

ANGULAR

martes, 4 de noviembre de 2025 12:58

Angular es el Framework SPA de Google.

Es igual que el resto de Frameworks, tiene detalles distintos y características iguales.
No tiene state y la organización de los componentes es diferente, ya que se separan los ficheros
Según la lógica.

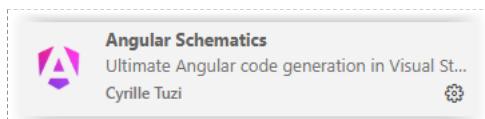
Utiliza el lenguaje **TypeScript** (opcional en algunos elementos) y la diferencia respecto a lo
Que estamos haciendo es el tipado de variables/objetos.

Tenemos dos formas de trabajar en Angular, igual que el resto de Frameworks o con una
Característica nueva única de Angular, por ahora, trabajamos igual que el resto.

Comenzamos instalando angular en nuestro equipo

npm install -g @angular/cli@latest

En VS code añadimos la siguiente extensión.



Comandos para nuestros proyectos:

1. Creación proyectos: **ng new**
2. Lanzar el server: **ng serve**

Vamos a crear nuestro primer proyecto, indicando que queremos utilizar un "jefe"

ng new primerangular --standalone=false --routing=false

```
C:\Users\Profesor MCSD Mañana\Documents\FRONT\ANGULAR
λ ng new primerangular --standalone=false --routing=false
```

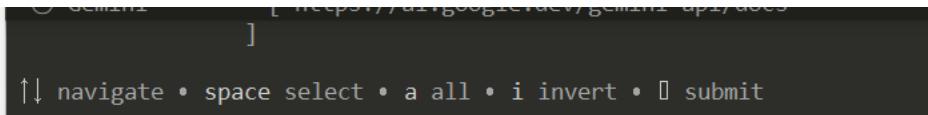
```
Cmder
Effective status: disabled
? Which stylesheet format would you like to use?
  CSS      [ https://developer.mozilla.org/docs/Web/CSS
  ]
  Sass (SCSS) [ https://sass-lang.com/documentation/syntax#scss
  ]
  Sass (Indented) [
https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less      [ http://lesscss.org

↑↓ navigate • ⌂ select
```

Le indicaremos que NO queremos cosas raras.

```
λ ng new primerangular --standalone=false --routing=false
✓ Which stylesheet format would you like to use? CSS
https://developer.mozilla.org/docs/Web/CSS
✓ Do you want to enable Server-Side Rendering (SSR) and Static Site
Generation (SSG/Prerendering)? No
? Do you want to create a 'zoneless' application without zone.js?
(y/N)n
```

```
? Which AI tools do you want to configure with Angular best
practices? https://angular.dev/ai/develop-with-ai
  None
  Claude [ https://docs.anthropic.com/en/docs/clause-code/memory
  ]
  Cursor [ https://docs.cursor.com/en/context/rules
  ]
  Gemini [ https://ai.google.dev/gemini-api/docs
```

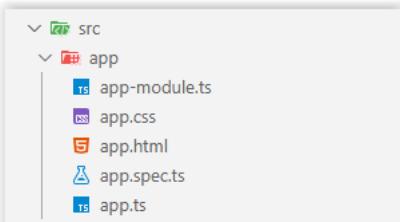


La base de cualquier SPA son los components y en Angular igual.
Un component lleva la extensión **TS**

A diferencia de otros Frameworks, se separan los ficheros en tres elementos:

1. **TS**: El código lógico de los scripts
2. **HTML**: El código de diseño de nuestro component
3. **CSS**: El diseño de nuestros components

Como son tres ficheros, se organiza todo en carpetas, es decir, cada component estará en una Carpeta organizado.



Si abrimos **app.ts** veremos el component con código lógico:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.html',
  standalone: false,
  styleUrls: ['./app.css'
})
export class App {
  protected readonly title = signal('primerangular');
}
```

Tenemos tres características importantes dentro del component:

1. **selector**: Es el nombre del selector que utilizaremos al dibujarlo en otros components.
2. **templateUrl**: El código HTML asociado a este component
3. **styleUrl**: El fichero CSS asociado a este component

Elementos básicos de Angular: **MODULE**. **app-module.ts**

Es el jefe de Angular. Es la clase dónde debemos declarar todo lo que vayamos a utilizar dentro De nuestro proyecto: Rutas o formularios.

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

import { App } from './app';

@NgModule({
  declarations: [
    App
  ],
  imports: [
    BrowserModule
  ],
  providers: [
    provideBrowserGlobalErrorListeners()
  ]
})
```

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

Nosotros vamos a trabajar igual que en otros Frameworks, tendremos una carpeta llamada **components** dentro de **src**

A su vez, cada component tendrá una carpeta para sus ficheros.

Cada component podrá contener objetos/variables que se declaran en la zona de **export class**

Comenzamos creando una carpeta llamada **components** y una carpeta llamada **primercomponent**

Por ahora, iremos creando todo manualmente hasta entender cómo funciona.

Creamos un nuevo component llamado **primer.component.ts**

PRIMERCOMPONENT.TS

```
import { Component } from '@angular/core';
//UN COMPONENT DEBE TENER SIEMPRE LA DECLARACION DE SU CONTENIDO
@Component ({
  //DEBEMOS DECLARAR EL NOMBRE DEL COMPONENT
  //MEDIANTE SU SELECTOR EN html
  //EN ANGULAR, LOS SELECTORES LLEVAN GUION
  selector: "primer-component",
  standalone: false,
  //POR AHORA, NO VOY A TENER HTML SEPARADO, PODEMOS INCLUIRLO
  //DENTRO DEL PROPIO COMPONENT
  template: `
    <h1>Soy el primer component de Angular!!!</h1>
  `
})
//CADA COMPONENT SIEMPRE DEBE TENER UNA CLASE ASOCIADA EN SU TS
//Dicho NOMBRE DE CLASE SI LLEVA MAYUSULAS Y SE DECLARA DENTRO DE app-
module.ts
export class PrimerComponent {
  //AQUI ES DONDE SE DECLARAN LAS VARIABLES
  //DICHAS VARIABLES DEBEN TENER UN TIPADO SIEMPRE (TypeScript)
  public titulo: string;
  public descripcion: string;
  public year: number;
  //EN ANGULAR, AL IGUAL QUE EN REACT, TENEMOS UN CONSTRUCTOR.
  //EN DICHO CONSTRUCTOR SERA DONDE INICIALIZAREMOS/INSTANCIAREMOS
  //LOS ELEMENTOS DE MI CLASE
  constructor() {
    //PARA ACCEDER A LOS OBJETOS DE LA CLASE, UTILIZAMOS LA PALABRA this
    this.titulo = "Hoy es martes";
    this.descripcion = "Hoy gana el Madrid!!!!";
    this.year = 2025;
  }
}
```

Para poder utilizar nuestro component en nuestra App, debemos "darlo de alta"/declararlo
Dentro de **MODULE**

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

import { App } from './app';
import { PrimerComponent } from '../primercomponent/primer.component';

@NgModule({
  declarations: [
    App,
    PrimerComponent
  ],
  providers: [
    provideBrowserGlobalErrorListeners()
  ]
})
```

Para dibujarlo, lo haremos dentro de **app.html**

APP.HTML

```
87      <div class="left-side">
88        
89      </div>
```

```

04      <defs>
31      |   </defs>
32      |   </svg>
33      |   <h1>Hello, {{ title() }}</h1>
34      |   <p>Congratulations! Your app is running. 🎉</p>
35      |   <primer-component></primer-component>
36      |   </div>
37      <div class="divider" role="separator" aria-label="Divider'>

```

Por supuesto, podemos utilizar las variables dentro del Template.

Para dibujar variables se utiliza la **doble llave**

```

template: `

<h1>Soy el primer component de Angular!!!</h1>
<h3 style="color:blue">{{ titulo }}</h3>
<h2 style="color:fuchsia">{{ descripcion }}, {{ year }}</h2>
`
```

Y veremos el resultado



Hello, primerangular

Congratulations! Your app is running. 🎉

Soy el primer component de Angular!!!

Hoy es martes

Hoy gana el Madrid!!!!, 2025

Vamos a crear un template externo, es decir, separar código lógico de dibujo HTML.

Sobre **primercomponent**, creamos un nuevo fichero llamado **primer.component.html**

```

<div>
  <h1>Soy el primer component de Angular!!!</h1>
  <h3 style="color: blue">{{ titulo }}</h3>
  <h2 style="color: fuchsia">{{ descripcion }}, {{ year }}</h2>
</div>

```

Debemos modificar el código del **@Component** y sustituir **template** por **templateUrl**

```

@Component ({
  //DEBEMOS DECLARAR EL NOMBRE DEL COMPONENT
})

```

```
//MEDIANTE SU SELECTOR EN html
//EN ANGULAR, LOS SELECTORES LLEVAN GUION
selector: "primer-component",
standalone: false,
//POR AHORA, NO VOY A TENER HTML SEPARADO, PODEMOS INCLUIRLO DENTRO DEL PROPIO COMPONENT
templateUrl: "./primer.component.html"
})
```

Por supuesto, también podemos incluir CSS externo.

Sobre **primercomponent**, creamos un nuevo css llamado **primer.component.css**

PRIMER.COMPONENT.CSS

```
h1 {
    background-color: lightgreen;
}

h2{
    background-color: lightskyblue;
}
```

Debemos incluir un key llamado **styleUrls**, que permite incluir más de un CSS dentro de nuestro Component

```
//DENTRO DEL PROPIO COMPONENT
templateUrl: "./primer.component.html",
styleUrls: ["./primer.component.css"]
})
```

HOOKS EN ANGULAR

Un Hook son los ciclos de vida dentro de los Frameworks SPA, es decir, lo que conocemos como **componentDidMount()** de React o **mounted()** de vue

Nos permiten saber los "**cómo**" de nuestros componentes:

- **ngOnInit**: Es el primer método que se ejecuta después del **constructor**.
Para utilizarlo, debemos implementar **OnInit** dentro de nuestro Component
El constructor se utilizará para instanciar nuestras variables
Este método se utiliza para trabajar con dichas variables
- **ngDoCheck**: Se ejecuta cuando el componente es redibujado con su **Render**.
Para implementarlo necesitamos **DoCheck**
- **ngOnDestroy**: Se ejecuta cuando el componente es destruido, es decir, cuando
Ha sido dibujado y quitado posteriormente de un Parent.

Los métodos pueden ser de acción (**void**) o pueden ser **return**, es decir, que devuelvan un valor.

Debemos indicar el tipo de método

Para crear métodos, es con la siguiente sintaxis dentro de Class:

```
nombreMetodoAccion(): void {
    //ACCIONES
}

nombreMetodoAccion(): TIPO DATO A DEVOLVER{
    //ACCIONES
    return valor;
}
```

Para los eventos, click, submit, on mouse over, la llamada se realiza con la siguiente sintaxis:

```
<button (click)="metodoAccion()">Pulsar</button>
```

Vamos a crear una carpeta llamada **hooksangular** y un componente llamado **hooksangular.component.ts** y creamos también un template llamado **hooksangular.component.html**

HOOKSANGULAR.COMPONENT.TS

```
import { Component, OnInit, DoCheck } from "@angular/core";
@Component({
  selector: "hooks-angular",
  standalone: false,
  templateUrl: "./hooksangular.component.html"
})
export class HooksAngular implements OnInit {
  public mensaje: string;
  constructor() {
    console.log("Constructor: Primer método de inicio de Component")
    this.mensaje = "Hoy es miércoles";
  }
  cambiarMensaje(): void {
    this.mensaje = "y mañana juernes!!!!";
  }
  ngOnInit(): void {
    console.log("Soy OnInit, después de constructor!!!!");
  }
  ngDoCheck(): void {
    console.log("NgCheck cambiando algo en Render!!!!");
  }
}
```

HOOKSANGULAR.COMPONENT.HTML

```
<div>
  <h1>Hooks Angular</h1>
  <h1 style="color: blue">{{ mensaje }}</h1>
  <button (click)="cambiarMensaje()">
    Ver mensaje
  </button>
</div>
```

Y veremos el resultado

Console

- Constructor: Primer método de inicio de Component
- Soy OnInit, después de constructor!!!!
- NgCheck cambiando algo en Render!!!!
- Angular is running in development mode.
- NgCheck cambiando algo en Render!!!!
- NgCheck cambiando algo en Render!!!!

DIRECTIVAS EN ANGULAR

Una directiva es código lógico dentro del HTML.

- **ngIf:** Condicionales
- **ngFor:** Para recorridos (bucles). Debemos utilizar la palabra **let** de forma obligatoria

let objeto in objetos

Como en VUE, las directivas van integradas en las etiquetas HTML

Para indicar una directiva se utiliza el asterisco antes de la directiva

```
<div *ngIf="condicion">
```

Vamos a realizar un clásico, un componente **deportes**

Dentro de **components** creamos un componente llamado **deportes.component.ts** y **deportes.component.html**

DEPORTES.COMPONENT.TS

```
import { Component } from "@angular/core";
@Component ({
  selector: "app-deportes",
  templateUrl: "./deportes.component.html",
  standalone: false
})
export class DeportesComponent {
  public sports: Array<string>;
  constructor() {
    this.sports = ["Canicas", "Curling", "Dardos", "Petanca"]
  }
}
```

DEPORTES.COMPONENT.HTML

```
<div>
  <h1>Directivas Angular</h1>
  <h2>Número de deportes: {{ sports.length }}</h2>
  <div *ngIf="sports.Length < 5">
    <p style="color:red">
      Tienes menos de 5 deportes
    </p>
  </div>
  <p>El primer deporte es {{ sports[0] }}</p>
  <ul>
    <li *ngFor="let deporte of sports; let i = index">
      Index: {{i}}, Deporte: {{deporte}}
    </li>
  </ul>
</div>
```

Y veremos el resultado.

Directivas Angular

Número de deportes: 4

Tienes menos de 5 deportes

El primer deporte es Canicas

- Index: 0, Deporte: Canicas
- Index: 1, Deporte: Curling
- Index: 2, Deporte: Dardos
- Index: 3, Deporte: Petanca

Con esta sintaxis, no existe un ELSE o ELSE IF tradicional.

Para poder realizar esta funcionalidad, necesitamos algo denominado **template**.

Tenemos una directiva llamada **ngTemplate** que, mediante un ID, podemos indicar Un TRUE o un FALSE para que lea un determinado código HTML.

Ejemplo:

```
<ng-template #templatetrue>
  <h1>TRUE</h1>
</ng-template>

<ng-template #templatefalse>
  <h1>FALSE</h1>
</ng-template>
```

Posteriormente, dentro de una directiva **ngIf**, podemos indicar que plantilla deseamos dibujar.

```
ngIf condicion == true templatetrue; else templatefalse
```

Si no utilizamos plantilla para TRUE, utiliza el de la propia etiqueta de la directiva **ngIf**

Modificamos el código de nuestro componente

```
<div *ngIf="sports.length < 5; else condicionelse">
  <p style="color: red">
    Tienes menos de 5 deportes
  </p>
</div>
<ng-template #condicionelse>
  <p style="color: blue">
    Tenemos más de 5 deportes!!!
  </p>
</ng-template>
```

Si deseamos tener más de un if, es decir, else if, debemos tener más plantillas.

```
ngIf condicion == 1; then PLANTILLA1; condicion == 2; then PLANTILLA2; else PLANTILLAELSE
```

```
<div *ngIf="sports.length < 5; then condiciontrue; else condicionelse">
</div>
<ng-template #condiciontrue>
  <p style="color: red">
    Tienes menos de 5 deportes
  </p>
</ng-template>
<ng-template #condicionelse>
  <p style="color: blue">
    Tenemos más de 5 deportes!!!
  </p>
</ng-template>
```

También tenemos directivas de estilo CSS, lo mismo que hicimos en VUE con Comics y el Año del comic.

Dicha directiva se llama **ngStyle** y podemos utilizarla **INLINE** en las etiquetas HTML

INLINE

```
<h2 [style.color]="condicion ? 'valor true': 'valor false'">Mensaje</h2>
```

Vamos a tener en el mismo ejemplo un Array de números.

Lo que haremos será evaluar cada número si es Par o Impar en Verde o Rojo.

```
export class DeportesComponent {
  public sports: Array<string>;
  public numeros: Array<number>
  constructor() {
    this.numeros = [4,5,6,7,78,99,2]
    this.sports = ["Canicas", "Curling", "Dardos", "Petanca", "Pelota Vasca"]
  }
}
```

Implementamos el código HTML

```
<ul>
  <li *ngFor="let num of numeros"
```

```
[style.color]="num % 2 == 0 ? 'green': 'red'">
    {{ num }}
</li>
</ul>
```

Podremos visualizar el resultado

Directivas Angular

- 4
- 5
- 6
- 7
- 78
- 99
- 2

Las directivas inline están bien, pero estamos limitados a cada estilo con cada directiva.
Si deseamos cambiar también el background, necesitamos otra directiva inline.

También tenemos las directivas de clase CSS

```
.estilo1 {
    //VERDE
}

.estilo2 {
    //ROJO
}
```

Posteriormente, podemos aplicar cada estilo con la directiva **[class.style]**

```
<h1 [class.estilo1]=condicion1
[class.estilo2]==condicion2>
```

Nos creamos un CSS para nuestro componente deportes: **deportes.component.css**

DEPORTES.COMPONENT.CSS

```
.par{
    background-color: #blue;
    color: #white;
    font-weight: bold;
}

.impar {
    background-color: #red;
    color: #yellow;
    font-weight: bold;
}
```

Aplicamos el CSS dentro de nuestro componente y escribimos la directiva en HTML

```
<ul>
    <li *ngFor="let num of numeros"
        [class.par]="num % 2 == 0"
        [class.impar]="num % 2 == 1">
            {{ num }}
    </li>
</ul>
```

Tenemos otra sintaxis para realizar lo mismo, utilizando la palabra **ngClass**

Dicha directiva utiliza la clase dentro del código.

```
<h1 [ngClass]="
{
    estilos1: condicion1,
    estilos2: condicion2
}>
">
```

```
<ul>
    <li *ngFor="let num of numeros"
        [ngClass]="{
            par: num % 2 == 0,
            impar: num % 2 == 1
        }">
            {{ num }}
        </li>
    </ul>
```

Tenemos otra sintaxis que podemos utilizar, de hecho, esta sintaxis es la recomendada por Angular. Actualmente.

Las directivas, actualmente podemos escribirlas con **@DIRECTIVA**

Dichas directivas ya NO están integradas en la etiqueta HTML, es más parecido a React con las llaves.

Se utiliza código JS puro.

Ejemplo IF:

```
@if (condicion) {
    <h1>Condición TRUE</h1>
}
```

Ejemplo Else If:

```
@if (condicion 1) {
    <h1>Soy Verdad</h1>
}@else if (condicion 2) {
    <h2>Soy el número 2</h2>
}@else {
    <h2>Soy el ELSE</h2>
}
```

Vamos a implementarlo, pero en otro componente llamado igual, pero versión 2 **deportesv2.component.ts**

```
<ul>
    @for (sport of sports; track sport){
        <li>{{sport}}</li>
    }@empty {
        <li>Sin elementos</li>
    }
</ul>
```

También podemos utilizar index con **\$index**

```
<ul>
    @for (sport of sports; track sport; let i = $index){
        <li>Index {{ i }}, {{sport}}</li>
    }@empty {
        <li>Sin elementos</li>
    }
</ul>
```

Con los IF tenemos la misma sintaxis:

```
<ul>
  @for (num of numeros; track num){
    @if (num % 2 == 0){
      <li style="color: green">{{ num }}</li>
    }@else {
      <li style="color: red">{{ num }}</li>
    }
  }
</ul>
```

Por supuesto, ya no necesitamos ni **ngTemplate** ni **ngClass**

```
<ul>
  @for (num of numeros; track num){
    @if (num % 2 == 0){
      <li class="par">{{ num }}</li>
    }@else {
      <li class="impar">{{ num }}</li>
    }
  }
</ul>
```

FORMULARIOS ANGULAR

El uso de formularios dentro de Angular está integrado, pero no está activado.
Cada vez que deseemos utilizar alguna herramienta, debemos decírselo a nuestro "jefe", **MODULE**

Tenemos dos formas de trabajar con formularios dentro de Angular:

- Model binding:** Al igual que hemos hecho en VUE, tendremos un modelo asociado a nuestros Controles de formulario HTML
- Object Reference:** Es lo mismo que hemos utilizando en React, una variable de referencia Del objeto HTML declarado en el código de Class.

Todos los controles de formulario, incluido el form deben tener un ID propio de Angular.
Los Ids de Angular se declaran con almohadilla: **#idobjeto**

También deben tener un **name**

El formulario debe tener una declaración **[ngForm]** para evitar **preventDefault**

Lo primero será, dentro de **app-module** dar de alta el uso de formularios en nuestra App.

APP-MODULE

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

```
@NgModule({
  declarations: [
    App,
    PrimerComponent,
    HooksAngular,
    DeportesComponent,
    DeportesV2Component
  ],
  imports: [
    BrowserModule, FormsModule
  ],
  providers: []
})
```

Vamos a comenzar utilizando **Binding** como ejemplo.
Nos creamos un componente llamado **formsbinding.component.ts**

Forms Binding Model

Datos recibidos

Nombre

Apellidos

Edad

**Nombre: Alumno, Apellidos Apellidos,
Edad: 12**

FORMSBINDING.COMPONENT.TS

```
import { Component } from "@angular/core";
@Component ({
  selector: "app-forms-binding",
  templateUrl: "./formsbinding.component.html",
  standalone: false
})
export class FormsBinding {
  public user: any;
  public mensaje: string;
  constructor() {
    this.mensaje = "";
    this.user = {
      nombre: "",
      apellidos: "",
      edad: 0
    }
  }
  recibirDatos(): void {
    this.mensaje = "Datos recibidos";
  }
}
```

FORMSBINDING.COMPONENT.HTML

```
<div>
  <h1>Forms Binding Model</h1>
  <h3>{{ mensaje }}</h3>
  <!-- UN FORMULARIO SIEMPRE DEBE TENER UN ID DE ANGULAR
  CON LA IGUALDAD A ngForm-->
  <form #userForm="ngForm" (ngSubmit)="recibirDatos()">
    <!-- LOS CONTROLES DEBEN TENER UNA ETIQUETA ngModel APUNTANDO AL ID
ANGULAR-->
    <!-- PARA INDICAR MODEL BINDING [(ngModel)]="Mi modelo" -->
    <label>Nombre</label>
    <input type="text" name="cajanombre"
    #cajanombre="ngModel"
    [(ngModel)]="user.nombre"/><br/>
    <label>Apellidos</label>
    <input type="text" name="cajaapellidos"
    #cajaapellidos="ngModel"
    [(ngModel)]="user.apellidos"/><br/>
    <label>Edad</label>
    <input type="text" name="cajaedad"
    #cajaedad="ngModel"
    [(ngModel)]="user.edad"/><br/>
    <button>Enviar datos</button>
  </form>
  <h2 style="color:blue">
    Nombre: {{user.nombre}}, Apellidos {{user.apellidos}},
    Edad: {{user.edad}}
  </h2>
</div>
```

FORMULARIOS REFERENCIA

Es exactamente igual a React con su maravilloso **createRef**

La gran ventaja es que no tenemos que escribir tanto código invasivo dentro del HTML y con los model.

La desventaja está en que si cambia el valor del formulario no cambia el modelo.

Debemos mantener los Ids de Angular: #idsangular

Pongamos que tenemos la siguiente caja en HTML

```
<input type="text" name="cajatexto" #cajatexto/>
```

Mediante la librería **ViewChild** podemos asociar un objeto dentro del Class con el Id de Angular.

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

```
@ViewChild("cajatexto") miCaja: ElementRef;
```

Posteriormente, en nuestro código, para recuperar el valor la caja:

```
this.miCaja.nativeElement.value
```

Tenemos que mantener dos normas:

1. El formulario debe seguir teniendo **ngForm** apuntando a su ID
2. Los objetos Form deben tener un ID de Angular

Vamos a realizar un nuevo componente por referencia. Vamos a sumar dos números.

Creamos un nuevo componente llamado **sumarnumeros.component.ts**

SUMARNUMEROS.COMPONENT.TS

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

@Component({
  selector: "app-sumar-numeros",
  templateUrl: "./sumarnumeros.component.html",
  standalone: false
})
export class SumarNumerosComponent {
  @ViewChild("cajanumero1") cajaNumero1Ref: ElementRef;
  @ViewChild("cajanumero2") cajaNumero2Ref: ElementRef;
  public suma: number;
  constructor() {
    this.suma = 0;
    //EN ANGULAR, AUNQUE SEAN REFERENCIAS, TODAS LAS VARIABLES DEBEN
    //SER INSTANCIADAS
    this.cajaNumero1Ref = new ElementRef(0);
    this.cajaNumero2Ref = new ElementRef(0);
  }
  sumarNumeros(): void {
    let num1 = this.cajaNumero1Ref.nativeElement.value;
    let num2 = this.cajaNumero2Ref.nativeElement.value;
    this.suma = parseInt(num1) + parseInt(num2);
  }
}
```

SUMARNUMEROS.COMPONENT.HTML

```
<div>
  <h1 style="color: red">Forms Referencia</h1>
  <form #sumarForm="ngForm">
    <label>Número 1</label>
    <input type="number" name="cajanumero1"
    #cajanumero1/><br/>
    <label>Número 2</label>
    <input type="number" name="cajanumero2"
    #cajanumero2/>
    <button (click)="sumarNumeros()">
      Mostrar suma
    </button>
  </form>
  <h2 style="color: fuchsia">
    La suma es: {{suma}}
  </h2>
</div>
```

Y veremos el resultado:

Forms Referencia

Número 1
 Número 2 Mostrar suma

La suma es: 11

Como hemos visto, Angular nos obliga a inicializar todos los objetos de la clase.
 Podemos "saltarnos" dicha norma **SOLO** si tenemos formulario de Referencia.

Para evitar instanciar el objeto new **ElementRef**, en la declaración de la variable se utiliza la **ADMIRACION**

```
export class SumarNumerosComponent {
  @ViewChild("cajanumero1") cajaNumero1Ref!: ElementRef;
  @ViewChild("cajanumero2") cajaNumero2Ref!: ElementRef;
```

CREACION DE COMPONENTS DINAMICOS ANGULAR

Con la extensión que hemos instalado para Angular, tenemos un comando para poder generar los Components de forma dinámica, incluido ponerlos dentro de Module automáticamente.

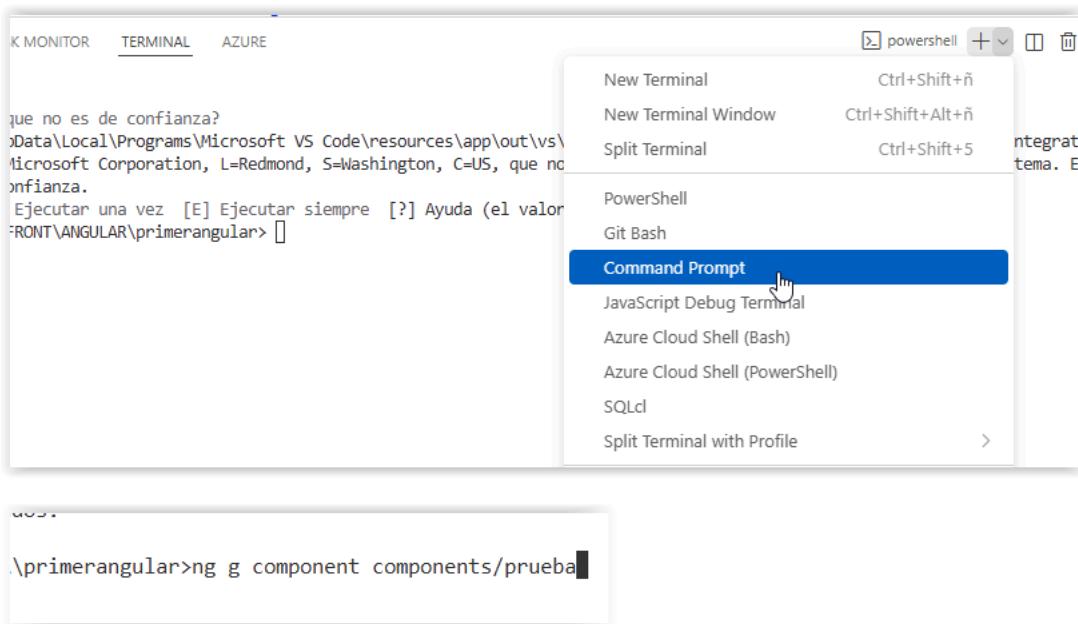
Nota: A veces, no incluye el component dentro de MODULE y me vuelvo loco.

Utilizamos la siguiente instrucción:

ng g component CARPETA/COMPONENTNAME

Vamos a utilizar la línea de comandos de VS code para hacerlo.

En el Terminal de VS Code (Control + ñ), utilizamos Command Prompt



Siempre pondremos un **punto** con la palabra **component** al final para mantener la dinámica de La sintaxis de **Angular**

Como podemos comprobar, nos crea el component dentro de **app/components**

```
Microsoft Windows [Versión 10.0.26100.6899]
(c) Microsoft Corporation. Todos los derechos reservados.
```

```
C:\Users\Profesor MCSD Mañana\Documents\FRONT\ANGULAR\primerangular>ng g component components/prueba
```

```
CREATE src/app/components/prueba/prueba.spec.ts (556 bytes)
CREATE src/app/components/prueba/prueba.ts (203 bytes)
CREATE src/app/components/prueba/prueba.css (0 bytes)
CREATE src/app/components/prueba/prueba.html (22 bytes)
UPDATE src/app/app-module.ts (1147 bytes)
```

Tenemos el mismo comando, pero podemos indicar dónde deseamos que nos genere el component.

```
ng g component NOMBRECOMPONENT --path src/components --skip-import
```

Creamos un nuevo component llamado **test.component**

```
C:\Users\Profesor MCSD Mañana\Documents\FRONT\ANGULAR\primerangular>ng g component test.component --path src/components --skip-import
CREATE src/components/test.component/test.component.spec.ts (606 bytes)
CREATE src/components/test.component/test.component.ts (234 bytes)
CREATE src/components/test.component/test.component.css (0 bytes)
CREATE src/components/test.component/test.component.html (30 bytes)
```

TABLA MULTIPLICAR PRACTICA

Realizar un nuevo component llamado **tablamultiplicar** donde pediremos un número al usuario
Mediante un Formulario y debemos dibujar la tabla de multiplicar con Operación y Resultado
De dicho número.

Forms: Al gusto (ref o binding)

- Debemos hacerlo con Array<number>
- Hacemos el dibujo dinámico con ngFor y con @for, es decir, el Array solamente tendrá el Resultado de las operaciones.

Tabla multiplicar

Introduzca número	<input type="text" value="7"/>	Tabla multiplicar
Operación	Resultado	
7 * 1	7	
7 * 2	14	
7 * 3	21	
7 * 4	28	
7 * 5	35	
7 * 6	42	
7 * 7	49	
7 * 8	56	
7 * 9	63	
7 * 10	70	

TABLA.MULTIPLICAR.COMPONENT.TS

```
import { Component, ViewChild, ElementRef } from '@angular/core';
@Component({
  selector: 'app-tabla-multiplicar',
  standalone: false,
  templateUrl: './tabla.multiplicar.component.html',
  styleUrls: ['./tabla.multiplicar.component.css'],
})
export class TablaMultiplicarComponent {
  @ViewChild("cajanumero") cajaNumero!: ElementRef;
  public numero: number;
  public numeros: Array<number>;
  constructor() {
    this.numero = 0;
    this.numeros = new Array<number>();
  }
  mostrarTabla(): void {
  }
}
```

```

        this.numero = parseInt(this.cajaNumero.nativeElement.value);
        let aux = new Array<number>();
        for (var i = 1; i <= 10; i++){
            var operacion = this.numero * i;
            aux.push(operacion);
        }
        this.numeros = aux;
    }
}

```

TABLA.MULTIPLICAR.COMPONENT.HTML

```

<div>
    <h1>Tabla multiplicar</h1>
    <form #tablaForm="ngForm">
        <label>Introduzca número</label>
        <input type="number" name="cajanumero" #cajanumero/>
        <button (click)="mostrarTabla()">
            Tabla multiplicar
        </button>
    </form>
    @if (numeros.length > 0){
        <table border="1" style="background-color:lightblue">
            <thead>
                <tr>
                    <th>Operación</th>
                    <th>Resultado</th>
                </tr>
            </thead>
            <tbody>
                @for (num of numeros; track num; let i = $index){
                    <tr>
                        <td>{{ numero }} * {{i + 1}}</td>
                        <td>{{ num }}</td>
                    </tr>
                }
            </tbody>
        </table>
    }
    <table border="1" *ngIf="numeros.Length > 0">
        <thead>
            <tr>
                <th>Operación</th>
                <th>Resultado</th>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="Let num of numeros; Let i = index">
                <td>{{ numero }} * {{i + 1}}</td>
                <td>{{ num }}</td>
            </tr>
        </tbody>
    </table>
</div>

```

RUTAS CON ANGULAR

Por defecto, Angular SI que tiene integrado Routing dentro de su entorno, no es necesario traer Nada mediante **npm**

Vamos a crear un nuevo proyecto llamado **rutasangular**

ng new rutasangular --standalone=false --routing=false

```
C:\Users\Profesor MCSD Mañana\Documents\FRONT\ANGULAR
λ ng new rutasangular --standalone=false --routing=false
```

En Angular tenemos Routing, pero debemos activarlo para trabajar con él dentro de **module**

Funciona igual que React y Vue, necesitamos un fichero para mapear las rutas a cada component

Dicho fichero lo declararemos dentro de **module** para que puedan utilizarlo todos los components

La navegación de los links se realiza mediante etiquetas tradicionales **<a>**, pero con un atributo especial

<a [routerlink]="path">Navegar

En Angular el fichero de rutas se llama **app.routing.ts**

Nota: En Angular, no se utilizan las barras en las rutas

Necesitamos los siguientes elementos:

Un Array con las rutas y components

```
Routes = [
    { path: "/", component: MiComponent }
```

]

Para dibujar cada componente dentro de una ZONA. <router-outlet>

Para comodidad, vamos a trabajar en la ruta **app/components**

Realizamos nuestro clásico de Home, Musica y Cine

Sobre **public** creamos una carpeta llamada **assets** y dentro una carpeta llamada **images** con cada Imagen para cada componente.

También una imagen para un 404 Not Found

Sobre la carpeta **app** creamos una nueva carpeta llamada **components**

Creamos los tres components:

```
ng g component components/home.component
```

```
ng g component components/cine.component
```

```
ng g component components/musica.component
```

```
<div>
  <h1>Música Component</h1>
  
</div>
```

Sobre **src/app** creamos un nuevo fichero llamado **app.routing.ts**

Nota: Las rutas en angular en la declaración no llevan Barra

APP.ROUTING.TS

```
import { HomeComponent } from './components/home.component/home.component';
import { CineComponent } from './components/cine.component/cine.component';
import { MusicaComponent } from './components/musica.component/musica.component';
//NECESITAMOS UNA SERIE DE MODULOS QUE SE ENCUENTRAN DENTRO DE ANGULAR/ROUTER
import { Routes, RouterModule } from '@angular/router';
import { ModuleWithProviders } from '@angular/core';
//NECESITAMOS UN ARRAY CON LAS RUTAS, DICHO ARRAY SERA DE TIPO Routes
const appRoutes: Routes = [
  { path: "", component: HomeComponent},
  { path: "cine", component: CineComponent},
  { path: "musica", component: MusicaComponent}
]
//DESDE ESTA CLASE DEBEMOS EXPORTAR EL ARRAY DE ROUTES COMO PROVIDER
export const appRoutingProvider: any[] = [];
//LAS PROPIAS RUTAS A EXPORTAR
export const routing: ModuleWithProviders<any> =
RouterModule.forRoot(appRoutes);
```

El siguiente paso es declarar nuestras rutas dentro de **app.module.ts**

APP.MODULE.TS

```
import { routing, appRoutingProvider } from './app.routing';
```

Dentro de imports declaramos **routing**

Dentro de providers declaramos **appRoutingProvider**

```
@NgModule({
  declarations: [
    App,
    HomeComponent,
    CineComponent,
    MusicaComponent
  ],
  imports: [
    BrowserModule, routing
```

```

    ],
  providers: [
    provideBrowserGlobalErrorListeners(),
    appRoutingProvider
  ],
  bootstrap: [App]
)
export class AppModule { }

```

Para comprobar la funcionalidad, simplemente escribimos en **app.component** la etiqueta **<router-outlet>**

APP.COMPONENT.HTML

```

<div>
  <h1>Ejemplo rutas</h1>
  <hr/>
  <router-outlet></router-outlet>
</div>

```

El siguiente paso es crear un componente para el menú. Creamos un nuevo componente llamado **menu.component**

```

<div>
  <ul id="menu">
    <li>
      <a [routerLink]="/">Home | </a>
    </li>
    <li>
      <a [routerLink]="/cine">Cine | </a>
    </li>
    <li>
      <a [routerLink]="/musica">Música</a>
    </li>
  </ul>
</div>

```

A diferencia de VUE y REACT, si no encuentra una ruta nos lleva a Home y no muestra component

Para indicar una ruta No encontrada, debemos utilizar el doble asterisco en el Path y siempre debe Ser el último.

Creamos un último componente llamado **notfound.component**

Modificamos **app.routing.ts** incluyendo doble asterisco en el último path apuntando a notfound

```

//NECESITAMOS UN ARRAY CON LAS RUTAS, DICHO ARRAY
const appRoutes: Routes = [
  { path: "", component: HomeComponent },
  { path: "cine", component: CineComponent },
  { path: "musica", component: MusicaComponent },
  { path: "**", component: NotfoundComponent }
]

```

RUTAS CON PARAMETROS

Las rutas son iguales al resto de Frameworks, los parámetros se declaran con los dos puntos.

Dichas rutas deben estar declaradas/mapeadas en Routing

```
{ path: "ruta/:parametro", component: PruebaComponent}
```

Lo que cambia es cómo recuperamos dichos parámetros.

En Angular los parámetros se recuperan mediante un objeto por "arte de magia".

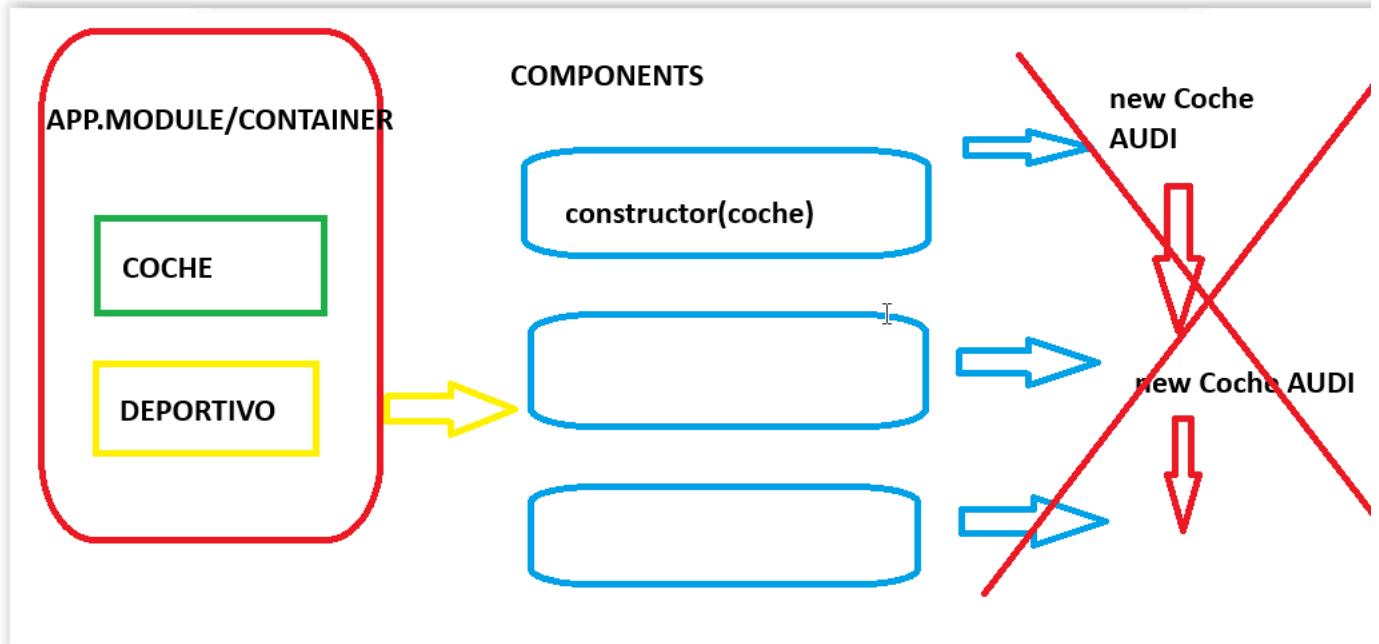
Dicho objeto nos ofrece los parámetros por su nombre.

Angular tiene de serie dos arquitecturas llamadas **IoC** (Inversión de Control) y **Dependency Injection**

- **IoC:** Inversión de control. Nos permite inyectar nuestras propias clases en un punto y poder recuperar dichas clases en cualquier punto de la aplicación
- **Dependency Injection:** Es la forma de recuperar dichas clases.

Estas dos arquitecturas nos permiten recuperar objetos en los constructores por arte de magia.

Este concepto se denomina principio de Hollywood: No nos llame usted, ya le llamamos nosotros



En nuestro ejemplo actual para recuperar rutas, solamente se utiliza DI.

Los proveedores (IoC) están ya incluidos dentro de **Providers** y **Routing**

En el momento de trabajar, recibiremos en el constructor de cualquier componente un objeto.

Nota: En Angular, los objetos inyectados se denominan con el guion bajo.

```
export class MenuComponent {
  constructor(_coche){}

}
```

En nuestro ejemplo de Routing, debemos recibir un objeto de tipo **ActivatedRoute**

```
export class MenuComponent {
  constructor(_activateRoute: ActivatedRoute){

}
}
```

Posteriormente, una vez que tenemos el objeto inyectado, debemos suscribirnos a la "escucha" de los parámetros de la ruta.

Debemos dividir las acciones:

- En el constructor, se reciben los objetos.
- En ngOnInit se utilizan los objetos.

Para recuperar parámetros se realiza mediante el **:name** que hemos puesto en **path** del **routing**

```
ngOnInit(): void {
    this._activateRoute.params.subscribe((parametros: Params) => {
        //AQUI TENEMOS LOS PARAMETROS POR SU NAME
    })
}
```

Vamos a probar esto con un component para el **número doble**.

Creamos un component llamado **numero.doble.component**

Sobre Routing, vamos a crear dos rutas para el mismo component, una sin parámetro y otra con Parámetro.

APP.ROUTING.TS

```
//NECESITAMOS UN ARRAY CON LAS RUTAS, DICHO ARRAY SERA DE TIPO Routes
const appRoutes: Routes = [
    { path: "", component: HomeComponent },
    { path: "cine", component: CineComponent },
    { path: "musica", component: MusicaComponent },
    { path: "numerodoble", component: NumeroDobleComponent },
    { path: "numerodoble/:numero", component: NumeroDobleComponent },
    { path: "**", component: NotfoundComponent }
```

El siguiente paso es realizar el diseño del component

NUMERODOBLE.COMPONENT.HTML

```
<div>
    <h1>Doble Routing</h1>
    @if (numero) {
        <h2 style="color: blue">
            El doble de {{numero}} es {{doble}}
        </h2>
    }@else{
        <h2 style="color: red">
            No hemos recibido parámetro
        </h2>
    }
</div>
```

NUMERODOBLE.COMPONENT.TS

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
@Component({
    selector: 'app-numero-doble',
    standalone: false,
    templateUrl: './numero.doble.component.html',
    styleUrls: ['./numero.doble.component.css'],
})
export class NumeroDobleComponent implements OnInit {
    public doble: number;
    public numero!: number;
    constructor(private _activeRoute: ActivatedRoute) {
        this.doble = 0;
    }
    ngOnInit(): void {
        //AQUI ES DONDE NOS SUBSCRIBIMOS A LOS PARAMETROS
        this._activeRoute.params.subscribe((parametros: Params) => {
            //DENTRO DE Params ES DONDE RECIBIMOS LOS PARAMETROS POR SU :name
            //LA SINTAXIS PARA RECUPERARLOS ES params['PARAMETER NAME']
            //EN ESTE EJEMPLO, EL PARAMETRO ES OPCIONAL
            if (parametros['numero'] != null) {
                this.numero = parametros['numero'];
            }
        })
    }
}
```

```
//EL PARAMETRO SIEMPRE SON STRING
this.numero = parseInt(parametros['numero']);
this.doble = this.numero * 2;
}
}
}
```

En el menú component, creamos unas rutas con parámetros

MENUCOMPONENT

```
<li>
    <a [routerLink]="/numerodoble">Doble</a>
</li>
<li>
    <a [routerLink]="/numerodoble/77">Doble 77</a>
</li>
<li>
    <a [routerLink]="/numerodoble/7">Doble 7</a>
</li>
```

Y podremos comprobar el resultado de la aplicación:

Ejemplo rutas

[Home](#) | [Cine](#) | [Música](#) | [Doble](#) | [Doble 77](#) | [Doble 7](#)

Doble Routing

El doble de 77 es 154

REDIRECCION CON ROUTING

Por supuesto, podemos navegar con routing mediante código class.

Para realizar este nuevo elemento de navegación, es necesario recibir, dentro del constructor, otro objeto de la clase **Router**

Posteriormente, dicho **Router** tiene un método **navigate()** que nos permite ir a otras rutas dentro del Path.

```
constructor(
  private _router: Router
){
}

goToHome(): void {
  this._router.navigate(["/"])
}
```

Tenemos la posibilidad de enviar parámetros dentro del objeto **Router**

Si queremos enviar parámetros, se realiza con los parámetros a continuación de la ruta del Path

Para enviar parámetros se realiza con coma, cada parámetro en el orden de la ruta.
No se pone la última barra en la navegación.

```
this._router.navigate(["/rutaparams"], params)
```

Para probarlo, tendremos un botón sin parámetros que nos llevará a Home desde Número doble.

Posteriormente, utilizando Routing, haremos la navegación con parámetros enviando el número Doble.

NUMERODOBLE.COMPONENT.TS

```
constructor(
  private _activeRoute: ActivatedRoute,
  private _router: Router
){
  this.doble = 0;
}

goToHome(): void {
  this._router.navigate(["/"]);
}
```

Incluimos un botón en el dibujo del componente

```
<button (click)="goToHome()">
  Go to home
</button>
```

El siguiente paso es crear un método que recibirá un número y utilizará Routing para poder Navegar. Haremos la redirección sobre nosotros mismos.

```
redirect(num: number): void {
  this._router.navigate(["/numerodoble", num])
```

Dibujamos una serie de botones enviando un número al método

```
<button (click)="redirect(44)">
  Número 44
</button>
<button (click)="redirect(55)">
  Número 55
</button>
<button (click)="redirect(14)">
  Número 14
</button>
```

Y tendremos el resultado

Doble Routing

```
Go to home Número 44 Número 55 Número 14
```

El doble de 14 es 28

Por último, si deseamos realizar links dinámicos, la sintaxis es la misma que con Router navigate.

```

<li *ngFor="let valor of Array">
    <a [routerLink]="/path, valor">
        Component {{valor}}
    </a>
</li>

```

```

@for (valor of Array; track valor){
    <li>
        <a [routerLink]="/path, valor">
            Component {{valor}}
        </a>
    </li>
}

```

Vamos a realizar un clásico, la Tabla de multiplicar con Rutas.

[Tabla 64](#) | [Tabla 45](#) | [Tabla 97](#) | [Tabla 88](#) | [Tabla 68](#) | [Tabla 44](#) |

Tabla multiplicar routing 68

Operación	Resultado
68 * 1	68
68 * 2	136
68 * 3	204
68 * 4	272
68 * 5	340
68 * 6	408

Necesito un component llamado **tablamultiplicarrouting** que recibirá un parámetro de un Número (no opcional) y mostramos la tabla de multiplicar.

También necesitamos un component **menutablamultiplicar** donde generamos 6 números aleatorios Y dibujamos sus rutas dinámicas con **routerLink**

MENUTABLAMULTIPLICAR.TS

```

import { Component } from '@angular/core';
@Component({
  selector: 'app-menu.tabla.component',
  standalone: false,
  templateUrl: './menu.tabla.component.html',
  styleUrls: ['./menu.tabla.component.css'],
})
export class MenuTablaComponent {
  public numerosRandom: Array<number>;
  constructor() {
    this.numerosRandom = new Array<number>();
    for (var i = 1; i <= 6; i++) {
      let random = Math.floor((Math.random() * 100)) + 1;
      this.numerosRandom.push(random);
    }
  }
}

```

MENUTABLAMULTIPLICAR.HTML

```

<ul id="menu">
    <for (num of numerosRandom; track num){>

```

```

<li>
    <a [routerLink]="/tabla", num>Tabla {{num}}</a>
</li>
}
</ul>

```

APP.ROUTING.TS

```
{ path: "tabla/:numero", component: TablaMultiplicarComponent},
```

TABLAMULTIPLICAR.COMPONENT.TS

```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
@Component({
  selector: 'app-tabla.multiplicar.component',
  standalone: false,
  templateUrl: './tabla.multiplicar.component.html',
  styleUrls: ['./tabla.multiplicar.component.css'],
})
export class TablaMultiplicarComponent implements OnInit {
  public numero: number;
  public numerosTabla: Array<number>;
  constructor(private _activeRoute: ActivatedRoute) {
    this.numero = 0;
    this.numerosTabla = new Array<number>();
  }
  ngOnInit(): void {
    this._activeRoute.params.subscribe((params: Params) => {
      this.numero = parseInt(params['numero']);
      this.generarTabla();
    })
  }
  generarTabla(): void {
    let aux = new Array<number>();
    for (var i = 1; i <= 10; i++) {
      let operacion = this.numero * i;
      aux.push(operacion);
    }
    this.numerosTabla = aux;
  }
}

```

TABLAMULTIPLICAR.COMPONENT.HTML

```

<div>
  <h1 style="color:blue">
    Tabla multiplicar routing {{numero}}
  </h1>
  <table border="1">
    <thead>
      <tr>
        <th>Operación</th>
        <th>Resultado</th>
      </tr>
    </thead>
    <tbody>
      @for (num of numerosTabla; track num; let i = $index){
        <tr>
          <td>{{numero}} * {{ i + 1}}</td>
          <td>{{ num }}</td>
        </tr>
      }
    </tbody>
  </table>
</div>

```

MODELOS EN ANGULAR

Creamos un nuevo proyecto llamado **angularcomunicacionmodelos**

Un Model representa un objeto dentro de Angular y podemos definirlo con una serie de propiedades.

Por ejemplo, un Model es un objeto Departamento, pero está definido con sus propiedades tipadas, lo que hace que sea más sólido.

La ventaja de tener modelos es la comprobación de posibles errores en la asignación de propiedades y poder enviar/recibir datos entre components o servicios.

La ventaja de utilizar Models es que solamente definimos el objeto una vez a lo largo de nuestra App.

Ejemplo de un Model:

```

export class Persona {
    public nombre: string;
    public edad: number;

    //DEBEMOS UTILIZAR UN CONSTRUCTOR PARA INICIALIZAR LAS PROPIEDADES
    constructor() {
        this.nombre = "Alumno";
        this.edad = 22;
    }
    //UN CONSTRUCTOR U OTRO, NO EXISTE LA SOBRECARGA
    constructor(name: string, age: number){
        this.nombre = name;
        this.edad = age;
    }
}

```

Posteriormente, podremos crear objetos del Modelo Persona en cualquier lugar.

```

public personaje: Persona = new Persona();
personaje.nombre = "algo"
personaje.edad = 11;

```

Si utilizamos el segundo constructor (amarillo)

```
public personaje: Persona = new Persona("Nombre", 55);
```

Vamos a realizar un ejemplo utilizando objetos productos.

Modelos Producto Angular

Nike Air Jordan, Precio: 150



Nike Air Mag, Precio: 1900



Vamos a trabajar sobre **app**. Creamos una carpeta llamada **models** y dentro una clase nueva llamada **producto.ts**

PRODUCTO.TS

```

export class Producto {
    //LAS PROPIEDADES PARA QUE SEAN ACCESIBLES DEBEN SER PUBLICAS
    public nombre: string;
    public imagen: string;
    public precio: number;

    //VAMOS A CREAR UN CONSTRUCTOR CON PARAMETROS PARA QUE RECIBA
    //LA INFORMACION DE LAS PROPIEDADES DIRECTAMENTE
    constructor(name: string, image: string, price: number){
        this.nombre = name;
        this.imagen = image;
        this.precio = price;
    }
}

```

Sobre app creamos una carpeta llamada **components** y un component llamado **listaproductos**

LISTAPRODUCTOS.COMPONENT.HTML

```
<div>
  <h1>Modelos Producto Angular</h1>
  @for (prod of productos; track prod){ 
    <div>
      <h2 style="color:blue">
        {{prod.nombre}}, Precio: {{prod.precio}}
      </h2>
      <img src={{prod.imagen}} style="width: 150px; height: 150px"/>
    </div>
  }
</div>
```

LISTAPRODUCTOS.COMPONENT.TS

```
import { Component } from '@angular/core';
import { Producto } from '../../../../../models/producto';
@Component({
  selector: 'app-lista-productos',
  standalone: false,
  templateUrl: './lista.productos.component.html',
  styleUrls: ['./lista.productos.component.css'],
})
export class ListaProductosComponent {
  public producto: Producto;
  public productos: Array<Producto>;
  constructor() {
    this.producto = new Producto("Camiseta",
"https://contents.mediadecathlon.com/p2965406/k\$d31324aa28817ec6ae130efda4df04ef/picture.jpg", 180);
    this.productos = new Array<Producto>();
    this.productos.push(this.producto);
    this.productos = [
      new Producto(
        "Nike Air Jordan",
        "https://images.zapantojos.com/media/2022/07/39253d37.jpg",
        150
      ),
      new Producto(
        "Nike Air Mag",
        "https://limitedresell.com/img/anblog/b/b-654d14cf06f5-anblog\_thumb.jpg",
        1900
      ),
      new Producto(
        "New Balance 998",
        "https://www.sneakers-actus.fr/wp-content/uploads/2023/06/NB-998-U998TE-x-Teddy-Santis-Plum-Purple-MIUSA.jpg",
        140
      ),
      new Producto(
        "J-Hayber Olimpo",
        "https://jhayber.com/documents/images/products/63638-850-1.jpg",
        60
      ),
      new Producto(
        "Triple S Balenciaga",
        "https://cdn1.jolicloset.com/imgr/full/2024/05/1321192-1/plastico-zapatillas-balenciaga-triple-s-de-políuretano-blanco-amarillo.jpg",
        650
      )
    ]
  }
}
```

Una vez que ya lo tenemos, existe otra sintaxis para los modelos y sus propiedades. Dicha sintaxis, lo que hace es un constructor con parámetros y propiedades todo a la vez

Es más fácil de crear en código y en este ejemplo, estamos obligados a utilizar un constructor Con parámetros.

La sintaxis es extraña.

```
export class Producto {
  constructor(
    public nombre: string,
    public imagen: string,
    public precio: number
  ) {}
```

Implementamos Routing en este proyecto.
Quiero un menú con un Link y mostrar los productos en <router-outlet>

COMUNICACIÓN COMPONENTES ANGULAR

Por supuesto, podemos comunicarnos entre componentes en Angular.

La comunicación es bidireccional y cambia la sintaxis.

Las propiedades para comunicar entre componentes (**props**) utilizan la siguiente sintaxis:

PADRE COMPONENT

```
<hijo-component [propiedad] = "valor"></hijo-component>
```

Posteriormente, dentro del componente hijo, para recuperar las propiedades tenemos una clase llamada `@Input`

```
export class HijoComponent {
  @Input propiedad: tipoDato
}
```

Vamos a realizar comunicación entre padre e hijo.

Para ello, tendremos un modelo `Coche`, tendremos un componente `PadreCoches` y otro componente `HijoCoche`

Comenzamos realizando solamente un HIJO y comprobamos su funcionalidad

Pontiac Firebird

Arrancando!!!

Velocidad actual 0 km/h

Sobre `models` creamos una nueva clase llamada `coche.ts`

COCHE.TS

```
export class Coche {
  constructor(
    public marca: string,
    public modelo: string,
    public velocidad: number,
    public aceleracion: number,
    public estado: boolean
  ){}
}
```

Creamos un componente llamado `hijocoche.component.ts`

Dentro de su CSS, creamos un diseño rojo o verde para visualizar el estado.

```
.rojo {
  background-color: lightcoral;
}

.verde {
```



HIJOCOCHECOMPONENT.TS

```
import { Component } from '@angular/core';
import { Coche } from '../../../../../models/coche';
@Component({
  selector: 'app-hijo-coche',
  standalone: false,
  templateUrl: './hijo.coche.component.html',
  styleUrls: ['./hijo.coche.component.css'],
})
export class HijoCocheComponent {
  public car: Coche;
  public mensaje: string;
  constructor() {
    this.car = new Coche("Pontiac", "Firebird", 240, 25, false);
    this.mensaje = "";
  }
  comprobarEstado(): boolean {
    if (this.car.estado == false){
      this.mensaje = "El coche está apagado!!!";
      this.car.velocidad = 0;
      return false;
    }else{
      this.mensaje = "Arrancando!!!";
      return true;
    }
  }
  encenderCoche(): void {
    this.car.estado = !this.car.estado;
    this.comprobarEstado();
  }
  acelerarCoche(): void {
    if (this.comprobarEstado() == false){
      alert("¿dónde vas? Que estoy apagado!!!!")
    }else{
      this.car.velocidad += this.car.aceleracion;
    }
  }
}
```

HIJOCOCHECOMPONENT.HTML

```
<div>
  <h1 style="color:blue">
    {{car.marca}} {{car.modelo}}
  </h1>
  <h2 [ngClass]="{
    verde: car.estado == true,
    rojo: car.estado == false
  }">
    {{mensaje}}
  </h2>
  <h2>Velocidad actual {{car.velocidad}} km/h</h2>
  <button (click)="encenderCoche()">
    Start/stop coche
  </button>
  <button (click)="acelerarCoche()">
    Acelerar coche
  </button>
</div>
```

El siguiente paso es comprobar **props** con un Parent que contenga múltiples coches.

Tenemos dos formas de plantearlo:

1. Enviando propiedades y capturando las propiedades con @Input **individuales**

```
@for (car of arrayCoches) {
  <hijo-coche [marca]="car.marca" [modelo]="car.modelo">
}
```

HijoCocheComponent

```
@Input marca: string
@Input modelo: string
```

2. Enviando un objeto desde **props** y solamente teniendo un @Input

```
@for (car of arrayCoches) {
  <hijo-coche [cochecito]="car">
}
```

HijoCocheComponent

```
@Input coche: Coche
```

Creamos un nuevo componente llamado **PadreCoches**

PADRECOCHESCOMPONENT.TS

```
import { Component } from '@angular/core';
import { Coche } from '../../../../../models/coche';
@Component({
  selector: 'app-padre-coches',
  standalone: false,
  templateUrl: './padre-coches.component.html',
  styleUrls: ['./padre-coches.component.css'],
})
export class PadreCochesComponent {
  public cochesArray: Array<Coche>;
  constructor() {
    this.cochesArray = [
      new Coche("Ford", "Mustang", 400, 30, false),
      new Coche("Volkswagen", "Escarabajo", 110, 4, false),
      new Coche("Lamborghini", "Diablo", 800, 72, false)
    ]
  }
}
```

PADRECOCHESCOMPONENT.HTML

```
<div>
  <h2 style="color: red">Padre coches</h2>
  @for (coche of cochesArray; track coche){
    <app-hijo-coche [car]="coche"></app-hijo-coche>
  }
</div>
```

Si incluimos ahora mismo el objeto Padre Coche, podremos comprobar que las propiedades NO
Son opcionales

NG8002: Can't bind to 'car' since it isn't a known property of 'app-hijo-coche'.

1. If 'app-hijo-coche' is an Angular component and it has 'car' input, then verify that it is part of this module.
2. If 'app-hijo-coche' is a Web Component then add '**CUSTOM_ELEMENTS_SCHEMA**' to the '**@NgModule.schemas**' of this component to suppress this message.
3. To allow any property add '**NO_ERRORS_SCHEMA**' to the '**@NgModule.schemas**' of this component.

El siguiente paso es capturar **props** dentro de **hijo coche** mediante un objeto **@Input**

HIJOCOCHECOMPONENT.TS

```
import { Component, Input } from '@angular/core';
import { Coche } from '../../../../../models/coche';

@Component({
  selector: 'app-hijo-coche',
  standalone: false,
  templateUrl: './hijo.coche.component.html',
  styleUrls: ['./hijo.coche.component.css'],
})
export class HijoCocheComponent {
  @Input() car!: Coche;
  public mensaje: string;
```

```
constructor() {
    this.mensaje = "";
}
```

Y podremos comprobar la funcionalidad

[Productos | Padre coches](#)

Padre coches

Ford Mustang

Velocidad actual 400 km/h

[Start/stop coche](#) [Acelerar coche](#)

Volkswagen Escarabajo

Velocidad actual 110 km/h

[Start/stop coche](#) [Acelerar coche](#)

Lamborghini Diablo

Velocidad actual 800 km/h

[Start/stop coche](#) [Acelerar coche](#)

ENVIAR INFORMACION HIJO A PARENT

Para enviar información de un hijo a un parent se debe realizar mediante métodos
(Seleccionar favorito)

Para enviar la información desde el hijo al parent, utilizamos **@Output**

La variable **@Output** debe ser de tipo **Emitter**, lo que significa que tendremos un método en el Parent que estará asociado en el hijo.

COMPONENTE HIJO

`@Output metodoPadre: Emitter;`

El hijo tendrá un método a su vez:

```
metodoHijo(event): void {
    this.metodoPadre.emit();
}
```

CODIGO HTML DEL HIJO

`<button (click)="metodoHijo($event)">Llamar a papá</button>`

COMPONENT PADRE

`metodoPadre(event): void {`

`}`

HTML PADRE

`<hijo [propiedad]="valor" (metodoPadre)="metodoPadre($event)">`

Vamos a realizar un ejemplo sencillo/clásico: Deporte favorito.
Tendremos un componente llamado **PadreDeportes** y **HijoDeporte**

Padre deportes

Deporte favorito: Futbol

- Futbol

- Basket

- Beisbol

- Natación

PADREDEPORTES.COMPONENT.TS

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-padre-deportes.component',
  standalone: false,
  templateUrl: './padre-deportes.component.html',
  styleUrls: ['./padre-deportes.component.css'],
})
export class PadreDeportesComponent {
  public deportes: Array<string>;
  public mensaje: string;
  //NECESITAMOS UN METODO QUE RECIBIRA EVENT QUE VA A SER CUALQUIER TIPADO
  (any)
  seleccionarFavoritoPadre(event: string): void {
    this.mensaje = "Deporte favorito: " + event;
    console.log("Comunicando al padre!!! " + event);
  }
  constructor() {
    this.mensaje = "";
    this.deportes = ["Futbol", "Basket", "Beisbol", "Natación", "Dados"]
  }
}
```

PADREDEPORTES.COMPONENT.HTML

```
<div>
  <h1>Padre deportes</h1>
  <h2 style="color:fuchsia">{{mensaje}}</h2>
  <ul>
    @for (deporte of deportes; track deporte){
      <li>
        <app-hijo-deporte [sport]="deporte"
        (seleccionarFavoritoPadre)="seleccionarFavoritoPadre($event)"
        ></app-hijo-deporte>
      </li>
    }
  </ul>
</div>
```

HIJODEPORTE.COMPONENT.TS

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'app-hijo-deporte',
  standalone: false,
  templateUrl: './hijo-deporte.component.html',
  styleUrls: ['./hijo-deporte.component.css'],
})
export class HijoDeporteComponent {
  @Input() sport!: string;
  @Output() seleccionarFavoritoPadre: EventEmitter<any> = new
  EventEmitter();
  //NECESITAMOS UN METODO PARA QUE CADA HIJO REALICE CLICK Y, A SU VEZ,
  //LLAMAMOS AL PARENT
  seleccionarFavoritoHijo(): void {
    //DENTRO DE emit() ENVIREMOS LOS PARAMETROS QUE NECESITEMOS
    this.seleccionarFavoritoPadre.emit(this.sport);
  }
}
```

HIJODEPORTE.COMPONENT.HTML

```
<div>
  <p style="color:blue">{{ sport }}</p>
  <button (click)="seleccionarFavoritoHijo()">
    Seleccionar favorito
  </button>
</div>
```

Vamos a realizar una práctica de Comics.

Librería comics

Título	Peter
Descripción	Porker
Imagen	https://i.pinimg.com/original
Nuevo comic	

[Spiderman](#) [Wolverine](#) [Guardianes de la Galaxia](#) [Avengers](#) [Spawn](#) [Pete](#)

[Hombre araña](#) [Lobezno](#)



[Favorito](#) [Delete](#)



[Favorito](#) [Delete](#)



[Favorito](#) [Delete](#)

[Yo soy Groot](#)

[Los Vengadores](#) [Todd MacFarlane](#) [Porke](#)



[Favorito](#) [Delete](#)



[Favorito](#) [Delete](#)



[Favori](#)

Tendremos un componente llamado **LibreriaComponent** y otro llamado **ComicComponent**

Tendremos también un Modelo Comic.

La funcionalidad será:

- Dibujar todos los comics desde un Parent
- Crear nuevos comics dentro de un Parent
- Eliminar un Comic
- Seleccionar favorito

Descargamos **Comics_Angular.txt**

Sobre **models** creamos un nuevo modelo llamado **comic.ts**

COMIC.TS

```
export class Comic {
  constructor(
    public titulo: string,
    public imagen: string,
    public descripcion: string
  ){}}
```

Sobre **Modules** habilitamos formularios.

APP MODULE.TS

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

```
@NgModule({
  declarations: [
    App,
    ListaProductosComponent,
```

```

    MenuComponent,
    HijoCochesComponent,
    PadreCochesComponent,
    PadreDeportesComponent,
    HijoDeporteComponent,
    LibreriaComponent,
    ComicComponent
],
imports: [
  BrowserModule, routing, FormsModule
],

```

LIBRERIA.COMPONENT.TS

```

import { Component, ViewChild, ElementRef } from '@angular/core';
import { Comic } from '../../../../../models/comic';
@Component({
  selector: 'app-libreria',
  standalone: false,
  templateUrl: './libreria-component.html',
  styleUrls: ['./libreria-component.css'],
})
export class LibreriaComponent {
  @ViewChild("cajatitulo") cajaTitulo!: ElementRef;
  @ViewChild("cajadcripcion") cajaDescripcion!: ElementRef;
  @ViewChild("cajaImagen") cajaImagen!: ElementRef;
  public comics: Array<Comic>;
  public comicFavorito!: Comic;
  createComic(): void {
    let titulo = this.cajaTitulo.nativeElement.value;
    let descripcion = this.cajaDescripcion.nativeElement.value;
    let imagen = this.cajaImagen.nativeElement.value;
    let comicNew = new Comic(titulo, imagen, descripcion);
    this.comics.push(comicNew);
  }
  seleccionarFavorito(favorito: Comic): void{
    this.comicFavorito = favorito;
  }
  deleteComic(index: number){
    this.comics.splice(index, 1);
  }
  constructor() {
    this.comics = [
      new Comic(
        "Spiderman",
        "https://images-na.ssl-images-amazon.com/images/I/61AYfL5069L.jpg",
        "Hombre araña"
      ),
      new Comic(
        "Wolverine",
        "https://i.etsystatic.com/9340224/r/il/42f0e1/1667448004/il_570xN.1667448004_sqy0.jpg",
        "Lobezno"
      ),
      new Comic(
        "Guardianes de la Galaxia",
        "https://lanocheamericana.net/wp-content/uploads/2021/11/guardianes-de-la-galaxia-15-90-panini.jpg",
        "Yo soy Groot"
      ),
      new Comic(
        "Avengers",
        "https://d26lpennugtm8s.cloudfront.net/stores/057/977/products/ma_avengers_01_01-891178138c020318f315132687055371-640-0.jpg",
        "Los Vengadores"
      ),
      new Comic(
        "Spawn",
        "https://i.pinimg.com/originals/e1/d8/ff/e1d8ff4aeab5e567798635008fe98ee1.png
      ",
      "Todd MacFarlane"
    );
  }
}

```

LIBRERIA.COMPONENT.HTML

```

<div>
  <h1>Librería comics</h1>

```

```

<form #comicForm="ngForm">
  <label>Título</label>
  <input type="text" name="cajatitulo" #cajatitulo/><br/>
  <label>Descripción</label>
  <input type="text" name="cajadcripcion" #cajadcripcion/><br/>
  <label>Imagen</label>
  <input type="text" name="cajaimagen" #cajaimagen/><br/>
  <button (click)="createComic()">
    Nuevo comic
  </button>
</form>
@if (comicFavorito){
  <div style="background-color: lightcyan">
    <h2 style="color:fuchsia">{{comicFavorito.titulo}}</h2>
    <h3 style="color:blueviolet">{{comicFavorito.descripcion}}</h3>
    
  </div>
}
@for (c of comics; track c; let i = $index){
  <app-comic [comic]="c" [index]="i"
    (seleccionarFavorito)="seleccionarFavorito($event)"
    (deleteComic)="deleteComic($event)"></app-comic>
}
</div>

```

COMIC.COMPONENT.TS

```

import { Component, Input, Output, EventEmitter } from '@angular/core';
import { Comic } from '../../../../../models/comic';
@Component({
  selector: 'app-comic',
  standalone: false,
  templateUrl: './comic-component.html',
  styleUrls: ['./comic-component.css'],
})
export class ComicComponent {
  @Input() comic!: Comic;
  @Input() index!: number;
  @Output() seleccionarFavorito: EventEmitter<any> = new EventEmitter<any>();
  @Output() deleteComic: EventEmitter<any> = new EventEmitter<any>();
  eliminarComic(): void {
    this.deleteComic.emit(this.index);
  }
  marcarFavorito(): void {
    this.seleccionarFavorito.emit(this.comic);
  }
}

```

COMIC.COMPONENT.HTML

```

<div style="float:left">
  <h2 style="color:blue">{{comic.titulo}}</h2>
  <h3 style="color:red">{{ comic.descripcion}}</h3>
  <br/>
  <button (click)="marcarFavorito()">
    Favorito
  </button>
  <button (click)="eliminarComic()" style="background-color: red">
    Delete
  </button>
</div>

```

SERVICIOS EN ANGULAR

Un servicio es una clase que contiene tanto métodos de acción como métodos para recuperar datos.

Se utilizan de forma distinta a como hemos visto en VUE.

El servicio debe ser inyectado, mediante **IoC** en nuestro **module** en la zona de **providers**

Los servicios en angular se denominan con la siguiente sintaxis: **service.descripcion.ts**

Trabajaremos dentro de **src/app** y una carpeta llamada **services**

No podemos utilizar una clase service tan alegramente, tiene que ser una clase "especial" para poder inyectarla y utilizarla dentro de **providers**, es decir, debe ser una clase **injectable**

```

@Injectable
export class ServicePrueba {
}

```

Una vez decorada la clase y con sus métodos, debemos utilizar la clase a través del propio component (aislado) o desde providers (module)

Para probar esta nueva funcionalidad vamos a crear un servicio que nos devolverá los comics mediante Un método llamado **getComics()**

Sobre `src/app` creamos una nueva carpeta llamada `services` y una clase llamada `service.comics.ts`

SERVICE.COMICS.TS

```
import { Injectable } from '@angular/core';
import { Comic } from '../models/comic';
@Injectable()
export class ServiceComics {
  getComics(): any {
    let comics = new Array<Comic>();
    comics = [
      new Comic(
        "Spiderman",
        "https://images-na.ssl-images-amazon.com/images/I/61AYfL5069L.jpg",
        "Hombre araña"
      ),
      new Comic(
        "Wolverine",
        "https://i.etsystatic.com/9340224/r/il/42f0e1/1667448004/il_570xN.1667448004_sqy0.jpg",
        "Lobezno"
      ),
      new Comic(
        "Guardianes de la Galaxia",
        "https://lanocheamericana.net/wp-content/uploads/2021/11/guardianes-de-la-galaxia-15-90-panini.jpg",
        "Yo soy Groot"
      ),
      new Comic(
        "Avengers",
        "https://d261penugtm8s.cloudfront.net/stores/057/977/products/ma_avengers_01_01-891178138c020318f315132687055371-640-0.jpg",
        "Los Vengadores"
      ),
      new Comic(
        "Spawn",
        "https://i.pinimg.com/originals/e1/d8/ff/e1d8ff4aeab5e567798635008fe98ee1.png"
      ),
      new Comic(
        "Todd MacFarlane"
      )
    ];
    return comics;
  }
}
```

Para poder recuperar los comics del servicio, necesitamos `providers`

Vamos a probar primero a recuperar los comics desde el propio component de librería.

Nota: En el component, para poder recuperar objetos desde `inyección`, necesitamos declararlos

Dentro de `providers` del propio component

Los objetos NO se utilizan desde el constructor, siempre debemos utilizarlos desde `ngOnInit`

LIBRERIA.COMPONENT.TS

```
import { Component, ViewChild, ElementRef, OnInit } from '@angular/core';
import { Comic } from '../models/comic';
import { ServiceComics } from '../services/service.comics';
@Component({
  selector: 'app-libreria',
  standalone: false,
  templateUrl: './libreria-component.html',
  styleUrls: ['./libreria-component.css'],
  //DEBEMOS DECLARAR EL SERVICIO PARA PODER RECUPERARLO
  //DENTRO DE UN CONSTRUCTOR
  providers: [ServiceComics]
})
export class LibreriaComponent implements OnInit{
  @ViewChild("cajatitulo") cajaTitulo!: ElementRef;
  @ViewChild("cajadcripcion") cajaDescripcion!: ElementRef;
  @ViewChild("cajaImagen") cajaImagen!: ElementRef;
  public comics!: Array<Comic>;
  public comicFavorito!: Comic;
  constructor(private _service: ServiceComics) {}
  ngOnInit(): void {
    this.comics = this._service.getComics();
  }
  createComic(): void {
    let titulo = this.cajaTitulo.nativeElement.value;
    let descripcion = this.cajaDescripcion.nativeElement.value;
    let imagen = this.cajaImagen.nativeElement.value;
    let comicNew = new Comic(titulo, imagen, descripcion);
    this.comics.push(comicNew);
  }
  seleccionarFavorito(favorito: Comic): void{
    this.comicFavorito = favorito;
  }
}
```

```
    deleteComic(index: number){
      this.comics.splice(index, 1);
    }
}
```

Tal y como lo hemos planteado es como se utiliza si hubieramos creado nuestro proyecto como **standalone=true**, es decir, sin **module/jefe**

Vamos a centralizar la inversión de control y la inyección mediante **module**

Realizamos la inversión de control del Service en un único lugar (container) y lo inyectamos en los Components/Clases que necesitemos

APP MODULE.TS

```
import { ServiceComics } from './services/service.comics';

@NgModule({
  declarations: [
    App,
    ListaProductosComponent,
    MenuComponent,
    HijoCochesComponent,
    PadreCochesComponent,
    PadreDeportesComponent,
    HijoDeporteComponent,
    LibreriaComponent,
    ComicComponent
  ],
  imports: [
    BrowserModule, routing, FormsModule
  ],
  providers: [
    provideBrowserGlobalErrorListeners(), appRoutingProvider, ServiceComics
  ],
})
```

BOOTSTRAP ANGULAR

Vamos a utilizarlo mediante CDN igual que hicimos con VUE.

También podemos hacerlo mediante **npm**, pero es más rollo y os buscáis la vida si queréis hacerlo así.

The screenshot shows a web page titled "Include via CDN". It contains two code snippets:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8...">
```

```
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8...">
```

Incluimos los Links dentro de **src/index.html**

```
<!doctype html>
<html lang="en">
```

```

<head>
  <meta charset="utf-8">
  <title>Angular comunicacion modelos</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
  <app-root></app-root>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/js/bootstrap.bundle.min.js"></script>
</body>
</html>

```

SERVICIOS API ANGULAR

Dentro de Angular, por supuesto, podemos utilizar **axios**, simplemente lo instalamos mediante **npm** y lo importamos.

En este Framework tenemos librerías de peticiones integradas dentro del propio Angular.

Dichas librerías con peticiones Api, debemos utilizarlas con su declaración en Module o en Component

La librería para realizar peticiones Api se llama **provideHttpClient** de **@angular/common/http**

Dicha librería, para poder utilizarla, debemos tenerla en la zona de **providers**

```

providers: [
  provideHttpClient(),
  provideBrowserGlobalErrorListeners(),
  appRoutingProvider,
  ServiceComics
],

```

Para poder realizar peticiones Api se realiza mediante Promesas.

Tenemos dos formas de plantear peticiones dentro de un servicio si realizamos un GET.

1. Devolver directamente los datos una vez leídos
2. Devolver la propia promesa para que el componente se suscriba

Creamos un nuevo proyecto con routing y standalone a false llamado **serviciosangular**.

`ng new serviciosangular --standalone=false`

Vamos a leer el siguiente servicio simple:

<https://servicioapipersonasmvcpgs.azurewebsites.net/api/personas>

```

[
  {
    "IdPersona": 1,
    "Nombre": "Lucia",
    "Email": "lucia@gmail.com",
    "Edad": 19
  },
  {
    "IdPersona": 2,
    "Nombre": "Adrian",
    "Email": "adrian@gmail.com",
    "Edad": 24
  }
]

```

Api Personas

Lucia, lucia@gmail.com

Adrian, adrian@gmail.com

Adrián, adrian@gmail.com

Alejandro, alejandro@gmail.com

Sara, sara@gmail.com

Comenzamos sobre **src/app** creando una nueva carpeta llamada **models** y una clase llamada **persona.ts**

PERSONA.TS

```
export class Persona {
    //PARA REALIZAR EL BINDING AUTOMATICO
    //LAS PROPIEDADES DEBEN LLAMARSE COMO EL JSON
    constructor(
        public IdPersona: number,
        public Nombre: string,
        public Email: string,
        public Edad: number
    ){}
}
```

En nuestro **Module** agregamos, dentro de **providers** el proveedor **provideHttpClient()**

APP MODULE.TS

```
@NgModule({
  declarations: [
    App
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [
    provideHttpClient(),
    provideBrowserGlobalErrorListeners()
  ],
  bootstrap: [App]
})
export class AppModule { }
```

Sobre **services** creamos un nuevo servicio llamado **service.personas.ts**

SERVICE.PERSONAS.TS

```
import { Injectable } from "@angular/core";
import { Persona } from "../models/persona";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
@Injectable()
export class ServicePersonas {
    //PARA PODER REALIZAR PETICIONES, NECESITAMOS EL OBJETO
    //HttpClient.
    //Dicho objeto debemos inyectarlo en las clases que utilicemos con APIs
    constructor(private _http: HttpClient){}
    //SI VAMOS A DEVOLVER LA PETICION, EL OBJETO A DEVOLVER ES
    //UN Observable<any> PARA PODER SUSCRIBIRNOS
    getPersonas(): Observable<any> {
        let urlApi = "https://servicioapipersonasmvcpgs.azurewebsites.net/";
        let request = "api/personas"
        return this._http.get(urlApi + request);
    }
}
```

```
}
```

Inyectamos el servicio dentro de **Module**

```
import { ServicePersonas } from './services/service.personas';
@NgModule({
  declarations: [
    App,
    PersonasApiComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [
    ServicePersonas,
    provideHttpClient(),
    provideBrowserGlobalErrorListeners()
  ],
})
```

Creamos un nuevo componente, en la carpeta **components**, llamado **personasapi.component.ts**

PERSONASAPI.COMPONENT.TS

```
import { Component, OnInit } from '@angular/core';
import { ServicePersonas } from '../../../../../services/service.personas';
import { Persona } from '../../../../../models/persona';
@Component({
  selector: 'app-personas-api',
  standalone: false,
  templateUrl: './personas-api-component.html',
  styleUrls: ['./personas-api-component.css'],
})
export class PersonasApiComponent implements OnInit {
  public personas!: Array<Persona>;
  constructor(private _service: ServicePersonas){}
  ngOnInit(): void {
    this._service.getPersonas().subscribe(response => {
      console.log("leyendo")
      this.personas = response;
    })
  }
}
```

PERSONASAPI.COMPONENT.HTML

```
<div>
  <h1 style="color:blue">Api Personas</h1>
  <ul class="list-group" *ngIf="personas.length > 0">
    <li *ngFor="let person of personas" class="list-group-item">
      {{person.Nombre}}, {{person.Email}}
    </li>
  </ul>
</div>
```

Otra opción que tenemos en el Servicio es poder devolver directamente una Promesa (objeto).

SERVICE.PERSONAS.TS

```
getPersonasPromise(): Promise<any> {
  let urlApi = "https://servicioapipersonasmvcpgs.azurewebsites.net/";
  let request = "api/personas"
  let promise = new Promise((resolve) => {
    this._http.get(urlApi + request).subscribe(response => {
      resolve(response)
    })
  })
  return promise;
}
```

Modificamos la petición de nuestro componente.

```
ngOnInit(): void {
  this._service.getPersonasPromise().then(response => {
    console.log("Reading")
    this.personas = response;
  })
}
```

El siguiente paso que vamos a realizar es hacer un componente aislado (standalone=true)
Lo que quiere decir esto es que tengo que realizar todo dentro del componente, por ejemplo, los Providers o Routing

La idea de los componentes aislados radica en múltiples proyectos.
Actualmente, si nos llevamos nuestro componente de Personas API a otro proyecto, no funciona.

Si tenemos un componente aislado, simplemente lo migramos a otro proyecto y es funcional.

Creamos un nuevo componente llamado **personasstandalone**

Vamos a copiar código y demás a ver cómo podemos hacerlo funcionar...

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

@Component({
  selector: 'app-personas-standalone',
  standalone: true,
  templateUrl: './personas-standalone-component.html',
  styleUrls: ['./personas-standalone-component.css'],
})
export class PersonasStandaloneComponent {
```

Dibujamos dentro de App nuestro componente

APP.COMPONENT.HTML

```
<div style="width: 50%; margin: auto">
  <app-personas-standalone></app-personas-standalone>
  <app-personas-api></app-personas-api>
</div>
```

Lo hemos creado con el maravilloso comando **ng g component** y nos lo crea **standalone=false** y, además,

Lo dibuja dentro de **declarations** y **Module**

```
import { PersonasStandaloneComponent } from './componentes/personas-standalone';
@NgModule({
  declarations: [
    App,
    PersonasApiComponent,
    PersonasStandaloneComponent
  ],
  imports: [
```

No podemos dibujar un componente STANDALONE=true en un proyecto con MODULO directamente
En **declarations**, debe ir como **imports**

```
imports: [
  BrowserModule,
  AppRoutingModule,
  PersonasStandaloneComponent
```

Una vez que hemos comprobado que lo está dibujando, vamos a intentar hacer una réplica del Component del Service Api Personas

Copiamos la declaración del Array de Personas y el código HTML.

```
export class PersonasStandaloneComponent {
  public personas!: Array<Persona>;
  constructor() {
    this.personas = new Array<Persona>();
    let p1 = new Persona(1, "A", "A", 22);
    this.personas.push(p1);
    let p2 = new Persona(2, "B", "B", 32);
    this.personas.push(p2);
  }
}
```

Podremos comprobar que NO muestra los datos.

```
src/app/components/personas-standalone-component/personas-standalone-c
omponent.html:4:13:
  4 |   {#personas *ngFor="let person of personas" class="list-group-it
em">
  |   </li>
  |   ~~~~~
  |
  | Warning occurs in the template of component PersonasStandaloneComponent.
```

Nos está dando error porque este componente NO entiende nada, está solito, es decir, que NO tiene Ningún import ni información propia.

Las directivas NgFor y NgIf deben ser declaradas dentro de un módulo o component.

```
@Component({
  selector: 'app-personas-standalone',
  standalone: true,
  templateUrl: './personas-standalone-component.html',
  styleUrls: ['./personas-standalone-component.css'],
  imports: [NgFor, NgIf]
})
```

Si utilizamos el lenguaje @for, no es necesario utilizar import para trabajar

```
<div>
  <h1 style="color: red">Api Personas StandAlone!!!</h1>
  @if (personas) {
    <ul class="list-group">
      @for (person of personas; track person){
        <li class="list-group-item">
          {{person.Nombre}}, {{person.Email}}
        </li>
      }
    </ul>
  }
</div>
```

Por último, implementamos el Servicio en este componente aislado mediante **providers** y copiamos el código
Del component anterior.

```
@Component({
  selector: 'app-personas-standalone',
  standalone: true,
  templateUrl: './personas-standalone-component.html',
  styleUrls: ['./personas-standalone-component.css'],
  providers: [ServicePersonas]
})
```

PERSONASSTANDALONE.COMPONENT.TS

```
import { Component, OnInit } from '@angular/core';
import { Persona } from '../../models/persona';
import { ServicePersonas } from '../../services/service.personas';
@Component({
  selector: 'app-personas-standalone',
  standalone: true,
  templateUrl: './personas-standalone-component.html',
  styleUrls: ['./personas-standalone-component.css'],
  providers: [ServicePersonas]
})
export class PersonasStandaloneComponent implements OnInit {
  public personas!: Array<Persona>;
  constructor(private _service: ServicePersonas){}
  ngOnInit(): void {
    this._service.getPersonasPromise().then(response => {
      console.log("Reading")
      this.personas = response;
    })
  }
}
```

El siguiente concepto es utilizar, por ahora, Global para centralizar nuestrasUrls de los Apis.

En Angular, tenemos dos opciones:

1. **Global:** Una clase de declaración de variables tal y como hemos hecho en el resto de Frameworks

Sobre src/app creamos una nueva clase llamado global.ts

```
export var Global = {
  urlApiPersonas: "https://servicioapipersonasmvcpgs.azurewebsites.net/"}
```

Dentro de nuestro Service, utilizamos Global

SERVICE.PERSONAS.TS

```
import { Global } from "../Global";
@.Injectable()
export class ServicePersonas {

  getPersonasPromise(): Promise<any> {
    let urlApi = Global.urlApiPersonas;
    let request = "api/personas"
    let promise = new Promise((resolve) => {
      this._http.get(urlApi + request).subscribe(response => {
        resolve(response)
      })
    })
    return promise;
  }
}
```

2. **Utilizar variables de entorno de Angular.** Dichas variables se llaman **environments** y es lo mismo que

Utilizar Global, pero enfocado solamente al lenguaje Angular
 Existe una gran ventaja al utilizar este tipo de arquitectura que es llevar el entorno a producción sin
 Tener que hacer nada.

Cuando hablamos de entornos, estamos hablando de dos tipos:

1. Developer: En el entorno de desarrollo NO se utilizan elementos de producción, ya que nos Podemos cargar todo haciendo pruebas, lo que es un problema.
 Se suelen utilizar recursos, apis, bbdd, de testing para las pruebas.
2. Production: Una vez que tenemos un desarrollo finalizado para publicar, dichos recursos son Modificados para utilizar los reales.

Si utilizamos variables de entorno dentro de Angular en lugar de Global, se generan dos ficheros de **Environment**, uno que es para Developer y otro para producción.

En los dos ficheros debemos tener las mismas Keys y distintos values.

Automáticamente, cuando estamos en producción, el propio Angular, recupera el fichero **environment** de Producción.

Debemos generar los ficheros para el entorno.

ng g environments

```
C:\Users\Profesor MCSD Mañana\Documents\FRONT\ANGULAR\serviciosangular>ng g environments
CREATE src/environments/environment.ts (31 bytes)
CREATE src/environments/environment.development.ts (31 bytes)
UPDATE angular.json (3012 bytes)
```

Nos ha creado dos ficheros:

1. Producción: **TS**
2. Desarrollo: **Development**

Escribimos las variables de Api dentro de nuestros ficheros.

```
export const environment = {
  urlApiPersonas: "https://servicioapipersonasmvcpgs.azurewebsites.net/"
};
```

Por último, simplemente utilizamos el nuevo environment dentro del Servicio Personas

SERVICE.PERSONAS.TS

```
import { environment } from "../../environments/environment.development";

getPersonasPromise(): Promise<any> {
  let urlApi = environment.urlApiPersonas;
  let request = "api/personas"
  let promise = new Promise((resolve) => {
    this._http.get(urlApi + request).subscribe(response => {
      resolve(response)
    })
  })
  return promise;
}
```

Y dentro de **angular.json** tenemos la información si llevamos a producción nuestro proyecto

```
"development": {
  "optimization": false,
  "extractLicenses": false,
  "sourceMap": true,
  "fileReplacements": [
    {
      "replace": "src/environments/environment.ts",
      "with": "src/environments/environment.development.ts"
    }
  ]
},
```

```

        "replace": "src/environments/environment.ts",
        "with": "src/environments/environment.development.ts"
    }
]
}

```

Necesito lo siguiente:

Creamos un nuevo proyecto llamado **angularviernes**.

Dicho proyecto es de investigación.

1. Quiero que sea **standalone=true**
2. Quiero incluir **routing**, pero el que viene de serie

Con estas dos instrucciones, necesito un proyecto con un Link de Home y un link para visualizar las personas

Con Models, Service y con Environments

Versión 2:

Quiero probar consumir el Servicio Api de Personas con dos opciones nuevas:

1. Axios.
2. Fetch (No es la librería de VUE, no tenemos que instalar nada)

Vamos a realizar un nuevo servicio consumiendo Coches en el siguiente Api

<https://apicochespaco.azurewebsites.net/>

Para visualizar cómo funciona, voy a realizarlo con **fetch**

Introducimos la Url dentro de Environment

Creamos un nuevo Modelo llamado **Coche**

```

export class Coche{
    constructor(
        public idcoche: number,
        public marca: string,
        public modelo: string,
        public conductor: string,
        public imagen: string
    ){}
}

```

Sobre **services**, creamos un nuevo servicio llamado **service.coches.ts**

SERVICE.COCHES.TS

```

import { Coche } from "../models/coche";
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { environment } from "../../environments/environment.development";
import { Observable } from "rxjs";
@Injectable()
export class ServiceCoches {
    constructor(private _http: HttpClient){}
    getCochesHttpClient(): Observable<any> {
        let request = "webresources/coches";
        let url = environment + request;
        return this._http.get(url);
    }
    getCochesPromise(): Promise<any> {
        let request = "webresources/coches";
        let url = environment + request;
        let promise = new Promise((resolve) => {
            fetch(url).then(response => {
                resolve(response);
            })
        })
        return promise;
    }
    getCoches(): any {
        let request = "webresources/coches";
    }
}

```

```

        let url = environment + request;
        fetch(url).then(response => {
            return response;
        })
    }
}

```

Inyectamos nuestro servicio dentro de module

APP MODULE.TS

```

providers: [
    ServiceCoches,
    ServicePersonas,
    provideHttpClient(),
    provideBrowserGlobalErrorListeners()
],
bootstrap: [App]

```

Creamos un nuevo component llamado **coches.component.ts**

COCHES.COMPONENT.TS

```

import { Component, OnInit } from '@angular/core';
import { ServiceCoches } from '../../../../../services/service.coches';
import { Coche } from '../../../../../models/coche';
@Component({
    selector: 'app-coches',
    standalone: false,
    templateUrl: './coches-component.html',
    styleUrls: ['./coches-component.css'],
})
export class CochesComponent implements OnInit {
    public coches!: Array<Coche>;
    constructor(private _service: ServiceCoches){}
    ngOnInit(): void {
        this._service.getCochesPromise().then(response => {
            this.coches = response;
        })
        // this.coches = this._service.getCoches();
        // this._service.getCoches().then(response => {
        //     this.coches = response;
        // })
    }
}

```

COCHES.COMPONENT.HTML

```

<div>
    <h1 style="color:blue">Api Coches Fetch</h1>
    @for (car of coches; track car){
        <div class="card" style="width: 18rem;">
            
            <div class="card-body">
                <h5 class="card-title">{{car.marca}} {{car.modelo}}</h5>
                <p class="card-text">{{car.conductor}}</p>
            </div>
        </div>
    }
</div>

```

Hemos utilizado el siguiente método:

```

getCochesPromise(): Promise<any> {
    let request = "webresources/coches";
    let url = environment.urlApiCoches + request;
    console.log(url);
    let promise = new Promise((resolve) => {
        fetch(url).then(response => {
            resolve(response.json());
        })
    })
    return promise;
}

```

```

this._service.getCochesPromise().then(response => {
    this.coches = response;
})

```

Desventajas:

1. Mucho código en comparación con HttpClient

```
getCochesHttpClient(): Observable<any> {
  let request = "webresources/coches";
  let url = environment.urlApiCoches + request;
  return this._http.get(url);
}

this._service.getCochesHttpClient().subscribe(response => {
  this.coches = response;
})
```

2. ¿Cuál es la principal diferencia/fortaleza de Angular respecto al resto? **Típado**
Si devolvemos cualquier COSA, pues estamos sometidos a errores de cualquier cosa.

```
getCoches(): Promise<Array<Coche>> {
  let request = "webresources/coches";
  let url = environment.urlApiCoches + request;
  //EXTRAER LOS DATOS TENEMOS QUE SEGUIR HACIENDOLO
  //LA DIFERENCIA ESTA EN LA SINTAXIS
  const coches =
    fetch(url).then(response => response.json());
  return coches;
}
```

En el siguiente ejemplo, tendremos dos versiones:

1. Versión simple mostrando los datos de la plantilla por su función
2. Versión múltiple, mostrando los datos de la plantilla por múltiples funciones

Llamaremos al siguiente servicio Api.

<https://apiplantillacore.azurewebsites.net/>

Api Plantilla Función simple

Enfermero

Mostrar plantilla

Id	Hospital	Sala	Apellido	Función	Turno	Salario
3106	19	6	Hernández J.	Enfermero	T	575570
8422	22	6	Bocina G.	Enfermero	M	183547

Necesitamos un componente llamado **plantillafuncionsimple**

Un Modelo **plantilla** para capturar los datos

PLANTILLA.TS

```
export class Plantilla {
  constructor(
    public idEmpleado: number,
    public idHospital: number,
    public idSala: number,
    public apellido: string,
    public funcion: string,
    public turno: string,
    public salario: number
 ){}
}
```

```
}
```

Un Servicio llamado **service.plantillas.ts** para las peticiones.

SERVICE.PLANTILLAS.TS

```
import { Plantilla } from "../models/plantilla";
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { environment } from "../../environments/environment.development";
@Injectable()
export class ServicePlantilla {
  constructor(private _http: HttpClient){}
  getFunciones(): Observable<Array<string>> {
    let request = "api/plantilla/funciones";
    let url = environment.urlApiPlantilla + request;
    //DENTRO DE HTTPGET PODEMOS INDICAR EL TIPO DE OBJETO A DEVOLVER
    //MEDIANTE DIAMONDS <T>
    return this._http.get<Array<string>>(url);
  }
  getPlantillaFuncion(funcion: string): Promise<Array<Plantilla>> {
    let request = "api/plantilla/plantillafuncion/" + funcion;
    let url = environment.urlApiPlantilla + request;
    const plantillas = fetch(url).then(response => response.json());
    return plantillas;
  }
}
```

Damos de alta a **ServicePlantilla** y **FormsModule** dentro de **module**

APP MODULE.TS

```
imports: [
  FormsModule,
  BrowserModule,
  AppRoutingModule,
  PersonasStandaloneComponent
],
providers: [
  ServicePlantilla,
  ServiceCoches,
  ServicePersonas,
  provideHttpClient(),
  provideBrowserGlobalErrorListeners()
],
```

Creamos un menú e implementamos **Router** para mostrar los datos

El diseño será un desplegable **<select>**, un botón para lanzar la acción y una tabla para Dibujar los datos de la plantilla.

PLANTILLAFUNCSIMPLE.COMPONENT.TS

```
import { Component, OnInit } from '@angular/core';
import { ElementRef, ViewChild } from '@angular/core';
import { Plantilla } from '../../models/plantilla';
import { ServicePlantilla } from '../../services/service.plantillas';
@Component({
  selector: 'app-plantilla-simple',
  standalone: false,
  templateUrl: './plantilla-simple-component.html',
  styleUrls: ['./plantilla-simple-component.css'],
})
export class PlantillaSimpleComponent implements OnInit {
  public funciones!: Array<string>;
  @ViewChild("selectFuncion") selectFuncion!: ElementRef;
  public plantillas: Array<Plantilla>;
  constructor(private _service: ServicePlantilla){
    this.plantillas = new Array<Plantilla>();
  }
  mostrarPlantilla(): void {
    let funcion = this.selectFuncion.nativeElement.value;
    this._service.getPlantillaFuncion(funcion).then(response => {
      this.plantillas = response;
    })
  }
}
```

```

ngOnInit(): void {
    this._service.getFunciones().subscribe(response => {
        this.funciones = response;
    })
}
}

```

PLANTILLAFUNCIONSIMPLE.COMPONENT.HTML

```

<div>
    <h1 style="color:red">Api Plantilla Función simple</h1>
    <form #plantillaForm="ngForm">
        <select name="selectfuncion" #selectfuncion class="form-control">
            @for (funcion of funciones; track funcion){ 
                <option>{{funcion}}</option>
            }
        </select>
        <button class="btn btn-outline-secondary"
        (click)="mostrarPlantilla()">
            Mostrar plantilla
        </button>
    </form>
    @if (plantillas.length > 0){
        <table class="table table-bordered">
            <thead>
                <tr>
                    <th>Id</th>
                    <th>Hospital</th>
                    <th>Sala</th>
                    <th>Apellido</th>
                    <th>Función</th>
                    <th>Turno</th>
                    <th>Salario</th>
                </tr>
            </thead>
            <tbody>
                @for (empleado of plantillas; track empleado){
                    <tr>
                        <td>{{empleado.idEmpleado}}</td>
                        <td>{{empleado.idHospital}}</td>
                        <td>{{empleado.idSala}}</td>
                        <td>{{empleado.apellido}}</td>
                        <td>{{empleado.funcion}}</td>
                        <td>{{empleado.turno}}</td>
                        <td>{{empleado.salario}}</td>
                    </tr>
                }
            </tbody>
        </table>
    }
</div>

```

El siguiente paso es realizar la versión 2 del ejemplo.

Vamos a realizar lo mismo, pero con petición múltiple al Api.

Api Plantilla Función múltiple

Enfermera

Enfermero

Interino

Mostrar plantilla

Id	Hospital	Sala	Apellido	Función	Turno	Salario
1009	22	6	Higueras D.	Enfermera	T	200247
1280	45	4	Amigo R.	Interino	N	9221341
3754	19	6	Díaz B.	Enfermera	T	526770

La petición para trabajar es la siguiente:

<https://apiplantillacore.azurewebsites.net/api/plantilla/plantillafunciones?funcion=Enfermero&funcion=Enfermera>

GET /api/Plantilla/PlantillaFunciones

Creamos otro component llamado **plantillafuncionmultiple**

Copiamos el diseño del component plantillasimple a plantillamultiple y cambiamos "ALGO"

Como hicimos en React, debemos tener un método en el servicio que recibirá los datos
Para la petición.

Vamos a tener un método que recibirá un Array de string con las funciones seleccionadas
y, posteriormente, en dicho método, con un bucle, hacemos la petición.

SERVICE.PLANTILLAS.TS

```
getPlantillaFunciones(funciones: Array<string>): Observable<Array<Plantilla>> {
    //?funcion=Enfermero&funcion=Enfermera
    let datos = "";
    for (var funcion of funciones){
        datos += "funcion=" + funcion + "&";
    }
    //?funcion=Enfermero&funcion=Enfermera&
    //ELIMINAMOS EL ULTIMO CARACTER &
    datos = datos.substring(0, datos.length - 1);
    let request = "api/plantilla/plantillafunciones?" + datos;
    let url = environment.urlApiPlantilla + request;
    return this._http.get<Array<Plantilla>>(url);
}
```

Modificamos el código del HTML del component múltiple

```
<h1 style="color: #blue">Api Plantilla Función múltiple</h1>
<form #plantillaForm="ngForm">
    <select name="selectfunciones" #selectfunciones
            class="form-control"
            multiple size="5">
        @for (funcion of funciones; track funcion){
            <option>{{funcion}}</option>
        }
    </select>
    <button class="btn btn-info" (click)="mostrarPlantilla()">
        Mostrar plantilla
    </button>
</form>
```

PLANTILLAMULTIPLE.COMPONENT.TS

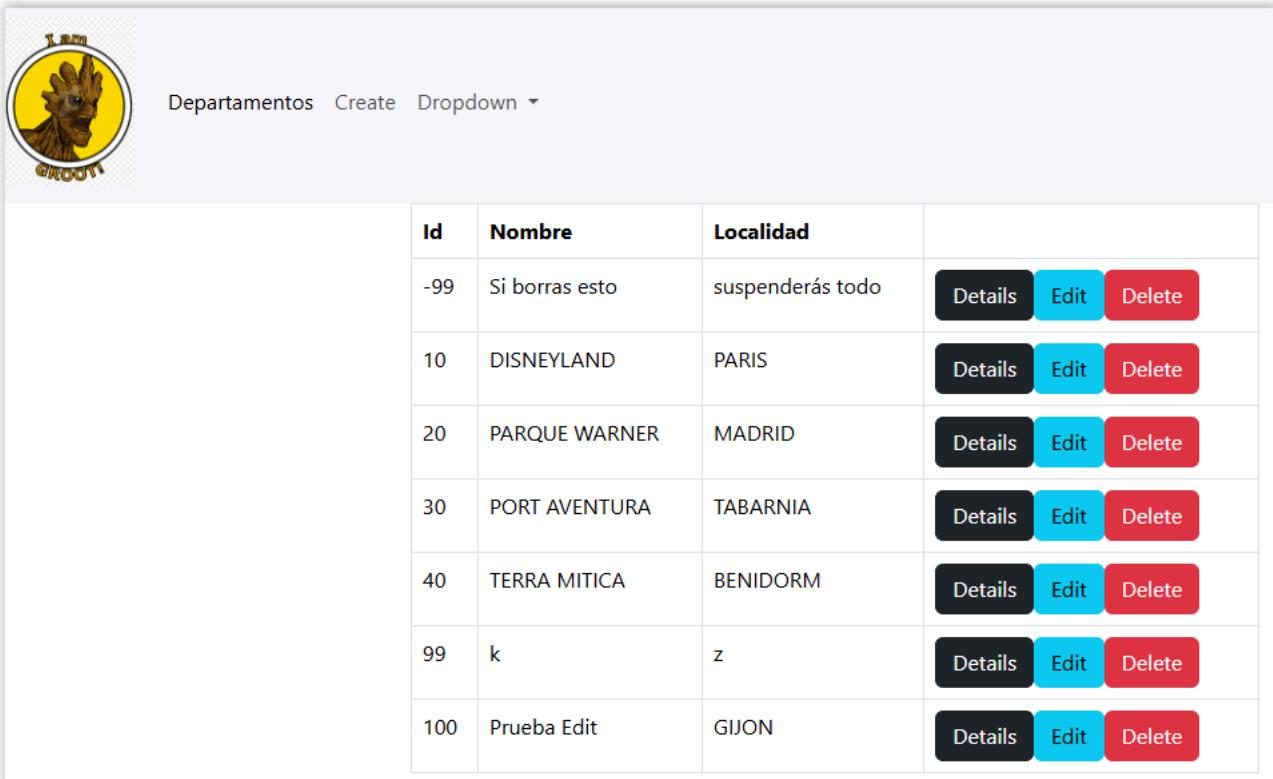
```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { Plantilla } from '../../../../../models/plantilla';
import { ServicePlantilla } from '../../../../../services/service.plantillas';
@Component({
    selector: 'app-plantilla-multiple',
    standalone: false,
    templateUrl: './plantilla-multiple-component.html',
    styleUrls: ['./plantilla-multiple-component.css'],
})
export class PlantillaMultipleComponent implements OnInit {
    public funciones!: Array<string>;
    public plantillas: Array<Plantilla>;
    @ViewChild("selectfunciones") selectFunciones!: ElementRef;
    public funcionesSeleccionadas: Array<string>;
    constructor(private _service: ServicePlantilla){
        this.plantillas = new Array<Plantilla>();
        this.funcionesSeleccionadas = new Array<string>();
    }
    mostrarPlantilla(): void {
        let aux = new Array<string>();
        for (var option of this.selectFunciones.nativeElement.options){
            if (option.selected == true){
                aux.push(option.value);
            }
        }
        this.funcionesSeleccionadas = aux;
    }
    this._service.getPlantillaFunciones(this.funcionesSeleccionadas).subscribe
        (response => {
            this.plantillas = response;
        })
}
ngOnInit(): void {
```

```

    this._service.getFunciones().subscribe(response => {
      this.funciones = response;
    })
}
}

```

Vamos a realizar el clásico del CRUD de departamentos.



	Id	Nombre	Localidad	
	-99	Si borras esto	suspenderás todo	<button>Details</button> <button>Edit</button> <button>Delete</button>
	10	DISNEYLAND	PARIS	<button>Details</button> <button>Edit</button> <button>Delete</button>
	20	PARQUE WARNER	MADRID	<button>Details</button> <button>Edit</button> <button>Delete</button>
	30	PORT AVENTURA	TABARNIA	<button>Details</button> <button>Edit</button> <button>Delete</button>
	40	TERRA MITICA	BENIDORM	<button>Details</button> <button>Edit</button> <button>Delete</button>
	99	k	z	<button>Details</button> <button>Edit</button> <button>Delete</button>
	100	Prueba Edit	GIJON	<button>Details</button> <button>Edit</button> <button>Delete</button>

Utilizamos el siguiente servicio Api.

<https://apicruddepartamentoscore.azurewebsites.net/index.html>

Creamos un nuevo proyecto llamado **angularcruddepartamentos**

Creamos los environments: **ng g environments**

```

export const environment = {
  urlApiDepartamentos: "https://apicruddepartamentoscore.azurewebsites.net/"
};

```

Agregamos los links de Bootstrap sobre **index.html**

Creamos un component **DepartamentosComponent** y otro **MenuComponent**

Montamos Router dentro de nuestra App.

Abrimos **Module** y habilitamos Formularios y Servicios

APP MODULE TS

```

@NgModule({
  declarations: [
    App,
    MenuComponent,
    DepartamentosComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [
    ...
  ]
})

```

```

    provideHttpClient(),
    provideBrowserGlobalErrorListeners()
],
bootstrap: [App]
)

```

Sobre src/app creamos una nueva carpeta llamada **models** y una clase llamada **departamento**

DEPARTAMENTO.TS

```

export class Departamento {
  constructor(
    public numero: number,
    public nombre: string,
    public Localidad: string
  ) {}
}

```

Sobre src/app creamos una carpeta llamada **services** y una clase llamada **service.departamentos.ts**

SERVICE.DEPARTAMENTOS.TS

```

import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Departamento } from '../models/departamento';
import { environment } from '../../../../../environments/environment.development';
@Injectable()
export default class ServiceDepartamentos {
  constructor (private _http: HttpClient){}
  getDepartamentos(): Observable<Array<Departamento>> {
    let request = "api/departamentos";
    let url = environment.urlApiDepartamentos + request;
    return this._http.get<Array<Departamento>>(url);
  }
}

```

DEPARTAMENTOS.COMPONENT.TS

```

import { Component, OnInit } from '@angular/core';
import ServiceDepartamentos from '../../../../../services/service.departamentos';
import { Departamento } from '../../../../../models/departamento';
@Component({
  selector: 'app-departamentos',
  standalone: false,
  templateUrl: './departamentos-component.html',
  styleUrls: ['./departamentos-component.css'],
})
export class DepartamentosComponent implements OnInit {
  public departamentos!: Array<Departamento>;
  constructor(private _service: ServiceDepartamentos) {}
  ngOnInit(): void {
    this._service.getDepartamentos().subscribe(response => {
      this.departamentos = response;
    })
  }
}

```

DEPARTAMENTOS.COMPONENT.HTML

```

<div>
@if (departamentos.length > 0){
  <table class="table table-bordered">
    <thead>
      <tr>
        <th>Id</th>
        <th>Nombre</th>
        <th>Localidad</th>
      </tr>
    </thead>
    <tbody>
      @for (dept of departamentos; track dept){
        <tr>
          <td>{{dept.numero}}</td>
          <td>{{dept.nombre}}</td>
          <td>{{dept.localidad}}</td>
        </tr>
      }
    </tbody>
  </table>
}

```

```
</table>
}
</div>
```

A continuación, vamos a realizar el POST de insertar.

Comenzamos creando un componente llamado **createcomponent** y lo ponemos a jugar en el menú.

En Angular, para enviar la información con **data**, debemos pasar el objeto a **json** con el método **JSON.stringify()**

Dicha información será enviada con los Headers de la petición, es decir, debemos Utilizar una clase nueva llamada **HttpHeaders** con toda la información que vayamos a enviar.

SERVICE.DEPARTAMENTOS.TS

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

createDepartamento(departamento: Departamento): Observable<any> {
    //IGUAL QUE EN REACT O JQUERY
    let json = JSON.stringify(departamento);
    //CREAMOS LA CABECERA DE LA PETICION
    let header = new HttpHeaders();
    //INDICAMOS EL TIPO DE OBJETO A ENVIAR EN DATA
    header = header.set("Content-type", "application/json");
    let request = "api/departamentos";
    let url = environment.urlApiDepartamentos + request;
    return this._http.post(url, json, {headers: header});
}
```

Diseñamos nuestro componente.

CREATE.COMPONENT.HTML

```
<div>
    <h1>Create departamento</h1>
    <form #departamentoForm="ngForm">
        <label>Id departamento</label>
        <input type="number" name="cajaId" #cajaId class="form-control"/>
        <label>Nombre</label>
        <input type="text" name="cajaNombre" #cajaNombre class="form-control"/>
        <label>Localidad</label>
        <input type="text" name="cajaLocalidad" #cajaLocalidad class="form-control"/>
        <button class="btn btn-warning" (click)="insertDepartamento()">
            Create
        </button>
    </form>
</div>
```

CREATE.COMPONENT.TS

```
import { Component, OnInit, ElementRef, ViewChild } from '@angular/core';
import { Router } from '@angular/router';
import ServiceDepartamentos from '../../../../../services/service.departamentos';
import { Departamento } from '../../../../../models/departamento';
@Component({
    selector: 'app-create',
    standalone: false,
    templateUrl: './create-component.html',
    styleUrls: ['./create-component.css'],
})
export class CreateComponent {
    @ViewChild("cajaId") cajaId!: ElementRef;
    @ViewChild("cajaNombre") cajaNombre!: ElementRef;
    @ViewChild("cajaLocalidad") cajaLocalidad!: ElementRef;
    constructor(
        private _service: ServiceDepartamentos,
        private _router: Router
    ){}
    insertDepartamento(): void {
        let id = parseInt(this.cajaId.nativeElement.value);
        let nombre = this.cajaNombre.nativeElement.value;
        let localidad = this.cajaLocalidad.nativeElement.value;
        let departamento = new Departamento(id, nombre, localidad);
        this._service.createDepartamento(departamento).subscribe(response => {
            console.log("Insertado");
            this._router.navigate(["/"]);
        })
    }
}
```

El siguiente paso es visualizar **Details** de un departamento mediante Routing y parámetros
En la ruta

Creamos un nuevo componente llamado **detailscomponent** y lo ponemos primero en la ruta

```
const routes: Routes = [
  { path: "", component: DepartamentosComponent },
  { path: "create", component: CreateComponent },
  { path: "details/:id/:nombre/:localidad", component: DetailsComponent },
];

```

Posteriormente, sobre (Home) **DepartamentosComponent** debemos enviar los parámetros En nuestra ruta y tenemos dos formas:

1. Concatenando

```
<a [routerLink]="/details/" + dept.numero
+ '/' + dept.nombre + '/' + dept.Localidad"
class="btn btn-dark">
  Details
</a>
```

2. Parámetros separados mediante coma

```
<a [routerLink]="/details", dept.numero, dept.nombre,dept.localidad"
class="btn btn-dark">
  Details
</a>
```

El siguiente paso es ir al componente de Details y recuperar los parámetros.

DETAILS.COMPONENT.TS

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
import { Departamento } from '../../../../../models/departamento';
@Component({
  selector: 'app-details',
  standalone: false,
  templateUrl: './details-component.html',
  styleUrls: ['./details-component.css'],
})
export class DetailsComponent implements OnInit {
  public departamento!: Departamento;
  constructor(private _activeRoute: ActivatedRoute) {}
  ngOnInit(): void {
    this._activeRoute.params.subscribe((params: Params) => {
      let id = parseInt(params["id"]);
      let nombre = params["nombre"];
      let localidad = params["localidad"];
      this.departamento = new Departamento(id, nombre, localidad);
    })
  }
}
```

DETAILS.COMPONENT.HTML

```
<h1>Details</h1>
<a [routerLink]="/">Back to index</a>
@if (departamento){
  <ul class="list-group">
    <li class="list-group-item">Id {{departamento.numero}}</li>
    <li class="list-group-item">Nombre: {{departamento.nombre}}</li>
    <li class="list-group-item">Localidad: {{departamento.localidad}}</li>
  </ul>
}
```

Vamos a realizar la acción de editar un departamento.

Vamos a enviar un ID por Url como parámetro, buscamos el departamento al cargar el componente Y mostramos los datos para editar.

Creamos un nuevo componente llamado **edit** y lo ponemos en la ruta mediante un parámetro **:id**

APP.ROUTING.TS

```
const routes: Routes = [
  { path: "", component: DepartamentosComponent },
  { path: "create", component: CreateComponent },
  { path: "details/:id/:nombre/:localidad", component: DetailsComponent },
];
```

```

    { path: "create", component: CreateComponent },
    { path: "details/:id/:nombre/:localidad", component: DetailsComponent },
    { path: "edit/:id", component: EditComponent }
];

```

Incluimos otro router link dentro de (Home) departamentos component para llamar a edición

DEPARTAMENTOSCOMPONENT

```

<a [routerLink]="/edit", dept.numero>
  Edit
</a>

```

Sobre nuestro servicio, creamos un nuevo método para buscar un departamento por ID

SERVICE.DEPARTAMENTOS.TS

```

findDepartamento(idDepartamento: number): Observable<Departamento> {
  let request = "api/departamentos/" + idDepartamento;
  let url = environment.urlApiDepartamentos + request;
  return this._http.get<Departamento>(url);
}

updateDepartamento(departamento: Departamento): Observable<any> {
  let json = JSON.stringify(departamento);
  let header = new HttpHeaders().set("Content-type", "application/json");
  let request = "api/departamentos";
  let url = environment.urlApiDepartamentos + request;
  return this._http.put(url, json, {headers: header});
}

```

Realizamos el dibujo de Edit component.

EDITCOMPONENT.HTML

```

<h1 style="color:blue">Edit</h1>
<a [routerLink]="/">Back to Home</a>
@if (departamento){
  <form #editForm="ngForm">
    <label>Id: </label>
    <input type="number" #cajaId name="cajaId"
      value={{departamento.numero}}
      class="form-control" disabled/>
    <label>Nombre</label>
    <input type="text" name="cajanombre" #cajanombre
      value={{departamento.nombre}} class="form-control"/>
    <label>Localidad</label>
    <input type="text" name="cajalocalidad" #cajalocalidad
      value={{departamento.Localidad}} class="form-control"/>
    <button class="btn btn-info" (click)="updateDepartamento()">
      Update
    </button>
  </form>
}

```

EDITCOMPONENT.TS

```

import { Component, ViewChild, ElementRef, OnInit } from '@angular/core';
import { Departamento } from '../../../../../models/departamento';
import ServiceDepartamentos from '../../../../../services/service.departamentos';
import { ActivatedRoute, Params, Router } from '@angular/router';
@Component({
  selector: 'app-edit',
  standalone: false,
  templateUrl: './edit-component.html',
  styleUrls: ['./edit-component.css'],
})
export class EditComponent implements OnInit {
  public departamento!: Departamento;
  @ViewChild("cajaId") cajaId!: ElementRef;
}

```

```

@ViewChild("cajanombre") cajaNombre!: ElementRef;
@ViewChild("cajalocalidad") cajaLocalidad!: ElementRef;
constructor(
  private _service: ServiceDepartamentos,
  private _activeRoute: ActivatedRoute,
  private _router: Router
) {}
ngOnInit(): void {
  this._activeRoute.params.subscribe((params: Params) => {
    let id = parseInt(params["id"]);
    this._service.findDepartamento(id).subscribe(response => {
      this.departamento = response;
    })
  })
}
updateDepartamento(): void {
  let id = parseInt(this.cajaId.nativeElement.value);
  let nombre = this.cajaNombre.nativeElement.value;
  let localidad = this.cajaLocalidad.nativeElement.value;
  let editDepartamento = new Departamento(id, nombre, localidad);
  this._service.updateDepartamento(editDepartamento).subscribe(response =>
{
  console.log("Edit")
  this._router.navigate(["/details"], {editDepartamento.numero
    , editDepartamento.nombre, editDepartamento.localidad});
})
}
}

```

El último paso que nos queda es Eliminar. Tenemos dos posibilidades.

1. Eliminar mediante Router Link enviando el ID por params

```
<a [routerLink]="/[delete', 10]">Delete 10</a>
```

2. Eliminar con un método con un Button

```
<button (click)="deleteDepartamento(10)">Delete 10</button>
```

Creamos un nuevo método dentro del Service

SERVICE.DEPARTAMENTOS.TS

```

deleteDepartamento(idDepartamento: number): Observable<any> {
  let request = "api/departamentos/" + idDepartamento;
  let url = environment.urlApiDepartamentos + request;
  return this._http.delete(url);
}

```

Vamos a modificar el código de Departamentos Component (Home)

DEPARTAMENTOS.COMPONENT.TS

```

import { Component, OnInit } from '@angular/core';
import ServiceDepartamentos from '../../../../../services/service.departamentos';
import { Departamento } from '../../../../../models/departamento';
@Component({
  selector: 'app-departamentos',
  standalone: false,
  templateUrl: './departamentos-component.html',
  styleUrls: ['./departamentos-component.css'],
})
export class DepartamentosComponent implements OnInit {
  public departamentos!: Array<Departamento>;
  constructor(private _service: ServiceDepartamentos) {}
  loadDepartamentos(): void {
    this._service.getDepartamentos().subscribe(response => {
      this.departamentos = response;
    })
  }
  ngOnInit(): void {
    this.loadDepartamentos();
  }
  deleteDepartamento(id: number): void {
    this._service.deleteDepartamento(id).subscribe(response => {
      console.log("delete");
      this.loadDepartamentos();
    })
  }
}

```

DEPARTAMENTOS.COMPONENT.HTML

```
<button class="btn btn-danger">
```

```
(click)="deleteDepartamento(dept.numero)">  
    Delete  
</button>
```

Vamos a realizar una aplicación para dibujar Empleados

```
[  
  {  
    "idEmpleado": 0,  
    "apellido": "string",  
    "oficio": "string",  
    "salario": 0,  
    "director": 0  
  }  
]
```

Tenéis un "pequeño" problema para acceder a los datos, están con seguridad.

El API, mediante un Login, necesita acceder a los datos recuperando un token

Nota: No nos sirve swagger para probar esto, debemos hacerlo con un Client.

<https://apiempleadoscoreoauth.azurewebsites.net/index.html>

Abrimos insomnia y vamos a visualizar cómo podemos realizar las peticiones.

Ejemplos de usuarios para la prueba:

- FORD, 7322
 - REY, 7839
 - GUTIERREZ, 7614
 - TOBMO, 7777

Para realizar este tipo de peticiones debemos seguir los siguientes pasos:

1. Petición a nuestro Login

Auth	
POST	/Auth/Login
Example Value	Schema
{ "userN ^{ame} : "string", "password": "string" }	

2. La petición nos devolverá un Token

The screenshot shows a Postman interface with the following details:

- Request Method:** POST
- Request URL:** https://apiempleadoscoreoauth.azurewebsites.net
- Status:** 200 OK
- Response Time:** 375 ms
- Response Size:** 382 B
- Timestamp:** Just Now

The request body is JSON:

```
1 {  
2   "userName": "REY",  
3   "password": "7839"  
4 }
```

The response body is JSON:

```
1 {  
2   "response":  
3     "eyJhbGciOiJIUzI1NiIsInR5cCI6Ikp  
XVCJ9.eyJvc2VyRGF0YSI6IntcIk1kRW  
1wbGVhZG9cIjo3ODM5LFwiQXB1bGxpZG  
9cIjpcI1JFWVwiLFwiT2ZpY21vXCI6XC  
JQUKVT SURT1RFXCIsXCJTYWxhcmlvXC  
I6NjUwMDAwLFwiRGlyZWN0b3JcIjowfS  
IsIm5iziI6MTc2MjkzOTkxOSwiZXhwIj
```

```
OXNZYy01QWNTESLCJpc3M101JOdHKwcz
ovL2xvY2FsaG9zdDo0NDM3NS8iLCJhdW
Qi0iJBcGlFbXBsZWfkB3NDb3JlT0F1dG
gifQ.tjkFOLKcQvWmDNQ7ZaTbUVDDxjB
bzy-6sduseZgzbM"
```

3 }

3. Con el token, lo enviamos mediante **bearer** a nuestro servicio

Una vez que tenemos el Token, debemos hacer las peticiones, enviando en el Header la siguiente Información:

KEY: Authorization

VALUE: **bearer TOKEN**

Header	Value
Accept	/*
Host	<calculated at runtime>
User-Agent	insomnia/11.6.2
Authorization	bearer eyJhbGciOiJIUzI1N...

```

1  [
2   {
3     "idEmpleado": 7322,
4     "apellido": "FORD",
5     "oficio": "VENDEDOR",
6     "salario": 129000,
7     "director": 7919
8   },
9   {
10    "idEmpleado": 7369,
11    "apellido": "SANCHAS",
12    "oficio": "EMPLEADO",
13    "salario": 104000,
14    "director": 7902
15  },

```

Realizar una aplicación llamada **angularempleadosauth**

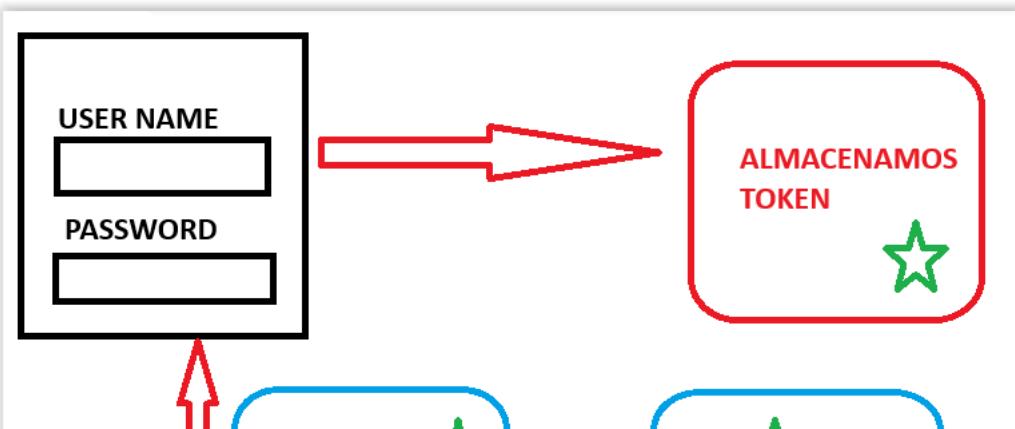
Debemos realizar las siguientes acciones:

- Un Component Home estático para el inicio de la página
- Tendremos un component Login para pedir la información del usuario y recuperar el Token
- Tendremos otro component dónde podremos visualizar el Perfil del empleado que Se ha validado.
- Tendremos otro component para mostrar los Subordinados del empleado del Login
- Por último, debemos almacenar la información intentando realizar la petición al Token Solamente una vez, es decir, almacenar en "algún sitio" nuestro token.

Global, Environment, localStorage

Si no tenemos Token, que nos muestre Login.

Si tenemos Token, veremos los datos.





Solución al ejemplo:

<https://github.com/serraguti/angularemployeedosauth2024>

Home Perfil Subordinados Login

User

Password

Log In

Home Perfil Subordinados Login

Perfil del empleado

Apellido REY
Oficio: PRESIDENTE
Salario: 650000
Director: 0

Home Perfil Subordinados Login

Subordinados

Apellido	Oficio	Salario
JIMENEZ	DIRECTOR	386789
NEGRO	DIRECTOR	370539
CEREZO	DIRECTOR	318539
SERRA	DIRECTOR	390039

EMPLEADO.TS

```
export class Empleado {
  constructor(
    public idEmpleado: number,
    public apellido: string,
    public oficio: string,
    public salario: number,
    public director: number
  ) {}
}
```

LOGIN.TS

```
export class Login {
  constructor()
```

```

    public userName: string,
    public password: string
  }={}
}

```

SERVICE.EMPLEADOS.TS

```

import { Injectable } from "@angular/core";
import { Login } from "../models/login";
import { Observable } from "rxjs";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { environment } from "../../environments/environment";
@Injectable()
export class ServiceEmpleados {
  public token: string;
  constructor(private _http: HttpClient) {
    this.token = "";
  }
  loginEmpleado(user: Login): Observable<any> {
    let json = JSON.stringify(user);
    let header = new HttpHeaders().set("Content-type", "application/json");
    let request = "auth/login";
    let url = environment.apiUrlEmpleados + request;
    return this._http.post(url, json, {headers: header});
  }
  getPerfilEmpleado(): Observable<any> {
    let request = "api/empleados/perfilempleado";
    let url = environment.apiUrlEmpleados + request;
    let header = new HttpHeaders().set("Authorization", "bearer " + this.token);
    return this._http.get(url, {headers: header});
  }
  getSubordinados(): Observable<any> {
    let request = "api/empleados/subordinados";
    let url = environment.apiUrlEmpleados + request;
    let header = new HttpHeaders().set("Authorization", "bearer " + this.token);
    return this._http.get(url, {headers: header});
  }
}

```

RETO 1

El siguiente reto que tenéis que lograr es poder subir ficheros a un Server de un Api.

Un servidor Api contiene una petición para poder subir un fichero con los siguientes datos:

- Filename: el nombre del fichero
- FileContent: El contenido, en string base64, para poder subir el fichero.

Nuestra URL para el Api está en la siguiente dirección:

<https://apipostfiles.azurewebsites.net/>

Como hemos visto, debemos enviar la información de un Model:

**FILEMODEL.TS**

```

export class FileModel {
  constructor(
    public filename: string,
    public filecontent: string
  ){}
}

```

Posteriormente, la petición al servicio es como cualquier petición con un POST mediante Un Model.

SERVICE.POSTFILES.TS

```

import { HttpClient, HttpHeaders } from "@angular/common/http";
import { Injectable } from "@angular/core";
import { Observable } from "rxjs";
import { FileModel } from "../models/filemodel";
@Injectable()
export class ServicePostFiles {
  constructor(private _http: HttpClient) {}
  //VOY A RECIBIR DIRECTAMENTE EL OBJETO EN EL METODO DE INSERTAR
  postFile(fileModel: FileModel): Observable<any>{
    let json = JSON.stringify(fileModel);
  }
}

```

```
//DEBEMOS INDICAR EN LA PETICION QUE TIPO DE FORMATO TIENE EL OBJETO
A ENVIAR
    let header = new HttpHeaders();
    header = header.set("Content-type", "application/json");
    let request = "api/testingfiles";
    let url = "https://apipostfiles.azurewebsites.net/" + request;
    return this._http.post(url, json, {headers: header});
}

}
```

En nuestro Component, he creado un Form con un input file y un button con un método

TESTINGFILES.COMPONENT.HTML

```
<h1>Test Files Post</h1>
<form #fileForm="ngForm">
  <label>Selecciona fichero</label>
  <input type="file" name="cajafile" #cajafile/><br/>
  <button (click)="subirFichero()">
    Subir fichero
  </button>
</form>
<p>{{fileContent}}</p>
@if (urlFileUpload){
  <img src={{urlFileUpload}}/>
}
```

En el componente de typescript necesitamos un **FileReader**, recuperar el nombre del fichero Y el propio file.

TESTINGFILES.COMPONENT.TS

```
import { Component, OnInit, ViewChild, ElementRef } from '@angular/core';
import { ServicePostFiles } from '../../../../../services/service.postfiles';
import { FileModel } from '../../../../../models/filemodel';
@Component({
  selector: 'app-testingfiles',
  templateUrl: './testingfiles.component.html',
  styleUrls: ['./testingfiles.component.css'
})
export class TestingfilesComponent implements OnInit {
  @ViewChild("cajafile") cajaFileRef!: ElementRef;
  public fileContent: string;
  public urlFileUpload!: string;
  constructor(private _service: ServicePostFiles) {
    this.fileContent = "";
  }
  ngOnInit(): void {
  }
  subirFichero(): void{
    //ESTE ES EL FICHERO QUE DEBEMOS LEER
    var file = this.cajaFileRef.nativeElement.files[0];
    //ELIMINAMOS LAS BARRAS QUE INCLUYE EL TIPO FILE EN EL NAME
    //YA QUE VIENE LA RUTA Y NECESITAMOS EL NOMBRE DEL FICHERO
    var miPath = this.cajaFileRef.nativeElement.value.split("\\\\");
    //NOS QUEDAMOS CON EL ULTIMO VALOR, QUE ES EL NOMBRE DEL FILE
    var ficheroNombre = miPath[2];
    console.log(ficheroNombre);
    //CREAMOS UN LECTOR PARA LEER EL FICHERO
    var reader = new FileReader();
    reader.readAsArrayBuffer(file);
    reader.onloadend = () => {
      let buffer: ArrayBuffer;
      buffer = reader.result as ArrayBuffer;
      var base64: string;
      //LA FUNCION btoa CONVIERTA BYTES A BASE64
      base64 = btoa(
        new Uint8Array(buffer)
          .reduce((data, byte) => data + String.fromCharCode(byte), '')
      );
      this.fileContent = base64;
      var newFileModel =
        new FileModel(ficheroNombre, base64);
      this._service.postFile(newFileModel).subscribe(response => {
        console.log(response);
        this.urlFileUpload = response.urlFile;
      })
    };
  }
}
```

Solución al ejemplo:

<https://github.com/serraguti/angularpostfillesservices>

RETO 2

El siguiente reto también es investigación.

Sabemos hacerlo perfectamente, pero debemos buscar la forma de hacerlo mejor.

Vamos a realizar "algo parecido" a lo que hicimos de Apuestas

<https://apiapuestas.azurewebsites.net/index.html>

Por un lado, tendremos **jugador**

JUGADOR.TS

```
export class Jugador {
  constructor(
    public idjugador: number,
    public nombre: string,
    public posicion: string,
    public imagen: string,
    public fechaNacimiento: string,
    public pais: string,
    public idEquipo: number
  ){}
}
```

EQUIPO.TS

```
export class Equipo {
  constructor(
    public idEquipo: number,
    public nombre: string,
    public imagen: string,
    public champions: number,
    public web: string,
    public descripcion: string,
    public finalista: number
  ){}
}
```

Al final, lo que tenemos que devolver es el conjunto.

DATOSEQUIPO.TS

```
import { Equipo } from "./equipo";
import { Jugador } from "./jugador";

export class DatosEquipo {
  public equipo!: Equipo;
  public jugadores!: Array<Jugador>;
  constructor(){}
}
```

Debemos tener un servicio que nos devuelva los datos:

SERVICE.FUTBOL.TS

```
getJugadoresEquipo(idEquipo: number): Observable<Array<Jugador>> {
  let request = "api/jugadores/jugadoresequipos/" + idEquipo;
  let url = environment.urlApiEjemplos + request;
  return this._http.get<Array<Jugador>>(url);
}

findEquipo(idEquipo: number): Observable<Equipo> {
  let request = "api/equipos/" + idEquipo;
  let url = environment.urlApiEjemplos + request;
  return this._http.get<Equipo>(url);
}

getEquipos(): Observable<Array<Equipo>> {
  let request = "api/equipos";
  let url = environment.urlApiEjemplos + request;
  return this._http.get<Array<Equipo>>(url);
}
```

```
ngOnInit(): void {
  this._activeRoute.params.subscribe((params: Params) => {
    let idEquipo = parseInt(params['idequipo']);
    let datos: DatosEquipo;
    datos = new DatosEquipo();
    this._service.findEquipo(idEquipo).subscribe(response => {
      response.descripcion = response.descripcion.substring(0, 250);
      datos.equipo = response;
    })
    this._service.getJugadoresEquipo(idEquipo).subscribe(response => {
      datos.jugadores = response;
    })
    this.dataequipo = datos;
  })
}
```

```
})
}
```

Debemos crear un método dentro del Service que nos devuelva un Observable
De **DATOS_EQUIPO** cuando ya tengamos todos los datos.

En Angular, existe una clase llamada **Fork Join**

Ejemplo de Usabilidad

CLASE DE RESPUESTA DE UN CONJUNTO DE PETICIONES

```
export class ServerResponse {
    public client: string;
    public otherdata: string;
    constructor(){
        this.client = "";
        this.otherdata = "";
    }
}
```

EN UNA CLASE SERVICE, REALIZAMOS EL FORK JOIN

```
import { Observable, of, forkJoin, Observer } from "rxjs";
```

Tendremos varios métodos, cada uno con sus peticiones a la Api y con la Devolución de sus datos. Datos de Ejemplo aquí:

```
getClientData(): Observable<any> {
    return of({ client: 'Client 1' });
}
//TENEMOS OTRO METODO QUE LUEGO VOY A CONVERTIR EN JUGADORES
getOtherData(): Observable<any> {
    // Fake server latency
    return of({ other: 'Aquí ponemos un retraso de 2 segundos' })
        .pipe(delay(2000));
}
```

Y realizamos un método conjunto, devolviendo la respuesta de las dos peticiones a la vez,
Por ejemplo, en nuestro caso **ServerResponse**

```
getData(): Observable<ServerResponse> {
    const allOperations = forkJoin(
        {
            requestOne: this.getClientData(),
            requestTwo: this.getOtherData()
        }
    );
    const observable: Observable<ServerResponse> = new
    Observable((observer: Observer<ServerResponse>) => {
        allOperations.subscribe(({requestOne, requestTwo}) => {
            const data = new ServerResponse();
            data.client = requestOne;
            data.otherdata = requestTwo;
            observer.next(data);
            observer.complete();
        });
    });
    return observable;
}
```

SOLUCION AL EJEMPLO

<https://github.com/serraguti/angularequiposjugadores>

