggleable]="true">
  <p>
    Saludos {{selectedClient.name}} es un placer invitarlo
    a nuestro evento anual de programadores.
  </p>

  <p-button (onClick)="changeClient()" label="Change Client">
  </p-button>
</p-fieldset>
```

Veamos algunas salidas:

— i18nSelect Pipe

Saludos Martin es un placer invitarlo a nuestro evento anual de programadores.

Change Client

— i18nSelect Pipe

Saludos Adrea es un placer invitarla a nuestro evento anual de programadores.

Change Client

Como podemos ver, en la segunda imagen, el nombre "Andrea" no se corresponde con "Invitarlo", podríamos agregar un **ngIf** que valide el género e imprima **Invitarla** pero para eso tenemos el pipe **i18nSelect**

Agreguemos un MAP, el ***invitationMap** que tiene el género como Key, y la palabra a cambiar como el Value.

```
public invitationMap = {
  male: 'invitarlo',
  female: 'invitarla'
};
```

y cambiemos nuestro template por:

```
<p>
  Saludos {{selectedClient.name}} es un placer
  {{ selectedClient.gender | i18nSelect: invitationMap }}
  a nuestro evento anual de programadores.
</p>
```

El resultado: P

— i18nSelect Pipe

Saludos Adrea es un placer invitarla a nuestro evento anual de programadores.

Change Client

i18nPlural Pipe

Creamos el siguiente template,

```
<p-fieldset legend="i18nPlural Pipe" [toggleable]="true">
  <p>
```

```

        Actualmente tenemos {{clients.length}} clientes esperando.
    </p>
    <p-button
        (onClick)="deleteClient()"
        label="Delete Client">
    </p-button>
</p-fieldset>
```

El método **deleteClient()** simplemente elimina un cliente de la lista, si ejecutamos varias veces el eliminar hasta dejar un solo cliente en el arreglo, se leerá:

```
Actualmente tenemos 1 clientes esperando.
```

Al igual que el **i18nSelect**, podemos proporcionar un Mapa y usar el pipe **i18nPlural**

Creamos el mapa:

```

public totalClientsMap = {
    '=0': 'no tenemos ningún cliente esperando.',
    '=1': 'tenemos un cliente esperando.',
    'other': 'tenemos # clientes esperando.'
};
```

Cambiamos nuestro Template a esto:

```

<p>
    Actualmente {{clients.length | i18nPlural: totalClientsMap}}
</p>
```

Esto genera las siguientes salidas:

```

Actualmente tenemos 4 clientes esperando.
Actualmente tenemos 3 clientes esperando.
Actualmente tenemos 2 clientes esperando.
Actualmente tenemos un cliente esperando.
Actualmente no tenemos ningún cliente esperando.
```

Slice Pipe

El **SlicePipe** Crea un nuevo Array o cadena que contiene un subconjunto (porción) de los elementos.

Creamos este método, crea un arreglo con los nombres de los clientes:

```
public clientNames(): string[] {
    return this.clients.map(client => client.name);
}
```

El siguiente Template:

```
<p-fieldset legend="Slice Pipe" [toggleable]="true">
    <b>Originals</b>
    <pre> {{ clientNames() }}</pre>
    <b>Slice:0:1</b>
    <pre> {{ clientNames() | slice:0:1 }}<
    <b>Slice:1:1</b>
    <pre> {{ clientNames() | slice:1:2 }}<
    <b>Slice:2:3</b>
    <pre> {{ clientNames() | slice:2:3 }}<
    <b>Slice:3</b>
    <pre> {{ clientNames() | slice:3 }}</pre>
</p-fieldset>
```

Genera:

```
Originals
Freddy,Adrea,Martin,Ana
Slice:0:1
Freddy
Slice:1:1
Adrea
Slice:2:3
Martin
Slice:3
Ana
```

Json Pipe

El siguiente template:

```
<p-fieldset legend="Json Pipe" [toggleable]="true">
    <pre> {{ clients | json }}</pre>
</p-fieldset>
```

Genera el contenido del arreglo de objetos Clients en formato JSON:

```
[  
{
```

```

        "name": "Freddy",
        "gender": "male"
    },
    {
        "name": "Adrea",
        "gender": "female"
    },
    {
        "name": "Martin",
        "gender": "male"
    },
    {
        "name": "Ana",
        "gender": "female"
    }
]

```

Key-Value Pipe

El keyValue permite transformar un Objeto tipo Mapa, Key - Value, en iterable. Por ejemplo

```

<p-fieldset legend="KeyValue Pipe" [toggleable]="true">
    <ul *ngFor="let client of clients[0] | keyvalue">
        <li>
            {{client | json }}
        </li>
    </ul>
</p-fieldset>

```

Dato que **clients[0]** retorna un objeto de tipo client

```

interface Client {
    name: string;
    gender: gender;
}

```

Este no es iterable, si omitimos el **keyvalue** obtenemos un error:

```
Type 'Client' is not assignable to type 'NgIterable<any>'
```

En resumen, **keyvalue** transforma Objeto o Mapa en un arreglo de pares Key-Value.

La salida del Template es:

```
{ "key": "gender", "value": "male" }  
{ "key": "name", "value": "Freddy" }
```

Async Pipe

Definamos un Observable en el componente

```
public myObservableTimer = interval(1000);
```

Cada 1 segundo, emitirá un evento.

Si colocamos esto en el Template:

```
<p-fieldset legend="Async Pipe" [toggleable]="true">  
    <pre>{{ myObservableTimer }}</pre>  
</p-fieldset>
```

No se emite ningún resultado, esto se debe a que no nos hemos suscrito al observable, si agregamos el pipe **async**

```
<pre>{{ myObservableTimer | async }}</pre>
```

Observaremos que cada 1 segundo se imprimir un valor, 0,1,2,... esto se debe a que **async** automáticamente se suscribe al Observable y recibe los eventos.

Además, si navegamos a otra página, y si colocamos un console.log con el valor emitido por el observable, veremos que la app deja de recibir los eventos, esto significa que **async** no solamente se suscribe, sino, que también elimina la suscripción una vez se destruye el componente.

Nueva sección: Pipes Personalizados

¿Qué veremos en esta sección?

Este es un breve listado de los temas fundamentales:

- Pipes personalizados
- Argumentos hacia los Pipes
- SortableTable manual y usando PrimeNg

Creando un Custom Pipe: Toggle-Case

Vamos a crear un módulo para tener nuestros customs Pipes en un lugar independiente

```
$ ng g m custom-pipes
CREATE src/app/custom-pipes/custom-pipes.module.ts
```

Procedemos a crear el Custom Pipe

```
$ ng g p custom-pipes/toggle-case/toggle-case
CREATE src/app/custom-pipes/toggle-case/toggle-case.pipe.spec.ts (204
bytes)
CREATE src/app/custom-pipes/toggle-case/toggle-case.pipe.ts (225 bytes)
UPDATE src/app/custom-pipes/custom-pipes.module.ts (284 bytes)
```

Analicemos el código autogenerado

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'toggleCase'
})
export class ToggleCasePipe implements PipeTransform {

  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }
}
```

Nuevamente observamos una simple clase con una anotación y una que otra implementación.

La Clase **ToggleCasePipe** tiene una anotación **@Pipe** que recibe como argumento requerido el **name** del pipe. Además esta misma clase implementa el **PipeTransform** y por lo tanto debe de implementar el

método transform

Dicho método acepta un parámetro **value** que es el valor que se desea transformar. Además acepta ninguno o varios parámetros adicionales **args**

Podemos implementar un CustomPipe Simple que cambie una Cadena de Minúsculas a Mayúsculas o lo contrario. Ese es el objetivo de nuestro Pipe.

Podemos cambiar el tipo del **value** porque solo aceptaremos un **string** y podemos remover los **args** porque no vamos a ocupar parámetros adicionales.

Definamos nuestro transform de la siguiente forma:

```
transform(value: string): string {
    if (!value || typeof value !== 'string' || value.length === 0) {
        return '';
    }

    if (value === value.toUpperCase()) {
        return value.toLowerCase();
    } else {
        return value.toUpperCase();
    }
}
```

Si el valor recibido no está definido, o no es un string o es un string vacío, retornamos un string vacío. El resto del código se explica por si mismo.

Ahora usemos el Custom Pipe, como es un módulo, el componente que dese usarlo, su módulo debe importarlo.

Como deseamos implementarlo en el componente **CustomsPageComponent** y este pertenece al módulo **ProductsModule** este último debe importar el módulo **CustomPipesModule**

NOTA: Dado que aún no implementamos los standalone, debemos usar los módulos por el momento.

Una vez importado en el **ProductsModule** ya podemos usarlo en nuestro template:

```
<p-card header="My TogleCase">
    <pre class="wrapper">{{ "{{ 'Testing' | toggleCase }}" }} produces {{ 'Testing' | toggleCase }}</pre>
    <pre class="wrapper">{{ "{{ 'TESTING' | toggleCase }}" }} produces {{ 'TESTING' | toggleCase }}</pre>
</p-card>
```

Custom Angular Pipes.

My TogleCase

```
{{ 'Testing' | toggleCase }} produces TESTING
{{ 'TESTING' | toggleCase }} produces testing
```

Aagumentos en Custom Pipes

Creemos un nuevo Pipe, uno que reciba un string, y tres parámetros, el primero un número, y los otros boolean, el primero define el ancho del texto generado, el segundo si el usuario quiere usar números y el tercero si quiere usar caractéres especiales. El objetivo del Pipe es generar un RandomPassword de ancho N con o sin Números y caracteres especiales.

```
{{value_expression | randomPassword [ : lenght [ : useNumer [ : useSpecialCharacter ]]]}}
```

Creamos el Pipe

```
$ ng g p custom-pipes/random-password/random-password
CREATE src/app/custom-pipes/random-password/random-password.pipe.spec.ts
(220 bytes)
CREATE src/app/custom-pipes/random-password/random-password.pipe.ts (233
bytes)
UPDATE src/app/custom-pipes/custom-pipes.module.ts (421 bytes)
```

Implementamos el código:

NOTA: Colocar un RandomPasswordGenerator en un Pipe quizás no es la mejor idea, pero solo se hace con propósitos educativos, para indicar como usar los parámetros.

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'randomPassword'
})
export class RandomPasswordPipe implements PipeTransform {

  private symbols = "~`!@#$%^&*()|:;,+-*"
  private alphabet =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
  transform(value: string, ...args: unknown[]): string {
    /
  }
}
```

```
const [length, allowNumbers, allowSpecial] = args as [number, boolean, boolean];

let randomNumber = '';
if (allowNumbers) {
    randomNumber = this.generateRandomNumber();
}

let randomSpecialChars = '';
if (allowSpecial) {
    randomSpecialChars = this.generateRandomString(Math.floor(length / 2)
, this.symbols);
}

let characters: string[] = value.split('')
.concat(randomNumber.split(''))
.concat(randomSpecialChars.split(''));

let refill = '';
if (characters.length < length) {
    refill = this.generateRandomString(length - characters.length,
this.alphabet);
}

characters = characters.concat(refill.split(''));
return this.shuffleArray(characters).join('').slice(0, length);
}

private generateRandomNumber(): string {
    return (Math.floor(Math.random() * (99999 - 10000 + 1)) +
10000).toString();
}

private generateRandomString(length: number, charset: string): string {
    let returnString = '';
    for (let i = 0; i < length; i++) {
        const indiceAleatorio = Math.floor(Math.random() * charset.length);
        returnString += charset.charAt(indiceAleatorio);
    }
    return returnString;
}

private shuffleArray(array: string[]): string[] {
    for (let i = array.length - 1; i > 0; i--) {
        const j = Math.floor(Math.random() * (i + 1));
        [array[i], array[j]] = [array[j], array[i]];
    }
    return array;
}
```

Ahora lo implementamos en nuestro template:

```
<p-card header="My RandomPassword">
    <pre class="wrapper">{{ "{{ 'Testing' | randomPassword:15:true:true }}"}}
}} produces '{{ 'Testing' | randomPassword:15:true:true }}'</pre>
    <pre class="wrapper">{{ "{{ 'BASE' | randomPassword:20:true:false }}"}}
}} produces '{{ 'BASE' | randomPassword:20:true:false }}'</pre>
    <pre class="wrapper">{{ "{{ 'BASE' | randomPassword:20:false:false }}"}
}} produces '{{ 'BASE' | randomPassword:20:false:false }}'</pre>
</p-card>
```

La Salida Sería:

My RandomPassword

```
{{ 'Testing' | randomPassword:15:true:true }} produces 'tn-5/.e087)T%*g'
{{ 'BASE' | randomPassword:20:true:false }} produces '5KQx83hBNAS4ARBEL3zK'
{{ 'BASE' | randomPassword:20:false:false }} produces 'eS10kLUVEAfEXSgaZUBW'
```

El SortBy Pipe

El siguiente ejemplo simplemente es para demostrar que el PIPE puede ser usado para transformar más que simples datos, podemos aplicarlo para cambiar los datos en un arreglo de objetos. El siguiente pipe ordena datos en una tabla, los datos provienen de un arreglo de productos. Además se implementa un toolbar de PrimeNG para mostrar botones que inician las acciones de ordenamiento, el código completo es el siguiente:

Primero necesitamos una interfaz para los productos.

```
export interface Product {
    code: string;
    name: string;
    category: string;
    quantity: number;
    discount: boolean;
}
```

Luego implementamos el código del componente Table.

```
Component({
    selector: 'app-tables-page',
    templateUrl: './tables-page.component.html',
    styleUrls: ['./tables-page.component.css']
})
export class TablesPageComponent {
```

```

public sortBy?: keyof Product | '' = '';

public products: Product[] = [
  code: 'AAA-1111',
  name: 'PlayStation 5',
  category: 'Games',
  quantity: 10,
  discount: true
]

public changeSort(sortBy: keyof Product | ''): void {
  console.log(sortBy);
  this.sortBy = sortBy;
}
}

```

NOTA: **products** contiene más datos, aca se muestra un solo registro para reducir el código en la documentación.

La propiedad **sortBy** únicamente puede tomar los valores de los KEY del tipo de datos Products, es decir: **code, name, category, quantity y discount** El valor de dicha propiedad es modificado por el método **changeSort**

Luego implementamos el PIPE **SortByPipe**

```

@Pipe({
  name: 'sortBy'
})
export class SortByPipe implements PipeTransform {

  transform(value: Product[], sortBy?: keyof Product | ''): Product[] {
    if (sortBy === '') {
      return value;
    }

    if (sortBy !== undefined) {
      return value.sort((a, b) => (a[sortBy] > b[sortBy]) ? 1 : -1);
    }

    return value;
  }
}

```

y otro Pipe para transformar el true:false del campo **discount** en algo mas legible

```

@Pipe({
  name: 'booleanConverter'
})

```

```
export class BooleanConverterPipe implements PipeTransform {  
  
    transform(value: boolean, ...args: string[]): string {  
        return (value) ? args[0] : args[1];  
    }  
}
```

Finalmente el Template del Componente que muestra la tabla:

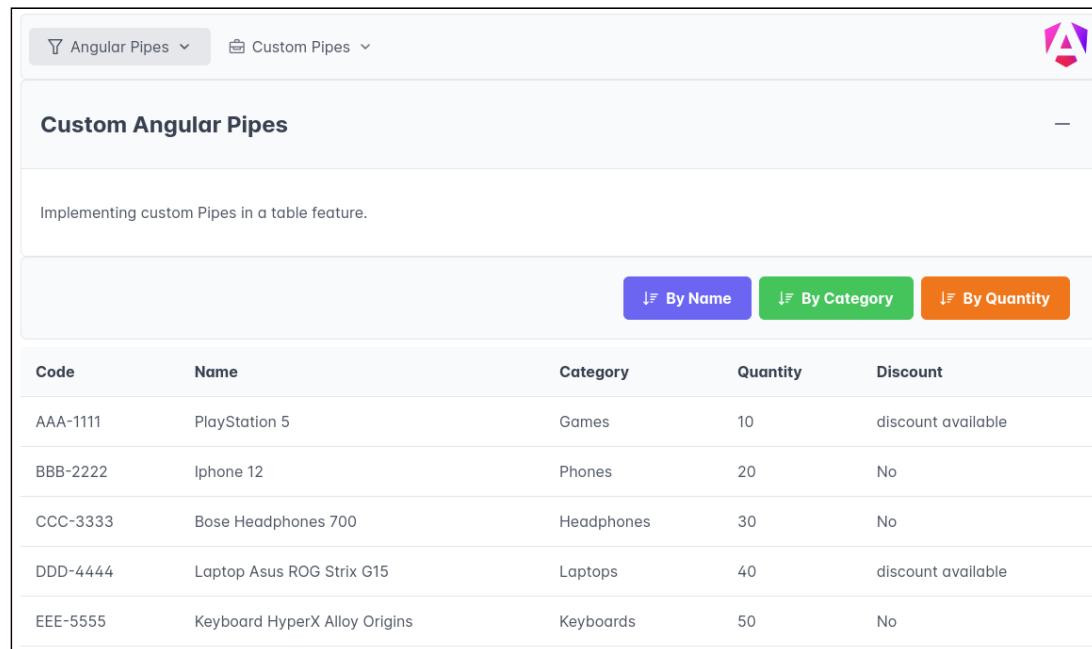
```
<p-panel [toggleable]="true">  
    <p-header>  
        <h2>Custom Angular Pipes</h2>  
    </p-header>  
  
    <p>  
        Implementing custom Pipes in a table feature.  
    </p>  
  
</p-panel>  
  
<p-toolbar>  
    <div class="p-toolbar-group-start">  
  
    </div>  
    <div class="p-toolbar-group-end">  
        <button pButton  
            label="By Name"  
            icon="pi pi-sort-amount-down"  
            class="mr-2"  
            (click) = "changeSort('name')">  
        </button>  
        <button pButton  
            label="By Category"  
            icon="pi pi-sort-amount-down"  
            class="p-button-success mr-2"  
            (click) = "changeSort('category')">  
        </button>  
        <button pButton  
            label="By Quantity"  
            icon="pi pi-sort-amount-down"  
            class="p-button-warning mr-2"  
            (click) = "changeSort('quantity')">  
        </button>  
    </div>  
</p-toolbar>  
  
<div class="grid">  
    <div class="col mt-2">  
        <p-table [value]="products | sortBy:sortBy" [tableStyle]:"{ 'min-width': '50rem' }">  
            <ng-template pTemplate="header">  
                <tr>
```

```

<th>Code</th>
<th>Name</th>
<th>Category</th>
<th>Quantity</th>
<th>Discount</th>
</tr>
</ng-template>
<ng-template pTemplate="body" let-product>
<tr>
<td>{{ product.code }}</td>
<td>{{ product.name }}</td>
<td>{{ product.category }}</td>
<td>{{ product.quantity }}</td>
<td>{{ product.discount | booleanConverter:'discount
available':'No'}}</td>
</tr>
</ng-template>
</p-table>
</div>
</div>

```

El resultado:



The screenshot shows a web application interface. At the top, there are two dropdown menus: "Angular Pipes" and "Custom Pipes". On the right side, there is a small purple logo. Below the menus, the title "Custom Angular Pipes" is displayed. A sub-instruction "Implementing custom Pipes in a table feature." is present. At the bottom of the page, there is a toolbar with three buttons: "By Name" (purple), "By Category" (green), and "By Quantity" (orange). The main content area contains a table with the following data:

Code	Name	Category	Quantity	Discount
AAA-1111	PlayStation 5	Games	10	discount available
BBB-2222	Iphone 12	Phones	20	No
CCC-3333	Bose Headphones 700	Headphones	30	No
DDD-4444	Laptop Asus ROG Strix G15	Laptops	40	discount available
EEE-5555	Keyboard HyperX Alloy Origins	Keyboards	50	No

Se importaron dos nuevos módulos de primeNG para este componente **TableModule** y **ToolbarModule**

El **sortBy** pipe se implementa sobre el arreglo productos:

```
[value]="products | sortBy:sortBy"
```

Donde **sortBy** (el parámetro) viene de la propiedad pública definida por el componente y que es modificada por cada botón:

```
(click) = "changeSort('quantity')
```

Listo, con esto hemos creado un SortBy mediante un Pipe, el cual puede ser extendido, como agregar parámetros ASC / DESC o incluso hacerlo genérico para que ordene otro tipo de arreglos.

Nueva Sección: LazyLoad y Layouts

¿Qué veremos en esta sección?

Este es un breve listado de los temas fundamentales:

- Rutas Hijas
- Rutas Principales
- LazyLoad
- Multiples estilos en la misma SPA

Esta sección es fundamental para seguir el curso, ya que de aquí en adelante, implementaremos la carga perezosa en cada módulo principal de las futuras aplicaciones que haremos.

Comprender el Lazyload no es difícil, y la forma como lo veremos aquí nos ayudará a manejar los dos posibles casos de uso. Uno de ellos es cuando la ruta hija no tiene estilo especial y otra en la que requiere un estilo diferente por cada módulo.

Nueva APP

Crearemos una nueva APP para mostrar el uso de Componentes Layouts y con especial enfasis en ruteo.

```
ng new herosApp --standalone false
```

Generar módulos iniciales

```
ng g m auth --routing
CREATE src/app/auth/auth-routing.module.ts (247 bytes)
CREATE src/app/auth/auth.module.ts (272 bytes)

ng g m heros --routing
CREATE src/app/heros/heros-routing.module.ts (248 bytes)
CREATE src/app/heros/heros.module.ts (276 bytes)

ng g m shared
CREATE src/app/shared/shared.module.ts (192 bytes)

ng g m material
CREATE src/app/material/material.module.ts (194 bytes)
```

El módulo **shared** es el único que vamos a importar dentro de **app.module.ts**

Todo lo demás se hará con LazyLoad.

Creando Componentes:

Agregamos la estructura básica de los componentes para el módulo **Auth**

```
$ ng g c auth/pages/layout-page --skip-tests  
$ ng g c auth/pages/login-page --skip-tests  
$ ng g c auth/pages/register-page --skip-tests
```

Creamos los componentes para el módulo **Heros**

```
$ ng g c heros/pages/hero-page --skip-tests  
$ ng g c heros/pages/layout-page --skip-tests  
$ ng g c heros/pages/list-page --skip-tests  
$ ng g c heros/pages/new-page --skip-tests  
$ ng g c heros/pages/search-page --skip-tests
```

Creamos los componentes para el módulo **Shared**

```
$ ng g c shared/pages/Error404-page --skip-tests
```

En este caso si vamos a exportar el **Error404PageComponent** en el módulo **SharedModule** porque vamos a usarlos en una ruta por default en el **app.routing.module.ts**

```
@NgModule({  
  declarations: [Error404PageComponent],  
  imports: [CommonModule],  
  exports: [Error404PageComponent]  
})  
export class SharedModule { }
```

Material

En este módulo vamos a importar todo lo referente a Angular Material, similar a lo que hicimos con PrimeNG, removemos el **CommonModule** de **material.module.ts**

Estructura final

Por el momento esta sería la estructura de nuestro proyecto:

```
[fcruz@fedora app]$ tree  
.  
├── app.component.css  
├── app.component.html  
├── app.component.spec.ts  
└── app.component.ts
```

```
├── app.module.ts
├── app-routing.module.ts
└── auth
    ├── auth.module.ts
    ├── auth-routing.module.ts
    └── pages
        ├── layout-page
        │   ├── layout-page.component.css
        │   ├── layout-page.component.html
        │   └── layout-page.component.ts
        ├── login-page
        │   ├── login-page.component.css
        │   ├── login-page.component.html
        │   └── login-page.component.ts
        └── register-page
            ├── register-page.component.css
            ├── register-page.component.html
            └── register-page.component.ts
├── heros
    ├── heros.module.ts
    ├── heros-routing.module.ts
    └── pages
        ├── hero-page
        │   ├── hero-page.component.css
        │   ├── hero-page.component.html
        │   └── hero-page.component.ts
        ├── layout-page
        │   ├── layout-page.component.css
        │   ├── layout-page.component.html
        │   └── layout-page.component.ts
        ├── list-page
        │   ├── list-page.component.css
        │   ├── list-page.component.html
        │   └── list-page.component.ts
        ├── new-page
        │   ├── new-page.component.css
        │   ├── new-page.component.html
        │   └── new-page.component.ts
        └── search-page
            ├── search-page.component.css
            ├── search-page.component.html
            └── search-page.component.ts
└── material
    └── material.module.ts
└── shared
    ├── pages
    │   └── error404-page
    │       ├── error404-page.component.css
    │       ├── error404-page.component.html
    │       └── error404-page.component.ts
    └── shared.module.ts
```

Configuración de Rutas

Iniciamos con la configuración de las rutas, primeramente nos vamos al **AuthRoutingModule**.

```
// localhost:4200/auth
const routes: Routes = [
  {
    path: '',
    component: LayoutPageComponent,
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class AuthRoutingModule { }
```

Y luego en el de **HerosRoutingModule**

```
// localhost:4200/heros
const routes: Routes = [
  {
    path: '',
    component: LayoutPageComponent,
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class HerosRoutingModule { }
```

NOTA: cada **LayoutPageComponent** corresponde al propio de cada módulo.

Luego definimos las rutas principales, en el **AppRoutingModule**

```
//localhost:4200
const routes: Routes = [
  {
    path: 'auth',
    loadChildren: () => import('./auth/auth.module').then(m =>
m.AuthModule),
  },
  {
    path: 'heros',
    loadChildren: () => import('./heros/heros.module').then(m =>
```

```
m.HerosModule),
},
{
  path: '404',
  component: Error404PageComponent,
},
{
  path: '',
  redirectTo: 'heros',
  pathMatch: 'full',
},
{
  path: '**',
  redirectTo: '404',
}
];
}

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

auth y **heros** son cargado de modo Lazy.

Por el momento tenemos creadas ciertas Rutas las cuales responden a estas URL:

```
http://localhost:4200/
http://localhost:4200/auth
http://localhost:4200/heros
http://localhost:4200/404
```

Aún debemos configurar las rutas internas, las que responden por ejemplo a **auth/login**

Rutas Children

El módulo **AuthRoutingModule** actualmente tiene configurada la ruta padre *localhost:4200/auth* podemos agregar las rutas hijas con el arreglo **children**

```
// localhost:4200/auth
const routes: Routes = [
{
  path: '',
  component: LayoutPageComponent,
  children: [
    // localhost:4200/auth/login
    {
      path: 'login', component: LoginPageComponent
    },
  ],
}
```

```
// localhost:4200/auth/register
{
  path: 'register', component: RegisterPageComponent
},
{
  path: '**', redirectTo: 'login',
}
]
}
];
;

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class AuthRoutingModule { }
```

También aplicamos el mismo cambio en el módulo **HerosRoutingModule**

```
// localhost:4200/heros
const routes: Routes = [
{
  path: '',
  component: LayoutPageComponent,
  children: [
    // localhost:4200/heros/list
    { path: 'list', component: ListPageComponent },
    // localhost:4200/heros/create
    { path: 'create', component: NewPageComponent },
    // localhost:4200/heros/edit/1
    { path: 'edit/:id', component: NewPageComponent },
    // localhost:4200/heros/search
    { path: 'search/', component: SearchPageComponent },
    // localhost:4200/heros/1
    { path: ':id', component: HeroPageComponent },
    { path: '**', redirectTo: 'list' }
  ]
};
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class HerosRoutingModule { }
```

NOTA: Es importante el orden de los paths, vemos que el último corresponde a **:id**, si colocamos este path de primero en la lista, cualquier otro path va a coincidir con este, de modo que se coloca al final.

Con estos cambios, ya tenemos las rutas listas:

```
// Parents  
  
http://localhost:4200/  
http://localhost:4200/auth  
http://localhost:4200/heros  
http://localhost:4200/404  
  
// Nuevos  
  
// Auth  
http://localhost:4200/auth/login  
http://localhost:4200/auth/register  
  
// Heros  
http://localhost:4200/heros/list  
http://localhost:4200/heros/create  
http://localhost:4200/heros/edit/1  
http://localhost:4200/heros/search  
http://localhost:4200/heros/1
```

Router Outlet

Con estas rutas, si cargamos cualquier ruta hija, por ejemplo: `http://localhost:4200/auth/login` vemos únicamente el **Layout-page works!** Esto corresponde al **LayoutPageComponent** del **AuthModule**, pero, ¿Cómo mostramos la página o componente hijo? usaremos nuevamente el **router-outlet**

Agregamos esto al template de cada **LayoutPageComponent** tanto del Auth como del Heros

```
<router-outlet></router-outlet>
```

Ahora si cargamos el mismo URL `http://localhost:4200/auth/login` Vemos:

```
layout-page works!  
  
login-page works!
```

Nueva Sección: Continuación de Heros APP / Con Angular Material:

¿Qué veremos en esta sección?

Este es un breve listado de los temas fundamentales:

- Angular Material
- Interfaces y tipado
- Pipes personalizados
- Variables de entorno
- Autocomplete de AngularMaterial
- Peticiones HTTP
- JSON-Server
- Prime Flex

Esta sección tiene por objetivo principal aprender a utilizar Angular Material, es la primera de varias secciones donde lo usaremos. Al final del día trabajaremos con muchos componentes de Angular material que les ayudará a ver cualquier otro tipo de paquete modularizado de la misma manera y ustedes sabrán aplicarlo.

En la siguiente sección después de esta, continuaremos la aplicación pero realizaremos un CRUD.

Instalación de Material

Vamos a usar dos cosas

- Angular Material
- PrimeFlex

De Angular Materials vamos a usar sus componentes y de PrimerFlex algunos estilos.

```
$ ng add @angular/material
```

Salida del Comando:

```
i Using package manager: npm
✓ Found compatible package version: @angular/material@17.0.4.
✓ Package information loaded.

The package @angular/material@17.0.4 will be installed and executed.
Would you like to proceed? Yes
✓ Packages successfully installed.
? Choose a prebuilt theme name, or "custom" for a custom theme: Indigo/Pink
[ Preview: https://material.angular.io?theme=indigo-pink ]
? Set up global Angular Material typography styles? Yes
```

```
? Include the Angular animations module? Include and enable animations
UPDATE package.json (1112 bytes)
✓ Packages installed successfully.
UPDATE src/app/app.module.ts (576 bytes)
UPDATE angular.json (3005 bytes)
UPDATE src/index.html (519 bytes)
UPDATE src/styles.css (181 bytes)
[fcruz@fedora 05-herosApp]$
```

En **src/app/app.module.ts** se importa el **BrowserAnimationsModule** porque hemos habilitado las animaciones, también se agregan unas fuentes en **src/index.html** (Roboto) que son agregadas tambien en nuestro archivo **src/styles.css** y se debe a que aceptamos esta sección "**Set up global Angular Material typography styles**"

Instalación de PrimeFlex

En este caso lo instalaremos desde el CDN

```
<link rel="stylesheet"
      href="https://unpkg.com/primeflex@latest/primeflex.css">
```

No vamos a instalar los themes de PrimeFlex para usar el de Angular Material.

Importar los módulos de Material

Importaremos los módulos que vamos a usar en la APP:

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

import { MatAutocompleteModule } from '@angular/material/autocomplete';
import { MatButtonModule } from '@angular/material/button';
import { MatCardModule } from '@angular/material/card';
import { MatChipsModule } from '@angular/material/chips';
import { MatDialogModule } from '@angular/material/dialog';
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatGridListModule } from '@angular/material/grid-list';
import { MatIconModule } from '@angular/material/icon';
import { MatInputModule } from '@angular/material/input';
import { MatListModule } from '@angular/material/list';
import { MatProgressSpinnerModule } from '@angular/material/progress-spinner';
import { MatSelectModule } from '@angular/material/select';
import { MatSidenavModule } from '@angular/material/sidenav';
import { MatSnackBarModule } from '@angular/material/snack-bar';
import { MatToolbarModule } from '@angular/material/toolbar';

@NgModule({
  exports: [
    MatAutocompleteModule,
```

```

        MatButtonModule,
        MatCardModule,
        MatChipsModule,
        MatDialogModule,
        MatFormFieldModule,
        MatGridListModule,
        MatIconModule,
        MatInputModule,
        MatListModule,
        MatProgressSpinnerModule,
        MatSelectModule,
        MatSidenavModule,
        MatSnackBarModule,
        MatToolbarModule,
    ]
})
export class MaterialModule { }

```

Pantalla del Login (Auth)

Dado que usaremos componente de Material en el **AuthModule** debemos importar el **MaterialModule**

Luego, apliquemos un cambio en el template del **LayoutPageComponent**

```

<div class="grid p-3">
  <div class="col-12 mt-5 md:col-6 md:col-offset-3 md:mt-8">
    <router-outlet></router-outlet>
  </div>
</div>

```

NOTA: Las Clases son de PrimeFlex

En el **LoginPageComponent** template:

```

<div class="flex flex-column">
  <span class="text-lg mb-4">Login</span>

  <mat-form-field>
    <mat-label>Usuario</mat-label>
    <input type="text" matInput placeholder="User Name">
  </mat-form-field>
  <mat-form-field>
    <mat-label>Password</mat-label>
    <input type="password" matInput placeholder="Password">
  </mat-form-field>

  <button mat-button mat-flat-button color="primary">
    <mat-icon>login</mat-icon>
    Login
  </button>
</div>

```

```

</button>
</div>

<div class="flex justify-content-end mt-5">
  <a routerLink="/auth/register">SingIn</a>
</div>

```

De nuevo, estamos usando clases de PrimeFlex y los controles de Material:

- mat-form
- mat-label
- matInput
- mat-icon

De esta misma forma podemos crear otra pantalla para el create, con un campo adicional para confirmar password.

El resultado:

The image contains two side-by-side screenshots of an Angular application's login interface. The left screenshot, titled 'Login', shows a form with two input fields labeled 'Usuario' and 'Password', a blue button labeled 'Login' with a right-pointing arrow icon, and a purple link labeled 'SingIn'. The right screenshot, titled 'Create new Account', shows a similar form with three input fields: 'Usuario', 'Password', and 'Confirm Password', a blue button labeled 'Create' with a file icon, and a purple link labeled 'Login'.

SideNav

Continuamos aplicando cambios, esta vez trabajamos en el LayoutPage del HerosModule, vamos a crear un SideNav y un Menu, el cual va a ser mostrado por todas las vistas renderizadas bajo HerosModule, entonces debemos de colocar esto en el componente padre del módulo, es decir en el template del **LayoutPageComponent**

Antes, debemos importar el **MaterialModule** en nuestro **HerosModule**

Luego creamos un Arreglo para las opciones del menú:

```
export class LayoutPageComponent {  
  
    public sidebarItems = [  
        {label: 'List', icon: 'list', url: './list'},  
        {label: 'New', icon: 'add', url: './create'},  
        {label: 'Search', icon: 'search', url: './search'},  
    ];  
}
```

Y luego creamos el Template del **LayoutPageComponent**

```
<mat-sidenav-container fullscreen>  
    <mat-sidenav #sidenav mode="push" [ngStyle]="{{width: '250px'}}">  
        <mat-toolbar color="accent">  
            <span>Menu</span>  
            <span class="spacer"></span>  
            <button mat-icon-button (click)="sidenav.toggle()">  
                <mat-icon matListItemIcon>{{item.icon}}</mat-icon>  
            </button>  
        </mat-toolbar>  
  
        <mat-nav-list>  
            <mat-list-item *ngFor="let item of sidebarItems"  
                [routerLink]="item.url"  
                (click)="sidenav.toggle()">  
                <mat-icon matListItemIcon>{{item.icon}}</mat-icon>  
                {{item.label}}  
            </mat-list-item>  
        </mat-nav-list>  
    </mat-sidenav>  
  
    <mat-toolbar color="primary">  
        <button mat-icon-button (click)="sidenav.toggle()">  
            <mat-icon>menu</mat-icon>  
        </button>  
        <span class="spacer"></span>  
        <button mat-button>  
            Logout  
        </button>  
    </mat-toolbar>  
  
    <div class="container">  
        <router-outlet></router-outlet>  
    </div>  
</mat-sidenav-container>
```

Todo el contenido está dentro del **mat-sidenav-container** el cual contiene 3 secciones:

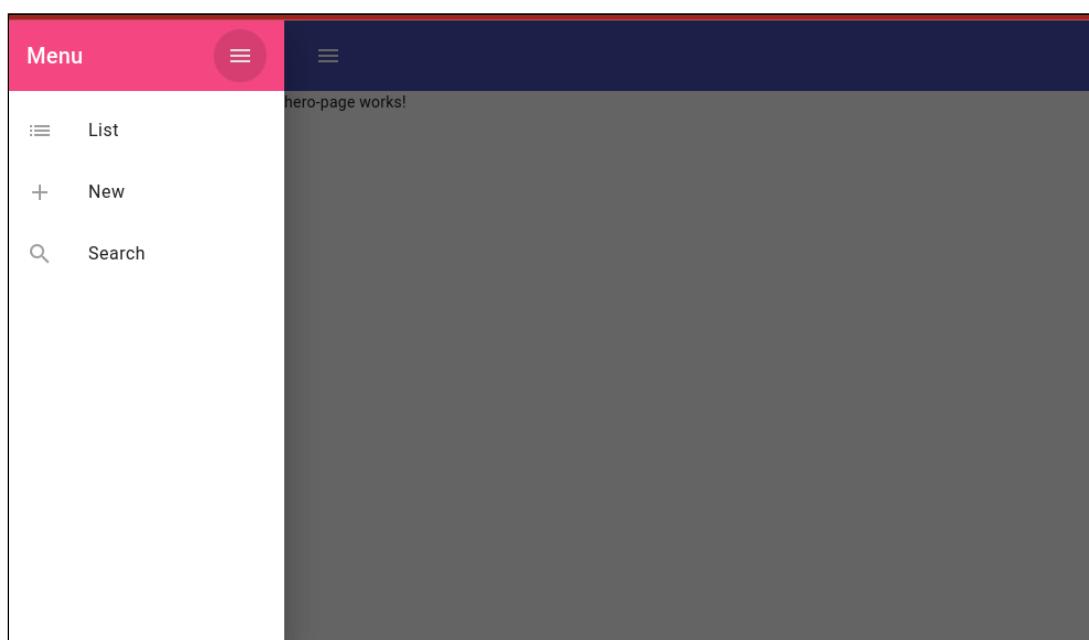
- Un SideBar (mat-sidenav)
- Un Toolbar (mat-toolbar)
- Un container para renderizar los componentes (router-outlet)

El Toolbar tiene un botón que hace aparecer el SideBar y viceversa

El Sidebar internamente cuenta con dos elementos

- mat-toolbar (Muestra el título y el botón para ocultar la sección)
- mat-nav-list (El que muestra las opciones del menú)

Veamos el resultado:



Json Server

Implementaremos un Servidor JSON únicamente en desarrollo, <https://www.npmjs.com/package/json-server>

```
npm install --save-dev json-server
```

Creamos un directorio **data** en el root de nuestro proyecto y agregamos un json con datos para mostrar, por ejemplo:

```
{
  "users": [
    {
      "id": 1,
      "usuario": "John Doe",
      "email": "john.doe@gmail.com"
```

```
        },
    ],
    "heroes": [
        {
            "id": "dc-batman",
            "superhero": "Batman",
            "publisher": "DC Comics",
            "alter_ego": "Bruce Wayne",
            "first_appearance": "Detective Comics #27",
            "characters": "Bruce Wayne"
        },
        {
            "id": "dc-superman",
            "superhero": "Superman",
            "publisher": "DC Comics",
            "alter_ego": "Kal-El",
            "first_appearance": "Action Comics #1",
            "characters": "Kal-El"
        }
    ]
}```
```

En nuestro Package JSON, agregamos un script:

```
```json
"backend": "json-server --watch data/db.json --port 3004"
```

Y finalmente ejecutamos el script en la línea de comandos: (En el root del proyecto)

```
$ npm run backend

> 05-heros-app@0.0.0 backend
> json-server --watch data/db.json --port 3004

\{^_^}/ hi!

Loading data/db.json
Done

Resources
http://localhost:3004/users
http://localhost:3004/heroes

Home
http://localhost:3004
```

## Variables de Entorno

Angular 17 ya no incluye por defecto la configuración de los environments, haremos la configuración manual.

Primero, dentro de `src` creamos un directorio `environments` con dos archivos

- `environments.ts`
- `environments.prod.ts`

Creamos las configuraciones de acuerdo al ambiente, por ejemplo nuestro URL para consumir el JSON Server

```
export const environments = {
 baseUrl: 'http://localhost:3004'
}
```

Y luego en el archivo **angular.json** en la sección:

```
Projects
 05-herosApp
 architect
 build
 configurations
 production
```

Agreamos el **fileReplacements**:

```
configurations": {
 "production": {
 "fileReplacements": [
 {
 "replace": "src/environments/environment.ts",
 "with": "src/environments/environment.prod.ts"
 }
],
 }
},
```

## Crear Interfaz

Primero debemos crear algunas interfaces, en el direcotrio `src/app/heroes/` creamos **interfaces** y dentro de este el archivo **heroes.interface.ts**

Usaremos la extensión **Paste JSON as code**, ejecutamos en la linea de comandos :

```
curl http://localhost:3004/heroes
```

Copiamos la salida y en el archivo **heroes.interface.ts** ejecutamos el comando **Paste JSON as code** se genera el siguiente código:

```
export interface Heroe {
 id: string;
 superhero: string;
 publisher: Publisher;
 alter_ego: string;
 first_appearance: string;
 characters: string;
}

export enum Publisher {
 DCComics = "DC Comics",
 MarvelComics = "Marvel Comics",
}
```

## Crear Servicio

Creamos un servicio:

```
@Injectable({providedIn: 'root'})
export class HeroesService {
 constructor(private httpClient: HttpClient) { }

 private baseUrl: string = environments.baseUrl;

 getHeroes(): Observable<Heroe[]> {
 return this.httpClient.get<Heroe[]>(`${this.baseUrl}/heroes`);
 }
}
```

NOTA: El HttpClientModule debe ser importado, lo haremos en el app.module.ts

## Nuevo Componente

Vamos a consumir el servicio, para mostrar el listado de Heroes, pero antes, necesitamos un componente para mostrar la información de un Heroe en específico,

```
ng g c heros/components/hero-card
```

Este componente recibirá la información de un Heroe (@Input)

```
@Component({
 selector: 'heros-hero-card',
 templateUrl: './hero-card.component.html',
 styleUrls: ['./hero-card.component.css']
})
export class HeroCardComponent implements OnInit{

 @Input()
 public hero!: Heroe;

 ngOnInit(): void {
 if(!this.hero){
 throw new Error('HeroCardComponent: Hero is required');
 }
 }
}
```

Su Template:

```
<mat-card>
 <mat-card-header>
 <mat-card-title>{{ hero.superhero }}</mat-card-title>
 <mat-card-subtitle>{{ hero.alter_ego }}</mat-card-subtitle>
 </mat-card-header>
 <mat-card-content class="mt-2">
 <h4>{{ hero.publisher }}</h4>
 <p>First apparition {{ hero.first_appearance }}</p>

 <mat-chip-listbox>
 <mat-chip *ngFor="let character of hero.characters.split(',') | slice:0:3">
 {{ character }}
 </mat-chip>
 </mat-chip-listbox>
 </mat-card-content>
 <mat-divider></mat-divider>

 <mat-card-actions>
 <button
 mat-button
 mat-raised
 color="primary"
 [routerLink]="/heroes/edit", hero.id]>
 <mat-icon>edit</mat-icon>
 View
 </button>

 <button>
```

```
mat-button
mat-raised
color="primary"
[routerLink]=["/heroes", hero.id]>
<mat-icon>more_horiz</mat-icon>
More
</button>
</mat-card-actions>
</mat-card>
```

Luego necesitamos mostrar esta información, lo haremos en el **ListPageComponent**

```
@Component({
 selector: 'app-list-page',
 templateUrl: './list-page.component.html',
 styleUrls: ['./list-page.component.css'
})
export class ListPageComponent implements OnInit{

 public heroes: Heroe[] = [];

 constructor(private heroesService: HeroesService) { }
 ngOnInit(): void {
 this.heroesService.getHeroes().subscribe(heroes => this.heroes =
heroes);
 }

}
```

Y finalmente el Template:

```
<h1>Heroes List</h1>

<mat-divider></mat-divider>

<div class="grid mb-8 pt-2">
 <div
 *ngFor="let hero of heroes"
 class="col-12 sm:col-4 md:col-3 xl:col-2">

 <heros-hero-card [hero]="hero"></heros-hero-card>
 </div>
</div>
```

The screenshot shows a mobile application interface titled "Heroes List". The top navigation bar includes a menu icon and a "Logout" button. The main content area displays a grid of hero cards, each containing the hero's name, origin, first appearance, and a list of aliases. Each card also features a "View" and "More" button.

Hero	Origin	First Appearance	Aliases	Action Buttons
Batman	DC Comics	Detective Comics #27	Bruce Wayne	View
Superman	DC Comics	Action Comics #1	Kal-El	View
Flash	DC Comics	Flash Comics #1	Jay Garrick	View
Green Lantern	DC Comics	All-American Comics #16	Alan Scott	View
Green Arrow	DC Comics	More Fun Comics #73	Oliver Queen	View
Wonder Woman	DC Comics	All Star Comics #8	Princess Diana	View

## Mostrar la imagen del heroe.

Los archivos de cada heroe lo tenemos almacenados en el directorio:

```
src/assets/heros/
```

El nombre del archivo es el `hero.id` y todos tienen una extensión jpg. Este es un URL válido a una de las imágenes

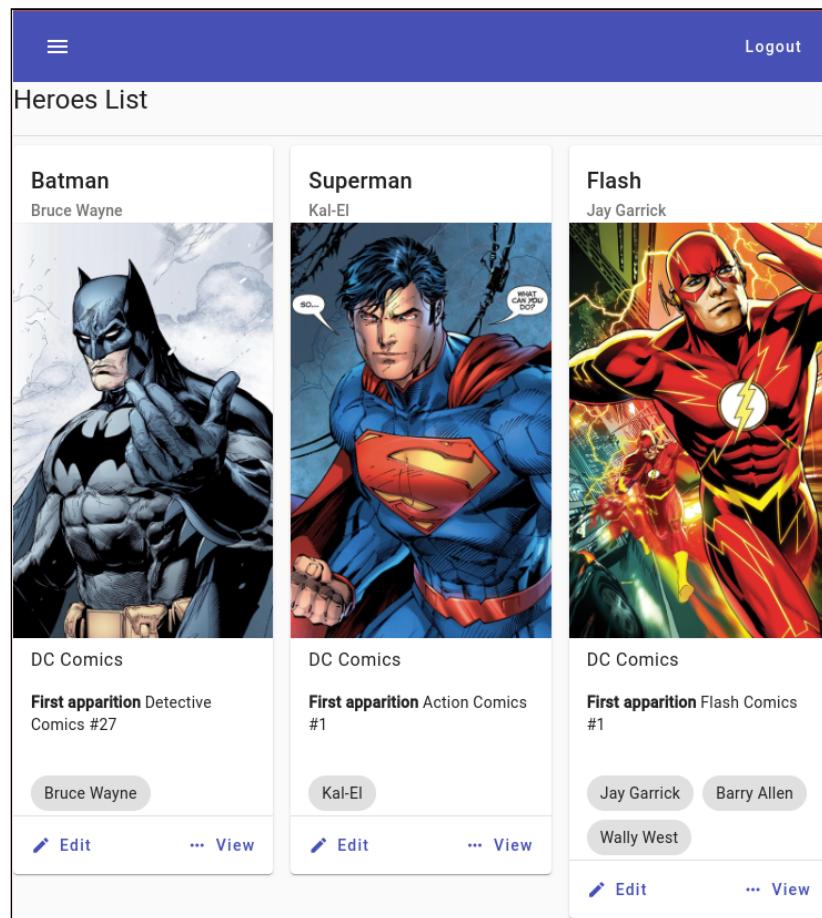
```
http://localhost:4200/assets/heros/dc-superman.jpg
```

Podríamos aregar al cardComponent, en el template el siguiente código:

```

```

Ahora ya podemos ver la imagen del Heroe en nuestro Card:



otra opción sería agregar un PIPE, el cual recibe le `hero.id` y genera el URL de la imagen.

## Creando HeroImage Pipe

En la línea de comandos:

```
$ ng g p heros/pipes/hero-image --flat --skip-tests
CREATE src/app/heros/pipes/hero-image.pipe.ts (223 bytes)
UPDATE src/app/heros/heros.module.ts (1043 bytes)
```

```
@Pipe({
 name: 'heroImage'
})
export class HeroImagePipe implements PipeTransform {
```

```
transform(hero: Heroe): string {
 if (!hero.id && !hero.alt_img) {
 return 'assets/no-image.png';
 }

 if (hero.alt_img) {
 return hero.alt_img;
 }
 return `assets/heroes/${hero.id}.jpg`;
}
```

Y ahora cambiamos nuestro template del **HeroCardComponent**

```

```

De esta forma, tendremos un único lugar donde cambiar en el caso de que nuestro directorio de imágenes cambie.

## Obtener Heroe

Agregamos un método para obtener la información de un heroe:

```
getHeroById(id: string): Observable<Heroe | undefined> {
 return this.httpClient.get<Heroe>(`${this.baseUrl}/heroes/${id}`)
 .pipe(catchError(err => of(undefined)));
}
```

El método **getHeroById** retorna un **Observable** que puede ser un heroes (si lo encuentra) o un **undefined**

Luego consumimos esto en el **HeroPageComponent**

```
@Component({
 selector: 'app-hero-page',
 templateUrl: './hero-page.component.html',
 styleUrls: ['./hero-page.component.css']
})
export class HeroPageComponent implements OnInit{

 public hero?: Heroe;

 constructor(private heroService: HeroesService,
 private activatedRoute: ActivatedRoute,
 private router: Router) { }

 ngOnInit(): void {
 /
```

```

 this.activatedRoute.params
 .pipe(
 switchMap(({id}) => this.heroService.getHeroeById(id))
).subscribe(hero => {
 if (!hero) {
 return this.router.navigateByUrl('/heroes/list');
 }
 this.hero = hero;

 return;
 });
 }
}

```

El constructor inyecta 3 servicios:

- **HeroesService** para obtener la información del Héroe
- **ActivatedRoute** para acceder al URL activo, y más específicamente a los parámetros, dado que necesitamos el ID del Héroe.
- **Router** para redireccionar al usuario en caso de que la búsqueda no retorne nada.

Toda la lógica está en el **ngOnInit**: Accedemos al parámetro ID mediante el **activatedRoute**, luego usamos el **switchMap** para generar otro Observable con el resultado de la búsqueda y finalmente nos suscribimos para analizar si se ha encontrado resultados o no.

Recuerden, esta implementación es incorrecta, ya que genera un **CallbackHell**, o un Subscribe dentro de otro Subscribe:

```

ngOnInit(): void {
 this.activatedRoute.params.subscribe(({id}) => {
 this.heroService.getHeroeById(id).subscribe(hero => {
 this.hero = hero;
 });
 });
}

```

Finalmente podemos trabajar en el template para mostrar información de un Héroe en específico.

## Autocomplete con Material

Iniciemos con la plantillas:

```

<div class="flex flex-column p-2">
 <h3>Searcher</h3>
 <mat-form-field>
 <mat-label>Hero Searcher</mat-label>
 <input

```

```

 type="text"
 matInput
 [FormControl]="searchControl"
 (input)="searchHero()"
 [matAutocomplete]="auto"/>

 <mat-autocomplete autoActiveFirstOption #auto="matAutocomplete">
 <mat-option *ngFor="let option of heros" [value]="option.id">
 {{option.superhero}}
 </mat-option>
 <mat-option
 *ngIf="heros.length === 0 && searchControl.pristine === false"
 value="">
 No Hero found!
 </mat-option>

 <mat-option
 *ngIf="heros.length === 0 && searchControl.pristine === true"
 value="">
 Search your hero!
 </mat-option>

 </mat-autocomplete>
 </mat-form-field>
 </div>

```

Estamos usando componentes de material para mostrar un texto con AutoComplete. El template necesita una lista de **heros** para llenar las opciones de acuerdo a los criterios de búsqueda.

Ademas controlamos el texto inicial **Search your hero!** el cual se muestra únicamente al inicio, una vez que el usuario ingresa cualquier término, se mostraran los resultados o el texto **No Hero found!** eso se controla con los datos en el arreglo **hero** y con la propiedad **pristine** la cual es TRUE solamente cuando el control se encuentra en su estado inicial.

En cuando al Componente, (TS code):

```

export class SearchPageComponent {

 public searchControl = new FormControl('');

 public heros: Heroe[] = [];

 constructor(private herosService: HeroesService) { }

 searchHero() {
 const value: string = this.searchControl.value || '';
 this.herosService.getSuggestions(value.trim())
 .subscribe(heros => this.heros = heros)
 }
}

```

Dado que estamos usando **ReactiveForms** de Angular, para enlazar nuestro **matInput** debemos importar el módulo **ReactiveFormsModuleModule**, lo haremos en el **HerosModule**

```
import { ReactiveFormsModule } from '@angular/forms';
```

Y lo agregamos a los **Imports**

Con esto ya tenemos un autoComplete,

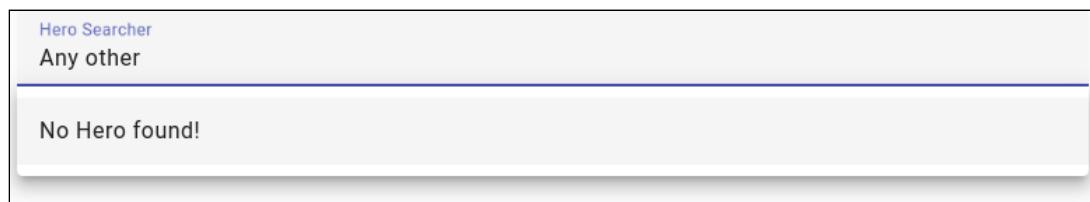
Estado inicial:



Tras buscar un obtener resultados



Sin resultados



# Nueva Sección: CRUD con Json-Server:

## ¿Qué veremos en esta sección?

Este es un breve listado de los temas fundamentales:

CRUD

Pipes puros e impuros

Snacks

Dialogs

Inyección de servicios manualmente

Esta sección les dará las bases para poder realizar CRUD completos hacia cualquier backend basado en servicios web, mediante comunicación JSON

## Agregando los Métodos al Servicio

Agregamos los métodos: Add/Update/Delete

```
addHeroe(heroe: Heroe): Observable<Heroe> {
 return this.httpClient.post<Heroe>(`${this.baseUrl}/heroes`, heroe);

updateHeroe(heroe: Heroe) : Observable<Heroe> {
 if (!heroe.id) throw new Error('Hero id is required to Update');
 return this.httpClient.put<Heroe>(`${this.baseUrl}/heroes/${heroe.id}`, heroe);

deleteHeroe(id: string): Observable<boolean> {
 return this.httpClient.delete(` ${this.baseUrl}/heroes/${id}`)
 .pipe(
 catchError(err => of(false)),
 map(resp => true));
}
```

En el caso del **deleteHeroe** el API retorna un arreglo vacío [] con status 200 cuando se borra un Heroe, y retorna un objeto vacío {} con status 404 cuando se intenta borrar un Heroe con ID inexistente, por lo tanto, en ese caso retornaremos un Observable con valor **true/false**

Para ello usamos el **catchError** para enviar un **false** o caso contrario enviamos un **true**

## Formularios Reactivos

Los formularios reactivos permite mover la mayor parte de la lógica del lado del componente.

Vamos a usar un Formulario Reactivo para enlazar nuestro formulario, en el componente **NewPageComponent** creamos este objeto:

```
public heroForm = new FormGroup({
 id: new FormControl<string>(''),
 superhero: new FormControl<string>('', {nonNullable: true}),
 publisher: new FormControl<Publisher>(Publisher.DCComics),
 alter_ego: new FormControl<string>(''),
 first_appearance: new FormControl<string>(''),
 characters: new FormControl<string>(''),
 alt_img: new FormControl<string>('')
});
```

**heroForm** es de tipo **FormGroup** el cual acepta un objeto cuyas propiedades son los campos de nuestro formulario, cada propiedad se define con el tipo **FormControl**, al mismo se le puede indicar el tipo de datos y algunas validaciones adicionales.

Ahora vamos a conectar el TemplateHTML con el formulario.

En resumen el Template del **NewPageComponent** tiene estas secciones

```
<div class="grid">
 <div class="col-12 sm:col-6 mt-2">
 <mat-card>
 <mat-card-content>
 <div class="grid">
 <!-- Input Super Hero-->
 <!-- Input Alter Ego-->
 <!-- Input First Appearance-->
 <!-- Input Characters-->
 <!-- Input Publisher-->
 </div>
 <!-- Buttons Sections-->
 </mat-card-content>
 </mat-card>
 </div>

 <!-- Second Column with the Image -->
</div>
```

Vamos a cambiar el

que agrupa los controles (Input) por un **<form class="grid" [formGroup]="heroForm">**

Esto enlaza nuestro Form del Template con el FormGroup del Componente. Pero debemos hacer lo mismo a nivel de Campos.

Por ejemplo, el campo **Super Hero**

```
<mat-label>Super Hero</mat-label>
<input type="text"
matInput
placeholder=""
required="">
```

A dicho Input le agregaremos:

```
formControlName="superhero"
```

Esto enlaza el Input **superhero** con el Campo del mismo nombre que definimos en nuestro FormGroup a nivel del componente. Lo mismo aplicamos para los otros Input.

En el caso del Publisher, es un control tipo **mat-select**, tal como si fuese un control más, agregamos el **formControlName**

```
<mat-select
required=""
formControlName="publisher">
<mat-option
*ngFor="let publisher of publishers"
[value]="publisher.id">
{{ publisher.desc}}
</mat-option>
</mat-select>
```

Ahora creamos un método

```
onSubmit() {
 let formValid = this.heroForm.valid;
 let value = this.heroForm.value;
 let rawValue = this.heroForm.getRawValue();

 console.log(rawValue);
}
```

Y lo enlazamos en nuestro formulario

```
<form class="grid" [formGroup]="heroForm" (ngSubmit)="onSubmit()">
```

Dado que nuestro botón de guardar lo hemos dejado fuera del formulario, tenemos que agregar tambien el onSubmit

```
<button mat-flat-button color="primary" (click)="onSubmit()">
 <mat-icon>save</mat-icon>
 Save
</button>
```

Si hacemos click en el botón **save** en consola veremos este objeto:

```
{
 "id": "",
 "superhero": "",
 "publisher": "DC Comics",
 "alter_ego": "",
 "first_appearance": "",
 "characters": "",
 "alt_img": ""
}
```

Además el **this.heroForm.valid** es *false* dado que no se han agregado los valores para los campos requeridos.

## Geter dentro del Componente

El valor actual del **this.heroForm.value** aunque es muy similar a la interfaz de Heroe, realmente no es del mismo tipo, podemos crear un metodo GET, para retornar el valor del Heroe mostrado en el formulario.

```
get currentHero(): Heroe {
 return this.heroForm.value as Heroe;
}
```

Luego podemos usar este valor para mostrar la imagen con el **heroImage** pipe

```

```

Recordemos que el PIPE mencionado usa un default Image

```
if (!hero.id && !hero.alt_img) {
 return 'assets/no-image.png';
}
```

Por esta razón vemos el placeholder de la imagen:

The screenshot shows a英雄编辑界面 (Edit / Add Hero Superman). The interface includes a sidebar with navigation icons and a main content area. The main content area contains a form with the following fields:

- Super Hero\* (必填)
- Alter Ego\*
- First Appearance\* (必填)
- Characters\*
- Creators\* (必填)
  - DC Comics
- Alt-img

At the bottom of the form are two buttons: "Delete" (red) and "Save" (blue).

## Completar el OnSubmit

Apliquemos los siguientes cambios en el OnSubmit

```
onSubmit() {
 if (this.currentHero.id) {
 this.heroService.updateHeroe(this.currentHero)
 .subscribe(resp => {
 // mostrar mensaje
 });
 return;
 }

 this.heroService.addHeroe(this.currentHero)
 .subscribe(resp => {
 // mostrar mensaje y navegar /heroes/list
 })
}
```

Con estos cambios, por el momento, podemos agregar un registro, dado que al cargar el formulario, siempre muestra un formulario vacío. De modo que el ID no ha sido asignado al Heroe.

## Carga un Heroe

Para cargar un Heroe seleccionado desde el Grid, vamos a agregar un cambio en el **NewPageComponent**

Durante la inicialización del componente, vamos a determinar si estamos cargando un heroe para editararlo, si es así, vamos a tomar de los params el ID del heroe y lo buscaremos con nuestro servicio.

Finalmente hacemos un reset del FORM y le asignamos el heroe encontrado, el código es el siguiente

```

ngOnInit(): void {
 if (!this.router.url.includes('edit')) return;

 this.activatedRoute.params
 .pipe(
 switchMap(({ id }) => this.heroService.getHeroeById(id)),
).subscribe(hero => {
 if (!hero) return this.router.navigate(['/heroes/list']);

 this.heroForm.reset(hero);
 });
}

```

Con este cambio, al hacer click sobre el botón **edit** de un heroe, se abre la pantalla en modo de edición.

## SnackBar

MatSnackBar es un servicio para mostrar notificaciones. Actualmente al actualizar un registro, no vemos ningún mensaje que confirme que se ha aplicado el cambios.

Como es un servicio, lo importaremos y lo inyectaremos en el constructor.

```

constructor(
 private snackbar: MatSnackBar) { }

```

Creamos un método, ya que vamos a desplegar diferentes mensajes dependiendo de ciertas acciones en este formulario:

```

showSnackbar(message: string) {
 this.snackbar.open(message, 'ok!', {
 duration: 2500,
 horizontalPosition: 'center',
 verticalPosition: 'top'
 });
}

```

Y luego lo llamamos donde sea necesario. Por ejemplo al editar, si la acción fue realizada con éxito:

```

this.showSnackbar('Record updated successfully');
this.router.navigate(['/heros/list']);

```

## Dialog

Para eliminar un Heroe, necesitamos un confirmación, para ellos usaremos el componente de Material: Dialog.

Este componente renderiza un dialogo en pantalla, como un pop-up, a su vez dicho componente necesita otro, que es un componente que contiene la información a mostrar.

Crearemos un componente en el directorio shared/components/dialogs

```
ng g c shared/components/dialog/confirm
```

Dado que necesitamos pasarle datos al componente, como el titulo y el mensaje a mostrar, creamos una interfaz propia del componente:

Directorio: **shared/components/dialog/interfaces**

```
export interface ConfirmData {
 title: string;
 message: string;
}
```

Luego creamos el TS del componente:

```
import { Component, Inject } from '@angular/core';
import { MAT_DIALOG_DATA, MatDialogRef } from '@angular/material/dialog';
import { ConfirmData } from '../interfaces/confirm.interface';

@Component({
 selector: 'app-confirm',
 templateUrl: './confirm.component.html',
 styleUrls: ['./confirm.component.css']
})
export class ConfirmComponent {
 constructor(
 public dialogRef: MatDialogRef<ConfirmComponent>,
 @Inject(MAT_DIALOG_DATA) public data: ConfirmData,
) {}

 onNoClick(): void {
 this.dialogRef.close(false);
 }
 onConfirm(): void {
 this.dialogRef.close(true);
 }
}
```

Un par de apuntes, el constructor de ConfirmComponent tiene dos parámetros:

```
public dialogRef: MatDialogRef<ConfirmComponent>: MatDialogRef
```

Es un servicio proporcionado por Angular Material que representa una referencia a un cuadro de diálogo. En este caso, se utiliza para cerrar el cuadro de diálogo cuando sea necesario. **ConfirmComponent** es el tipo de componente asociado al cuadro de diálogo.

```
@Inject(MAT_DIALOG_DATA) public data: ConfirmData
```

Se utiliza para injectar datos en el cuadro de diálogo. **MAT\_DIALOG\_DATA** es una constante proporcionada por Angular Material que permite injectar datos en el cuadro de diálogo. **ConfirmData** es el tipo de datos que se espera.(*Interfaz*)

Finalmente el Template:

```
<h1 mat-dialog-title>{{ data.title }}</h1>
<div mat-dialog-content>
 <p>{{ data.message }}</p>
</div>
<div mat-dialog-actions>
 <button mat-button (click)="onNoClick()">Cancel</button>
 <button mat-button (click)="onConfirm()" cdkFocusInitial>Ok</button>
</div>
```

Listo, antes de continuar, debemos exportar el **ConfirmComponent** en nuestro **SharedModule**

Ahora es momento de usarlo, nos vamos al **NewPageComponent**, primero inyectamos un servicio en el Constructor:

```
private dialog: MatDialog
```

Luego agregamos un método para eliminar el Heroe:

```
onDeleteHero() {
 const confirmData: ConfirmData = {
 title: 'Delete Hero',
 message: `Are you sure you want to delete the hero
${this.currentHero.superhero}?`
 };
 const dialog = this.dialog.open(ConfirmComponent, {
 width: '500px',
 data: confirmData,
 });

 dialog.afterClosed()
```

```

.pipe(
 switchMap(resp => {
 if (resp)
 return this.heroService.deleteHeroe(this.currentHero.id);
 return of(false);
 })
).subscribe(resp => {
 if (!resp) {
 return;
 }

 this.showSnackbar('Record deleted successfully');
 this.router.navigateByUrl('/heros/list');
});

dialog.afterClosed()
.subscribe(resp => {
 if (resp) {
 console.log(resp);

 this.heroService.deleteHeroe(this.currentHero.id!)
 .subscribe(resp => {
 this.showSnackbar('Record deleted successfully');
 this.router.navigateByUrl('/heros/list');
 });
 }
});
}

```

Primero, **ConfirmData** es la interfaz que expone el componente **ConfirmDialog**, entonces debemos crear un objeto de este tipo y asignar los datos que se mostrarán en el Dialog:

Luego llamamos el método **open** del dialog y le pasamos la configuración, incluidos los datos.

Finalmente nos suscribimos al **afterClosed** y ejecutamos el **deleteHeroe**

El **afterClosed** retornara un valor True|False, dependiendo del botón presionado (OK, CANCEL), si es true, procedemos con la eliminación del Heroe.

Otra forma de implementar el **afterClose** seria:

```

dialog.afterClosed()
.pipe(
 filter((result: boolean) => result),
 switchMap(() => this.heroService.deleteHeroe(this.currentHero.id!)),
 filter((wasDeleted: boolean) => wasDeleted),
)
.subscribe(() => {
 this.showSnackbar('Record deleted successfully');
 this.router.navigateByUrl('/heros/list');
});

```

El resultado del AfterClosed puede ser:

True: Si presiona en OK False: Si presiona en Cancelar undefined: Si presiona fuera del dialog

Por lo tanto el primer filter, filtra únicamente los eventos TRUE.

Luego el switchMap ejecuta el deleteHeroe, el cual puede retornar: TRUE: Si fue eliminado o FALSE: si ocurrió un error.

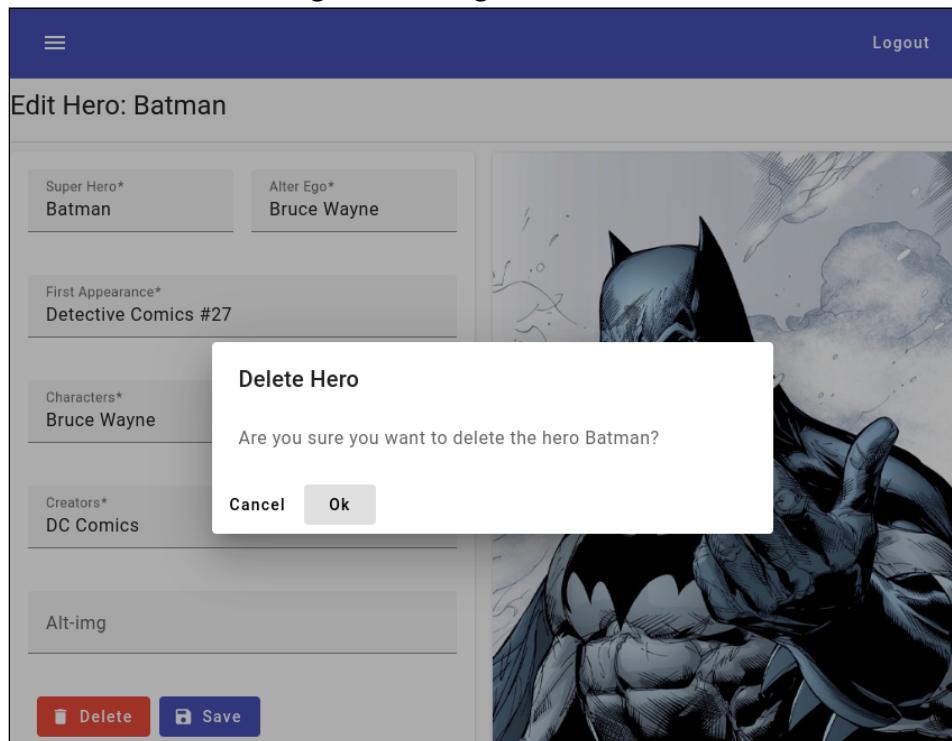
De nuevo se aplica un filtro para mantener en el pipe los eventos TRUE

Finalmente al suscribirse, lo único que puede llegar a escuchar son EVENTOS TRUE, es decir que efectivamente se eliminó el registro, por lo tanto, directamente manda el mensaje a pantalla y redirecciona al list.

NOTA: Para que esto funcione, hay que hacer una corrección en el deleteHeroe del servicio, el catchError debe estar al final del pipe. Es decir map -> catchError. Anteriormente el catchError era lo primero que se ejecutaba en el PIPE.

```
return this.httpClient.delete(` ${this.baseUrl}/heroes/${id}`)
.pipe(
 map(resp => true),
 catchError(err => of(false)));
```

El resultado final es el siguiente dialog al momento de eliminar un heroe:



# Nueva Sección: Guards (Protección de Rutas):

## ¿Qué veremos en esta sección?

Este es un breve listado de los temas fundamentales:

- Protección de ruta-
- Rutas privada-
- Rutas pública-
- Servicio de autenticació-
- Guard-
- Can Activat-
- Can Matc-
- Mantener la sesión del usuario

Esta es una sección muy importante, donde controlaremos de forma básica una autenticación, mucho más adelante en el curso, realizaremos autenticación mediante JWT, pero antes de llegar a eso que son temas de Backend, necesito que comprendamos cómo Angular nos puede servir para proteger nuestras rutas.

## Fake Login

Para mostrar la utilidad de los Guards, vamos a simular el login/ logout del usuario, no vamos a autenticarlo por el momento. Usaremos el siguiente servicio dentro del módulo de Autenticación

```
@Injectable({providedIn: 'root'})
export class AuthService {

 private baseUrl: string = environments.baseUrl;

 private user?: User;

 constructor(private httpClient: HttpClient) { }

 get currentUser(): User | undefined {
 if (!this.user) return undefined;

 return structuredClone(this.user);
 }

 login(email: string, password: string): Observable<User> {

 return this.httpClient.get<User>(`${this.baseUrl}/users/1`)
 .pipe(
 tap(user => this.user = user),
 tap(user => localStorage.setItem('token', user.id.toString())),
)
 }
}
```

```

 logout(): Observable<boolean> {
 this.user = undefined;
 localStorage.clear();
 return of(true);
 }
}

```

Lo importante aca es que cuando se llame el método **login** se devolverá siempre el user con ID 1, notar que no estamos validando user ni password. Adicionalmente, guardaremos en el localStorage, en el UserID

También agregamos el **logout** que elimina el Token del localStorage y hace undefined el usuario previamente "logeado"

El Login lo usaremos en el **LoginPageComponent**

```

onLogin() {
 this.authService.login('admin@gmail.com', '123456')
 .subscribe(user => {
 this.router.navigate(['/']);
 })
}

```

Y el Logout lo tenemos que usar en el **LayoutPageComponent** ya que este muestra el botón logout en la Header.

```

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

onLogout() {
 this.authService.logout()
 .subscribe(() => {
 this.router.navigate(['/auth/login']);
 });
}

```

Listo, con estos métodos tenemos un fakeLogin que guarda el ID del usuario en el LocalStorage y lo elimina cuando hace Logout,

El objetivo es validar que el usuario esté logeado para permitirlo navegar en la APP, caso contrario debe ser redireccionado al *auth/login*

## Mostrar el usuario Logeado

Vamos a mostrar en el Header BAR el usuario Logeado, en el template del **LayoutPageComponent** agreagamos esto:

```
{{currentUser?.user | titlecase}}
```

Y definimo el método:

```
get currentUser(): User | undefined {
 return this.authService.currentUser;
}
```

Ahora, luego de logearnos, podemos ver el nombre del usuario en la barra



Pero si recargamos la página se pierde los datos de sesión y el nombre se borra, aunque, en nuestro localStorage tenemos la información del Token.

Lo que necesitamos es verificar que al recargar las páginas, debemos inspeccionar el localStorage, y si tenemos un Token guardado, debemos cargar la información de nuevo a la sesión.

## Verificar Sesión Activa

Vamos a usar un método en nuestro AuthService que valida si hay un Token

```
checkAuth(): Observable<boolean> {
 const token = localStorage.getItem('token');
 if (!token) return of(false);
 return this.httpClient.get<User>(`${this.baseUrl}/users/1`)
 .pipe(
 tap(user => this.user = user),
 map(user => !!user),
 catchError(() => of(false))
)
}
```

Este método hace dos cosas:

- Regresa FALSE si el token no existe
- Regresa TRUE y a la vez reasigna el this.user si existe el Token

NOTA: `!!user` regresa TRUE si el usuario fue encontrado o False caso contrario, es la negación de la negación.

Si el usuario está definido: `!user` Regresa False luego `!False` regresa True.

Si el usuario NO está definido: `!user` Regresa TRUE, luego `!True` regresa False

## Implementando el Guards

En Angular, los guards (o guardianes) son clases que implementan interfaces específicas para controlar la navegación en una aplicación. Estos guards se utilizan para proteger o permitir el acceso a ciertas rutas o componentes en función de ciertas condiciones.

Los Guards son servicios pero en lugar de inyectarlos en los constructores, se colocan en un lugar específico, los Routers.

NOTA: En Angular 17, se han deprecado varias interfaces de los Guards, aca se muestra un ejemplo tomado de la documentación de Angular.

Dentro del directorio *Auth* creamos el directorio *guards* y en el archivo *auth.guard.ts* creamos este código:

```
import { Injectable, inject } from '@angular/core';
import { ActivatedRouteSnapshot,
 CanActivateFn,
 CanMatchFn,
 Route,
 Router,
 RouterStateSnapshot,
 UrlSegment } from '@angular/router';
import { AuthService } from '../services/auth.service';
import { Observable, tap } from 'rxjs';

const checkAuthStatus = (): boolean | Observable<boolean> => {
 const authService: AuthService = inject(AuthService);
 const router: Router = inject(Router);

 return authService.checkAuth().pipe(
 tap((isAuthenticated) => {
 if (!isAuthenticated) {
 router.navigate(['/auth/login']);
 }
 })
);
};

export const canActivateAuthGuard: CanActivateFn = (
 route: ActivatedRouteSnapshot,
 state: RouterStateSnapshot
) => {
 return checkAuthStatus();
};

export const canMatchAuthGuard: CanMatchFn = (
 route: Route,
 segments: UrlSegment[]
) => {
 return checkAuthStatus();
};
```

Varias cosas por analizar:

- No necesitamos una clase para implementar los guards
- La función local **checkAuthStatus** inyecta el AuthService y el Router con el **inject**

Tanto el **canActivateGuard** como el **canMatchGuard** implementan las interfaces respectivas **CanActivateFn** y **CanMatchFn**, estas regresan un valor True o False dependiendo de lo que retorne el método **checkAuthStatus**

Un Valor True indica que se puede continuar con la navegación, un valor False indica lo contrario.

Luego debemos usar estos Guards, actualizamos nuestro **app.routing.ts**

```
//localhost:4200
const routes: Routes = [
 {
 path: 'auth',
 loadChildren: () => import('./auth/auth.module').then(m =>
 m.AuthModule),
 },
 {
 path: 'heros',
 loadChildren: () => import('./heros/heros.module').then(m =>
 m.HerosModule),
 canActivate: [canActivateAuthGuard],
 canMatch: [canMatchAuthGuard],
 },
 //Otras rutas aca
];
```

Ambos Guards entraran en servicio cuando el usuario intente navegar a cualquier ruta del módulo **heros**

Con este cambio, si el usuario está logeado puede navegar a los URL's, caso contrario es redireccionado al Login.

## Redirect en el Login

El caso contrario es: Cuando hay una sesión abierta y el usuario intenta abrir la página del login. Esto debe redireccionar el ./ya que el usuario, al ver el login View puede interpretar que ha cerrado sesión, y abandonar su dispositivo o estación de trabajo. Si otro usuario abre la APP en usando otro URL, podrá ver el contenido ya que el usuario anterior no cerró sesión.

Para lograr esto creamos una copia del **auth.guard.ts** y vamos a cambiar la condición en el método **checkAuthStatus** En lugar de:

```
return authService.checkAuth().pipe(
 tap((isAuthenticated) => {
 if (!isAuthenticated) {
 router.navigate(['/auth/login']);
 }
 })
);
```

Implementamos esto:

```
return authService.checkAuth().pipe(
 tap((isAuthenticated) => {
 if (isAuthenticated) {
 router.navigate(['/heros/list']);
 }
 }),
 map((isAuthenticated) => !isAuthenticated),
);
```

NOTA: Sin el `map((isAuthenticated) => !isAuthenticated)`, estaba enviado siempre al 404Page, esto porque el TAP hace un navigate, pero el método como tal estaba regresando un TRUE (todo ese return regresa algo, un TRUE). Al regresar un TRUE, intenta cargar el Auth/login y a la vez el Hero/Lis causando un problema interno de redirecionamiento.

El map retorna lo opuesto al **isAuthenticated** de modo que si está autenticado, retorna un False y la navegación al /Auth/Login se cancela.

Otros cambios en el Guard, es que debemos renombramos los métodos **canActivateAuthGuard** y **canMatchAuthGuard** a **canActivatePublicGuard** y **canMatchPublicGuard** respectivamente y luego agregamos los nuevos Guards al AppRoutingModule en el path del login:

```
const routes: Routes = [
 {
 path: 'auth',
 loadChildren: () => import('./auth/auth.module').then(m =>
 m.AuthModule),
 canActivate: [canActivatePublicGuard],
 canMatch: [canMatchPublicGuard],
 },
 {
 path: 'heros',
 loadChildren: () => import('./heros/heros.module').then(m =>
 m.HerosModule),
 canActivate: [canActivateAuthGuard],
 canMatch: [canMatchAuthGuard],
 }
 ...
];
```

De esta forma, si el usuario está logeado, y trata de cargar el loginView será redireccionado al **/heros/list**, si no está logeado, podrá ver normalmente la página de logeo.

# Nueva Sección: Formularios Reactivos:

## ¿Qué veremos en esta sección?

Este es un breve listado de los temas fundamentales:

- Formularios Reactivos
- Lazyload y tareas relacionadas
- Validaciones propias de Angular
- Validaciones personalizadas
- Validaciones asíncronas
- Arreglos y objetos anidados
- FormBuilder
- FormGroup
- FormArray

## Nueva APP

Creamos una nueva app

```
ng new 06-formsApp --standalone --routing
```

Usaremos bootstrap, pegamos el link en nuestro index

```
<link
 href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
 rel="stylesheet" integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/DwykC2MPK8M2HN"
 crossorigin="anonymous">
```

Editamos el archivo *package.json* y agreamos el parámetro **-o** al script start

```
"start": "ng serve -o",
```

Creamos tres módulos

```
$ ng g m auth --routing
CREATE src/app/auth/auth-routing.module.ts (247 bytes)
CREATE src/app/auth/auth.module.ts (272 bytes)

$ ng g m reactive --routing
CREATE src/app/reactive/reactive-routing.module.ts (251 bytes)
```

```
CREATE src/app/reactive/reactive.module.ts (288 bytes)
```

```
$ g m shared
CREATE src/app/shared/shared.module.ts (192 bytes)
```

El ShareModulo lo agregamos al AppModule porque vamos a requiereir usar los componentes del Share en nuestra APP. Ademas es el único módulo sin Routing, porque solo va a proporcionar Componentes como utilidades.

En cambio Auth y Reactive, si van a tener sus enrutadores, porque haremos una carga Lazy y estos deben de agregar rutas Hijas. Generalmente, los Modulos que exponen rutas, contiene Componentes tipo Páginas, y estas agrupan otros componentes que pueden ser propios del módulo, o que vengan de Shared u otros módulos de terceros como Material.

## Creando estructura de Directorios y paginas

- Creamos dos dorectorios **pages** dentro de cada módulo: **Auth y Reactive**
- Creamos Los componentes Auth/register, Reactive/basic, Reactive/dinamic y Reactive/switch, los directorios quedarán así:

```
[fcruz@fedora app]$ tree
.
├── app.component.css
├── app.component.html
├── app.component.spec.ts
├── app.component.ts
├── app.module.ts
├── app-routing.module.ts
└── auth
 ├── auth.module.ts
 ├── auth-routing.module.ts
 └── pages
 └── register-page
 ├── register-page.component.css
 ├── register-page.component.html
 └── register-page.component.ts
└── reactive
 └── pages
 ├── basic-page
 │ ├── basic-page.component.css
 │ ├── basic-page.component.html
 │ └── basic-page.component.ts
 ├── dinamic-page
 │ ├── dinamic-page.component.css
 │ ├── dinamic-page.component.html
 │ └── dinamic-page.component.ts
 └── switch-page
 ├── switch-page.component.css
 ├── switch-page.component.html
 └── switch-page.component.ts
```

```

| └── reactive.module.ts
| └── reactive-routing.module.ts
└── shared
 └── shared.module.ts

```

En nuestro AppRoutingModule creamos las rutas Roots:

```

const routes: Routes = [
 {
 path: 'reactive',
 loadChildren: () => import('./reactive/reactive.module').then(m =>
 m.ReactiveModule)
 },
 {
 path: 'auth',
 loadChildren: () => import('./auth/auth.module').then(m =>
 m.AuthModule)
 },
 {
 path: '**',
 redirectTo: 'reactive'
 }
];

```

Agregamos los ChildRoutes para el módulo de Reactive

```

const routes: Routes = [
 {
 path: '',
 children:[
 { path: 'basic', component: BasicPageComponent},
 { path: 'dinamic', component: DinamicPageComponent},
 { path: 'switch', component: SwitchPageComponent},
 { path: '**', redirectTo: 'basic'}
]
 }
];

```

Y Hacemos lo mismo con el módulo de Auth.

```

const routes: Routes = [
 { path: 'singup', component: RegisterPageComponent },
 { path: '**', redirectTo: 'singup' }
];

```

NOTA: Cada uno de estos componentes, tiene un template ya diseñado, el cual puede ser descargado del repo del curso.

### [Repo del Curso Oficial](#)

o del repo personal:

### [Personal Repo](#)

## Barra Lateral

Vamos a crear un componente en el directorio shared/components

```
ng g c shared/components/side-menu --skip-tests
```

Lo exportamos en el SharedModule,

```
@NgModule({
 declarations: [
 SideMenuComponent
],
 imports: [
 CommonModule
],
 exports:[
 SideMenuComponent
]
})
export class SharedModule { }
```

En el **SideMenuComponent** agrearemos estos menuItems:

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

interface MenuItem {
 title: string;
 route: string;
}

@Component({
 selector: 'shared-side-menu',
 templateUrl: './side-menu.component.html',
 styleUrls: ['./side-menu.component.css'
})
export class SideMenuComponent {

 public reactiveMenu: MenuItem[] = [
 { title: 'Basicos', route: './reactive/basic' },
 { title: 'Avanzados', route: './reactive/avanzado' }
]
```

```

 { title: 'Dinamicos', route: './reactive/dinamic' },
 { title: 'Switches', route: './reactive/switch' }
];

public authMenu: MenuItem[] = [
 { title: 'Registro', route: './auth/register' }
];
}

```

Y el template:

```

<h2>Pages</h2>

<hr>

<h3 class="mt-2">Reactive</h3>
<ul class="list-group">
 <li
 *ngFor="let item of reactiveMenu"
 class="list-group-item"
 [routerLink]=[item.route]
 routerLinkActive="active">
 {{ item.title }}

<h3 class="mt-2">Validations</h3>

<ul class="list-group">
 <li
 *ngFor="let item of authMenu"
 class="list-group-item"
 [routerLink]=[item.route]
 routerLinkActive="active">
 {{ item.title }}


```

NOTA: para que el **[routerLink]** funcione dentro del */i* debemos importar el **RouterModule** en el **SharedModule**

Ahora debemos crear nuestro Layout principal, en el app.component.html

```

<div class="row mt-5">
 <div class="col">
 <router-outlet></router-outlet>
 </div>

 <div class="col-12 col-sm-4">
 <shared-side-menu></shared-side-menu>
 </div>

```

```
</div>
</div>
```

Listo, al cargar la app podemos ver el url **http://localhost:4200/reactive/basic** porque es esta nuestra página por default? Eso se debe a las rutas. En nuestro Root Router (app.routing.ts) hemos dicho que cualquier path **[\*\*]** sera redireccionado al reactive.

**Path Inicial:** **http://localhost:4200/**

```
{
 path: '**',
 redirectTo: 'reactive'
}
```

**Path redireccionado** **http://localhost:4200/reactive/**

y el path **reactive** carga en modo Lazy el módulo **ReactiveModule**

```
{
 path: 'reactive',
 loadChildren: () => import('./reactive/reactive.module').then(m =>
m.ReactiveModule)
}
```

Y cuando cargamos este módulo, cargamos su reglas de ruteo. Y aca hemos definido que cualquier path que no haga match, será redireccionado al **basic**

**Path Entrada al módulo:** **http://localhost:4200/reactive/**

```
{ path: '**', redirectTo: 'basic'}
```

Y el path **basic** carga el **BasicPageComponent**

```
{ path: 'basic', component: BasicPageComponent},
```

**Path final:** **http://localhost:4200/reactive/basic**

Resultado final:

The screenshot shows a web application interface. On the left, there's a main content area titled "Reactivos: Básicos". It contains three input fields with validation messages:

- Nombre del producto: "error message"
- Precio del producto: "El precio debe de ser 0 o mayor"
- Existencias del producto: "Las existencias deben de ser 0 o mayor"

On the right, there's a sidebar titled "Pages" with a "Reactive" section containing "Basicos", "Dinamicos", and "Switches". Below that is another section titled "Validations" with "Registro". At the bottom of the sidebar is a blue "Guardar" button.

## Primer Form

En el componente **BasicPageCompoenent** agregaremos un Formulario reactivo, hay dos formas de hacerlo

Usando el **FormGroup**:

Primero importaremos el módulo **ReactiveFormsModule** en **ReactiveModule** y luego en nuestro componente **BasicPageComponent** agregamos el siguiente código:

```
public productForm: FormGroup = new FormGroup({
 name: new FormControl(''),
 price: new FormControl(0),
 quantity: new FormControl(0)
});
```

La otra forma es usando un  **FormBuilder**, para ello necesitamos injectar un servicio

```
public productForm: FormGroup = this.formBuilder.group({
 name: [''],
 price: [0],
 quantity: [0]
});
```

Esta forma no requiere que se importe el **ReactiveFormsModule** pero usaremos muchas características de los formularios reactivos que lo vamos a mantener en el Import.

## Enlazar el HTML con el productForm

Para enlazarlo usaremos la directiva **[formGroup]** que precisamente viene del **ReactiveFormsModule** y le asignaremos el nombre del objeto que definimos en TS.

```
<form [formGroup]="productForm" autocomplete="off">
```

NOTA: **autocomplete** OFF indica que nosotrosaremos las validaciones de nuestro lado.

Creamos un método:

```
onSubmit() {
 console.log(this.productForm.value);
}
```

Y lo usamos en el click del botón que hace el submit de nuestro formulario

```
<button type="submit"
 (click)="onSubmit()"
 class="btn btn-primary float-end">
 Guardar
</button>
```

O mejor aún usamos la directiva **ngSubmit**

```
<form
 [formGroup]="productForm"
 (ngSubmit)="onSubmit()"
 autocomplete="off">
```

Si aún agregando datos, al presionar **Guardar** veremos un objeto sin valores en sus propiedades.

```
{
 "name": "",
 "price": "",
 "quantity": ""
}
```

## Concretar campos de los formularios.

Usaremos la directiva **formControlName** y le asignaremos el nombre de la propiedad, por ejemplo, esta sería **name** para el Nombre del producto.

```
<input formControlName="name"
 type="text"
 class="form-control"
 placeholder="Nombre del producto">
```

Repetimos el mismo proceso para los demás campos del formulario. (Price y quantity) Ahora podemos ver los valores iniciales en cada campo. Además, al agregar datos a los campos y luego presionar **Guardar** vemos en consola:

```
{
 "name": "PS5",
 "price": 750,
 "quantity": 1
}
```

## FormValidator

Angular ya incluye validadores, podemos agregar un conjunto de reglas por campo para facilitar las validaciones, por ejemplo, el nombre del producto:

- Debe ser requerido
- Debe contener al menos 3 caracteres
- Y un máximo de 20

Estas validaciones las agregamos al FormGroup, actualmente solo definimos un valor inicial sin validaciones. Las validaciones se agregan en el arreglo, como 2do y 3er parametro.

```
public productForm: FormGroup = this.formBuilder.group({
 name: [''],
 price: [0],
 quantity: [0]
});
```

En el caso del Name, agregamos:

```
name: ['', [Validators.required, Validators.minLength(3),
 Validators.maxLength(20)]],
```

Aplicamos un par de reglas para los campos price y quantity y probamos el Submit.

Al hacer el submit, simplemente vemos el objeto en el console.log, esto porque no estamos validando el estado del formulario, si agregamos esta condición, los datos no se imprimen

```
onSubmit() {
 if (this.productForm.invalid) {
 this.productForm.markAllAsTouched();
 return;
 }
 console.log(this.productForm.value);
}
```

nota: `this.productForm.markAllAsTouched();` permite mostrar todos los errores en pantalla que tenga el formulario en ese momento. Sin esa instrucción, el formulario es invalido, no ejecuta el Submit, pero tampoco se muestran los errores.

## Mostrar errores en pantalla

El manejo de los errores es importante, debemos proporcionar información exacta al usuario cuando ocurre un error.

Hay varias formas de mostrar un error en el control, cuando no se cumple una condición, por ejemplo, en el caso del nombre del producto tenemos dos reglas

- Requerido
- Ancho mínimo de 3 caracteres

Podemos implementarlos con el `*ngIf`

```
<input formControlName="name"
 type="text"
 class="form-control"
 placeholder="Nombre del producto">
<span *ngIf="productForm.controls['name'].getError('required')"
 class="form-text text-danger">
 Este campo es requerido.

<span *ngIf="productForm.controls['name'].getError('minlength')"
 class="form-text text-danger">
 Este campo requiere al menos 3 caracteres.

```

Como se puede observar accedemos al error específico usando el método `getError` de control.

Si probamos el cambio, veremos el primer error, cuando el campo está vacío, al agregar el primer carácter, se oculta el primero error y se muestra el segundo, y hasta que agregamos el 3 carácter, el control no mostrará ningún error.

Un inconveniente de usar esta forma es que el error se muestra de entrada cuando vamos a agregar un nuevo producto, porque el campo está vacío, podríamos agregar una condición más:

```
<span *ngIf="productForm.controls['name'].getError('required')
 && productForm.controls['name'].touched" class="form-text text-
danger">
```

Esto empieza a ser tedioso, imagina un formulario con 10 campos, y cada campo con al menos 1 o 2 validaciones, y uno que otro con más de 4 validaciones. Definitivamente no es para nada eficiente.

## Métodos de ayuda de Errores

Otra forma es crear método que ayuden a detectar errores, por ejemplo

```
isValidField(field: string): boolean | null {
 return this.productForm.controls[field].errors
 && this.productForm.controls[field].touched;
}
```

Luego podemos cambiar nuestro template a algo mas sencillo:

```

 Este campo es requerido.

```

Pero esto no esta del todo bien, porque no estamos evaluando un error específico, por lo tanto no vamos a poder mostrar un mensaje adecuado.

Creemos otro método:

```
getFieldError(field: string): string | null {

 if (!this.productForm.controls[field]) {
 return null;
 }
 const errors = this.productForm.controls[field].errors || {};
 for (const key in errors) {
 if (errors.hasOwnProperty(key)) {
 switch (key) {
 case 'required':
 return 'This field is required';
 case 'minlength':
 return `The minimum length is ${errors[key].requiredLength}`;
 case 'maxlength':
 return `The maximum length is ${errors[key].requiredLength}`;
 default:
 return null;
 }
 }
 }
 return null;
}
```

Ahora en lugar de dos **span** podemos tener uno solo con el nuevo método.

```
<input formControlName="name"
 type="text"
 class="form-control"
```

```

placeholder="Nombre del producto">

 {{ getFieldError('name')}}


```

Esto está un poco mejor pero aún necesita trabajo, porque podemos dejar por fuera de las validaciones algún tipo de error, por ejemplo el valor min que hemos establecido en precio

```
price: [0, [Validators.required, Validators.min(0)]] ,
```

Si agregamos un valor negativo, no vemos el error. Tendríamos que agregar otro CASE

```

case 'min':
 return `The minimum value is ${errors[key].min}`;

```

Esto lo mejorarímos con un servicio de validaciones más adelante.

## Formularios Dinámicos

No sabemos de antemano cuantos elementos tendrá el formulario, podemos agregar controles dinámicamente y cada control debe tener ciertas validaciones.

Para nuestro formulario dinámico iniciemos creando el código TS.

```

export class DynamicPageComponent {

 constructor(private formBuilder: FormBuilder) { }

 public form = this.formBuilder.group({
 name: ['', [Validators.required, Validators.minLength(3)]],
 favoriteGames: this.formBuilder.array([
 ['Metal Gear', Validators.required],
 ['Final Fantasy', Validators.required],
 ['The Witcher', Validators.required]
])
 });

 get favoriteGames() {
 return this.form.get('favoriteGames') as FormArray;
 }

 onSubmit() {
 if (this.form.invalid) {
 this.form.markAllAsTouched();
 return;
 }
 this.form.reset();
 }
}

```

```

 }
}
```

En el template enlazamos el formulario:

```
<form [formGroup]="form" (ngSubmit)="onSubmit()" autocomplete="off">
```

Y enlazamos los campos iniciales, el Name:

```
<input formControlName="name"
 class="form-control"
 placeholder="Nombre de la persona">
```

Para la parte dinámica, tenemos originalmente este HTML

```
<!-- Lista de Juegos Favoritos -->
<div class="mb-3 row">
 <label class="col-sm-3 col-form-label">Favoritos</label>
 <div class="col-sm-9" >

 <!-- Duplicar este bloque por cada elemento dinamico-->
 <div class="mb-1">
 <div class="input-group">
 <input class="form-control">
 <button class="btn btn-outline-danger" type="button">
 Eliminar
 </button>
 </div>

 Este campo es requerido

 </div>
 </div>
</div>
```

Primero necesitamos

Y luego agregar un ngFor para renderizar los elementos de nuestro arreglo de controles, es decir **favoriteGames** de nuestro **form** para ellos usaremos nuestro metodo **get favoriteGames**

```
<div class="col-sm-9" formArrayName="favoriteGames">
 <!-- Duplicar este bloque por cada elemento dinamico-->
 <div class="mb-1" *ngFor="let favorite of favoriteGames.controls; let i = index">
```

```
<div class="input-group">
 <input [formControlName]="i" class="form-control">
 <button class="btn btn-outline-danger"
 type="button">
 Eliminar
 </button>
</div>

 Este campo es requerido

</div>
</div>
```

Notar que usamos un `ngFor` para renderizar dinámicamente los controles de nuestro array.

Luego vemos que para acceder al nombre del control, usamos el index del arreglo

```
<input [formControlName]="i"
```

Con esto ya estaremos renderizando 3 controles, pero aun falta trabajar con las validaciones dinámicas.

# Validaciones Dinámicas

Vamos a copiar (por ahora) las funciones que usamos en el **BasicPageComponent** y agregaremos un método más para validar los controles de un array.

```

 case 'minlength':
 return `The minimum length is ${errors[key].requiredLength}`;
 case 'maxlength':
 return `The maximum length is ${errors[key].requiredLength}`;
 case 'min':
 return `The minimum value is ${errors[key].min}`;
 default:
 return null;
 }
}
return null;
}

```

Para validar nuestro campo Name, el cual no es un arreglo usamos el método anterior

```

<div class="col-sm-9">
 <input formControlName="name"
 class="form-control"
 placeholder="Nombre de la persona">

 {{ getFieldError('name') }}

</div>

```

Y en el caso del error que se renderiza en el arreglo, usamos el nuevo método, le pasamos el favoriteGames y el index para indicar que elemento del array queremos validar.

```


 Este campo es requerido.


```

Por el momento dejaremos el mensaje de error tal como se ve en la plantilla.

## Eliminar Controles dinámicos

Simplemente basta con remover el control del arreglo

TS:

```

onDelete(index: number) {
 this.favoriteGames.removeAt(index);
}

```

Y luego lo usamos en el template.

```
<button (click)="onDelete(i)" class="btn btn-outline-danger"
 type="button">
 Eliminar
</button>
```

## Agregar Controles dinámicos

Aca también necesitamos un método para agregar, pero dado que tenemos en el template un input Box para capturar el contenido del control, demos agregar algo al template:

El template luce de esta forma:

Reactivos: Dinámicos		Pages
Nombre	<input type="text" value="Nombre de la persona"/>	
Agregar	<input type="button" value="Agregar favorito"/>	<input style="background-color: #007bff; color: white; border: none; border-radius: 3px; padding: 2px 10px; font-weight: bold; margin-left: 10px;" type="button" value="Agregar favorito"/>
Favoritos	<input type="text" value="Metal Gear"/> <div style="border: 1px solid red; padding: 2px; margin-left: 10px;"> <input style="border: none; background-color: red; color: white; padding: 2px 10px; font-size: small;" type="button" value="Eliminar"/> </div> <input type="text" value="Final Fantasy"/> <div style="border: 1px solid red; padding: 2px; margin-left: 10px;"> <input style="border: none; background-color: red; color: white; padding: 2px 10px; font-size: small;" type="button" value="Eliminar"/> </div> <input type="text" value="The Witcher"/> <div style="border: 1px solid red; padding: 2px; margin-left: 10px;"> <input style="border: none; background-color: red; color: white; padding: 2px 10px; font-size: small;" type="button" value="Eliminar"/> </div>	<b>Reactive</b> <ul style="list-style-type: none"> <li>Basicos</li> <li style="background-color: #007bff; color: white; border: 1px solid #007bff; padding: 2px 10px; border-radius: 3px;">Dinamicos</li> <li>Switches</li> </ul> <b>Validations</b> <ul style="list-style-type: none"> <li>Registro</li> </ul>
		<input style="background-color: #007bff; color: white; border: none; border-radius: 3px; padding: 2px 10px; font-weight: bold;" type="button" value="Guardar"/>

La idea es que al presionar Agregar Favorito, se agregue un control mas a nuestro arreglo y este sea renderizado.

Dado que necesitamos validar que el usuario por lo menos agregue algo de información en el Input, podemos hacer un Control Reactivo aislado.

Es decir, en el TS agregamos esto

```
public newFavorite: FormControl = new FormControl(' ', [Validators.required,
Validators.minLength(3)]);
```

Adicionalmente creamos un **onAdd** método:

```
onAdd() {
 if (this.newFavorite.invalid) {
 return;
 }
 const newGameName = this.newFavorite.value;
 this.favoriteGames.push(this.formBuilder.control(newGameName,
Validators.required));
 this.newFavorite.reset();
}
```

Y este control lo enlazamos en nuestro template junto con la acción de Agregar un favorito.

```
<div class="input-group">
 <input [FormControl]="newFavorite" class="form-control"
 placeholder="Aregar favorito">
 <button (click)="onAdd()" class="btn btn-outline-primary"
 type="button">
 Agregar favorito
 </button>
</div>
```

Dado que es un control independiente, necesita su propio validador, podemos crear esta función:

```
isValidNewFavorite(): boolean | null {
 return this.newFavorite.invalid
 && this.newFavorite.touched;
}
```

Y agregamos el span del control:

```

 Valor incorrecto

```

De esta forma ya tenemos un formulario dinámico.

### Reactivos: Dinámicos

Nombre	Facuta 00001	
Agregar	Aregar favorito	<button>Agregar favorito</button>
Favoritos	Ps5	<button>Eliminar</button>
	RTX 4080	<button>Eliminar</button>
	Monitor 27	<button>Eliminar</button>
	Mouse	<button>Eliminar</button>
	Teclado	<button>Eliminar</button>
	Cooler	<button>Eliminar</button>
	Desktop	<button>Eliminar</button>
	Chair	<button>Eliminar</button>
		<button>Guardar</button>

Más información sobre formularios reactivos

<https://angular.io/guide/reactive-forms>

# Nueva Sección: Validaciones

## ¿Qué veremos en esta sección?

¿Qué veremos en esta sección?

Este es un breve listado de los temas fundamentales:

- Validaciones manuales
- Validaciones asíncronas
- Validar contra expresiones regulares
- Separar la lógica de validaciones
- Estado del formulario
- Mensajes de error personalizados

Más adelante tendremos una sección de directivas, la cual es un excelente complemento para el manejo de errores de formularios reactivos, pero luego llegarán a ella.

## FormControl en Radio Buttons

Supongamos que tenemos el siguiente formulario

```
public form: FormGroup = this.formBuilder.group({
 gender : ['M', Validators.required],
 wantsNotifications: [true, Validators.required],
 termsAndConditions: [false, Validators.requiredTrue]
});
```

El **gender** es un string M|F (Masculino | Femenino), el formulario HTML es representado como un **RadioButton**

**wantsNotifications** es un true|false, es un checkbox pero se le agrega la clase **form-switch** de bootstrap para convertirlo en un slice button.

Finalmente **termsAndConditions** es un checkbox normal con valores true|False

**Reactivos: Switches**

Género	<input checked="" type="radio"/> Masculino <input type="radio"/> Femenino
Notificaciones	<input checked="" type="checkbox"/> Quiero recibir notificaciones <input checked="" type="checkbox"/> Términos y condiciones de uso
<button>Guardar</button>	

Para los **RadioButtons**, usaremos el mismo FormControlName

```

<div class="form-check">
 <input formControlName="gender"
 class="form-check-input"
 type="radio"
 value="M"
 id="radioMasculino">
 <label class="form-check-label" for="radioMasculino">
 Masculino
 </label>
</div>
<div class="form-check">
 <input formControlName="gender"
 class="form-check-input"
 type="radio"
 value="F"
 id="radioFemenino">
 <label class="form-check-label" for="radioFemenino">
 Femenino
 </label>
</div>

```

En el caso del **wantsNotifications** y **termsAndConditions** se usan normalmente sus FormControlName respectivos.

Con respecto a las validaciones agregamos las funciones que hemos venido usando anteriormente, únicamente al campo de las Términos y Condiciones necesita validación.

```


 {{getFieldError('termsAndConditions')}}

```

Al hacer submit del formulario veremos el siguiente objeto:

```
{
 "gender": "F",
 "wantsNotifications": true,
 "termsAndConditions": true
}
```

**NOTA:** Si nuestro backend espera únicamente un objeto con el **gender** y el **wantsNotifications** dejando del lado del frontEnd la responsabilidad de aceptar los términos y condiciones, no debemos enviar este objeto, para ello podemos eliminar ese valor del request

```
const { termsAndConditions, ... newFormValue } = this.form.value;
```

En este punto podemos enviar únicamente el **newFormValue**

```
{
 "gender": "F",
 "wantsNotifications": true
}
```

## Formulario de Registro

Este es un típico formulario de registro, los errores se muestran en su estado inicial porque no hemos enlazado ningún formControl y no hemos realizados las validaciones requeridas en cada campo.

### Validaciones Reactivas

Nombre	<input type="text"/> Nombre del usuario Debe de ser en formato de nombre y apellido
Email	<input type="text"/> Email del usuario mensaje de error del email
Username	<input type="text"/> Nombre código del usuario El username no puede ser Strider
Password	<input type="text"/> Password de su cuenta La contraseña debe de ser mayor de 6 caracteres.
Confirmar	<input type="text"/> Confirmar la contraseña Las contraseñas deben de ser iguales

**Crear**

Lo primero es injectar el FormBuilder y luego crear nuestro Form

```
constructor(private FormBuilder: FormBuilder) { }

public form = this.formBuilder.group({
 name: ['', [Validators.required, Validators.minLength(6)]],
 email: ['', [Validators.required]],
 userName: ['', [Validators.required, Validators.minLength(6)]],
 password: ['', [Validators.required, Validators.minLength(8)]],
 password_confirmation: ['', [Validators.required]],
});
```

Estas son las validaciones básicas, vamos a agregar otras validaciones manuales, además, usaremos un servicio para proveer las validaciones.

Luego debemos importar el **ReactiveFormsModule** en el **\*AuthModule** para poder usar la etiquetas especiales de los ReactiveFormsModule en nuestro Html

Luego enlazamos nuestro forms en el template, tanto a nivel de group como a nivel de controles.

```
<form [formGroup]="form" (ngSubmit)="onSubmit()" autocomplete="off">
```

## Validator Functions

Podemos crear nuestras propias funciones de validaciones y reusarlas en cualquier formulario. Para ello podemos crearlas en nuestro módulo Shared para que estén disponibles. Podemos crearlas como un servicio o bien como simples funciones, este último será la forma en como implementaremos la siguiente validación.

Imagina que nos han dado el siguiente requerimiento, **El userName no puede ser ninguno de estos valores: admin, Administrator, root**

Podemos pensar en una función que reciba dos cosas

- Un Valor que no debe de tomar el control
- El control en sí

El parámetro **valor** podríamos cambiarlo por un arreglo, de esa forma podemos aceptar en una sola llamada a la función varios valores.

Y como estamos usando los Validators de Angular, podemos retornar de una vez un objeto de tipo **ValidationErrors**

```
/**
 * @description
 * Defines the map of errors returned from failed validation checks.
 *
 * @publicApi
 */
export declare type ValidationErrors = {
 [key: string]: any;
};
```

La función luce de la siguiente forma:

```
import { FormControl, ValidationErrors } from "@angular/forms";

export const cantbeThisValue = (control : FormControl, value: string[]) : ValidationErrors => {
 if (value.map (v =>
v.toLowerCase().trim()).includes(control.value.toLowerCase().trim())) {
 return {
 cantbeThisValue: true
 }
 }
 return {};
}
```

Dado que recibimos un arreglo de valores que el control no debe aceptar, debemos tratar cada posible valor del arreglo y transformarlo a un valor que pueda ser comparado, es decir aplicamos un **map** para eliminar espacios en blanco y para pasarlo a minúsculas. Lo mismo hacemos con el valor del control

Si el valor del control está incluid en el arreglo, entonces generamos un objeto de tipo **ValidationErrors** que no es más que un key:value.

```
return {
 cantbeThisValue: true
}
```

NOTA: Si retornamo un objeto vacío {} Angular no lo incluye en la lista de errores.

Luego debemos usar esta función, y lo haremos de esta forma: Al momento de crear nuestro **form** con el **FormBuilder** al especificar el campo **userName** podemos contruirlo de esta forma

```
userName: ['',
 [Validators.required,
 Validators.minLength(5),
 (control: FormControl) => cantbeThisValue(control, ['admin',
 'administrator', 'root'])
]
,
```

La parte que nos interesa en esta sección es esta línea

```
(control: FormControl) => cantbeThisValue(control, ['admin',
 'administrator', 'root'])
```

Dado que nuestra función recibe como primer argumento el Control, debemos usar una función anónima para pasar el control.

Normalmente, si la función no recibe parámetro adicionales, podemos hacer una llamada de este tipo

```
email: ['',
 [Validators.required,
 isValidEmail()
]
,
```

Como se observa, **isValidEmail** no se le pasa ningún parámetro, ya que implicitamente el control se pasa como primer argumento. Dentro de nuestra función **isValidEmail**, vamos a tener acceso al control y hacer la validación requerida.

Regresando a nuestro ejemplo, dado que debemos explicitamente enviar el **Control** `userName` y a la vez los valores no permitidos, debemos usar la función anónima para pasar el control en sí.

Con estos cambios, podemos probar las validaciones, si escribimos **admin** en el input `UserName`, veremos que el formulario es Inválido y el objeto `Errors` tiene este valor

```
{
 "cantbeThisValue": true
}
```

Luego podemos modificar la función `getFieldError` para agregar un nuevo Key `cantbeThisValue`

```
getFieldError(field: string): string | null {
 if (!this.productForm.controls[field]) {
 return null;
 }

 const errors = this.productForm.controls[field].errors || {};
 for (const key in errors) {
 if (errors.hasOwnProperty(key)) {
 switch (key) {
 case 'cantbeThisValue':
 return 'Incorrect value';
 // Other Keys here!
 default:
 return null;
 }
 }
 }
 return null;
}
```

## Validación del Email

Podemos usar expresiones regulares para validar ciertos patrones, por ejemplo, creamos la siguiente constante en nuestro archivo `validators.ts`

```
export const emailPattern: string = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,4}$";
```

Luego agregamos la validación al campo Email:

```
email: ['', [Validators.required,
 Validators.pattern(CustomValidators.emailPattern)]],
```

En el caso de no cumplir con el Pattern indicado, vamos a obtener el siguiente error:

```
{
 "pattern": {
 "requiredPattern": "^[a-z0-9._%+-]+@[a-z0-9.-]+\\.[a-z]{2,4}$",
 "actualValue": "email"
 }
}
```

También podríamos crear nuestra propia función y retornar un objeto con mejor información.

NOTA: he realizado un cambio en el import para permitir acceder a las funciones de nuestro validators.ts, con **CustomValidators.emailPattern** o **CustomValidators.cantbeThisValue**

```
import * as CustomValidators from '../../../../../shared/validators/validators';
```

## Servicio de Validaciones

Vamos a crear una carpeta **services** en el directorio **shared** y dentro de este nuevo directorio creamos el archivo: **validator.service.ts** creamos un nuevo servicio manualmente,

```
@Injectable({providedIn: 'root'})
export class ValidatorService {
```

Y dentro de este servicio vamos a pegar todas las funciones validadoras que hemos estado usando en las secciones anteriores.

- El email pattern
- El método cantbeThisValue

```
public cantbeThisValue = (control : FormControl, value: string[]) : ValidationErrors => {}
```

- Las funciones isValidField

```
public isValidField(form: FormGroup, field: string): boolean | null {}
```

- El `isValidField`, en este caso tenemos que pasar el `FormGroup`

```
public isValidField(form: FormGroup, field: string): boolean | null {}
```

- El `isValidArray`

```
public isValidFieldArray(formArray: FormArray, index: number): boolean | null {}
```

- Y el `GetFieldError`

```
public getFieldError(form: FormGroup, field: string): string | null {}
```

Luego debemos de hacer un par de cambios en el componente, en lugar de usar una función validadora local o importada de nuestro `validator.ts` ahora debemos inyectar el servicio

```
constructor(
 private formBuilder: FormBuilder,
 private validatorSercice: ValidatorService) { }
```

Removemos el Import anterior

```
import * as CustomValidators from '../../../../../shared/validators/validators';
```

Y donde nos marque error, usamos el servicio. Es decir en lugar de **CustomValidators.[METHOD]** lo reemplazamos por **this.validatorSercice.[METHOD]**

Debemos de mantener algunos de los métodos anteriores en nuestro componentes, porque estos siguen siendo usados por el template, por ejemplo el **isValidField** y el **getErrorMessage** pero estos métodos ahora llaman el servicio en lugar de hacer la validación localmente:

```
isValidField(field: string) {
 return this.validatorSercice.isValidField(this.form, field);
}

getErrorMessage(field: string) {
 return this.validatorSercice.getFieldError(this.form, field);
}
```

Revisemos nuevamente el proceso de validación completo, todo inicial en el componente, quien define un **FormGroup** por ejemplo para el **UserName**

```
public form = this.formBuilder.group({
 // Otros campos se definen aca
 userName: ['',
 [
 // Otras validaciones se definen aca
 (control: FormControl) =>
 this.validatorService.cantbeThisValue(control, ['admin', 'administrator',
 'root'])
]
],
});
```

Luego el Template hace el Enlace (primeramente a nivel del FormGroup) y luego con el campo:

```
<input formControlName="userName">
 <!-- otras propiedades del input -->
</input>

 Nombre de usuario no permitido.

```

Cuando el usuario escribe **admin** en el **userName**, al perder el focus se aplica las validaciones en este caso manda a llamar el **cantbeThisValue** este valor no cumple con la regla, por lo tanto el campo **UserName** tiene un error.

Dado que el campo tiene un error, la llamada desde el template al metodo **isValidField('userName')** hace que el **span** con el texto *Nombre de usuario no permitido.* sea visible al usuario.

Una vez aplicadas todos los cambios en el template veremos un formulario sin mensajes de errores y listo para aceptar datos

## Validaciones Reactivas

Nombre	Leonel Messi
Email	messi@argentina.com
Username	TheGoat
Password	.....
Confirmar	.....

**Crear**

Dado que con estos datos el formulario es válido, al hacer el submit, enviaríamos los siguientes datos al server:

```
{
 "name": "Leonel Messi",
 "email": "messi@argentina.com",
 "user_name": "TheGoat",
 "password": "12345678",
 "password_confirmation": "12345678"
}
```

NOTA: Aún hace falta implementar los validadores para los passwords.

## Validadores Asíncronos

Hasta el momento, todos los validadores que hemos agregado son Síncronos, al llamarlos, esperamos una respuesta y el flujo continuará cuando regresan una respuesta, usualmente un objeto. Estos validadores síncronos los hemos colocado como segundo argumento, este segundo argumento puede ser un arreglo de validadores o un único validador.

Pero este formato acepta un tercer argumento, también un único validador o un arreglo de validadores, con la diferencia que estos son **Asíncronos**, es decir regresaran una promesa o un observable.

```
public formName = this.formBuilder.group({
 fieldName: ['', [Validadores Síncronos], [Validadores Asíncronos]]
});
```

Para crear un Validador Asíncrono debemos implementar la interfaz **AsyncValidator** y debemos definir el método **Validate**, el método **registerOnValidatorChange** es opcional.

```
validate(control: AbstractControl<any, any>): Promise<ValidationErrors | null> | Observable<ValidationErrors | null> {
 throw new Error('Method not implemented.');
}
```

Este método acepta un control, y retorna ya sea un Promise o un Observable. Vamos a retornar observables en nuestro caso.

Las validaciones asincronas, colocan el formulario en un estado de **pending** es decir, que existen, en este caso, validaciones que se están ejecutando. Para ver ese estado tendríamos que retrazar la ejecución del validador a propósito (delay) para poder observar este efecto.

Otro ejemplo sería implementar una validacion que reuqiere hacer alguna verificación en el backend, mientras no se complete dicha operación, el formulario estaría en estado **Pending**

Implementemos un Validador de Email, que consulte si el email ya ha sido registrado por otro usuario. Dado que no tenemos un backend vamos a crear un Fake httpCall.

```
import { Injectable } from '@angular/core';
import { AbstractControl, AsyncValidator, ValidationErrors } from '@angular/forms';
import { Observable, of } from 'rxjs';

@Injectable({providedIn: 'root'})
export class AsyncEmailValidator implements AsyncValidator {

 constructor() { }

 validate(control: AbstractControl): Observable<ValidationErrors | null>
 {
 const email = control.value;

 const httpCallObservable = new Observable<ValidationErrors | null>(
 (suscriber) => {
 if (email === 'example@domain.com') {
 suscriber.next({ emailTaken: true });
 suscriber.complete();
 return;
 }

 suscriber.next(null);
 suscriber.complete();
 }
)

 return httpCallObservable;
 }
}
```

En este código, por medio de un **new Observable** creamos un fake HTTP que simula una validación en el backend. Si el email **example@domain.com** es proporcionado por el Usuario, el Observable emite un **{ emailTaken: true }** de lo contrario emite un **null**

Ahora es momento de usar nuestro validador, dado que lo hemos injectado con root **@Injectable({providedIn: 'root'})** lo cual lo hace accesible desde cualquier parte de nuestra app, podemos instanciarlos como **new AsyncEmailValidator()** y injectarlos en el constructor.

Usemos el NEW

```
email: ['',
 [Validators.required,
 Validators.pattern(this.validatorService.emailPattern)],
 [new AsyncEmailValidator()]],
```

Insertamos un segundo arreglo con el **new AsyncEmailValidator()**, dado que ya hemos agregado un primero arreglo de validadores síncronos, demos agregar los **Async** como Arreglo.

**Si todos los validadores del primer grupo se cumplen, se hace las validaciones de los Async.**

Si agregamos el email **example@domain.co** se dispara una primera vez el Async validator, porque ese email es válido segund el Pattern, que permite entre 2 y 4 caracteres luego del punto separador del dominio **[a-z]{2,4}\$** Cuando se hace esta primer llamada ASYNC a nuestro nuevo validador, este retorna un null, pero tan pronto completamos el email **example@domain.com** se llama nuevamente el validador ASYNC y esta vez retorna:

```
{ "emailTaken": true }
```

Para mostrar el mensaje en pantalla, debemos llamar el **getErrorMessage**

```

 {{ getErrorMessage('email') }}

```

Pero debemos agregar el índice **emailTaken** en nuestro Switch del método **getErrorMessage** del servicio: **SyncValidatorService**

```
case 'emailTaken':
 return `The email is already taken`;
```

Con estos cambios ya hemos implementado un Validador Asíncrono y hemos mostrado el error correspondiente en pantalla.

## Validar dos campos diferentes

Esta es una situación particular, anteriormente aplicamos validadores a nivel de cada campo, pero en este caso no podemos aplicarlo a un solo campo, porque solo tenemos acceso al control al que aplicamos el validador.

Lo más transparente es subir un nivel, para poder acceder a todos los campos del formulario al momento de la validación, es decir que en lugar de pasar implícitamente un CAMPO en específico, pasamos todo el formulario, y luego podemos comparar los campos que necesitemos.

LA validación de **passwords iguales** es un ejemplo claro.

Necesitamos dos cosas

- Un lugar adecuado que tenga acceso a todos los campos del formulario para usar la función Validadora.
- Una función validadora que compare dos valores y retorne un ValidationError si son iguales

Primero, debemos colocar este validador siempre dentro de la definición del **Form** una vez que definimos los campos y sus validadores, como segundo parámetro podemos pasar un arreglo de funciones validadoras, en este caso se aplicaran a todo el formulario, o dicho de otra forma, estas funciones reciben implícitamente como argumento el formulario.

```
public formName = this.formBuilder.group({
 fieldName: ['', [Validadores Síncronos], [Validadores Asíncronos]]
}, {
 validators: [
 Validators Síncronos Globales
]
});
```

Si la función validadora a nivel de FormGroup recibe el formulario, entonces podemos hacer un validador que acepte dicho formulario, y haga la validación con los campos requeridos, la función en cuestión es la siguiente:

```
/**
 * Compare two fields of a form and return an error if they are not equal
 *
 * @param field1 Field name 1 to compare
 * @param field2 Field name 2 to compare
 *
 * @returns A function that compares the values of the fields and
 * returns null if they are equal or an error if they are not equal
 */
public fieldsMatch = (field1: string, field2: string) => {
 return (formGroup: AbstractControl): ValidationErrors | null => {
 const control1 = formGroup.get(field1);
 const control2 = formGroup.get(field2);
 const fieldValue1 = control1?.value;
```

```
const fieldValue2 = control2?.value;

if (fieldValue1 === fieldValue2) {
 control2?.setErrors(null);
} else {
 control2?.setErrors({ notEqual: true });
}

return null;
}
}
```

La Función **fieldsMatch** recibirá implícitamente el formulario desde donde se manda a llamar, adicionalmente debe recibir el nombre de los dos controles que deben compararse.

Pero en este caso **fieldsMatch** retorna otra función, la cual captura el FormGroup. Internamente podemos acceder a los nombres de los campos y a la vez extraer sus valores usando el FormGroup.

Finalmente debemos agregar el nuevo KEY-ERROR **notEqual** a nuestro servicio **getFieldError**

# Nueva Sección: Selectores Anidados, Nueva APP

## ¿Qué veremos en esta sección?

El objetivo de la sección básicamente es aprender a reaccionar cuando información de un formulario cambia, en este caso haremos:

- Selectores anidados
- Llenar selectores desde peticiones HTTP
- Encadenamiento de operadores de RXJS
- Explicar el por qué la gente dice que TypeScript no es del todo "Type Safe"

Creamos una nueva APP

```
ng new 07-selectorsApp --standalone false --routing
```

Luego creamos un módulo, **Country**

```
ng g m country
```

Creamos la siguiente estructura de directorios, dentro de **src/app/country**

```
└── country
 ├── country.module.ts
 ├── interfaces
 ├── pages
 └── services
```

Luego creamos nuestro primer componente, dentro de **pages**

```
ng g c country/pages/selector-page
CREATE src/app/country/pages/selector-page/selector-page.component.css (0 bytes)
CREATE src/app/country/pages/selector-page/selector-page.component.html (28 bytes)
CREATE src/app/country/pages/selector-page/selector-page.component.spec.ts (644 bytes)
CREATE src/app/country/pages/selector-page/selector-page.component.ts (226 bytes)
```

Renombremos nuestro selector, a **country-selector-page**

```

@Component({
 selector: 'country-selector-page',
 templateUrl: './selector-page.component.html',
 styleUrls: ['./selector-page.component.css']
})
export class SelectorPageComponent {
}

```

Editemos el **app-routing.module.ts** y carguemos en forma LazyLoading el **CountryModule**

```

const routes: Routes = [
 {
 path: 'selector',
 loadChildren: () => import('./country/country.module').then(m =>
m.CountryModule)
 },
 {
 path: '**',
 redirectTo: 'selector'
 }
];

```

Ahora debes crear nuestro **country-routing.module.ts**, este pudo haber sido creado con el parametro --**routing** cuando creamos el módulo.

```

const routes: Routes = [
 {
 path: '',
 component: SelectorPageComponent
 }
];

@NgModule({
 imports: [RouterModule.forChild(routes)],
 exports: [RouterModule]
})
export class CountryRoutingModule { }

```

Otra forma, quizás mejor, es usar childrens, porque es probable que necesitemos agregar mas paginas en nuestro componente Country,

```

const routes: Routes = [
 {
 path: '',
 children: [
 ...
]
 }
];

```

```

 {
 path: 'countries',
 component: SelectorPageComponent
 },
 {
 path: '**',
 redirectTo: 'countries'
 }
]
}
];

```

Finalmente lo conectamos con nuestro módulo principal **CountryModule** en el imports:

```

@NgModule({
 declarations: [
 SelectorPageComponent
],
 imports: [
 CommonModule,
 CountryRoutingModule
]
})
export class CountryModule { }

```

Listo, ya estamso cargando el SelectorPage cuando iniciamos la APP en la ruta:

<http://localhost:4200/selector> o bien <http://localhost:4200/selector/countries> si aplicamos la segunda solución de mostrar las rutas con **children**

## Selector-Page Component

Creamos un formulario con formBuilder, tal como se indica a continuación:

```

constructor(private formBuilder: FormBuilder) { }

public form: FormGroup = this.formBuilder.group({
 region: ['', Validators.required]
});

```

Enlazamos el **form** en nuestro template y hacemos lo mismo a nivel de campos, enlazado a tres selectores.

```

<h3 class="mt-3">Selectores Anidados</h3>
<hr>

<form [FormGroup]="form" (ngSubmit)="onSubmit()">
 <div class="row mb-3">
 <div class="col">

```

```

<div class="mb-3">
 <label for="form-label">Region</label>
 <select formControlName="region"
 class="form-select form-select-lg">
 <option selected>Seleccione Uno</option>
 <!-- Llenar desde Rest Api-->
 </select>
</div>

</div>
</div>
</form>

```

Vamos a usar el <https://restcountries.com/> obtenemos un ejemplo del JSON y usando **Past as JSON code** creamos una interfaz en `/country/intergaces/country.interface.ts` y agreemos un enum para las regiones, en el mismo archivo de interfaces

```

export enum Region {
 Africa = "Africa",
 Americas = "Americas",
 Asia = "Asia",
 Europe = "Europe",
 Oceania = "Oceania",
}

```

Creamos un servicio:

```
$ ng g s country/services/countries --skip-tests
CREATE src/app/country/services/countries.service.ts (138 bytes)
```

Y agreemos el siguiente código:

```

private _regions: Region[] = [
 Region.Africa,
 Region.Americas,
 Region.Asia,
 Region.Europe,
 Region.Oceania
];

get regions(): Region[] {
 return [...this._regions];
}

```

NOTA: no debemos regresar el `this._regions` porque esto pasaria la referencia del objeto, y se podría modificar por el que lo consume, por lo tanto lo que retornamos es un nuevo arreglo, el cual se construye a

partir de los elementos en ...`this._regions`

Ahora inyectamos el servicio en nuestro componente, y como es privado, (el servicio) no podemos accederlo desde el template, por lo tanto hacemos un getter en el componente

```
get Regions() {
 return this.countriesServices.regions;
}
```

Y en el template, llenamos el select:

```
<select formControlName="region"
 class="form-select form-select-lg">
 <option selected>Seleccione Uno</option>
 <option *ngFor="let region of Regions" [value]="region">
 {{ region }}
 </option>
</select>
```

## Segundo selector Animado

Usaremos una petición http para obtener los países de la región seleccionada, por lo tanto debemos importar el **HttpClientModule** en el AppModule.

Ademas usaremos una versión más reducida de la interfaz **Country**,

```
export interface CountriesResponse {
 name: string;
 cca3: string;
 borders: string[];
}
```

Además necesitamos agregar este nuevo método en nuestro servicio:

```
getCountriesByRegion(region: Region): Observable<CountriesResponse[]> {

 if (!region) {
 return of([]);
 }

 const url = `${this.baseUrl}/region/${region}?
fields=name,cca3,borders`;

 return this.httpClient.get<Country[]>(url)
 .pipe(
 map(countries => countries.map(country => (
 /
))
)
}
```

```
{
 name: country.name.common,
 cca3: country.cca3,
 borders: country.borders ?? []
}
)),
);
}
```

Este método retorna un `Observable<CountriesResponse[]>` esta interfaz reducida solo define el country name (como string), el cca3 y las fronteras o borders.

Hacemos la petición y luego pasamos esa respuesta por un **pipe**, la función **map** toma cada valor (country) de la respuesta, y lo transforma a la versión reducida.

Ahora necesitamos consumir este servicio en el Componente, donde definimos previamente el primer selector.

Creamos un método **onRegionChange** el cual crea un **Listener** con el **valueChanges** sobre el primer selector, en otras palabras, cuando el selector cambie el valor seleccionado se va a disparar este Evento.

```
onRegionChange(): void {
 this.form.get('region')?.valueChanges
 .pipe(
 switchMap(region =>
 this.countriesServices.getCountriesByRegion(region))
 .subscribe(countries => {
 this.countries = countries;
 });
}
```

Ahora hace falta llamar este método, y lo haremos en el `OnInit`

```
ngOnInit(): void {
 this.onRegionChange();
}
```

Cuando se inicializa el componente, el `OnInit` va a llamar el **onRegionChange** quien va a crear el Listener sobre el selector de Regiones.

Agregamos el Template el segundo selector

```
<div *ngIf="countries.length > 0" class="row mb-3">
 <div class="col-6">
 <div class="mb-3">
 <label for="form-label">Countries</label>
 <select formControlName="country">
```

```

 class="form-control">
 <option selected>Seleccione el País</option>
 <option *ngFor="let country of countries"
[value]="country.cca3">
 {{ country.name }}
 </option>
 </select>
 </div>
</div>
</div>

```

Listo con este cambio podemos ver la lista de países una vez seleccionamos una región.

## Tercer Selector Anidado

Mostraremos los países fronterizos con el país seleccionado en el segundo selector. Dado que la información de las fronteras viene en un arreglo de string, el cual contiene códigos de países, y lo que deseamos mostrar se el nombre del país fronterizo vamos hacer una llamada adicional.

En el Servicio agregamos dos nuevos métodos:

```

getCountryByCode(cca3: string): Observable<CountriesResponse> {
 const url = `${this.baseUrl}/alpha/${cc3}?fields=name,cc3,borders`;
 return this.httpClient.get<Country>(url).pipe(
 map(country => {
 {
 name: country.name.common,
 cc3: country.cc3,
 borders: country.borders ?? []
 }
)));
}

getCountryBordersByCode(borders: string[]): Observable<CountriesResponse[]>
{
 if (!borders || borders.length === 0) {
 return of([]);
 }
 const requests: Observable<CountriesResponse>[] = [];
 borders.forEach(cca3 => {
 requests.push(this.getCountryByCode(cca3));
 });
 return combineLatest(requests);
}

```

**getCountryByCode** obtiene el nombre de país, el código y sus fronteras. Este método lo usaremos cuando cambiemos un País del segundo selector.

**getCountryBordersByCode** Va a recibir un arreglo de strings[] los cuales son los códigos de países de las fronteras. Este arreglo lo vamos a tomar de la respuesta de **getCountryByCode** (country.borders) Notar que estamos usando un método diferente, dado que el API (probablemente si) no nos permite hacer consultas mandando un arreglo de códigos, tenemos que enviar tantas peticiones como códigos de fronteras vengan en el arreglo.

Explicación adicional:

Esta línea `requests.push(this.getCountryByCode(cca3));` no hace la llamada al API, simplemente agregar el request a nuestro arreglo de requests.

El que se encarga de hacer todas las peticiones es `combineLatest(requests);` esto se hace de una sola vez, todas al mismo tiempo. El método se encargara de gestionarlas y enviar una respuesta final.

Con estos métodos combinados, vamos a obtener la lista de fronteras, pero en forma de Paises, de modo que podremos mostrar el nombre del país fronterizo en lugar del Código.

Una vez implementados estos métodos, agregamos un nuevo Listener en nuestro Componente:

```
public borders: CountriesResponse[] = [];

onCountryChange(): void {
 this.form.get('country')?.valueChanges
 .pipe(
 tap(() => this.form.get('border')?.reset('')),
 tap(() => this.borders = []),
 filter(code => code !== ''),
 switchMap(code => this.countriesServices.getCountryByCode(code)),
 switchMap(country =>
 this.countriesServices.getCountryBordersByCode(country?.borders || [])
).subscribe(countryResponse => {
 this.borders = countryResponse || [];
 });
);
}
```

Primero definidmos un arreglo **borders** el cual contendrá la lista de paises fronterizos.

Segundo agregamos el Listener sobre el campo **country** con el *valueChanges*

Cuando se seleccione un país occuren las siguientes tareas

- Limpamos el selector de las fronteras, por si se había mostrado algo anteriormente
- Limpiamos internamente el arreglo de Borders.
- Si el Código seleccionado es vacío o no definido, filtramos ese evento, es decir no continúa en el PIPE
- Con SwitchMap hacemos un llamado al **getCountryByCode** para obtener la información del País seleccionado, recordemos que aca vamos a tener, el nombre del país, su código y lo mas importante los códigos de los países fronterizos en forma de arreglo de String[]

- Con el Segundo Switch, mandamos el arreglo de paises fronterizos al método **getCountryBordersByCode** y este retornara un arreglo de Paises
- Luego asignamos este arreglo de países a nuestra variable pública **this.borders**

Finalmente el template va a mostrar la lista de Fronteras con Nombres de Países

```
<div *ngIf="borders.length > 0" class="row mb-3">
 <div class="col-6">
 <div class="mb-3">
 <label for="form-label">Borders</label>
 <select formControlName="border"
 class="form-control">
 <option selected>Seleccione la Frontera </option>
 <option *ngFor="let border of borders"
 [value]="border.cca3">
 {{ border.name }}
 </option>
 </select>
 </div>
 </div>
</div>
```

# Nueva Sección: El ciclo de un componente

## ¿Qué veremos en esta sección?

Se explicarán todos los pasos del ciclo de vida de un componente (también se aplican a las directivas que veremos después).

Creamos una nueva APP

```
ng new 08-lifeCycle --standalone false --routing
```

Creamos un modulo

```
ng g m products --routing
CREATE src/app/products/products-routing.module.ts (251 bytes)
CREATE src/app/products/products.module.ts (288 bytes)
```

Agregamos el ProductsModule al Import del AppModule, no vamos hacer carga lazy en esta app.

Creamos dos directorios **pages** y **comonents** dentro de **Products**

Creamos los componentes

```
ng g c products/pages/product
ng g c products/components/price
```

```
products
 ├── components
 └── pages
 └── product
 ├── product.component.css
 ├── product.component.html
 ├── product.component.spec.ts
 └── product.component.ts
 └── products.module.ts
 └── products-routing.module.ts
```

Luego agregamos las rutas necesarias y actualizamos los módulos, el objetivo es centrarnos en el ciclo de vida del componente.

En **ProductComponent** vamos a implementar todos los eventos del ciclo de vida, en el siguiente código se encuentran en el orden de ejecución.

```
export class ProductComponent implements OnChanges, OnInit, DoCheck,
AfterContentInit, AfterContentChecked, AfterViewInit, AfterViewChecked,
OnDestroy{

 // Ocurre cuando Angular inicializa el componente
 constructor() { }

 // Responde cuando Angular establece o restablece propiedades
 // de entrada vinculadas a datos. El método recibe un objeto
 // SimpleChanges de los valores de propiedad actuales y anteriores.
 ngOnChanges(changes: SimpleChanges): void {}

 // Inicialice la directiva o el componente después de que
 // Angular muestre por primera vez las propiedades vinculadas
 // a datos y establezca las propiedades de entrada de la
 // directiva o el componente.
 ngOnInit(): void {}

 // Detecta y actúa sobre cambios que Angular no puede o no
 // quiere detectar por sí solo.
 ngDoCheck(): void {}

 // Responde después de que Angular proyecta contenido externo
 // en la vista del componente o en la vista en la que se
 // encuentra una directiva.
 ngAfterContentInit(): void {}

 // Responde después de que Angular verifica el contenido
 // proyectado en la directiva o componente.
 ngAfterContentChecked(): void {}

 // Responde después de que Angular inicializa las vistas
 // y las vistas secundarias del componente, o la vista
 // que contiene la directiva.
 ngAfterViewInit(): void {}

 // Responde después de que Angular verifica las vistas
 // y las vistas secundarias del componente, o la vista
 // que contiene la directiva.
 ngAfterViewChecked(): void {}

 // Limpiezas justo antes de que Angular destruya la directiva o
 // componente.
 // Cancele la suscripción de Observables y separe los controladores
 // de eventos para evitar pérdidas de memoria.
 ngOnDestroy(): void {}

}
```

Si agregamos un console.log con el nombre del método, veremos este orden de ejecución.

```
ngOnInit
ngDoCheck
ngAfterContentInit
ngAfterContentChecked
ngAfterViewInit
ngAfterViewChecked
ngDoCheck
ngAfterContentChecked
ngAfterViewChecked
```

El **ngOnChanges** no se muestra porque el componente no tiene ningún elemento que pueda cambiar.

Si agregamos un propiedad pública

```
public isProductVisible: boolean = true;
```

Y lo enlazamos en el Template:

```
<button (click)="isProductVisible = !isProductVisible">
 Testing
</button>
```

Al presinar el botón vemos estos eventos:

```
ngDoCheck
ngAfterContentChecked
ngAfterViewChecked
```

## OnChange con @Input

Agreguemos esto al PriceComponent

```
export class PriceComponent implements OnInit, OnChanges, OnDestroy {

 @Input() price: number = 0;

 ngOnInit(): void {
 console.log('PriceComponent - ngOnInit');
 }

 ngOnChanges(changes: SimpleChanges): void {
 }
}
```

```
 console.log(' PriceComponent - ngOnChanges');
 console.log(' ' + JSON.stringify(changes));
}

ngOnDestroy(): void {
 console.log(' PriceComponent - ngOnDestroy');
}
}
```

Usemos este componente en nuestro ProductoComponent

```
<p>product works!</p>

<app-price
 *ngIf="isProductVisible"
 [price]="currentPrice">
</app-price>

<button (click)="isProductVisible = !isProductVisible">
 Toggle product
</button>

<button (click)="increasePrice()">
 Increase Price
</button>
```

**increasePrice()** aumenta en +1 el precio del producto, por medio de una propiedad public `currentPrice`

```
public currentPrice: number = 0;
```

Si ejecutamos la APP, vemos el Log, observamos

The screenshot shows a browser's developer tools with the 'Console' tab selected. At the top, there are two buttons: 'Testing' and 'Increase Price'. Below the buttons, the console output displays the following logs:

```

product works!
price works!
Testing Increase Price
Angular is running in development mode.
ngOnInit
ProductComponent - ngDoCheck
ProductComponent - ngAfterContentInit
ProductComponent - ngAfterContentChecked
 PriceComponent - ngOnChanges
 {"price": {"currentValue": 0, "firstChange": true}}
 PriceComponent - ngOnInit
ProductComponent - ngAfterViewInit
ProductComponent - ngAfterViewChecked
ProductComponent - ngDoCheck
ProductComponent - ngAfterContentChecked
ProductComponent - ngAfterViewChecked
> |

```

El objeto **change** también tiene una propiedad **previousValue** la cual es undefined, por eso no se muestra en el JSON.

Agreguemos un `console.log(JSON.stringify(changes));` al **ngOnChanges** del ProductComonent, Si presionamos el botón que incrementa el precio en +1 veremos esto:

The screenshot shows a browser's developer tools console with the following log entry:

```

ProductComponent - ngDoCheck
ProductComponent - ngAfterContentChecked
 PriceComponent - ngOnChanges
 {"price": {"previousValue": 0, "currentValue": 1, "firstChange": false}}
ProductComponent - ngAfterViewChecked
> |

```

Primero, el cambio de la propiedad interna **currentPrice** de ProductComponent, no dispara el **ngOnChanges** del ProductoComponent, pero si el del componente hijo **PriceComponent** Y podemos ver el valor anterior y el nuevo valor, además de que **firstChange** es **false**

EL **firstChange** es TRUE solo al momento de que el componente Padre le pasa el valor de **0** y el componente precio lo acepta por el **@Input**

Tocar repetidamente el botón de incrementar el precio generará estos cambios en el objeto **changes** :

```
[
 {"price": {"currentValue": 0, "firstChange": true}},
 {"price": {"previousValue": 0, "currentValue": 1, "firstChange": false}},
 {"price": {"previousValue": 1, "currentValue": 2, "firstChange": false}},
 {"price": {"previousValue": 2, "currentValue": 3, "firstChange": false}}
]
```

El registro anterior incluye la primera carga del componente, y luego click en el botón incrementar precio por 3 veces.

## OnDesroy del Hijo

Tenemos este botón:

```
<button (click)="isProductVisible = !isProductVisible">
 Toggle product
</button>
```

El cual cambia el valor de la proiedad **isProductVisible** y a la vez oculta el componente Hijo

```
<app-price
 *ngIf="isProductVisible"
 [price]="currentPrice">
</app-price>
```

Si presionamos ese botón, el componente Precio debe de destruirse.

```
ProdcutComponent - ngDoCheck
ProdcutComponent - ngAfterContentChecked
 PriceComponent - ngOnDestroy
ProdcutComponent - ngAfterViewChecked
>
```

Si nuevamente presionamos el mismo botón, el ciclo de vida de dicho componente se activa nuevamente y mostrará su estado inicial

```
{"price": {"currentValue": 0, "firstChange": true}}
```

## OnDestroy

Cualquier observable que emita constantemente valores debe ser cancelada su suscripción al momento de destruir el elemento, caso contrario, seguirá emitiendo valores y puede causar problemas de rendimiento.

Si creamos el siguiente Interval

```
ngOnInit(): void {
 console.log(' PriceComponent - ngOnInit');
 interval(1000).subscribe(value => {
 console.log(' PriceComponent - interval: ' + value);
 });
}
```

Al salir del Componente Price (ocultarlo) veremos en consola que seguirá imprimiendo valores.

Podemos usar el OnDestroy para cancelar la suscripción. Para ello asignaremos la suscripción a una variable pública

```
public interval$?: Subscription;

ngOnInit(): void {
 this.interval$ = interval(1000).subscribe(value => {
 console.log(' PriceComponent - interval: ' + value);
 });
}

ngOnDestroy(): void {
 this.interval$?.unsubscribe();
}
```

Con este cambio, al ocultar el componente, se llama el ngOnDestroy y el intervalo se cancela. Generalmente cualquier objeto que genere o escuche eventos, debe ser destruido junto con el componente.

# Nueva Sección: Nueva Aplicación de Mapas:

## ¿Qué veremos en esta sección?

¿Qué veremos en esta sección?

Este es un breve listado de los temas fundamentales:

- Manejo de librerías escritas en JavaScript en TypeScript
- Uso de Mapas basados en Mapbox (el API es similar a la de Google Maps)
- Marcadores
- Eventos
- FlyTo
- Coordenadas geográficas
- Componentes para reutilización de mapas
- Mantener objetos de forma persistente
- @types
- Zoom
- Range

## Crear nueva APP y preparar proyecto

Hacemos lo de siempre, creamos nueva app, modificamos el scrit del start para agregar el parámetro **-o**, limpiamos el APPComponent.html

```
ng new 09-mapApp --standalone false --routing
```

Agregamos en el index.html la referencias a bootstrap y fontawesome

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN" crossorigin="anonymous">
<script src="https://kit.fontawesome.com/8de2637909.js" crossorigin="anonymous"></script>
```

Creamos un módulo:

```
$ ng g m maps --routing
CREATE src/app/maps/maps-routing.module.ts (247 bytes)
CREATE src/app/maps/maps.module.ts (272 bytes)
```

Componentes

```
ng g c maps/components/minimap
ng g c maps/components/site-menu

ng g c maps/layout/maps-layout --skip-selector
ng g c maps/pages/full-screen-page --skip-selector
ng g c maps/pages/properties-page --skip-selector
```

--skip-selector omite agregar un selector al componente, este es un componente especial que no necesita ser incluido como un TAG, eso lo veremos adelante. El componente se crea de la siguiente forma:

```
@Component({
 templateUrl: './maps-layout.component.html',
 styleUrls: ['./maps-layout.component.css']
})
export class MapsLayoutComponent {
```

Esta es la estructura actual:

```
$ tree app
app
├── app.component.css
├── app.component.html
├── app.component.ts
├── app.module.ts
├── app-routing.module.ts
└── maps
 ├── components
 │ ├── mini-map
 │ │ ├── mini-map.component.css
 │ │ ├── mini-map.component.html
 │ │ └── mini-map.component.ts
 │ └── site-menu
 │ ├── site-menu.component.css
 │ ├── site-menu.component.html
 │ └── site-menu.component.ts
 ├── layout
 │ └── maps-layout
 │ ├── maps-layout.component.css
 │ ├── maps-layout.component.html
 │ └── maps-layout.component.ts
 ├── maps.module.ts
 └── maps-routing.module.ts
└── pages
 ├── full-screen-page
 │ ├── full-screen-page.component.css
 │ ├── full-screen-page.component.html
 │ └── full-screen-page.component.ts
 └── properties-page
 ├── properties-page.component.css
 ├── properties-page.component.html
 └── properties-page.component.ts
```

En el archivo **src/app/maps/layout/maps-layout/maps-layout.component.html** agregamos un **<router-outlet></router-outlet>**

En el **AppRoutingModule** agregamos una ruta, con carga Lazy

```
const routes: Routes = [
 {
 path: 'maps',
 loadChildren: () => import('./maps/maps.module').then(m =>
mMapsModule)
 },
 {
 path: '**',
 redirectTo: 'maps',
 }
];
```

En cambio, en el **MapsRoutingModule** agregaremos estas rutas:

```
const routes: Routes = [
 {
 path: '',
 component : MapsLayoutComponent,
 children: [
 { path: 'fullscreen', component: FullScreenPageComponent},
 { path: 'properties', component: PropertiesPageComponent},
 { path: '**', redirectTo: 'fullscreen'}
]
 }
];
```

## Maps Menu lateral (sideMenu)

Nuestro **MapsLayoutComponent** va a funcionar como un contenedor general de nuestros componentes dentro del modulo de mapas. **MapsLayoutComponent** es un componente que pone a disposición un espacio para que los demás componentes puedan renderizarse.

Vamos a crear el sideMenu y lo primero que haremos es agregar al template del **MapsLayoutComponent**

```
<maps-site-menu></maps-site-menu>
<router-outlet></router-outlet>
```

Podemos agregar CSS a los componentes desde el padre, podríamos agregar esto en nuestro Layout CSS Component:

```
maps-site-menu {
 left: 20px;
 position: fixed;
 top: 20px;
 width: 180px;
 z-index: 999;
}
```

Para agregar las opciones al menú, haremos lo mismo, creamos un `menuItem[]` y agregamos rutas y nombre de los items. Esto lo hacemos en el `siteMenuComponent`

```
interface MenuItems {
 router: string,
 name: string,
 icon: string
}

@Component({
 selector: 'maps-site-menu',
 templateUrl: './site-menu.component.html',
 styleUrls: ['./site-menu.component.css']
})
export class SiteMenuComponent {
 public menuItems: MenuItems[] = [
 { router: '/maps/fullscreen', name: 'Fullscreen', icon: 'fa-solid fa-desktop' },
 { router: '/maps/properties', name: 'Properties', icon: 'fa-solid fa-house-chimney-window' },
]
}
```

Y luego renderizamos con un `NgFor`.

```
<ul class="list-group">
 <li
 *ngFor="let item of menuItems"
 class="list-group-item"
 [routerLink]=[item.router]
 routerLinkActive="active">
 <i class="menu-icon {{item.icon}}></i>{{ item.name}}


```

## MapBox

Usaremos este servicio, de modo que se crear una cuenta y un token de acceso el cual usaremos en nuestro ENV files.

# Archivos de Environment

Crearemos los siguientes archivos

Nombre	Ubicación	Descripción
.env	./	Contiene las variables de entorno, no se sube al repositorio
.env.template	./	Es un template que indica que variables se deben de configurar. Este archivo si debe ser enviado al repositorio
.environment	./src/environments	Se genera mediante un script, Angular usará este archivo para inicializar nuestras variables. Al igual que .env, es un archivo que no debe enviarse al repositorio, ya que contendrá información sensible.

## Construir Archivo .environment

Usaremos **dotenv**: Dotenv es un módulo de dependencia cero que carga variables de entorno desde un archivo .env en **Process.env**.

Instalamos el paquete en Desarrollo:

```
$ npm install dotenv --save-dev
```

Necesitamos transformar el contenido en nuestro archivo privado .env en una versión para que el entorno de Angular pueda usar dichas variables, para ello usaremos node y un script en JS

```
const { writeFileSync, mkdirSync } = require('fs');

require('dotenv').config();

const targetPath = './src/environments/environment.ts';

const envFileContent = `
export const environment = {
 mapbox_key: "${process.env.MAPBOX_KEY}",
};

mkdirSync('./src/environments', { recursive: true });

writeFileSync(targetPath, envFileContent);
```

Este script **set-envs.js** lo guardamos en el path **./scripts/set-envs.js**

Luego agregamos una nueva tarea en el package.json

```
"scripts": {
 "envs": "node ./scripts/set-envs.js",
 "start": "npm run evns && ng serve -o",
 "watch": "npm run evns && ng build --watch --configuration development",
}
```

Probamos la tarea

```
$ npm run envs
> 09-map-app@0.0.0 envs
> node ./scripts/set-envs.js
```

Como resultado podremos ver nuestro nuevo archivo de configuración  
**./src/environments/environment.ts**

Para acceder a las propiedades debemos importar este archivo

```
import { environment } from './../../../../environments/environment';
```

y luego acceder a la propiedad requerida

```
accessToken: environment.mapbox_key
```

## Instalar MapBox

```
npm install --save mapbox-gl
npm i --save-dev @types/mapbox-gl
```

NOTA: MapBox no incluye archivos de declaración (\*d.ts), por lo tanto debemos instalador por separado

Agregamos en el index.html

```
<link href='https://api.mapbox.com/mapbox-gl-js/v3.1.2/mapbox-gl.css'
rel='stylesheet' />
```

## Importar MapBox

Implementaremos el mapa en el **fullScreenComponent**

```
import { AfterViewInit, Component } from '@angular/core';
import { Map } from 'mapbox-gl';
import { environment } from '../../../../environments/environment';

@Component({
 templateUrl: './full-screen-page.component.html',
 styleUrls: ['./full-screen-page.component.css']
})
export class FullScreenPageComponent implements AfterViewInit {

 @ViewChild('map') divMap!: ElementRef;

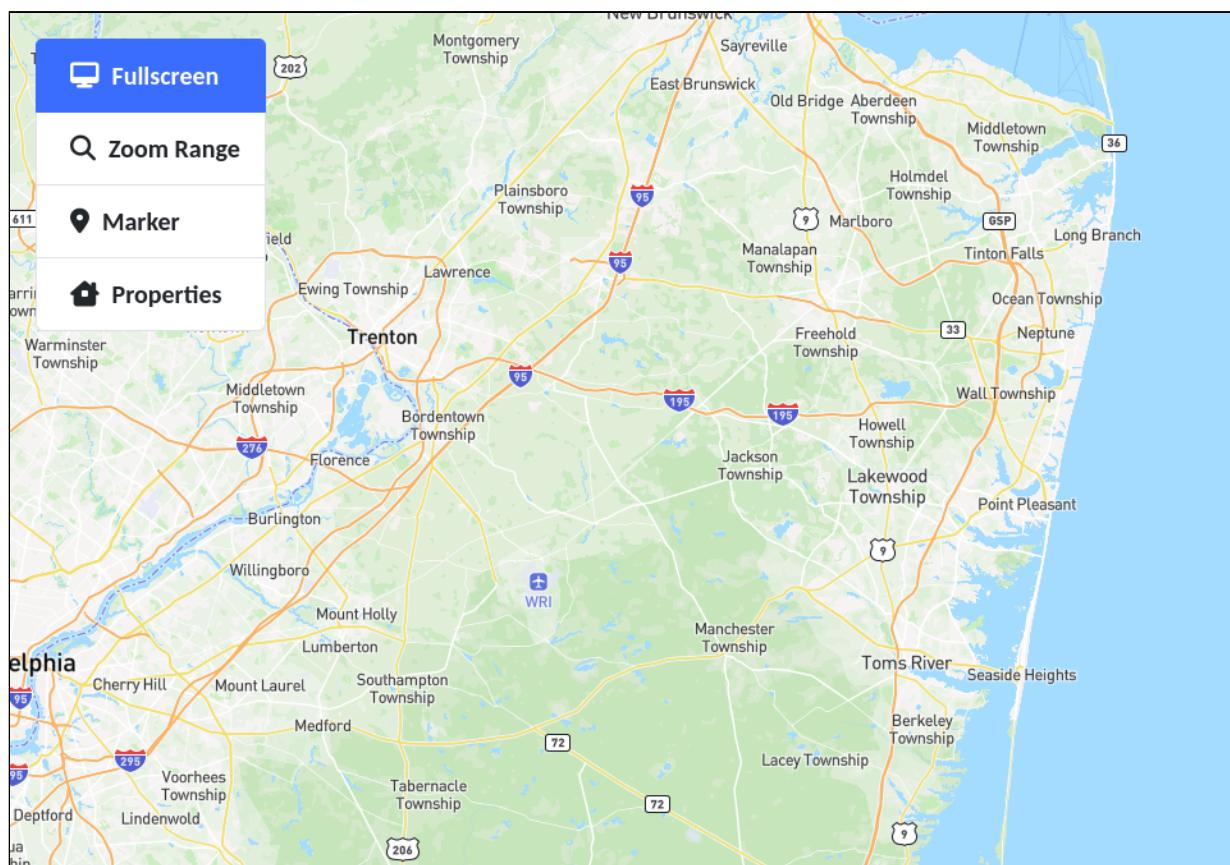
 ngAfterViewInit(): void {
 const map = new Map({
 accessToken: environment.mapbox_key,
 container: this.divMap.nativeElement, // container ID
 style: 'mapbox://styles/mapbox/streets-v12', // style URL
 center: [-74.5, 40], // starting position [lng, lat]
 zoom: 9, // starting zoom
 });
 }
}
```

Agregamos un contenedor, usaremos una referencia local **#map** visible en el componente mediante el **ViewChild**

```
<div #map id="map">
</div>
```

Y algo de estilo:

```
#map {
 width: 100vw;
 height: 100vh;
}
```



## Zoom in/out

Implementamos estos métodos:

```
public zoom: number = 10;

mapListener() {
 this.map.on('zoom', (ev) => {
 this.zoom = this.map.getZoom();
 });
}

OnRangeChange(event: any) {
 this.map.setZoom(event.target.value);
}

zoomIn() {
 this.map.zoomIn();
}

zoomOut() {
 this.map.zoomOut();
}
```

El método **this.mapListener()**; lo llamaremos dentro del **ngAfterViewInit** luego de inicializar el mapa.

Y el Template:

```
<div #map id="map"></div>

<div class="floatting-range p-2">
 <div class="floatting-content">
 <button type="button" class="btn btn-primary" (click)="zoomOut()">-
 </button>
 <input
 type="range"
 class="form-range p-2"
 min="-2"
 max="18"
 [value]="zoom"
 (change)="OnRangeChange($event)"
 >
 <button type="button" class="btn btn-primary" (click)="zoomIn()">+
 </button>
</div>

 Zoom: {{zoom | number: '1.1-1'}}

 Ln, Lt: {{"0,0"}}

</div>
```

Un poco de estilo

```
#map {
 width: 100vw;
 height: 100vh;
}

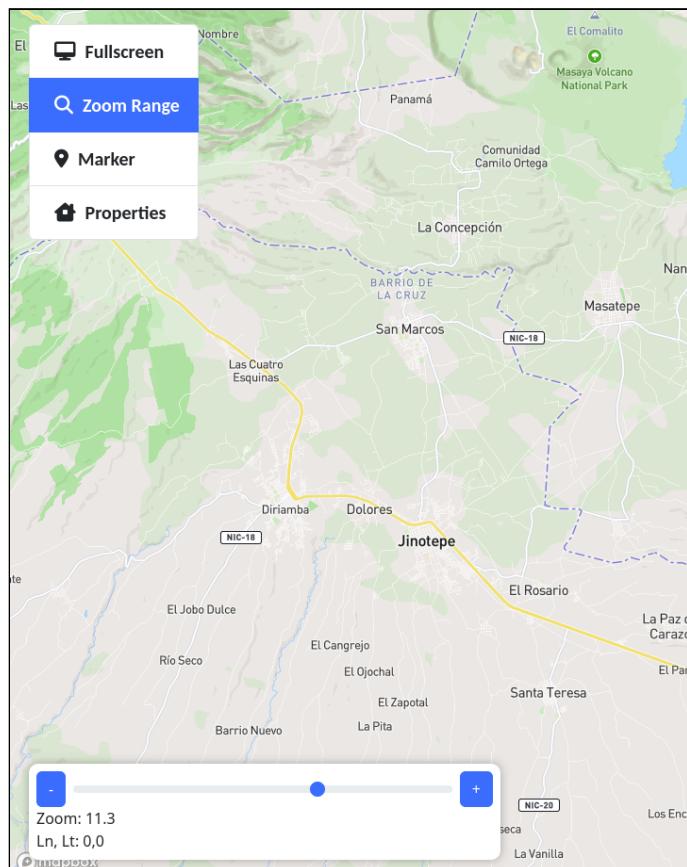
.floatting-range {
 position: fixed;
 bottom: 20px;
 left: 20px;
 z-index: 999;
 width: 500px;
 background-color: white;
 border-radius: 10px;
 box-shadow: 0 0 10px rgba(0, 0, 0, 0.3);
}

.floatting-content {
 display: flex;
```

```


```

Con esto hemos implementado un Zoom in/out en el mapa y hemos conectado los controles en ambas direcciones, al hacer zoom en el mapa se actualizan los controles, al mover los controles se cambia el zoom en el mapa.



## Latitud y Longitud

Creemos una propiedad **LngLat** y la inicializamos

```
public currentLatLng: LngLat = new LngLat(-85.579765542208,
11.577481296026505);
```

Mostramos en el Template

```

 Zoom: {{zoom | number: '1.1-1'}} - ({{currentLatLng.lat | number: '1.4-4'}},
 {{currentLatLng.lng | number: '1.4-4'}})

```

La enlazamos al Mapa con un Listener

```
mapListener() {
 // Otros listeners

 this.map.on('move', (ev) => {
 this.currentLatLng = this.map.getCenter();
 });
}
```

Y listo, con esto podemos ver la Latitud y Longitud del centro del mapa cada vez que nos desplazamos.

## Destruir Listeners

Una tarea importante es limpiar los listeners creados en nuestro componente:

```
ngOnDestroy(): void {
 this.map.remove();
}
```

## Marcadores

El siguiente método, crea un marcador, en la latitud y longitud definida con las opciones enviadas como parámetro

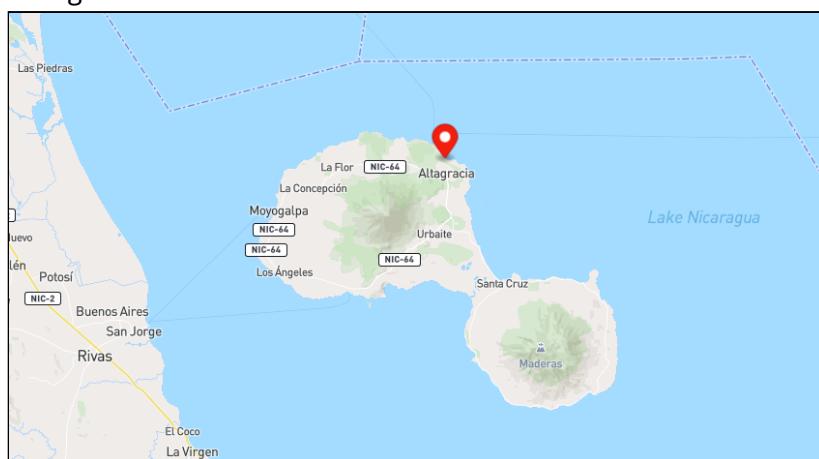
```
addMarker(lngLat: LngLat, options: MarkerOptions) {
 const marker = new Marker({
 ...options
 })
 .setLngLat(lngLat)
 .addTo(this.map);

 if (options.draggable) {
 marker.on('drag', (ev) => {
 this.currentLatLng = marker.getLatLng();
 });
 }
}
```

Podemos llamar este método con las siguientes opciones:

```
const makerOptions = {
 color: '#FF0000',
 draggable: true,
}
this.addMarker(this.currentLatLng, makerOptions);
```

Esto genera:



## Guardar en el Local Storage

Los marcadores que agregamos pueden ser guardados en el local storage, necesitamos dos métodos para guardar y recuperar la información. Nota, no vamos a guardar el objeto Marker, ya que contiene muchos datos, en su lugar guardaremos la latitud y longitud.

## Código completo del Componente

```
export interface MarkerInfo {
 marker: Marker;
 color: string;
}

export interface PlainMarker {
 lng: number;
 lat: number;
 color: string;
}

@Component({
 templateUrl: './full-screen-page.component.html',
 styleUrls: ['./full-screen-page.component.css']
})
export class FullScreenPageComponent implements OnInit, AfterViewInit,
OnDestroy{

 @ViewChild('map') divMap?: ElementRef;

 public zoom: number = 10;

 public map!: Map;

 public currentLatLng: LngLat = new LngLat(-85.579765542208,
 11.577481296026505);

 private markersList: MarkerInfo[] = [];

 ngOnInit(): void {
 /
```

```
}

ngAfterViewInit(): void {

 if(!this.divMap) {
 return;
 }

 this.map = new Map({
 accessToken: environment.mapbox_key,
 container: this.divMap.nativeElement, // container ID
 style: 'mapbox://styles/mapbox/streets-v12', // style URL
 center: this.currentLatLng, // starting position [lng, lat]
 zoom: this.zoom, // starting zoom
 });

 this.mapListener();

 this.loadFromLocalStorage();
}

ngOnDestroy(): void {
 this.map.remove();
}

mapListener() {
 this.map.on('zoom', (ev) => {
 this.zoom = this.map.getZoom();
 });

 this.map.on('move', (ev) => {
 this.currentLatLng = this.map.getCenter();
 });
}

addMarker(lngLat: LngLat, options: MarkerOptions) {

 const color = options.color ||
 '#' + Math.floor(Math.random() * 16777215).toString(16);

 const marker = new Marker({
 ...options,
 color
 })
 .setLngLat(lngLat)
 .addTo(this.map);

 if (options.draggable) {
 marker.on('dragend', (ev) => {
 this.saveToLocalStorage();
 });
 }
}
```

```
 this.markersList.push({marker, color});
 this.saveToLocalStorage();
 }

removeMarker(index: number) {
 const marker = this.markersList[index];
 marker.marker.remove();
 this.markersList.splice(index, 1);
 this.saveToLocalStorage();
}

flyToMarker(index: number) {
 const marker = this.markersList[index];
 this.map.flyTo({
 center: marker.marker.getLatLng()
 });
}

onRangeChange(event: any) {
 this.map.setZoom(event.target.value);
}

zoomIn() {
 this.map.zoomIn();
}

zoomOut() {
 this.map.zoomOut();
}

onAddNewMarker() {
 const makerOptions = {
 draggable: true,
 }
 this.addMarker(this.map.getCenter(), makerOptions);
}

get markers(): MarkerInfo[]{
 return [...this.markersList]
}

saveLocalStorage() {
 const plainMarkers: PlainMarker[] = this.markersList.map((marker) => {
 return {
 lng: marker.marker.getLatLng().lng,
 lat: marker.marker.getLatLng().lat,
 color: marker.color
 }
 });
 localStorage.setItem('plainmarkers', JSON.stringify(plainMarkers));
}

loadFromLocalStorage() {
```

```
const plainMarkers = localStorage.getItem('plainmarkers');
if (!plainMarkers) {
 return;
}

const markers: PlainMarker[] = JSON.parse(plainMarkers);
markers.forEach((marker) => {
 const lngLat = new LngLat(marker.lng, marker.lat);
 const markerOptions = {
 color: marker.color,
 draggable: true
 }
 this.addMarker(lngLat, markerOptions);
});
}
```

Y el Template completo

```
<div #map id="map"></div>

<div class="floating-style floating-range p-2">
 <div class="floating-content">
 <button type="button" class="btn btn-primary" (click)="zoomOut()">-
 </button>
 <input
 type="range"
 class="form-range p-2"
 min="-2"
 max="18"
 [value]="zoom"
 (change)="OnRangeChange($event)"
 >
 <button type="button" class="btn btn-primary" (click)="zoomIn()">+
 </button>
 <button type="button" class="btn btn-primary m-2"
 (click)="onAddNewMarker()">
 +Marker
 </button>
 </div>

 Zoom: {{zoom | number: '1.1-1'}} - ({{currentLatLng.lat | number:
 '1.4-4'}}), {{currentLatLng.lng | number: '1.4-4'}})

</div>

<div class="floating-style floating-markers-list p-2">
 <div class="floating">
 <h5>Markers</h5>
 <hr>
 <ul class="list-group">
```

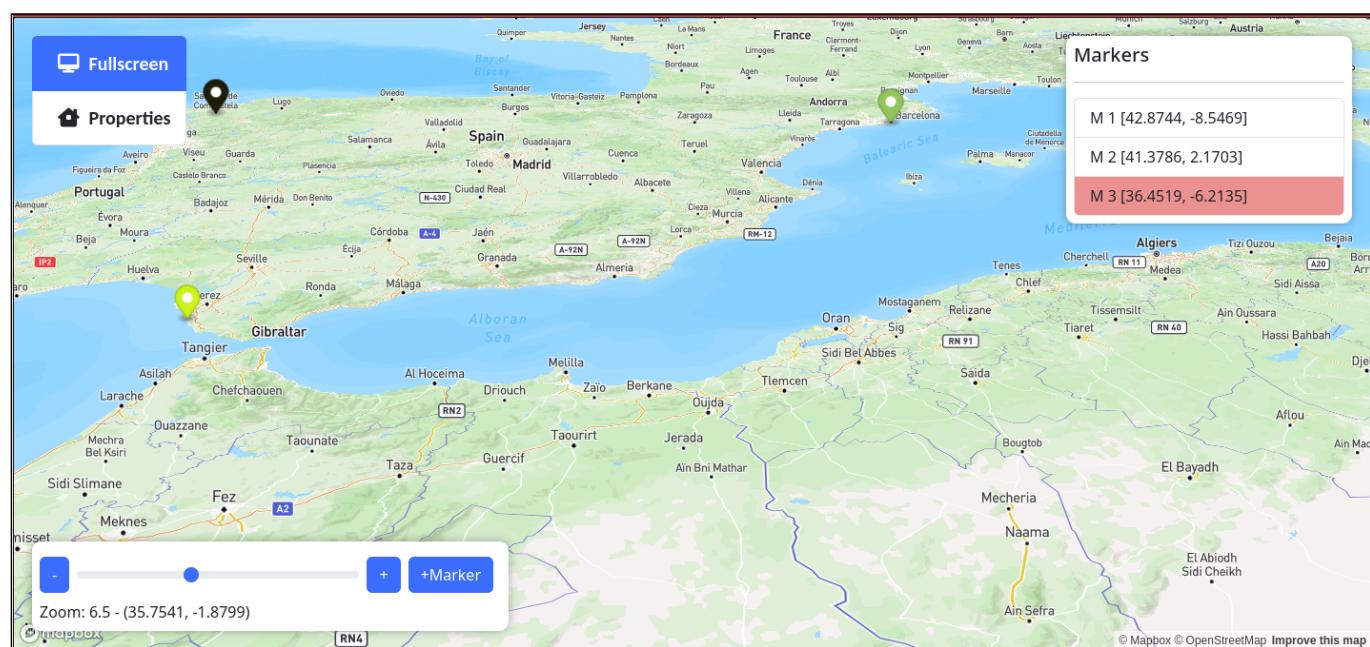
```

<li *ngFor="let marker of markers; let i = index"
 class="list-group-item"
 (dblclick)="removerMarker(i)"
 (click)="flyToMarker(i)">
 M {{ i + 1 }} [{{marker.marker.getLatLng().lat| number:'1.4-4'}}, {{marker.marker.getLatLng().lng | number: '1.4-4'}}]

</div>

```

Resultado final del Full Map View:



## Listado de Propiedades, uso de miniMaps

El objetivo es mostrar una grilla, simulando una lista de propiedades, con un mapa mostrando la ubicación, el mapa lo implementaremos con el componente miniMap.

Este componente recibe una latitud, longitud y se muestra en modo **interactive: false** lo que bloquea las funciones de zoom, drag, etc. Es una imagen estática.

El código del Minimap:

```

@Component({
 selector: 'app-mini-map',
 templateUrl: './mini-map.component.html',
 styleUrls: ['./mini-map.component.css']
})
export class MiniMapComponent {

 @Input() public lngLat: [number, number] = [0, 0];
 @ViewChild('map') divMap?: ElementRef;
}

```

```

public map!: Map;

ngAfterViewInit(): void {
 if(!this.divMap || !this.lngLat) {
 return;
 }

 this.map = new Map({
 accessToken: environment.mapbox_key,
 container: this.divMap.nativeElement,
 style: 'mapbox://styles/mapbox/streets-v12',
 center: this.lngLat,
 zoom: 12,
 interactive: false
 });

 new Marker().setLngLat(this.lngLat).addTo(this.map);
}
}

```

Y el Template:

```
<div #map></div>
```

Un poco de estilos

```

div {
 width: 100%;
 height: 150px;
 margin: 0px;
}

```

Y listo, ahora trabajaremos en el PropertiesComponent, usará un arreglo de propiedades, y usará el miniMap para mostrarla

El código del componente:

```

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

interface House {
 title: string;
 description: string;
 lngLat: [number, number];
 price: number;
}

```

```

}

@Component({
 templateUrl: './properties-page.component.html',
 styleUrls: ['./properties-page.component.css'
})
export class PropertiesPageComponent {

 public houses: House[] = [
 {
 title: 'Casa residencial, Canadá',
 description: 'Bella propiedad en Katana, Canadá',
 lngLat: [-75.92722289474008, 45.280015511264466],
 price: 1000000
 },
 // Se agregan mas elementos en este arreglo
]
}

```

Y el Template:

```

<div class="row mt-3">
 <div class="col-2"></div>
 <div class="col">
 <h3>Listado de Propiedades</h3>
 <hr>
 </div>
</div>

<div class="row mt-2">
 <div class="col-2"></div>
 <div class="col">
 <div class="row">
 <div *ngFor="let house of houses" class="col-4 mb-2">
 <div class="card">

 <app-mini-map class="card-img-top"
[lngLat]="house.lngLat"></app-mini-map>
 <div class="card-body">
 <h5 class="card-title">{{house.title}}</h5>
 <p class="card-text">{{house.description}}</p>
 </div>
 </div>
 </div>
 </div>
 </div>
</div>

```

Como resultado obtenemos:

**Listado de Propiedades**

The screenshot shows a web application interface titled "Listado de Propiedades". On the left, there are two buttons: "Fullscreen" and "Properties", with "Properties" being highlighted. Below these are five cards, each containing a small map from Mapbox and a brief description of a property:

- Casa residencial, Canadá**  
Bella propiedad en Katana, Canadá
- Casa de playa, México**  
Hermosa casa de playa en Acapulco, México
- Apartamento, Argentina**  
Lujoso apartamento en el corazón de Buenos Aires, Argentina
- Local comercial, España**  
Local comercial disponible en Madrid, España, cerca de El Jardín Secreto.
- Casa de Playa, California**  
Bella Casa en California, vive tu Sueño.

Con esto se da por concluida esta sección, hemos aprendido como incorporar otro mapa, hay muchas cosas que se pueden hacer, pero esto representa una buena base a partir de la cual se pueden implementar muchas aplicaciones con Mapas.

## Nueva Sección: Standalone Component

### ¿Qué veremos en esta sección?

Esta sección tiene por objetivo explicar y usar esta nueva característica de Angular, puntualmente veremos:

- ¿Qué son?
- ¿Para qué sirven?
- ¿Cómo usarlos?
- ¿Cómo cargarlos de forma perezosa?
- ¿Cómo conectarlos entre sí?
- ¿Cómo integrarlos en módulos y componentes tradicionales?

### Standalone Como Página

Un "standalone component" en Angular es un componente independiente que puede ser utilizado de manera autónoma sin depender de otros componentes en la misma aplicación. Para crearlo con el CLI usaremos `--standalone true`

El siguiente comando crear un **standalone component** en la ruta `app/alone/pages/`

```
ng g c alone/pages/alone-page --standalone true
```

Exáminemos la estructura de este componente:

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

@Component({
 selector: 'app-alone-page',
 standalone: true,
 imports: [],
 templateUrl: './alone-page.component.html',
 styleUrls: ['./alone-page.component.css']
})
export class AlonePageComponent {
```

El componente es a su vez un módulo que puede ser importado por otro componente.

A continuación cargaremos el componente en modo Lazy, en lugar de loadChilder, usaremos el loadComponent, y cargaremos el AlonePageComponent tal como si fuese un módulo.

```
{
 path: 'alone',
 loadComponent: () => import('./alone/pages/alone-page/alone-
page.component').then(m => m.A lonePageComponent)
}
```

Si navegamos al path <http://localhost:4200/alone> veremos nuestro componente, y no lo hemos importado en ningún lugar, solamente lo hemos cargado como si fuese un módulo más.

## Standalone Como Componente reutilizable

Creemos un nuevo componente

```
ng g c alone/components/counter-alone --standalone
```

Agregamos un simple contador

```
export class CounterAloneComponent {

 public counter: number = 0;

 decrement() {
 this.counter--;
 }
}
```

```

increment() {
 this.counter++;
}
}

```

Un template:

```

<h1>Counter {{ counter }}</h1>
<button (click)="decrement()">-</button>
<button (click)="increment()">+</button>

```

Si queremos usar este Standalone componente dentro de otro standalone component, debemos importar el componente, en este caso el Counter dentro del componente padre,

```

@Component({
 selector: 'app-alone-page',
 standalone: true,
 imports: [CounterAloneComponent],
 templateUrl: './alone-page.component.html',
 styleUrls: ['./alone-page.component.css']
})
export class AlonePageComponent {
}

```

Y luego ya podemos usarlo en el template:

```

<div class="row">
 <counter-alone></counter-alone>
</div>

```

## Standalone en Módulos Tradicionales

Si un componente Standalone debe ser utilizando dentro de un compomente tradicional, debemos importar el Standalone Component dentro del módulo que contiene el componente padre. Por ejemplo, en el siguiente código, importamos el CounterAloneComponent dentro del **MapsModule** a partir de ese momento, cualquier componente definido dentro de **MapsModule** podrá usar dicho componente.

```

@NgModule({
 declarations: [
 // Other declarations
],
 imports: [

```

```
 CommonModule,
 MapsRoutingModule,
]
})
export class MapsModule { }
```

## Ejercicio práctico

Agregaremos un Like Button, en cada propiedad a la venta en nuestro PropertiesComponent, usando standalone component. Este botón acepta la cantidad de likes iniciales y emite un evento cuando el visitante da click en el botón

Primero creamos el componente:

```
ng g c standalone/components/like-btn --standalone
```

El código del componente sería:

```
@Component({
 selector: 'like-btn',
 standalone: true,
 imports: [],
 templateUrl: './like-btn.component.html',
 styleUrls: ['./like-btn.component.css'
})
export class LikeBtnComponent {

 @Input() likes: number = 0;

 @Output() onLike: EventEmitter<number> = new EventEmitter<number>();

 liked() {
 this.likes++;
 this.onLike.emit(1);
 }
}
```

El template:

```
<button
 type="button"
 class="btn btn-primary"
 (click)="liked()">
 <i class="fa-regular fa-thumbs-up"></i>
 {{ likes }}
</button>
```

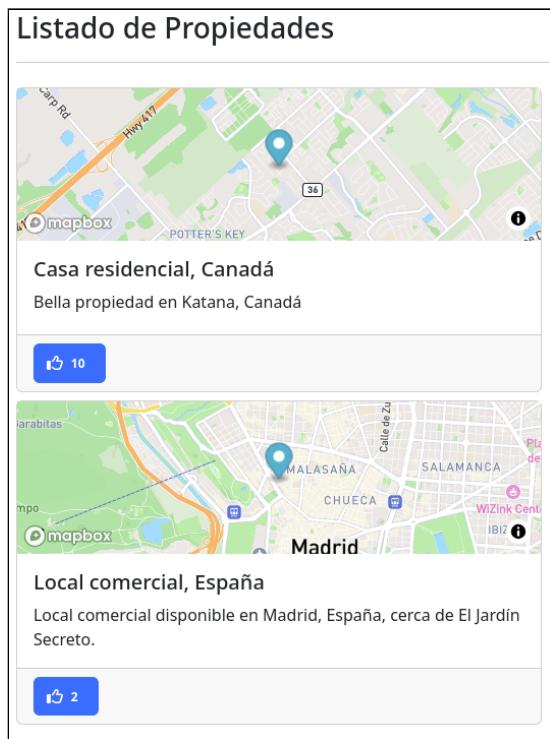
Lo importamos en el **MapsModule** y lo usamos en nuestro template

```
<div class="card-footer">
 <like-btn [likes]="" (onLike)="liked(i)"></like-btn>
</div>
```

Finalmente debemos definir el método del lado del **PropertiesPageComponent** que va a procesar el evento final, puede ser una llamada AJAX para aumentar los likes de dicha propiedad.

```
liked(index: number) {
 console.log(this.houses[index].title + ' liked');
}
```

Resultado:



## Transformar el SideMenu en Standalone

Debemos mover todo el directorio del componente, que se encuentra dentro del módulo de maps, al directorio donde tenemos los standalone component.

AActualizamos las referencias y agregamos al SideMenu:

```
@Component({
 standalone: true,
```

```
imports: [CommonModule, RouterModule]
})
```

CommonModule lo importamos porque usamos el NGFor para iterar sobre las opciones del menú, y el RouterModule porque agregamos links en cada opción.

```
<ul class="list-group">
 <li
 *ngFor="let item of menuItems"
 class="list-group-item"
 [routerLink]=[item.router]
 routerLinkActive="active"
 >
 <i class="menu-icon {{item.icon}}></i>{{ item.name}}


```

Con esto ya tenemos el componente en modo standalone, ahora debemos hacer los cambios donde sea que debamos utilizarlo, primero en el **MapsModule**, porque el menú lo usamos en el **MapsLayoutComponent** que está dentro de dicho módulo, y segundo lo agregaremos en nuestro **AlonePageComponent** dado que este también es un standalone component, entonces lo importamos directamente en dicho componente:

```
@Component({
 selector: 'app-alone-page',
 standalone: true,
 imports: [CounterAloneComponent, SiteMenuComponent],
 templateUrl: './alone-page.component.html',
 styleUrls: ['./alone-page.component.css']
})
export class AlonePageComponent {
```

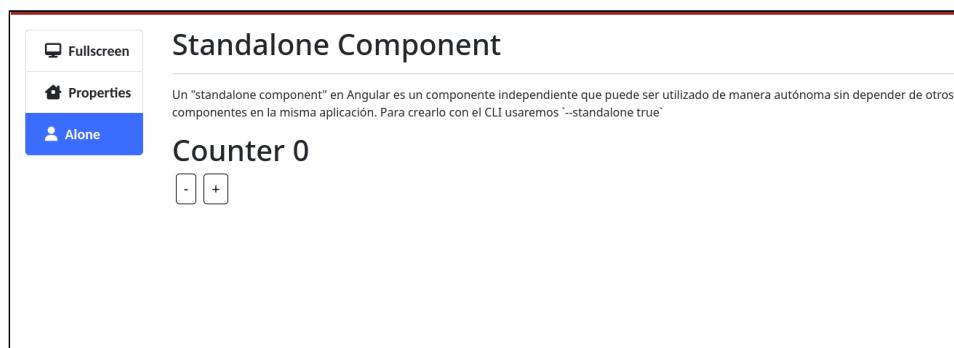
Y luego lo ubicamos en el template, y finalmente movemos algunos estilos CSS que habíamos creado a nivel del Menu, como componente del MAPA, para que ahora sean parte del Standalone Components, de esta forma nuestro menú contiene todo lo necesario para renderizarse correctamente donde se necesite

```
ul {
 left: 20px;
 position: fixed;
 top: 20px;
 z-index: 999;
}
```

Antes, debemos agregar una opción más al MenuItems para poder navegar al Alone Page

```
public menuItems: MenuItems[] = [
 { router: '/maps/fullscreen', name: 'Fullscreen', icon: 'fa-solid fa-desktop' },
 { router: '/maps/properties', name: 'Properties', icon: 'fa-solid fa-house-chimney-window' },
 { router: '/alone', name: 'Alone', icon: 'fa-solid fa-user' }
]
```

El resultado:



## Nueva Sección - Directivas personalizadas y Signals

### ¿Qué veremos en esta sección?

En esta sección vamos a crear directivas personalizadas en Angular. La idea será crear una directiva robusta que nos sirva para resumir la forma cómo los errores en los formularios son mostrados, y así no tener que colocar infinidad de condiciones dentro del HTML.

Esta directiva que crearemos, es de uso real y dará una buena idea de qué podemos hacer con ellas.

Además iniciaremos el estudio de los Signals:

Una señal es un wrapper alrededor de un valor que puede notificar a los consumidores interesados cuando ese valor cambia. Las señales pueden contener cualquier valor, desde primitivas simples hasta estructuras de datos complejas.

Disponibles a partir de la versión 16 de Angular.

### Nueva APP

```
ng new 10-DirectivesApp --standalone false --routing
ng g m products --routing
ng g c products/pages/product-page --skip-selector
ng g m shared
* ng g d shared/directives/customLabel
```

\* El comando se ejecuta posteriormente

La estructura de la aplicación se describe a continuación

```
src/app/
 └── app.component.css
 └── app.component.html
 └── app.component.spec.ts
 └── app.component.ts
 └── app.module.ts
 └── app-routing.module.ts
 └── products
 └── pages
 └── product-page
 └── product-page.component.css
 └── product-page.component.html
```

```

 | └── product-page.component.ts
 ├── products.module.ts
 └── products-routing.module.ts
└── shared
 ├── directives
 | ├── custom-label.directive.spec.ts
 | └── custom-label.directive.ts
 └── shared.module.ts

```

Agregamos las rutas

## AppRoutingModule

```

const routes: Routes = [
 {
 path: '',
 loadChildren: () => import('./products/products.module').then(m =>
m.ProductsModule)
 },
 {
 path: '**',
 redirectTo: '/products'
 }
];

```

## ProductsRoutingModule

```

const routes: Routes = [
 {
 path: '',
 children: [
 { path: 'product', component: ProductPageComponent},
 { path: '**', redirectTo: 'product'}
]
 }
];

```

## Formulario Reactivo

Para crear el formulario inyectaremos el formBuilder, anteriormente lo hemos realizado de esta forma

```
constructor(private formBuilder: FormBuilder) { }
```

También podemos inyectarlo de esta forma

```
private formBuilder = Inject(FormBuilder);
```

Agregamos un par de métodos:

```
@Component({
 templateUrl: './product-page.component.html',
 styleUrls: ['./product-page.component.css']
})
export class ProductPageComponent {

 private formBuilder = inject(FormBuilder);

 public color: string = 'green';

 public productForm: FormGroup = this.formBuilder.group({
 name: ['', [Validators.required, Validators.minLength(6),
 Validators.email]],
 });

 public changeColor(): void {
 this.color = this.generateRandomColor();
 }

 public generateRandomColor(): string {
 return '#' + Math.floor(Math.random()*16777215).toString(16);
 }
}
```

Lo enlazamos en el HTML.

## Crear Directiva

En el módulo de Shared vamos a crear nuestra primer directiva:

```
ng g d shared/directives/customLabel
CREATE src/app/shared/directives/custom-label.directive.spec.ts (245 bytes)
CREATE src/app/shared/directives/custom-label.directive.ts (151 bytes)
UPDATE src/app/shared/shared.module.ts (296 bytes)
```

El código generado para nuestra primer directiva se muestra a continuación:

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

@Directive({
 selector: '[appCustomLabel]'
})
```

```
export class CustomLabelDirective {

 constructor() { }

}
```

Notar que el selector se coloca entre llaves cuadradas [] y se nombra usando el formato loweCamelCase. A diferencia del selector del componente que iba entre comillas simples y separado por guiones (kebab-case)

Además de la anotación **@Directive** esta se trata como un componente en nuestro módulo, es decir debemos declararla e importarla en el módulo **SharedModule**

```
@NgModule({
 declarations: [
 CustomLabelDirective
],
 imports: [
 CommonModule
],
 exports: [
 CustomLabelDirective
]
})
export class SharedModule { }
```

Luego importamos dicho módulo en el **ProductsModule** para usar la nueva directiva en el formulario.

## Uso de nuestra directiva

Tenemos nuestro formulario, con un campo **name** y un **span** que deberá mostrar el error asociado al campo

```
<div class="col-auto">
 <label>Nombre:</label>
</div>
<div class="col-auto">
 <input formControlName="name" type="text" class="form-control">
</div>
<div class="col-auto">

</div>
```

Agregamos nuestra directiva **customLabel** al span que debe de mostrar el error. Por el momento nuestra directiva no hace nada, por tanto no vemos ningún cambio.

## Implementar la Directiva

Vamos a nuestro código de la directiva y aplicamos estos cambios:

```
@Directive({
 selector: '[CustomLabel]'
})
export class CustomLabelDirective {

 private _errors: ValidationErrors | null | undefined = null;
 @Input() set errors(value: ValidationErrors | null | undefined) {
 this._errors = value;
 this.setErrorMessages()
 }

 private htmlElement?: ElementRef<HTMLElement>;

 constructor(private el: ElementRef<HTMLElement>) {
 this.htmlElement = el;
 }

 setErrorMessages(): void {
 if (!this.el) return;

 if (!this._errors) {
 this.el.nativeElement.textContent = '';
 return;
 }

 const errors = Object.keys(this._errors || {});

 if (errors.includes('required')) {
 this.el.nativeElement.textContent = 'This field is required';
 return;
 }

 if (errors.includes('minlength')) {
 const reqLen = this._errors['minlength'].requiredLength;
 this.el.nativeElement.textContent =
 `This field must be at least ${reqLen} characters long`;
 return;
 }

 if (errors.includes('email')) {
 this.el.nativeElement.textContent =
 'This field must be a valid email';
 return;
 }
 }
}
```

Hemos definido un `@Input() set errors ...` por lo tanto debemos de pasar una lista de Errores.  
Nuestro template debe de cambiar:

```
<span customLabel
 [errors]="productForm.get('name')?.errors">

```

Listo, de esta forma podremos ver el mensaje de error asociado al campo, pero esta vez usando una Directiva personalizada

# Nueva Sección: Signals:

## ¿Qué veremos en esta sección?

En esta sección empezaremos a utilizar las señales en Angular

(Requerimiento mínimo Angular 16+)

Puntualmente veremos:

- Utilización
- Modificación
- Mutación
- Actualización

### Definición: Signals:

Una señal es un Wrapper alrededor de un valor que puede notificar a los consumidores interesados cuando ese valor cambia. Las señales pueden contener cualquier valor, desde primitivas simples hasta estructuras de datos complejas.

Entonces, cual es la diferencia de los Signals con los Observables o EventEmitters?

Los Observables son potentes pero pueden ser complicados y propensos a errores si no se usan correctamente. Los **Signals**, en cambio, están diseñados para ser más simples y accesibles, facilitando el manejo de valores reactivos en situaciones más directas, especialmente en el contexto de la interfaz de usuario.

Los **Observables** de RxJS son una abstracción más general para manejar flujos de datos asíncronos, **EventEmitter** es específico de Angular y se utiliza para la comunicación entre componentes o servicios dentro de la misma aplicación Angular. Observables pueden ser utilizados en un contexto más amplio y son parte de **RxJS**, mientras que **EventEmitter** está diseñado específicamente para Angular y se utiliza para la comunicación entre componentes.

La elección entre **Observables** y **Signals** dependerá de la complejidad de los flujos de datos y de las necesidades específicas del desarrollo en Angular.

## Nuevo Módulo

Creamos un módulo

```
ng g m signals --routing
```

Y Creamos la siguiente estructura:

```

src/app/signals/
├── components
├── interfaces
└── layout
 └── layout
 ├── layout.component.css
 ├── layout.component.html
 └── layout.component.ts
└── pages
 ├── counter-page
 │ ├── counter-page.component.css
 │ ├── counter-page.component.html
 │ └── counter-page.component.ts
 ├── properties-page
 │ ├── properties-page.component.css
 │ ├── properties-page.component.html
 │ └── properties-page.component.ts
 └── user-info-page
 ├── user-info-page.component.css
 ├── user-info-page.component.html
 └── user-info-page.component.ts
└── services
└── signals.module.ts
└── signals-routing.module.ts

```

Nuestro Sistema de Rutas:

### SignalsRoutingModule

```

const routes: Routes = [
{
 path: '',
 component: LayoutComponent,
 children: [
 {path: 'counter', component: CounterPageComponent},
 {path: 'user-info', component: UserInfoPageComponent},
 {path: 'properties', component: PropertiesPageComponent},
 {path: '**', redirectTo: 'counter'}
]
};
];

```

### AppRoutingModule

```

const routes: Routes = [
{
 path: 'products',
 loadChildren: () => import('./products/products.module').then(m =>
m.ProductsModule)

```

```

},
{
 path: 'signals',
 loadChildren: () => import('./signals/signals.module').then(m =>
m.SignalsModule)
},
{
 path: '**',
 redirectTo: '/signals'
}
];

```

Agregamos un menú lateral:

```

@Component({
 selector: 'app-side-menu',
 templateUrl: './side-menu.component.html',
 styleUrls: ['./side-menu.component.css'
})
export class SideMenuComponent {

 public menuItems: MenuItem[] = [
 { title: 'Contador', route: 'counter' },
 { title: 'Usuario', route: 'user-info' },
 { title: 'Mutaciones', route: 'properties' }
]
}

```

Y lo Implementamos en el Template:

```

<ul class="list-group">
 <li
 *ngFor="let item of menuItems"
 class="list-group-item"
 [routerLink]=[item.route]
 routerLinkActive="active">
 {{item.title}}


```

## Crear una primer Señal

Vamos a transformar el menuItems en un **signal** es decir, vamos a envolverlo en un wrapper para que este emita valores cuando algo cambien dentro del menú:

```

public menuItems = signal<MenuItem[]>([
 { title: 'Contador', route: 'counter' },
 /

```

```
{ title: 'Usuario', route: 'user-info' },
{ title: 'Mutaciones', route: 'properties' }
]);
```

**menuItemsSignal** directamente no es un arreglo de **MenuItem** ahora es una señal.

Si imprimimos esto en el template: `{{menuItemsSignal}}` observaremos

```
() => { producerAccessed(node); return node.value; }
```

Si lo imprimimos como una función, obtenemos una lista de objetos, que son los elementos del arreglo

```
[object Object],[object Object],[object Object]
```

Así que para tener nuevamente nuestro menú, debemos cambiar el NgFor de esto:

```
*ngFor="let item of menuItems"
```

a esto

```
*ngFor="let item of menuItems()"
```

## Counter Signal

Definamos un Singla Counter

```
public counter = signal(0);
```

Podemos incrementar en 1 el valor del Counter de esta forma:

```
this.counter.set(this.counter() + 1);
```

O mejor aún con el update

```
this.counter.update((value) => value + 1);
```

Ahora, si queremos imprimir el cuadrado del valor, podríamos pensar en multiplicar el valor actual por el mismo valor

```
<h2>Squeare Counter {{ counter() * counter() }}</h2>
```

Una solución mejor es usar una función Computada:

```
public squeareComputed = computed(() => this.counter() ** 2);
```

Y cambiamos el template

```
<h2>Squeare Counter {{ squeareComputed() }}</h2>
```

**squeareComputed** es una función computada que se va a ejecutar cuando el valor de nuestro signal **counter** cambie.

## Nuevo Servicio

Agregamos un servicio

```
ng g s signals/services/user-service
```

Importamos el **HttpClientModule** en el AppModule.

Usaremos un FakeAPI <https://reqres.in/api/users/1>

```
{
 "data": {
 "id": 2,
 "email": "janet.weaver@reqres.in",
 "first_name": "Janet",
 "last_name": "Weaver",
 "avatar": "https://reqres.in/img/faces/2-image.jpg"
 },
 "support": {
 "url": "https://reqres.in/#support-heading",
 "text": "To keep ReqRes free, contributions towards server costs are appreciated!"
 }
}
```

Y generamos las interfaces

```
export interface SingleUserResponse {
 data: User;
 support: Support;
}

export interface User {
 id: number;
 email: string;
 first_name: string;
 last_name: string;
 avatar: string;
}

export interface Support {
 url: string;
 text: string;
}
```

Y agregamos este código al servicio:

```
@Injectable({
 providedIn: 'root'
})
export class UserServiceService {

 private httpClient = inject(HttpClient);
 private baseUrl = 'https://reqres.in/api/users';

 getUserById(id: number): Observable<User> {
 return this.httpClient.get<SingleUserResponse>(` ${this.baseUrl}/ ${id}`)
 .pipe(
 map(response => response.data)
);
 }
}
```

Vamos a crear una pantalla, con dos botones, los cuales al presionar cualquiera de ellos se generara un ID (anterior o siguiente), y cuando ese valor cambia, llamaremos nuestro servicio para obtener la data del usuario con ese ID, posteriormente mostramos los datos del usuario en pantalla:

El Template:

```
<h1>Información del Usuario</h1>
<hr>
```

```

<div class="row">
 <div class="col-4">
 <h3>Usuario Actual: {{userId()}}</h3>
 <button
 (click)="loadUser(userId() - 1)"
 class="btn btn-primary">
 Anterior
 </button>

 <button
 (click)="loadUser(userId() + 1)"
 class="btn btn-primary">
 Siguiente
 </button>
 </div>
 <div class="col-8" *ngIf="currentUser()">
 <h3>Usuario</h3>
 <hr>
 <p>Nombre: {{ fullName() }}</p>
 <p>Correo:{{ currentUser()?.email}}</p>

 </div>
 <div *ngIf="!userWasFound()" class="col-8">
 <h3>Usuario no encontrado</h3>
 </div>
</div>

```

Y el código completo del componente:

```

import { HttpClient } from '@angular/common/http';
import { Component, OnInit, computed, inject, signal } from '@angular/core';
import { SingleUserResponse, User } from '../../../../../interfaces/user-request.interface';
import { Observable, catchError, map, of } from 'rxjs';
import { UserServiceService } from '../../../../../services/user-service.service';

@Component({
 selector: 'app-user-info-page',
 templateUrl: './user-info-page.component.html',
 styleUrls: ['./user-info-page.component.css']
})
export class UserInfoPageComponent implements OnInit{

 private userService = inject(UserServiceService);

 public userId = signal<number>(1);

 public currentUser = signal<User | undefined>(undefined);
}

```

```

public userWasFound = signal<boolean>(true);

public fullName = computed<string>(() => {
 return this.currentUser()?.first_name +
 ' ' +
 + this.currentUser()?.last_name);

ngOnInit(): void {
 this.loadUser(this.userId());
}

loadUser(id: number) {
 if (id < 1) {
 this.userWasFound.update(() => false);
 return;
 }

 this.userId.set(id);
 this.currentUser.update(() => undefined);

 this.userService.getUserById(id).subscribe({
 next: (user) => {
 this.currentUser.update(() => user);
 this.userWasFound.update(() => true);
 },
 error: (err) => {
 this.userWasFound.update(() => false);
 this.currentUser.update(() => undefined);
 }
 })
}
}

```

Analicemos este código, primero hacemos una inyección de dependencias, esta vez no usamos el constructor.

```
private userService = inject(UserServiceService);
```

Luego tenemos varios **signals**

```
public userId = signal<number>(1);
public currentUser = signal<User | undefined>(undefined);
public userWasFound = signal<boolean>(true);
```

**UserId** simplemente es de tipo numérico, cada vez que cambiamos su valor, se hará una búsqueda por medio del servicio, esto se debe a la acción de los botones en el template:

```

<button
 (click)="loadUser(userId() + 1)"
 class="btn btn-primary">
 Siguiente
</button>

```

EL **currentUser** almacena el usuario actual y el **userWasFound** simplemente es un flag true/false. Recordemos que estas son variables, pero que estan envueltas dentro de la estructura de un **signal**

El método **loadUser** hace lo siguiente

- Verifica el ID
- Establece el nuevo valor de **userId**
- Borra cualquier dato almacenado en **currentUser**
- Usa el Servicio para obtener el Usuario según el ID

Anteriormente, usabamos **Pipes** para capturar los errores, en este caso lo hacemos de una manera mas limpia, dentro del mismo **subscribe** usamos:

- Next: Para obtener el siguiente valor emitido, este va a contener el usuario retornado por el servicios
- Error: En caso de ocurrir un error lo capturamos con esta etiqueta.

El Resultado es la siguiente pantalla:

## Mutaciones

Creamos el siguiente template

```

<h1>Mutaciones</h1>
<hr>

<div class="row">
 <div class="col-6">

 <input
 type="text"
 class="form-control mb-2"

```

```
#txtEmail
placeholder="Email"
(input)="onFieldUpdated('email', txtEmail.value)"
[value]="user().email">

<input
 type="text"
 class="form-control mb-2"
#txtFirstName
placeholder="First Name"
(input)="onFieldUpdated('first_name', txtFirstName.value)"
[value]="user().first_name">

<input
 type="text"
 class="form-control mb-2"
#txtLastName
placeholder="Last Name"
(input)="onFieldUpdated('last_name', txtLastName.value)"
[value]="user().last_name">

</div>
<div class="col-6">
 <pre>
 {{ user() | json }}
 </pre>

 <pre>
 {{ fullName() }}
 </pre>
</div>
</div>
```

Y escribamos este componente:

```
export class PropertiesPageComponent {

 public user = signal<User>({
 id: 2,
 email: 'janet.weaver@reqres.in',
 first_name: 'Janet',
 last_name: 'Weaver',
 avatar: 'https://reqres.in/img/faces/2-image.jpg'
 });

 public fullName = computed<string>(() => {
 return this.user()?.first_name +
 ' ' +
 this.user()?.last_name);
 }

 onFieldUpdated(field: string, value: string) {
 this.user.set({ ...this.user(), [field]: value });
 }
}
```

```
 }
}
```

Aca tenemos un **user** que es un signal y se inicializa con un objeto tipo **User** con sus respectivos datos.

Además tenemos una variable **fullName** que es computada y une el first y last Name

Y cada uno de los Input del formulario estan enlazados a cada una de las propiedades del objeto, (email, first\_name y last\_name) además cada vez que cambiamos el valor del Input llamamos el método **onFieldUpdated**

Esta forma de actualizar nuestro objeto es potencialmente peligrosa, ya que se puede inyectar propiedades adicionales

Podemos aplicar una validación, que consiste en que el parámetro **field** sea únicamente de un tipo string pero que solo pueda tomar los campos de la interfaz User, es decir, **\*\*id, email, first\_name ... \*\*\***

```
onFieldUpdated(field: keyof User, value:string){}
```

Otra forma similar de implementar el método **onFieldUpdated** es con el Update,

```
this.user.update((current) => ({...current, [field]: value}));
```

Este código funciona igual, pero es la misma implementación.

Otra forma, quizás mas extensa y tediosa pero más segura es crear algo similar a esto

```
onFieldUpdated(field: keyof User, value:string){
 this.user.update((current) => {
 switch(field){
 case 'email':
 current.email = value;
 break;
 case 'first_name':
 current.first_name = value;
 break;
 case 'last_name':
 current.last_name = value;
 break;
 }
 return current;
 })
}
```

Esto es mas seguro, porque explicitamente indicamos que campos vamos a actualizar, pero podemos lograr esto haciendo una validación del **field** y seguir usando el operador spread.

El resultado de esto es que cada vez que escribimos en la caja de texto, el valor se actualiza en la propiedad correspondiente y a la vez la variable computada FullName es ejecutada:

```

 {
 "id": 2,
 "email": "janet.weaver@reqres.in",
 "first_name": "Janet Jackson",
 "last_name": "Weaver",
 "avatar": "https://reqres.in/img/faces/2-image.jpg"
 }

```

Janet Jackson Weaver

NOTA: Solo una aclaración, en el método onFieldUpdated usando el Switch final, no recalcula el FullName, porque estamos modificando directamente el current Value, mobx-state-tree (MST) no detecta el cambio cuando actualizamos directamente las propiedades del objeto observable.

Lo que deberíamos hacer es retornar un nuevo objeto con los valores modificados.

Por ejemplo:

```

onFieldUpdated(field: keyof User, value:string){
 //this.user.set({...this.user(), [field]: value});
 //this.user.update((current) => ({...current, [field]: value}));

 this.user.update((current) => {

 var newUser = {...current};
 switch(field){
 case 'email':
 newUser.email = value;
 break;
 case 'first_name':
 newUser.first_name = value;
 break;
 case 'last_name':
 newUser.last_name = value;
 break;
 }
 return newUser;
 })
}

```

## Efectos con Señales

Un efecto es una función que se ejecuta automáticamente en respuesta a algún cambio en el estado observado. En el siguiente caso, el efecto se dispara cuando hay cambios en el signal llamado user.

```
public userChangeEffect = effect(() => {
 console.log(this.user().first_name + ' has been updated');
});
```

La dependencia entre el efecto y el cambio en user se establece automáticamente debido a cómo **mobx-state-tree** rastrea las dependencias. Cuando defines un efecto con `effect(() => { ... })`, el código dentro de la función se ejecuta de manera reactiva cada vez que alguna propiedad observada dentro del bloque de efecto cambia.

En tu caso, `this.user()` es una referencia a un signal, y cuando llamas a `this.user().first_name` dentro de la función del efecto, **mobx-state-tree** registra automáticamente una dependencia entre el efecto y user.

Entonces, cada vez que se actualizan cualquier propiedad del User, el efecto se dispara automáticamente porque está observando esas propiedades. La reactividad de **mobx-state-tree** se encarga de mantener esta relación de dependencia de manera eficiente.

Los effect tienen una limpieza automática, cada vez que se destruye el componente, se limpian los efectos.

# Nueva Sección - Creando un Backend con NestJS

¿Qué veremos en esta sección?

- Crearemos un API REST con NestJS, dado las similitudes entre Angular y NestJS, esta será el inicio de dos secciones en la que construiremos un APP, que consuma un REST API.

## Inicio del Backend

Iniciamos configurar las herramientas necesarias para crear nuestro API Rest, instalamos el CLI de nestjs:

```
npm i -g @nestjs/cli
```

Luego creamos nuestra APP

```
nest new nestBackend
```

Seleccionamos NPM.

```
npm run start:dev
```

## Configurar Docker en Fedora

```
sudo dnf -y install dnf-plugins-core
sudo dnf config-manager --add-repo
https://download.docker.com/linux/fedora/docker-ce.repo

sudo dnf install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin

sudo systemctl start docker
```

## Docker Desktop

```
sudo dnf install /home/fcruz/Downloads/docker-desktop-4.27.2-x86_64.rpm
```

Configuramos el File Sharing

```
grep "$USER" /etc/subuid >> /dev/null 2>&1 || (echo "$USER:100000:65536" |
sudo tee -a /etc/subuid)
grep "$USER" /etc/subgid >> /dev/null 2>&1 || (echo "$USER:100000:65536" |
sudo tee -a /etc/subgid)
```

En fedora, si al abrir la APP Desktop no funciona, intenta reiniciar docker, o bien en la misma APP desktop, abre las configuraciones y resetea los valores de fabrica, y reinicia nuevamente la APP

```
gpg --generate-key
Add Real name
Add Email address

pub ed##### 2024-02-11 [SC] [expires: 2026-02-10]
pass init ed#####

Sigin in Desktop and follow instructions
```

## Mongo Compass

---

Descargar el RPM (<https://www.mongodb.com/try/download/compass>)  
[<https://www.mongodb.com/try/download/compass>]

```
npm install mongodb-compass-1.42.0.x86_64.rpm
```

## Docker Compose

---

Vamos a ir a nuestro proyecto de nestjs y crearemos un archivo en la raíz del mismo

### **docker-compose.yml**

```
version: '3'

services:
 db:
 container_name: mongo-db
 image: mongo:7.0.5
 volumes:
 - ./mongo:/data/db
 ports:
 - "27017:27017"
 restart: always
```

Ejecutamos el comando siguiente en la terminal:

```
docker compose up -d
```

-d : Detached, permite cerrar la terminal y dejar

El comando anterior debí correrlo como root en fedora, y antes fue necesario hacer un:

```
$ docker login
```

```
docker compose up -d
[+] Running 9/9
 ✓ db 8 layers [██████████] 0B/0B Pulled
27.8s
 ✓ 57c139bbda7e Pull complete
3.5s
 ✓ 2a7c884ecb1c Pull complete
0.7s
 ✓ ddcbc3e219f6 Pull complete
1.1s
 ✓ 347abad2978b Pull complete
2.0s
 ✓ 33ad7d6a71b8 Pull complete
1.9s
 ✓ 46b764746687 Pull complete
2.5s
 ✓ 0d22e1cb5ef2 Pull complete
22.2s
 ✓ fef1dc21c099 Pull complete
3.3s
[+] Running 1/2
 ! Network nest-backend_default Created
0.9s
 ✓ Container mongo-db Started
```

NOTA para Fedora: Si no llegamos a ver el contenedor en Docker-Desktop es posible que debamos abrir sesión y para ellos debemos seguir esta guía:

(<https://docs.docker.com/desktop/get-started/#credentials-management-for-linux-users>)  
[<https://docs.docker.com/desktop/get-started/#credentials-management-for-linux-users>]

```
rm -rf ~/.password-store/docker-credential-helpers
pass init ed#####
docker login
```

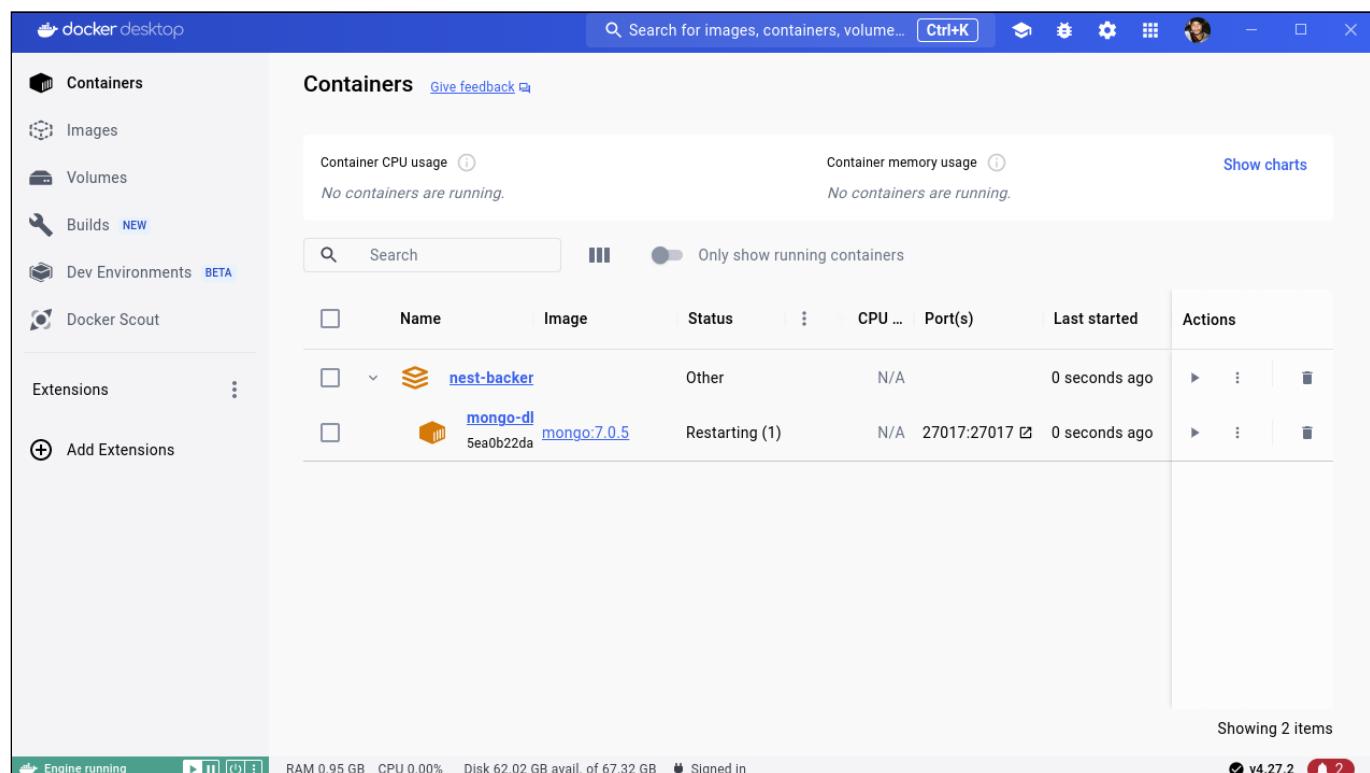
```
docker compose up -d
docker ps
```

Tambien puede ser necesario agregar nuestro usuario al grupo de Docker:

Como Root:

```
sudo usermod -aG docker $USER
sudo chown -R $USER:$USER ~/.docker/
rm -rf ~/.docker/config.json
sudo systemctl restart docker
```

Al final debemos de observar nuestro contenedor:



## Crear CRUD con Nestjs

Borramos el Controller y el Service y vamos a crear un **resource** Un recurso en NestJS permite agregar de una sola vez, todas las operaciones necesarias para una entidad de datos. Por ejemplo, si necesitamos aplicar un CRUD para nuestro proceso de Autenticación, haremos lo siguiente

```
[fcruz@fedora nest-backend]$ nest g resource auth
? What transport layer do you use? (Use arrow keys)
❯ REST API
 GraphQL (code first)
 GraphQL (schema first)
 Microservice (non-HTTP)
 WebSockets
```

```
Would you like to generate CRUD entry points? Yes
CREATE src/auth/auth.controller.spec.ts (556 bytes)
CREATE src/auth/auth.controller.ts (883 bytes)
CREATE src/auth/auth.module.ts (241 bytes)
CREATE src/auth/auth.service.spec.ts (446 bytes)
CREATE src/auth/auth.service.ts (607 bytes)
CREATE src/auth/dto/create-auth.dto.ts (30 bytes)
CREATE src/auth/dto/update-auth.dto.ts (169 bytes)
CREATE src/auth/entities/auth.entity.ts (21 bytes)
UPDATE package.json (1984 bytes)
UPDATE src/app.module.ts (191 bytes)
✓ Packages installed successfully.
[fcruz@fedora nest-backend]$
```

Notar como el CLI de Nestjs nos ha generado el controlador, los modulos, el servicio etc etc..

La estructura del proyecto actual sería:

```
src/
├── app.module.ts
└── auth
 ├── auth.controller.ts
 ├── auth.module.ts
 ├── auth.service.ts
 ├── dto
 │ ├── create-auth.dto.ts
 │ └── update-auth.dto.ts
 └── entities
 └── auth.entity.ts
└── main.ts
```

## Conectar Nestjs con Mongo

La Documentación de nestjs provee suficiente información para implementar nuestra conexión con el manejador de base de datos, en este caso con MongoDB

<https://docs.nestjs.com/techniques/mongodb>

Instalamos el ORM De Mongo

```
npm i @nestjs/mongoose mongoose
```

Actualizamos nuestro AppModule

```
import { Module } from '@nestjs/common';
import { AuthModule } from './auth/auth.module';
```

```
import { MongooseModule } from '@nestjs/mongoose';

@Module({
 imports: [AuthModule,
 MongooseModule.forRoot('mongodb://localhost:27017')],
 controllers: [],
 providers: [],
})
export class AppModule {}
```

Se pueden usar variables de entorno para especificar el URL de conexión

## Usar Variables de Entorno

Creamos un archivo .env y .env.template y agregamos las configuraciones necesarias, por el momento:

```
MONGO_URI=mongodb://localhost:27017
```

.env debe ser agregado al .gitignore

Instalamos la siguiente dependencia

```
npm install @nestjs/config
```

Luego, en nuestro app.module.ts, en la sección de los imports

```
@Module({
 imports: [
 ConfigModule.forRoot(),
 MongooseModule.forRoot(process.env.MONGO_URI),
 AuthModule],
 controllers: [],
 providers: [],
})
export class AppModule {
```

## Crear modelos y esquema

Renombramos al entidad Auth.entity.ts por user.entity.ts

```

import { Prop, Schema, SchemaFactory } from "@nestjs/mongoose";

@Schema()
export class User {
 @Prop({ unique: true, required: true })
 email: string;

 @Prop({ required: true })
 name: string;

 @Prop({ minlength: 6, required: true })
 password: string;

 @Prop({ default: true })
 isActive: boolean;

 @Prop({ type: [String], default: ['user'] })
 role: string[];
}

export const UserSchema = SchemaFactory.createForClass(User);

```

Algunas anotaciones, primero **@Schema()** Cada esquema se asigna a una colección de **MongoDB** y define la forma de los documentos dentro de esa colección.

Cada propiedad de la clase contiene un **@Prop**, por medio del cual definimos las características de dicha propiedad, por ejemplo si es un valor único en la colección, si es requerido, si tiene un ancho mínimo, etc.

En el caso del **role**, dado que es un arreglo, definimos el tipo de datos **String** de la base de datos y el valor inicial.

Para finalizar en el **AuthModule**, en la sección de Imports, le indicamo que deseamos agregar el Schema a Mongoose.

```

...
imports: [
 MongooseModule.forFeature([
 { name: 'User', schema: UserSchema }])
]
})
export class AuthModule {}

```

Esto hará que el Schema y la entidad users se muestren en MongoDB

Name and Definition	Type	Size	Usage	Properties
> _id_	REGULAR	4.1 KB	1 (since Sun Feb 11 2024)	UNIQUE
> email_1	REGULAR	4.1 KB	0 (since Sun Feb 11 2024)	UNIQUE

## Validaciones en los DTO

Renombramos el `create-auth.dto.ts` por `create-user.dto.ts` y definimos los campos que esperamos en el request.

Instalamos un par de clases para realizar validaciones en el DTO.

```
npm install class-validator class-transformer
```

Luego en nuestro `Main.ts`, agregamos el validator, como un `useGlobalPipes` con dos parametros

```
sync function bootstrap() {
 const app = await NestFactory.create(AppModule);
 app.useGlobalPipes(new ValidationPipe({
 whitelist: true,
 forbidNonWhitelisted: true,
 }));
 await app.listen(3000);
```

**whitelist:** significa que la instancia del objeto validado devuelto después de la validación solo contendrá propiedades que tengan decoradores de validación aplicados. En otras palabras, se eliminarán del objeto todas las propiedades que no estén asociadas a algún decorador de validación.

**forbidNonWhitelisted:** Cuando `forbidNonWhitelisted` se establece en `true`, la validación generará un error si se encuentran propiedades no permitidas en el objeto validado. En otras palabras, si `whitelist` está activado y `forbidNonWhitelisted` está configurado en `true`, la validación fallará si el objeto contiene propiedades que no tienen decoradores de validación y que, por lo tanto, no deberían estar presentes según la configuración de `whitelist`.

Agregamos las propiedades en nuestro DTO (`create-user.dto.ts`) y las reglas de validación

```

import { IsEmail, IsString, MinLength } from 'class-validator';

export class CreateUserDto {

 @IsEmail()
 email:string;

 @IsString()
 name:string;

 @MinLength(8)
 password:string;
}

```

## Crear registro en Mongo DB

---

Primero debemos inyectar, en nuestro servicio **AuthService** el modelo a usar, en este caso el USER

```
constructor(@InjectModel(User.name) private userModel: Model<User>) {}
```

Usamos el decorador **@InjectModel** y le pasamos dos parametros

**User.name** User ese el modelo de datos, y name devuelve el nombre de la clase, que generalmente es el nombre del modelo de Mongoose

Luego declaramos una variable privada **userModel** del tipo **Model** Model es una clase proporcionada por Mongoose que representa un modelo para una colección MongoDB específica. En este caso, es un modelo para la colección 'users' que se ha definido previamente.

Hacemos una prueba, sin validaciones adicionales ni encriptación de password, únicamente para probar la interacción con MongoDB

```

create(createAuthDto: CreateUserDto) : Promise<User> {
 const user = new this.userModel(createAuthDto);
 return user.save();
}

```

Al ejecutar este curl

```

curl --location 'http://localhost:3000/auth' \
--header 'Content-Type: application/json' \
--data-raw '{
 "email": "myemail@gmail.com",
 "name": "Juan el Terrible",
}

```

```

 "password": "A_NoS?!///"
}

```

Comprobamos en Mongo Compass:

The screenshot shows the Mongo Compass interface with a single document listed. The document contains the following fields and values:

- `_id: ObjectId('65cab296c942b5430e6f797a')`
- `email : "myemail@gmail.com"`
- `name : "Juan el Terrible"`
- `password : "A_NoS?!///"`
- `isActive : true`
- `role : Array (1)`
- `--v : 0`

## Encriptar Contraseña

Instalamos el paquete bcryptJS

```

npm install bcryptjs
npm i --save-dev @types/bcryptjs

```

Luego usamos `bcryptjs.hashSync` para generar el password encriptado.

```

async create(createAuthDto: CreateUserDto) : Promise<User> {
 const { password } = createAuthDto;
 createAuthDto.password = bcryptjs.hashSync(password, 10);

 const createdUser = new this.userModel(createAuthDto);

 await createdUser.save()

 // Remove password from the response
 const { password: pass, ...result } = createdUser.toJSON();
 return result;
}

```

Esta petición

```
{
 "email": "Felipe@gmail.com",
 "name": "Felipe Cruz",
}
```

```

 "password": "felipecruz"
}
```

Genera este registro

```
{
 "email": "Felipe@gmail.com",
 "name": "Felipe Cruz",
 "isActive": true,
 "role": [
 "user"
],
 "_id": "65cb8490f5f37a69a17f668a",
 "__v": 0
}
```

Como se puede observar, el método **create** ha eliminado el password de la respuesta. Pare ello fue necesario hacer opcional el campo password en la Entidad, porque el create retorna `Promise<User>`

## Login

---

Creamos el DTO especial para el Login

```

import { IsEmail, Min, MinLength } from "class-validator";

export class LoginDto {
 @IsEmail()
 email: string;

 @MinLength(8)
 password: string;
}
```

Agregamos un controlador

```

@Post('login')
login(@Body() LoginDto: LoginDto) {
 return this.authService.login(LoginDto);
}
```

Y agregamos el servicio

```

async login(loginDto: LoginDto) {
 const { email, password } = loginDto;
 /
}
```

```
const user = await this.userModel
 .findOne({ email });

 if (!user) {
 throw new UnauthorizedException(`User with email ${email} not
found`);
 }

 if (!bcryptjs.compareSync(password, user.password)) {
 throw new UnauthorizedException('Invalid password');
 }

 if (!user.isActive) {
 throw new UnauthorizedException('User is not active');
 }

 const { password: pass, ...result } = user.toJSON();

 return {
 message: 'User logged in successfully',
 user,
 "token": "_token_"
 }

}
```

## Token JWT

---

El estándar **RFC 7519** se refiere a JSON Web Token (JWT), que es un formato compacto y seguro para transmitir información entre dos partes. Es como un pequeño paquete de datos que se utiliza para enviar información de forma segura a través de la web.

Un **JWT** consiste en tres partes separadas por puntos: **el encabezado, la carga útil y la firma**. El **encabezado** proporciona información sobre cómo se debe procesar el token, la **carga útil** contiene los datos que se están transmitiendo y la **firma** se utiliza para verificar que el remitente del token es quien dice ser.

JWT se utiliza comúnmente en autenticación y autorización en aplicaciones web. Por ejemplo, después de que te autenticas en un sitio web, recibes un JWT que contiene información sobre tu sesión. Este token se adjunta a tus solicitudes posteriores para que el servidor pueda verificar tu identidad y autorizar tus acciones. Es una forma eficiente y segura de manejar la información entre diferentes partes de una aplicación web.

Instalamos las librerías:

```
npm install --save @nestjs/jwt
```

Necesitamos un Secret, lo creamos en nuestro .env.template y generamos un valor en nuestro .env,

```
// JWT Secret
// node -e "console.log(require('crypto').randomBytes(32).toString('hex'))"
JWT_SECRET=Add_strong_secret_here
```

Configuramos el módulo:

```
@Module({
 controllers: [AuthController],
 providers: [AuthService],
 imports: [
 ConfigModule.forRoot(),

 MongooseModule.forFeature([
 { name: 'User', schema: UserSchema }]),

 JwtModule.register({
 global: true,
 secret: process.env.JWT_SECRET,
 signOptions: { expiresIn: '6h' }
 })
]
})
export class AuthModule {}
```

Registraremos el **JwtModule.register()** y como necesitamos acceder al **process.env.JWT\_SECRET** es requerido importar la **ConfigModule.forRoot()**

Creamos una interfaz

```
export interface JwtPayload {
 id: string,
 created_at?: number,
 expires_at?: number
}
```

En el constructo del **AuthService** inyectamos el **JwtService**

```
private jwtService: JwtService
```

Creamos un método para generar el JWT

```
getJwtToken(payload: JwtPayload): string {
 const token = this.jwtService.sign(payload);
 return token;
}
```

Y ahora en lugar de regresar

```
"token": "_token_"
```

Regresamos el token real:

```
return {
 message: 'User logged in successfully',
 user,
 "token": this.getJwtToken({ id: user.id })
};
```

## Registro de un usuario

---

Reutilizaremos el Create, para implementar un nuevo endpoint, el de registrar un usuario:

Primero debemos crear una interfaz

```
export interface LoginResponse {
 user: User;
 token: string;
}
```

Esta interfaz la usaremos como respuesta tanto en el Login como en el register.

Luego crearemos un nuevo DTO,

```
export class RegisterUserDto {

 @IsEmail()
 email:string;

 @IsString()
 name:string;

 @MinLength(8)
 password:string;
}
```

Este es igual al CreateUserDto, pero lo manejamos en un DTO separado porque en un futuro podríamos usar el register como operación pública y el create como de solo acceso para administradores.

Luego implementamos el Register en el controller:

```
@Post('register')
register(@Body() registerUserDto: RegisterUserDto) {
 return this.authService.register(registerUserDto);
}
```

También implementamos el servicio:

```
async register(registerUserDto: RegisterUserDto): Promise<LoginResponse> {

 var user = await this.create(registerUserDto);

 return {
 user,
 token: this.getJwtToken({ id: user._id })
 };
}
```

NOTA: Debemos agregar la siguiente propiedad en User.Entity.ts para poder acceder al id generado por MongoDB

```
_id?: string;
```

## Barrel index.ts

podemos crear un archivo index.ts, en un directorio específico, e importar todas las clases de una vez, por ejemplo en el DTO

```
export * from './create-user.dto';
export * from './login.dto';
export * from './register-user.dto';
export * from './update-auth.dto';
```

Luego, en lugar de hacer estas importaciones:

```
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateAuthDto } from './dto/update-auth.dto';
```

```
import { LoginDto } from './dto/login.dto';
import { RegisterUserDto } from './dto/register-user.dto';
```

Lo cambiamos por:

```
import { CreateUserDto, LoginDto, RegisterUserDto, UpdateAuthDto } from
'./dto/';
```

Aplicamos el mismo cambio en el Servicio.

## Guards: Protección de Rutas

Vamos a proteger ciertas rutas, solamente los usuarios logeados deben de tener permiso para ejecutar ciertos endpoints.

Este endpoint, retorna toda la lista de usuario, aún sin estar logeado.

```
curl --location 'http://localhost:3000/auth'
```

Para proteger este endpoint debemos implementar un **guards** lo haremos con el comando siguiente

```
nest g gu auth/guards/auth
CREATE src/auth/guards/auth/auth.guard.spec.ts (160 bytes)
CREATE src/auth/guards/auth/auth.guard.ts (299 bytes)
```

Implementamos el Guard

```
@Injectable()
export class AuthGuard implements CanActivate {

 constructor(@InjectModel(User.name) private userModel: Model<User>,
 private jwtService: JwtService) {}

 async canActivate(
 context: ExecutionContext,
): Promise<boolean> {

 const request = context.switchToHttp().getRequest();

 const token = this.extractTokenFromHeader(request);

 if (!token) {
 throw new UnauthorizedException('Token not found');
 }
 }
}
```

```

 }

 try {
 const payload = await this.jwtService.verifyAsync<JwtPayload>(
 token,
 {
 secret: process.env.JWT_SECRET
 }
);

 var user = await this.authService.findUserById(payload.id);

 if (!user) {
 throw new UnauthorizedException('User not found');
 }

 if (!user.isActive) {
 throw new UnauthorizedException('User is not active');
 }

 request['user'] = user
 } catch (e) {
 throw new UnauthorizedException(e)
 }

 return true;
}

private extractTokenFromHeader(request: Request): string | undefined {
 const [type, token] = request.headers['authorization']?.split(' ') ??
[][];
 return type === 'Bearer' && token ? token : undefined;
}
}

```

Si hacemos una llamada, e imprimimos el request `console.log(request.headers);` obentemos:

```
{
 authorization: 'Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY1Y2I4NDkwZjVmMzdhnjlhMTdmNj
Y4YSIsImlhCI6MTcwODAwOTM1MCwiZXhwIjoxNzA4MDMwOTUwfQ.OCsxtEMGLQ_BcoHZ7He5A0
GDRTIpPs2h02Uu0aulkHE',
 'user-agent': 'PostmanRuntime/7.36.1',
 accept: '*/*',
 'postman-token': '0da21e00-e832-4393-8d03-12226280e590',
 host: 'localhost:3000',
 'accept-encoding': 'gzip, deflate, br',
 connection: 'keep-alive'
}
```

El método **extractTokenFromHeader** obtiene el token

Si el token existe lo validamos con **verifyAsync**, le debemos de pasar el SecretKey para validararlo.

Finalmente si el token es válido, guardamos el user en el request.

```
request['user'] = user;
```

### findUserById:

```
async findUserById(id: string): Promise<User> {
 const user = await this.userModel.findById(id);
 const { password, ...result } = user.toJSON();

 return result;
}
```

Hasta este punto hemos intercegado el request, se extrajo el Token, se validó y se guardo el User ID en la variable User del request.

Implementamos el Guard en el Controllador.

```
@UseGuards(AuthGuard)
@Get()
findAll(@Request() req: Request) {
 console.log(req['user']);
 return this.authService.findAll();
}
```

El console.log imprimirá los datos del usuario, sin el password:

```
{
 _id: new ObjectId('65cb8490f5f37a69a17f668a'),
 email: 'Felipe@gmail.com',
 name: 'Felipe Cruz',
 isActive: true,
 role: ['user'],
 __v: 0
}
```

# Nueva Sección: Nueva Aplicación: Auth App

## ¿Qué veremos en esta sección?

Esta sección principalmente busca conectar el Frontend (Angular) con el backend (Node+Nest) que hicimos en la sección anterior, pero puntualmente veremos:

- Functional Guards
- Manejo de autenticación
- Señales
- Effectos
- Manejo de errores
- SweetAlert
- Determinar estado de autenticación
- Manejo de JWTs
- Headers de petición Http

Vamos a crear la APP que consumirá nuestro backend:

```
ng new AuthApp --standalone false --routing
```

Ambas APP comparte el mismo directorio.

```
11-NestJS-Angular/
├── AuthApp -> Port 3001
└── nest-backend -> Port 3000
```

NOTA: Esta APP usa un style.css y unos assets descargados del curso.

Agreamos todos los elementos que vamos a necesitar

```
Modulos
$ ng g m auth --routing
$ ng g m dashboard --routing

Components
$ ng g c dashboard/layouts/dashboard-layout --skip-selector
$ ng g c auth/pages/loginPage --skip-selector
$ ng g c auth/pages/registerPage --skip-selector
$ ng g c auth/layouts/authLayout --skip-selector

Services
$ ng g s auth/services/auth
```

```
app
├── app.component.css
├── app.component.html
├── app.component.spec.ts
├── app.component.ts
├── app.module.ts
├── app-routing.module.ts
└── auth
 ├── auth.module.ts
 ├── auth-routing.module.ts
 ├── components
 ├── guards
 ├── interfaces
 └── layouts
 └── auth-layout
 ├── auth-layout.component.css
 ├── auth-layout.component.html
 └── auth-layout.component.ts
 └── pages
 ├── login-page
 │ ├── login-page.component.css
 │ ├── login-page.component.html
 │ └── login-page.component.ts
 └── register-page
 ├── register-page.component.css
 ├── register-page.component.html
 └── register-page.component.ts
 └── services
 └── auth.service.ts
└── dashboard
 ├── dashboard.module.ts
 ├── dashboard-routing.module.ts
 └── layouts
 └── dashboard-layout
 ├── dashboard-layout.component.css
 ├── dashboard-layout.component.html
 └── dashboard-layout.component.ts
 └── pages
```

## Sistema de Rutas

**AuthRoutingModule** va a contener:

```
const routes: Routes = [
{
 path: '',
 component: AuthLayoutComponent,
 children: [
 {
 /
```

```
 path: 'login',
 component: LoginPageComponent
 },
 {
 path: 'register',
 component: RegisterPageComponent
 },
 {
 path: '**',
 redirectTo: 'login',
 }
]
};

];
```

### El **DashboardRoutingModule**:

```
const routes: Routes = [
{
 path: '',
 component: DashboardLayoutComponent
}
];
```

### Y nuestro **AppRoutingModule**:

```
const routes: Routes = [
{
 path: 'auth',
 loadChildren: () => import('./auth/auth.module').then(m =>
m.AuthModule)
},
{
 path: 'dashboard',
 loadChildren: () => import('./dashboard/dashboard.module').then(m =>
m.DashboardModule)
},
{
 path: '**',
 redirectTo: 'auth'
}
];
```

Descargamos los diseños para **auth-layout.component.html**

```
<div class="limiter">
 <div class="container-login100" style="background-image:
url('.../.../.../assets/images/bg-01.jpg');">
 <div class="wrap-login100 p-l-55 p-r-55 p-t-65 p-b-54">

 <router-outlet />

 </div>
 </div>
</div>
```

Y también para \*\*login-page.component.html \*\*

```
<form class="login100-form"
 autocomplete="off">

 Login

 <div class="wrap-input100 m-b-23">
 Correo
 <input class="input100"
 type="email"
 placeholder="Correo electrónico">

 </div>

 <div class="wrap-input100">
 Contraseña
 <input class="input100"
 type="password"
 placeholder="Ingrese su contraseña">

 </div>

 <div class="text-right p-t-8 p-b-31"></div>

 <div class="container-login100-form-btn">
 <div class="wrap-login100-form-btn">
 <div class="login100-form-bgbtn"></div>
 <button class="login100-form-btn"
 type="submit">
 Login
 </button>
 </div>
 </div>

 <div class="flex-col-c p-t-60">

 ¿No tienes cuenta?

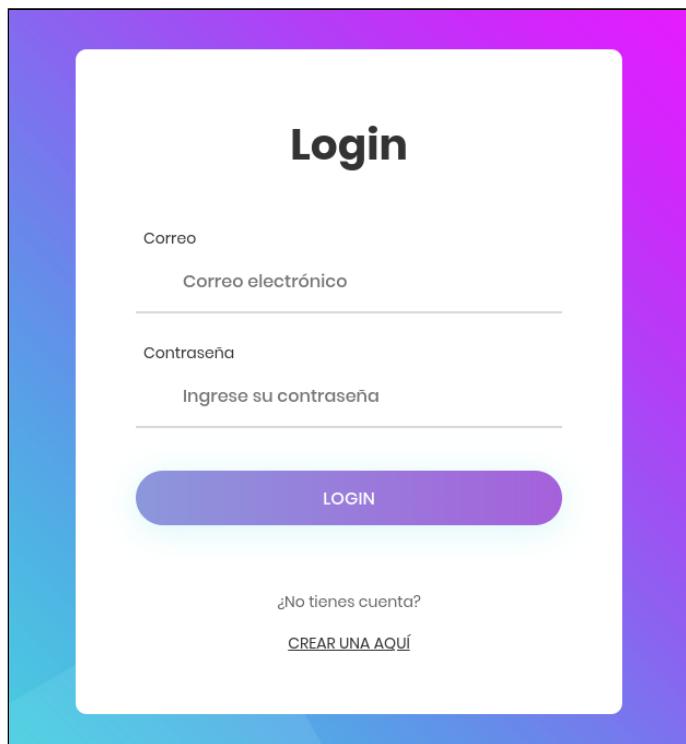
 </div>
```

```


 Crear una aquí

</div>
</form>
```

Resultado:



A continuación usaremos **ReactiveFormsModule** y dado que solo lo usaremos en nuestro AuthModule, lo vamos a importar en ese módulo.

```
NgModule({
 declarations: [
 // Declaraciones
],
 imports: [
 // Otros imports
 ReactiveFormsModule,
]
})
export class AuthModule { }
```

Luego enlazamos el formulario del HTML con el Form en nuestro componente

```
export class LoginPageComponent {

 private formBuilder = inject(FormBuilder);
```

```

public form: FormGroup = this.formBuilder.group({
 email: ['', [Validators.required, Validators.email]],
 password: ['', [Validators.required, Validators.minLength(6)]]
});

onSubmit(): void {
}
}

```

Y en el HTML aplicamos los cambios

```

<form [formGroup]="form" (ngSubmit)="onSubmit()">
<!-- Otros tags html-->
<input formControlName="email" type="email">
<input formControlName="password" type="password">

```

## Creando el Servicio

---

En el app Modulo importamos el **HttpClientModule**

Creamos dos interfaces

```

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

export interface LoginResponse {
 user: User;
 token: string;
}

```

```

export interface User {
 _id: string;
 email: string;
 name: string;
 password: string;
 isActive: boolean;
 role: string[];
}

```

Creamos un Enum

```

export enum AuthStatus {
 CHECKING = 'CHECKING',
}

```

```

 AUTHENTICATED = 'AUTHENTICATED',
 UNAUTHENTICATED = 'UNAUTHENTICATED',
};

}

```

Y luego implementamos el servicio

```

export class AuthService {

 private readonly baseUrl = environment.baseUrl;
 private httpClient: HttpClient = inject(HttpClient);
 private _currentUser = signal<User | null>(null);
 private _authStatus = signal<AuthStatus>(AuthStatus.CHECKING);
 // No exponemos el current User directamente, sino que lo exponemos a
 // través de una propiedad computada
 public currentUser = computed(() => this._currentUser);
 // No exponemos el authStatus directamente
 public authStatus = computed(() => this._authStatus);

 constructor() { }

 login(email: string, password: string): Observable<boolean> {

 const url = `${this.baseUrl}/auth/login`;
 const body = { email, password };

 return this.httpClient.post<LoginResponse>(url, body)
 .pipe(
 tap(({user, token}) => {
 this._currentUser.set(user);
 this._authStatus.set(AuthStatus.AUTHENTICATED);
 localStorage.setItem('token', token);
 console.log({user, token});
 }),
 map(() => true)
);
 }
}

```

Y cuando el usuario hace click en el botón login llamamos el onSubmit:

```

onSubmit() {
 console.log('Vino ACA');
 return this.authService.login(this.form.value.email,
 this.form.value.password)
 .subscribe((result) => {
 console.log('Result **', result);
 });
}

```

# CORS

Si intentamos logearnos, vamos a tener un error:

```
Access to XMLHttpRequest at 'http://localhost:3000/auth/login' from origin 'http://localhost:4200' has been blocked by CORS policy: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

Una solución válida para entornos de desarrollos consiste en configurar CORS en el API,

```
app.enableCors();
```

Pero en producción debemos ser más específicos e indicar quienes son los orígenes habilitados:

```
// Configura CORS en tu aplicación NestJS
const allowedOrigins = ['http://localhost:4200'];

app.use(
 cors({
 origin: (origin, callback) => {
 // Verifica si el origen de la solicitud está en la lista de dominios permitidos
 const isAllowed = allowedOrigins.includes(origin);
 callback(null, isAllowed);
 },
 methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
 credentials: true,
 }),
);
```

## Manejo de Errores

Con la implementación actual, no estamos manejando adecuadamente los errores, por ejemplo, al ingresar un password incorrecto, vemos un error en la consola, pero el usuario no se percata del error.

```
{
 "headers": {
 "normalizedNames": {},
 "lazyUpdate": null
 },
 "status": 401,
 "statusText": "Unauthorized",
 "url": "http://localhost:3000/auth/login",
 "ok": false,
```

```

 "name": "HttpErrorResponse",
 "message": "Http failure response for http://localhost:3000/auth/login:
401 Unauthorized",
 "error": {
 "message": "Invalid password",
 "error": "Unauthorized",
 "statusCode": 401
 }
}

```

Para manejar el error, Instalemos el sweetalert2 para mostrar el mensaje al usuario:

```
npm install sweetalert2
```

Luego agregaremos un catchError (funcion de rxJs) como última función en el pipe

```

catchError((err) => {
 return throwError(() => err.error.message);
})

```

Y en el componente, en el onSubmit

```

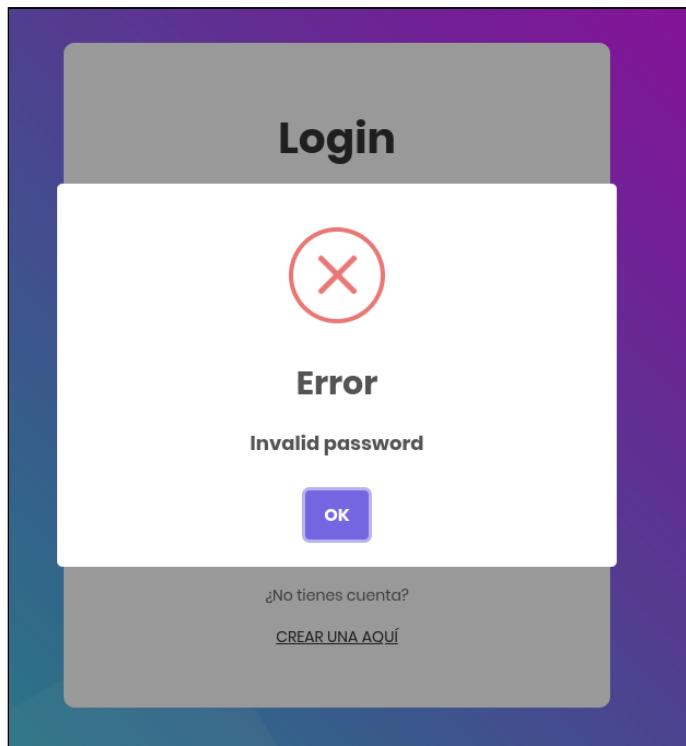
onSubmit() {
 return this.authService.login(this.form.value.email,
 this.form.value.password)
 .subscribe({
 next: () => console.log('Logged in'),
 error: (error) => {

 var errorMessage = '';

 if (error instanceof Array) {
 error.forEach((err) => {
 errorMessage += err + '
';
 });
 error = errorMessage;
 } else {
 errorMessage = error.message;
 }
 Swal.fire('Error', error, 'error')
 }
 });
}

```

Cualquier error será mostrado en un Alert:



## Funciones Guards, protección de rutas:

Primero haremos el redirect al Dashboard, cuando el usuario se ha logeado. En el **LoginPageComponent**注入amos el router y luego lo usamos en el **Next** del **subscribe** en el método **onSubmit**

```
private router = inject(Router);

return this.authService.login(this.form.value.email,
this.form.value.password)
.subscribe({
 next: () => this.router.navigate(['/dashboard']),
 error: // Manejo de error.
});
```

Una vez que el usuario se autentica, es redirigido correctamente al Dashboard.

Pero el Dashboard no tiene protección alguna, podríamos acceder al url sin estar Logeados. Esa protección la implementaremos con un Guard.

Otra situación es que el Usuario logeado no se mantiene en memoria. Si agregamos este cambio en el **DashboardLayoutComponent**

```
export class DashboardLayoutComponent {
 private authService = inject(AuthService);
 public user = computed(() => this.authService.currentUser());
}
```

y mostramos en el template

```
<pre>
 {{ user() | json }}
</pre>
```

Veremos que luego de logearnos y ser redirigidos, podremos ver la información de usuario en pantalla, pero si recargamos el dashboard se pierde la información.

Vamos a implementar el Guard primeramente:

```
$ ng g g auth/guards/is-authenticated --functional
? Which type of guard would you like to create? CanActivate
CREATE src/app/auth/guards/is-authenticated.guard.spec.ts (506 bytes)
CREATE src/app/auth/guards/is-authenticated.guard.ts (139 bytes)
```

Agreguemos el Guard

```
import { CanActivateFn, Router } from '@angular/router';
import { AuthService } from '../services/auth.service';
import { inject } from '@angular/core';

export const isAuthenticatedGuard: CanActivateFn = (route, state) => {
 const authService = inject(AuthService);
 const router = inject(Router);

 if (authService.authStatus() === 'AUTHENTICATED')
 return true;

 router.navigate(['/auth/login']);
 return false;
}
```

Este guard lo podemos usar en aquellas rutas que queremos proteger, por ejemplo el del dashboard. Para ello actualizamos nuestro sistema de rutas, en el **AppRoutingModule**

```
{
 path: 'dashboard',
 loadChildren: () => import('./dashboard/dashboard.module').then(m =>
m.DashboardModule),
 canActivate: [isAuthenticatedGuard]
},
```

Cómo funciona el Guard? Esta es la secuencia de eventos:

- El Usuario se logea

- El servicio tiene un Signal Computed public llamado authStatus, este es actualizado a **AUTHENTICATED** solo si el usuario ha proporcionado credenciales correctas
- El router redirige al usuario al **Dashboard**
- Cuando se carga el **Dashboard** el Guard entra en acción, y verifica que el usuario esté autenticado. Caso contrario es redirigido al Login.

## Check Token

Vamos a crear una interfaz

```
import { User } from "./user.interface";

export interface LoginResponse {
 user: User;
 token: string;
}
```

Y agregaremos un nuevo método en el servicio AuthService para validar el estado del token,

```
checkAuth(): Observable<boolean> {
 const url = `${this.baseUrl}/auth/check-token`;
 const token = localStorage.getItem('token');

 const header = new HttpHeaders().set('Authorization', `Bearer ${token}`);

 return this.httpClient.get<CheckTokenReponse>(url, { headers: header })
 .pipe(
 map(({token, user}) => {
 this._currentUser.set(user);
 this._authStatus.set(AuthStatus.AUTHENTICATED);
 localStorage.setItem('token', token);
 return true;
 }),
 catchError((err) => {
 if (401 === err.status) {
 this._currentUser.set(null);
 this._authStatus.set(AuthStatus.UNAUTHENTICATED);
 localStorage.removeItem('token');
 this.router.navigate(['/auth/login']);
 }
 return of(false);
 })
);
}
```

Si durante la validación del token, esté sigue siendo válido, usaremos el nuevo token en nuestra app. Si por el contrario es un token expirado, limpiaremos las variables necesarios y vamos a redireccionar al usuario al Login.

## Reaccionando a los cambios de Autenticación

Vamos a implementar un fujo que valide el estado de la autenticacion, primero agregamos esto en el

```
export class AppComponent {
 private authService = inject(AuthService);
 private router = inject(Router);

 public checkAuthStatus = computed(() => {
 if (this.authService.authStatus() === AuthStatus.CHECKING) {
 return false;
 }
 return true;
 });

 public authStatusChangeEffect = effect(() => {
 this.authService.authStatus();
 });
}
```

En el template principal **app.component.html**

```
<h1 *ngIf="!checkAuthStatus()">Loading...</h1>
<router-outlet *ngIf="checkAuthStatus()"></router-outlet>
```

Y en el constructor del servicio de autenticacion:

```
constructor() {
 this.checkAuth().subscribe();
}
```

- Cuando se crea una instancia del servicio (constructor), se inicia automáticamente el proceso de verificación de autenticación (**checkAuth()**).
- Mientras se verifica la autenticación, el componente principal (AppComponent) puede mostrar un mensaje de carga

```
<h1 *ngIf="!checkAuthStatus()">Loading...</h1>
```

- Cuando la verificación de autenticación se completa (ya sea éxito o error), se emite un valor a través del observable al que se suscribió en el constructor (`this.checkAuth().subscribe()`).
- Dependiendo del resultado de la verificación, el componente principal decide mostrar el contenido principal o redirigir al componente de inicio de sesión (

```
<router-outlet *ngIf="checkAuthStatus()"></router-outlet>
```

## Logout

El logout consiste en limpiar el token y la información del usuario que manejamos en memoria.

Creamos un método en el Servicio

```
logout() {
 this._currentUser.set(null);
 this._authStatus.set(AuthStatus.UNAUTHENTICATED);
 localStorage.removeItem('token');
}
```

Y simplemente lo mandamos a llamar desde el componente

# Nueva Sección - Deploy en Producción

¿Qué veremos en esta sección?

El objetivo de esta sección es desplegar a producción nuestra base de datos, backend y frontend para poder ser consumidos en cualquier parte del mundo.

Puntualmente veremos:

- Aprovisionamiento de base de datos MongoDB
- Railway y MongoAltas
- Variables de entorno para producción
- Hash Strategy
- Publicar aplicación de Node (Nest)
- Publicar aplicación de Angular
- Seleccionar bases de datos
- Pruebas reales.

## Aprovisionamiento MongoDB

Realizaremos el aprovisionamiento de la base de datos en Railway, pero hay varios servicios que les permiten preparar la base de datos Mongo de formas gratuita.

<https://railway.app/>

Abrimos una cuenta en Railway y creamos un nuevo proyecto, **Provisional Mongo DB**.

Creamos una base de datos nueva **auth-db** con una colección llamada **demo**

Luego copiamos la cadena de conexión, ubicada en la pestaña **Variables**, copiar el contenido de **MONGO\_URL**

Abrimos localmente **mongoDB Compass** y agregamos una nueva conexión.

## Configuración del Backend

En el archivo .env de nuestro backend vamos a comentar la línea:

```
MONGO_URI=mongodb://localhost:27017/mean-db
```

Y vamos a agregar la cadena de conexión de Railway.

**RailWay** crea una base de datos **test** por default, y si no se especifica el nombre de la base de datos que nuestro backend va a utilizar, todos los objetos van a ser creado en **test**, por lo tanto debemos agregar una nueva variable de Environment en nuestro .env y .env.template

```
// Used for railWay
MONGO_DB_NAME=auth-db
```

Luego debemos especificar este nombre de base de datos en nuestro backend service, específicamente en el **AppModule**

```
Module({
 imports: [
 ConfigModule.forRoot(),
 MongooseModule.forRoot(process.env.MONGO_URI,
 process.env.MONGO_DB_NAME ?
 {} : {}),
 AuthModule,],
 controllers: [],
 providers: [],
)
export class AppModule {
}
```

Si MONGO\_DB\_NAME está definido, lo usamos, caso contrario no se configura el dbName.

## Configuración del Puertos

Cuando se despliegan servicios en la nuber, el puerto donde corre la APP es asignado por el proveedor del servicio, por lo tanto debmos preparar nuestra APP para que tome dicho puerto, esto lo hacemos con este cambio

```
await app.listen(process.env.PORT || 3000);
```

Si el entorno de despliegue ha definido un puerto, lo hará por medio de la variable de entorno **process.env.PORT**, si no existe usamos el puerto 3000

Localmente, si no definimos PORT simplemente seguirá usando el puerto 3000.

## Desplegar Backend en RailWay

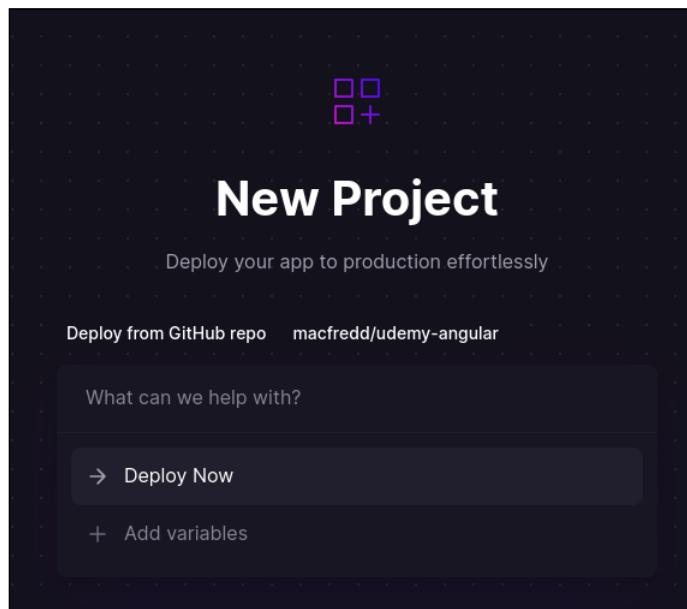
En el Package.Json vamos a cambiar el script start por estos comandos:

```
"start": "node dist/main",
```

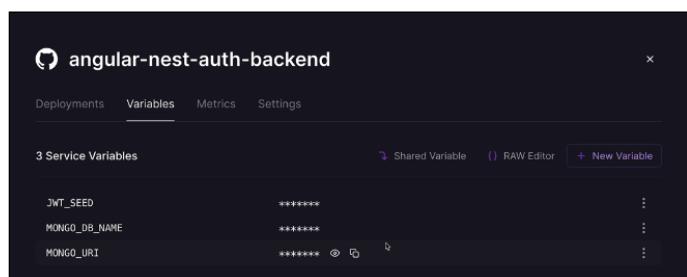
Subimos ese cambio al repositorio y abrimos Railway.

Agregamos un nuevo proyecto, y seleccionamos **Github Repo** y seleccionamos el repositorio de nuestro backend.

Conectamos nuestro repositorio y antes de hacer el deploy debemos de agregar las variables de entorno que nuestro backend necesita



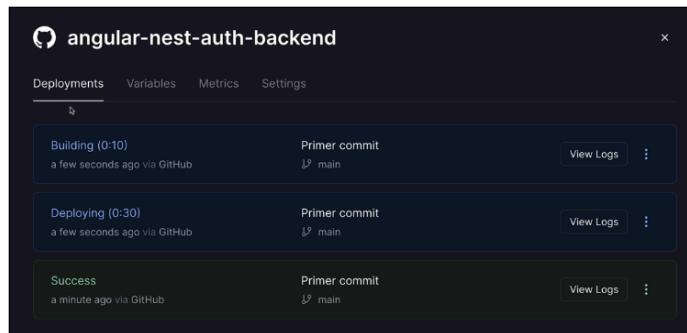
Las variables de entorno son las mismas que hemos definido en nuestro .env, apuntando a la base de datos en la nube



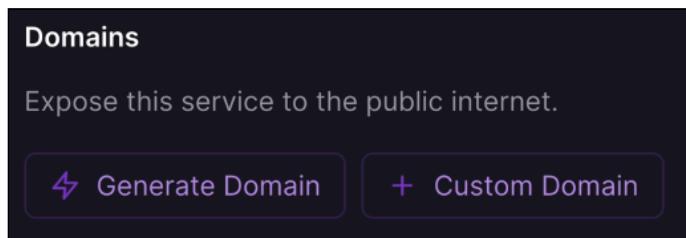
NOTA: Debemos generar un nuevo Secret para JWT.

El repositorio únicamente debe de contender el proyecto a deployar.

Luego en la pestaña de deployment, veremos como se completa el proceso:



Una vez finalizado el deployment, vamos a Settings y buscamos la configuración del Dominio. Por default, el servicio no esta disponible, es un servicio interno.



Generamos un Dominio. y luego copiamos dicho domino y lo configuraagmos en nuestras app's

El URL debe contener el https://

## Desplegar FrontEnd en RailWay

Usaremos Netlify para desplegar nuestro frontEnt

Primero debemos cambiar el URL de nuestro API en el .env del proyecto de Angular

Luego debemos agregar la opción `useHash: true` en el **AppRoutingModule**

```
@NgModule({
 imports: [RouterModule.forRoot(routes, {useHash: true})],
 exports: [RouterModule]
})
export class AppRoutingModule {}
```

Posteriormente hacemos el build

```
ng build
```

Y finalmente, luego de abrir una cuenta en Netlify, subimos la carpeta build de nuestro proyecto.

Se han eliinado los servicios creados en la nube. Solamente se usaron con el propósito de seguir el curso.

# Nueva Sección: Novedades de Angular 17

## ¿Qué veremos en esta sección?

Esta sección está dedicada a explorar nuevas funcionalidades en Angular, puntualmente quiero que veamos sobre:

- Señales
  - Observables a señales
  - Señales para estados en servicios
  - Señales de solo lectura (asReadOnly)
  - Señales computadas
- Defer
  - Funcionalidad
  - Triggers
- Control Flow
  - @if
  - @for
  - @else
  - @empty
  - @switch
  - @case
  - @default
- View Transition API (Aún no compatible con todos los navegadores)
- Nuevo sistema de detección de cambios
- Tailwind con Angular
- Configuración de Path alias en TypeScript

## Nueva Aplicación

Creamos una aplicación:

```
ng new 12-NewFeatures
```

Se ha creado la nueva aplicación usando el estandar sin Modulos, es decir usaremos el standalone true por defecto

Esto introduce algunos cambios importantes, primero es que no tenemos el **AppModule** ya que la aplicación en sí no se construye como un módulo.

Otro aspecto importante es que ahora tenemos un archivo **appConfig**

```

import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
 providers: [provideRouter(routes)]
};

```

Anteriormente nuestro iniciador o bootstrap en el **main.ts** únicamente incluía el AppModule, tal como se indica a continuación:

```

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

```

Ahora lo que pasamos como parámetro es el **AppComponent** standalone y el **appConfig**

```

bootstrapApplication(AppComponent, appConfig)
 .catch(err) => console.error(err));

```

## Tailwind en Angular

Instalamos Tailwind para angular

```

npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init

```

Este último comando crea el archivo **tailwind.config.ts** en el cual debemos actualizar el **content** para indicar las rutas de nuestros archivos html y ts.

```

/** @type {import('tailwindcss').Config} */
module.exports = {
 content: [
 "./src/**/*.{html,ts}",
],
 theme: {
 extend: {},
 },
 plugins: [],
}

```

Agregamos estas directivas a nuestro archivo **src/style.css**

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Si el editor muestra un error, Primero debemos instalar una extensión en VCode **Tailwind CSS IntelliSense** y luego debemos asociar el archivo CSS con Tailwind CSS. Abrimos el archivo CSS donde importas Tailwind CSS, presiona:

```
Ctrl + Shift + P
```

y busca **Change Language Mode**. Dentro de la barra de búsqueda, escribe tailwindcss y selecciónalo

## Creando elementos de la APP

Creamos un componente

```
$ ng g c dashboard/dashboard --skip-selector --skip-tests --flat
$ ng g c dashboard/pages/change-detection --skip-selector --skip-tests
$ ng g c dashboard/pages/control-flow --skip-selector --skip-tests
$ ng g c dashboard/pages/deferred-options --skip-selector --skip-tests
$ ng g c dashboard/pages/deferred-views --skip-selector --skip-tests
$ ng g c dashboard/pages/user --skip-selector --skip-tests
$ ng g c dashboard/pages/users --skip-selector --skip-tests
$ ng g c dashboard/pages/view-transition --skip-selector --skip-tests
$ ng g c shared/heavy-loaders-slow --skip-tests --inline-style
```

Renombramos la carpeta **heavy-loaders-slow** por **heavy-loaders** y agregamos otro componente:

```
$ ng g c shared/heavy-loaders/heavy-loaders-fast --skip-tests --inline-style
$ ng g c shared/heavy-loaders/users-loader --skip-tests --inline-style --flat
$ ng g c shared/menu/side-menu --skip-tests --flat
$ ng g c shared/titles --skip-tests
```

Creamos carpetas para las interfaces, servicios, etc.

La estructura de directorios en este momento debe verse de la siguiente forma

He borrado los CSS y HTML para simplificar la estructura de directorios y archivos

```
src/app/
├── app.component.css
├── app.component.html
├── app.component.spec.ts
├── app.component.ts
├── app.config.ts
├── app.routes.ts
└── dashboard
 ├── dashboard.component.ts
 └── pages
 ├── change-detection
 │ └── change-detection.component.ts
 ├── control-flow
 │ └── control-flow.component.ts
 ├── deferred-options
 │ └── deferred-options.component.ts
 ├── deferred-views
 │ └── deferred-views.component.ts
 ├── user
 │ └── user.component.ts
 ├── users
 │ └── users.component.ts
 └── view-transition
 └── view-transition.component.ts
└── interfaces
└── services
└── shared
 ├── heavy-loaders
 │ ├── heavy-loaders-fast.component.ts
 │ ├── heavy-loaders-slow.component.ts
 │ └── users-loader.component.ts
 ├── menu
 │ └── side-menu.component.ts
 └── titles
 └── titles.component.ts
```

## Agregando las Rutas

Podemos agregar la ruta al dashboard de dos formas:

```
{
 path: 'dashboard',
 loadComponent: () => import('./dashboard/dashboard.component').then(c =>
c.DashboardComponent)
}
```

Si queremos cargar directamente el componente, podemos hacerlo de esta forma

```
{
 path: 'dashboard',
 loadComponent: () => import('./dashboard/dashboard.component')
}
```

Para que acepte esta segunda forma, en el export debemos especificar que sea el **default class**.

```
export default class DashboardComponent {
}
```

Completamos todas las rutas

```
export const routes: Routes = [
 {
 path: 'dashboard',
 loadComponent: () =>
 import('./dashboard/dashboard.component')
 .then(c => c.DashboardComponent),
 children: [
 {
 path: 'change-detection',
 title: 'Change Detection',
 loadComponent: () =>
 import('./dashboard/pages/change-detection/change-detection.component')
 .then(c => c.ChangeDetectionComponent)
 },
 {
 path: 'control-flow',
 title: 'Control Flow',
 loadComponent: () =>
 import('./dashboard/pages/control-flow/control-flow.component')
 .then(c => c.ControlFlowComponent)
 },
 {
 path: 'deferred-options',
 title: 'Deferred Options',
 loadComponent: () =>
 import('./dashboard/pages/deferred-options/deferred-options.component')
 .then(c => c.DeferredOptionsComponent)
 },
 {
 path: 'deferred-views',
 title: 'Deferred Views',
 loadComponent: () =>
 import('./dashboard/pages/deferred-views/deferred-views.component')
 .then(c => c.DeferredViewsComponent)
 },
]
 }]
```

```
},
{
 path: 'user/:id',
 title: 'User',
 loadComponent: () =>
 import('./dashboard/pages/user/user.component')
 .then(c => c.UserComponent)
},
{
 path: 'user-list',
 title: 'Users List',
 loadComponent: () =>
 import('./dashboard/pages/users/users.component')
 .then(c => c.UsersComponent)
},
{
 path: 'view-transition',
 title: 'View Transition',
 loadComponent: () =>
 import('./dashboard/pages/view-transition/view-
transition.component')
 .then(c => c.ViewTransitionComponent)
},
{
 path: '',
 redirectTo: 'change-detection',
 pathMatch: 'full'
}
]
},
{
 path: '',
 redirectTo: '/dashboard',
 pathMatch: 'full'
},
{
 path: '**',
 redirectTo: '/dashboard',
}
];
```

Agregamos **router-outlet** al dashboard.component.html

```
<router-outlet></router-outlet>
```

## Dashboard Tailwind

Vamos a usar un template para nuestro dashboard, obtenemos el código de aca:

<https://tailwindcomponents.com/component/dashboard-navigation>

Eliminaremos los imports que aparecen en el inicio del HTML y solo usaremos el siguiente código para el Dashboard template:

```
<div class="flex bg-slate-100 overflow-y-scroll w-screen h-screen antialiased text-black selection:bg-blue-600 selection:text-white">
 <div class="flex relative w-screen">
 <app-side-menu></app-side-menu>

 <div class="text-black px-2 mt-2 w-full">
 <router-outlet></router-outlet>
 </div>

 </div>
</div>
```

Y vamos a extraer el HTML del `<div id="menu">` y lo vamos a copiar en nuestro **sideMenuComponent**

Se aplicaron un par de cambios menores para mostrar el Componente hijo en el área del dashboard.

## Menu Items

Vamos a extraer de las rutas las opciones para crear un menú en nuestro dashboard.

```
export class SideMenuComponent {

 public menuItems = [
 routes
 .map(route => route.children ?? [])
 .flat()
 .filter(route => route && route.path !== '/')
 .filter(route => !route.path?.includes(':'))
][0];

}
```

Usaremos este arreglo para crear nuestras opciones del Menú

## Control Flow: FOR

Vamos a usar un ciclo **@For** en lugar del **\*NgFor** que usábamos para recorrer los ítems del menú, tal como se ve en la plantilla, el nuevo FOR es más fácil de leer y tiene una sintaxis igual a la que se usa en la programación con estos bloques de código.

```
<div
 id="menu"
 class="bg-gray-900 min-h-screen z-10 text-slate-300 w-64 left-0 h-screen overflow-y-scroll">
 /
```

```

<div id="logo" class="my-4 px-6">
 <h1 class="text-lg md:text-2xl font-bold text-white">
 Dash8.</h1>
 <p class="text-slate-500 text-sm">Manage your actions and
activities</p>
</div>
<div id="nav" class="w-full px-6">

 @for (item of menuItems; track $index) {
 <a [routerLink]="item.path" routerLinkActive="bg-blue-800"
 href="#" class="w-full px-2 inline-flex space-x-2 items-center
border-b border-slate-700 py-3 hover:bg-white/5 transition ease-linear
duration-150">
 <div class="flex flex-col">
 {{
item.title}}

 </div>

 }

 </div>
</div>

```

## Configurando Alias

Este es un tema aparte, muchas veces tenemos que configurar rutas en los imports y lo hacemos con path relativos, por ejemplo

```
import { someting } from '../../../../../somePath'
```

Si pudieramos agregar un alias, por ejemplo `@myBasePath` y cambiar eso por esta ruta, seía mucho mejor:

```
import { someting } from '@myBasePath/somePath'
```

Para configurar estos alias, agregamos los paths en el archivo **tsconfig.json**, por ejemplo para agregar la ruta al **shared**:

```
{
 "compileOnSave": false,
 "compilerOptions": {
 "paths": {
 "@shared/*": ["./src/app/shared/*"]
 }
 }
}
```

```
 }
}
```

Con este tipo de cambios podremos reemplazar la primer línea por la segunda:

```
import { SideMenuComponent } from './shared/menu/side-menu.component';
import { SideMenuComponent } from '@shared/menu/side-menu.component';
```

## Control Flow

Estas son estructuras de control, para mas información:

[https://angular.io/guide/control\\_flow](https://angular.io/guide/control_flow)

### Novedades en el @Input

#### Required

Hace que el valor input sea requerido, es decir se necesita enviar un valor desde la implementación del componente.

```
@Input({ required: true}) title!: string
```

#### Transform

Transoforma el Valor del Atributo

```
@Input({ transform: booleanAttribute }) flag: boolean = false;
```

Esto permique que si envío el flag lo colocara en true, caso contrario en falso

```
<app-some-component flag>
```

Eso enviaría TRUE al componente

```
<app-some-component>
```

Y eso enviaría FALSE al componente

## Change Detection on Push

El sistema de detección de cambios (Change Detection) es el proceso por el cual Angular identifica y aplica los cambios en el modelo y actualiza la vista en consecuencia. La estrategia "OnPush" es una de las estrategias de detección de cambios en Angular.

Cuando se utiliza la estrategia "OnPush" en un componente, este solo se vuelve a renderizar cuando se detectan cambios en las referencias de entrada (@Input) o cuando se emite un evento (@Output). Esto es diferente de la estrategia predeterminada, que vuelve a renderizar un componente cada vez que se realiza cualquier cambio en cualquier parte del árbol de componentes.

La estrategia "OnPush" puede ayudar a mejorar el rendimiento de la aplicación al reducir la cantidad de ciclos de detección de cambios, ya que los componentes solo se volverán a renderizar cuando sea necesario.

Agreguemos esto a nuestro componente

```
@Component({
 standalone: true,
 imports: [TitlesComponent, CommonModule],
 templateUrl: './change-detection.component.html',
 styleUrls: ['./change-detection.component.css']
})
export class ChangeDetectionComponent {

 public frameworkAsSignal = signal({
 name: 'Angular',
 version: '12.0.0'
 });

 public frameworkAsProperty = {
 name: 'Angular',
 version: '12.0.0'
 };

 constructor() {
 setTimeout(() => {
 this.frameworkAsProperty.name = 'React';
 console.log('ChangeDetectionComponent: setTimeout');
 }, 3000);
 }
}
```

Template:

```
<app-titles title="Change Detection on Push!"></app-titles>

<div class="container">
 <div class="row">
 <div class="col-12">
 frameworkAsSignal
 <pre>{{ frameworkAsSignal() | json }}</pre>
 /
```

```
frameworkAsProperty
<pre>{{ frameworkAsProperty | json }}</pre>
</div>
</div>
```

Cuando recargamos la página, ambos objetos son iguales:

```
frameworkAsSignal
{
 "name": "Angular",
 "version": "12.0.0"
}
frameworkAsProperty
{
 "name": "Angular",
 "version": "12.0.0"
}
```

Transcurridos 3 segundos, los objetos cambian:

```
frameworkAsSignal
{
 "name": "Angular",
 "version": "12.0.0"
}
frameworkAsProperty
{
 "name": "React",
 "version": "12.0.0"
}
```

Esta es la forma por default de trabajar del **changeDetection** renderiza todo cuando existen cambios en objetos referenciados desde la vista.

El modo **Default** no se especifica en el componente, pero es el modo que se usa en Angular.

```
changeDetection: ChangeDetectionStrategy.Default,
```

Existe otra estrategia **onPush**

```
@Component({
 standalone: true,
 imports: [TitlesComponent, CommonModule],
 changeDetection: ChangeDetectionStrategy.OnPush,
```

```
 templateUrl: './change-detection.component.html',
 styleUrls: ['./change-detection.component.css'
})
```

En este caso hemos cambiado la estrategia a **ChangeDetectionStrategy.OnPush** el resultado es que al cargar los elementos siguen siendo iguales, incluso al pasar los 3 segundos siguen siendo iguales, es decir la vista no se renderiza.

```
frameworkAsSignal
{
 "name": "Angular",
 "version": "12.0.0"
}
frameworkAsProperty
{
 "name": "Angular",
 "version": "12.0.0"
}
```

Porque el OnPush no detecta el cambio del `this.frameworkAsProperty.name = 'React'`; y lo renderiza?

En este código, estás modificando la propiedad name del objeto frameworkAsProperty, pero la referencia del objeto en sí no ha cambiado. Por lo tanto, la estrategia **OnPush** no detectará este cambio como una razón para volver a renderizar el componente.