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 and 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`:

template/hero-form-template.component.html (name)

```
<input id="name" name="name"
class="form-control"
      required minlength="4"
appForbiddenName="bob"
      [(ngModel)]="hero.name"
#name="ngModel" >

<div *ngIf="name.invalid && (name.dirty
|| name.touched)"
    class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters
long.
  </div>
  <div
*ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
```

```
    </div>

  </div>
```

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:

reactive/hero-form-reactive.component.ts (validator functions)

```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    'name': new
FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
      forbiddenNameValidator(/bob/i) //
<-- Here's how you pass in the custom
validator.
    ]),
    'alterEgo': new
FormControl(this.hero.alterEgo),
    'power': new
FormControl(this.hero.power,
Validators.required)
  });

}

get name() { return
this.heroForm.get('name'); }
```

```
get power() { return
this.heroForm.get('power'); }
```

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.

reactive/hero-form-reactive.component.html (name with error msg)

```
<input id="name" class="form-control"
      formControlName="name" required >

<div *ngIf="name.invalid && (name.dirty
|| name.touched)"
    class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters
long.
  </div>
  <div
*ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
  </div>
</div>
```

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:

shared/forbidden-name.directive.ts (forbiddenNameValidator)

```
/** A hero's name can't match the given regular expression */
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
  return (control: AbstractControl): {[key: string]: any} | null => {
    const forbidden = nameRe.test(control.value);
    return forbidden ? {'forbiddenName': {value: control.value}} : null;
  };
}
```

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`.

reactive/hero-form-reactive.component.ts (validator functions)

```
this.heroForm = new FormGroup({
  'name': new
FormControl(this.hero.name, [
    Validators.required,
    Validators.minLength(4),
    forbiddenNameValidator(/bob/i) //
<-- Here's how you pass in the custom
validator.
  ]),
  'alterEgo': new
FormControl(this.hero.alterEgo),
  'power': new
FormControl(this.hero.power,
Validators.required)
});
```

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.

shared/forbidden-name.directive.ts (providers)

```
providers: [{provide: NG_VALIDATORS,
useExisting:
ForbiddenValidatorDirective, multi:
true}]
```

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:

shared/forbidden-name.directive.ts (directive)

```
@Directive({
  selector: '[appForbiddenName]',
  providers: [{provide: NG_VALIDATORS,
useExisting:
ForbiddenValidatorDirective, multi:
true}]
})
export class
ForbiddenValidatorDirective implements
Validator {
  @Input('appForbiddenName')
forbiddenName: string;

  validate(control: AbstractControl):
{[key: string]: any} | null {
    return this.forbiddenName ?
forbiddenNameValidator(new
RegExp(this.forbiddenName, 'i'))
(control)
                              : null;
```

```
    }
  }
```

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

template/hero-form-template.component.html (forbidden-name-input)

```
<input id="name" name="name"
class="form-control"
      required minlength="4"
appForbiddenName="bob"
      [(ngModel)]="hero.name"
#name="ngModel" >
```

> 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.

forms.css (status classes)

```
.ng-valid[required], .ng-valid.required
{
  border-left: 5px solid #42A948; /*
green */
}

.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /*
red */
}
```

Cross field validation

This section shows how to perform cross field validation. It assumes some basic knowledge of creating custom validators.

> If you haven't created custom validators before, start by reviewing the custom validators section.

In the following section, we will make sure that our heroes do not reveal their true identities by filling out the Hero Form. We will do that by validating that the hero names and alter egos do not match.

Adding to reactive forms

The form has the following structure:

```
const heroForm = new FormGroup({
  'name': new FormControl(),
  'alterEgo': new FormControl(),
  'power': new FormControl()
});
```

Notice that the name and alterEgo are sibling controls. To evaluate both controls in a single custom

validator, we should perform the validation in a common ancestor control: the `FormGroup`. That way, we can query the `FormGroup` for the child controls which will allow us to compare their values.

To add a validator to the `FormGroup`, pass the new validator in as the second argument on creation.

```
const heroForm = new FormGroup({
  'name': new FormControl(),
  'alterEgo': new FormControl(),
  'power': new FormControl()
}, { validators:
identityRevealedValidator });
```

The validator code is as follows:

shared/identity-revealed.directive.ts

```
/** A hero's name can't match the
hero's alter ego */
export const identityRevealedValidator:
ValidatorFn = (control: FormGroup):
ValidationErrors | null => {
  const name = control.get('name');
  const alterEgo =
control.get('alterEgo');

  return name && alterEgo && name.value
=== alterEgo.value ? {
'identityRevealed': true } : null;
};
```

The identity validator implements the `ValidatorFn` interface. It takes an Angular control object as an argument and returns either null if the form is valid, or `ValidationErrors` otherwise.

First we retrieve the child controls by calling the `FormGroup`'s `get` method. Then we simply compare the values of the `name` and `alterEgo` controls.

If the values do not match, the hero's identity remains secret, and we can safely return null. Otherwise, the hero's identity is revealed and we must mark the form as invalid by returning an error object.

Next, to provide better user experience, we show an appropriate error message when the form is invalid.

reactive/hero-form-template.component.html

```
<div
*ngIf="heroForm.errors?.identityReveal
 && (heroForm.touched ||
heroForm.dirty)" class="cross-
validation-error-message alert alert-
danger">
    Name cannot match alter ego.
</div>
```

Note that we check if:

- the `FormGroup` has the cross validation error returned by the `identityRevealed` validator,
- the user is yet to interact with the form.

Adding to template driven forms

First we must create a directive that will wrap the validator function. We provide it as the validator using the `NG_VALIDATORS` token. If you are not sure why, or you do not fully understand the syntax, revisit the previous section.

shared/identity-revealed.directive.ts

```
@Directive({
  selector: '[appIdentityRevealed]',
  providers: [{ provide: NG_VALIDATORS,
useExisting:
IdentityRevealedValidatorDirective,
multi: true }]
})
export class
IdentityRevealedValidatorDirective
implements Validator {
  validate(control: AbstractControl):
ValidationErrors {
    return
identityRevealedValidator(control)
  }
}
```

Next, we have to add the directive to the html template. Since the validator must be registered at the highest level in the form, we put the directive on the `form` tag.

template/hero-form-template.component.html

```
<form #heroForm="ngForm"
appIdentityRevealed>
```

To provide better user experience, we show an appropriate error message when the form is invalid.

template/hero-form-template.component.html

```
<div
*ngIf="heroForm.errors?.identityReveal
 && (heroForm.touched ||
heroForm.dirty)" class="cross-
validation-error-message alert alert-
danger">
    Name cannot match alter ego.
</div>
```

Note that we check if:

- the form has the cross validation error returned by the `identityRevealed` validator,
- the user is yet to interact with the form.

This completes the cross validation example. We managed to:

- validate the form based on the values of two sibling controls,
- show a descriptive error message after the user interacted with the form and the validation failed.

Async Validation

This section shows how to create asynchronous validators. It assumes some basic knowledge of creating custom validators.

The Basics

Just like synchronous validators have the `ValidatorFn` and `Validator` interfaces,

asynchronous validators have their own counterparts: `AsyncValidatorFn` and `AsyncValidator`.

They are very similar with the only difference being:

- They must return a Promise or an Observable,
- The observable returned must be finite, meaning it must complete at some point. To convert an infinite observable into a finite one, pipe the observable through a filtering operator such as `first`, `last`, `take`, or `takeUntil`.

It is important to note that the asynchronous validation happens after the synchronous validation, and is performed only if the synchronous validation is successful. This check allows forms to avoid potentially expensive async validation processes such as an HTTP request if more basic validation methods fail.

After asynchronous validation begins, the form control enters a `pending` state. You can inspect the control's `pending` property and use it to give visual feedback about the ongoing validation.

A common UI pattern is to show a spinner while the async validation is being performed. The following example presents how to achieve this with template-driven forms:

```
<input [(ngModel)]="name"
#model="ngModel" appSomeAsyncValidator>
<app-spinner *ngIf="model.pending">
</app-spinner>
```

Implementing Custom Async Validator

In the following section, validation is performed asynchronously to ensure that our heroes pick an alter ego that is not already taken. New heroes are constantly enlisting and old heroes are leaving the service. That means that we do not have the list of available alter egos ahead of time.

To validate the potential alter ego, we need to consult a central database of all currently enlisted heroes. The process is asynchronous, so we need a special validator for that.

Let's start by creating the validator class.

```
@Injectable({ providedIn: 'root' })
export class UniqueAlterEgoValidator
implements AsyncValidator {
  constructor(private heroesService:
HeroesService) {}

  validate(
    ctrl: AbstractControl
  ): Promise<ValidationErrors | null> |
Observable<ValidationErrors | null> {
    return
this.heroesService.isAlterEgoTaken(ctrl.

      map(isTaken => (isTaken ? {
uniqueAlterEgo: true } : null)),
      catchError(() => of(null))
    );
  }
}
```

As you can see, the `UniqueAlterEgoValidator` class implements the `AsyncValidator` interface. In the constructor, we inject the `HeroesService` that has the following interface:

```
interface HeroesService {
  isAlterEgoTaken: (alterEgo: string)
=> Observable<boolean>;
}
```

In a real world application, the `HeroesService` is responsible for making an HTTP request to the hero database to check if the alter ego is available. From the validator's point of view, the actual implementation of the service is not important, so we can just code against the `HeroesService` interface.

As the validation begins, the `UniqueAlterEgoValidator` delegates to the `HeroesService` `isAlterEgoTaken()` method with the current control value. At this point the control is marked as `pending` and remains in this state until the observable chain returned from the `validate()` method completes.

The `isAlterEgoTaken()` method dispatches an HTTP request that checks if the alter ego is available, and returns `Observable<boolean>` as the result. We pipe the response through the `map` operator and transform it into a validation result. As always, we return `null` if the form is valid, and `ValidationErrors` if it is not. We make sure to handle any potential errors with the `catchError` operator.

Here we decided that `isAlterEgoTaken()` error is treated as a successful validation, because failure to make a validation request does not necessarily mean that the alter ego is invalid. You could handle the error differently and return the `ValidationError` object instead.

After some time passes, the observable chain completes and the async validation is done. The `pending` flag is set to `false`, and the form validity is updated.

Note on performance

By default, all validators are run after every form value change. With synchronous validators, this will not likely have a noticeable impact on application performance. However, it's common for async validators to perform some kind of HTTP request to validate the control. Dispatching an HTTP request after every keystroke could put a strain on the backend API, and should be avoided if possible.

We can delay updating the form validity by changing the `updateOn` property from `change` (default) to `submit` or `blur`.

With template-driven forms:

```
<input [(ngModel)]="name"
[ngModelOptions]="{updateOn: 'blur'}">
```

With reactive forms:

```
new FormControl('', {updateOn:
'blur'});
```

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