8.- Formularios

En desarrollo web las aplicaciones usan formularios para loguear usuarios, actualizar un perfil, introducir información sensible y actualizar muchos otros datos.

Angular provee dos formas diferentes de crear formularios: *reactive* y *template-driven*. Ambos recogen los eventos de las etiquetas *input* desde la vista, validan los datos introducidos, crean un modelo de formulario y proveen una manera de recoger posibles cambios.

En general:

- *Reactive forms*: son más robustos, son más escalables, reusables y testeables. Si la aplicación se base mayoritariamente en formularios, una zona de administración por ejemplo, se debe usar esta forma de generar formularios.
- *Template-driven forms*: son más útiles para añadir simples formularios.

Diferencias claves

	Reactive	***Template-driven***
Preparación	Se crea en el componente	Creado por directivas
Modelo	Estructural	No estructural
Previsibilidad	Síncrona	Asíncrona
Validación	Funciones	Directivas
Mutabilidad	Inmutable	Mutable
Escalabilidad	Acceso a API a bajo nivel	Abstracción de la API

8.1.- *Reactive Forms*

Los formularios *reactive* utilizan un enfoque explícito e inmutable para administrar el estado de un formulario. Cada cambio en el estado del formulario devuelve un nuevo estado, que mantiene la integridad del modelo entre cambios. También se debe de tener en cuenta a la hora de trabajar con ellos que se crean bajo la lógica de *Observable* de Angular, proporcionan por lo tanto un acceso síncrono al modelo de datos, con operaciones del objeto *Observable* y seguimiento de cambios.

Registro del formulario

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

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule { }
```

Debemos importar el módulo de *ReactiveFormsModule* en el módulo principal o submódulo donde vayamos a trabajar con formularios.

Primer formulario

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

@Component({
  selector: 'app-name-editor',
  templateUrl: './name-editor.component.html',
  styleUrls: ['./name-editor.component.css']
})
export class NameEditorComponent {
  name = new FormControl('');
}
```

FormControl es la clase más básica para realizar un formulario *reactive*. Esto creará una instancia para guardar el valor de *name* como una propiedad de clase. Se debe de inicializar la variable a través del constructor de *FormControl(‘’)*.

En el html debemos colocar la propiedad bindeada a la variable:

```
<label>  Name:
  <input type="text" [formControl]="name">
</label>
```

Para mostrar el valor de *name* fuera de la etiqueta de input:

- Plantilla:

```
<p>
  Value: {{ name.value }}
</p>
```

- Componente:

```
this.name.valueChanges.subcribe( (value) => {
  console.log(value);
});

//O también

this.name.value
```

8.1.1.- FormGroup

Para crear un objeto con varios *FormControl* utilizaremos el objeto *FormGroup*.

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

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
  });
}
```

En la plantilla:

```
<form [formGroup]="profileForm">

  <label>
    First Name:
    <input type="text" formControlName="firstName">
  </label>

  <label>
    Last Name:
    <input type="text" formControlName="lastName">
  </label>

</form>
```

Crearemos un binding de la propiedad de *formGroup* con el objeto creado en el componente. Esta instancia es suficiente para que Angular reconozca en cada propiedad de *formControlName* el bindeo con las variables del objeto *profileForm*.

Para recuperar los valores del formulario se hace igual que para el caso anterior:

```
console.warn(this.profileForm.value);
```

8.1.2.- Grupos anidados

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

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl('')
    })
  });
}
```

Con este ejemplo podemos ver lo sencillo que es crear subgrupos dentro del *FormGroup* principal. Y para implementarlo en el html debemos colocar la etiqueta *formGroupName* para referenciar el objeto:

```
<div formGroupName="address">
  <h3>Address</h3>
  <label> Street:
    <input type="text" formControlName="street">
  </label>
  <label> City:
    <input type="text" formControlName="city">
  </label>
  <label> State:
    <input type="text" formControlName="state">
  </label>
  <label> Zip Code:
    <input type="text" formControlName="zip">
  </label>
</div>
```

8.1.3.- FormBuilder

FormBuilder es una forma diferente de realizar también un formulario *reactive*. Tan solo se debe de cambiar la forma de crear una instancia del objeto a usar en el html.

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

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: [''],
    lastName: [''],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });

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

Para más información de este objeto visitar la web de Angular.

8.2.- *Template-driven forms*

Registro del formulario

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

@NgModule({
  imports: [
    // other imports ...
    FormsModule
  ],
})
export class AppModule { }
```

Debemos importar el módulo de *FormsModule* en el módulo principal o submódulo donde vayamos a trabajar con formularios.

Plantilla HTML

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm">
    <div class="form-group">
      <label for="name">Name</label>
      <input type="text" class="form-control" id="name" required [(ngModel)]="model.name" name="name"
            #name="ngModel">
      <div [hidden]="name.valid || name.pristine" class="alert alert-danger"> Name is required </div>
    </div>

    <div class="form-group">
      <label for="alterEgo">Alter Ego</label>
     <input type="text" class="form-control" id="alterEgo" [(ngModel)]="model.alterEgo"
           name="alterEgo">
    </div>

    <div class="form-group">
      <label for="power">Hero Power</label>
      <select class="form-control" id="power" required  [(ngModel)]="model.power" name="power"
            #power="ngModel">
        <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
      </select>
      <div [hidden]="power.valid || power.pristine" class="alert alert-danger"> Power is required </div>
    </div>

    <button type="submit" class="btn btn-success" [disabled]="!heroForm.form.valid">Submit</button>
    <button type="button" class="btn btn-default" (click)="newHero(); heroForm.reset()">
          New Hero</button>
</form>
```

Como se puede ver un formulario *Template-driven* es una combinación de la arquitectura básica de Angular con alguna excepción, como:

- #heroForm=”ngForm” => ngForm añade las funciones de formularios a nuestra variable heroForm dentro de nuestra etiqueta <form>
- (ngSubmit) => Evento para ejecutar la función de enviar los datos tras pulsar el boton con *type=”submit”*
- #var=”ngModel” => Al igual que la variable del formulario, añadiremos las funcionalidades de formulario a cada variable.

Componente

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

import { Hero }    from '../hero';

@Component({
  selector: 'app-hero-form',
  templateUrl: './hero-form.component.html',
  styleUrls: ['./hero-form.component.css']
})
```

```
export class HeroFormComponent {

  powers = ['Really Smart', 'Super Flexible',
          'Super Hot', 'Weather Changer'];

  model = new Hero();

  submitted = false;

  onSubmit() { this.submitted = true; }

  newHero() {
    this.model = new Hero(42, '', '');
  }
}
```

8.3.- Validación

8.3.1.- Template-driven validation

Para añadir validación a este tipo de formularios, se añadirá la misma forma de validación que cualquier formulario de HTML. Angular usa directivas para unir esos atributos con sus funciones de validación:

- Los atributos *required*, *minlenght* o *forbiddenName* en la etiqueta <input>
- *#var=”ngModel”* exporta la clase *NgModel* en una variable. Se usa para validar los estados de la etiqueta donde se coloque.

```
<input id="name" name="name" class="form-control" required minlength="4" appForbiddenName="bob"
    [(ngModel)]="hero.name" #name="ngModel" >

<div *ngIf="name.invalid && (name.dirty || name.touched)" class="alert alert-danger">

  <div *ngIf="name.errors.required"> Name is required. </div>
  <div *ngIf="name.errors.minlength"> Name must be at least 4 characters long.  </div>
  <div *ngIf="name.errors.forbiddenName"> Name cannot be Bob. </div>

</div>
```

8.3.2.- Reactive form validation

Para formulario *reactive* la validación se puede implementar también en el HTML pero *FormControl* tiene una opción para pasarle los parámetros de la validación:

```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    'name': new FormControl(this.hero.name, [ Validators.required, Validators.minLength(4),
      forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom validator.
    ]),
    'alterEgo': new FormControl(this.hero.alterEgo),
    'power': new FormControl(this.hero.power, Validators.required)
  });

}

get name() { return this.heroForm.get('name'); }

get power() { return this.heroForm.get('power'); }
```

El parámetro de validación puede ser un solo valor o un array de valores de validación. La lógica del HTML será parecida a la validación de *Template form*:

```
<input id="name" class="form-control" formControlName="name" >

<div *ngIf="heroForm.get('name').invalid && (heroForm.get('name').dirty ||
heroForm.get('name').touched)" class="alert alert-danger">

  <div *ngIf="heroForm.get('name').errors.required"> Name is required. </div>
  <div *ngIf="heroForm.get('name').errors.minlength"> Name must be at least 4 characters long. </div>
  <div *ngIf="heroForm.get('name').errors.forbiddenName"> Name cannot be Bob. </div>
</div>
```

8.3.3.- Clases CSS

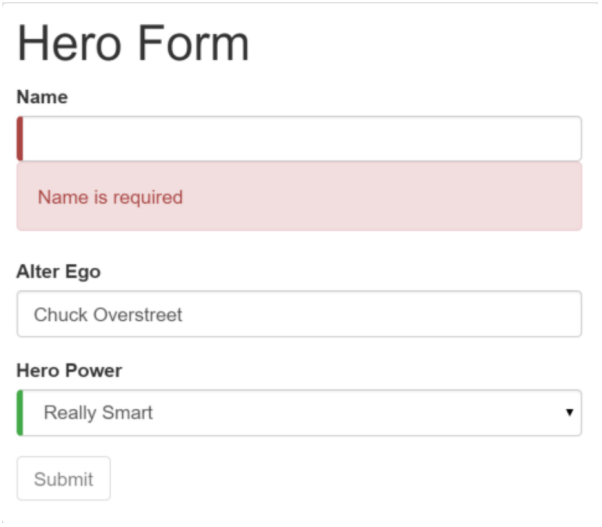
Angular provee una serie de clases que se añaden automáticamente a los elementos de control, con ellas se puede editar el estilo del elemento según su estado de validación.

.ng-valid	.ng-invalid	.ng-pending	.ng-dirty
.ng-pristine	.ng-untouched	.ng-touched	

Ejemplo:

```
.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}
```

```
.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /* red */
}
```



Estas clases son añadidas automáticamente en el HTML con los atributos de HTML por lo que si se quiere modificar el CSS a través de estas clases se deben colocar los atributos, como por ejemplo *'required'*, aunque no hagan falta como en el caso de *Reactive Forms* en realidad, pero si queremos modificar el estilo como en el ejemplo anterior se deben de colocar. Esto no afectará a la funcionalidad del formulario en el caso de ser *Reactive Form*.