* [Introduction](https://app.pluralsight.com/guides/building-custom-validators-in-angular" \l "module-introduction)
* [The Form of Validators](https://app.pluralsight.com/guides/building-custom-validators-in-angular#module-theformofvalidators)
* [Custom Validators in Angular](https://app.pluralsight.com/guides/building-custom-validators-in-angular#module-customvalidatorsinangular)
* [Creating a Custom Form Validator](https://app.pluralsight.com/guides/building-custom-validators-in-angular#module-creatingacustomformvalidator)
* [**Conclusion**](https://app.pluralsight.com/guides/building-custom-validators-in-angular#module-conclusion)

Introduction

Most applications for web, desktop, and mobile contain various forms with fields to collect user input. But before submitting values to the server, we must validate them.

Validating form values before the submit operation is a crucial task because most of the application contains the form, so there is a chance a user may enter incorrect values or a hacker may try to steal our valuable data.

The Form of Validators

Angular supports two types of primary forms:

* Template-driven forms
* Reactive forms

Angular forms are the standard forms, but there may be additional fieldssuch as input box, checkbox, select, and so on.

Every field in a form can act as a FormControl that returns complete information about that single field, including whether the form is valid or not, the actual field value, and other information as well.

By combining all of the form controls, we can have a FormGroup, and it’s required that every single form in Angular have one FormGroup.

These are the two form approaches by which we can create forms, form controls, and form groups, Apart from that, we should have form validation so that we can validate our form before sending values to the server.

Custom Validators in Angular

To validate the different form fields, we should have some business logic so that we will be able to send the valid values to the server.

Angular supports two types of form validators: in-built validators and custom validators. Let’s go through the list of in-built form validators in Angular.

* Min
* Max
* Minlength
* Maxlength
* Required
* Compose
* Pattern

Other validators are also supported apart from these inbuilt ones, and we can also create a custom form validator if we have a specific requirement to validate any field in the form.

Creating a Custom Form Validator

As we know, sometimes in-built validators may not fulfill our validation requirement, so we may need to write custom validation logic to create custom form validators.

Custom validators take the value from the FormControl, where every input acts as a FormControl. So let’s create a form along with a validator function.

Create a new file called customvalidator.validator.ts and paste the following function.

1import { FormGroup, AbstractControl } from "@angular/forms";

// To validate password and confirm password

export function ComparePassword(

controlName: string,

matchingControlName: string

) {

return (formGroup: FormGroup) => {

const control = formGroup.controls[controlName];

const matchingControl = formGroup.controls[matchingControlName];

if (matchingControl.errors && !matchingControl.errors.mustMatch) {

return;

}

if (control.value !== matchingControl.value) {

matchingControl.setErrors({ mustMatch: true });

} else {

matchingControl.setErrors(null);

}

};

}

This is our simple custom validator function, which accepts the name of control along with matchingControlName, which basically compares two different fields for our example.

The next step is to use that custom validator function into our component to initialize the form along with the custom form validator name.

Create a new component called myform, open myform.component.ts, and paste the following source code.

1import { Component, OnInit } from "@angular/core";

2import {

3 FormBuilder,

4 FormGroup,

5 Validators,

6 AbstractControl

7} from "@angular/forms";

8import {

ComparePassword,

} from "../customvalidator/customvalidator.validator";

@Component({

3 selector: "app-myform",

templateUrl: "./myform.component.html",

styleUrls: ["./myform.component.css"]

})

export class MyformComponent implements OnInit {

registerForm: FormGroup;

submitted = false;

constructor(private formBuilder: FormBuilder) {}

ngOnInit() {

this.registerForm = this.formBuilder.group(

{

firstName: ["", [Validators.required]],

lastName: ["", [Validators.required]],

email: ["", [Validators.required, Validators.email]],

29 password: ["", [Validators.required, Validators.minLength(6)]],

confirmPassword: ["", Validators.required]

},

{

// Used custom form validator name

validator: ComparePassword("password", "confirmPassword")

}

);

}

// Getter function in order to get form controls value

get f() {

return this.registerForm.controls;

}

4 onSubmit() {

this.submitted = true;

// Returns false if form is invalid

if (this.registerForm.invalid) {

return;

}

console.log("Form Values" + JSON.stringify(this.registerForm.value));

}

54}

typescript

Let me explain what we have done in this component.

* We have initialized the form using FormBuilder and combined all the form controls using FormBuilder.
* Along with each and every form control, we have used in-built validators including required, email, and minlength.
* Apart from those in-built validators, we have used an additional validator, ComparePassword, which accepts the password, confirms it, and returns the result based on the function execution.

This is how we have developed our custom form validator. Let’s implement two more custom form validators for the first name and last name fields.

1// To validate first name

2export function ValidateFirstName(control: AbstractControl) {

3 if (!control.value.startsWith("@")) {

4 return { validFname: true };

5 }

6 return null;

7}

typescript

In this function, we will get complete form control using AbstractControl, which is the base class for FormControl, FormGroup, and FormArray.

Using that function, we are validating that the string we are using as the first name should start with the @ operator; otherwise, we return null as a result.

1// To validate last name

2export function ValidateLastName(control: AbstractControl) {

3 if (control.value.length <= 3) {

4 return { validLname: true };

5 }

6 return null;

7}

typescript

This is yet another custom form validator that accepts the last name string and validates that the last name should be more than three characters long.

So far, we have created three different custom form validators:

* ComparePassword
* ValidateFirstName
* ValidateLastName

Before moving to the template section, we need to modify our form group source code into the component like this.

1ngOnInit() {

2 this.registerForm = this.formBuilder.group(

3 {

4 firstName: ["", [Validators.required, ValidateFirstName]],

5 lastName: ["", [Validators.required, ValidateLastName]],

6 email: ["", [Validators.required, Validators.email]],

7 password: ["", [Validators.required, Validators.minLength(6)]],

8 confirmPassword: ["", Validators.required]

9 },

10 {

11 // Used custom form validator name

12 validator: ComparePassword("password", "confirmPassword")

13 }

14 );

15}

typescript

Now we are done with our custom form validator functions. Our next step is to create visual form into our app template.

Open myform.component.html and paste following lines of code.

1<form [formGroup]="registerForm" (ngSubmit)="onSubmit()">

2 <table>

3 <tr>

4 <td></td>

5 </tr>

6 <tr>

7 <td>First Name :</td>

8 <td>

9 <input type="text" formControlName="firstName" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.firstName.errors }" />

10 <div \*ngIf="submitted && f.firstName.errors" class="invalid-feedback">

11 <div \*ngIf="f.firstName.errors.required">First Name is required</div>

12 </div>

13 <div \*ngIf="registerForm.get('firstName').errors &&

14 registerForm.get('firstName').dirty &&

15 registerForm.get('firstName').errors.validFname">

16 First name should start with @

17 </div>

18 </td>

19 </tr>

20 <tr>

21 <td>Last Name :</td>

22 <td>

23 <input type="text" formControlName="lastName" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.lastName.errors }" />

24 <div \*ngIf="submitted && f.lastName.errors" class="invalid-feedback">

25 <div \*ngIf="f.lastName.errors.required">Last Name is required</div>

26 </div>

27 <div \*ngIf="registerForm.get('lastName').errors &&

28 registerForm.get('lastName').dirty &&

29 registerForm.get('lastName').errors.validLname">

30 Last name should be > 3

31 </div>

32 </td>

33 </tr>

34 <tr>

35 <td>Email :</td>

36 <td>

37 <input type="text" formControlName="email" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.email.errors }" />

38 <div \*ngIf="submitted && f.email.errors" class="invalid-feedback">

39 <div \*ngIf="f.email.errors.required">Email is required</div>

40 <div \*ngIf="f.email.errors.email">Email must be a valid email address</div>

41 </div>

42 </td>

43 </tr>

44 <tr>

45 <td>Password :</td>

46 <td>

47 <input type="password" formControlName="password" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.password.errors }" />

48 <div \*ngIf="submitted && f.password.errors" class="invalid-feedback">

49 <div \*ngIf="f.password.errors.required">Password is required</div>

50 <div \*ngIf="f.password.errors.minlength">Password must be > 6 character long</div>

51 </div>

52 </td>

53 </tr>

54 <tr>

55 <td>Confirm Password :</td>

56 <td>

57 <input type="password" formControlName="confirmPassword" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.confirmPassword.errors }" />

58 <div \*ngIf="submitted && f.confirmPassword.errors" class="invalid-feedback">

59 <div \*ngIf="f.confirmPassword.errors.required">Confirm Password is required</div>

60 <div \*ngIf="f.confirmPassword.errors.mustMatch">Password should be same as Confirm password</div>

61 </div>

62 </td>

63 </tr>

64 <tr>

65 <td>

66 <button class="btn btn-primary">Register</button>

67 </td>

68 </tr>

69 </table>

70</form>

html

This looks huge at first glance, but don't worry—we'll look at each of the controlsbriefly.

The most important part is the form tag, which denotes that this is a form along with the additional properties, like formGroup, that we have initialized in the component.

Along with the form, we have one action attached to it called ngSubmit. This means that as soon as the form gets validate and submitted, the function onSubmit will be triggered into the component.

In the form, we have different input controls, along with the additional div section in order to show some error into the form, which looks like this.

1<input type="text" formControlName="firstName" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.firstName.errors }" />

2

3<div \*ngIf="submitted && f.firstName.errors" class="invalid-feedback">

4 <div \*ngIf="f.firstName.errors.required">First Name is required</div>

5</div>

html

The div section is used to show the error for the required field validation, which directly comes from the error object of the form group.

But if we have a custom form validator, it's possible that the error object may be different. For that, we can get the errors by using the form name, which can be implemented like this into the template.

1<div \*ngIf="registerForm.get('firstName').errors &&

2 registerForm.get('firstName').dirty &&

3 registerForm.get('firstName').errors.validFname"

4>

5 The first name should start with @

6</div>

html

Here we are getting the errors for the specific form of control by using its name and other parameters, like touched, dirty, and validFname, which is our custom error generated directly from the custom validator function.

For the password comparison, we have created an error object called mustMatch into the custom validator function, hence it can be used like this.

1<input type="password" formControlName="confirmPassword" class="form-control" [ngClass]="{ 'is-invalid': submitted && f.confirmPassword.errors }" />

2

3<div \*ngIf="submitted && f.confirmPassword.errors" class="invalid-feedback">

4 <div \*ngIf="f.confirmPassword.errors.required">Confirm Password is required</div>

5 <div \*ngIf="f.confirmPassword.errors.mustMatch">

6 Password should be same as Confirm password

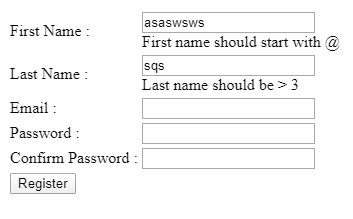
7 </div>

8</div>

html

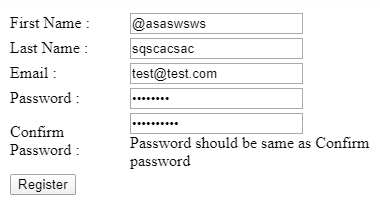
This is how we have accessed the different errors based on the error object, whether it’s froman in-built validator or from the custom validator function.

Let’s run this complete, full-fledged example and see how it works.



As you can see, we have provided wrong values against the validation criteria, and this is how the error appeared with the form control.

Now let’s provide the wrong password, confirm the password, and see whether the password comparison validator function works or not.



Provide incorrect value to each of the form fields, and we can see the error appears showing that our password and confirm password do not match.

We have now implemented complete end-to-end custom form validators using different strategies and structures. Thus we can use different approaches based on the custom validation criteria.

Conclusion

In this guide, we have gone through the different types of validators and understood why we need custom form validator in our Angular application.

Custom form validators are pretty handy when it comes to custom validation logic for the underlying business logic, and I hope this guide will help you to implement custom form validations. Stay tuned for more advanced guides.