

Custom Validator in Angular Reactive Form

5 Comments / 6 minutes of reading / May 18, 2023



[Angular Tutorial](#)

[Passing Parameter to Custom](#)

[Validation in Reactive Forms](#)

[Validator](#)



Learn how to build a custom validator in Angular Reactive form. A data entry form can contain a large no of fields. The [Angular forms](#) module makes it easier to create, manage, and validate the form fields. There are two ways in which you can create forms in Angular. One is [Reactive forms](#) & the other one is template-driven forms. Learn [how to create reactive forms](#) & [how to create template-driven forms](#).

Table of Contents

[Built-in Validators](#)

[Custom Validator in Angular Reactive Form](#)

[How to Build Custom Validator](#)

[ValidatorFn](#)

[Custom Validator Example](#)

[Using the Custom Validator](#)

[Accessing the Errors from Custom Validator](#)

[Next Steps](#)

[Summary](#)

Built-in Validators

Validating the Forms is very important, otherwise, we will end up having invalid data in our database. The [Angular Forms](#) Module provides a few built-in validators to help us to validate the form. They are listed below.

1. [Required validator](#)
2. [Min length Validator](#)
3. [Max length Validator](#)
4. [Pattern Validator](#)
5. [Email Validator](#)

We covered them in the [validation in reactive forms](#) & [validation in template-driven forms](#) tutorial.

Custom Validator in Angular Reactive Form

Built-in validators are useful but do not cover all use cases. This is where we use the custom validator. It is very easy to create a custom validator in Angular.

How to Build Custom Validator

Building a custom Validator is as easy as creating a Validator function. It is a function, which must implement [ValidatorFn](#) Interface.

ValidatorFn

The [ValidatorFn](#) is an Interface, which defines the signature of the Validator function.

```
1  
2 interface ValidatorFn {  
3   (control: AbstractControl): ValidationErrors | null
```

```
4 }  
5
```

The function takes the [AbstractControl](#). This is the base class for [FormControl](#), [FormGroup](#), and [FormArray](#). The validator function must return a list of errors i.e [ValidationErrors](#) or null if the validation has passed

Custom Validator Example

[Create a new angular application](#). Add the following code in app.component.html

```
1  
2 <h1>Custom Validator in Angular</h1>  
3  
4 <h2>Reactive Form</h2>  
5  
6 <form [formGroup]="myForm" (ngSubmit)="onSubmit()" novalidate>  
7  
8   <div>  
9     <label for="numVal">Number :</label>  
10    <input type="text" id="numVal" name="numVal" formControlName="numVal">  
11  </div>  
12  
13  <p>Is Form Valid : {{myForm.valid}} </p>  
14  
15  <p>  
16    <button type="submit" [disabled]="!myForm.valid">Submit</button>  
17  </p>  
18  
19 </form>  
20
```

[Source Code](#)

Our example app has numVal form field. We want it to be greater than 10.

Angular does not have any built-in validator for that. Hence let us build a custom Validator gte

Create a new file `gte.validator.ts` under the `app` folder.

```
1
2 import { AbstractControl, ValidationErrors } from '@angular/forms'
3
4 export function gte(control: AbstractControl): ValidationErrors | null {
5
6     const v=+control.value;
7
8     if (isNaN(v)) {
9         return { 'gte': true, 'requiredValue': 10 }
10    }
11
12    if (v <= 10) {
13        return { 'gte': true, 'requiredValue': 10 }
14    }
15
16    return null
17
18 }
19
```

[Source Code](#)

First, import the `AbstractControl` and `ValidationErrors` from the `@angular/forms`

```
1
2 import { AbstractControl, ValidationErrors } from '@angular/forms'
3
```

The validator function must adhere to the [ValidatorFn](#) Interface. It should receive the `AbstractControl` as its parameter. It can be [FormControl](#), `FormGroup` or `FormArray`.

The function must validate the control value and return `ValidationErrors` if any errors are found otherwise `null`.

```
1
2 export function gte(control: AbstractControl): ValidationErrors | null {
3
```

The `ValidationErrors` is a key-value pair object of type `[key: string]: any` and it defines the broken rule. The key is the string and should contain the name of the broken rule. The value can be anything, but usually set to `true`.

The validation logic is very simple. Check if the value of the control is a number using the [isNaN](#) method. Also, check if the value is less than or equal to 10. If both the rules are valid and then return `null`

```
1
2   const v=+control.value;
3
4   if (isNaN(v)) {
5       return { 'gte': true, 'requiredValue': 10 }
6   }
7
8   if (v <= 10) {
9       return { 'gte': true, 'requiredValue': 10 }
10  }
11
12  return null
13
```

If the validation fails then return the `ValidationErrors`. You can use anything for the key, but it is advisable to use the name of the validator i.e `gte` as the key. Also, assign `true` as value. You can as well assign a string value.

```
1  
2 return { 'gte': true, 'requiredValue': 10 }  
3
```

You can return more than one key-value pair as shown in the above example. The second key `requiredValue` returns the value 10. We use this in the template to show the error message.

Using the Custom Validator

To use this validator first, import it in the component class.

```
1  
2 import { gte } from './gte.validator';  
3
```

Add the validator to the Validator collection of the [FormControl](#) as shown below.

```
1  
2 myForm = new FormGroup({  
3   numVal: new FormControl("", [gte]),  
4 })  
5  
6
```

The complete `app.component.ts` is shown below.

```
1  
2 import { Component } from '@angular/core';  
3 import { FormGroup, FormControl, AbstractControl, ValidationErrors } from '@angular/forms';  
4 import { gte } from './gte.validator';  
5  
6 @Component({  
7   selector: 'app-root',  
8   templateUrl: './app.component.html',  
9   styleUrls: ['./app.component.css']  
10 })
```

```
11 export class AppComponent {  
12  
13   constructor() {  
14   }  
15  
16   myForm = new FormGroup({  
17     numVal: new FormControl('', [gte]),  
18   })  
19  
20   get numVal() {  
21     return this.myForm.get('numVal');  
22   }  
23  
24   onSubmit() {  
25     console.log(this.myForm.value);  
26   }  
27 }  
28
```

[Source Code](#)

Accessing the Errors from Custom Validator

We need to provide a meaningful error message to the user.

Validators return `ValidationErrors`. They are added to the control's `errors` collection of the control. The `valid` property of the control is set to `false`.

Hence we check if the `valid` property. We also check the `dirty` and `touched` property. Because we do not want to display the error message when the form is displayed for

the first time.

```
1
2 <div>
3   <label for="numVal">Number :</label>
4   <input type="text" id="numVal" name="numVal" formControlName="numVal">
5   <div *ngIf="!numVal.valid && (numVal.dirty || numVal.touched)">
6     <div *ngIf="numVal.errors.gte">
7       The number should be greater than {{numVal.errors.requiredValue}}
8     </div>
9   </div>
10
11 </div>
12
```

We check if the `gte` is true and display the error message. Note that `gte` is the name of the key we used while creating the validator.

We also make use of the `requiredValue` to show a meaningful message to the user.

```
1
2 <div *ngIf="numVal.errors.gte">
3   The number should be greater than {{numVal.errors.requiredValue}}
4 </div>
5
```

Now, we have successfully built a custom validator in reactive forms.

Next Steps

Our Custom validator is simple, but it can be improved.

Instead of the hardcoded value of 10, we want it set it while defining the validator. i.e we want to send the parameter to the validator function.