3 - DATA

> Formularios, tablas y modelos de datos en Angular

Las **aplicaciones Angular 7 son excelentes para el tratamiento de datos** en el navegador. Su razón de ser fue la recogida de información mediante formularios y la presentación de páginas dinámicas de forma sencilla.

Vamos a ver cómo la librería `@angular/forms` enlaza **las vistas, los controladores y los modelos**; y cómo se hace la presentación de datos en **listas y tablas**.

Partiendo de la aplicación tal cómo quedó en Páginas y rutas Angular SPA, al finalizar tendrás una aplicación que recoge y presenta datos.

> Código asociado a este artículo en *GitHub*: AcademiaBinaria/angular-board/

1. Binding

1.0 Base

Los formularios son el punto de entrada de información a nuestros sistemas. Llevan con nosotros desde el inicio de la propia informática y se han comido una buena parte del tiempo de programación. En *Angular* han prestado una especial atención a ellos facilitando su desarrollo, **desde pantallas simples hasta complejos procesos**.

Para empezar crearemos un proceso sencillo. algo que permita mantener una lista de contactos empresariales. Con lo aprendido en el tema de Páginas y rutas Angular SPA creamos un par de ficheros.

```
ng g m contacts --routing true
ng g c contacts/contacts
```

Y asignamos sus rutas delegadas en `app-routing` y en `contacts-routing`:

```
// app-routing
{
  path: 'contacts',
  loadChildren: './contacts/contacts.module#ContactsModule'
},
// contacts-routing
{
  path: '',
  component: ContactsComponent
}
```

Y finalizamos con un enlace en el `HeaderComponent`

```
<a routerLink="contacts" class="button">
  <span> Contacts</span>
</a>
```

La clave para entender cómo funciona *Angular* está en el concepto de **enlace entre elementos html de las vistas y propiedades de modelos** de datos, el llamado `binding`.

Para realizar el *binding* usaremos **directivas** en ambos sentidos.

1.1 Enlace del modelo hacia la vista

Vamos a crear un pequeño modelo de datos. Para empezar agregamos algunas propiedades. En `contacts.component.ts`:

```
public header = 'Contacts';
public description = 'Manage your contact list';
public numContacts = 0;
public counterClass = 'tag secondary';
public formHidden = false;
```

En `contacts.component.html` mostramos cabeceras con estilo

```
<h2>{{ header }}</h2>
<p>{{ description | uppercase }}</p>
<p>
  You have
  <mark [class]="counterClass">{{ numContacts }}</mark>
  contacts right now.
</p>
```

La interpolación entre { { } }

En el fichero `contacts.component.ts` tienes en su vista *html* encontrarás elementos ajenos al lenguaje. Son las directivas. La primera que encuentras es `{{ header }}`. Esas dobles llaves encierran expresiones que se evaluarán en tiempo de ejecución. La llamamos **directiva de interpolación** y es la manera más cómoda y usual de mostrar contenido dinámico en Angular.

> La expresión interna hace referencia a variables que se obtienen de las propiedades de la clase controladora del componente. En este caso `ContactsComponent` y `header`, con su valor *Contacts* en ejecución. Este enlace mantiene la vista permanentemente actualizada a través de un potente sistema de detección del cambio.

Las tuberías |

Si queremos que la presentación del dato sea distinta a su valor real, podemos usar **funciones de transformación** especiales. Se llaman tuberías o *pipes* y se indican mediante el carácter `|`.

El *framework* nos provee de casos básicos como `uppercase, lowercase, date, number...`. También dispones de un mecanismo para crear tus propios *pipes*.

Los atributos evaluados []

En *Html* disponemos de atributos para asignar valores a propiedades de los elementos. Esos atributos reciben los valores como constantes. Pero, si se encierran entre corchetes se convierten en un **evaluador de expresiones** y puede recibir una variable o cualquier otra expresión.

Como por ejemplo usando una *clase css* cuyo valor cambia en tiempo de ejecución. O para deshabilitar un elemento dinámicamente.

1.2 Enlace de la vista hacia el modelo

En `contacts.component.html` también actuamos sobre la vista, para manipular el modelo... y de vuelta a la vista. Por ejemplo con mecanismo simple de ocultación de un elemento.

```
<input value="Show Form" type="button" (click)="formHidden = false" />
<input value="Hide Form" type="button" (click)="formHidden = true" />
<form [ngClass]="{ 'hidden' : formHidden }">
  <fieldset><legend>Contact Form</legend></fieldset>
</form>
```

Las clases CSS como atributos especiales

Para el caso concreto de determinar las clases CSS aplicables a un elemento de manera dinámica, usaremos la directiva `ngClass`. La cual recibe un objeto cuyas propiedades son nombres de clases CSS y sus valores son expresiones booleanas. Si se cumplen se aplica la clase y si no, se quita la clase.

```
[ngClass]="{ 'hidden' : formHidden }"
```

En este caso se oculta el elemento dependiendo del valor de la expresión `formHidden`. Pero ¿Cómo se manipula esa variable?

Los eventos ()

Cualquier evento asociado a un elemento puede ejecutar una instrucción sin más que incluirlo entre paréntesis. Idealmente dicha instrucción debe llamar a un método o función de la clase controladora. Aunque si es trivial puedes dejarla en el Html.

```
(click)="formHidden = true"
```

2. Doble Binding

La comunicación del modelo hacia la vista es sólo el principio. En *Angular* también podrás **comunicar la vista hacia el modelo**, permitiéndole al usuario modificar los datos a través de formularios. Es lo que se conoce como *double binding*.

2.1 El doble enlace al modelo [(ngModel)]

La directiva `[(ngModel)]` se compone de un atributo *custom* `ngModel` y lo rodea de los símbolos `[()]`. Esta técnica es conocida como *banana in a box* porque su sintaxis requiere un `()` dentro de un `[]` y une las capacidades de las expresiones y los eventos facilitando la comunicación bidireccional.

```
[(ngModel)]="model.property"
```

Usa la comunicación en ambos sentidos

- **(banana)** : de la vista al modelo
- **[box]** : del modelo a la vista

> Atención: La directiva `ngModel` viene dentro del módulo `FormsModule` que hay que importar explícitamente.

Por ejemplo `[(ngModel)]="contact.name"` enlaza doblemente la propiedad del modelo `contact.name` con el elemento `<input>` de la vista. Cada tecleo del usuario se registra en la variable. Y el valor de la variable se muestra en el `<input>`.

Dada un modelo como este en `contacts.component.ts`:

```
public contact = { name: '' };
```

Podemos enlazarlos en la plantilla

```
<section>
  <label for="name">Name</label>
  <input name="name" type="text" [(ngModel)]="contact.name" placeholder="Contact
name" />
</section>
```

> Es muy útil mantener en desarrollo un espía visual de lo que está pasando con los datos. Algunas extensiones como Augury aportan muchas más prestaciones, pero al empezar el *pipe json* te ayudará mucho.

```
<pre>{{ contact | json }}</pre>
```

La directiva `ngModel` es mucho más potente de lo visto aquí. Entre otras cosas permite decidir el criterio de actualización (a cada cambio o al salir del control). También se verá más adelante el asunto de la validación, que requiere un trato especial. Cuando empiezas con Angular Forms, **un input y su ngModel** asociado serán tus mejores amigos.

2.2 Form

Hay más usos de las directivas en los formularios. Por ejemplo, dado el siguiente modelo:

```
public contact = { name: '', isVIP: false, gender: '' };
```

Le vendría muy bien un *check box*.

CheckBox

```
<section>
  <label for="isVIP">Is V.I.P.</label>
  <input name="isVIP" type="checkbox" [(ngModel)]="contact.isVIP" />
</section>
```

Y un para de *radio buttons*.

Radio Buttons

```
<section>
  <label for="gender">Gender</label>
  <input name="gender" value="male" type="radio" [(ngModel)]="contact.gender" />
  <i>Male</i>
  <input name="gender" value="female" type="radio" [(ngModel)]="contact.gender" />
  <i>Female</i>
</section>
```

3 Estructuras

Los anteriores modificadores actúan a nivel de contenido del HTML. Veremos ahora una para de directivas que afectan directamente a **la estructura del árbol DOM**. Son las llamadas directivas estructurales que comienzan por el signo `*`

3.1 Repetitivas *ngFor

Una situación que nos encontramos una y otra vez es la de las repeticiones. Listas de datos, tablas o grupos de opciones son ejemplos claros. Hay una directiva en *Angular* para esa situación, la `*ngFor="let iterador`

`of array"`. **La directiva `*ngFor` forma parte del grupo de directivas estructurales**, porque modifica la estructura del DOM, en este caso insertando múltiples nodos hijos a un elemento dado.

> Puedes ver un ejemplo del uso la directiva `*ngFor` en el componente `ContactsComponent`. Se emplea para recorrer un array de tipos de estado laboral. Es el caso de uso *más repetido de las repeticiones*; mostrar listas de datos.

Dado el siguiente modelo:

```
public workStatuses = [
  { id: 0, description: 'unknow' },
  { id: 1, description: 'student' },
  { id: 2, description: 'unemployed' },
  { id: 3, description: 'employed' }
];
public contact = { name: '', isVIP: false, gender: '', workStatus: 0 };
```

Montamos las opciones de un *select* html recorriendo el array y usando el iterador `wkSt` para acceder a sus datos.

```
<section>
  <label for="workStatus">Work Status</label>
  <select name="workStatus" [(ngModel)]="contact.workStatus">
    <option *ngFor="let wkSt of workStatuses" [value]="wkSt.id">
      <span>{{ wkSt.description }}</span>
    </option>
  </select>
</section>
```

3.2 Condicionales *ngIf

La directiva estructural más utilizada es la `*ngIf`, la cual consigue que un elemento se incluya o se elimine en el *DOM* en función de los datos del modelo.

> En el ejemplo puedes ver que la uso para mostrar el campo empresa cuando el contacto está trabajando. En otro aparecerá el campo de estudios.

Dado el siguiente modelo:

```
public contact = {
  name: '',
  isVIP: false,
  gender: '',
  workStatus: '0',
  company: '',
  education: ''
};
```

```
<section *ngIf="contact.workStatus=='3'; else education">
  <label for="company">Company Name</label>
  <input name="company" type="text" [(ngModel)]="contact.company" />
</section>
<ng-template #education>
  <label for="education">Education</label>
  <input name="education" type="text" [(ngModel)]="contact.education" />
</ng-template>
```

> if **condition** else **template**

Identificadores con hashtag

En el código anterior apreciarás que aparece un elemento `<ng-template>` no estándar con el atributo llamado `#education` precedido por un `#`. La directiva `#` genera un identificador único para el elemento al que se le aplica y permite referirse a él en otros lugares del código.

Ese truco permite que `*ngIf` muestre otro elemento cuando la condición principal falle. El otro elemento tiene que ser el componente especial `<ng-template>` que se usa para envolver una rama opcional del DOM. Para localizarlo se usa el identificador `#`.

4 Modelo y controlador

Los componentes los hemos definido como **bloques de construcción de páginas. Mediante una vista y un controlador** resuelven un problema de interacción o presentación de modelos. En los puntos anteriores te presenté la vista. Toca ahora estudiar el modelo y el controlador.

4.1 El modelo y su interInterfaces y modelos

Sin ir muy lejos en las capacidades que tendría un modelo de datos clásico, vamos al menos a beneficiarnos del ***TypeScript* para definir la estructura de datos**. Esto facilitará la programación mediante el autocompletado del editor y reducirá los errores de tecleo mediante la comprobación estática de tipos.

Para ello necesito una interfaz sencilla. Esto es puro *TypeScript*, no es ningún artificio registrable en Angular. Esos sí, en algún sitio tienen que estar. Yo suelo usar una ruta como `contacts/models`, pero es algo completamente arbitrario.

```
export interface Option {
  id: number;
  description: string;
}

export interface Contact {
  name: string;
  isVIP: boolean;
  gender: string;
```

```
  workStatus: number | string;
  company: string;
  education: string;
}
```

> Te recomiendo que **no uses clases para definir modelos** a menos que necesites agregarle funcionalidad imprescindible. Las interfaces, ayudan al control de tipos en tiempo de desarrollo, igual que las clases, pero sin generar nada de código en tiempo de ejecución, al contrario que las clases. Ojo al uso de tipos compuestos como `number | string`

Se usan para tipificar las propiedades que conforman nuestro modelo para la vista.

```
public workStatuses: Option[] = [
    { id: 0, description: 'unknow' },
    { id: 1, description: 'student' },
    { id: 2, description: 'unemployed' },
    { id: 3, description: 'employed' }
  ];
public contact: Contact = {
    name: '',
    isVIP: false,
    gender: '',
    workStatus: 0,
    company: '',
    education: ''
  };
public contacts: Contact[] = [];
```

4.2 ViewModel en el controlador

La parte de **lógica del componente** va en la clase que se usa para su definición. Como ya has visto podemos usar su constructor para reclamar dependencias y usar los interfaces para responder a eventos de su ciclo de vida. Repasemos el `ContactsComponent` viéndolo como la clase que es: no solo propiedades, también **métodos**

```
public saveContact() {
  this.contacts.push({ ...this.contact });
  this.numContacts = this.contacts.length;
}
```

Ahora se trata de invocar el método desde la vista. Es muy buena práctica llevar la lógica al controlador y no escribirla en la vista.

```
<input value="Save" type="submit" (click)="saveContact()" />
```

Podemos decir que las propiedades públicas de la clase actuarán como *binding* de datos con la vista. Mientras que los métodos públicos serán invocados desde los eventos de la misma vista.

OnInit

Los componentes son clases con un **ciclo de vida** al que puedes enganchar tu código en algunos pasos. Por ejemplo al iniciarse el componente.

El CLI hace que las clase del componente implemente la interfaz `OnInit` y eso permite al framework invocar al método `ngOnInit` en cuanto el componente esté listo para su uso. Que no suele ser justo durante la construcción, si no un poco después. Te recomiendo que lleves toda la lógica de inicialización a dicho método.

```
public workStatuses: Option[];
public contact: Contact;
public contacts: Contact[];
constructor() {}
public ngOnInit() {
  this.workStatuses = [
    { id: 0, description: 'unknow' },
    { id: 1, description: 'student' },
    { id: 2, description: 'unemployed' },
    { id: 3, description: 'employed' }
  ];
  this.contact = {
    name: '',
    isVIP: false,
    gender: '',
    workStatus: 0,
    company: '',
    education: ''
  };
  this.contacts = [];
}
```

Un listado de repaso

Para mostrar lo que ahora estamos guardando en una lista, nada más sencillo que usar de nuevo a `*nFor` y a `*ngIf` para tratar listas vacías.

```
<ul *ngIf="contacts.length>0; else empty">
  <li *ngFor="let contact of contacts">
    <span>{{ contact.name }}</span>
    <input value="Delete" type="button" (click)="deleteContact(contact)" />
  </li>
</ul>
<ng-template #empty> <i>No data yet</i> </ng-template>
```

Y ya puestos incluso puedes animarte a borrar contactos. Es fácil, los métodos pueden recibir argumentos. Y la vista los puede enviar.

```
public deleteContact(contact: Contact) {
  this.contacts = this.contacts.filter(c => c.name !== contact.name);
  this.numContacts = this.contacts.length;
}
```

Mira el código completo de **la clase** `ContactsComponent`en el fichero `contacts.component.ts` para tener una visión completa del componente. Como ves, **las propiedades** `header, numContacas, formHidden, contacts ...` se corresponden con las utilizadas en las directivas de enlace en la vista. **Los métodos** `saveContact(), deleteContact()` son invocados desde eventos de elementos del *html*.

Juntos, **la vista y su clase controladora**, resuelven un problema de interacción con el usuario **creando un componente**. Todas las páginas que diseñes serán variaciones y composiciones de estos componentes.

> Y esto es sólo el comienzo. La idea de componente será fundamental en la web del mañana para la creación de páginas mediante `web components`. Pero eso ya se verá más adelante...