Validating form input

You can improve overall data quality by validating user input for accuracy and completeness. This page shows how to validate user input from the UI and display useful validation messages, in both reactive and template-driven forms.

Prerequisites

Before reading about form validation, you should have a basic understanding of the following.

- TypeScript and HTML5 programming.
- Fundamental concepts of Angular app design.
- The two types of forms that Angular supports.
- Basics of either Template-driven Forms or Reactive Forms.

> Get the complete example code for the reactive and template-driven forms used here to illustrate form validation. Run the live example / download example.

Validating input in template-driven forms

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 that 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>
```

Notice the following features illustrated by the example.

- The `<input>` element carries the HTML validation attributes: `required` and `minlength`. It also carries a custom validator directive, `forbiddenName`. For more information, see the 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`.

> To prevent the validator from displaying errors before the user has a chance to edit the form, you should check for either the `dirty` or `touched` states in a control.
>
> - When the user changes the value in the watched field, the control is marked as "dirty".
> - When the user blurs the form control element, the control is marked as "touched".

Validating input in reactive forms

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

Validator functions can be either synchronous or asynchronous.

- **Sync validators**: Synchronous 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**: Asynchronous 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`.

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

Built-in validator functions

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, as in the following example.

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'); }
```

In this example, the `name` control sets up two built-in validators—`Validators.required` and `Validators.minLength(4)`—and one custom validator, `forbiddenNameValidator`. (For more details see custom validators below.)

All of these validators are synchronous, so they are passed as the second argument. Notice that you can support multiple validators by passing the functions in as an array.

This example also 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 shorthand 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>
```

This form differs from the template-driven version in that it no longer exports any directives. Instead, it uses the `name` getter defined in the component class.

Notice that the `required` attribute is still present in the template. Although it's not necessary for validation, it should be retained to for accessibility purposes.

Defining custom validators

The built-in validators don't always match the exact use case of your application, so you sometimes need to create a custom validator.

Consider the `forbiddenNameValidator` function from previous reactive-form examples. 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 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 custom validators to reactive forms

In reactive forms, add a custom validator by passing 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 custom validators to template-driven forms

In template-driven forms, add a directive to the template, where the directive wraps the validator function. For example, 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, as shown in the following example. `NG_VALIDATORS` is a predefined 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): ValidationErrors | null {
    return this.forbiddenName ? forbiddenNameValidator(new
RegExp(this.forbiddenName, 'i'))(control)
                              : null;
  }
}
```

Once the `ForbiddenValidatorDirective` is ready, you can 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" >
```

> Notice 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

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`

In the following example, 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

A cross-field validator is a custom validator that compares the values of different fields in a form and accepts or rejects them in combination. For example, you might have a form that

offers mutually incompatible options, so that if the user can choose A or B, but not both. Some field values might also depend on others; a user might be allowed to choose B only if A is also chosen.

The following cross validation examples show how to do the following:

- Validate reactive or template-based form input based on the values of two sibling controls,
- Show a descriptive error message after the user interacted with the form and the validation failed.

The examples use cross-validation to ensure that heroes do not reveal their true identities by filling out the Hero Form. The validators do this by checking that the hero names and alter egos do not match.

Adding cross-validation 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, you must perform the validation in a common ancestor control: the `FormGroup`. You query the `FormGroup` for its child controls so that you can 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:
AbstractControl): 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.

The validator retrieves the child controls by calling the `FormGroup`'s get method, then compares the values of the `name` and `alterEgo` controls.

If the values do not match, the hero's identity remains secret, both are valid, and the validator returns null. If they do match, the hero's identity is revealed and the validator must mark the form as invalid by returning an error object.

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

reactive/hero-form-template.component.html

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

This `*ngIf` displays the error if the `FormGroup` has the cross validation error returned by the `identityRevealed` validator, but only if the user has finished interacting with the form.

Adding cross-validation to template-driven forms

For a template-driven form, you must create a directive to wrap the validator function. You provide that directive as the validator using the `NG_VALIDATORS` token, as shown in the following example.

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);
  }
}
```

You must add the new directive to the HTML template. Because the validator must be registered at the highest level in the form, the following template puts 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?.identityRevealed && (heroForm.touched ||
heroForm.dirty)" class="cross-validation-error-message alert alert-
danger">
    Name cannot match alter ego.
</div>
```

This is the same in both template-driven and reactive forms.

Creating asynchronous validators

Asynchronous validators implement the `AsyncValidatorFn` and `AsyncValidator` interfaces. These are very similar to their synchronous counterparts, with the following differences.

- The `validate()` functions 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`.

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 the more basic validation methods have already found invalid input.

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 operation.

A common UI pattern is to show a spinner while the async validation is being performed. The following example shows how to achieve this in a template-driven form.

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

Implementing a custom async validator

In the following example, an async validator ensures that heroes pick an alter ego that is not already taken. New heroes are constantly enlisting and old heroes are leaving the service, so the list of available alter egos cannot be retrieved ahead of time. To validate the potential alter ego entry, the validator must initiate an asynchronous operation to consult a central database of all currently enlisted heroes.

The following code create the validator class, `UniqueAlterEgoValidator`, which implements the `AsyncValidator` interface.

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

The constructor injects the `HeroesService`, which defines the following interface.

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

In a real world application, the `HeroesService` would be 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 the example 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. The `validate()` method pipes the response through the `map` operator and transforms it into a validation result.

The method then, like any validator, returns `null` if the form is valid, and `ValidationErrors` if it is not. This validator handles any potential errors with the `catchError` operator. In this case, the validator treats the `isAlterEgoTaken()` error 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 asynchronous validation is done. The `pending` flag is set to `false`, and the form validity is updated.

Optimizing performance of async validators

By default, all validators run after every form value change. With synchronous validators, this does not normally have a noticeable impact on application performance. Async validators, however, commonly 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.

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

With template-driven forms, set the property in the template.

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

With reactive forms, set the property in the `FormControl` instance.

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