This directive adds a **custom validator** to a form control in a **template-driven form** to ensure that a user is at least a specified minimum age.

Usage in template:

html

CopyEdit

<input type="number" name="age" [(ngModel)]="age" [appMinimumAge]="18" />

**🔍 Code Breakdown**

**✅ Imports**

ts

CopyEdit

import { Directive, Input } from '@angular/core';

import { AbstractControl, ValidationErrors, Validator } from '@angular/forms';

* @Directive: Decorator to define a **custom directive**.
* Input: Used to accept values from the template ([appMinimumAge]="18").
* AbstractControl: The object representing the form control.
* ValidationErrors: Return type for validation — { [key: string]: any } if invalid, or null if valid.
* Validator: Interface to implement validate() method — the custom validator logic.

**🧩 Directive Decorator**

ts

CopyEdit

@Directive({

selector: '[appMinimumAge]',

providers: [

{

provide: 'NG\_VALIDATORS',

useExisting: MinimumAgeDirective,

multi: true

}

]

})

**1. selector: '[appMinimumAge]'**

* This makes your directive usable like this:

html

CopyEdit

<input [appMinimumAge]="18" />

**2. providers: [...]**

This is **dependency injection** configuration.

* provide: 'NG\_VALIDATORS': You're saying "I want to register a **custom validator**."
* useExisting: MinimumAgeDirective: Angular should use this class to provide the validator.
* multi: true: Multiple validators can be provided on the same control (e.g. required, minlength, etc).

Without this providers array, your validator would **never run**.

**📥 Accepting Input**

ts

CopyEdit

@Input('appMinimumAge') minAge!: number;

* @Input(...): This connects the directive to the value passed in the template, like [appMinimumAge]="18".
* minAge!: Tells TypeScript you'll definitely initialize this later (using !).

**✅ Validation Logic**

ts

CopyEdit

validate(control: AbstractControl): ValidationErrors | null {

const value = control.value;

if (value === null || value === undefined) {

return null; // Empty field - maybe another validator handles 'required'

}

const age = +value; // Convert to number

if (isNaN(age) || age < this.minAge) {

return {'minAge': { requiredAge: this.minAge, actualAge: age }};

}

return null;

}

**🔍 Step-by-step**

1. **Extract the value** from the form control:

ts

CopyEdit

const value = control.value;

1. **Handle empty values** (allow other validators to catch them like required):

ts

CopyEdit

if (value === null || value === undefined) return null;

1. **Convert input to number**:

ts

CopyEdit

const age = +value;

1. **Validation condition**:

ts

CopyEdit

if (isNaN(age) || age < this.minAge)

1. **If invalid**, return an error object with details:

ts

CopyEdit

return {'minAge': { requiredAge: this.minAge, actualAge: age }};

1. **If valid**, return null (no error):

ts

CopyEdit

return null;

**🧱 Constructor**

ts

CopyEdit

constructor() { }

This is not doing anything now, but could be used to inject services if needed.

**✅ Summary: Why Use It?**

| **Feature** | **Why It's Useful** |
| --- | --- |
| @Directive | Creates a reusable attribute-based validator |
| @Input('appMinimumAge') | Accepts dynamic input (18, 21, etc.) from the template |
| providers: NG\_VALIDATORS | Registers this directive as a validator |
| Implements Validator | Provides the validate() function that Angular calls automatically |
| Return `ValidationErrors | null` |
| Works with ngModel | Allows integration with **template-driven forms** (not just reactive) |

**🧪 Bonus – How Angular Uses It**

When Angular sees this:

html

CopyEdit

<input type="number" [appMinimumAge]="18" name="age" [(ngModel)]="age" />

It:

1. Detects the directive via the [appMinimumAge] selector.
2. Calls the validate() method with the input’s AbstractControl.
3. Reacts to the returned result:
   * null = valid
   * { minAge: {...} } = invalid, show errors in the UI

Password Directive:

**✅ Objective**

We want to show an error if:

yaml

CopyEdit

password !== confirmPassword

So let’s build a directive like:

html

CopyEdit

<input name="password" [(ngModel)]="password" />

<input name="confirmPassword" [(ngModel)]="confirmPassword" [appMatchPassword]="password" />

**📦 Step-by-step Implementation**

**1. 🔧 Create the Directive match-password.directive.ts**

ts

CopyEdit

import { Directive, Input } from '@angular/core';

import { AbstractControl, NG\_VALIDATORS, ValidationErrors, Validator } from '@angular/forms';

@Directive({

selector: '[appMatchPassword]',

providers: [

{

provide: NG\_VALIDATORS,

useExisting: MatchPasswordDirective,

multi: true

}

]

})

export class MatchPasswordDirective implements Validator {

@Input('appMatchPassword') matchTo!: string; // the password value to match

validate(control: AbstractControl): ValidationErrors | null {

if (!control || !control.parent) return null;

const passwordControl = control.parent.get('password');

const confirmPasswordControl = control;

if (!passwordControl || !confirmPasswordControl) return null;

const password = passwordControl.value;

const confirmPassword = confirmPasswordControl.value;

if (password !== confirmPassword) {

return { passwordMismatch: true };

}

return null;

}

}

**✅ Usage in Template (Template-Driven Form)**

html

CopyEdit

<input

type="password"

name="password"

[(ngModel)]="password"

#passwordRef="ngModel"

required

/>

<input

type="password"

name="confirmPassword"

[(ngModel)]="confirmPassword"

[appMatchPassword]="password"

#confirmPasswordRef="ngModel"

required

/>

This approach will be best for the apps that wants authentication for every task

<!-- Show error -->

<div \*ngIf="confirmPasswordRef.invalid && confirmPasswordRef.touched">

<p \*ngIf="confirmPasswordRef.errors?.['passwordMismatch']" class="text-red-500 text-xs">

Passwords do not match

</p>

</div>

**🧠 How It Works**

* @Input('appMatchPassword') matchTo: You bind the actual password to compare against.
* validate(control: AbstractControl) runs every time the confirmPassword value changes.
* It compares control.value (confirm password) to the passed-in password value.
* If mismatched, it returns { passwordMismatch: true } and Angular marks the control invalid.

**⚠️ Important Note**

For template-driven forms, we don’t have access to FormControlName, so we manually compare values via control.parent.get('password').

**✅ Result**

You get full validation with a clean API:

html

CopyEdit

<input name="confirmPassword" [appMatchPassword]="password" />

Q. Why use forward ref?

**🔍 Problem Without forwardRef**

When you register a custom validator directive in Angular using the providers array like this:

ts

CopyEdit

providers: [

{

provide: NG\_VALIDATORS,

useExisting: MinimumAgeDirective, // ❌ This line causes an issue

multi: true

}

]

Angular **evaluates this line before** the MinimumAgeDirective class is fully defined.

That’s because:

* The providers array is processed **at the time of decorator execution**.
* At that moment, MinimumAgeDirective is not **yet constructed** in memory.
* Therefore, useExisting: MinimumAgeDirective leads to **undefined**, which throws a runtime error or causes your validator to never register.

**✅ How forwardRef() Fixes This**

Angular’s forwardRef() function **wraps a function** that returns the reference **later**, after all declarations are fully initialized.

ts

CopyEdit

providers: [

{

provide: NG\_VALIDATORS,

useExisting: forwardRef(() => MinimumAgeDirective), // ✅ This delays evaluation

multi: true

}

]

This tells Angular:

“Hey, I know MinimumAgeDirective isn’t defined yet, but once it is, use it for NG\_VALIDATORS.”

**🧠 When to Use forwardRef**

Use it when:

* You're referencing a class **within its own decorator**.
* The referenced class hasn't been fully defined yet at the time of execution.

You **don’t need** it when:

* The class is defined in another file/module and already compiled/loaded.
* You’re not using it inside the decorator (providers, useExisting, etc.) of the same class.

**🛠️ Summary**

| **🔍 Concept** | **✅ Explanation** |
| --- | --- |
| Why it’s needed | Prevents premature reference to a class that hasn’t been constructed yet |
| What it does | Defers evaluation until class is fully defined |
| Where it's used | useExisting, useClass, useFactory inside decorators of the **same class** |