# DWC

# (Desarrollo Web en entorno cliente)

# Índice

# 1.- Formularios en Angular

Los formularios en Angular son una herramienta fundamental para la interacción del usuario con la aplicación. Permiten capturar datos del usuario, validar su entrada y realizar acciones en base a la información proporcionada.

Ya hemos trabajado con el two-way data binding en campos de formulario de forma independiente, cuando trabajamos con formularios nos interesa manipular estos campos de forma global, para ello, incluiremos estos campos dentro de un formulario, **los agrupamos en la etiqueta form**.

En Angular disponemos de dos formas para trabajar con los formularios, **formularios basados en plantillas** y **formularios reactivos**.

La diferencia entre ellos es la forma en la que se gestiona la lógica del formulario, es decir **las validaciones**.

- **En los formularios basados en plantillas** la lógica se encuentra en la plantilla, es decir en el fichero HTML del componente. En este caso se encuentra diseño y lógica en la vista
- **En los formularios reactivos la lógica** se encuentra en el controlador, es decir en el fichero HTML. En este caso se encuentra separados el diseño y la lógica.

| Característica | Formularios Basados en Plantillas | Formularios Reactivos |
|---|---|---|
| Estructura | Definidos en la plantilla HTML | Creados con clases TypeScript |
| Enfoque | Declarativo | Imperativo |
| Control de la lógica y validación | Menor control | Mayor control |
| Manejo de formularios complejos | Dificultoso | Fácil |
| Reutilización de componentes | Limitada | Efectiva |
| Curva de aprendizaje | Baja | Más pronunciada |
| Sintaxis | Simple | Más compleja |
| Cantidad de código | Menor | Mayor |

**¿Cuál elegir?**, pues para formularios, sencillos y sin mucha complicación lo más habitual es utilizar formularios basados en plantilla, cuando el formulario se empieza a complicar deberíamos utilizar formularios reactivos.

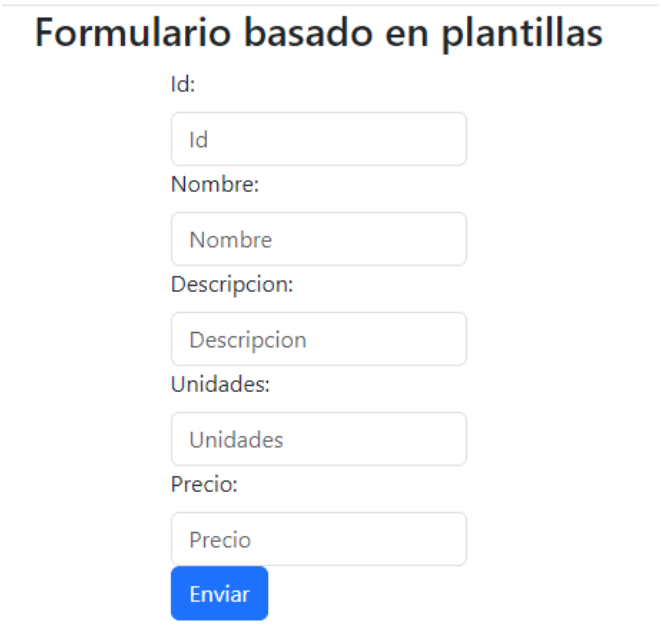
## 2.- Formularios basados en plantillas

Para trabajar con formularios basados en plantillas vamos a crear un componente llamado formularioPlantilla.

Vamos a crear un formulario para poder gestionar los artículos.

```
<div class="container">
    <form>
        <label class="form-label">Id:</label>
        <input type="text"
            class="form-control"
            placeholder="Id">

        <label class="form-label">Nombre:</label>
        <input type="text"
            class="form-control"
            placeholder="Nombre">

        <label class="form-label">Descripcion:</label>
        <input type="text"
            class="form-control"
            placeholder="Descripcion">

        <label class="form-label">Unidades:</label>
        <input type="number"
            class="form-control"
            placeholder="Unidades">

        <label class="form-label">Precio:</label>
        <input type="number"
            class="form-control"
            placeholder="Precio">

            <div>
            <button class="btn btn-primary">Enviar</button>
        </div>
    </form>
</div>
```

El resultado es:




Si añadimos en la etiqueta form **la propiedad action con una url** habremos conseguido un formulario que al pulsar en el botón de enviar se manden los datos a la url indicada.

Como nos encontramos trabajando con Angular, lo más normal es que **queramos acceder a los datos del formulario para poder trabajar con ellos**. Además, como estamos en el **desarrollo de una SPA**, al mandar los datos la página se recargaría de nuevo y nosotros no queremos que eso suceda.

Para poder acceder a los datos de los campos del formulario deberemos utilizar el **two-way data binding** que nos permitirá enlazar los valores de los campos con variables definidas en el controlador.

Cuando trabajemos con formularios deberemos **incluir en nuestro proyecto el módulo FormsModule.** Para ello vamos al controlador y lo importamos. Además, lo debemos incluir en los imports del componente

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

@Component({
  selector: 'app-formularios-plantillas',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './formularios-plantillas.component.html',
  styleUrl: './formularios-plantillas.component.css'
})
export class FormulariosPlantillasComponent {
}
```

Para realizar el data-binding con los campos del formulario deberemos crear una variable en el controlador que será la que asociemos a los inputs de nuestro formulario en la vista. Definimos el modelo a usar en la variable articulo

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { Articulo } from '../../Modelos/articulo';

@Component({
  selector: 'app-formularios-plantillas',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './formularios-plantillas.component.html',
  styleUrl: './formularios-plantillas.component.css'
})
export class FormulariosPlantillasComponent {
  articulo:Articulo={id:"",
                     nombre:"",
                     descripcion:"",
                     precio:0,
                     unidades:0}
}
```

La inicializamos con los valores que deseemos, en nuestro caso los string con la cadena vacía y los numbers a 0

Para poder realizar el data-binding deberemos introducir la directiva [(ngModel)] en cada uno de los campos del formulario y asociarle la variable con la que se hará el enlace de datos.

Muy importante, **al trabajar con ngModel deberemos asignar la propiedad name** a los inputs, en caso contrario Angular no podrá acceder a los valores de los controles del formulario.

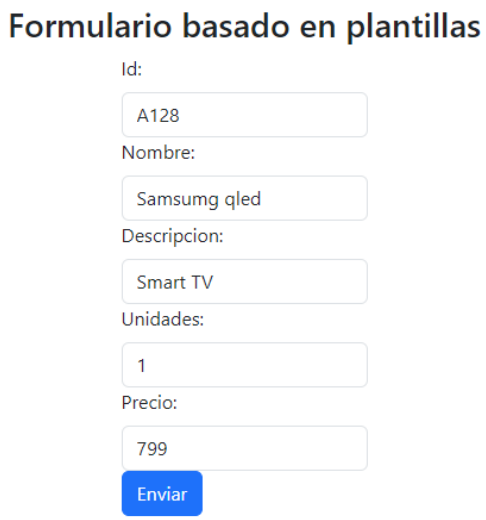
Vamos a añadir también al final del componente una interpolación de la variable articulo para poder visualizar el valor de la variable y ver que va cambiando cada vez que modifiquemos los controles del formulario. Para ello deberemos importar el CommonModule en el TS

```
@Component({
  selector: 'app-formularios-plantillas',
  standalone: true,
  imports: [FormsModule,CommonModule],
  templateUrl: './formularios-plantillas.component.html',
  styleUrl: './formularios-plantillas.component.css'
})
```

Añadimos el pipe json en la interpolación

```
{{articulo | json}}
```

El resultado es:




{ "id": "A128", "nombre": "Samsumg qled", "descripcion": "Smart TV", "precio": 799, "unidades": 1 }

Cada vez que modifiquemos el valor de un campo aparecerá reflejado en el objeto artículo que usamos para el two-way data binding

Si ahora pulsáramos en el botón de enviar los datos no se mandarían a la url especificada, pues al incluir el FormsModule se ha desactivado el funcionamiento tradicional de los forms. Para poder procesar los datos deberemos recurrir a otra forma.

Ahora ya no nos hará falta action en la etiqueta del form y para poder procesar el envío utilizaremos el método **ngSubmit**. Este método se ejecutará cada vez que se intente mandar el formulario.

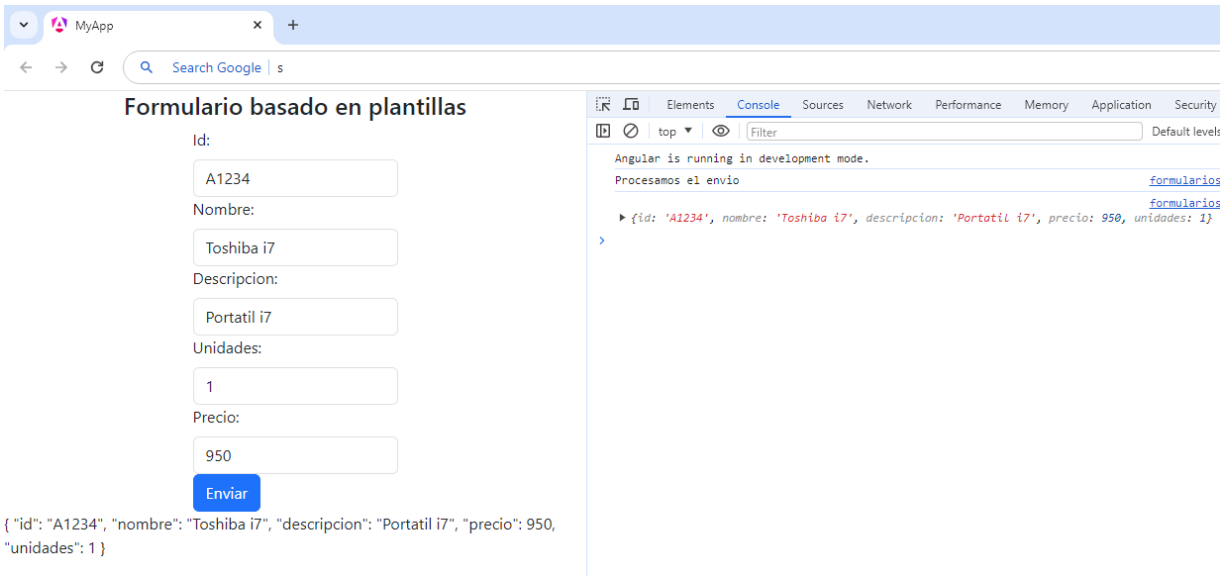
Asignamos un método de nuestro controlador que gestione el envío, en nuestro caso voy a asignarle un método llamado onSubmit que lo que hará es mostrar por consola los datos del formulario.

```
<form (ngSubmit)="onSubmit()">
```

Creamos el método

```
  onSubmit(){
    console.log('Procesamos el envio')
    console.log(this.articulo)
  }
```

El resultado es




De esta manera hemos conseguido acceder a los datos de nuestro formulario, en el ejemplo los hemos mostrado por la consola, pero podríamos hacer lo que queramos con ellos, lo más habitual sería mandarlos a una API REST para poder tener una BD de datos en la cual nuestra información se quedara almacenada.

## 2.1.- Validaciones

Muchas veces nos interesa realizar validaciones sobre los campos del formulario, es decir, nos interesa que la información introducida cumpla unas características, y si no se cumple la validación que nos muestre un mensaje de error indicándolo y que además no se pueda mandar los datos.

Vamos a ver cómo podemos realizar las validaciones en formularios basados en plantillas. **Hemos de definir las validaciones en la plantilla.**

Vamos a poner los siguientes criterios de validación:

- Como queremos que los campos de nuestro formulario contengan información vamos añadirles la **propiedad required**, de esa manera habrá un error si el usuario no introduce nada
- Vamos a poner que el nombre y la descripción tengan longitud mínima de 5 caracteres, para ello añadiremos **la propiedad minlength**
- Vamos aponer que el precio y las unidades sean positivas, para ello vamos a añadir **la propiedad min**

Para poder comprobar el estado de nuestros campos deberemos **crear una variable de plantilla para cada uno de ellos** y asignarles el valor de **ngModel**, de esta manera podremos acceder al estado del input.

Incluimos las variables de plantilla en los inputs introduciendo #id, #nombre, #descripcion, #precio y #unidades en sus correspondientes inputs.

El formulario quedaría así:

```
<form (ngSubmit)="onSubmit()">
  <label class="form-label">Id:</label>
  <input type="text" class="form-control" placeholder="Id"
         [(ngModel)]="articulo.id" name="id" required #id="ngModel">

  <label class="form-label">Nombre:</label>
  <input type="text" class="form-control" placeholder="Nombre"
        [(ngModel)]="articulo.nombre" name="nombre" required minlength="5"
#nombre="ngModel">

  <label class="form-label">Descripcion:</label>
  <input type="text" class="form-control" placeholder="Descripcion"
        [(ngModel)]="articulo.descripcion" name="descripcion" required
minlength="5" #descripcion="ngModel">

   <label class="form-label">Unidades:</label>
   <input type="number" class="form-control" placeholder="Unidades"
        [(ngModel)]="articulo.unidades" name="unidades" required min="1"
#unidades="ngModel">

   <label class="form-label">Precio:</label>
   <input type="number" class="form-control" placeholder="Precio"
        [(ngModel)]="articulo.precio" name="precio" required min="1"
#precio="ngModel">

   <div>
     <button class="btn btn-primary">Enviar</button>
   </div>
</form>
```

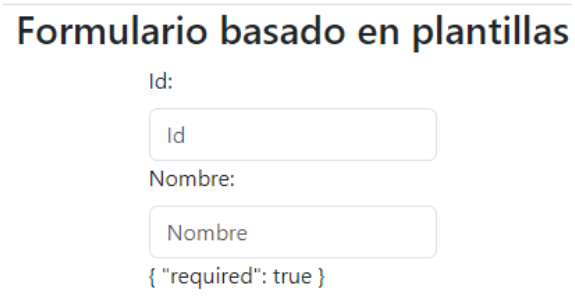
Ahora ya podemos **acceder al estado de nuestros inputs a través de la propiedad errors de cada input referenciado por su variable de tipo plantilla**.

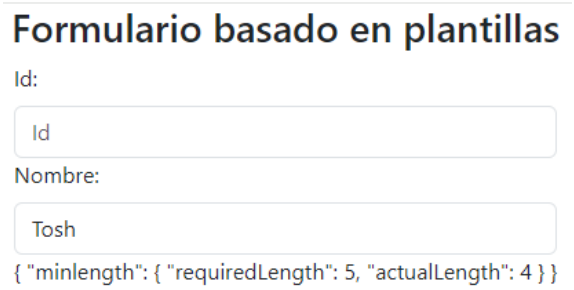
Este es un array que se creara con el nombre de los elementos de validación que no se cumplen, si no hay errores esta propiedad será null

Vamos a interpolar los errores del campo nombre, para ello añadimos al final del componente la interpolación del campo nombre usando el pipe json.

```
<label class="form-label">Nombre:</label>
<input type="text" class="form-control" placeholder="Nombre"
      [(ngModel)]="articulo.nombre" name="nombre" required minlength="5"
#nombre="ngModel">

<p>{{nombre.errors | json}}</p>
```

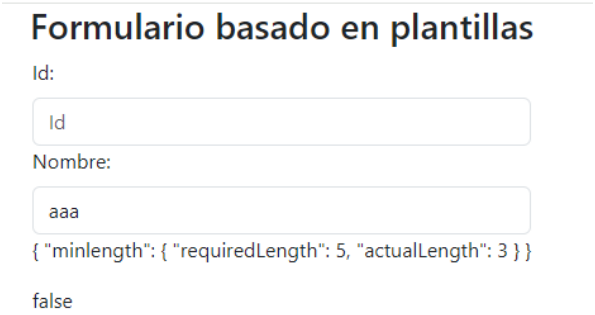
El resultado al cargar el formulario es:




También podemos trabajar con la **propiedad valid o invalid** del campo, esta propiedad lo que nos mostrara es true o false, en función de que se cumplan las reglas de validación indicadas o no. Para comprobarlo vamos a interpolar también dicha propiedad al final del componente.

```
<label class="form-label">Nombre:</label>
  <input type="text" class="form-control" placeholder="Nombre"
        [(ngModel)]="articulo.nombre" name="nombre" required minlength="5"
#nombre="ngModel">
<p>{{nombre.errors | json}}</p>
<p>{{nombre.valid  | json}}</p>
```

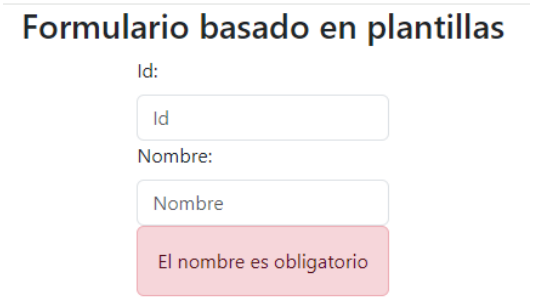



De esta manera podemos comprobar si hay errores en el campo y si un campo es válido

Con el valor de la propiedad valid podríamos mostrar u ocultar un mensaje de error para cuando no se cumpla la regla de validación. Para que nos aparezcan los mensajes de error vamos a incluir en nuestras vistas unas etiquetas de error para cada uno de los campos en función de la propiedad invalid.

```
<label class="form-label">Nombre:</label>
<input type="text" class="form-control" placeholder="Nombre"
       [(ngModel)]="articulo.nombre" name="nombre" required minlength="5"
#nombre="ngModel">
@if(nombre.invalid){
   <div class="alert alert-danger">El nombre es obligatorio</div>
}
```

El resultado sería:




Con esto hemos conseguido que se muestre un mensaje cuando haya un error en algún campo que disponga de validaciones. Aun así, el usuario podría intentar mandar el formulario pulsando en el botón de enviar. Lo habitual sería no dejar mandar el formulario mientras haya errores en la información de los campos. **Para ello podemos utilizar las propiedades valid o invalid del formulario.**

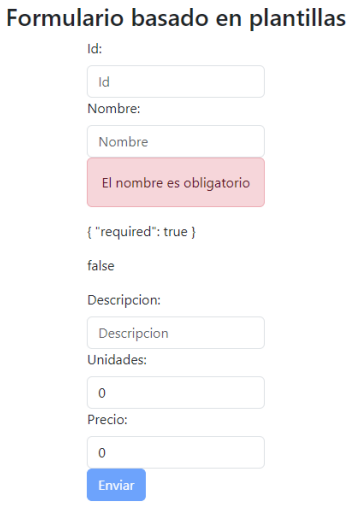
Utilizaremos un property-binding de la propiedad disabled del formulario con el valor de la propiedad invalid del formulario

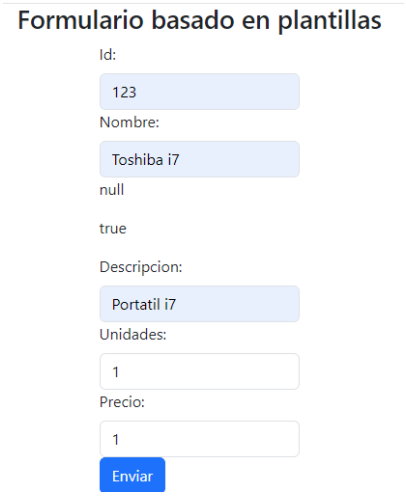
Para esto tendremos que referenciar el formulario con un nombre

```
<form (ngSubmit)="onSubmit()" #miFormulario="ngForm">
```

De esta manera el botón de enviar solo estará habilitado cuando no haya ningún error en los campos del formulario.

```
<div>
 <button class="btn btn-primary" disabled="miFormulario.invalid">Enviar</button>
</div>
```



Con esto hemos conseguido que se muestre un mensaje cuando haya un error y que no se pueda mandar el formulario si hay errores. Pero aún tenemos un problema y es que cuando se carga el componente y se muestra el formulario, el campo está vacío, pero es porque aún no hemos escrito nada en él. Como da error se muestra el mensaje.

Para poder solucionar este problema deberemos trabajar con los estados de los controles inputs.

## 2.2.- Estados

Lo que realmente queremos hacer es que se muestre el mensaje cuando no se cumpla la regla de validación y además aún no se haya escrito en el control. Para esto Angular dispone de unas clases asociadas a los inputs que indican su estado.

Las clases que utiliza Angular para determinar el estado del input son:

- **ng-untouched:** indica que aún no se ha entrado en el control
- **ng-pristine**: indica que aún no se ha modificado el contenido del input
- **ng-valid**: indica que el contenido es válido, es decir cumple las reglas de validación

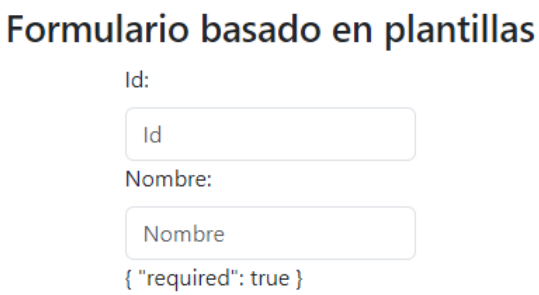
Estos estados del input irán cambiando en función de lo que hagamos sobre el input.

- Si entramos al input se modificara el estado de **touched a untouched**.
- Si modificamos el contenido del input se modificará el estado de **pristine a dirty**
- Si el contenido es válido se modificará el estado de **invalid a valid**

Lo que nos interesaría seria que los mensajes de error aparecieran cuando haya un error y además se haya entrado a ese campo o se haya escrito en él. Con lo cual deberíamos modificar la lógica del @if para que se cumplan estas condiciones.

```
@if(nombre.invalid && (nombre.dirty || nombre.touched)){
  <div class="alert alert-danger">El nombre es obligatorio</div>
}
```

De esta manera sólo aparecerá el error cuando ya se haya entrado en el campo o se haya modificado el valor



## 3.- Formularios Reactivos

Hemos visto los formularios basados en plantillas que son muy útiles cuando trabajamos con formularios pequeños, **cuando trabajemos con formularios grandes nos será más interesante trabajar con formularios reactivos.**

En los formularios reactivos seguimos teniendo el acceso a las propiedades de los controles para determinar los errores y los estados, la diferencia va ser donde y como se realizarán.

Al igual que para trabajar con formulario basado en plantilla s tuvimos que importar el módulo Formsmodule para trabajar con formularios reactivos **deberemos importar el módulo ReactiveFormsModule**.

Creamos un componente nuevo y añadimos el módulo ReactiveFormsModule

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

@Component({
  selector: 'app-formulario-reactivo',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './formulario-reactivo.component.html',
  styleUrl: './formulario-reactivo.component.css'
})
export class FormularioReactivoComponent {

}
```

Una vez añadido al módulo principal ya podremos trabajar con él en nuestros componentes. Creamos ahora un formulario al igual que hicimos en el caso de formularios basados en plantillas.

```
<div class="container">
    <form>
        <label class="form-label">Id:</label>
        <input type="text"
            class="form-control"
            placeholder="Id">

        <label class="form-label">Nombre:</label>
        <input type="text"
            class="form-control"
            placeholder="Nombre">

        <label class="form-label">Descripcion:</label>
        <input type="text"
```

```
            class="form-control"
            placeholder="Descripcion">

        <label class="form-label">Unidades:</label>
        <input type="number"
            class="form-control"
            placeholder="Unidades">

        <label class="form-label">Precio:</label>
        <input type="number"
            class="form-control"
            placeholder="Precio">

            <div>
            <button class="btn btn-primary">Enviar</button>
        </div>
    </form>
</div>
```

A partir de este formulario básico de HTML vamos a implementar un formulario reactivo, para ello haremos uso de diferentes elementos de Angular:

- FormGroup y FormControl
- Validators
- FormBuilder

## 3.1.- FormGroup y FormControl

Para poder gestionar los controles del formulario tenemos que asociar los campos del formulario con variables creadas en el controlador y para poder gestionar el formulario deberemos asignar el formulario a otra variable del controlador

- Para vincular los datos de los inputs con las variables creadas en los formularios reactivos debemos indicar que las variables son una **instancia de la clase FormControl**, para ello deberemos importar en nuestro componente dicha clase.
- Para vincular el formulario con todos esos controles deberemos usar una **instancia de la clase FormGroup**, para ello deberemos importar en nuestro componente dicha clase.

Importamos en nuestro componente las clases y creamos las variables de la instancia FormControl. Esta clase permite asignar el valor inicial del control y posteriormente veremos que también podremos añadirle un array de validaciones.

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule } from
'@angular/forms';

@Component({
  selector: 'app-formulario-reactivo',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './formulario-reactivo.component.html',
  styleUrl: './formulario-reactivo.component.css'
})
export class FormularioReactivoComponent {

  miFormulario: FormGroup = new FormGroup({
    id: new FormControl(''),
    nombre: new FormControl(''),
    descripcion: new FormControl(''),
    unidades: new FormControl(0,),
    precio: new FormControl(0,),
  })

}
```

Finalmente, para poder vincular el formulario y los inputs con las variables deberemos añadir en la plantilla:

- Al formulario la propiedad formGroup con la variable de tipo FormGroup
- A cada input la propiedad formControlName con el valor de cada variable.de tipo FormControl

```
<form [formGroup]="miFormulario" >
  <label class="form-label">Id:</label>
   <input type="text"
          class="form-control"
          placeholder="Id"
          formControlName="id">

  <label class="form-label">Nombre:</label>
   <input type="text"
          class="form-control"
          placeholder="Nombre"
          formControlName="nombre">

  <label class="form-label">Descripcion:</label>
  <input type="text"
         class="form-control"
         placeholder="Descripcion"
```

```
        formControlName="descripcion">

  <label class="form-label">Unidades:</label>
  <input type="number"
  class="form-control"
  placeholder="Unidades"
  formControlName="unidades">

  <label class="form-label">Precio:</label>
  <input type="number"
        class="form-control"
        placeholder="Precio"
        formControlName="precio">

  <div>
    <button class="btn btn-primary">Enviar</button>
  </div>
</form>
```

Finalmente, para poder comprobar que se han vinculado los inputs a las variables vamos a interpolar los valores de las variables.

En este caso, al contrario que los formularios basados en plantillas, los controles están vinculados a cada FormControl del FormGroup, por ello deberemos interpolar con esos valores.

Para sacar el nombre deberemos de interpolar:

```
{{miFormulario.get('nombre')?.value | json}}
```

miFormulario hace referencia al FormControl, de ahí accedemos a los controles mediante el método get y el nombre del campo, una vez obtenido el campo accedemos al valor mediante la propiedad value. Finalmente usamos el pipe json para mostrar el objeto.

Los interrogantes son porque el método get puede hacer referencia a un control que no exista, al usar el ? indica que es opcional y así no genera error.

Si mostramos los datos de nombre y unidades

```
<p>{{miFormulario.get('nombre')?.value   | json}}</p>
<p>{{miFormulario.get('unidades')?.value | json}}</p>
```

El resultado será:


Esta forma de acceder desde la plantilla a los controles del controlador puede llegar a ser un poco “pesada” por tener que especificar todo. Hay una alternativa que nos simplifica la notación en la plantilla, esta forma es usar los getters para cada control.

Lo que vamos a hacer es definir métodos get para cada control que devolverán el value de ese control, de esta manera en la plantilla podremos utilizar el nombre del control

```
  get id(){
    return this.miFormulario.get('id')
  }
  get nombre(){
    return this.miFormulario.get('nombre')
  }
  get descripcion(){
    return this.miFormulario.get('descripcion')
  }
  get unidades(){
    return this.miFormulario.get('unidades')
  }
  get precio(){
    return this.miFormulario.get('precio')
  }
```

Ahora desde la plantilla para interpolar utilizaremos:

```
<p>{{nombre?.value   | json}}</p>
<p>{{unidades?.value | json}}</p>
```

Esto nos facilita mucho la escritura del código y además lo utilizaremos para acceder a los controles del formulario para las validaciones

## 3.2.- Validators

Las validaciones en los formularios reactivos se realizan de forma diferente a los formularios basados en plantillas.

En los formularios basados en plantillas poníamos las reglas de validación en el input, es decir en la plantilla, por ejemplo, indicábamos que los campos fueran requeridos.

**En los formularios reactivos todas las validaciones se van a realizar en el controlador**, para ello deberemos importar **una clase llamada Validators**.

Una vez importada la clase validators a nuestro componente

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule, Validators } from
'@angular/forms';
```

Ya podemos incluir las reglas de validación, para ello en la instancia de cada variable, añadimos la regla utilizando la clase validators. Deberemos incluir el tipo de regla, después del valor inicial.

```
miFormulario: FormGroup = new FormGroup({
    id: new FormControl('', Validators.required),
    nombre: new FormControl('',[Validators.required, Validators.minLength(5)]),
    descripcion: new FormControl('',[Validators.required,
Validators.minLength(5)]),
    unidades: new FormControl(0,[Validators.required, Validators.min(1)]),
    precio: new FormControl(0,[Validators.required, Validators.min(1)])
  })
```

Los Validators son un array de validadores que ya viene definidos en Angular y que solo hemos de añadirlos como reglas de validación.

Angular proporciona un conjunto de validadores integrados para facilitar la validación de datos en formularios. Estos validadores se utilizan para verificar si los valores introducidos por el usuario cumplen con ciertos criterios, como ser requeridos, tener un formato específico o estar dentro de un rango determinado.

Tipos de validadores:

- **required:** Comprueba si el valor es nulo o una cadena vacía.
- **email:** Valida el formato de una dirección de correo electrónico.
- **min:** Comprueba si el valor es mayor o igual que un valor mínimo especificado.
- **max:** Comprueba si el valor es menor o igual que un valor máximo especificado.
- **minlength:** Valida la longitud mínima de una cadena.
- **maxlength:** Valida la longitud máxima de una cadena.
- **pattern:** Comprueba si el valor coincide con una expresión regular especificada.

También existe la posibilidad de crear validadores personalizados.

Ahora ya podríamos comprobar si el valor de los inputs cumple las reglas de validación y mostrar un mensaje de error en caso de que no lo hiciera. Para ello deberíamos comprobar los estados de los inputs de forma similar a la que hicimos en los formularios basados en plantillas.

Si no se cumple la regla de validación:

- El control tendrá el valor de invalid
- Se generará un array llamado errors con cada tipo de error producido en la validación

Vamos a comprobar el error para el nombre, que ha de ser requerido y con mínimo 5 caracteres. Vamos a comprobar que se cumplen las dos validaciones que sea requerido y con al menos 5 caracteres.

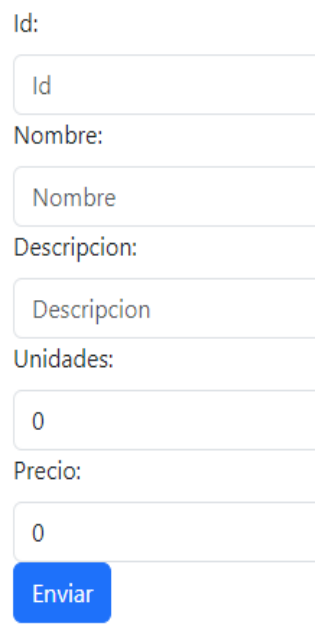
Incluimos en la plantilla los divs para mostrar los errores.

```
@if(nombre?.invalid && (nombre?.dirty || nombre?.touched)){
  <div class="alert alert-danger">
  @if(nombre?.errors?.['required']){
    <div> Nombre es obligatorio.</div>
  }
  @if(nombre?.errors?.['minlength']){
    <div >Nombre debe tener al menos 5 caracteres. </div>
  }
  </div>
}
```

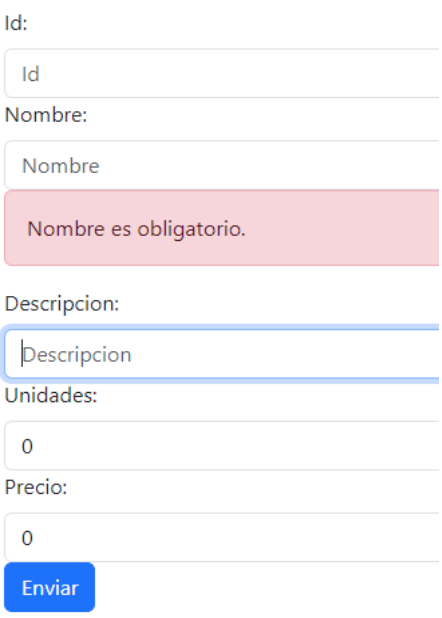
De esta manera estamos comprobando:

- Que el campo sea invalido y que se haya entrado en él (igual que en formularios basados en plantilla)
- Si el control es invalido es porque se ha producido un error y aparecerá en el array de errors del control.
- Accedemos a cada posible error para mostrar en su caso el div de error correspondiente.

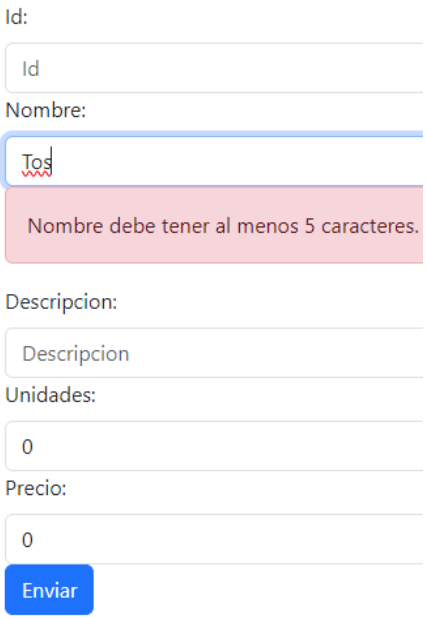
Ahora ya se muestran los divs de error solo cuando se ha entrado al campo nombre y si se ha dejado vacío o con menos de 5 caracteres.




Inicio sin nada




Entrar al campo sin poner nada




Escribir menos de 5 caracteres

/Una vez gestionado los errores, para poder gestionar el envío del formulario deberemos añadir en la etiqueta de form el evento que gestiona el envío del formulario, **en los formularios reactivos es submit**.

```
<form [formGroup]="miFormulario" (submit)="onSubmit()">
```

En este caso estoy llamando al envento onSubmit que será el encargado de manipular los datos del formulario

```
onSubmit(){
    console.log('Procesamos Formulario...')
    console.log (this.miFormulario.value)
}
```

Ahora nos faltaría igual que en los formularios basados en plantilla acceder al estado del formulario para poder procesarlo, puesto que sería recomendable que solo se pudiera mandar el formulario si no hay ningún tipo de error. Esto lo hacíamos deshabilitando el botón de enviar en función del valor invalid del formulario

```
<button class="btn btn-primary"
        [disabled]="miFormulario.invalid">Enviar</button>
```

De esta manera solo se activará el botón cuando el formulario sea válido o lo que es l mismo estará deshabilitado mientras haya algún tipo de error.

## 3.3.- FormBuilder

Hemos visto cómo podemos definir los campos de nuestro **formulario Reactivo** en un **FormGroup** mediante la instancia de **FormControl** y el uso de **Validators**.

Todo funciona bien, pero si tuviéramos muchos campos en nuestro formulario tendríamos que declararlos todos en el FormGroup instanciándolos mediante la clase FormControl. Esto puede ser muy repetitivo y bastante tedioso, Angular nos proporciona otra forma de definir un FormGroup, esta forma es utilizando la clase **FormBuilder**.

Para poder utilizar la clase FormBuilder **primero deberemos de importarla en nuestro componente**.

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

Una vez importado **deberemos inyectarlo** en el componente, para ello realizaremos lo mismo que hacíamos con los servicios, para ello definiremos una variable de este tipo FormBuilder en nuestro constructor.

```
constructor(private fb:FormBuilder){}
```

Para definir nuestro FormGroup utilizaremos la variable fb creada en el constructor utilizando el **método group** que nos permitirá definir los controles del formulario. Los controles del formulario se definen como un objeto dentro del método group, dentro de este objeto colocamos lo mismo que poníamos en el FormGroup pero sin añadir la instancia de FormControl. En este caso, cada control se definirá como un array en el cual la primera posición será el valor por defecto y la segunda posición serán las validaciones a aplicar en ese control

```
miFormulario: FormGroup = this.fb.group({
    id: ['', Validators.required],
    nombre: ['',[Validators.required, Validators.minLength(5)]],
    descripcion: ['',[Validators.required, Validators.minLength(5)]],
    unidades: [0,[Validators.required, Validators.min(1)]],
    precio: [0,[Validators.required, Validators.min(1)]]
})
```

Esta forma de definir el grupo para el formulario es igual que la anterior pero de esta forma nos hemos evitado tener que definir cada control utilizando new FormControl y además es más simple y limpio.

El resultado es el mismo que antes, se mostraran los errores indicados por las validaciones, el botón de enviar estará deshabilitado mientras haya algún error en los controles y el valor del formulario recogerá los valores del formulario en cada momento