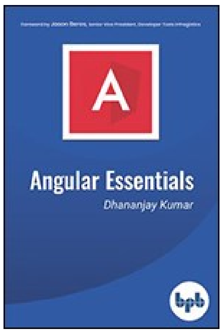


Angular Essentials: The Essential Guide to Learn Angular

by Dhananjay Kumar

Chapter 8: Reactive Forms

In this chapter, you will learn about Angular Reactive Forms. Following topics will be covered in this chapter:

- Creating Reactive Forms
- Adding Validations
- Using FormBuilder
- Custom Validators
- Passing parameters to Custom Validators
- setValue and patchValue
- Conditional Validation

Creating Reactive Form

Reactive forms work on model-driven approach. Validation logic and initial state of controls are defined by model object. Each change in the form state returns a new state of the model. Every control of reactive forms emits an observable, which gives status and value of the form controls. Since validation logic is part of the component class, writing tests for reactive forms is easier.

To start working with reactive forms, first add **ReactiveFormsModule** in the App Module as shown in *code listing 8.1*.

Code Listing 8.1

```
import { BrowserModule } from '@angular/platform-
browser';
import  { NgModule  }  from '@angular/core';
import  {ReactiveFormsModule}  from '@angular/forms';
import  { AppComponent  }  from './app.component';

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

Once module is imported, you need to import following classes in the component:

- FormGroup
- FormControl
- FormArray

The `FormControl` class corresponds to one individual form control, tracking its value and validity. While creating your reactive form, you will create an object of the `FormControl` class to add a control in the form. The `FormControl` constructor takes three parameters:

- Initial data value, which can be null.
- Array of synchronous validators. This is an optional parameter.
- Array of asynchronous validators. This is an optional parameter.

In the component class, you can create a **`FormControl`** as shown in *code listing 8.2.*

Code Listing 8.2

```
export class AppComponent {
      email = new FormControl('');
}
```

We are not passing any optional parameters like sync validations or async validations, but we will explore these parameters while adding validation to a **`FormControl`**.

On the View, you can use email **`FormControl`** as shown in *code listing 8.3.*

Code Listing 8.3

```
<input   [formControl]='email'
      type="text"
      placeholder="Enter Email" />
{{email.value  |  json}}
```

As you see, we are using property binding to bind the **`formControl`** email to the input element on the view. In a form, there will be more than one controls, to work with multiple controls you need **`ForGroup`** class. **`FormGroup`** is a group of **`FormControls`**. You can encapsulate various **`FormControls`** inside a **`FormGroup`**, which offers an API for:

- Tracking the validation of group of controls or form
- Tracking the value of group of controls or form

It contains child controls as its property and it corresponds to the top lever form on the view. You can think of a **`FormGroup`** as a single object, which aggregates the values of child **`FormControl`**. Each individual form control is the property of the **`FormGroup`** object.

You can create an object of **`FormGroup`** class as shown in *code listing 8.4.*

Code Listing 8.4

```
loginForm = new FormGroup({
         email:  new FormControlp  (' '),
         password:  new FormControlp   (' ')
});
```

Here we have created a login form, which is a **`FormGroup`**. It consists of two form controls for email and password. It is very easy to use a **`FormGroup`** on the template as shown in *code listing 8.5.*

Code Listing 8.5

```
   <form  [formGroup]='loginForm'  novalidate class="form">
   <input formControlName='email'
         type="text"
         class="form-control"
         placeholder="Enter Email"  />
   <input formControlName='password'
         type="password"
         class="form-control"
         placeholder="Enter Password" />
</form>
{{loginForm.value   |  json}}
{{loginForm.status   |  json }}
```

Here we're using property binding to bind your **`FormGroup`** with the form and **`formControlName`** directive to attach **`FormControl`** to a particular element on the template.

From last chapter, you have used a template driven form, you will notice that the HTML code on template is much leaner now: there is no **`ngModel`** or name attached with elements. You can find value and status of the form by using value and status property. Now, you no longer need to use template reference variable to find status and value of the form.

To submit the form, let us add a submit button on the form and a function to be called. We will modify the form as shown in *code listing 8.6.*

Code Listing 8.6

```
<form (ngSubmit)='loginUser()' [formGroup]='loginForm' novalidate class="form">
<input formControlName='email' type="text"class = "form-control" placeholder="Enter Email"  />
   <input formControlName='password' type="password"
class="form-control" placeholder="Enter Password"  />
   <button class="btn btn-default">Login</button>
</form>
```

In the component class, you can add a function to submit the form as shown in *code listing 8.7.*

Code Listing 8.7

```
export class AppComponent implements OnInit  {
   loginForm:  FormGroup;
   ngOnInit() {
      this.loginForm = new FormGroup({
            email:  new FormControl(null,  Validators.
required),
         password:  new FormControl()
      });
   }
   loginUser()   {
      console.log(this.loginForm.status);
      console.log(this.loginForm.value);
   }
}
```

Here we have just added a function called `loginUser` to handle the form submit event. Inside this function, you can read the value and status of `FormGroup` object `loginForm` using the status and value properties. As you can see, this gives you an object which aggregates the values of individual form controls.

Adding Validation

To add validation to `FormControls`, first import Validators from **@ angular/forms,** then you can use Validators while creating controls as shown in the *code listing 8.8.*

Code Listing 8.8

```
ngOnInit()   {
         this.loginForm = new FormGroup({
               email:  new FormControl(null,  Validators.
required),
            password:  new FormControl()
         });
      }
```

On the template, you can use the `FormGroup` get method to find an error in a particular form control and use it. In the *code listing 8.9,* we are checking the validation error for an email and displaying the error div.

Code Listing 8.9

```
   <div class="alert alert-danger" *ngIf="loginForm.
get('email').hasError('required')   &&        loginForm.
get('email').touched">
      Email is required
   </div>
```

You can also disable your submit button by default, and enable it when the form is valid to allow submission. This can be done as shown in *code listing 8.10:*

Code Listing 8.10

```
   <button [disabled]='loginForm.invalid' class="btn
btn-default">Login</button>
```

Putting everything together, the template with reactive forms should look like *code listing 8.11.*

Code Listing 8.11

```
<form (ngSubmit)='loginUser()' [formGroup]='loginForm'
novalidate class="form">
         <input   formControlName='email'    type="text"
class="form-control" placeholder="Enter Email" />
      <div class="alert   alert-danger" *ngIf="loginForm.
get('email').hasError('required')   &&        loginForm.
get('email').touched">
         Email is required
      </div>
      <input formControlName='password' type="password" class="form-control" placeholder="Enter Password"  />
   <divclass="alert alert-danger" *ngIf=" IloginForm.
get('password').valid && loginForm.get('email').
touched">
Password is required and should less than 10 characters
      </div>
      <button [disabled]='loginForm.invalid' class="btn
btn-default">Login</button> </form>
```

In addition, the component class will be as shown in *code listing 8.12.*

Code Listing 8.12

```
import  {  Component,  OnInit  }  from '@angular/core';
import  {  FormGroup,  FormControl,  FormArray,  Validators
}  from '@angular/forms';
@Component({
   selector:   'app-root',
   templateUrl:   './app.component.html',
   styleUrls:   ['./app.component.css']
})
export class AppComponent implements OnInit {
   loginForm:  FormGroup;
   ngOnInit()   {
      this.loginForm = new FormGroup({
         email:  new FormControl(null,   [Validators.
required,  Validators.minLength(4)]),
         password: new FormControl(null,   [Validators.
required,  Validators.maxLength(8)])
         })
   }
   loginUser()   {
      console.log(this.loginForm.status);
      console.log(this.loginForm.value);
   }
}
```

Using FormBuilder

`FormBuilder` is used to simplify the syntax for **`FormGroup`** and **`FormControl`**. This is very useful when your form gets lengthy. Let us refactor **`loginForm`** to use **`FormBuilder`**. To do so, first import **`FormBuilder`** from **@angular/forms** then inject it to the component as shown in *code listing 8.13.*

Code Listing 8.13

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

You can use **`FormBuilder`** to create a reactive form as shown in the following listing. As you see, it has simplified the syntax as shown in *code listing 8.14.*

Code Listing 8.14

```
   this.loginForm = this.fb.group({
            email:    [null,    [Validators.required,
Validators.minLength(4)]],
         password:   [null,   [Validators.required,
Validators.maxLength(8)]]
      });
```

The template will be the same for both **`FormBuilder`** and **`FormControl`** classes. Putting everything together, Reactive form with the **`FormBuilder`** will look like, as shown in *code listing 8.15.*

Code Listing 8.15

```
import  {  Component,  Onlnit  }  from '@angular/core';
import { FormGroup, FormControl, FormArray, Validators,
FormBuilder }  from '@angular/forms';
@Component({
   selector:   'app-root',
   templateUrl:   './app.component.html',
   styleUrls:   ['./app.component.css']
})
export class AppComponent implements OnInit {
   loginForm:  FormGroup;
   constructor(private fb:  FormBuilder)   {
   }
   ngOnInit() {

      this.loginForm = this.fb.group({
         email:    [null,    [Validators.required,
Validators.minLength(4)]],
         password:   [null,   [Validators.required,
Validators.maxLength(8)]]
      });
   }
   loginUser()   {
      console.log(this.loginForm.status);
      console.log(this.loginForm.value);
   }
}
```

Custom Validators

Angular provides us many useful validators, including required, **`minLength, maxLength,`** and pattern. These validators are part of the Validators class, which comes with the **`@angular/forms`** package. Let us assume you want to add a required validation to the email control and a **`maxLength`** validation to the password control. You do that as shown in the *code listing 8.16.*

Code Listing 8.16

```
   this.loginForm = new FormGroup({
         email:  new FormControl(null,   [Validators.
required]),
      password: new FormControl(null,   [Validators.
required,  Validators.maxLength(8)]),
            age:  new FormControl(null)
         });
```

On the template, you can use validators to show or hide an error message as shown the *code listing 8.17.* Essentially, you are reading the **`formControl`** using the **`get`** () method and checking whether it as an eror ornot using the nasnrror() method. You are also checking whether the **`formConrrol`** is touched or nor using the touched property.

Code Listing 8.17

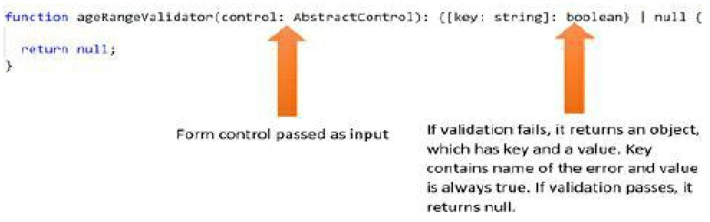
```
< inpul iorrrieont:rol Name='email' type="text" class = "form-
control" placeholder="Enter Email"  />
      <div class="alert   alert-danger" *ng!f="loginForm.
```

```
get('email').hasError('required')&&    loginForm.
get('email').touched">
         Email is required
   </div>
```

Let us say you want the age range to be from 18 to 45. Angular does not provide us ranae validation; therefore, wewill hav$_a$ towrite a custom validatorfor*this,.*

In Anivlaa, creating a austvm validator ieas simple as credting anoider function. Taeonla thine yaa nded takeep iv minU sthai It takos one iavut parameter of type **`AbstractControl`** and it returns an object of key-^{a}alue pan-rri the ralid^ionrails. *Let* uscreate a custom validator called **`ageRangeValidator`**, where the user should be able to enter an age only if it is in the given range.

A custom validator should look like as shown in *figure 8.1*.



```
function ageRangeValidator(control: AbstractControl): {[key: string]: boolean} | null {
  return null;
}
```

Form control passed as input

If validation fails, it returns an object, which has key and a value. Key contains name of the error and value is always true. If validation passes, it returns null.

Figure 8.1

The type of the first parameter is **`AbstractControl`** because it is a base class of **`FormControl`**, **`FormArray`**, and **`FormGroup`**, and it allows you to read the value of the control passed to the custom validator function. The custom validator returns either of the following:

- If the validation fails, it returns an object, which contains a key value pair. Key is the name of the error and the value is always Boolean true.
- If the validation does not fail, it returns null.

Now, we can implement the **`ageRangeValidator`** custom validator as shown in *code listing 8.18.*

Code Listing 8.18

```
function ageRangeValidator(control: AbstractControl):
{   [key:  string]:  boolean }   |  null  {
    if   (control.value  !== undefined &&  (isNaN(control.
value)   ||  control.value < 18   ||  control.value > 45))   {
      return {   'ageRange': true };
   }
   return null;
}
```

Here, we are hardcoding the maximum and minimum range in the validator. In the next section, we will see how to pass these parameters. Now, you can use **`ageRangeValidator`** with the age control as shown in *code listing 8.19.* As you see, you need to add the name of the custom validator function in the array:

Code Listing 8.19

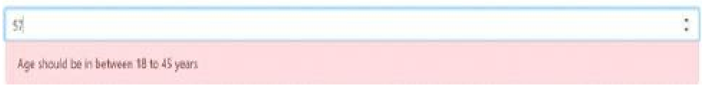
```
this.loginForm = new FormGroup({
            email:  new FormControl(null,   [Validators.
required]),
         password: new FormControl(null,   [Validators,
required,  Validators.maxLength(8)]),
      age: new FormControl(null, [ageRangeValidator])
      });
```

On the template, the custom validator can be used like any other validator. We are using the **`ageRange`** validation to show or hide the error message. Refer *code listing 8.20.*

Code Listing 8.20

```
<input formControlName='age' type="number" class="form-control" placeholder="Enter Age"  />
      <div class="alert alert-danger" *ngIf="loginForm.
get('age').dirty && loginForm.get('age').errors && loginForm.get ('c^ge') .errors.a geRa ng e ">
      Age shou Id be in between 18 to 45 years
   </div>
```

If the user does not enter an age between 18 to 45 then the reactive form will show an error as shown in *figure 8.2*.



```
57
Age should be in between 18 to 45 years
```

Figure 8.2

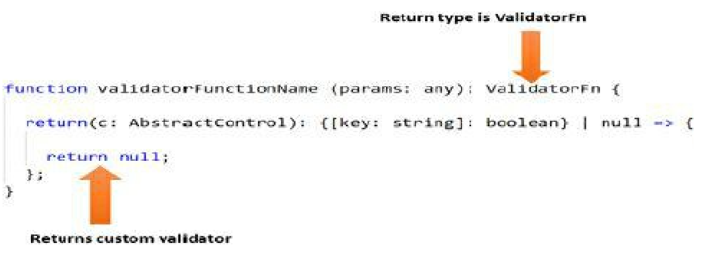
Now, agecontrol isworking withthe customvalidator. The onlyproblem with **`ageRangeValidator`** is that hardcoded age range that only validates numbers between 18 and 45. To avoid a fixed range, we need to pass the maximum and minimum age to **`ageRangeValidator`**.

Passing Parameters to a Custom Validator

An Angular custom validator does not directly take extra input parameters aside from the reference ofthe control. Topassextra parameters, youneed toadd a custom validator inside afactory function. The factoryfunction will thenreturn a custom validator. You heard it right; inJavaScript, a functioncanreturn another function. Essentially, to pass parameters to a custom validator, youneedtofollowthesesteps:

- Create a factory function and pass parameters that will be passed to the custom validator to this function.
- The return type of the factory function should be ValidatorFn which is partof @angular/forms.
- Return the custom validator from the factory function.

The factory function syntax will be as shown in *figure 8.3*.



```
function validatorFunctionName (params: any): ValidatorFn {
  return(c: AbstractControl): {[key: string]: boolean} | null => {
    return null;
  };
}
```

Figure 8.3

Now you can refactor the **`ageRangeValidator`** to accept input parameters as shown in *code listing 8.21.*

Code Listing 8.21

```
function ageRangeValidator(min: number, max: number): ValidatorFn {
   return (control: AbstractControl): { [key: string]:
boolean }   |  null => {
      if (control.value !== undefined && (isNaN(control.
value)   ||  control.value < min   ||  control.value > max))
{
            return {   'ageRange': true };
         }
         return null;
      };
}
```

We are using the input parameters max and min to validate age control. Now, you can use **`ageRangeValidator`** with age control and pass the values for max and min as shown in *code listing 8.22.*

Code Listing 8.22

```
   min = 10;
   max = 20;
   ngOnInit() {
      this.loginForm = new FormGroup({
         email:  new FormControl(null,   [Validators.
required]),
         password: new FormControl(null,   [Validators,
required,  Validators.maxLength(8)]),
   age: new FormControl(null, [ageRangeValidator(this.
min,  this.max)])
      });
   }
```

On the template, the custom validator can be used like any other validator. We are using **`ageRange`** validation to show or hide an error message as shown in *code listing 8.23.*

Code Listing 8.23

```
  <input formControlName='age' type="number"class = "form-
control" placeholder="Enter Age" />
   <div class="alert alert-danger" *ngIf="loginForm.
get('age').dirty && loginForm.get('age').errors &&
loginForm.get('age').errors.ageRange ">
     Age should be in between {{min}} to {{max}} years
   </div>
```

In this case, if the user does not enter an age between 10 and 20, the error message will be displayed.

setValue and patchValue

`setValue` and **`patchValue`** methods are used to set controls' values. These methods exist on both **`formArray`** and **`formControl.`**

Purpose of both **`setValue`** and **`patchValue`** methods is to set control's values with one major difference, **setValue** sets values of all controls inside a **`FormGroup`**, whereas **`patchValue`** can set values of a specific control.

setValue

FormGroup's class **`setValue`** method sets values of all controls inside **`FormGroup`**. If you want to set control value of **`loginForm`** created in previous section, you can do that as shown in *code listing 8.24.*

Code Listing 8.24

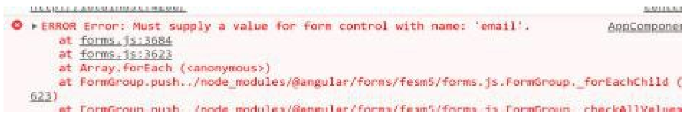
```
   this.loginForm.setValue({email: 'debugmode@outlook.
com', password:   'abc',  age  :   '30'});
```

As you see that, we are updating value of all controls. If you try to partially update control values, Angular will throw error. Consider *code listing 8.25.*

Code Listing 8.25

```
this.loginForm.setValue({password:  'abc', age :  '30'});
```

We are not setting value for email control; hence, **`setValue`** method will throw exception as shown in *image 8.4.*



```
▸ ERROR Error: Must supply a value for form control with name: 'email'.        AppCompone
      at forms.js:3684
      at forms.js:3623
      at Array.forEach (<anonymous>)
      at FormGroup.push../node_modules/@angular/forms/fesm5/forms.js.FormGroup._forEachChild (
   623)
```

Figure 8.4

`patchValue`

The **`patchValue`** allows you to set value of a particular control in form group. Using **`patchValue`**, you can opt out some controls and can update value of controls you desire. You can update password and age control of **`loginForm`** as shown in *code listing 8.26.* Keep in mind that we are not updating value of email control and still Angular is not complaining about that.

Code Listing 8.26

```
this.loginForm.patchValue({password:   'abc',  age   : '30'});
```

Both **`patchValue`** and **`setValue`** method has two more nullable parameters:

- onlySelf
- emitEvent

For **`setValue,`** when **`onlySelf`** is set to true, each change only affects this control and not its parent. Default value is set to **false.**

For **`setValue,`** when **`emitEvent`** is true or not supplied, both **`statusChanges`** and **`valueChanges`** observables emit events with latest status and value for updated control. When false, no events are emitted.

For **`patchValue,`** when **`onlySelf`** is set to true, each change only affects this control and not its parent. Default value is set to **`true.`**

For **`patchValue,`** when **`emitEvent`** is true or not supplied, both **`statusChanges`** and **`valueChanges`** observables emit events with latest status and value for updated control. When false, no events are emitted.

So other major difference you keep in mind about **`FormGroup setValue`** and **`patchValue`** methods is that by default **`setValue`** update all parent controls whereas **`patchValue`** only updates itself.

Note: `FormControl` class also has **`setValue`** and **`patchValue`** methods. Their behavior is little different from **`FormGroup`** class methods.

Conditional Validation

To understand conditional validation, let us modify login form created in previous section as shown in *code listing 8.27.*

Code Listing 8.27

```
this.loginForm = this.fb.group({
         email:   [null, Validators.required],
              password:   [null,   [Validators.required,
Validators.maxLength(8)]],
         phonenumber:   [null],
         notification:   ['email']
      });
```

On the template, we will add radio button group to handle Send Notification option. Consider *code listing 8.28.*

Code Listing 8.28

```
<form (ngSubmit)='loginUser()' [formGroup]='loginForm'
novalidate class="form">
      <input     formControlName='email'     type="text"
class="form-control" placeholder="Enter Email"  />
   <div class="alert    alert-danger" *ngIf="loginForm.
get('email').hasError(’required')   &&        loginForm.
get('email').touched">
      Email is required
   </div>
   <input  formControlName='password'   type="password"
class="form-control" placeholder="Enter Password"  />
   <input   formControlName='phonenumber'   type="text"
class="form-control"   placeholder="Enter   Phone  Number"
/>
   <div class="alert    alert-danger" *ng!f="loginForm.
get('phonenumber').hasError(’required') && loginForm.
get('phonenumber').touched">
      Phone Number is required
   </div>
   <br />
      <label   class='control-label'>Send  Notification</
```

```
label>
   <br />
   <label class="radio-inline">
            <input     type="radio"      value="email"
formControlName="notification">Email
   </label>
   <label class="radio-inline">
            <input     type="radio"      value="phone"
formControlName="notification">Phone '
   </label>
   <br />
   <button   [disabled]='loginForm.invalid'   class="btn
btn-default">Login</button>
   </form>
```

In Reactive forms both **`FormControls`** and **`FormGroups`** have a **`valueChanges`** method. It returns an observable type, so you can subscribe to it, to work with real-time value changing of **`FormControls`** or **`FormGroups`.** In our example, we need to subscribe to **`valueChanges`** of notification **`FormControl`** as shown in *code listing 8.29.*

Code Listing 8.29

```
formControlValueChanged()   {
      this.loginForm.get('notification').valueChanges.
subscribe(
         (mode:  string)  => {
            console.log(mode);
         });
   }
```

You need to call above function on **`ngOnlnit`** life cycle hook. Now when you change the selection for notification on the form in the browser console you can see, you have the most recent value. Keep in mind that, we are not handling any event on the radio button to get the latest value. Angular has a **`valueChanges`** method which returns recent value as observable on the **`FormControl`** and **`FormGroup`,** and we are subscribed to that for recent value on notification **`FormControl`.**

Our requirement is that when the notification is set to phone, then **`phonenumber FormControl`** should be a required field and if it is set to email, then **`phonenumber FormControl`** should not have any validation.

Let us modify **`formControlValueChnaged`**O function as shown in *code listing 8.30* to enable conditional validation on **`phonenumber FormControl`.**

Code Listing 8.30

```
formControlValueChanged()   {
         const   phoneControl    =    this.loginForm.
get('phonenumber');
      this.loginForm.get('notification').valueChanges.
subscribe(
         (mode:  string)  => {
            console.log(mode);
            if   (mode ===  'phone')   {
         phoneControl.setValidators([Validators.
required]);
         } else if  (mode ===  'email')   {
            phoneControl.clearValidators();
         }
         phoneControl.updateValueAndValidity();
      });
   }
```

There are a lot of codes above, so let us talk through line by line.

- Using get method of FormBuilder getting an instance of phone number FormControl
- Subscribing to the valueChanges method on notification FormControl
- Checking the current value of notification FormControl
- If the current value is phone, using setValidators method of FormControl to set required validator on phonenumber control

- If the current value is email, using clearValidators method of FormControl to clear all validation on phonenumber control
- In last calling updateValueAndValidity method to update validation rules of phonecontrol

Run the application and you will see that as you change notification value, validation of `phonenumber` is getting changed. By using the power of Angular Reactive Form's `valueChanges` method, you can achieve conditional validations and many other functionalities such as reacting to changes in the underlying data model of the reactive form.

Summary

In this chapter, we learnt about Reactive Forms in Angular. You use reactive forms to keep all validation logic and model in the component class. In this chapter you learnt about following topics:

- Create Reactive Forms
- Adding Validations
- Using FormBuilder
- Custom Validators
- Passing parameters to Custom Validators
- setValue and patchValue
- Conditional Validation