

# PROGRAMADO EL FRONTEND CON ANGULAR



# CARACTERÍSTICAS DE ANGULAR

- **Framework para la creación de aplicaciones Web de lado cliente**
- **Basado en la creación de plantillas HTML gestionadas por componentes**
- **Simplifica la interacción con el usuario**
- **Vinculación a datos**
- **Implementación de código mediante TypeScript (superconjunto JavaScript orientado a objetos)**
- **Aplicaciones basadas en patrón MVC**

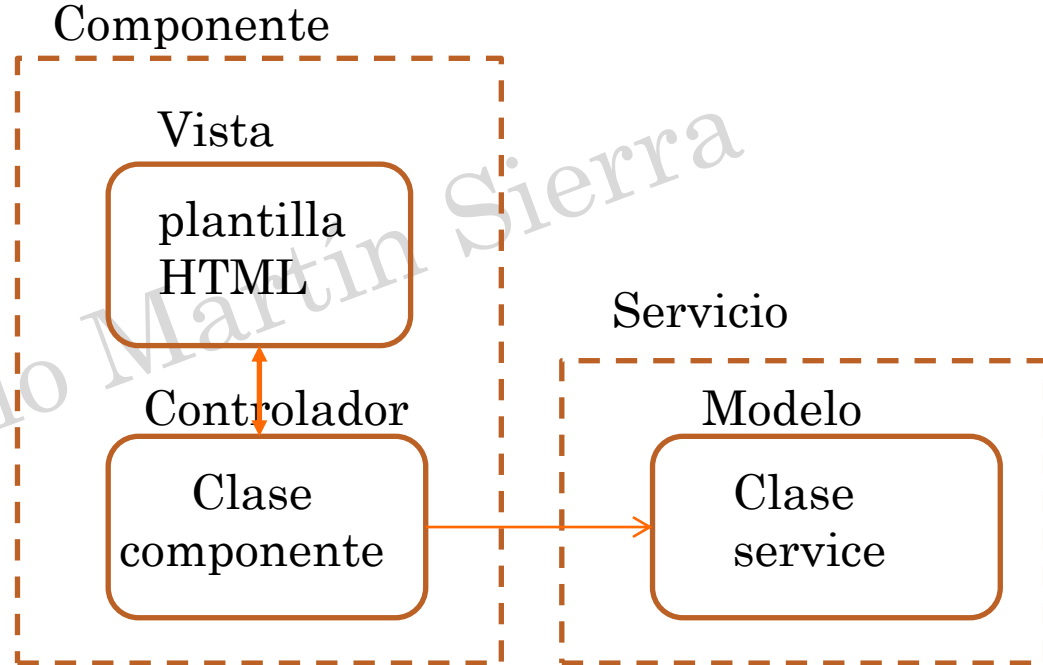


# PATRÓN MVC

➤ **Modelo: lógica de negocio**

➤ **Controlador: gestión de eventos de usuario**

➤ **Vista: generación de respuestas**



# CONFIGURACIÓN

➤ **Instalación de Node.js. Servidor para publicación de aplicaciones Angular. Incluye npm, una herramienta para gestión de dependencias. Descargable en:**

`https://nodejs.org/es/download/`

➤ **Angular cli. Herramienta para crear proyectos angular. Se instala con npm:**

`> npm install -g @angular/cli`

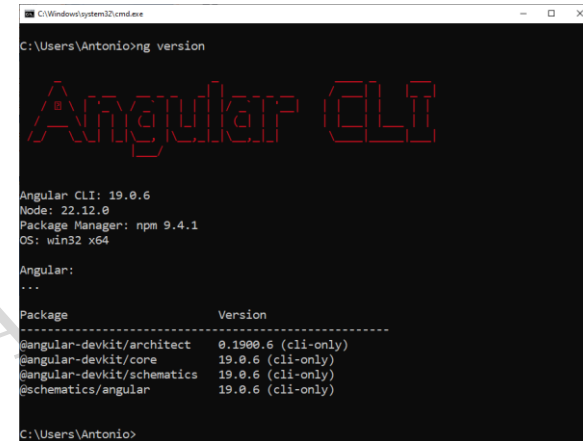
➤ **Visual Studio Code. Editor de proyectos angular para facilitar la codificación.**



# VERSIONES

➤ Se puede comprobar las versiones de herramientas mediante el comando:

>ng version



```
C:\Users\Antonio>ng version

Angular CLI
Angular CLI: 19.0.6
Node: 22.12.0
Package Manager: npm 9.4.1
OS: win32 x64

Angular:
...
Package      Version
-----
@angular-devkit/architect 0.1900.6 (cli-only)
@angular-devkit/core      19.0.6 (cli-only)
@angular-devkit/schematics 19.0.6 (cli-only)
@schematics/angular       19.0.6 (cli-only)

C:\Users\Antonio>
```

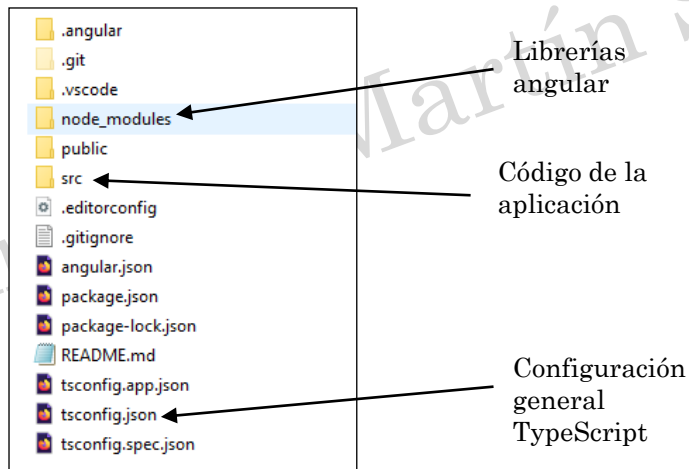


# CREACIÓN DE UNA APLICACIÓN

➤ Se crea desde línea de comandos utilizando:

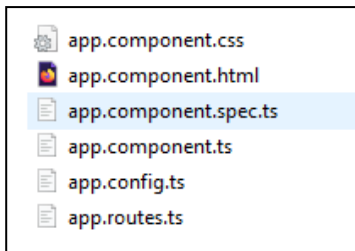
```
>ng new nombreApp
```

➤ Se crea una estructura similar a esta:



# COMPONENTES

➤ Al crear un proyecto se genera automáticamente un componente dentro de la carpeta src:



➤ Un componente define el aspecto de una página de la aplicación y controla su comportamiento

➤ Consta de dos elementos:

- **Plantilla.** Archivo HTML que genera la vista de la página
- **Componente.** Clase TypeScript donde se implementa la funcionalidad de la página



# PLANTILLA

- **Bloque HTML que forma un componente y establece el aspecto de la página.**
- **Puede incluir vínculos hacia la clase del componente para generar dinámicamente contenido y suministrar datos de usuario a dicha clase**

```
<div class="formulario">
  <div>
    <label for="">Introduce código: </label>
    <input type="text" [(ngModel)]="codigo">

    <br>
    <p><a>{{producto.name}}</a><br>
      <a>{{producto.precio}}</a>
    </p>
  </div>
</div>
```

Hacia la clase  
componente

Volcado desde el  
componente hacia  
la página





# CLASE DEL COMPONENTE (CONTROLADOR)

➤ Define el comportamiento de la página, recogiendo datos de esta, respondiendo a evento y generando resultados

Etiqueta para referirse al componente desde la página principal

Módulos externos requeridos

Plantilla HTML del componente

Código del componente

```
@Component({
  selector: 'app-buscador',
  imports: [FormsModule, CommonModule],
  templateUrl: './buscador.component.html',
  styleUrls: ['./buscador.component.css']
})
export class BuscadorComponent {
  producto: Producto
  constructor(private buscadorService: BuscadorService){
    producto = new Producto();
  }
  codigo: string;
}
```

# LA PÁGINA INDEX.HTML

- Es la página que se carga en el navegador al ejecutar una aplicación Angular
- Contiene referencias a componentes mediante la etiqueta de selector

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Aplicación Angular</title>
  <base href="/">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-buscador></app-buscador>
</body>
</html>
```

Referencia al  
componente que será  
procesado al solicitar  
la página

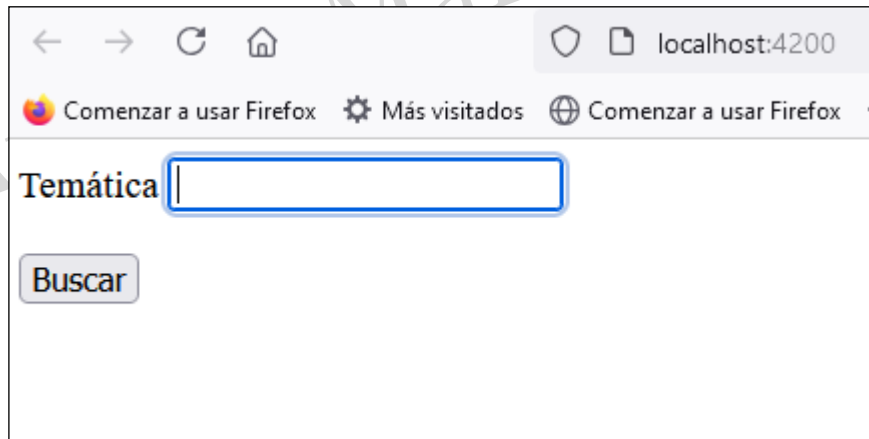


# EJECUCIÓN DE LA APLICACIÓN

➤ Para ejecutar una aplicación angular utilizaremos el comando:

```
>ng serve -o
```

➤ La aplicación se ejecutará automáticamente en un servidor node.js, se abrirá un navegador y se lanzará una solicitud de index.html



# VINCULACIÓN A DATOS

➤ A través de directivas asociamos el contenido de controles HTML de la platilla a propiedades del componente

```
<input type="text" [(ngModel)]="nombre">
```

Se debe importar el módulo *FormsModule* en el componente

➤ Mediante los interpoladores se vuelcan propiedades del componente en la página

```
<h2>{{texto}}</h2>
```

```
export class DatosComponent {  
  nombre:string;  
  texto:string;  
  :  
}
```




# PASO DE PARÁMETROS AL COMPONENTE

➤ Se pueden pasar parámetros al componente desde la página index mediante atributos de la etiqueta asociada, utilizando *property binding*

```
<app-root [level]="5"></app-root>
```

```
export class DatosComponent {  
  level:number;  
  :  
}
```

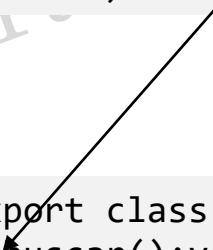


# EVENTOS

- Una de las funcionalidades de la capa front es capturar eventos o acciones de usuario.
- En angular los eventos se manejan a través de funciones de respuesta definidas en el componente

```
<input type="button" value="Buscar" (click)="buscar()">
```

```
export class BuscadorComponent {  
  buscar():void{  
    :  
  }  
}
```



# DIRECTIVA NGIF

➤ Puede ser incluida en cualquier etiqueta HTML para que dicha etiqueta sea o no procesada en función de una condición.

➤ Su formato:

```
<etiqueta *ngIf="expresion">...</etiqueta>
```

➤ Si *expresion* es evaluada como falso, la etiqueta será eliminada del árbol de objeto DOM.

➤ Para poder utilizar esta etiqueta es necesario importar el **CommonModule** en el componente:

```
@Component({  
  selector: 'app-buscador',  
  imports: [CommonModule],  
  :
```



# DIRECTIVA NGFOR

➤ Se incluye en una etiqueta para que esta aparecerá tantas veces como se indique en la expresión de iteración asignada a la directiva.

➤ Su formato:

```
<etiqueta *ngFor="let variable of array">...</etiqueta>
```

➤ Ejemplo:

Genera tantas filas <tr> como elementos haya en la colección o array "agenda"

```
<tr *ngFor="let c of agenda">  
  <td>{{c.nombre}}</td><td>{{c.edad}}</td><td>{{c.telefono}}</td>  
</tr>
```

➤ El uso de esta directiva también requiere la importación de CommonModule



# DIRECTIVA NGSWITCH

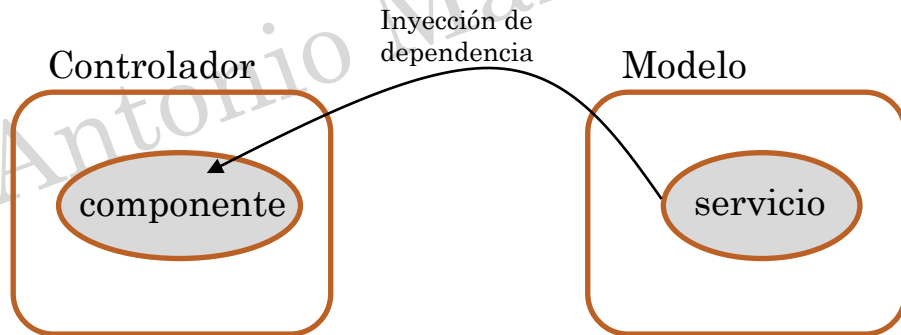
- Utilizamos esta directiva para mostrar diferentes elementos de un conjunto de ellos, en función del resultado de una expresión.
- Procesa la etiqueta cuyo valor @ngSwitchWhen coincide con el resultado de la expresión:

```
<div [ngSwitch]="seleccionado.nombre">  
  <div *ngSwitchWhen="Ana">Vive cerca</div>  
  <div *ngSwitchWhen="Belén">Acaba de mudarse</div>  
  <div *ngSwitchWhen="Marcos">Vive ahí desde siempre</div>  
  <div *ngSwitchDefault>desconocidos</div>  
</div>
```



# SERVICIOS

- Encapsulan la lógica de negocio de la aplicación (Modelo).
- Exponen su funcionalidad al componente controlador a través de métodos.
- El servicio es inyectado en el componente para que pueda hacer uso del mismo



# IMPLEMENTACIÓN DE UN SERVICIO

➤ Se implementa en una clase estándar anotada con **@Injectable**:

```
@Injectable({
  providedIn: 'root'
})
export class BuscadorService {
  buscar(tematica:string):String []{
    :
  }
}
```

➤ En el componente se inyecta a través del constructor:

```
export class BuscadorComponent {
  constructor(private buscadorService:BuscadorService){

  }
```

Inyección de servicio en un  
atributo de la clase



# PETICIONES HTTP

➤ Se realizan a través del componente `HttpClient` incluido en la librería `http`.

➤ Este componente deberá ser utilizado desde un servicio. Puede ser inyectado a través del constructor:

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

@Injectable({
  providedIn: 'root'
})
export class BuscadorService {
  constructor(private http: HttpClient) {
  }
}
```

➤ El proveedor del módulo debe ser declarado en `app.config`:

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

export const appConfig: ApplicationConfig = {
  providers: [provideHttpClient(),...]
};
```



# OBSERVABLES

- Una petición HTTP (get, post, ...) devuelve un Observable
- A un Observable se suscribe el componente controlador para procesar el resultado cuando esté disponible:

componente

```
export class PaisComponent {  
  paises:Pais[];  
  constructor(private paisesService:PaisesService){  
  }  
  cargarPaises():void{  
    this.paisesService.obtenerPaises().subscribe(data=>this.paises=data);  
  }  
}
```

suscripción al observable

servicio

```
export class PaisesService{  
  constructor(private http:HttpClient){  
  }  
  public obtenerPaises(): Observable<Pais[]> {  
    return this.http.get<Pais[]>(this.url); //Observable  
  }  
}
```

# ENVÍO DE DATOS EN PETICIONES HTTP

➤ Desde un cliente se pueden enviar datos a un recurso externo de la siguiente manera:

- Path variables. Los datos se envían como parte de la dirección
- QueryString. Se envían parámetros en la URL en parejas clave=valor
- Form-urlencoded. Se envían parámetros en el cuerpo de la petición en parejas clave=valor
- JSON. Se pueden enviar datos como un documento JSON en el cuerpo de la petición



# PATH VARIABLES Y QUERYSTRING

## ➤ Path variable:

```
export class BuscadorService{  
    find(cod:number):Observable<Item> {  
        return this.http.get<Item>("http://localhost:8000/buscador/${cod}");  
    }  
}
```

## ➤ Envío como QueryString:

```
export class BooksService{  
    listBooks(isbn: string):Observable<Book[]>{  
        return this.http.get<Book[]>("http://localhost:9000/books",{  
            params:{"isbn":isbn}  
        });  
    }  
}
```

# FORM-URLENCODED

➤ **Utilizado habitualmente en peticiones post cuando el backend espera recibir un formulario de datos:**

```
export class EmpleadosService{
  save(cod:number,name:string,age:number):Observable<void>{
    let params=new HttpParams();
    let heads=new HttpHeaders();
    //los parámetros se definen en un objeto params
    params=params.set("codigo",cod);
    params=params.set("nombre",name);
    params=params.set("edad",age);
    //se debe establecer un encabezado con el tipo de contenido
    heads=heads.set("Content-Type","application/x-www-form-urlencoded");
    return this.http.post<void>(url,params,{"headers":heads});
  }
}
```



# JSON

➤ **Forma de envío habitual para enviar un grupo de datos a un servicio REST en el cuerpo de la petición**

```
export class ClientesService{  
  save(cliente:Cliente):Observable<void>{  
    let heads=new HttpHeaders();  
    //se debe establecer un encabezado con el tipo de contenido  
    heads=heads.set("Content-Type","application/json");  
    return this.http.post<void>(url,cliente,{"headers":heads});  
  }  
}
```

Amo



# CABECERAS DE RESPUESTA

➤ Si queremos tener acceso a las cabeceras de respuesta, se debe incluir el parámetro `observe:"response"` en la lista de parámetros opcionales de la petición:

```
buscar(tematica:string):Observable<any>{  
    return this.http.get(this.urlBase+"buscar",  
        {params:{tematica:tematica},observe:"response"});  
}
```

➤ En el componente:

```
buscar():void{  
    this.buscadorService.buscar(this.tematica)  
        .subscribe(data=>{  
            console.log(data.headers)); //cabeceras  
            console.log(data.body);    //cuerpo  
        })  
}
```

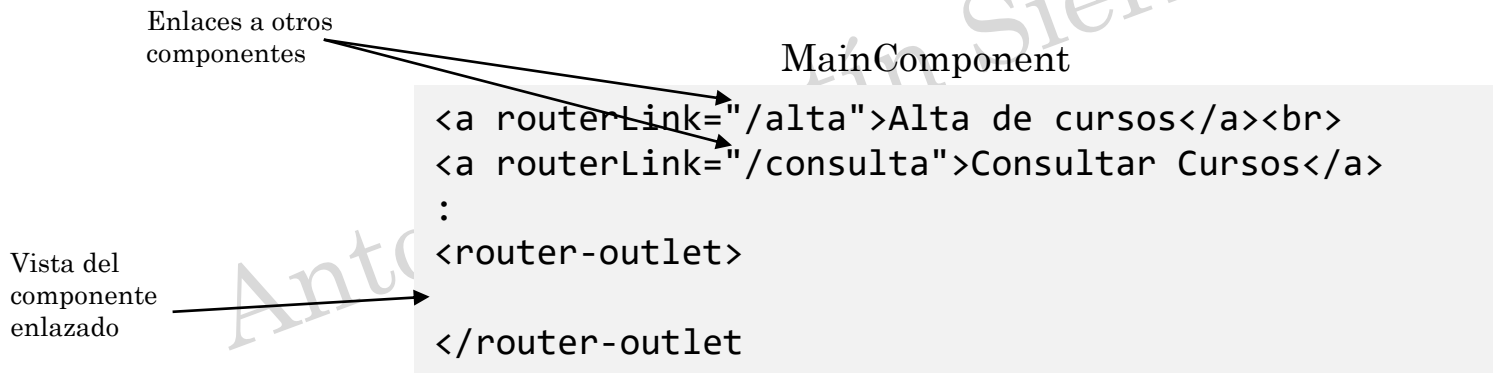
# ROUTING

- Es la capacidad para navegar entre componentes
- Enlaces en la vista de un componente provocan la carga de otros componentes
- La navegación puede realizarse también desde código
- El routing es habilitado por defecto al crear la aplicación.
- Las reglas de navegación se definen en el archivo `app-routing.module.ts`.



# CREAR RUTAS

- Los enlaces a otros componentes se generan en la vista del componente principal mediante el atributo *routelink*.
- En esta vista se incluirá una etiqueta `<router-outlet>` donde se mostrará la vista del componente enlazado:



- La navegación se puede realizar desde código mediante:

```
this.router.navigate(["/eliminar"])
```

# MAPEADO DE RUTAS A COMPONENTES

➤ La asociación de rutas a componentes se realiza en el archivo `app-routes.ts` :

```
const routes: Routes = [{  
  path:"alta",  
  component:AltaComponent  
},  
{  
  path:"consulta",  
  component: ConsultaComponent  
},  
{  
  path:"eliminar",  
  component: EliminarComponent  
}  
];
```



# CONFIGURACIÓN

➤ **En el main.ts se debe registrar el configurador de rutas:**

```
bootstrapApplication(AppComponent, {  
  providers: [provideRouter(routes)]  
});
```

➤ **Los componentes que utilicen routing, deben importar el RouterModule:**

```
@Component({  
  selector: 'app-home',  
  templateUrl: './home.component.html',  
  imports: [RouterModule]  
})
```



# ALMACENAMIENTO DE DATOS

- **TypeScript dispone de dos objetos para almacenar datos de usuario como cadenas de caracteres:**
  - **localStorage.** Almacena datos de forma permanente, incluso después de cerrar el navegador
  - **sessionStorage.** Almacena datos hasta que se cierra la sesión, lo cual ocurre al cerrar el navegador o la pestaña del mismo
- **En ambos casos, los datos se almacenan como clave-valor**



# MÉTODOS

➤ **setItem(key,value).** Almacena un objeto con una clave asociada. Si ya existe la clave, se reemplaza el dato:

```
localStorage.setItem("token","AAFG456RF39340HH");  
sessionStorage.setItem("user",JSON.stringify({name="h",role:"promotor"}));
```

➤ **getItem(key).** Devuelve un objeto a partir de su clave. Si no existe, devolverá null:

```
console.log(localStorage.setItem("token"));  
const data=JSON.parse(sessionStorage.getItem("user"));
```

➤ **removeItem(key).** Elimina el objeto cuya clave se indica





# VALIDACIÓN DE FORMULARIOS

➤ Angular permite validar formularios sin incluir código :

Identificadores para acceder a las propiedades de validación

```
<form #form="ngForm" (ngSubmit)="alta(form)">
  Usuario:<input type="text" name="user" #user="ngModel"
                                     [(ngModel)]="user" required><br>
  <div *ngIf="user.invalid && user.touched">
    <small *ngIf="user.errors?.['required']">La URL es obligatoria.</small>
  </div>
  :
  <input type="submit" value="Guardar" [disabled]="form.invalid"><br>
</form>
```

Propiedades de validación

El botón se deshabilita si el formulario no es válido



# VALIDACIÓN EN EL COMPONENTE

➤ En ocasiones es necesario tener acceso a las propiedades de validación en el código del componente:

```
alta(form:any):void{  
  if(form.valid){  
  }  
}
```

Recibe el componente  
formulario

Se tiene acceso a las mismas  
propiedades que en la vista



# PROPIEDADES DE VALIDACIÓN

➤ Tanto en la vista como en el controlador, se puede utilizar una serie de propiedades del componente:

- **invalid.** Es true si el componente incumple alguna de las reglas de validación definidas. En el caso del formulario, será true si alguno de los componentes tiene esta propiedad a true.
- **valid.** Propiedad contraria a la anterior
- **touched.** True si el componente ha recibido el foco en algún momento
- **disabled.** Indica si el control está deshabilitado
- **errors.** Proporciona acceso a todos los errores de validación. Devuelve null si no hay ninguno



# ANGULAR MATERIAL

➤ **Biblioteca de componentes UI para crear diseños atractivos y accesibles.**

➤ **Debe ser instalada con el comando:**

```
>ng add @angular/material
```

➤ **En el componente donde se vayan a utilizar los objetos material se deberá importar el módulo correspondiente en el decorador:**

```
imports: [  
  MatDialogModule  
]
```



# CUADRO DE DIÁLOGO

➤ Se define como un componente Angular:

```
export class CuadroDialogoComponent {  
  constructor(private dialogRef: MatDialogRef<CuadroDialogoComponent>,  
    @Inject(MAT_DIALOG_DATA) public data: { mensaje: string }  
  ) {}  
  cerrar() {  
    this.dialogRef.close();  
  }  
}
```

```
<h2 mat-dialog-title>Información</h2>  
  
<mat-dialog-content>  
  <p>{{ data.mensaje }}</p>  
</mat-dialog-content>  
<mat-dialog-actions align="end">  
  <button mat-button (click)="cerrar()">Cerrar</button>  
</mat-dialog-actions>
```



# USO CUADRO DE DIÁLOGO

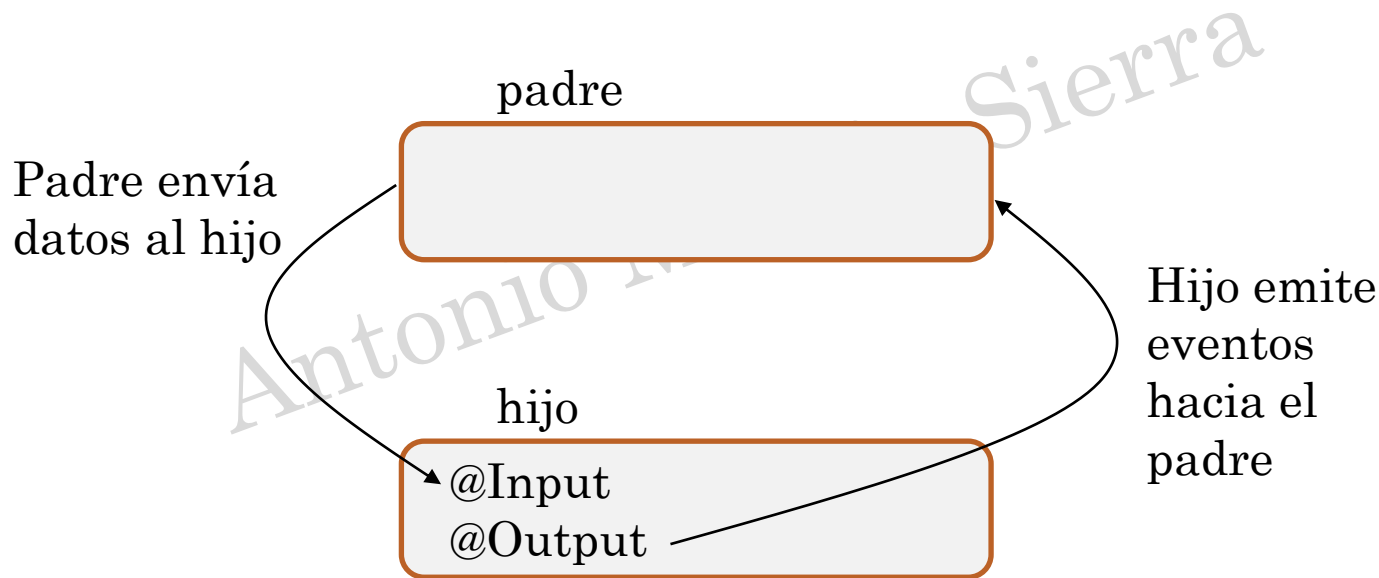
➤ Para utilizar el cuadro de diálogo desde otro componente, se debe inyectar MatDialog

```
constructor(private buscadorService:BuscadorService,private router:Router,  
             private dialog: MatDialog){  
  
}  
alta(form:any):void{  
  if(form.valid){  
    this.buscadorService.alta(this.resultado).subscribe({  
      next:data=>this.dialog.open(CuadroDialogoComponent,{  
        data: { mensaje: "Contacto creado correctamente" }  
      }),  
      error:err=>this.dialog.open(CuadroDialogoComponent,{  
        data: { mensaje: "No se pudo añadir!" +err }  
      })  
    });  
    this.router.navigate(["/buscar"]);  
  }else{  
    alert("No se pudo enviar el resultado!");  
  }  
}
```



# COMPONENTES PADRE-HIJO

- Cuando un componente va a ser reutilizado en distintas partes de la aplicación.
- Facilita la modularización de la aplicación



# EJEMPLO PADRE-HIJO

## componente hijo

```
export class HijoComponent {  
  mensaje:string;  
  @Input() nombre:string;  
  @Output() eventoMensaje:EventEmitter<string>=  
    new EventEmitter();  
  enviar(){  
    this.eventoMensaje.emit(this.mensaje);  
  }  
}
```

genera el evento  
hacia el padre

```
Nombre recibido:{{nombre}}<br><br>  
Introduce mensaje para el padre:  
<input type="text" [(ngModel)]="mensaje"><br><br>  
<input type="button" value="Enviar" (click)="enviar()">
```

## componente padre

```
export class AppComponent {  
  data:string;  
  textoRecibido:string;  
  recibir(param:string):void{  
    this.textoRecibido=param;  
  }  
}
```

```
Nombre: <input type="text" [(ngModel)]="data"><br>  
<hr>  
<app-hijo [nombre]="data"  
  (eventoMensaje)="recibir($event)"></app-hijo>  
<hr>  
Texto recibido del hijo:{{textoRecibido}}
```

Método del padre en el que  
se responde al evento  
generado por el hijo

variable del padre con  
contenido para hijo





# CARRITO

## componente hijo

```
export class ItemProductoComponent {
  @Input() producto: Producto; // Recibe el producto desde el padre
  @Output() agregar = new EventEmitter<Producto>();
  @Output() eliminar = new EventEmitter<number>();
  agregarCarrito() {
    this.agregar.emit(this.producto);
  }
  eliminarCarrito() {
    this.eliminar.emit(this.producto.codigo);
  }
}
```

```
<div class="container">
  <table class="table">
    <tbody>
      <tr>
        <td>{{ producto.producto }}</td>
        <td>{{ producto.categoria }}</td>
        <td>{{ producto.precio }}</td>
        <td><button (click)="agregarCarrito()">Agregar</button></td>
        <td><button (click)="eliminarCarrito()">
          Eliminar</button></td>
      </tr>
    </tbody>
  </table>
</div>
```

## componente padre

```
export class CarritoProductosComponent {
  productos: Producto[] = [
    { codigo: 1, producto: "teclado", categoria: "informática", precio: 35 },
    { codigo: 2, producto: "leche", categoria: "alimentación", precio: 1.4 },
    { codigo: 3, producto: "pan", categoria: "alimentación", precio: 2 },
    { codigo: 4, producto: "silla", categoria: "hogar", precio: 110 },
  ];
  carrito: Producto[] = [];
  // recibe el producto desde el hijo y lo agrega
  agregarCarrito(producto: Producto) {
    this.carrito.push(producto);
  }
  // recibe el código del producto y lo borra
  eliminarCarrito(id: number) {
    this.carrito = this.carrito.filter(p => p.codigo !== id);
  }
}
```

```
<tbody>
  <tr *ngFor="let prod of productos">
    <td scope="col" colspan="5">
      <app-item-producto [producto]="prod"
        (agregar)="agregarCarrito($event)"
        (eliminar)="eliminarCarrito($event)">
      </app-item-producto>
    </td>
  </tr>
</tbody>
:
<table class="table">
  <tr><th>Producto</th><th>Categoria</th><th>Precio</th></tr>
  <tr *ngFor="let item of carrito">
    <td>{{ item.producto }}</td>
    <td>{{ item.categoria }}</td>
    <td>{{ item.precio }}</td>
  </tr>
</table>
```