



Angular

Formularios Reactivos

Conceptos previos a Formularios

Antes de avanzar con formularios, analicemos algunos conceptos básicos que nos propone Angular:

- Expresiones
- Directivas
- Pipes

Expresiones

Angular nos permite trabajar con expresiones. Éstas se evaluarán previamente a volcar el resultado en la vista.

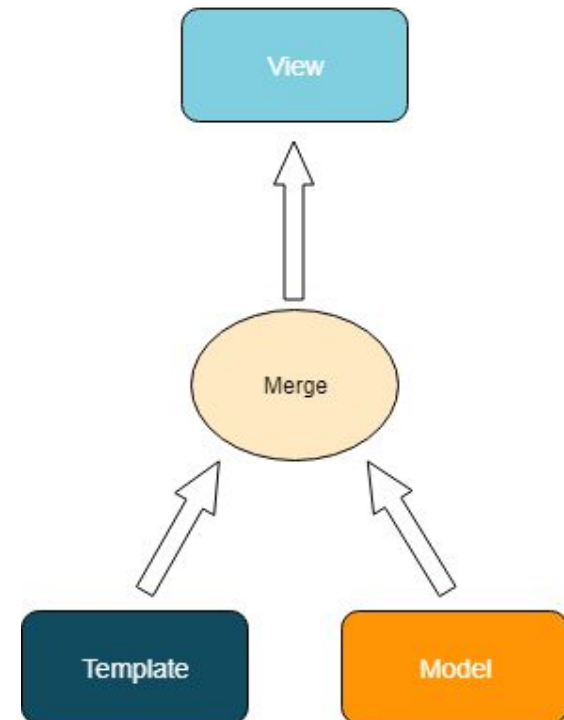
Ejemplo de expresiones en interpolaciones (One-way binding):

```
{{propiedad del componente}}
```

```
{{ método del componente}}
```

```
{{ 2020 + 1 }}
```

```
{{ ! valorBoleano }}
```



Directivas

Directivas

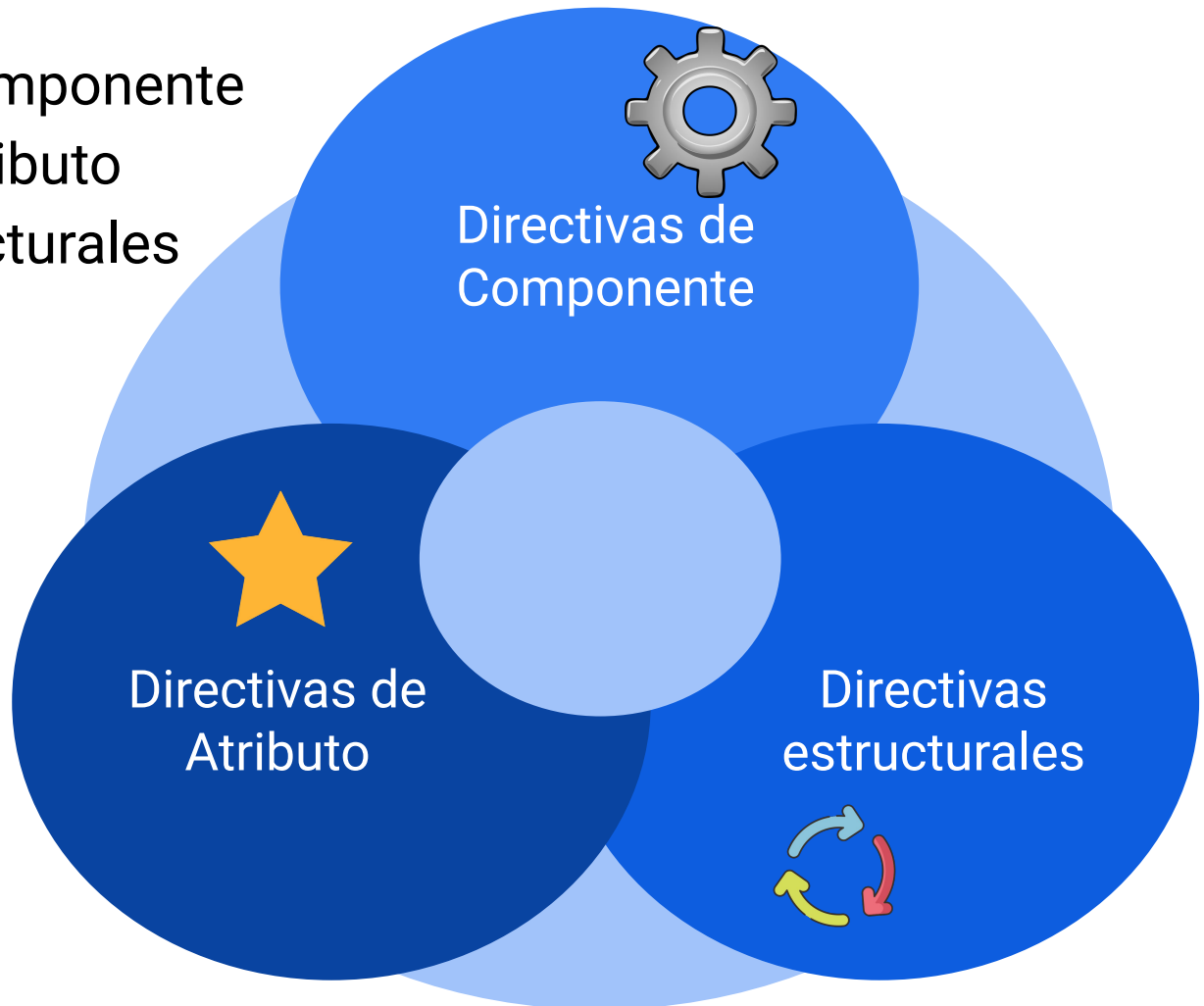
Son comandos que provee angular. Se pueden asignar a cualquier etiqueta por medio de atributos y **permiten manipular el DOM**.

Cuando se ejecuta la aplicación de angular, existe un “compilador” que se encarga de recorrer el documento y localizar las directivas para ejecutar los comportamientos que se especifican en las directivas.

Angular trae directivas para usar, pero podemos crear nuestras propias directivas también.

Tipos de directivas

- Directivas de componente
- Directivas de atributo
- Directivas estructurales



Directivas de Componente

Son las más utilizadas en Angular y como vimos previamente están compuestas por:

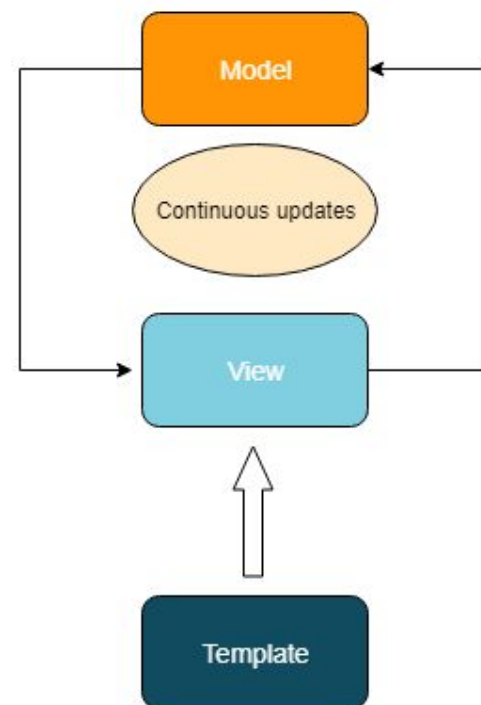
- La vista HTML
- Un archivo .ts (Typescript? que define el comportamiento.
- Un selector CSS que define cómo es utilizado en el template.
- Opcionalmente un archivo .css que define el estilo del componente y un archivo .ts para las pruebas unitarias.

Directivas de Atributo

Que son utilizadas para cambiar la apariencia o comportamiento de los elementos, componentes u otras directivas.

Las más comunes directivas de atributos son:

- **ngClass**. Permite agregar o remover clases CSS en función de un estado o expresión de manera dinámica.
- **ngModel**. Permite Two-way binding (Desde/Hacia el DOM) de manera dinámica.



Directiva: ngClass

Ejemplo: Supongamos que queremos cambiar el estado de un botón según esté apagado o encendido: OFF/ON:

En la vista:

```
<button class="btn" [ngClass]="{on: estadoPositivo, off:  
!estadoPositivo}" (click)="cambiarEstado()">{{texto}}</button>
```

En el archivo ts:

```
estadoPositivo: boolean = true;  
texto:string="si";  
  
cambiarEstado()  
{  
  this.estadoPositivo = !this.estadoPositivo;  
  if (this.estadoPositivo)  
  {this.texto="si";}  
  else  
  {this.texto="no";}  
}
```

Clase CSS de acuerdo a un estado (verdadero o falso)
- Se podría agregar una expresión

Directiva ngModel

ngModel es un enlace entre la variable y el elemento de la vista (en ambos sentidos).

Previamente importar FormsModule

- Si deseamos que el valor de la variable se muestre en un input type (one way binding):

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

- Si deseamos además, poder acceder a la variable que ingresa el usuario (two way binding)

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

Directivas estructurales

Cambian la estructura de la vista agregando o eliminando elementos. Son ejemplos:

- **ngIf**
- **ngFor**
- **ngSwitch**

Directiva estructural: ngFor

ngFor permite recorrer un array y para cada uno de sus elementos replicar una cantidad de elementos en el DOM
como se muestra a continuación:

```
<table>
  <thead>
    <th>Operación</th>
    <th>Monto</th>
  </thead>
  <tbody>
    <tr *ngFor="let element of movimientos" >
      <td>{{element.operacion}}</td>
      <td>{{element.monto}}</td>
    </tr>
  </tbody>
</table>
```

Directiva estructural: ngIf

ngIf permite ocultar o mostrar un elemento a partir de una expresión.

Ejemplo:

```
<table *ngIf="mostrar_movimientos">
  <thead>
    <th>Operación</th>
    <th>Monto</th>
  </thead>
  <tbody>
    <tr *ngFor="let element of movimientos" >
      <td>{{element.operacion}}</td>
      <td>{{element.monto}}</td>
    </tr>
  </tbody>
</table>
```

Directiva estructural: ngSwitch

ngSwitch permite mostrar u ocultar elementos dependiendo de una expresión.

Ejemplo:

```
<div [ngSwitch]="tipo_de_cliente">  
  <p *ngSwitchCase="1">Tipo de Cliente A</p>  
  <p *ngSwitchCase="2">Tipo de Cliente B</p>  
  <p *ngSwitchCase="3">Tipo de Cliente C</p>  
  <p *ngSwitchDefault>...</p>  
</div>
```

Pipes

Pipes son una herramienta que nos permite transformar visualmente la apariencia de los datos. Es decir, el formato.

Ejemplos:

```
<h2>{{ hoy | date: "d/M/yy" }}</h2>
```

```
<table *ngIf="mostrar_movimientos">
  <thead>
    <th>Operación</th>
    <th>Monto</th>
  </thead>
  <tbody>
    <tr *ngFor="let element of movimientos" >
      <td>{{ element.operacion }}</td>
      <td>{{ element.monto | currency }}</td>
    </tr>
  </tbody>
</table>
```

Formularios en angular

Angular nos provee dos tipos de formularios:

- **basados en plantilla:** proporcionan un enfoque basado en directivas.
- **reactivos:** proporcionan un enfoque basado en modelos y, comparados con los formularios basados en plantillas son más robustos, escalables, reusable y testeables.

Formularios en angular

	Reactivos	Basados en plantillas
Configuración	Explícita. Se crea en la clase del componente.	Implícita. Se crea a través de directivas.
Modelo de Datos	Estructurado e inmutable	No estructurado y mutable
Flujo de datos (entre las vistas y el modelo)	Síncrono	Asíncrono
Validación de formularios	Por medio de funciones	Por medio de directivas

Formularios Reactivos

Proveen un approach explícito e inmutable para administrar el estado del formulario cuyos valores cambian con el tiempo. Cada cambio de estado en el formulario retorna un nuevo valor permitiendo mantener la integridad con el modelo.

Además, cada elemento de la vista está directamente enlazado al modelo mediante una instancia de FormControl. Las actualizaciones de la vista al modelo y del modelo a la vista son síncronas y no dependen de la representación en la Interfaz de Usuario del cliente.

Dichos formularios están basados en flujos de datos del tipo **Observable**, donde cada entrada/valor puede ser accedido de manera asíncrona.

Formularios reactivos

Para que angular interprete nuestros formularios como reactivos, debemos importar el módulo de Formularios Reactivos en el archivo “app.module.ts” o en los módulos dónde necesites trabajar los formularios reactivos como sigue:

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

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

Formularios Reactivos

Clases

- **FormControl.**
- FormGroup
- **FormBuilder**
- FormArray

FormControl

Es la unidad más pequeña de un formulario reactivo. Ej. un cuadro de texto, un calendario, una lista desplegable, etc.

Para configurar un FormControl reactivo, primero importar la clase FormControl:

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


luego, declarar el nuevo form control:

```
@Component({  
  selector: 'app-iniciar-sesion',  
  templateUrl: './iniciar-sesion.component.html',  
  styleUrls: ['./iniciar-sesion.component.css']  
})  
export class IniciarSesionComponent implements OnInit {  
  mail= new FormControl('',[],[]);  
  constructor() { }  
  ngOnInit(): void {  
  }  
}
```

Form Control

Finalmente, enlazar el form control al template como sigue:

```
<form class="m-2">
  <div class="form-group">
    <label for="email">Email address:</label>
    <input type="email" [formControl]="mail"
class="form-control" placeholder="Enter email"
id="email">
  </div>
  . . .
```



¿Cómo agregamos la lógica que responde a los eventos en Angular ?

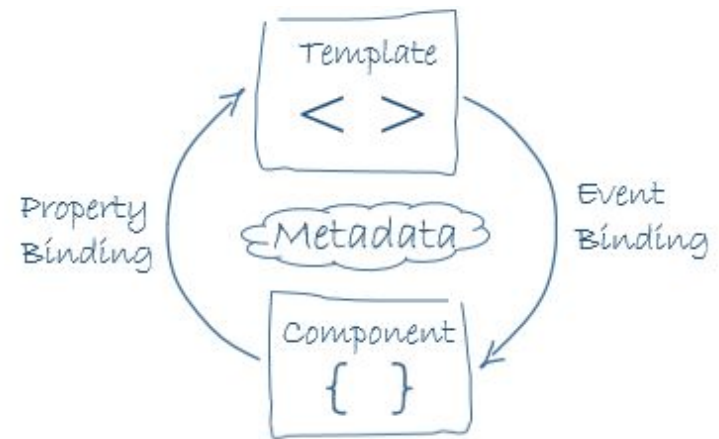
Para debatir en clase!!

Data Binding

Data Binding, nos abstrae de la lógica get/set asociada a insertar y actualizar valores en el HTML y, de convertir las respuestas de usuario (inputs, clicks, etc) en acciones concretas. Escribir toda esa lógica antes, era tedioso y propenso a errores dado que debíamos trabajar con javascript (o alguna librería como por ej. jquery).

Angular nos propone 4 formas de Data Binding:

1. Interpolación (Hacia el DOM)
2. **Property Binding (Hacia el DOM)**
3. **Event Binding (Desde el DOM)**
4. Two-way binding (Desde/Hacia el DOM)



Fuente de la imagen: <https://angular.io/docs>

Property Binding

Property Binding permite asignar un valor a un elemento del template o directivas.

Para enlazar una propiedad de un elemento HTML debemos encerrar entre corchetes la propiedad del DOM que deseamos configurar.

Ejemplos:

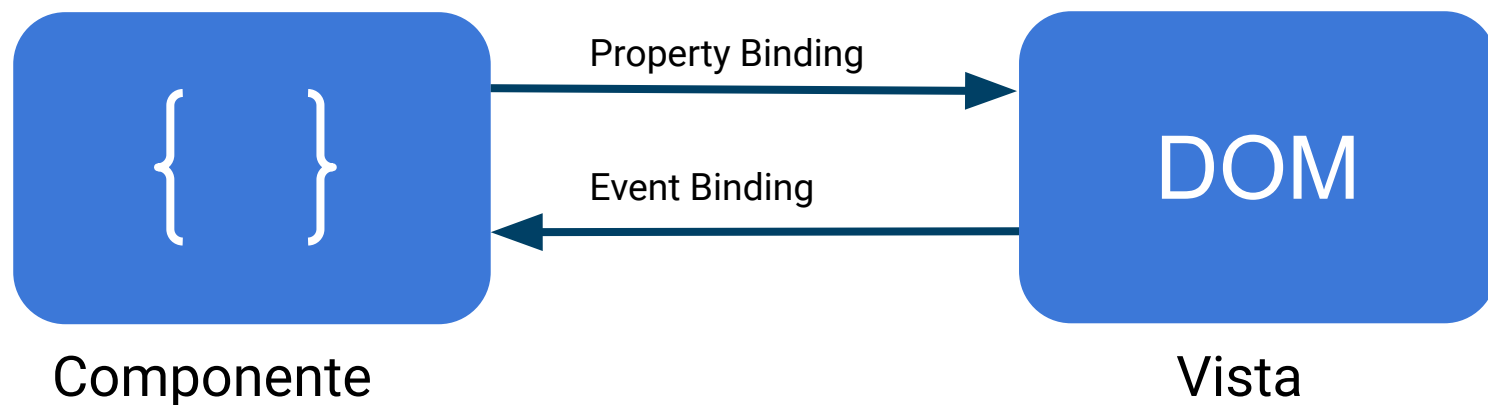
```
<img [src]="itemImageUrl">
```

```
<span [innerHTML]="propertyTitle"></span>
```

```
<p [class.border-danger]="!propertyValid"></p>
```


Event Binding

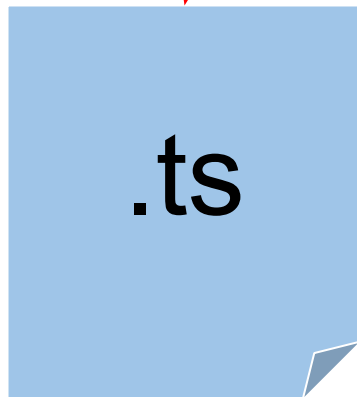
Event Binding, es el mecanismo de data binding (envío de datos) que nos permite trabajar con los **eventos del DOM**.



Event Binding

```
<button (click)="onSaludar()">Enviar</button>
```

```
onSaludar(event: Event)  
{  
  alert("Hola Mundo");  
}
```



Contiene la
lógica.



Contiene la vista.

Event Binding



¿Qué eventos podemos utilizar?

Todos los eventos definidos en el DOM, el listado completo lo podemos ver por ejemplo dentro de la web de [W3Schools](https://www.w3schools.com/js/default_events.asp).

¿Cómo funciona event binding?

Angular configura un manejador del evento por cada target (destino).

Una forma común de trabajar con un manejador de eventos es pasar al evento **\$event** al método dado que, el objeto **\$event** frecuentemente contiene la información que el método necesita como por ejemplo: el nombre del usuario, una url, etc.

¿Cómo funciona?

1. El método se enlaza al evento del elemento html (ej. input), el cual permite al manejador del evento escuchar los cambios.
2. Cuando el usuario realiza un cambio, el manejador de eventos de HTML genera el evento.
3. El binding se ejecuta dentro del contexto que incluye el objeto DOM **\$event**.
4. Angular recupera los cambios al llamar al método enlazado.

¿Cómo agregar validaciones a los formularios reactivos?

Validaciones de un Form Control en Angular

Angular nos provee la clase **Validators**, la cual contiene una serie de métodos estáticos que nos permiten validar las entradas de datos comunes tales como el formato del email, valores numéricos, máximos y mínimos, cantidad mínima y máxima de caracteres, entre otros.

Clic [aquí](#) para ver las validaciones que nos provee Angular.

Pasos:

1- Importar la clase Validators como sigue:

```
import {FormControl, Validators} from  
'@angular/forms';
```

2- En el componente, configurar las validaciones en la instancia del formControl como sigue:

```
mail= new FormControl('', [Validators.required,  
Validators.email]);
```

Para ver más validators clic [aquí](#)

Validaciones en Angular

3- En el template, agregar un `<div>` con las directivas necesarias a fin de mejorar la experiencia de usuario como sigue:

```
<div *ngIf="mail.errors && mail.dirty">
  <p *ngIf="mail.hasError('required')">
    El mail es requerido.
  </p>
  <p *ngIf="mail.hasError('email')">
    El formato del mail debe ser válido.
  </p>
</div>
</div>
```

.hasError('validator')

Método booleano que permite identificar si una validación falla o no.

.errors

Propiedad booleana que especifica que el form control tiene errores.

.dirty

Propiedad booleana que especifica si el form control está sucio. Es decir que el usuario no lo ha ingresado aún ningún dato.

Form Builder

Angular nos provee una sintaxis para la creación de nuevas instancias de FormGroup y FormControl a través del FormBuilder.

Para ello debemos:

1- En el componente, importar la clase:

```
import {Validators, FormGroup, FormBuilder} from '@angular/forms';
```

2- Luego, inyectar en el constructor el formBuilder:

```
constructor( private formBuilder: FormBuilder) {...
```

3- y finalmente, en el constructor crear el grupo de controles para el formulario:

```
this.form= this.formBuilder.group(  
  {  
    password:['',[Validators.required]],  
    mail:['',[Validators.required, Validators.email]]  
  }  
)
```


Form Builder

4- En el template, enlazar con el form builder y los form controls:

```
<form class="m-2" [formGroup]="form" (ngSubmit)="onSubmit()" novalidate>
  <div class="form-group">
    <label for="email">Email address:</label>
    <input type="email" formControlName="mail" class="form-control"
placeholder="Enter email" id="email">
  </div>
  <div class="form-group">
    <label for="pwd">Password:</label>
    <input type="password" formControlName="password" class="form-control"
placeholder="Enter password" id="pwd">
  </div>
  <button type="submit" class="btn btn-primary mt-2">Enviar</button>
</form>
```

Validaciones con Form Builder

Para configurar validaciones con Form Builder, importar la clase *Validators* y luego, utilizar el método `.get('field')` en las directivas `*ngIf`:

```
<div *ngIf="form.get('password').errors &&
form.get('password').touched">
  <p *ngIf="form.get('password').hasError('required') "
class="text-danger">
    El password es requerido.
  </p>
</div>
```

.touched. Propiedad booleana que especifica si el form control fue tocado por el usuario.

.errors. Propiedad booleana que especifica si el formulario tiene errores (falla una o más validaciones).

Validar previo enviar

Para hacer validaciones previo a enviar los datos al servidor, es buena práctica configurar el evento *onSubmit* en la etiqueta `<form>` como sigue:

```
<form class="m-2" [formGroup]="form"
```

```
(ngSubmit)="onEnviar($event)" novalidate>
```

Luego, en el componente configurar el método *onEnviar*:

```
onEnviar(event: Event)
{
  event.preventDefault(); //Cancela la funcionalidad por default.
  if (this.form.valid)
  {
    console.log(this.form.value);
  }
  else
  {
    this.form.markAllAsTouched(); //Activa todas las validaciones
  }
}
```

Clases de Bootstrap en formularios reactivos

¿Y si deseamos que el borde del elemento HTML esté en rojo? Para así, mejorar la experiencia de usuario:

1- Agregar borde en rojo cuando el dato ha sido mal ingresado:

```
<<input type="email" class="border border-3"  
[class.border-danger]="!mailValid"  
formControlName="mail" class="form  
placeholder="Enter email" id="email">
```

mailValid: es una propiedad del componente.

Property Binding

```
get passValid()  
{  
    return this.passField.touched && this.passField.valid;  
}
```

Referencias

<https://angular.io/guide/reactive-forms>

<https://angular.io/guide/event-binding-concepts>

<https://angular.io/api/forms/Validators>