13 - Formularios : formulario basado en plantillas

Angular proporciona dos enfoques diferentes para manejar formularios:

1. Formularios basados en plantillas
2. Formularios reactivos

Ambos capturan los eventos de entrada del usuario desde la vista, crean un modelo de formulario y un modelo de datos para actualizar y proporcionan una forma de rastrear los cambios.

Los formularios basados en plantillas son útiles para agregar un formulario simple a una aplicación. Son más fáciles de agregar que los formularios reactivos, pero no escalan tan bien como los formularios reactivos. Si tiene requisitos de formulario y lógica muy básica podemos utilizar sin problemas formularios basados en plantillas.

Veremos en este concepto como implementar en Angular formularios basados en plantillas.

Formulario reactivo (directiva ngModel)

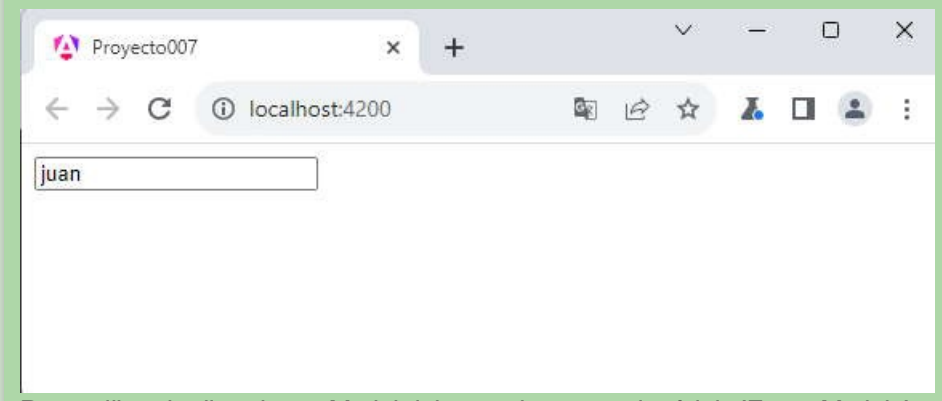
Por ejemplo si en la clase AppComponent tenemos la propiedad 'nombre' con el valor 'juan':

```
nombre='juan';
```

Luego en la vista definimos la directiva ngModel entre corchetes y le asignamos el nombre de la propiedad definida en la clase:

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

Cuando arrancamos la aplicación podemos observar que el control input aparece automáticamente con el valor 'juan':



Para utilizar la directiva ngModel debemos importar el módulo 'FormsModule':

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  nombre = 'juan';
}
```

Lo que hay que tener en cuenta que es un enlace en una única dirección: el valor de la propiedad de la clase se refleja en la interfaz visual. Si el operador cambia el contenido del control 'input' por ejemplo por el nombre 'ana' luego la propiedad 'nombre' de la clase sigue almacenando el valor 'juan'.

Si queremos que el enlace sea en las dos direcciones debemos utilizar la siguiente sintaxis:

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

Un primer ejemplo muy corto que podemos hacer es crear un proyecto (proyecto007) para que se ingrese el nombre y apellido de una persona y se muestre inmediatamente en la parte inferior.

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

@Component({
  selector: 'app-root',
  imports: [RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  nombre = '';
  apellido = '';
}
```

El archivo app.component.html donde definimos las directiva ngModel para cada control es:

```
<div>
  <p>Ingrese nombre<input type="text" [(ngModel)]="nombre"></p>
  <p>Ingrese apellido<input type="text" [(ngModel)]="apellido"></p>
  <p>Nombre completo:{{nombre}},{{apellido}}</p>
</div>

<router-outlet />
```

Ejecutemos nuestra aplicación desde la línea de comandos de Node.js.

En el navegador nos muestra como se actualizan las propiedades cada vez que ingresamos un caracter en los controles 'input':

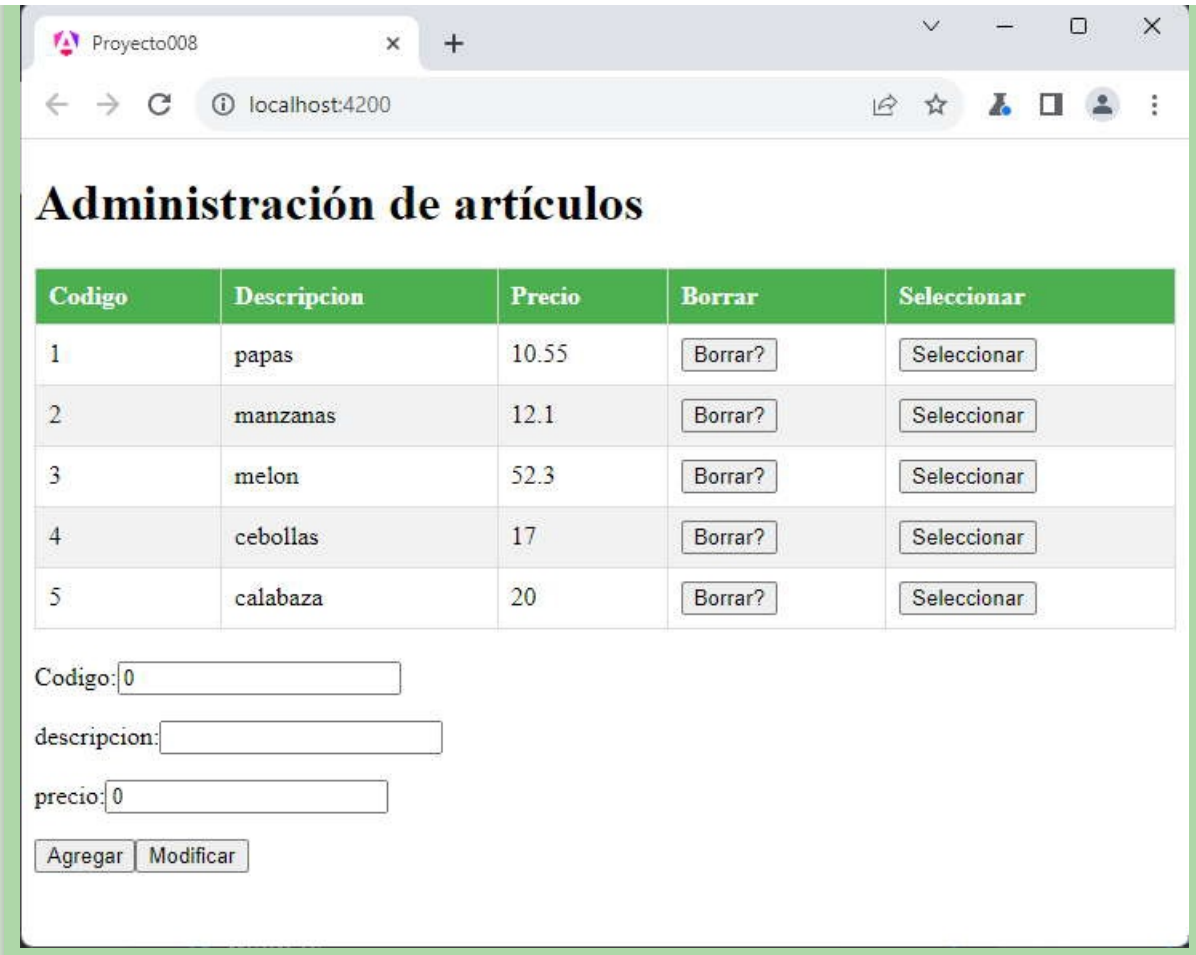
Es importante entender que hay un enlace en ambas direcciones. Si cambiamos el valor mediante el ingreso de datos por teclado en el control input, luego se ven reflejados en forma automática en las propiedades de la clase. De la misma forma si cambiamos el valor de la propiedad en la clase, luego se refleja en forma automática el valor en el formulario.

Podemos probar esta aplicación en la web aquí.

Problema

Confeccionar una aplicación que permita administrar un vector de objetos que almacena en cada elemento el código, descripción y precio de un artículo. Se debe poder agregar, borrar y modificar los datos de un artículo.

La interfaz visual de la aplicación debe ser similar a esta:



Podemos probar esta aplicación en la web aquí.

Crearemos el proyecto008 para resolver este problema.

1. Lo primero que debemos hacer es desde la línea de comandos de Node.js proceder a crear el proyecto008:

   ```
   ng new proyecto008
   ```

2. Para facilidad por el momento crearemos todo en una única componente, es decir en la que se crea automáticamente al crear el proyecto.

   En el archivo app.component.css implementamos la hoja de estilo para la tabla HTML:

```
/* Estilo para la tabla */
table {
  width: 100%;
  border-collapse: collapse;
  margin-bottom: 20px;
}

/* Estilo para las celdas de encabezado */
th, td {
  border: 1px solid #dddddd;
  padding: 8px;
  text-align: left;
}

/* Estilo alternado para filas */
tr:nth-child(even) {
  background-color: #f2f2f2;
}

/* Estilo para el encabezado de la tabla */
th {
  background-color: #4CAF50;
  color: white;
}
```

En el archivo app.component.html es donde mostramos una tabla HTML y un formulario para ingresar datos:

```
<div>
  <h1>Administración de artículos</h1>
  @if (hayRegistros()) {
  <table>
    <thead>
      <tr>
        <th>Codigo</th>
        <th>Descripcion</th>
        <th>Precio</th>
        <th>Borrar</th>
        <th>Seleccionar</th>
      </tr>
    </thead>
    <tbody>
      @for(art of articulos; track art.codigo) {
      <tr>
        <td>{{art.codigo}}</td>
        <td>{{art.descripcion}}</td>
        <td>{{art.precio}}</td>
        <td><button (click)="borrar(art.codigo)">Borrar?</button><
```

```
/td>
        <td><button (click)="seleccionar(art)">Seleccionar</button
></td>
      </tr>
      }
    </tbody>
  </table>
  } @else {
  <p>No hay articulos.</p>
  }

  <div>
    <p>
      Codigo:<input type="number" [(ngModel)]="art.codigo" />
    </p>
    <p>
      descripcion:<input type="text" [(ngModel)]="art.descripcion"
 />
    </p>
    <p>
      precio:<input type="number" [(ngModel)]="art.precio" />
    </p>
    <p>
      <button (click)="agregar()">Agregar</button>
      <button (click)="modificar()">Modificar</button>
    </p>
  </div>
</div>

<router-outlet />
```

3. La clase app.component.ts tenemos:

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

@Component({
  selector: 'app-root',
  imports: [RouterOutlet, FormsModule],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  art = {
    codigo: 0,
    descripcion: "",
```

```
    precio: 0
  }

  articulos = [{ codigo: 1, descripcion: 'papas', precio: 10.55 },
  { codigo: 2, descripcion: 'manzanas', precio: 12.10 },
  { codigo: 3, descripcion: 'melon', precio: 52.30 },
  { codigo: 4, descripcion: 'cebollas', precio: 17 },
  { codigo: 5, descripcion: 'calabaza', precio: 20 },
  ];

  hayRegistros() {
    return this.articulos.length > 0;
  }

  borrar(codigo: number) {
    for (let x = 0; x < this.articulos.length; x++)
      if (this.articulos[x].codigo == codigo) {
        this.articulos.splice(x, 1);
        return;
      }
  }

  agregar() {
    if (this.art.codigo == 0) {
      alert('Debe ingresar un código de articulo distinto a cero')
;
      return;
    }
    for (let x = 0; x < this.articulos.length; x++)
      if (this.articulos[x].codigo == this.art.codigo) {
        alert('ya existe un articulo con dicho codigo');
        return;
      }
    this.articulos.push({
      codigo: this.art.codigo,
      descripcion: this.art.descripcion,
      precio: this.art.precio
    });
    this.art.codigo = 0;
    this.art.descripcion = "";
    this.art.precio = 0;
  }

  seleccionar(art: { codigo: number; descripcion: string; precio:
number; }) {
    this.art.codigo = art.codigo;
    this.art.descripcion = art.descripcion;
```

```
    this.art.precio = art.precio;
  }

  modificar() {
    for (let x = 0; x < this.articulos.length; x++)
      if (this.articulos[x].codigo == this.art.codigo) {
        this.articulos[x].descripcion = this.art.descripcion;
        this.articulos[x].precio = this.art.precio;
        return;
      }
    alert('No existe el código de articulo ingresado');
  }
}
```

4. Ejecutemos la aplicación desde la ventana de Node.js mediante el comando:

```
ng serve -o
```

Listado

Pasemos a analizar las distintas partes de nuestra aplicación. Los archivos app.component.ts y app.component.html están totalmente integrados y con objetivos bien definidos cada uno. El archivo '.html' almacena la vista y el archivo '.ts' almacena el modelo de datos.

Definimos en el modelo (app.component.ts) un vector de objetos llamado 'articulos' y almacenamos 5 elementos:

```
articulos = [{ codigo: 1, descripcion: 'papas', precio: 10.55 },
{ codigo: 2, descripcion: 'manzanas', precio: 12.10 },
{ codigo: 3, descripcion: 'melon', precio: 52.30 },
{ codigo: 4, descripcion: 'cebollas', precio: 17 },
{ codigo: 5, descripcion: 'calabaza', precio: 20 },
];
```

En la vista (app.component.html) verificamos con un if si el vector tienen elementos:

```
@if (hayRegistros()) {
```

Como vemos llamamos al método 'hayRegistros()' que se encuentra implementado en el archivo *.ts:

```
hayRegistros() {
  return this.articulos.length > 0;
}
```

Si retorna true luego generamos una tabla HTML que muestre los datos del modelo y lo recorremos mediante un @for:

```
<table>
    <thead>
      <tr>
        <th>Codigo</th>
        <th>Descripcion</th>
        <th>Precio</th>
        <th>Borrar</th>
        <th>Seleccionar</th>
      </tr>
    </thead>
    <tbody>
      @for(art of articulos; track art.codigo) {
      <tr>
        <td>{{art.codigo}}</td>
        <td>{{art.descripcion}}</td>
        <td>{{art.precio}}</td>
        <td><button (click)="borrar(art.codigo)">Borrar?</button></td>
        <td><button (click)="seleccionar(art)">Seleccionar</button></td>
      </tr>
      }
    </tbody>
  </table>
```

Disponemos en cada fila dos botones y definimos sus respectivos eventos 'click' para que al ser presionados llamen a métodos del modelo para borrar o seleccionar el artículo respectivo. Los métodos envían como parámetro el artículo para saber cual borrar o seleccionar.

En la vista disponemos una serie de 'input' que nos permiten ingresar el código, descripción y precio de un artículo:

```
  <div>
    <p>
      Codigo:<input type="number" [(ngModel)]="art.codigo" />
    </p>
    <p>
      descripcion:<input type="text" [(ngModel)]="art.descripcion" />
    </p>
    <p>
      precio:<input type="number" [(ngModel)]="art.precio" />
    </p>
    <p>
      <button (click)="agregar()">Agregar</button>
      <button (click)="modificar()">Modificar</button>
    </p>
  </div>
```

Agregado

Podemos comprobar que los controles HTML tienen la directiva 'ngModel' bidireccional, es decir que cuando el operador carga un dato en el primer 'input' se actualiza automáticamente en el modelo el dato cargado en 'art.codigo':

Podemos comprobar que en el modelo tenemos definido un objeto llamado art con tres propiedes:

```
  art={
    codigo:0 ,
    descripcion:"",
    precio:0
  }
```

Al presionar el botón agregar se ejecuta el método 'agregar':

```
agregar() {
  if (this.art.codigo == 0) {
    alert('Debe ingresar un código de articulo distinto a cero');
    return;
  }
  for (let x = 0; x < this.articulos.length; x++)
    if (this.articulos[x].codigo == this.art.codigo) {
      alert('ya existe un articulo con dicho codigo');
      return;
    }
  this.articulos.push({
    codigo: this.art.codigo,
    descripcion: this.art.descripcion,
    precio: this.art.precio
  });
  this.art.codigo = 0;
  this.art.descripcion = "";
  this.art.precio = 0;
}
```

En este método primero recorremos el vector articulos para comprobar si hay algún otro artículo con el mismo código. En el caso que no exista procedemos a añadir un nuevo elemento llamando al método push y pasando un objeto que creamos en dicho momento con los datos almacenados en el objeto 'art' que se encuentra enlazado con el formulario.

Luego asignamos cero y cadena vacía a todas las propiedades del objeto art con el objetivo de borrar todos los 'input' del formulario.

Al agregar un elemento al vector 'Angular' se encarga de actualizar la vista sin tener que indicar nada en nuestro código.

Borrado

Cuando se presiona el botón de borrar se ejecuta el método 'borrar':

```
borrar(codigo: number) {
  for (let x = 0; x < this.articulos.length; x++)
    if (this.articulos[x].codigo == codigo) {
      this.articulos.splice(x, 1);
      return;
    }
}
```

Recorremos el vector y controlamos uno a uno el código del artículo seleccionado con cada uno de los elementos del vector. El que coincide lo eliminamos del vector llamando al método splice indicando la posición y cuantas componentes borrar a partir de ese.

Selección

Cuando se presiona el botón de seleccionar se ejecuta el método 'seleccionar':

```
seleccionar(art: { codigo: number; descripcion: string; precio: number; }) {
  this.art.codigo=art.codigo;
  this.art.descripcion=art.descripcion;
  this.art.precio=art.precio;
}
```

Lo único que hacemos es actualizar el objeto art del modelo con el artículo que acaba de seleccionar el operador (llega como parámetro el artículo seleccionado)

Modificación

Cuando presiona el botón de modificación se ejecuta el método:

```
modificar() {
  for(let x=0;x<this.articulos.length;x++)
    if (this.articulos[x].codigo==this.art.codigo)
    {
      this.articulos[x].descripcion=this.art.descripcion;
      this.articulos[x].precio=this.art.precio;
      return;
    }
  alert('No existe el código de articulo ingresado');
}
```

Buscamos el código de articulo del control 'input' dentro del vector, en caso de encontrarlo procedemos a modificar la descripción y precio.

Tener en cuenta que por ahora estamos haciendo aplicaciones que almacenan sus datos en forma local y no estamos enviando los mismos a un servidor, luego veremos distintas alternativas de enviar y recuperar datos de un servidor.

Recordar que podemos probar esta aplicación en la web aquí.

Retornar