Angular ngModel es un concepto clave en Angular que hay que explicar paso a paso vamos a empezar con ello .

El concepto más sencillo de entender es cuando tenemos un componente que posee una propiedad y deseamos que esa propiedad se muestre en una plantilla. La forma más sencilla es usar interpolación y solicitar a la plantilla que muestre el dato. Por ejemplo si tenemos un componente con la variable nombre.

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-c1',
  templateUrl: './c1.component.html',
  styleUrls: ['./c1.component.css']
})
export class C1Component implements OnInit {

  nombre:string="pedro"

  constructor() { }

  ngOnInit() {
  }

}
```

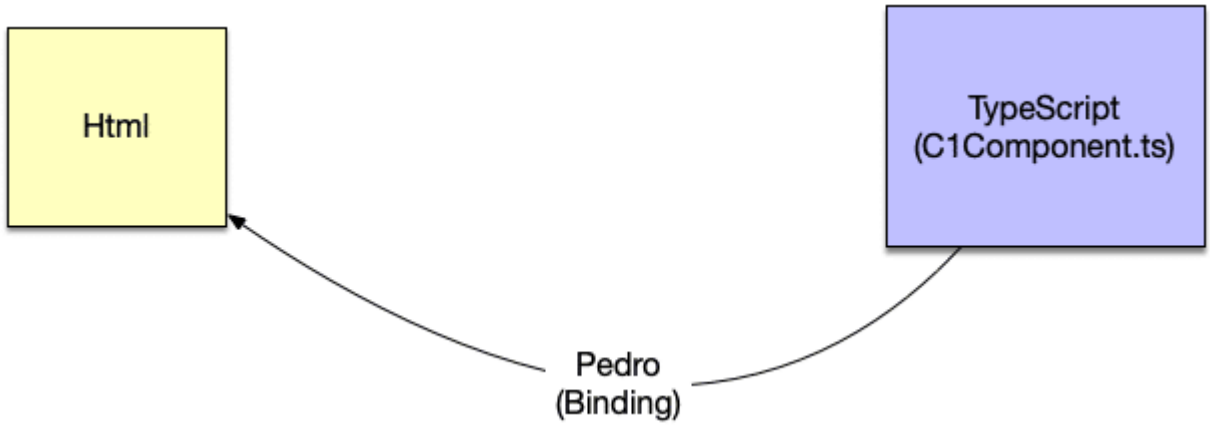
La plantilla mostrara la información usando interpolación:

```
<p>
  {{nombre}}
</p>
```

Podemos ver como la plantilla renderiza el resultado:

> pedro

Esto es lo que comúnmente se denomina Angular Binding y se trata de un binding unidireccional ,es decir del código de TypeScript a la plantilla. Es una de las partes en las que luego nos apoyaremos con Angular ngModel.



Si por ejemplo cambiamos en el constructor el valor del nombre al cabo de 2 segundos nos daremos cuenta que el dato quedará actualizado:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-c1',
  templateUrl: './c1.component.html',
  styleUrls: ['./c1.component.css']
})
export class C1Component implements OnInit {

  nombre:string="pedro"

  constructor() {
```

```
    setTimeout(()=> {

      this.nombre="ana";
    },2000);
  }

  ngOnInit() {
  }

}
```

Veremos en la plantilla el valor actualizado :).

> ana

Angular One Way Data Binding

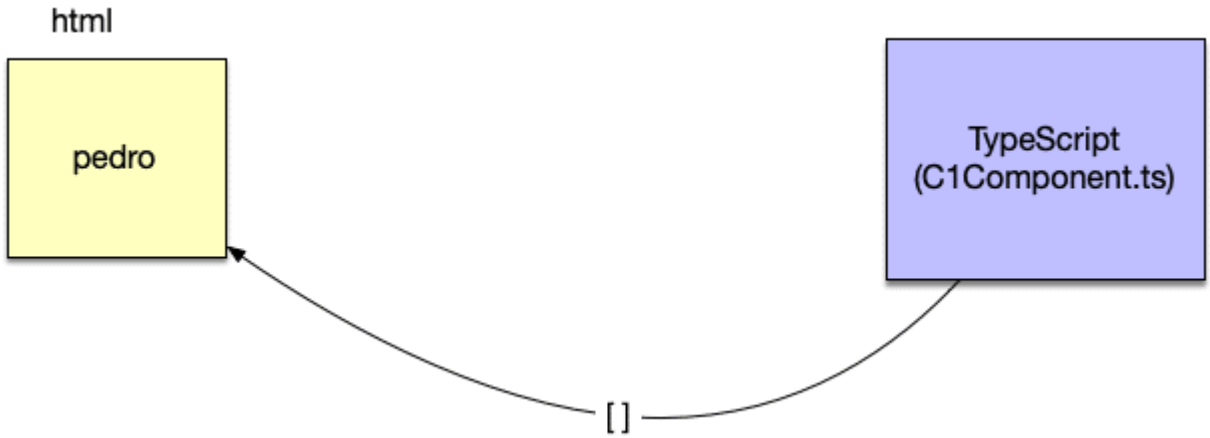
Acabamos de hace el ejemplo mas sencillo de bindings . Sin embargo en Angular los bindings a través de una interpolación son una especie de sintaxis sugar sobre los bindings reales . Es decir podríamos realizar el mismo tipo de binding de la siguiente forma:

```
<p [innerHtml]="nombre">
</p>
```

El resultado sería idéntico primero saldría Pedro y luego saldría Ana:

> ana

En este caso los corchetes [ ] definen un binding entre las variables de TypeScript y la plantilla del componente de Angular en una única dirección. Es lo que se denomina un binding unidireccional.



Angular y Eventos

De la misma manera en la cual nosotros podemos realizar bindings desde el código de TypeScript a la plantilla podemos hacer lo contrario. Es decir generar un evento que nos permita comunicarnos desde la plantilla con el componente y cambiar uno de sus valores. Para ello usaremos los paréntesis a la hora de ligar el evento con una función del componente de TypeScript.

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-c1',
  templateUrl: './c1.component.html',
  styleUrls: ['./c1.component.css']
})
export class C1Component implements OnInit {

  nombre:string="pedro"

  constructor() {

  }
```

```
  ngOnInit() {
  }

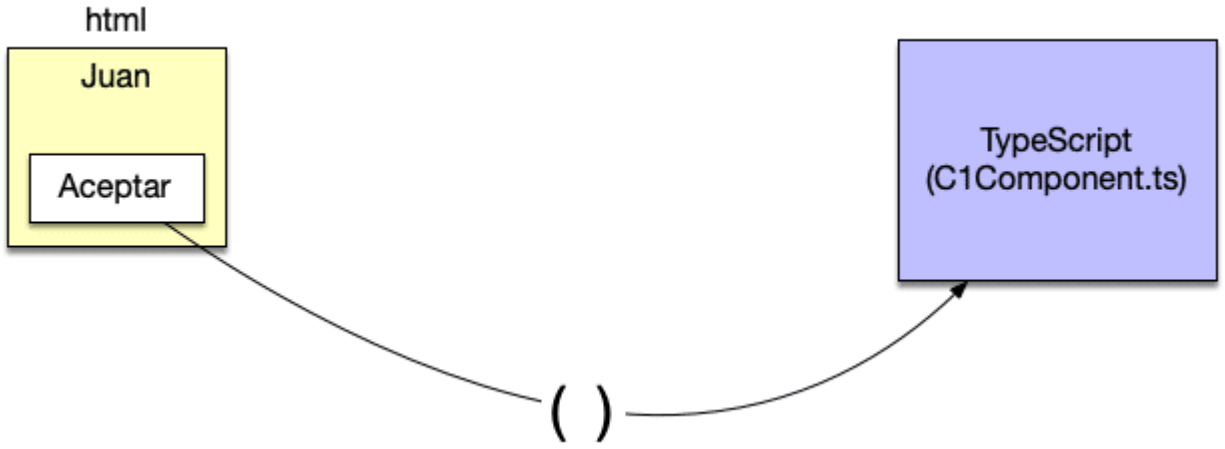
  cambiar() {

    this.nombre="juan";
  }
}
```

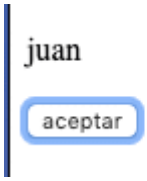
De esta manera la plantilla quedaría de la siguiente forma:

```
<p>
  {{nombre}}
</p>
<input type="button" value="aceptar" (click)="cambiar()"/>
```

Cuando pulsemos en el botón de aceptar automáticamente se cambiará el valor del nombre dentro de la plantilla:



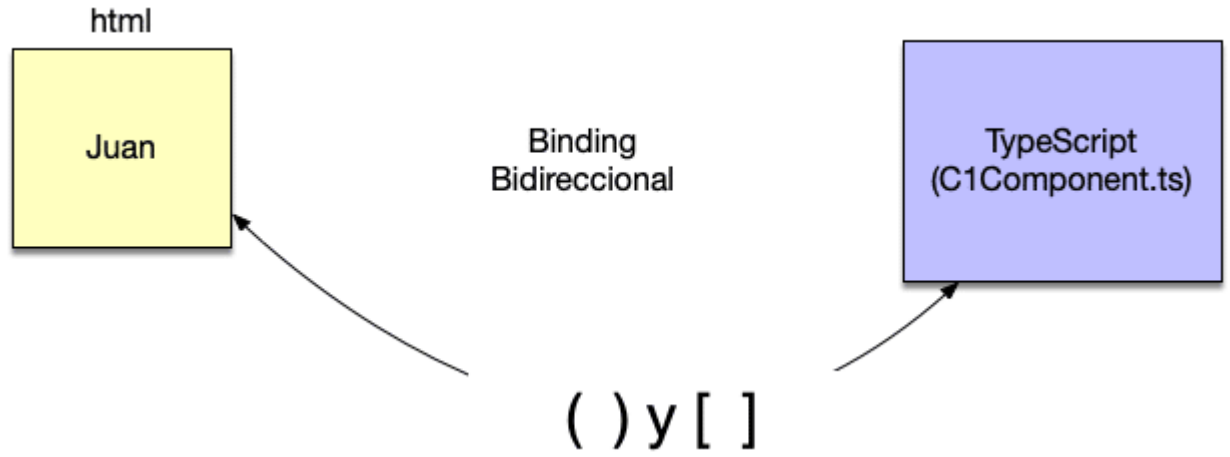
El valor de la interpolación se actualiza:



Acabamos de construir un binding unidireccional entre la plantilla y el componente haciendo uso de los corchetes y de una función cambiar() en el componente de Angular. Esta funcionalidad se podría haber construido de una forma un poco diferente y hacer que el binding fuera bidireccional. Por ejemplo con un código en la plantilla similar a:

```
<p [innerHtml]="nombre" (click)="cambiar()">
  {{nombre}}
</p>
```

En este caso el binding es bidireccional según carguemos nosotros la página aparecerá el nombre de Pedro pero si pulsamos sobre él se ejecutara el evento de click y pasara a valor Juan . Por lo tanto hemos construido un binding en ambas direcciones.



Angular ngModel

En muchas ocasiones nos encontramos que este tipo de bindings los que se denominan bidireccionales son los que más necesitamos en la programación del día a día. Para ello Angular nos provee una sintaxis más compacta que indica que este binding es BiDireccional combinando corchetes y paréntesis [()]. Para ello el primer paso que tenemos que realizar

es instalar a nivel de la aplicación el FormsModule que es el módulo que nos ayuda a gestionar formularios.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from "@angular/forms"
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { C1Component } from './c1/c1.component';

@NgModule({
  declarations: [
    AppComponent,
    C1Component
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

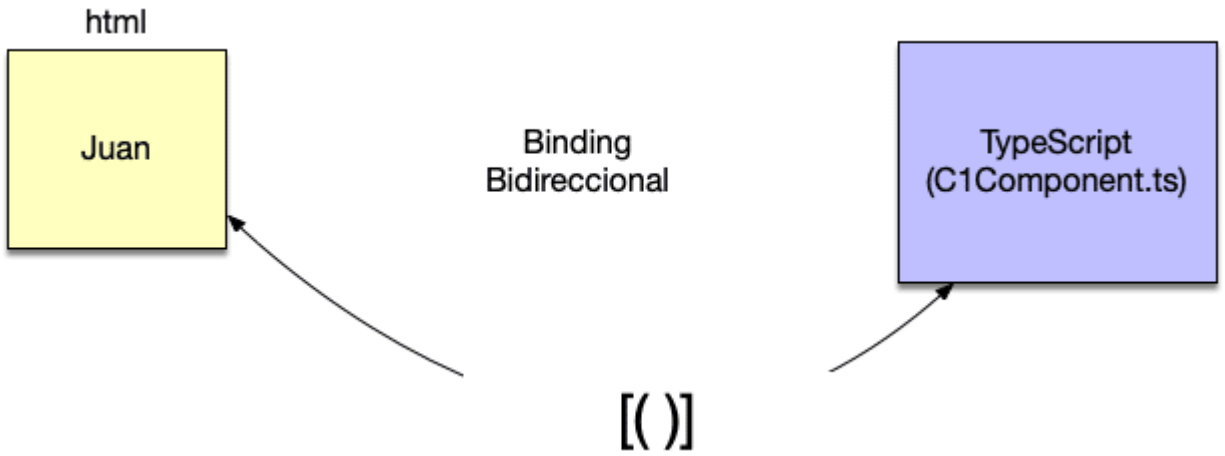
Una vez configurado el Forms módule podemos actualizar nuestra plantilla y usar la directiva de Angular ngModel para realizar un binding bidireccional como teníamos en el ejemplo anterior pero mucho más compacto con corchetes y paréntesis.

```
<input type="text" name="nombre" [(ngModel)]="nombre" />
  {{nombre}}
```

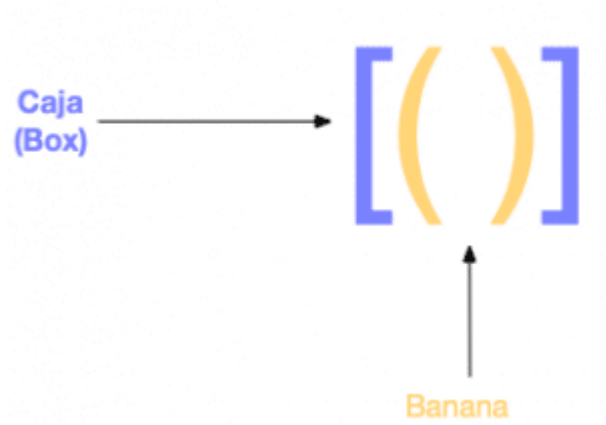
De esta manera tendremos una caja de texto que contiene el nombre que nosotros necesitamos . Si cambiamos el valor de la caja automáticamente nos cambiará el valor del texto que tenemos a nivel de la interpolación, el binding será bidireccional.



Acabamos de usar Angular la directiva ngModel para construir nuestro binding .



Muchas veces la gente cuando comienza con Angular se suele equivocar mucho a la hora de realizar los binding bidireccionales y que no sabe cómo poner los corchetes o los paréntesis. Para ello la gente de Angular decidió definir una regla de nemotecnia y llamo a este tipo de binding “banana in box” porque se trata de una banana dentro de una caja. Así es imposible confundirse.



Angular ngModel y objetos

Recordemos que los bindings no solo se pueden aplicar a tipos básicos sino que pueden ser aplicados a objetos complejos que provengan de clases y asignar las propiedad que nosotros deseemos. Por ejemplo si disponemos de la clase Persona:

```
export class Persona {

    nombre:string;
    edad:number;
}
```

Podemos usar Angular ngModel y realizar un two way data binding sobre sus propiedades:

```
import { Component, OnInit } from '@angular/core';
import { Persona } from '../persona';

@Component({
  selector: 'app-c1',
  templateUrl: './c1.component.html',
  styleUrls: ['./c1.component.css']
})
export class C1Component implements OnInit {
```

```
  persona:Persona;

  constructor() {

   this.persona= new Persona();
   this.persona.nombre="gema";
   this.persona.edad=20;
  }

  ngOnInit() {
  }

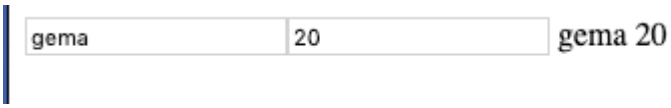
}
```

Acabamos de usar la clase Persona a nivel de componente nos falta ver su información a nivel de la propia plantilla:

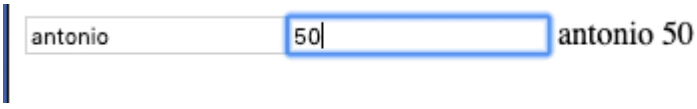
```
<input type="text" name="nombre" [(ngModel)]="persona.nombre" />

<input type="text" name="edad" [(ngModel)]="persona.edad" />
{{persona.nombre}}
{{persona.edad}}
```

Si vemos el resultado en el navegador nos cargará la información sobre el objeto:



Si cambiamos la información automaticamente los datos se actualizan en las interpolaciones.



Conclusiones

Aprendamos a usar binding bidireccionales o two way databindings con Angular apoyándonos en la sintaxis de banana in box [()] . Esto nos facilitará sobre manera el trabajo con componentes en el día a día. Sobre todo cuando trabajamos con estructuras de objetos complejas que demandan un mapeo de propiedades no trivial como acabamos de ver en este último caso de las personas.

Angular ngModel y Validaciones

Vamos a ver un ejemplo complementario en el cual se usan también validadores y es algo mas complejo:

```
<form #miFormulario="ngForm">
  <label for="nombre">Nombre:</label>
  <input
    type="text"
    id="nombre"
    name="nombre"
    [(ngModel)]="nombre"
    required
    class="form-control"
    #nombreInput="ngModel">
  <!-- Mensaje de error -->
  <div *ngIf="nombreInput.errors?.required && nombreInput.touched">
    El nombre es obligatorio.
  </div>

  <!-- Botón de envío -->
  <button [disabled]="miFormulario.invalid">Enviar</button>
</form>
```

Este es un ejemplo básico de cómo utilizar el validador required en Angular para gestionar campos obligatorios en un formulario.

1. El campo de entrada “nombre” está vinculado a la propiedad ngModel, y el validador required lo hace obligatorio.
2. El mensaje de error “El nombre es obligatorio” se mostrará cuando el usuario interactúe con el campo (es decir, cuando el campo haya sido tocado) y no haya ingresado ningún valor.
3. El botón de envío se mantendrá deshabilitado mientras el campo sea inválido, lo que significa que no permitirá el envío hasta que el usuario introduzca un valor válido en el campo.
   Este ejemplo es simple, utilizando solo la validación requerida para asegurarse de que el campo “nombre” no esté vacío antes de permitir el envío del formulario.

Otros artículos relacionados

- Angular ngFor
- Angular Select
- Angular ngIf