Form Validation

Improve overall data quality by validating user input for accuracy and completeness.

This page shows how to validate user input in the UI and display useful validation messages using both reactive and template-driven forms. It assumes some basic knowledge of the two forms modules.

If you're new to forms, start by reviewing the [Forms](guide/forms) and [Reactive Forms](guide/reactive-forms) guides.

Template-driven validation

To add validation to a template-driven form, you add the same validation attributes as you would with native HTML form validation. Angular uses directives to match these attributes with validator functions in the framework.

Every time the value of a form control changes, Angular runs validation and generates either a list of validation errors, which results in an INVALID status, or null, which results in a VALID status.

You can then inspect the control's state by exporting `ngModel` to a local template variable. The following example exports `NgModel` into a variable called `name`:

Note the following:

- The `<input>` element carries the HTML validation attributes: `required` and `minlength`. It also carries a custom validator directive, `forbiddenName`. For more information, see Custom validators section.
- `#name="ngModel"` exports `NgModel` into a local variable called `name`. `NgModel` mirrors many of the properties of its underlying `FormControl` instance, so you can use this in the template to check for control states such as `valid` and `dirty`. For a full list of control properties, see the AbstractControl API reference.
- The `*ngIf` on the `<div>` element reveals a set of nested message `divs` but only if the `name` is invalid and the control is either `dirty` or `touched`.
- Each nested `<div>` can present a custom message for one of the possible validation errors. There are messages for `required`, `minlength`, and `forbiddenName`.

Why check _dirty_ and _touched_? You may not want your application to display errors before the user has a chance to edit the form. The checks for `dirty` and `touched` prevent errors from showing until the user does one of two things: changes the value, turning the control dirty; or blurs the form control element, setting the control to touched.

Reactive form validation

In a reactive form, the source of truth is the component class. Instead of adding validators through attributes in the template, you add validator functions directly to the form control model in the component class. Angular then calls these functions whenever the value of the control changes.

Validator functions

There are two types of validator functions: sync validators and async validators.

- **Sync validators**: functions that take a control instance and immediately return either a set of validation errors or `null`. You can pass these in as the second argument when you instantiate a `FormControl`.
- **Async validators**: functions that take a control instance and return a Promise or Observable that later emits a set of validation errors or `null`. You can pass these in as the third argument when you instantiate a `FormControl`.

Note: for performance reasons, Angular only runs async validators if all sync validators pass. Each must complete before errors are set.

Built-in validators

You can choose to write your own validator functions, or you can use some of Angular's built-in validators.

The same built-in validators that are available as attributes in template-driven forms, such as `required` and `minlength`, are all available to use as functions from the `Validators` class. For a full list of built-in validators, see the Validators API reference.

To update the hero form to be a reactive form, you can use some of the same built-in validators—this time, in function form. See below:

{@a reactive-component-class}

Note that:

- The name control sets up two built-in validators— `Validators.required` and `Validators.minLength(4)` —and one custom validator, `forbiddenNameValidator`. For more details see the Custom validators section in this guide.
- As these validators are all sync validators, you pass them in as the second argument.
- Support multiple validators by passing the functions in as an array.
- This example adds a few getter methods. In a reactive form, you can always access any form control through the `get` method on its parent group, but sometimes it's useful to define getters as shorthands for the template.

If you look at the template for the name input again, it is fairly similar to the template-driven example.

Key takeaways:

- The form no longer exports any directives, and instead uses the `name` getter defined in the component class.
- The `required` attribute is still present. While it's not necessary for validation purposes, you may want to keep it in your template for CSS styling or accessibility reasons.

Custom validators

Since the built-in validators won't always match the exact use case of your application, sometimes you'll want to create a custom validator.

Consider the `forbiddenNameValidator` function from previous examples in this guide. Here's what the definition of that function looks like:

The function is actually a factory that takes a regular expression to detect a *specific* forbidden name and returns a validator function.

In this sample, the forbidden name is "bob", so the validator will reject any hero name containing "bob". Elsewhere it could reject "alice" or any name that the configuring regular expression matches.

The `forbiddenNameValidator` factory returns the configured validator function. That function takes an Angular control object and returns *either* null if the control value is valid *or* a validation error object. The validation error object typically has a property whose name is the validation key, `'forbiddenName'`, and whose value is an arbitrary dictionary of values that you could insert into an error message, `{name}`.

Custom async validators are similar to sync validators, but they must instead return a Promise or Observable that later emits null or a validation error object. In the case of an Observable, the Observable must complete, at which point the form uses the last value emitted for validation.

Adding to reactive forms

In reactive forms, custom validators are fairly simple to add. All you have to do is pass the function directly to the `FormControl`.

Adding to template-driven forms

In template-driven forms, you don't have direct access to the `FormControl` instance, so you can't pass the validator in like you can for reactive forms. Instead, you need to add a directive to the template.

The corresponding `ForbiddenValidatorDirective` serves as a wrapper around the `forbiddenNameValidator`.

Angular recognizes the directive's role in the validation process because the directive registers itself with the `NG_VALIDATORS` provider, a provider with an extensible collection of validators.

The directive class then implements the `Validator` interface, so that it can easily integrate with Angular forms. Here is the rest of the directive to help you get an idea of how it all comes together:

Once the `ForbiddenValidatorDirective` is ready, you can simply add its selector, `forbiddenName`, to any input element to activate it. For example:

You may have noticed that the custom validation directive is instantiated with `useExisting` rather than `useClass`. The registered validator must be _this instance_ of the `ForbiddenValidatorDirective`—the instance in the form with its `forbiddenName` property bound to “bob". If you were to replace `useExisting` with `useClass`, then you’d be registering a new class instance, one that doesn’t have a `forbiddenName`.

Control status CSS classes

Like in AngularJS, Angular automatically mirrors many control properties onto the form control element as CSS classes. You can use these classes to style form control elements according to the state of the form. The following classes are currently supported:

- `.ng-valid`
- `.ng-invalid`
- `.ng-pending`
- `.ng-pristine`
- `.ng-dirty`
- `.ng-untouched`
- `.ng-touched`

The hero form uses the `.ng-valid` and `.ng-invalid` classes to set the color of each form control's border.

You can run the to see the complete reactive and template-driven example code.