Semana 7

Curso de Angular

Semana 6
Formularios

Desarrollo de aplicaciones web con Angular
Cefire 2017/2018
Autor: Arturo Bernal Mayordomo

Index

Formularios de plantilla

Los formularios suelen ser una de las partes más importantes de una aplicación web. Nos permiten crear y actualizar datos, identificarnos, etc. Angular tiene una serie de características integradas para gestionar formularios, validarlos y facilitar el mostrar errores y *feedback* al usuario.

Para usar estas características, aunque simplemente sean directivas como ngModel para vincular datos de entrada, necesitamos importar **FormsModule** (en los módulos de la aplicación que vayan a requerirlo).

En este ejemplo, vamos a crear un formulario para modificar productos (ruta **products/edit/:id**). Lo primero será obtener los datos del producto, por ejemplo, usando el mismo Resolve que en la página de detalle → **ProductDetailResolve**.

```
  {
    path: 'edit/:id', component: ProductEditComponent,
    canActivate: [ProductDetailGuard],
    canDeactivate: [CanDeactivateGuard],
    resolve: {
      product: ProductDetailResolve
    }
  }
```

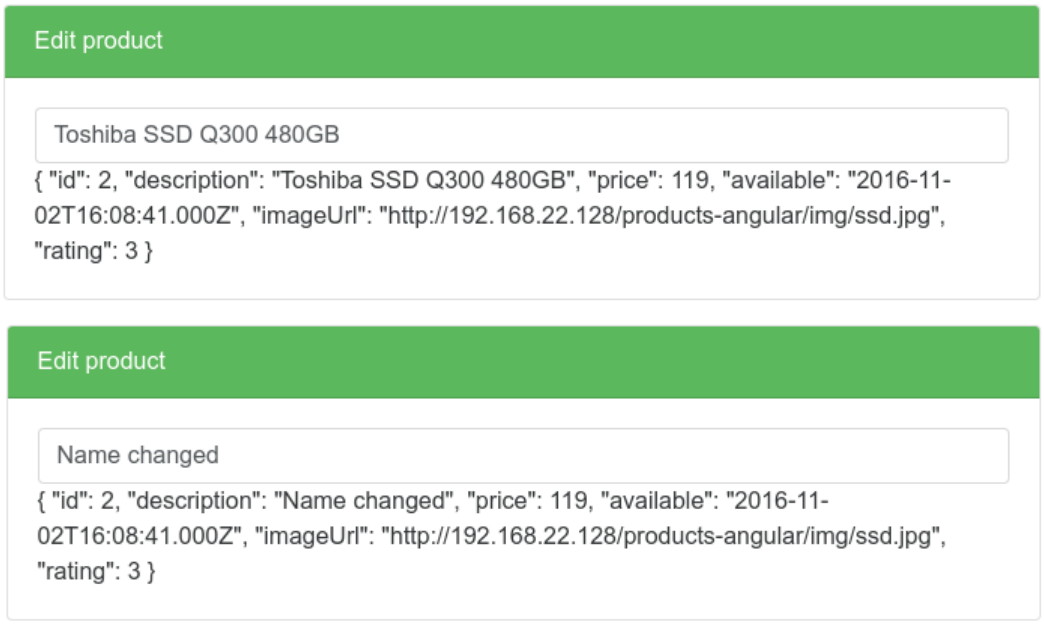
A continuación, desde el componente, asignamos el producto y empezamos:

```
export class ProductEditComponent implements OnInit, CanComponentDeactivate {
  product: IProduct;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.product = this.route.snapshot.data['product'];
  }

  canDeactivate() {
    return confirm("Are you sure you want to leave? Changes can be lost.");
  }
}
```

ngModel

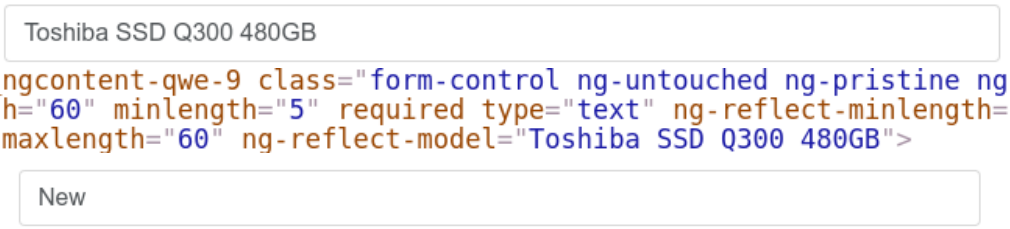
Hemos visto como funciona la directiva **ngModel** para vincular datos de un campo de entrada en 2 direcciones. Vamos a crear un campo (<input>) para modificar la descripción del producto en el componente.

```
<div class="card">
  <div class="card-header bg-success text-white">
    Editar producto
  </div>
  <div class="card-body">
    <div>
      <input type="text" class="form-control" [(ngModel)]="product.description">
    </div>
    <div>{{product | json}}</div>
  </div>
</div>
```



Se pueden utilizar atributos HTML5 de validación de campos como **required**, **min**, **max**, **pattern**, **minlength**, **maxlength**, etc. Estos atributos permiten la validación automática por parte de Angular y asignan una serie de **clases** al elemento en cuestión, dependiendo de si es válido o no, si ha sido modificado, etc.

```
<input type="text" class="form-control" [(ngModel)]="product.description"
       minlength="5" maxlength="60" required>
```



Estado	Clase (cierto)	Clase (falso)
Elemento visitado	ng-touched	ng-untouched
El valor ha cambiado	ng-dirty	ng-pristine
El valor es válido	ng-valid	ng-invalid

Esto nos permite entre otras cosas, jugar con el CSS para asignar estilos a los elementos del formulario en función de su estado. Otra forma de interactuar con la validación es creando una **referencia** a la directiva ngModel (es un objeto). Esta referencia es un atributo cuyo nombre empieza por almohadilla '#'. A partir de esta referencia, podemos acceder a sus propiedades y actuar en consecuencia.

```
<div>
    <input type="text" class="form-control" [(ngModel)]="product.description"
```

```
    minlength="5" maxlength="60" required #descModel="ngModel">
</div>
<div>Dirty: {{descModel.dirty}}</div>
<div>Valid: {{descModel.valid}}</div>
<div>Value: {{descModel.value}}</div>
<div>Errors: {{descModel.errors | json}}</div>
```

Podemos usar las clases autoasignadas o la referencia que hemos creado para añadir estilo al formulario usando CSS, ngClass, ngIf, etc. (Ejemplo de Bootstrap).

```
<div class="form-group">
  <input type="text" class="form-control" [(ngModel)]="product.description"
    minlength="5" maxlength="60" required #descModel="ngModel"
   [ngClass]="{'is-valid': descModel.touched && descModel.valid,
              'is-invalid': descModel.touched && !descModel.valid}">
</div>
```

Si ponemos un campo de formulario dentro de un elemento <form>, debemos asignar el atributo **name** o Angular nos emitirá un error.

```
<form>
    ...
    <input type="text" name="description" ...>
    ...
</form>
```

Por último. Si queremos procesar la entrada de datos del usuario detectando cuando cambia, debemos dividir la directiva bidireccional **[(ngModel)]**, en 2 directivas unidireccionales separadas: **[ngModel]** y **(ngModelChange)**. Cuando se lanza el evento de cambio, $event representa el valor actual del campo. Ejemplo de transformación del texto de entrada a mayúsculas:

```
<input type="text" class="form-control"
       [ngModel]="product.description"
       (ngModelChange)="product.description = $event.toUpperCase()" ...>
```

ngForm

La directiva ngForm está implícita en cualquier elemento <form>. Podemos crear una referencia igual que hacíamos con ngModel, y así poder comprobar las propiedades generales del formulario, como si este es válido o ha sido modificado. Un formulario sólo es válido si todos su campos lo son. La propiedad value, es un objeto JSON que contiene los valores de todos sus campos.

```
<form #productForm="ngForm" novalidate>
    <div class="form-group" ...>
        <input type="text" name="description" ...>
    </div>
</form>
<div>Touched: {{productForm.touched}}</div>
<div>Valid: {{productForm.valid}}</div>
<div>Value: {{productForm.value | json}}</div>
```

El atributo novalidate se recomienda para desactivar la validación integrada del navegador, dejando que esta sea controlada totalmente por Angular (misma experiencia en cualquier navegador).

A través de esta directiva podemos acceder a los campos individuales a partir de su atributo name (por ello es obligado ponerlo). Esto nos ahorra tener que crear referencias para cada uno de ellos, aunque la sintaxis no es precisamente corta.

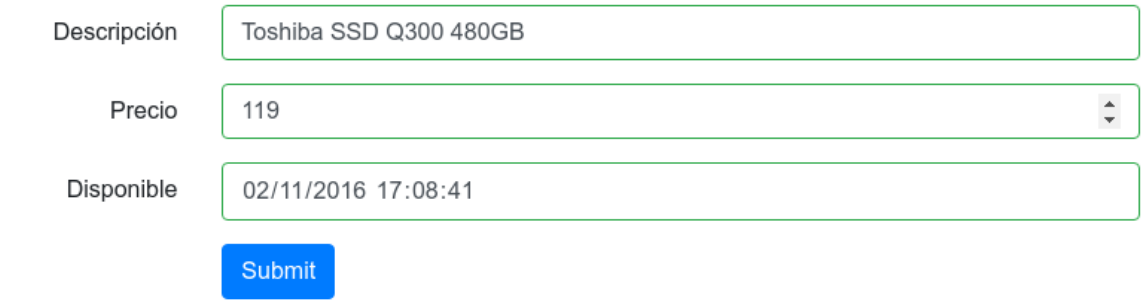
```
<div>Desc: {{productForm.control.get('description').value | json}}</div>
```

Tipos de campo

Casi todos los <input> (text, number, date, email, etc.) funcionan de la misma manera. Angular los valida según las reglas impuestas y usamos esta validación para cambiar el estilo de los campos.

```
<form #productForm="ngForm" novalidate>
 <div class="form-group row">
   <label class="col-sm-2 col-form-label text-sm-right">Descripción</label>
   <div class="col-sm-10">
     <input type="text" class="form-control" name="description" #descModel="ngModel"
        [(ngModel)]="product.description" minlength="5" maxlength="60" required
        [ngClass]="validClasses(descModel, 'is-valid', 'is-invalid')">
   </div>
 </div>
 <div class="form-group row">
   <label class="col-sm-2 col-form-label text-sm-right">Precio</label>
   <div class="col-sm-10">
     <input type="number" class="form-control" name="price" [(ngModel)]="product.price"
        min="0" step="0.01" required #priceModel="ngModel"
        [ngClass]="validClasses(priceModel, 'is-valid', 'is-invalid')">
   </div>
 </div>
 <div class="form-group row">
   <label class="col-sm-2 col-form-label text-sm-right">Disponible</label>
   <div class="col-sm-10">
     <input type="datetime-local" class="form-control" name="available"
        [(ngModel)]="product.available" #availModel="ngModel" required
        [ngClass]="validClasses(availModel, 'is-valid', 'is-invalid')">
   </div>
 </div>
 <div class="form-group row">
   <div class="offset-sm-2 col-sm-10">
     <button type="submit" class="btn btn-primary" [disabled]="productForm.invalid">
        Submit
     </button>
   </div>
 </div>
```

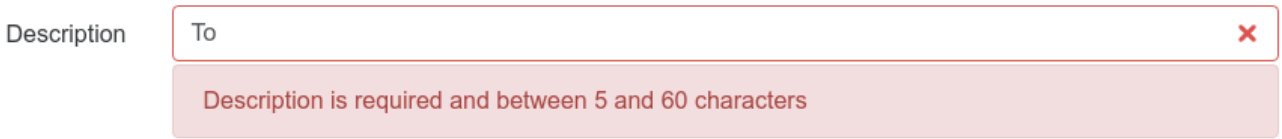
```
</form>
```



El campo datetime-local necesita una fecha en formato ligeramente diferente de como lo envía el servidor. En este caso, separando la fecha y la hora por una 'T' en lugar de un espacio. Para que lo interprete correctamente debemos cambiarlo:

```
ngOnInit() {
  this.product = this.route.snapshot.data['product'];
  this.product.available = this.product.available.replace(' ', 'T');
}
```

Además, hemos hecho que el botón de enviar se **deshabilite** cuando el formulario no sea válido.



Como puedes ver, hemos creado un método llamado validClasses para reducir un poco el código a la hora de asignar las clases CSS de Bootstrap. Los parámetros son: 1 → referencia al ngModel, 2 → clase cuando el elemento es válido, 3 → clase cuando la validación falla):

```
validClasses(ngModel: NgModel, validClass: string, errorClass: string) {
  return {
    [validClass]: ngModel.touched && ngModel.valid,
    [errorClass]: ngModel.touched && ngModel.invalid
  };
}
```

Si quisieramos usar la directiva ngModel con un **checkbox**, el valor debe ser booleano (true → seleccionado, false → no seleccionado):

```
<div class="form-check">
  <label class="form-check-label">
    <input class="form-check-input" type="checkbox" name="check"
           [(ngModel)]="booleanProperty">
    Ejemplo checkbox
  </label>
</div>
```

Para crear un grupo de botones radio, debemos agruparlos usando la misma propiedad en ngModel. Simplemente cambiamos el valor:

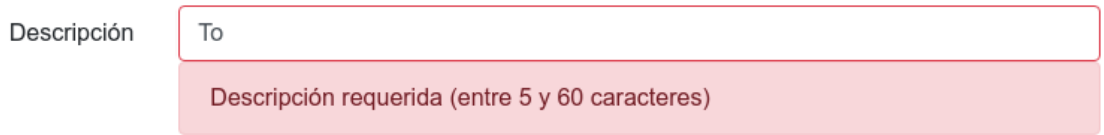
```
<div class="form-check">
  <label class="form-check-label">
    <input class="form-check-input" type="radio" name="radio" value="1"
           [(ngModel)]="someProperty">
    Opción 1
  </label>
</div>
<div class="form-check">
 <label class="form-check-label">
    <input class="form-check-input" type="radio" name="radio" value="2"
           [(ngModel)]="someProperty">
    Opción 2
  </label>
</div>
```

Por último, si quisieramos vincular un campo <select> a una propiedad, se debe poner la directiba ngMode sólo dentro de dicho elemento <select> (nunca se le asignará a un elemento <option>).

Validación

Como hemos observado. Podemos usar las propiedades de ngModel o las clases de validación automáticamente asignadas para asignar estilos o mostrar/ocultar ciertos errores.

```
<form #productForm="ngForm" novalidate>
  <div class="form-group row">
    <label for="inputEmail3" class="col-sm-2 col-form-label text-sm-right">
       Descripción
    </label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="description"
        [(ngModel)]="product.description" minlength="5" maxlength="60"
        required #descModel="ngModel" [ngClass]="validClasses(descModel,
'is-valid', 'is-invalid')">
    </div>
    <div class="offset-sm-2 col-sm-10">
      <div *ngIf="descModel.touched && descModel.invalid"
           class="alert alert-danger">
        Descripción requerida (entre 5 y 60 caracteres)
      </div>
    </div>
  </div>
  ...
</form>
```



Crear nuevos validadores

Puedes crear un nuevo validador que no exista en Angular, creando una directiva que implemente la interfaz **Validator**. Esta tendrá un método **validate** que recibirá el objeto del campo para validar. Vamos a crear un validador para restringir que la fecha actual deba ser posterior a una predefinida.

Para usar este tipo de directivas en varios módulos, debemos incluirla en algún módulo compartido que la exporte. Este tipo de directivas vamos a meterlas en una carpeta llamada **validators** para distinguirlas de las otras.

```
ng g directive min-date
```

```
import { Directive, Input } from '@angular/core';
import { Validator, NG_VALIDATORS, AbstractControl } from '@angular/forms';

@Directive({
  selector: '[minDate]',
  providers: [{provide: NG_VALIDATORS, useExisting: MinDateDirective,
               multi: true}]
})
export class MinDateDirective implements Validator {
  @Input() minDate;

  constructor() { }

  validate(c: AbstractControl): { [key: string]: any } {
    if (this.minDate && c.value) { // Si recibimos algún valor
      const dateControl = new Date(c.value); // Valor actual
      const dateMin = new Date(this.minDate); // Fecha mínima
      if (dateMin > dateControl) {
        return {'minDate': true}; // Error devuelto
      }
    }

    return null; // Sin error
  }
}
```

Así es como la usaríamos:

```
<div class="form-group col">
    <label for="filterDesc">Disponible:</label>
    <input type="date" [(ngModel)]="newProd.available"
           class="form-control" #availModel="ngModel"
           name="available" placeholder="Available date"
           required minDate="2017-09-01">
    {{availModel.status}} - Errores: {{availModel.errors | json}}
 </div>
```

Available:

dd/mm/aaaa

INVALID - Errors: { "required": true }

Available:

02/02/2014

INVALID - Errors: { " minDate ": true }

Available:

02/12/2018

VALID - Errors: null

Enviando el formulario

Cuando enviamos el formulario, podemos terminar de validar ciertas cosas, procesar los datos que vamos a mandar, y llamar al correspondiente servicio.

```
<form #productForm="ngForm" (ngSubmit)="update()" novalidate>
```

```
export class ProductEditComponent implements OnInit, CanComponentDeactivate {
  ...
  update() {
    // Otras validaciones, etc... (con return; cancelamos el envío)
    this.productService.updateProduct(this.product)
      .subscribe(
        ok => this.router.navigate([`/products/${this.product.id}`]),
        error => console.error(error)
      );
  }
}

export class ProductService {
```

```
  ...
    updateProduct(product: IProduct): Observable<boolean> {
    return this.http.put<OkResponse>(this.productURL + '/' + product.id,
product).pipe(
      catchError((resp: HttpErrorResponse) => throwError(`Error modificando
producto!. Código de servidor: ${resp.status}. Mensaje: ${resp.message}`)),
      map(resp => {
        if (!resp.ok) { throw resp.error; }
        return true;
      })
    );
  }
}
```

Validar antes de enviar

Desde el componente

Para acceder al objeto del formulario (NgForm) y comprobar su validación, podemos obtener una referencia de la plantilla usando el decorador @ViewChild y el nombre de la referencia.

```
<form class="form m-3" (ngSubmit)="addProduct()" #addForm="ngForm">

export class ProductAddComponent implements OnInit {
    @ViewChild('addForm') addForm: NgForm;
    ...

    addProduct() {
        if(this.addForm.valid) {
            this.productService.addProduct(this.newProd).subscribe(
                ok => {
                    this.router.navigate(['/products']);
                },
                error => console.error(error)
            );
        }
    }
    ...
}
```

Desde la plantilla

Usando la referencia en la plantilla, podemos deshabilitar en botón de envío hasta que el formulario sea válido.

```
<button type="submit" class="btn btn-primary mr-3"
        [disabled]="addForm.invalid">Add Product</button>
```

Subir un archivo en el objeto JSON

No podemos usar directivas como ngModel con campos de archivo ahora mismo en Angular (existen plugins para gestionar este tipo de campos). Sin embargo, a través de eventos como **change**, podemos procesar los campos y por ejemplo, transformar su contenido a base64 (string) para enviarlo junto al resto de valores.

Para ello, vamos a pasar al método una referencia al elemento <input>. En este caso queremos una referencia directa al objeto HTML, y no a ngModel u otro objeto de Angular. Para ello, creamos la referencia con '#' pero sin asignarle nada.

```
      <div class="form-group row">
        <label for="inputEmail3" class="col-sm-2 col-form-label text-sm-right">
```

```
          Imagen</label>
        <div class="col-sm-10">
          <input type="file" #fileImage class="form-control" accept="image/*"
                 (change)="changeImage(fileImage)">
        </div>
      </div>
      <div class="row">
        <div class="col-sm-10 offset-sm-2">
          <img class="product-img" [src]="product.imageUrl">
        </div>
      </div>
```

Este es el método que transforma el archivo a Base64. Simplemente debemos asignar a la propiedad **product .imageUrl** el resultado de la conversión.

```
changeImage(fileInput: HTMLInputElement) {
  if(!fileInput.files || fileInput.files.length === 0) return;

  let reader:FileReader = new FileReader();
  reader.readAsDataURL(fileInput.files[0]);
  reader.addEventListener('loadend', e => {
    this.product.imageUrl = reader.result;
  });
}
```

Formularios reactivos

Mientras que los formularios de plantilla son más sencillos de implementar, los **formularios reactivos** son más flexibles, dinámicos y más sencillos de probar (pruebas unitarias). Un formulario (independiente del tipo) se representa internamente con un objeto **FormGroup**, mientras que un campo se representa con **FormControl**.

Para empezar con los formularios reactivos, en lugar (o además) de importar FormsModule, importaremos **ReactiveFormsModule**.

```
import { ReactiveFormsModule } from '@angular/forms';
...
@NgModule({
  ...
  imports: [
    ...
    ReactiveFormsModule
  ],
  ...
})
export class AppModule { }
```

Creando un formulario Reactivo

Estos formularios están más centrados en el componente que en la plantilla HTML. Lo primero que haremos es crear un objeto **FormGroup** que representará al formulario en si (podemos crear más objetos FormGroup dentro de un formulario para agrupar campos del mismo). Cada campo del formulario que queramos controlar se representa con un objeto **FormControl**.

Por ejemplo, si queremos un formulario para registrar un usuario con los campos **name**, **email** y **password**, crearemos un formulario como este:

```
export class AppComponent implements OnInit {
  userForm: FormGroup;
  title = '';

  ngOnInit() {
    this.userForm = new FormGroup({
      name: new FormControl(),
      email: new FormControl(),
      password: new FormControl()
    });
  }
```

Los valores por defecto se los pasamos al constructor de FormControl:

```
receiveInfo: new FormControl(true) // Esto es un checkbox (checked = true)
```

En la plantilla, usaremos la directiva **formGroup** para vincular el elemento <form> con el objeto FormGroup que hemos creado en el componente, y también **formControlName** para vincular cada campo con su objeto FormControl.

```
<form [formGroup]="userForm">
  <input type="text" placeholder="Name" formControlName="name">
  <input type="email" placeholder="email" formControlName="email">
```

```
  <input type="password" placeholder="Password" formControlName="password">
</form>
```

A partir de ese objeto FormGroup, podemos acceder a los valores del formulario y de sus campos, sin necesidad de crearnos una referencia en la plantilla.

```
<p>{{userForm.value | json}}</p>
<p>Name modified: {{userForm.get('name').dirty}}</p>
```

Podemos establecer todos los valores de un formulario con el método setValue. Sin embargo, esto sobrescribe los valores de **todos** los campos. Para establecer sólo algunos valores, usaríamos **patchValue**.

```
export class AppComponent implements OnInit {
  ...
  setDemoData() {
    this.userForm.setValue({
      name: 'Test user',
      email: 'test@test.com',
      password: 'test'
    });
  }
}
```

Una forma más sencilla de crear un formulario reactivo con valores por defecto, es usar el servicio **FormBuilder** de Angular (el resultado es 100% equivalente).

```
export class AppComponent implements OnInit {
  ...
  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.userForm = this.fb.group({
      name: '', // Nombre del campo: Valor inicial
      email: '',
      password: ''
    });
  }
  ...
}
```

Validación

Como dijimos al principio, los formularios reactivos nos proporcionan mayor flexibilidad para controlar todo. Esto es especialmente cierto para la validación. En lugar de una validación con atributos HTML limitados, podemos crear nuevas validaciones, validar de un modo u otro según el tipo de usuario, dependiendo del valor de otro campo, etc.

Usando el servicio FormBuilder, especificamos los validadores después del valor por defecto en un campo (ahora debemos usar un array, ya que hay más de un valor asociado a un campo). Si necesitamos usar más de un validador para un campo, podemos incluir varios dentro de un array. Los validadores están definidos en la clase **Validators**.

```
  ngOnInit() {
    this.userForm = this.fb.group({
      name: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
```

```
      password: ['', [Validators.required, Validators.minLength(5)]]
    });
  }
```

En cualquier momento, se pueden modificar, o quitar los validadores de cualquier campo (FormControl) usando métodos como **setValidators** o **clearValidators**. Esto no re-evalúa el campo automáticamente, para forzarla, llamamos al método **updateValueAndValidity**. En el siguiente formulario, cuando cambiamos el sistema de notificación por correo o por teléfono, este último será obligatorio o no.

```
<form [formGroup]="userForm">
  <p><input type="text" placeholder="Name" formControlName="name"></p>
  <p><input type="email" placeholder="Email" formControlName="email"></p>
  <p><input type="tel" placeholder="Phone" formControlName="phone"></p>
  <p><input type="password" placeholder="Password"
formControlName="password"></p>
  <p>Notify by:
    <input type="radio" formControlName="notifications" value="email"
(change)="updateNotifMethod()"> Email
    <input type="radio" formControlName="notifications" value="phone"
(change)="updateNotifMethod()"> Phone
  </p>
</form>

export class AppComponent implements OnInit {
  userForm: FormGroup;
  title = '';

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.userForm = this.fb.group({
      name: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
      phone: ['', Validators.pattern(/[0-9]{9,}/)],
      notifications: 'email',
      password: ['', [Validators.required, Validators.minLength(5)]]
    });
  }

  updateNotifMethod() {
    const notif: string = this.userForm.get('notifications').value;
    const phoneControl = this.userForm.get('phone');
    if (notif === 'phone') { // phone (teléfono requerido)
      phoneControl.setValidators([Validators.required,
                                  Validators.pattern(/[0-9]{9,}/)]);
    } else { // email (teléfono no requerido)
      phoneControl.setValidators([Validators.pattern(/[0-9]{9,}/)]);
    }
    phoneControl.updateValueAndValidity(); // actualizamos validación
  }
}
```

Crear nuevos validadores

Crear nuestros propios validadores es más sencillo con los formularios reactivos, o al menos no requiere crear una directiva, sino una función (que podemos situar en un archivo aparte y exportarla para reutilizarla). Así es como quedaría el validador minDate que creamos anteriormente (en los formularios de plantilla):

```
import { AbstractControl, ValidatorFn } from '@angular/forms';

export function minDateValidator(minInputDate: string): ValidatorFn {
```

```
  return (c: AbstractControl): { [key: string]: any } => {
    if (c.value) {
      const minDate   = new Date(minInputDate);
      const inputDate = new Date(c.value);
      console.log(minDate, inputDate);
      return minDate <= inputDate ? null : {'minDate': minDate.toLocaleDateString()};
    }
    return null;
  };
}
```

Y cómo lo usaríamos cuando creamos un formulario.

```
  ngOnInit() {
    this.userForm = this.fb.group({
      ...
      birthDate: ['', minDateValidator('1900-01-01')]
    });
  }
```

Agrupando campos

Ya hemos visto que el formulario se representa en un objeto FormGroup. Dentro de dicho formulario podemos agrupar componentes usando también FormGroup para realizar cosas como validación entre campos relacionados (repetir email o contraseña, etc.), o comprobar si una parte del formulario (una dirección por ejemplo) es válida.

```
export class AppComponent implements OnInit {
  userForm: FormGroup;
  title = '';

  constructor(private fb: FormBuilder) {}

  ngOnInit() {
    this.userForm = this.fb.group({
      name: ['', Validators.required],
      emailGroup: this.fb.group({
        email: ['', [Validators.required, Validators.email]],
        emailConfirm: ['', [Validators.required, Validators.email]]
      }, { validator: matchEmail }),
      ...
    });
  }
  ...
}

export function matchEmail(c: AbstractControl): { [key: string]: any } {
  const email = c.get('email').value;
  const email2 = c.get('emailConfirm').value;
  return email === email2 ? null : { match: true };
}
```

```
<form [formGroup]="userForm">
  <p><input type="text" placeholder="Name" formControlName="name"></p>
  <div formGroupName="emailGroup">
    <p><input type="email" placeholder="Email" formControlName="email"></p>
    <p><input type="email" placeholder="Repeat Email"
formControlName="emailConfirm"></p>
  </div>
  ...
</form>
```

Reaccionando ante cambios

Cualquier FormGroup o FormControl contiene un observable llamado **valueChanges** que devuelve el nuevo valor (o valores si es un FormGroup) cada vez que este cambia. En lugar de usar el evento **(change)** en la plantilla HTML, nos subscribimos a ese observable:

```
export class AppComponent implements OnInit {
  ...
  ngOnInit() {
    ...
    this.userForm.get('notifications').valueChanges
        .subscribe(notif => this.updateNotifMethod(notif));
  }
}
```

Duplicando campos (FormArray)

Los objetos **FormArray**, nos permiten duplicar cualquier objeto FormControl (o FormGroup) tantas veces como queramos. Por ejemplo, si queremos introducir múltiples teléfonos, direcciones, etc.

```
export class AppComponent implements OnInit {
  userForm: FormGroup;
  phones: FormArray;
  ...

  ngOnInit() {
    this.userForm = this.fb.group({
      ...
      phones: this.fb.array([this.getPhoneControl()]),
      ...
    });

    this.phones = <FormArray>this.userForm.get('phones');
    ...
  }

  getPhoneControl(): FormControl {
    const control = this.fb.control('');
    control.setValidators(Validators.pattern(/[0-9]{9,}/));
    return control;
  }

  addPhone() {
    (<FormArray>this.userForm.get('phones')).push(this.getPhoneControl());
  }
  ...
}

<form [formGroup]="userForm">
  ...
  <p formArrayName="phones"
     *ngFor="let phone of phones.controls; let i = index">
    <input type="tel" placeholder="Phone {{i + 1}}" [formControlName]="i">
  </p>
  ...
</form>
<p><button (click)="addPhone()">Add Phone</button></p>
<p>{{userForm.value | json}}</p>
<p>Phone 1: {{phones.get('0').value}}</p>
```