	UNIVERSIDAD DON BOSCO **FACULTAD DE INGENIERÍA** **ESCUELA DE COMPUTACIÓN**	
CICLO: 02	**GUIA DE LABORATORIO #04**	
	Nombre de la Práctica:	**Angular parte I**
	MATERIA:	**Diseño y Programación de Software Multiplataforma**

I. OBJETIVOS

Que el estudiante:

- Diseñe aplicaciones web utilizando funciones de Angular.
- Diseñe aplicaciones con navegación.
- Aprender hacer interfaz para que el usuario pueda interactuar con la aplicación.
- Aprender a crear componente personalizado.
- Hacer uso de bingding en el diseño de interfaz angular.
- Aprender hacer interfaz para que el usuario pueda interactuar con la aplicación.
- Aprender a utilizar pipes dentro de la interfaz de usuario.
- Hacer uso de los componentes en el diseño de interfaz angular.

Contenido

II. INTRODUCCION TEORICA

TYPESCRIPT: CLASES

TypeScript incorpora muchas características de la programación orientada a objetos disponibles en lenguajes como Java y C#.

La sintaxis básica de una clase puede ser:

```
class Persona {
  nombre: string;
  edad: number;

  constructor(nombre:string, edad:number) {
    this.nombre = nombre;
    this.edad = edad;
  }

  imprimir() {
    console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);
  }
}

let persona1: Persona;
persona1 = new Persona('Juan', 45);
persona1.imprimir();
```

Los atributos se definen fuera de los métodos. Para acceder a los mismos dentro de los métodos debemos anteceder la palabra clave 'this':

```
  imprimir() {
    console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);
  }
```

El constructor es el primer método que se ejecuta en forma automática al crear un objeto de la clase 'Persona':

```
 constructor(nombre:string, edad:number) {
   this.nombre = nombre;
   this.edad = edad;
 }
```

MODIFICADORES DE ACCESO A PROPIEDADES Y MÉTODOS.

Podemos definir propiedades y métodos privados y públicos antecediendo las palabras claves 'private' y 'public'.

Veamos un ejemplo:

```
class Dado {
  private valor: number;

  public tirar() {
    this.generar();
  }

  private generar() {
    this.valor = Math.floor(Math.random() * 6) + 1 ;
  }

  public imprimir() {
    console.log(`Salió el valor ${this.valor}`);
  }
}

let dado1=new Dado();
dado1.tirar();
```

```
dado1.imprimir();
```

El objeto 'dado1' tiene acceso a todos los atributos y métodos que tienen el modificador 'public' por eso podemos llamar a los métodos 'tirar' e 'imprimir':

```
dado1.tirar();
dado1.imprimir();
```

Se generará un error si queremos acceder al atributo 'valor' o al método 'generar':

```
let dado1=new Dado();
dado1.valor = 6;
dado1.generar();
```

Si no agregamos modificador de acceso por defecto es 'public', luego tendremos el mismo resultado si utilizamos la siguiente sintaxis para declarar la clase:

```
class Dado {
 private valor: number;

 tirar() {
  this.generar();
 }

 private generar() {
  this.valor = Math.floor(Math.random() * 6) + 1;
 }

 imprimir() {
  console.log(`Salió el valor ${this.valor}`);
 }
}
```

```
let dado1=new Dado();
dado1.tirar();
dado1.imprimir();
```

Es decir nos ahorramos de escribir 'public' antes de las propiedades y métodos que queremos definir con esta característica.

DEFINICIÓN E INICIALIZACIÓN DE PROPIEDADES EN LOS PARÁMETROS DEL CONSTRUCTOR.

Esta característica no es común en otros lenguajes orientados a objetos y tiene por objetivo crear clases más breves.

En TypeScript podemos definir algunas propiedades de la clase en la zona de parámetros del constructor, con esto nos evitamos de su declaración fuera de los métodos.

Por ejemplo la clase Persona la podemos codificar con ésta otra sintaxis:

```
class Persona {

  constructor(public nombre:string, public edad:number) { }

  imprimir() {
    console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);
  }
}

let persona1: Persona;
persona1 = new Persona('Juan', 45);
persona1.imprimir();
```

Como vemos el constructor tiene un bloque de llaves vacías ya que no tenemos que implementar ningún código en su interior, pero al anteceder el modificador de acceso en la zona de parámetros los mismos pasan a ser propiedades de la clase y no parámetros:

```
  constructor(public nombre:string, public edad:number) { }
```

Podemos sin problemas definir propiedades tanto 'public' como 'private'.

La definición de propiedades en la zona de parámetros solo se puede hacer en el constructor de la clase y no está permitido en cualquier otro método.

MODIFICADOR READONLY

Disponemos además de los modificadores 'private' y 'public' uno llamado 'readonly'. Mediante este modificador el valor de la propiedad solo puede ser cargado en el constructor o al momento de definirlo y luego no puede ser modificado ni desde un método de la clase o fuera de la clase.

Veamos un ejemplo con una propiedad 'readonly':

```
class Articulo {
  readonly codigo: number;
  descripcion: string;
  precio: number;

  constructor(codigo:number, descripcion:string, precio:number) {
    this.codigo=codigo;
    this.descripcion=descripcion;
    this.precio=precio;
  }

  imprimir() {
    console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);
  }
}

let articulo1: Articulo;
articulo1 = new Articulo(1,'papas',12.5);
articulo1.imprimir();
```

Una vez que se inicia la propiedad 'codigo' en el constructor su valor no puede cambiar:

```
  imprimir() {
    this.codigo=7; //Error
```

```
  console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);
 }
```

El mismo error se produce si tratamos de cambiar su valor desde fuera de la clase:

```
let articulo1: Articulo;
articulo1 = new Articulo(1,'papas',12.5);
articulo1.codigo=7; //Error
articulo1.imprimir();
```

Podemos utilizar también la sintaxis abreviada de propiedades:

```
class Articulo {

 constructor(readonly codigo:number, public descripcion:string, public precio:number) { }

 imprimir() {
  console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);
 }
}
```

PROPIEDADES ESTÁTICAS

Las propiedades estáticas pertenecen a la clase y no a las instancias de la clase. Se las define antecediendo el modificador 'static'.

Con un ejemplo quedará claro este tipo de propiedades:

```
class Dado {
 private valor: number;
 static tiradas:number=0;
 tirar() {
  this.generar();
 }
```

```
  private generar() {
    this.valor = Math.floor(Math.random() * 6) + 1 ;
    Dado.tiradas++;
  }

  imprimir() {
    console.log(`Salió el valor ${this.valor}`);
  }
}

let dado1=new Dado();
dado1.tirar();
dado1.imprimir();
let dado2=new Dado();
dado2.tirar();
dado2.imprimir();
console.log(`Cantidad de tiradas de dados:${Dado.tiradas}`); // 2
```

Una propiedad estática requiere el modificador 'static' previo a su nombre:

```
  static tiradas:number=0;
```

Para acceder a dichas propiedades debemos anteceder el nombre de la clase y no la palabra clave 'this':

```
    Dado.tiradas++;
```

No importan cuantos objetos de la clase se definan luego todos esos objetos comparten la misma variable estática:

```
let dado1=new Dado();
dado1.tirar();
dado1.imprimir();
let dado2=new Dado();
```

```
dado2.tirar();
dado2.imprimir();
console.log(`Cantidad de tiradas de dados:${Dado.tiradas}`); // 2
```

Es por eso que la propiedad 'tiradas' almacena un 2 luego de tirar el primer y segundo dado.

La propiedad 'valor' es independiente en cada dado pero la propiedad 'tiradas' es compartida por los dos objetos.

MÉTODOS ESTÁTICAS

Igual que las propiedades estáticas los métodos estáticos se los accede por el nombre de la clase. Este tipo de métodos solo pueden acceder a propiedades estáticas.

```
class Matematica {
 static mayor(v1:number, v2: number): number {
  if (v1>v2)
    return v1;
  else
    return v2;
 }

 static menor(v1:number, v2: number): number {
  if (v1<v2)
    return v1;
  else
    return v2;
 }

 static aleatorio(inicio: number, fin: number): number {
  return Math.floor((Math.random()*(fin+1-inicio))+inicio);
 }
}
```

```
let x1=Matematica.aleatorio(1,10);

let x2=Matematica.aleatorio(1,10);

console.log(`El mayor entre ${x1} y ${x2} es ${Matematica.mayor(x1,x2)} `);

console.log(`El menor entre ${x1} y ${x2} es ${Matematica.menor(x1,x2)}`);
```

Debemos anteceder la palabra clave static al nombre del método.

Cuando llamamos a un método debemos anteceder también el nombre de la clase, no hace falta definir una instancia u objeto de la clase:

```
let x1=Matematica.aleatorio(1,10);
```

TYPESCRIPT: FUNCIONES Y MÉTODOS

TypeScript aporta varias características que JavaScript no dispone hasta el momento cuando tenemos que plantear funciones y métodos.

PARÁMETROS TIPADOS Y FUNCIONES QUE RETORNAN UN VALOR.

Podemos indicar a cada parámetro el tipo de dato que puede recibir y también el tipo de dato que retorna la función o método en caso que estemos en una clase:

```
function sumar(valor1:number, valor2:number): number {
  return valor1+valor2;
}


console.log(sumar(10, 5));
```

La función sumar recibe dos parámetros de tipo number y retorna un valor de tipo number. Luego si llamamos a esta función enviando un valor distinto a number el compilador nos avisará del error:

```
console.log(sumar('juan', 'carlos'));
```

Se genera un error: "Argument of type '"juan"' is not assignable to parameter of type 'number'.

Inclusive editores de texto moderno como Visual Studio Code pueden antes de compilarse avisar del error.

El tipado estático favorece a identificar este tipo de errores antes de ejecutar la aplicación. Lo mismo cuando una función retorna un dato debemos indicar al final de la misma dicho tipo:

```
function sumar(valor1:number, valor2:number): number {
```

La función sumar retorna un valor de tipo number.

Luego si la función retorna un tipo distinto a number se genera un error:

```
function sumar(valor1:number, valor2:number): number {
  return 'Hola mundo';
}
```

Como estamos retornando un string se genera el error: Type '"Hola mundo"' is not assignable to type 'number'

FUNCIONES ANÓNIMAS.

Una función anónima no especifica un nombre. Son semejantes a JavaScript con la salvedad de la definición de tipos para los parámetros:

```
const funcSumar = function (valor1:number, valor2:number): number {
  return valor1 + valor2;
}


console.log(funcSumar(4, 9));
```

PARÁMETROS OPCIONALES.

En TypeScript debemos agregar el caracter '?' al nombre del parámetro para indicar que el mismo puede o no llegar un dato:

```

function sumar(valor1:number, valor2:number, valor3?:number):number {
  if (valor3)
    return valor1+valor2+valor3;
  else
    return valor1+valor2;
```

```
}


console.log(sumar(5,4));
console.log(sumar(5,4,3));
```

El tercer parámetro es opcional:

```
function sumar(valor1:number, valor2:number, valor3?:number):number {
```

Luego a la función 'sumar' la podemos llamar pasando 2 o 3 valores numéricos:

```
console.log(sumar(5,4));
console.log(sumar(5,4,3));
```

Si pasamos una cantidad de parámetros distinta a 2 o 3 se genera un error en tiempo de compilación: ' error TS2554: Expected 2-3 arguments, but got 4.'

Los parámetros opcionales deben ser los últimos parámetros definidos en la función. Puede tener tantos parámetros opcionales como se necesiten.

PARÁMETROS POR DEFECTO.

Esta característica de TypeScript nos permite asignar un valor por defecto a un parámetro para los casos en que la llamada a la misma no se le envíe.

```
function sumar(valor1:number, valor2:number, valor3:number=0):number {
  return valor1+valor2+valor3;
}


console.log(sumar(5,4));
console.log(sumar(5,4,3));
```

El tercer parámetro almacena un cero si no se lo pasamos en la llamada:

```
console.log(sumar(5,4));
```

Puede haber varios valores por defecto, pero deben ser los últimos. Es decir primero indicamos los parámetros que reciben datos en forma obligatoria cuando los llamamos y finalmente indicamos aquellos que tienen valores por defecto.

PARÁMETROS REST.

Otra característica de TypeScript es la posibilidad de pasar una lista indefinida de valores y que los reciba un vector.

El concepto de parámetro Rest se logra antecediendo tres puntos al nombre del parámetro:

```
function sumar(...valores:number[]) {
  let suma=0;
  for(let x=0;x<valores.length;x++)
    suma+=valores[x];
  return suma;
}


console.log(sumar(10, 2, 44, 3));
console.log(sumar(1, 2));
console.log(sumar());
```

El parámetro 'valores' se le anteceden los tres puntos seguidos e indicamos que se trata de un vector de tipo 'number'. Cuando llamamos a la función le pasamos una lista de valores enteros que luego la función los empaqueta en el vector:

```
console.log(sumar(10, 2, 44, 3));
console.log(sumar(1, 2));
console.log(sumar());
```

La función con un parámetro Rest puede tener otros parámetros pero se deben declarar antes.

Los parámetros Rest no pueden tener valores por defecto.

OPERADOR SPREAD.

El operador Spread permite descomponer una estructura de datos en elementos individuales. Es la operación inversa de los parámetros Rest. La sintaxis se aplica anteponiendo al nombre de la variable tres puntos:

```
function sumar(valor1:number, valor2:number, valor3:number):number {
  return valor1+valor2+valor3;
}
```

```
const vec:number[]=[10,20,30];
const s=sumar(...vec);
console.log(s);
```

FUNCIONES CALLBACKS

Una función callback es una función que se pasa a otra función como parámetro y dentro de la misma es llamada.

```
function operar(valor1: number, valor2: number, func: (x: number, y:number)=>number): number {
  return func(valor1, valor2);
}

console.log(operar(3, 7, (x: number,y: number): number => {
  return x+y;
}))

console.log(operar(3, 7, (x: number,y: number): number => {
  return x-y;
}))

console.log(operar(3, 7, (x: number,y: number): number => {
  return x*y;
}))
```

La función operar recibe tres parámetros, los dos primeros son de tipo 'number' y el tercero es de tipo función:

```
function operar(valor1: number, valor2: number, func: (x: number, y:number)=>number): number {
```

La función que debe recibir debe tener como parámetros dos 'number' y retornar un 'number'.

Cuando llamamos a la función además de los dos enteros le debemos pasar una función que reciba dos 'number' y retorne un 'number':

```
console.log(operar(3, 7, (x: number,y: number): number => {
  return x+y;
}))
```

Como podemos observar llamamos a la función 'operar' tres veces y le pasamos funciones que procesan los dos enteros para obtener su suma, resta y multiplicación.

Para hacer más claro nuestro código TypeScript mediante la palabra clave type permite crear nuevos tipos y luego reutilizarlos:

```
type Operacion=(x: number, y:number)=>number;


function operar(valor1: number, valor2: number, func: Operacion): number {
  return func(valor1, valor2);
}


console.log(operar(3, 7, (x: number,y: number): number => {
  return x+y;
}))


console.log(operar(3, 7, (x: number,y: number): number => {
  return x-y;
}))


console.log(operar(3, 7, (x: number,y: number): number => {
  return x*y;
}))
```

El tipo Operacion tiene la firma de una función con dos parámetros de tipo 'number' y el retorno de un 'number'. Luego cuando declaramos la función operar definimos el tercer parámetro llamado 'func' de tipo 'Operacion':

```
function operar(valor1: number, valor2: number, func: Operacion): number {
```

PARÁMETROS DE TIPO UNIÓN.

Vimos en otro concepto que podemos definir variables que pueden almacenar más de un tipo de dato indicando los mismos el operador '|':

```
let edad: number | string;
edad=34;
console.log(edad);
edad='20 años';
console.log(edad);
```

Con parámetros podemos utilizar la misma sintaxis:

```
function sumar(valor1: number | string, valor2: number | string ): number | string  {
  if (typeof valor1 ==='number' && typeof valor2 ==='number')
    return valor1+valor2;
  else
    return valor1.toString() + valor2.toString();
}


console.log(sumar(4, 5));
console.log(sumar('Hola ', 2));
console.log(sumar('Hola ', 'Mundo'));
```

En este tipo de caso deberemos identificar que operación realizar según los tipos de datos de los parámetros. En el ejemplo si los dos parámetros se reciben tipos de datos 'number' procedemos a sumarlos como enteros:

```
  if (typeof valor1 ==='number' && typeof valor2 ==='number')
    return valor1+valor2;
```

En el caso contrario con que uno de los dos valores sea de tipo 'string' procedemos a concatenarlos, previamente los convertimos a string:

```
 else
  return valor1.toString() + valor2.toString();
```

ACOTACIONES

Hemos hecho siempre ejemplos con funciones, pero todos estos conceptos se aplican si planteamos métodos dentro de una clase:

```
class Operacion {
 sumar(...valores:number[]) {
  let suma=0;
  for(let x=0;x<valores.length;x++)
   suma+=valores[x];
  return suma;
 }
}

let op: Operacion;
op=new Operacion();
console.log(op.sumar(1,2,3));
```

TYPESCRIPT: HERENCIA

La herencia es otra característica fundamental de la programación orientada a objetos y TypeScript lo implementa.

La herencia significa que se pueden crear nuevas clases partiendo de clases existentes, que tendrá todas los atributos y los métodos de su 'superclase' o 'clase padre' y además se le podrán añadir otros atributos y métodos propios.

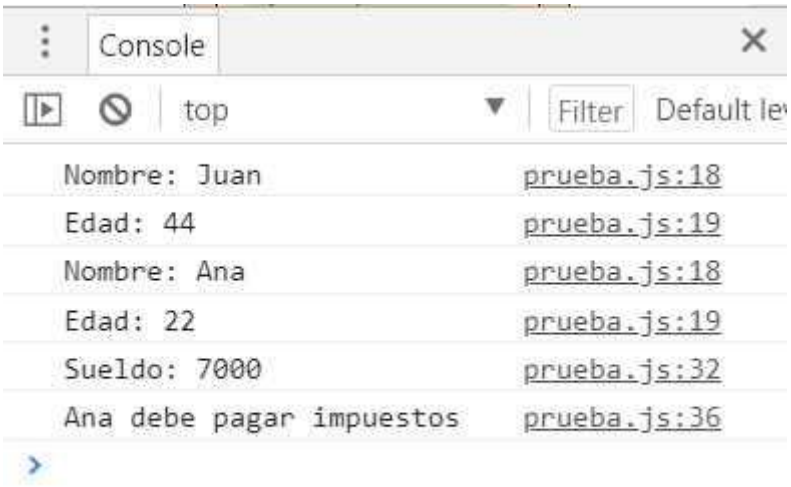
Veamos con un ejemplo la sintaxis que plantea TypeScript para implementar la herencia:

```
class Persona {
 protected nombre:string;
```

```
  protected edad:number;
  constructor(nombre:string, edad:number) {
    this.nombre = nombre;
    this.edad = edad;
  }

  imprimir() {
    console.log(`Nombre: ${this.nombre}`);
    console.log(`Edad: ${this.edad}`);
  }
}

class Empleado extends Persona {
  private sueldo:number;
  constructor(nombre:string, edad:number, sueldo:number) {
    super(nombre, edad);
    this.sueldo = sueldo;
  }

  imprimir() {
    super.imprimir();
    console.log(`Sueldo: ${this.sueldo}`);
  }

  pagaImpuestos() {
    if (this.sueldo>5000)
      console.log(`${this.nombre} debe pagar impuestos`);
    else
      console.log(`${this.nombre} no debe pagar impuestos`);
```

```
 }
}



const persona1=new Persona('Juan', 44);
persona1.imprimir();


const empleado1=new Empleado('Ana', 22, 7000);
empleado1.imprimir();
empleado1.pagaImpuestos();
```

El resultado de ejecutar en la consola del navegador es (recordar que la consola la podemos abrir desde el navegador presionando la tecla F12):



Mediante la palabra clave extends indicamos el nombre de la clase padre. Una clase puede heredar de una sola clase (en este ejemplo 'Persona'):

```
class Empleado extends Persona {
 private sueldo:number;
 constructor(nombre:string, edad:number, sueldo:number) {
  super(nombre, edad);
  this.sueldo = sueldo;
```

```
}
```

La subclase Empleado puede acceder a las propiedades de la clase padre si los mismos se definieron en forma public o protected:

```
imprimir() {
  console.log(`Sueldo: ${this.sueldo} y lo cobra ${this.nombre}`);
}
```

Con el modificador protected permitimos que la subclase pueda acceder a los atributos de la clase padre pero luego donde definamos un objeto de esta clase no los pueda acceder y permanezcan encapsulados:

```
const empleado1=new Empleado('Ana', 22, 7000);
empleado1.nombre='facundo'; //error
```

CLASES ABSTRACTAS

En algunas situaciones tenemos métodos y propiedades comunes a un conjunto de clases, podemos agrupar dichos métodos y propiedades en una clase abstracta.

Hay una sintaxis especial en TypeScript para indicar que una clase es abstracta.

No se pueden definir objetos de una clase abstracta y seguramente será heredada por otras clases de las que si podremos definir objetos.

Problema: Declarar una clase abstracta que represente una Operación. Definir en la misma tres propiedades valor1, valor2 y resultado, y tres métodos: constructor, imprimir y operar (éste último hacerlo abstracto). Plantear dos clases llamadas Suma y Resta que hereden de la clase Operación e implementen el método abstracto operar.

```
abstract class Operacion {
  public valor1: number;
  public valor2: number;
  public resultado: number=0;

  constructor(v1:number, v2:number) {
    this.valor1=v1;
    this.valor2=v2;
  }
```

```
  abstract operar():void;

  imprimir() {
    console.log(`Resultado: ${this.resultado}`);
  }
}

class Suma extends Operacion {
  constructor(v1:number, v2:number) {
    super(v1,v2);
  }

  operar() {
      this.resultado = this.valor1 + this.valor2;
  }
}

class Resta extends Operacion {
  constructor(v1:number, v2:number) {
    super(v1,v2);
  }

  operar() {
      this.resultado = this.valor1 - this.valor2;
  }
}

let suma1: Suma;
```

```
suma1=new Suma(10, 4);
suma1.operar();
suma1.imprimir();

let resta1: Resta;
resta1=new Resta(10, 4);
resta1.operar();
resta1.imprimir();
```

Mediante la palabra clave abstract indicamos que la clase debe definirse como abstracta, luego no se pueden definir objetos de la clase Operacion:

```
abstract class Operacion {
  public valor1: number;
  public valor2: number;
  public resultado: number=0;

  constructor(v1:number, v2:number) {
    this.valor1=v1;
    this.valor2=v2;
  }

  abstract operar():void;

  imprimir() {
    console.log(`Resultado: ${this.resultado}`);
  }
}
```

Dentro de la clase abstracta definimos un método abstracto llamado operar, esto obliga a todas las clases que heredan de Operación implementar el algoritmo de dicho método, sino se genera un error en tiempo de compilación.

La subclase Suma al heredar de Operación implementa el método operar:

```
class Suma extends Operacion {
  constructor(v1:number, v2:number) {
    super(v1,v2);
  }

  operar() {
      this.resultado = this.valor1 + this.valor2;
  }
}
```

Solo podemos definir objetos de las clases Suma y Resta. Se genera un error si tratamos de crear un objeto de la clase Operacion:

```
let suma1: Suma;
suma1=new Suma(10, 4);
suma1.operar();
suma1.imprimir();
```

TYPESCRIPT: INTERFACES

Una interface declara una serie de métodos y propiedades que deben ser implementados luego por una o más clases.

Las interfaces vienen a suplir la imposibilidad de herencia múltiple.

Por ejemplo podemos tener dos clases que representen un avión y un helicóptero. Luego plantear una interface con un método llamado volar. Las dos clases pueden implementar dicha interface y codificar el método volar (los algoritmos seguramente sean distintos pero el comportamiento de volar es común tanto a un avión como un helicóptero)

La sintaxis en TypeScript para declarar una interface es:

```
interface [nombre de la interface] {
   [declaración de propiedades]
   [declaración de métodos]
```

```
}
```

PROBLEMA

Definir una interface llamada Punto que declare un método llamado imprimir. Luego declarar dos clases que la implementen.

```
interface Punto {
  imprimir(): void;
}

class PuntoPlano implements Punto{
  constructor(private x:number, private y:number) {}

  imprimir() {
    console.log(`Punto en el plano: (${this.x},${this.y})`);
  }
}

class PuntoEspacio implements Punto{
  constructor(private x:number, private y:number, private z:number) {}

  imprimir() {
    console.log(`Punto en el espacio: (${this.x},${this.y},${this.z})`);
  }
}



let puntoPlano1: PuntoPlano;
puntoPlano1 = new PuntoPlano(10, 4);
puntoPlano1.imprimir();
```

```
let puntoEspacio1: PuntoEspacio;
puntoEspacio1 = new  PuntoEspacio(20, 50, 60);
puntoEspacio1.imprimir();
```

Para declarar una interface en TypeScript utilizamos la palabra clave interface y seguidamente su nombre. Luego entre llaves indicamos todas las cabeceras de métodos y propiedades. En nuestro ejemplo declaramos la interface Punto e indicamos que quien la implemente debe definir un método llamado imprimir sin parámetros y que no retorna nada:

```
interface Punto {
  imprimir(): void;
}
```

Por otro lado declaramos dos clases llamados PuntoPlano con dos propiedades y PuntoEspacio con tres propiedades, además indicamos que dichas clases implementarán la interface Punto:

```
class PuntoPlano implements Punto{
  constructor(private x:number, private y:number) {}

  imprimir() {
    console.log(`Punto en el plano: (${this.x},${this.y})`);
  }
}

class PuntoEspacio implements Punto{
  constructor(private x:number, private y:number, private z:number) {}

  imprimir() {
    console.log(`Punto en el espacio: (${this.x},${this.y},${this.z})`);
  }
}
```

La sintaxis para indicar que una clase implementa una interface requiere disponer la palabra clave implements y en forma seguida el o los nombres de interfaces a implementar. Si una clase hereda de otra también puede implementar una o más interfaces.

El método imprimir en cada clase se implementa en forma distinta, en uno se imprimen 3 propiedades y en la otra se imprimen 2 propiedades.

Luego definimos un objeto de la clase PuntoPlano y otro de tipo PuntoEspacio:

```
let puntoPlano1: PuntoPlano;
puntoPlano1 = new PuntoPlano(10, 4);
puntoPlano1.imprimir();


let puntoEspacio1: PuntoEspacio;
puntoEspacio1 = new  PuntoEspacio(20, 50, 60);
puntoEspacio1.imprimir();
```

Si una clase indica que implementa una interfaz y luego no se la codifica, se genera un error en tiempo de compilación informándonos de tal situación (inclusive el editor Visual Studio Code detecta dicho error antes de compilar):

```
prueba.ts(5,7): error TS2420: Class 'PuntoPlano' incorrectly implements interface 'Punto'.
  Property 'imprimir' is missing in type 'PuntoPlano'.
prueba.ts(20,13): error TS2339: Property 'imprimir' does not exist on type 'PuntoPlano'.
```

Este error se produce si codificamos la clase sin implementar el método imprimir:

```
class PuntoPlano implements Punto{
  constructor(private x:number, private y:number) {}
}
```

PROBLEMA

Se tiene la siguiente interface:

```
interface Figura {
  superficie: number;
  perimetro: number;
  calcularSuperficie(): number;
  calcularPerimetro(): number;
```

```
}
```

Declara dos clases que representen un Cuadrado y un Rectángulo. Implementar la interface Figura en ambas clases.

```
interface Figura {
  superficie: number;
  perimetro: number;
  calcularSuperficie(): number;
  calcularPerimetro(): number;
}

class Cuadrado implements Figura {
  superficie: number;
  perimetro: number;
  constructor(private lado:number) {
    this.superficie = this.calcularSuperficie();
    this.perimetro = this.calcularPerimetro();
  }

  calcularSuperficie(): number {
    return this.lado * this.lado;
  }

  calcularPerimetro(): number {
    return this.lado * 4;
  }
}
```

```
class Rectangulo implements Figura {
  superficie: number;
  perimetro: number;
  constructor(private ladoMayor:number, private ladoMenor:number) {
    this.superficie = this.calcularSuperficie();
    this.perimetro = this.calcularPerimetro();
  }

  calcularSuperficie(): number {
    return this.ladoMayor * this.ladoMenor;
  }

  calcularPerimetro(): number {
    return (this.ladoMayor * 2) + (this.ladoMenor * 2);
  }
}
let cuadrado1: Cuadrado;
cuadrado1 = new Cuadrado(10);
console.log(`Perimetro del cuadrado : ${cuadrado1.calcularPerimetro()}`);
console.log(`Superficie del cuadrado : ${cuadrado1.calcularSuperficie()}`);
let rectangulo1: Rectangulo;
rectangulo1 = new Rectangulo(10, 5);
console.log(`Perimetro del rectangulo : ${rectangulo1.calcularPerimetro()}`);
console.log(`Superficie del cuadrado : ${rectangulo1.calcularSuperficie()}`);
```

En este problema la interface Figura tiene dos métodos que deben ser implementados por las clases y dos propiedades que también deben definirlos:

```
interface Figura {
  superficie: number;
  perimetro: number;
```

```
  calcularSuperficie(): number;
  calcularPerimetro(): number;
}
```

La clase Cuadrado indica que implementa la interface Figura, esto hace necesario que se implementen los métodos calcularSuperficie y calcularPerimetro, y las dos propiedades:

```
class Cuadrado implements Figura {
  superficie: number;
  perimetro: number;
  constructor(private lado:number) {
    this.superficie = this.calcularSuperficie();
    this.perimetro = this.calcularPerimetro();
  }

  calcularSuperficie(): number {
    return this.lado * this.lado;
  }

  calcularPerimetro(): number {
    return this.lado * 4;
  }
}
```

La clase Cuadrado tiene una propiedad llamada lado que la recibe el constructor.

De forma similar la clase Rectangulo implementa la interface Figura:

```
class Rectangulo implements Figura {
  superficie: number;
  perimetro: number;
  constructor(private ladoMayor:number, private ladoMenor:number) {
    this.superficie = this.calcularSuperficie();
```

```
    this.perimetro = this.calcularPerimetro();
  }

  calcularSuperficie(): number {
    return this.ladoMayor * this.ladoMenor;
  }

  calcularPerimetro(): number {
    return (this.ladoMayor * 2) + (this.ladoMenor * 2);
  }
}
```

Finalmente definimos un objeto de la clase Cuadrado y otro de la clase Rectangulo, luego llamamos a los métodos calcularPerimetro y calcularSuperficie para cada objeto:

```
let cuadrado1: Cuadrado;
cuadrado1 = new Cuadrado(10);
console.log(`Perimetro del cuadrado : ${cuadrado1.calcularPerimetro()}`);
console.log(`Superficie del cuadrado : ${cuadrado1.calcularSuperficie()}`);
let rectangulo1: Rectangulo;
rectangulo1 = new Rectangulo(10, 5);
console.log(`Perimetro del rectangulo : ${rectangulo1.calcularPerimetro()}`);
console.log(`Superficie del cuadrado : ${rectangulo1.calcularSuperficie()}`);
```

Las interfaces exige que una clase siga las especificaciones de la misma y se implementen algoritmos más robustos. En nuestro ejemplo tanto la clase Rectangulo como Cuadrado tienen una forma similar de trabajar gracias a que implementan la interfaz Figura.

PARÁMETROS DE TIPO INTERFACE.

Un método o función puede recibir como parámetro una interface. Luego le podemos pasar objetos de distintas clases que implementan dicha interface:

```
interface Figura {
  superficie: number;
  perimetro: number;
  calcularSuperficie(): number;
  calcularPerimetro(): number;
}


class Cuadrado implements Figura {
  superficie: number;
  perimetro: number;
  constructor(private lado:number) {
    this.superficie = this.calcularSuperficie();
    this.perimetro = this.calcularPerimetro();
  }

  calcularSuperficie(): number {
    return this.lado * this.lado;
  }

  calcularPerimetro(): number {
    return this.lado * 4;
  }
}


class Rectangulo implements Figura {
  superficie: number;
  perimetro: number;
```

```
  constructor(private ladoMayor:number, private ladoMenor:number) {
    this.superficie = this.calcularSuperficie();
    this.perimetro = this.calcularPerimetro();
  }

  calcularSuperficie(): number {
    return this.ladoMayor * this.ladoMenor;
  }

  calcularPerimetro(): number {
    return (this.ladoMayor * 2) + (this.ladoMenor * 2);
  }
}

function imprimir(fig: Figura) {
  console.log(`Perimetro: ${fig.calcularPerimetro()}`);
  console.log(`Superficie: ${fig.calcularSuperficie()}`);
}

let cuadrado1: Cuadrado;
cuadrado1 = new Cuadrado(10);
console.log('Datos del cuadrado');
imprimir(cuadrado1);
let rectangulo1: Rectangulo;
rectangulo1 = new Rectangulo(10, 5);
console.log('Datos del rectángulo');
imprimir(rectangulo1);
```

La función imprimir recibe como parámetro fig que es de tipo Figura:

```
function imprimir(fig: Figura) {
  console.log(`Perimetro: ${fig.calcularPerimetro()}`);
  console.log(`Superficie: ${fig.calcularSuperficie()}`);
}
```

Podemos luego llamar a la función imprimir pasando tanto objetos de la clase Cuadrado como Rectangulo:

```
imprimir(cuadrado1);


imprimir(rectangulo1);
```

Es importante notar que solo podemos acceder a los métodos y propiedades definidos en la interface y no a propiedades y métodos propios de cada clase.

CREACIÓN DE OBJETOS A PARTIR DE UNA INTERFACE.

TypeScript permite crear objetos a partir de una interfaz. La sintaxis para dicha creación es:

```
interface Punto {
  x: number;
  y: number;
}


let punto1: Punto;
punto1 = {x:10, y:20};
console.log(punto1);
```

No podemos utilizar el operador new para la creación del objeto.

Podemos definir la variable e inmediatamente iniciarla:

```
let punto1: Punto = {x:10, y:20};
```

PROPIEDADES OPCIONALES.

Una interface puede definir propiedades opcionales que luego la clase que la implementa puede o no definirlas. Se utiliza la misma sintaxis de los parámetros opcionales, es decir se le agrega el caracter '?' al final del nombre de la propiedad.

```
interface Punto {
  x: number;
  y: number;
  z?: number;
}

let puntoPlano: Punto = {x:10, y:20};
console.log(puntoPlano);
let puntoEspacio: Punto = {x:10, y:20, z:70};
console.log(puntoEspacio);
```

Como vemos el objeto 'puntoPlano' solo implementa las propiedades 'x' e 'y'.

Se produce un error en tiempo de compilación si no implementamos todas las propiedades obligatorias, por ejemplo:

```
let puntoPlano: Punto = {x:10};
```

Esta línea genera un error ya que solo se define la propiedad 'x' y falta definir la propiedad 'y'.

HERENCIA DE INTERFACES.

TypeScript permite que una interface herede de otra:

```
interface Punto {
  x: number;
  y: number;
}

interface Punto3D extends Punto {
  z: number;
}
```

```
let punto1: Punto = {x:10, y:20};
let punto2: Punto3D = {x:23, y:13, z:12};
console.log(punto1);
console.log(punto2);
```

IV. PROCEDIMIENTO

CRUD ANGULAR SIN BASE DE DATOS

DISTINTAS FORMAS DE INSTALAR BOOTSTRAP.

Bootstrap es un framework de CSS que nos facilita y estandariza el desarrollo de sitios web.

Veremos que hay varias maneras de utilizar el framework original de Bootstrap y en conceptos futuros veremos cómo instalar versiones de Bootstrap adaptadas directamente al ambiente de Angular.

Primera forma:

Crearemos un proyecto nuevo para probar las distintas maneras de acceder a Bootstrap desde Angular:

1. Crear el proyecto en angular.

```
ng new crudAngular
```

La forma más sencilla es utilizar un **CDN** donde se encuentre localizado los archivos *.css y *.js del framework de Bootstrap y sus dependencias **(JQuery y Popper).**

Debemos modificar el archivo 'index.html' disponiendo los enlaces a los archivos respectivos y con eso ya tenemos acceso al framework de Bootstrap en todo el proyecto:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Proyecto038</title>
  <base href="/">
```

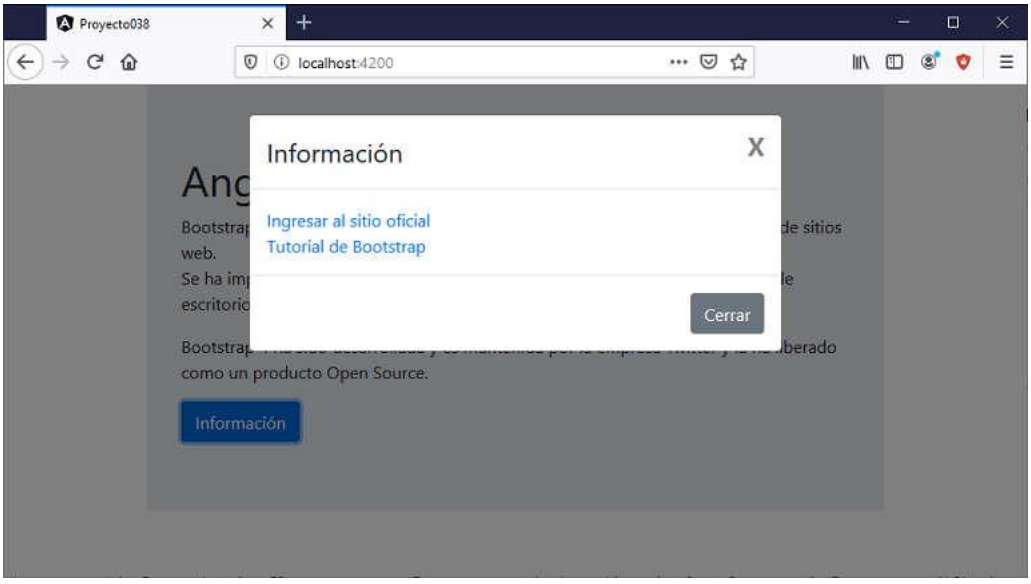
```
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <link rel="icon" type="image/x-icon" href="favicon.ico">
 <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
</head>
<body>
 <app-root></app-root>
 <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"></script>
 <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
 <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"></scrip>
</body>
</html>
```

Modifiquemos la componente que se ha creado por defecto y hagamos uso de las clases definidas en la librería de Bootstrap:

```
<div class="container">
 <div class="jumbotron">
  <h1>Angular con Bootstrap</h1>
  <p>Bootstrap 4 es un framework de CSS que nos facilita y estandariza el desarrollo de sitios web.<br>
    Se ha implementado pensando que se adapte tanto a las pantallas de equipos de
    escritorio como a móviles y tablets.</p>
  <p>Bootstrap 4 ha sido desarrollada y es mantenida por la empresa Twitter y la ha liberado como un producto Open
    Source.</p>
  <button type="button" class="btn btn-primary" data-toggle="modal" data-target="#dialogo1">Información</button>
  <div class="modal fade" id="dialogo1">
   <div class="modal-dialog">
    <div class="modal-content">
     <div class="modal-header">
```

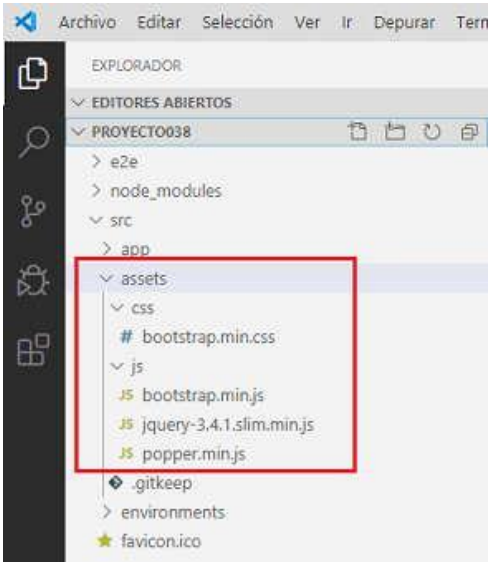
```
        <h4 class="modal-title">Información</h4>
        <button type="button" class="close" data-dismiss="modal">X</button>
      </div>
      <!-- cuerpo del diálogo -->
      <div class="modal-body">
        <a href="https://getbootstrap.com/">Ingresar al sitio oficial</a><br>
        <a href="https://tutorialesprogramacionya.com/bootstrap4ya">Tutorial de Bootstrap</a>
      </div>
      <!-- pie del diálogo -->
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-dismiss="modal">Cerrar</button>
      </div>
    </div>
  </div>
</div>
</div>
```

Si ejecutamos la aplicación podemos ver que tenemos disponible la librería *.css de Bootstrap como también acceso a las funcionalidades de Javascript de: JQuery, popper y Bootstrap en si mismo (de todos los elementos de Bootstrap que requieren Javascript):



Segunda forma:

Si queremos tener los archivos de Bootstrap, JQuery y Popper en nuestro servidor y no en un CDN tenemos como primera alternativa descargarlos y copiar los archivos a la carpeta 'assets', luego tenemos los archivos en forma local:

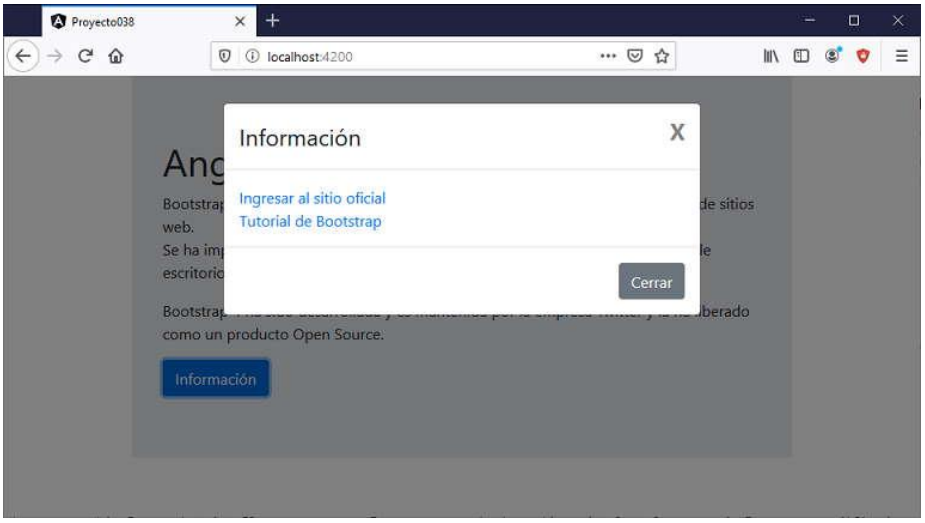


Solo nos falta cambiar las url en el archivo 'index.html':

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Proyecto038</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="assets/css/bootstrap.min.css">
</head>
<body>
  <app-root></app-root>
```

```
  <script src="assets/js/jquery-3.4.1.slim.min.js"></script>
  <script src="assets/js/popper.min.js"></script>
  <script src="assets/js/bootstrap.min.js"></script>
</body>
</html>
```

Ya podemos ejecutar y tener el mismo resultado:



Tercera forma:

instalar Bootstrap es empleando el gestor de paquetes de Node.Js Desde la línea de comandos, en la carpeta raíz de nuestro proyecto procedemos a instalar los tres paquetes:

```
npm install bootstrap --save
npm install jquery --save
npm install  @popperjs/core --save
```

La opción --save hace que npm incluya el paquete dentro de la sección de dependencies del archivo package.json en forma automática, lo que evita que tengamos que escribirlo en forma manual.

Si abrimos el archivo 'package.json' podemos ver que se han añadido las tres dependencias en nuestro proyecto:

```
{
  "name": "proyecto038",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "~9.0.0",
    "@angular/common": "~9.0.0",
    "@angular/compiler": "~9.0.0",
    "@angular/core": "~9.0.0",
    "@angular/forms": "~9.0.0",
    "@angular/platform-browser": "~9.0.0",
    "@angular/platform-browser-dynamic": "~9.0.0",
    "@angular/router": "~9.0.0",
    "@popperjs/core": "^2.0.6",
    "bootstrap": "^4.4.1",
    "jquery": "^3.4.1",
    "rxjs": "~6.5.4",
    "tslib": "^1.10.0",
    "zone.js": "~0.10.2"
  },
```

```
  "devDependencies": {
    "@angular-devkit/build-angular": "~0.900.1",
    "@angular/cli": "~9.0.1",
    "@angular/compiler-cli": "~9.0.0",
    "@angular/language-service": "~9.0.0",
    "@types/node": "^12.11.1",
    "@types/jasmine": "~3.5.0",
    "@types/jasminewd2": "~2.0.3",
    "codelyzer": "^5.1.2",
    "jasmine-core": "~3.5.0",
    "jasmine-spec-reporter": "~4.2.1",
    "karma": "~4.3.0",
    "karma-chrome-launcher": "~3.1.0",
    "karma-coverage-istanbul-reporter": "~2.1.0",
    "karma-jasmine": "~2.0.1",
    "karma-jasmine-html-reporter": "^1.4.2",
    "protractor": "~5.4.3",
    "ts-node": "~8.3.0",
    "tslint": "~5.18.0",
    "typescript": "~3.7.5"
  }
}
```

Cuarta forma:

Como último paso para poder utilizar Bootstrap en nuestro proyecto, debemos modificar el archivo 'angular.json' donde debemos especificar el path de los 4 archivos (en la propiedad 'build', en 'styles' indicamos el path donde se encuentra el archivo *.css y en 'script' los tres archivos respectivos, es importante el orden en que se importan los archivos Javascript):

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "proyecto038": {
      "projectType": "application",
      "schematics": {},
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/proyecto038",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "aot": true,
            "assets": [
              "src/favicon.ico",
              "src/assets"
```

```
          ],
          "styles": [
            "node_modules/bootstrap/dist/css/bootstrap.min.css",
            "src/styles.css"
          ],
          "scripts": [
            "node_modules/jquery/dist/jquery.min.js",
            "node_modules/@popperjs/core/dist/umd/popper.min.js",
            "node_modules/bootstrap/dist/js/bootstrap.min.js"
          ]
        },
        "configurations": {
          "production": {
            "fileReplacements": [
              {
                "replace": "src/environments/environment.ts",
                "with": "src/environments/environment.prod.ts"
              }
            ],
            "optimization": true,
            "outputHashing": "all",
            "sourceMap": false,
            "extractCss": true,
            "namedChunks": false,
            "extractLicenses": true,
            "vendorChunk": false,
            "buildOptimizer": true,
            "budgets": [
              {
```

```
            "type": "initial",
            "maximumWarning": "2mb",
            "maximumError": "5mb"
          },
          {
            "type": "anyComponentStyle",
            "maximumWarning": "6kb",
            "maximumError": "10kb"
          }
        ]
      }
    }
  },
  "serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
    "options": {
      "browserTarget": "proyecto038:build"
    },
    "configurations": {
      "production": {
        "browserTarget": "proyecto038:build:production"
      }
    }
  },
  "extract-i18n": {
    "builder": "@angular-devkit/build-angular:extract-i18n",
    "options": {
      "browserTarget": "proyecto038:build"
    }
```

```
    },
    "test": {
      "builder": "@angular-devkit/build-angular:karma",
      "options": {
        "main": "src/test.ts",
        "polyfills": "src/polyfills.ts",
        "tsConfig": "tsconfig.spec.json",
        "karmaConfig": "karma.conf.js",
        "assets": [
          "src/favicon.ico",
          "src/assets"
        ],
        "styles": [
          "src/styles.css"
        ],
        "scripts": []
      }
    },
    "lint": {
      "builder": "@angular-devkit/build-angular:tslint",
      "options": {
        "tsConfig": [
          "tsconfig.app.json",
          "tsconfig.spec.json",
          "e2e/tsconfig.json"
        ],
        "exclude": [
          "**/node_modules/**"
        ]
```

```
        }
       },
       "e2e": {
        "builder": "@angular-devkit/build-angular:protractor",
        "options": {
         "protractorConfig": "e2e/protractor.conf.js",
         "devServerTarget": "proyecto038:serve"
        },
        "configurations": {
         "production": {
          "devServerTarget": "proyecto038:serve:production"
         }
        }
       }
     }
   },
  },
  "defaultProject": "proyecto038",
  "cli": {
    "analytics": false
  }
}
```

Con este método no debemos modificar nada el archivo 'index.html':

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Proyecto038</title>
  <base href="/">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

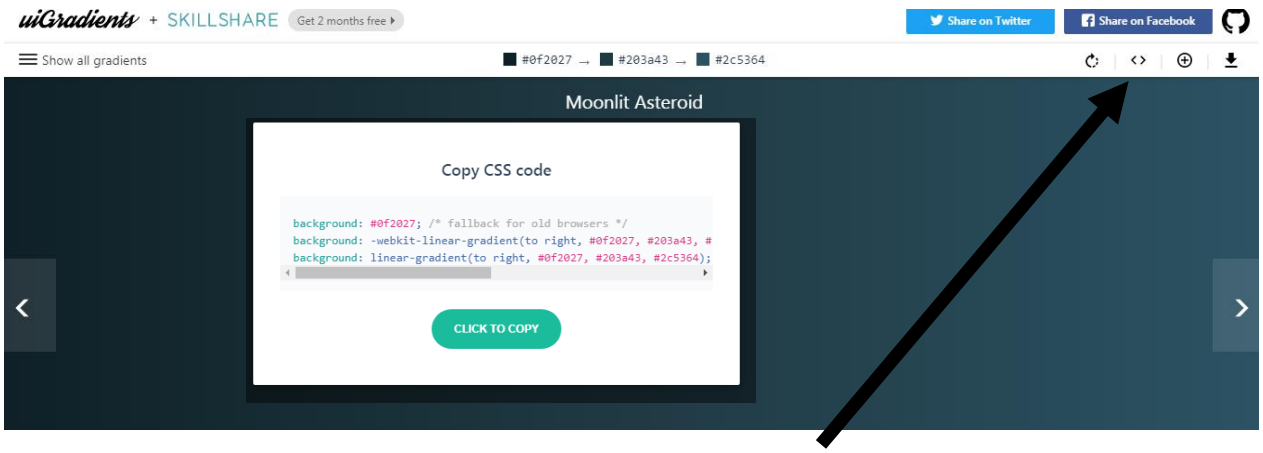
Hemos presentado en este concepto las cuatro formas de integrar el Bootstrap dentro de un proyecto Angular, cual aplicar será decisión personal.

Primer paso fue configurar Bootstrap en nuevo proyecto de crudAngular, cual opción es la mejor, la que sea mas factible implementar.

```
ng new crudAngular
```

2. Agregar fondo a la página, visitando la siguiente dirección y seleccionar el fondo que se desea implementar.

https://uigradients.com/#MoonlitAsteroid



Cuando se tenga el fondo seleccionado, se generar el código css, para llevarlo al proyecto de angular

src/styles.css

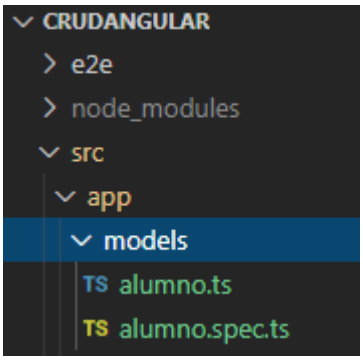
```
# styles.css > body
You, a few seconds ago | 1 author (You)
/* You can add global styles to this file, and also import other style files */

/* @import "~bootstrap/dist/css/bootstrap.css"; */

body {
    background: #e1eec3;
    /* fallback for old browsers */
    background: -webkit-linear-gradient(to right, #f05053, #e1eec3);
    /* Chrome 10-25, Safari 5.1-6 */
    background: linear-gradient(to right, #f05053, #e1eec3);
    /* W3C, IE 10+/ Edge, Firefox 16+, Chrome 26+, Opera 12+, Safari 7+ */
    height: 100vh;
}       You, 3 minutes ago • Uncommitted changes
```

3. En el archivo **app.module.ts ,** configuramos los siguientes :

```
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

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


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

4. Agregar una nueva clase, ir a la terminal de VSC:

```
f:\angularya\proyecto16\> ng g cl models/alumno
```



Dentro de la clase Alumno hacer los siguiente:

```
export class Alumno {
    id: number= 0;
    name: string = '';
    lastname: string = '';
    age: number = 0;
}
```

5. En el archivo **app.component.ts**, digitamos lo siguiente :

```
import { Component } from '@angular/core';
import { Alumno } from './models/alumno';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'CrudAngular';

// arreglo del tipo Alumno, que tiene 3 registro almacenados
  alumnoArray: Alumno[] = [
    {id:1, name:"Alex", lastname:"Campos",age:35},
```

```
    {id:2, name:"Maria",lastname:"Lopez",age:20},
    {id:3, name:"Juan", lastname:"Castro",age:25}
  ]

//atributo selecAlumno del tipo Alumno, por lo tanto, puede almacenar un objeto
  selectedAlumno: Alumno = {id:0, name:'', lastname:'', age:0 };

//un método que no retorna nada "void", recibe como parámetro una variable del
//tipo Alumno, para ser asignada al atributo selectAlumno y poder ser mostrado
// en pantalla.
  openForEdit(alumno: Alumno): void
  {
    this.selectedAlumno = alumno;
  }

//método que no retorna nada "void", NO recibe parámetro, pero realiza dos
//operaciones agregar / editar, si no hay registro seleccionado se guarda,
//de lo contrario limpia el atributo selectedAlumno en pantalla. [(ngModel)]
  addOrEdit(): void
  {
    if(this.selectedAlumno.id === 0) // INSERT
    {
      this.selectedAlumno.id = this.alumnoArray.length + 1;
      this.alumnoArray.push(this.selectedAlumno);
    }
    this.selectedAlumno = {id:0, name: '', lastname: '', age:0 };
  }

//método que no retorna nada "void", NO recibe parámetro, elimina del arreglo el
//registro, pero antes muestra en pantalla un confirmar, se recorre el arreglo
//realizando un "filter" para no almacenar el registro seleccionado en el nuevo
//arreglo "alumnoArray" , por eso ocupados el operador "!=" y luego limpiamos
//el registro seleccionado.
  delete(): void
  {
    if(confirm('¿Esta seguro de elimiar el Registro?'))
    {
      this.alumnoArray = this.alumnoArray.filter(x => x != this.selectedAlumno);
      this.selectedAlumno = {id:0, name: '', lastname: '', age:0 };
    }
  }

}
```

Javascript: filtrar elementos de un array con .filter()

El método **filter()** nos permite filtrar solo los elementos que deseamos (según ciertos criterios) y devolverlos en un nuevo array.

6. En el archivo **app.component.html**, digitamos lo siguiente:

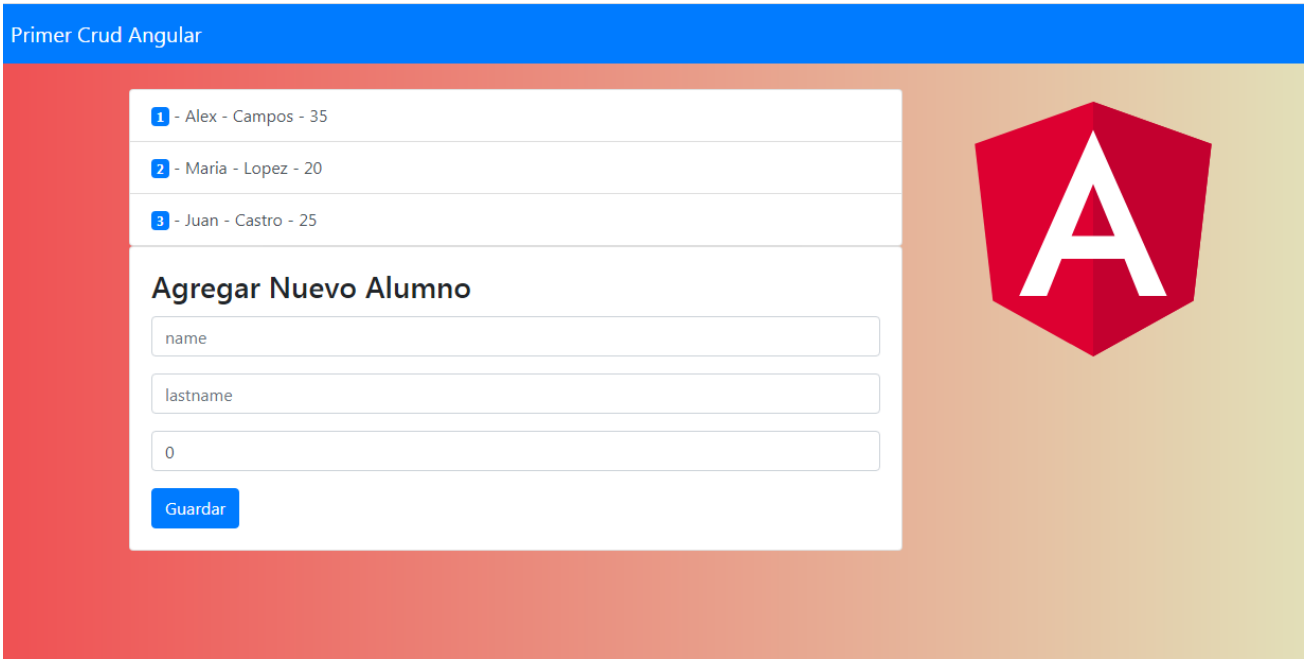
La mayor parte del código ya se ha trabajado en guias anteriores**, (click) , Interpolacion, [(ngModel)].** Lo nuevo es **[class.active] ,** que lo que realiza en poner activo con un color diferente el registro seleccionado.

```
<!-- margin-right property -->
<style>
.btn-space {
margin-right: 20px;
}
</style>


<nav class="navbar navbar-dark bg-primary">
<a class="navbar-brand" href="/">
Primer Crud Angular
</a>
</nav>

<div class="container">
<div class="row">
<div class="col-md-8 mt-4">
<! -- mt-4 espacio en blanco vertical -->
<ul class="list-group">
<li *ngFor="let alumno of alumnoArray" class="list-group-item list-group-item-
action"
(click)="openForEdit(alumno)"
[class.active]="alumno == selectedAlumno">
<span class="badge badge-primary">{{alumno.id}}</span>
- {{alumno.name}}
- {{alumno.lastname}}
- {{alumno.age}}
</li>
</ul>

<div class="card">
<div class="card-body">
<div class="card-title">
<h3>{{selectedAlumno.id == 0 ? "Agregar Nuevo Alumno" : "Actualizar Alumno"}}</h3>
```

```
</div>
<div class="form-group">
<input type="text" [(ngModel)]="selectedAlumno.name" placeholder="name"
class="form-control">
</div>
<div class="form-group">
<input type="text" [(ngModel)]="selectedAlumno.lastname" placeholder="lastname"
class="form-control">
</div>
<div class="form-group">
<input type="number" [(ngModel)]="selectedAlumno.age" placeholder="age"
class="form-control">
</div>
<button (click)="addOrEdit()" class="btn btn-primary btn-space">
Guardar
</button>
<button *ngIf="selectedAlumno.id !== 0" class="btn btn-danger" (click)="delete()">
Eliminar
</button>
</div>
</div>
</div>
<div class="col-md-4">
<! -- md-4 espacio en blanco vertical -->
<img width="300" alt="Angular Logo"
src="
HZpZXdCb3g9IjAgMCAyNTAgMjUwIj4KICAgIDxwYXRoIGZpbGw9IiNERDAwMzEiIGQ9Ik0xMjUgMzBMMzEuO
SA2My4ybDE0LjIgMTIzLjFMMTI1IDIzMGw3OC45LTQzLjcgMTQuMi0xMjMuMXoiIC8+CiAgICA8cGF0aCBma
WxsPSIjQzMwMDJGIiBkPSJNMTI1IDMwdjIyLjItLjFWMjMwbDc4LjktNDMuNyAxNC4yLTEyMy4xTDEyNSAzM
HoiIC8+CiAgICA8cGF0aCAgZmlsbD0iI0ZGRkZGRiIgZD0iTTEyNSA1Mi4xTDY2LjggMTgyLjZoMTgyLjZaM
S43LTI5LjJoNDkuNGwxMS43IDI5LjJIMTgzLjgyLjZoMjEuN2wxMS43IDI5LjJIMTgzLjgyLjZoMjEuN2wxM
jl6IiAvPgogICAgPC9zdmc+Cg==">
</div>
</div>
</div>
```

7. Como último paso hacer la prueba de la aplicación:

```
ng serve -o
```



El segundo ejemplo lleva el trabajo con base de datos y php

Comunicación entre Angular y PHP

Hemos dicho que Angular es un framework para el desarrollo de aplicaciones web de una sola página. Una situación muy común es tener que almacenar en un servidor de internet los datos que se ingresan en la aplicación Angular.

Existen muchas tecnologías para procesar los datos que envía y recibe la aplicación Angular, una de la más extendidas en el lenguaje PHP y mediante este acceder a una base de datos MySql.

En este concepto dejaremos en forma muy clara todos los pasos que debemos desarrollar tanto en el cliente (aplicación angular) como en el servidor (aplicación PHP y MySql)

PROBLEMA

Confeccionar una aplicación que permita administrar una lista de artículos, cada artículo almacena el código, descripción y precio. Se debe poder agregar, borrar y modificar los datos de un artículo almacenados en una base de datos MySQL y accedida con un programa en PHP.

Desde la línea de comandos de Node.js procedemos a crear:

```
f:\angularya> ng new crudangularphp
```

Recordemos que la propuesta del framework de Angular es delegar todas las responsabilidades de acceso a datos (peticiones y envío de datos) y lógica de negocios en otras clases que colaboran con las componentes. Estas clases en Angular se las llama servicios.

Crearemos el servicio 'articulos' para ello utilizamos Angular CLI:

```
f:\angularya\proyecto16\> ng generate service articulos
```

Con el comando anterior estamos creando la clase 'ArticulosService'

El código que debemos guardar en el archivo 'articulos.service.ts' es:

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

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

url='http://scratchya.com.ar/angular/problema016/'; // disponer url de su servidor q
ue tiene las páginas PHP

constructor(private http: HttpClient) { }

recuperarTodos() {
return this.http.get(`${this.url}recuperartodos.php`);
}

alta(articulo) {
return this.http.post(`${this.url}alta.php`, JSON.stringify(articulo));
}

baja(codigo:number) {
return this.http.get(`${this.url}baja.php?codigo=${codigo}`);
}

seleccionar(codigo:number) {
return this.http.get(`${this.url}seleccionar.php?codigo=${codigo}`);
}
```

```
modificacion(articulo) {
return this.http.post(`${this.url}modificacion.php`, JSON.stringify(articulo));
}
}
```

Presentaremos primero todos los archivos y luego explicaremos como se relacionan.

El archivo 'app.module.ts' se modifica con el siguiente código:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import {HttpClientModule} from '@angular/common/http';

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

Nuestra única componente debe implementar el siguiente código en el archivo 'app.component.ts':

```
import { Component } from '@angular/core';
import { Alumno } from './models/alumno';
import { ArticulosService } from './articulos.service';
import { HttpClient } from '@angular/common/http';


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'CrudAngular';
```

```
  // CRUD ANGULAR PHP Y MYSQL

  articulos = null;

  art = {
    codigo: 0,
    descripcion: null,
    precio: null
  }

  constructor(private articulosServicio: ArticulosService) { }

  ngOnInit() {
    this.recuperarTodos();
  }

  recuperarTodos() {
    this.articulosServicio.recuperarTodos().subscribe(result => this.articulos = res
ult);
  }

  alta() {
    this.articulosServicio.alta(this.art).subscribe(datos => {
      if (datos['resultado'] == 'OK') {
        alert(datos['mensaje']);
        this.recuperarTodos();
        this.art = {codigo: 0, descripcion: null, precio: null};
      }
    });
  }

  baja(codigo) {
    if (confirm('¿Esta seguro de elimiar el Registro?')) {
      this.articulosServicio.baja(codigo).subscribe(datos => {
        if (datos['resultado'] == 'OK') {
          alert(datos['mensaje']);
          this.recuperarTodos();
        }
      });
    }
  }

  modificacion() {
    this.articulosServicio.modificacion(this.art).subscribe(datos => {
```

```
      if (datos['resultado'] == 'OK') {
        alert(datos['mensaje']);
        this.recuperarTodos();
        this.art = {codigo: 0, descripcion: null, precio: null};
      }
    });
  }

  seleccionar(codigo) {
    this.articulosServicio.seleccionar(codigo).subscribe(result => this.art = result
[0]);
  }

  hayRegistros() {
    return true;
  }


}
```

Y el archivo 'app.component.html' con:

```
<! -- CRUD ANGULAR PHP Y MYSQL -->

    <div class="container">
        <h2>CRUD ANGULAR PHP Y MYSQL</h2>
        <div class="col-md-8 mt-4">
            <div class="card">
                <div class="card-body">
                    <div class="container">
                        <h2>Administración de artículos</h2>
                        <table class="table table-
hover" *ngIf="hayRegistros(); else sinarticulos">
                            <thead>
                                <tr>
                                    <th>Codigo</th>
                                    <th>Descripcion</th>
                                    <th>Precio</th>
                                    <th>Borrar</th>
                                    <th>Seleccionar</th>
                                </tr>
                            </thead>
                            <tbody>
                                <tr *ngFor="let art of articulos">
                                    <td>{{art.codigo}}</td>
```

```
                                    <td>{{art.descripcion}}</td>
                                    <td>{{art.precio}}</td>
                                    <td><button (click)="baja(art.codigo)" class="bt
n btn-danger">Borrar</button></td>
                                    <td><button (click)="seleccionar(art.codigo)"
                                            class="btn btn-
warning">Seleccionar</button></td>
                                </tr>
                            </tbody>
                        </table>
                    </div>

                    <ng-template #sinarticulos>
                        <p>No hay articulos.</p>
                    </ng-template>
                    <div>
                        <div class="form-group">
                            <input type="text" [(ngModel)]="art.descripcion" placeho
lder="descripcion"
                                class="form-control" />
                        </div>
                        <div class="form-group">
                            <input type="number" [(ngModel)]="art.precio" placeholde
r="precio" class="form-control" />
                        </div>
                        <button (click)="alta()" *ngIf="art.codigo == 0"
                            class="btn btn-primary btn-space">Agregar</button>
                        <button (click)="modificacion()" *ngIf="art.codigo !== 0"
                            class="btn btn-warning">Modificar</button>
                    </div>
                </div>
            </div>
        </div>
    </div>
```

Todos los archivos presentados son los necesarios en Angular, veamos ahora que tenemos en PHP y MySQL.

Primero debemos crear una base de datos en MySQL llamada 'bd1' y crear la siguiente tabla:

```
CREATE TABLE articulos (
    codigo int AUTO_INCREMENT,
    descripcion varchar(50),
    precio float,
    PRIMARY KEY (codigo)
  )
```

Tenemos una serie de archivos PHP que reciben datos en formato JSON y retornan también un JSON.

El archivo **'recuperartodos.php'** retorna en formato JSON todos los artículos:

```
<?php
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-
Type, Accept");

require("conexion.php");
$con=retornarConexion();

$registros=mysqli_query($con,"select codigo, descripcion, precio from articulos");
$vec=[];
while ($reg=mysqli_fetch_array($registros))
{
$vec[]=$reg;
}

$cad=json_encode($vec);
echo $cad;
//header('Content-Type: application/json');
?>
```

El archivo 'alta.php' recibe los datos en formato JSON y los almacena en la tabla:

```
<?php
  header('Access-Control-Allow-Origin: *');
  header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-
Type, Accept");

  $json = file_get_contents('php://input');

  $params = json_decode($json);

  require("conexion.php");
```

```
  $con=retornarConexion();



  mysqli_query($con,"insert into articulos(descripcion,precio) values
                  ('$params->descripcion',$params->precio)");


  class Result {}

  $response = new Result();
  $response->resultado = 'OK';
  $response->mensaje = 'datos grabados';

  header('Content-Type: application/json');
  echo json_encode($response);
?>
```

El archivo 'baja.php':

```
<?php
  header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-
Type, Accept");

require("conexion.php");
$con = retornarConexion();

mysqli_query($con, "delete from articulos where codigo=$_GET[codigo]");


class Result { }

$response = new Result();
$response -> resultado = 'OK';
$response -> mensaje = 'articulo borrado';

header('Content-Type: application/json');
echo json_encode($response);
?>
```

El archivo 'modificacion.php':

```
<?php
  header('Access-Control-Allow-Origin: *');
  header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-
Type, Accept");

  $json = file_get_contents('php://input');

  $params = json_decode($json);

  require("conexion.php");
  $con=retornarConexion();


  mysqli_query($con,"update articulos set descripcion='$params->descripcion',
                                          precio=$params->precio
                                          where codigo=$params->codigo");


  class Result {}

  $response = new Result();
  $response->resultado = 'OK';
  $response->mensaje = 'datos modificados';

  header('Content-Type: application/json');
  echo json_encode($response);
?>
```

El archivo 'seleccionar.php':

```
<?php
  header('Access-Control-Allow-Origin: *');
  header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-
Type, Accept");

  require("conexion.php");
  $con=retornarConexion();

  $registros=mysqli_query($con,"select codigo, descripcion, precio from articulos wh
ere codigo=$_GET[codigo]");

  if ($reg=mysqli_fetch_array($registros))
```

```
  {
    $vec[]=$reg;
  }

  $cad=json_encode($vec);
  echo $cad;
  //header('Content-Type: application/json');
?>
```

El archivo 'conexion.php':

```
<?php
function retornarConexion() {
  $con=mysqli_connect("localhost","root","","bd1");
  return $con;
}
?>
```

El resultado de ejecutar esta aplicación en el navegador, teniendo en funcionamiento el servidor con PHP y MySQL es:

Explicación

Ahora veremos como funciona esta aplicación cliente/servidor implementada con Angular en el cliente y PHP en el servidor.

Recuperación de todos los registros

Inmediatamente se inicia la aplicación Angular se crea el objeto de la clase 'AppComponent' (nuestra única componente), en esta clase debe llegar al constructor el objeto de la clase 'ArticulosService':

```
constructor(private articulosServicio: ArticulosService) {}
```

La clase ArticulosService está creada en el archivo 'articulos.service.ts':

```
export class ArticulosService {
...
}
```

En ningún momento creamos un objeto de ésta clase, sino el framework de Angular se encarga de esto.

Volviendo al archivo app.component.ts en el método ngOnInit procedemos a llamar al método recuperarTodos:

```
ngOnInit() {
this.recuperarTodos();
}
```

recuperarTodos tiene por objetivo utilizar el 'servicio' que llega al constructor para llamar al método 'recuperarTodos' del servicio propiamente dicho:

```
recuperarTodos() {
this.articulosServicio.recuperarTodos().subscribe(result => this.articulos = result);
}
```

Si vemos ahora el método 'recuperarTodos' de la clase 'ArticulosService', es el que tiene la responsabilidad de hacer una petición al servidor:

```
recuperarTodos() {
return this.http.get(`${this.url}recuperartodos.php`);
}
```

El método 'recuperarTodos' de la clase 'ArticulosService' retorna un objeto de la clase 'HttpClient'.

Ahora debemos entender porque podemos llamar al método 'suscribe':

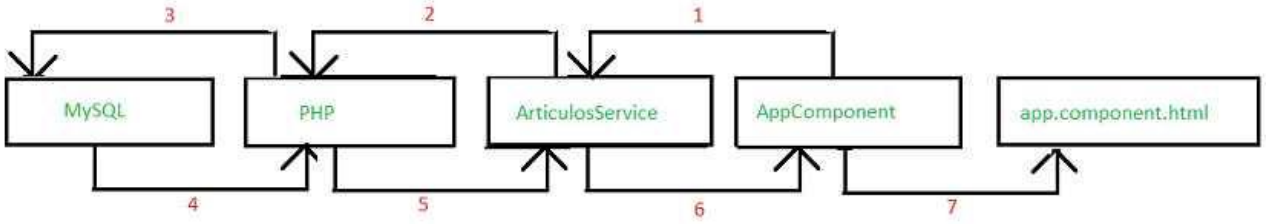
```
recuperarTodos() {
this.articulosServicio.recuperarTodos().subscribe(result => this.articulos = result);
}
```

El método 'suscribe' recibe los resultados y procedemos a asignar a la propiedad 'articulos', con esto, Angular se encarga de actualizar la página con todos los artículos recuperados. El proceso de actualizar la página sucede en el archivo 'app.component.html':

```
<div>
<h1>Administración de artículos</h1>
<table border="1" *ngIf="hayRegistros(); else sinarticulos">
<tr>
<td>Codigo</td><td>Descripcion</td><td>Precio</td><td>Borrar</td><td>Seleccionar</td>
</tr>
<tr *ngFor="let art of articulos">
<td>{{art.codigo}}</td>
<td>{{art.descripcion}}</td>
<td>{{art.precio}}</td>
<td><button (click)="baja(art.codigo)">Borrar?</button></td>
<td><button (click)="seleccionar(art.codigo)">Seleccionar</button></td>
</tr>
</table>
<ng-template #sinarticulos><p>No hay articulos.</p></ng-template>
<div>
<p>
descripcion:<input type="text" [(ngModel)]="art.descripcion" />
</p>
<p>
precio:<input type="number" [(ngModel)]="art.precio" />
```

```
</p>
<p><button (click)="alta()">Agregar</button>
<button (click)="modificacion()">Modificar</button></p>
</div>
</div>
```

El flujo de la información lo podemos representar con el siguiente esquema:



Alta

Veamos ahora los pasos cuando se agrega una fila en la tabla 'articulos'. Todo comienza cuando el operador presiona el botón de 'Agregar':

La etiqueta button tiene enlazado la llamada al método 'alta':

```
<p><button (click)="alta()">Agregar</button>
```

El método 'alta' se encuentra codificado en el archivo 'app.component.ts' dentro de la clase 'AppComponent':

```
alta() {
this.articulosServicio.alta(this.art).subscribe(datos => {
if (datos['resultado']=='OK') {
alert(datos['mensaje']);
this.recuperarTodos();
}
});
}
```

En este método procedemos a llamar al método 'alta' de la clase 'ArticulosService' y se le pasa como parámetro la propiedad 'art' que almacena la descripción y precio del artículo que el operador acaba de ingresar por teclado.

El método 'alta' de la clase 'ArticulosService' hace la llamada al servidor mediante el objeto 'http' de la clase 'HttpClient'. Se utiliza el método 'post' ya que se enviarán datos al servidor:

```
alta(articulo) {
return this.http.post(`${this.url}alta.php`, JSON.stringify(articulo));
}
```

Ahora se ejecuta el programa PHP definido en el archivo 'alta.php' donde procedemos a efectuar el insert en la tabla de MySQL:

```
<?php
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

$json = file_get_contents('php://input');

$params = json_decode($json);

require("conexion.php");
$con=retornarConexion();

mysqli_query($con,"insert into articulos(descripcion,precio) values
('$params->descripcion',$params->precio)");



class Result {}

$response = new Result();
$response->resultado = 'OK';
$response->mensaje = 'datos grabados';
```

```
header('Content-Type: application/json');
echo json_encode($response);
?>
```

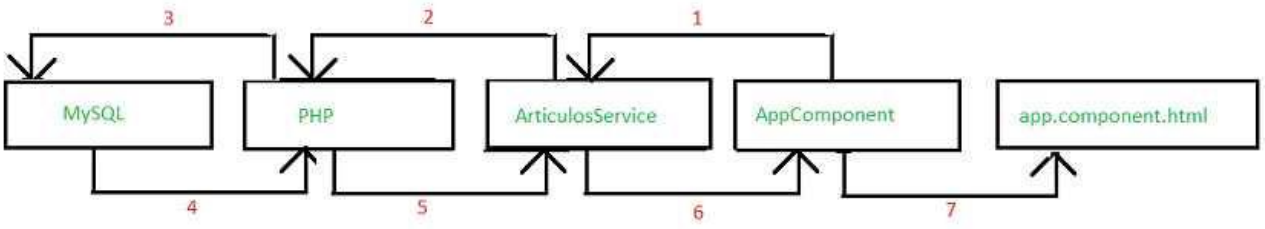
También dentro del programa PHP procedemos a retornar en formato JSON que la operación se efectuó en forma correcta.

En el método 'alta' de la clase 'AppComponent' que ya vimos, recibe los datos de la respuesta JSon, mostrando un mensaje si la carga se efectuó correctamente:

```
alta() {
this.articulosServicio.alta(this.art).subscribe(datos => {
if (datos['resultado']=='OK') {
alert(datos['mensaje']);
this.recuperarTodos();
}
});
}
```

También llamamos al método 'recuperarTodos' con el objetivo que se actualice la pantalla con los datos actuales de la tabla 'articulos'.

El flujo de la información para efectuar el 'alta' de un articulo en la base de datos es::



La actualización de la página HTML la logramos llamando al método 'recuperarTodos'.

Baja

¿El borrado de un artículo se efectúa cuando el operador presiona el botón con la etiqueta 'Borra?':

```
<td><button (click)="baja(art.codigo)">Borrar?</button></td>
```

Se llama al método 'baja' de la clase 'AppComponent' y se le pasa como parámetro el código de artículo a borrar.

El método baja:

```
baja(codigo) {
this.articulosServicio.baja(codigo).subscribe(datos => {
if (datos['resultado']=='OK') {
alert(datos['mensaje']);
this.recuperarTodos();
}
});
}
```

Llmamos al método 'baja' de la clase 'ArticulosService' y le pasamos como parámetro el código de artículo que queremos borrar.

El método 'baja' de la clase 'ArticulosService' procede a llamar al archivo baja.php y le pasa como parámetro el código de artículo que se debe borrar:

```
baja(codigo:number) {
return this.http.get(`${this.url}baja.php?codigo=${codigo}`);
}
```

Ahora en el servidor se ejecuta la aplicación PHP baja.php:

```
<?php
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

require("conexion.php");
$con=retornarConexion();
```

```
mysqli_query($con,"delete from articulos where codigo=$_GET[codigo]");



class Result {}

$response = new Result();
$response->resultado = 'OK';
$response->mensaje = 'articulo borrado';

header('Content-Type: application/json');
echo json_encode($response);
?>
```

Luego en el método 'baja' de la clase 'AppComponent' podemos mostrar un mensaje si la baja se ejecutó con éxito:

```
baja(codigo) {
this.articulosServicio.baja(codigo).subscribe(datos => {
if (datos['resultado']=='OK') {
alert(datos['mensaje']);
this.recuperarTodos();
}
});
}
```

Como podemos ver para actualizar la pantalla procedemos a llamar al método 'recuperarTodos'.

Consulta

La consulta o selección se dispara cuando el operador presiona el botón que tiene la etiqueta 'Seleccionar' y tiene por objetivo mostrar en los dos controles 'text' la descripción y precio del artículo:

```
<td><button (click)="seleccionar(art.codigo)">Seleccionar</button></td>
```

Al presionar el botón se llama el método 'seleccionar' de la clase 'AppComponent':

```
seleccionar(codigo) {
this.articulosServicio.seleccionar(codigo).subscribe(result => this.art = result[0]);
}
```

Ahora llamamos al método 'seleccionar' de la clase 'ArticulosService':

```
seleccionar(codigo:number) {
return this.http.get(`${this.url}seleccionar.php?codigo=${codigo}`);
}
```

Recuperamos del servidor llamando a la página 'seleccionar.php' los datos del artículo cuyo código pasamos como parámetro :

```
<?php
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

require("conexion.php");
$con=retornarConexion();

$registros=mysqli_query($con,"select codigo, descripcion, precio from articulos where codigo= $_GET [codigo]");

if ($reg=mysqli_fetch_array($registros))
{
$vec[]=$reg;
}

$cad=json_encode($vec);
```

```
echo $cad;
header('Content-Type: application/json');
?>
```

La ejecución del programa en PHP procede a recuperar la fila de la tabla que coincide con el código enviado y lo retorna con formato JSON.

En el método 'seleccionar' de la clase AppComponent al ejecutarse el método subscribe almacena en la propiedad 'art' el resultado devuelto por el servidor (con esta asignación se actualizan los dos controles 'input' del HTML):

```
seleccionar(codigo) {
this.articulosServicio.seleccionar(codigo).subscribe(result => this.art = result[0]);
}
```

Modificación

El último algoritmo que implementa nuestra aplicación es la modificación de la descripción y precio de un artículo que seleccionemos primeramente.

Cuando presionamos el botón que tiene la etiqueta 'Modificar' se ejecuta el método 'modificación':

```
<button (click)="modificacion()">Modificar</button></p>
```

El método 'modificación' se implementa en la clase 'AppComponent':

```
modificacion() {
this.articulosServicio.modificacion(this.art).subscribe(datos => {
if (datos['resultado']=='OK') {
alert(datos['mensaje']);
this.recuperarTodos();
}
});
}
```

Lo primero que hacemos en este método es llamar al método 'modificación' de la clase 'ArticulosService' y pasar como dato todos los datos del artículo seleccionado y las posibles modificaciones efectuadas.

El método 'modificacion' de la clase 'ArticulosService':

```
modificacion(articulo) {
return this.http.post(`${this.url}modificacion.php`, JSON.stringify(articulo));
}
```

Accede al servidor pidiendo el archivo 'modificacion.php' y pasando todos los datos del artículo mediante un 'post'.

El archivo 'modificacion.php' procede a cambiar la descripción y precio del artículo:

```
<?php
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

$json = file_get_contents('php://input');

$params = json_decode($json);

require("conexion.php");
$con=retornarConexion();



mysqli_query($con,"update articulos set descripcion='$params->descripcion',
precio=$params->precio
where codigo=$params->codigo");



class Result {}

$response = new Result();
$response->resultado = 'OK';
$response->mensaje = 'datos modificados';
```

```
header('Content-Type: application/json');
echo json_encode($response);
?>
```

En la clase 'AppComponent' podemos comprobar si el resultado fue 'OK' y actualizar nuevamente la página:

```
modificacion() {
this.articulosServicio.modificacion(this.art).subscribe(datos => {
if (datos['resultado']=='OK') {
alert(datos['mensaje']);
this.recuperarTodos();
}
});
}
```

Hemos explicado con este problema todos los pasos esenciales para implementar una aplicación en Angular que se comunica con un servidor web con PHP y acceder a una base de datos MySQL.

V. DISCUSION DE RESULTADOS

1. Para el primer ejemplo, realizarle algunos cambios, agregarle los siguientes campos: **dirección, teléfono y correo electrónico**.

2. Para el segundo ejemplo, realizarle algunos cambios, agregarle los siguientes campos: **proveedor y fabricante**.

   Nota: Para cada ejercicio agregar las validaciones que sean necesarias, por ejemplo, campos vacíos, formato de correo, etc.