



Angular 17





Tema 6. Formularios

Objetivos



- Sintaxis de formularios
- Técnicas habituales de programación y estilos

Contenidos

-  Formularios de Plantilla
-  Formularios Reactivos
-  Validación de formularios
-  Modelo de objetos

Formularios en Angular

Formularios basados en plantillas

- Modelo del formulario definido por las directivas de la plantilla
- Los cambios en los datos son asíncronos (ngModel)
- Validación mediante directivas
- Más fáciles de escribir

```
<form (ngSubmit)="enviarDatos(formulario)" #formulario="ngForm">
  <div>
    <label for="id">Identificador</label>
    <input type="number" id="id" name="id" class="corto"
      [(ngModel)]="dato.id" required="true" min="1"
      step="100" #identificador="ngModel">
```

Formularios reactivos (basados en el modelo)

- Modelo del formulario creado de forma explícita en TypeScript
- Cambios en los datos síncronos (submit)
- Validación mediante funciones en TypeScript
- Más versátiles

```
<form [formGroup]="formulario" (ngSubmit)="enviarDatos()">
  <div>
    <label for="id">Identificador</label>
    <input type="number" id="id" name="id" class="corto obligatorio"
      formControlName="identificador" step="100">

formulario=new FormGroup({
  identificador:new FormControl('', [Validators.required, Validators.min(1)])
```

Ambos usan internamente las mismas clases: FormGroup, FormControl, AbstractControl

Formularios basados en plantilla

Los formularios se enlazan con Angular directamente en la plantilla del componente, mediante directivas:

- Variable de plantilla = "directiva"
- ngForm
- ngModel
- Enlace bidireccional (two-way binding)

```
<form (ngSubmit)="enviarDatos(formulario)" #formulario="ngForm">
...
<input type="text" id="apellidos" name="apellidos"
      [(ngModel)]="dato.apellidos" #apellidos="ngModel">
```

FormControl (ngModel), FormGroup (ngForm) \implies AbstractControl

- value Valor del campo o array de campos del formulario
- valid / invalid Cumple las reglas de validación
- touched / untouched Ha ganado y perdido el enfoque
- pristine / dirty El valor del campo nunca se ha modificado

```
@if(apellidos.valid || apellidos.pristine || !apellidos.errors?.['required']){
  <p>El apellido es obligatorio</p>
}
```

```
comprobar(campo:NgModel, propiedad:string) {
  return campo.valid || campo.pristine || !campo.errors?.[propiedad]
}
```

Formularios basados en plantilla. Validación

Valida mediante **directivas**. Por defecto hay definida una directiva por cada atributo estándar de validación de HTML:

- pattern Expresión regular
- required El campo es obligatorio
- min / max Números o fechas en formato "yyyy-MM-dd"
- minlength / maxlength Longitud del texto

```
<input type="text" id="apellidos" name="apellidos"
[(ngModel)]="dato.apellidos" #apellidos="ngModel"
required minlength="3" maxlength="80">
```

```
<input type="number" id="salario" name="salario"
[(ngModel)]="dato.salario" #salario="ngModel"
min="12000" max="100000">
```

Para dibujar los mensajes de validación:

- Campos de estado: valid/invalid, pristine/dirty, touched/untouched, status
- Campos para validaciones: errors?.['validación'] / hasError('validación')
- Clases asociadas: ng-valid, ng-invalid, ng-pristine, etc.
- @if y las directivas hidden, disabled

```
<span [hidden]="apellidos.valid || apellidos.pristine || !apellidos.errors?.['minlength']">
  El tamaño mínimo es {{apellidos.errors?.['minlength']?.requiredLength}}
</span>
```

```
@if(apellidos.invalid && apellidos.dirty && apellidos.hasError('minlength')){
  <p>El tamaño mínimo es {{apellidos.errors?.['minlength']?.requiredLength}}</p>
}
```

Formularios reactivos

El modelo del formulario se define en TypeScript. Las clases «FormGroup» y «FormControl» se usan directamente.

```
formulario=new FormGroup({
  identificador:new FormControl(''),
  nombre:new FormControl(''),
  apellidos:new FormControl(''),
  ...
});

<form [formGroup]="formulario" (ngSubmit)="enviarDatos()">
  <input type="number" id="id" name="id" formControlName="identificador">
  <input type="text" id="nombre" name="nombre" formControlName="nombre">
  <input type="text" id="apellidos" name="apellidos" formControlName="apellidos">
  ...
</form>
```

FormControl, FormGroup \implies AbstractControl. El modelo de objetos es el mismo.

Disponemos del servicio «FormBuilder»:

```
formulario=this.formBuilder.group({
  identificador:[''],
  nombre: [''],
  apellidos: [''],
  ...
});
```

Podemos anidar formularios o añadir campos en ejecución (FormArray)

Formularios reactivos. Validación

Valida mediante **funciones**, en la definición de los controles. La clase «Validators» implementa las más habituales:

- pattern Expresión regular
- required El campo es obligatorio
- min / max Sólo valida números
- minlength / maxlength Longitud de un texto
- email Correo bien formado
- compose Une varias funciones de validación

```
formulario=new FormGroup({
  identificador:new FormControl('', [Validators.required, Validators.min(1)]),
  nombre:new FormControl('', [Validators.required, Validators.minLength(3)]),
  apellidos:new FormControl('', [Validators.required, Validators.minLength(3)]),
  ...
});
```

Es habitual definir reglas de validación propias (por ejemplo para fechas).

- Debe tener un parámetro de tipo «AbstractControl»
- Debe devolver un JSON con información sobre el error o null
- Suele emplearse la interfaz «ValidatorFn» para definir las, junto a métodos estáticos

```
static mayusculas(aceptaEspacios:boolean):ValidatorFn {
  return (control:AbstractControl): ValidationErrors | null => {
    if (control.value==null) return null;
    if (!aceptaEspacios && control.value.indexOf(' ')!==-1)
      return {'mayusculas': {'mayusculas': 'hay espacios', 'actual': control.value}};
    if (/^[A-ZÑÁÉÍÓÚ]+$/i.test(control.value)) return null;
    return {'mayusculas': {'mayusculas': 'no son mayúsculas', 'actual': control.value}};
  }
}
```


¿Qué hemos aprendido?

- Creación de formularios (plantilla y reactivos)
- Modelo de objetos
- Validación de formularios

Resumen de comandos

npm

npm install [-g] paquete
npm install
npm uninstall paquete
npm -version

ng (1)

ng version
ng help
ng serve [-o]
ng build
ng new nombre_proyecto
 --routing false
 --skip-tests
 --skip-git
 --no-standalone

ng (2)

ng generate component nombre_componente / ng g c
ng generate service nombre_servicio / ng g s
ng generate module nombre_módulo / ng g m

Ejercicio 06 A

Usando formularios basados en plantilla, crea una aplicación para ver, crear, borrar y modificar «productos». Aplica las validaciones que consideres necesarias.

El producto debe estar definido de esta forma:

```
export class Producto {
  constructor(
    public referencia:string | null,
    public nombre:string | null,
    public precio:number | null,
    public fechaAlta:string | null
  ){}
}
```

Quiero que crees el componente **detalle-desplegable**, que dibujará un «select» con los datos que le pasen desde fuera, y **detalle-pantalla**, que dibujará un formulario para crear o modificar cierto producto.

Serán usados por los componentes **nuevo**, **borrar** y **cambiar**, que son los que usarán el servicio. También crearás **ver**, para poder comprobar las modificaciones efectuadas.

Productos (Ejercicio 06 A)

Listado

Añadir

Eliminar

Cambiar

Modificar Productos

Seleccione un producto

Volcadora Taur F-34 Version 2

Seleccionar producto

Referencia de producto

VOL01

Nombre de producto

Volcadora Taur F-34 Version 2

Precio de compra

1500

Fecha de adquisición

14/05/2006

Modificar producto

Borrar

El producto se ha modificado correctamente.

Ejercicio 06 B

Modifica el proyecto anterior para que use formularios reactivos. Dependiendo de cómo lo hayas escrito, seguramente tendrás que cambiar únicamente los componentes **detalle-desplegable** y **detalle-pantalla**.

```
export class PantallaComponent implements OnChanges {
  @Input() producto: Producto = new Producto(null, null, null, null);
  ...
  formulario = new FormGroup({
    referencia: ...
    nombre: ...
    precio: ...
    fechaAlta: ...
  });

  ngOnChanges(cambios: SimpleChanges): void {
    if (cambios['producto'] && cambios['producto'].currentValue) {
      const p = cambios['producto'].currentValue;
      this.formulario.controls.referencia.setValue(p.referencia);
      this.formulario.controls.nombre.setValue(p.nombre);
      this.formulario.controls.precio.setValue(p.precio);
      this.formulario.controls.fechaAlta.setValue(p.fechaAlta);
      ...
    }
  }
}
```

Ambos componentes tienen que realizar tareas muy distintas (por ejemplo «detalle-pantalla» se usa para crear y para modificar). Tendrás que utilizar los métodos «on change» del ciclo de vida para que los valores se dibujen correctamente en los controles del formulario.