Angular Forms

Introduction to Angular Reactive Forms and Angular Form Validation

---

Introduction

- Handling user input with forms is the cornerstone of many common applications.
- Angular provides **two different approaches** to handling user input through forms:

    1. **Reactive forms** provide direct, explicit access to the underlying forms object model.

        Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable.

    2. **Template-driven forms** rely on directives in the template to create and manipulate the underlying object model.

        They are useful for adding a simple form to an app, such as an email list signup form.

Common form foundation classes

Both reactive and template-driven forms are built on the following base classes.

- `FormControl` tracks the value and validation status of an individual form control.
- `FormGroup` tracks the same values and status for a collection of form controls.
- `FormArray` tracks the same values and status for an array of form controls.
- `ControlValueAccessor` creates a bridge between Angular FormControl instances and built-in DOM elements.

Setup in template-driven forms

- In template-driven forms, the form model is implicit, rather than explicit.
- The directive `NgModel` creates and manages a `FormControl` instance for a given form element.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-template-favorite-color',
  template: `
    Favorite Color: <input type="text" [(ngModel)]="favoriteColor">
  `
})
export class FavoriteColorComponent {
  favoriteColor = '';
}
```

Setup in reactive forms

- With reactive forms, you define the form model directly in the component class.
- The `[formControl]` directive links the explicitly created `FormControl` instance to a specific form element in the view, using an internal value accessor.

```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite Color: <input type="text"
↪   [formControl]="favoriteColorControl">
  `
})
export class FavoriteColorComponent {
  favoriteColorControl = new FormControl('');
}
```

- By creating these controls in your component class, you get immediate access to listen for, update, and validate the state of the form input.
- Reactive forms have methods to change a control's value programmatically, which gives you the flexibility to update the value without user interaction.

```
updateName() {
  this.favoriteColorControl.setValue('green');
}
```

Register reactive forms module

To use reactive form controls, import `ReactiveFormsModule` from the `@angular/forms` package and add it to your NgModule's imports array.

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule { }
```

Grouping form controls

Forms typically contain several related controls. Reactive forms provide two ways of grouping multiple related controls into a single input form.

1. A **form group** defines a form with a fixed set of controls that you can manage together.
2. A **form array** defines a dynamic form, where you can add and remove controls at run time.

`profile-editor.component.ts`

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
  });
}
```

profile-editor.component.html

```
<form [formGroup]="profileForm">

  <label for="first-name">First Name: </label>
  <input id="first-name" type="text" formControlName="firstName">

  <label for="last-name">Last Name: </label>
  <input id="last-name" type="text" formControlName="lastName">

</form>
```

Creating nested form groups

Form groups can accept both individual form control instances and other form group instances as children.

This makes composing complex form models easier to maintain and logically group together.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl('')
    })
  });
}
```

```
<div formGroupName="address">
  <h2>Address</h2>
```

```
  <label for="street">Street: </label>
  <input id="street" type="text" formControlName="street">

  <label for="city">City: </label>
  <input id="city" type="text" formControlName="city">

  <label for="state">State: </label>
  <input id="state" type="text" formControlName="state">

  <label for="zip">Zip Code: </label>
  <input id="zip" type="text" formControlName="zip">
</div>
```

Save form data

The `FormGroup` directive listens for the **submit event** emitted by the form element and emits an `ngSubmit` event that you can bind to a callback function.

```
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">
```

```
onSubmit() {
  // Use EventEmitter with form value to pass to parent component
  console.warn(this.profileForm.value);
}
```

Use a button element to add a button to the bottom of the form to trigger the form submission.

```
<p>Complete the form to enable button.</p>
<button type="submit" [disabled]="!profileForm.valid">Submit</button>
```

Update parts of the data model

There are two ways to update the model value:

1. Use the `setValue()` method to set a new value for an individual control. The `setValue()` method strictly adheres to the structure of the form group and replaces the entire value for the control.

2. Use the `patchValue()` method to replace any properties defined in the object that have changed in the form model.

```
updateProfile() {
  this.profileForm.patchValue({
    firstName: 'Nancy',
    address: {
      street: '123 Drew Street'
    }
  });
}
```

Using the FormBuilder service to generate controls

The FormBuilder service provides convenient methods for generating controls.

Import the FormBuilder class from the @angular/forms package.

```
import { FormBuilder } from '@angular/forms';
```

Inject the FormBuilder service.

```
constructor(private fb: FormBuilder) { }
```

The FormBuilder service has three methods: `control()`, `group()`, and `array()`. These are factory methods for generating instances in your component classes including form controls, form groups, and form arrays.

```
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: [''],
    lastName: [''],
    address: this.fb.group({
      street: [''],
```

```
      city: [''],
      state: [''],
      zip: ['']
    }),
  });

  constructor(private fb: FormBuilder) { }
}
```

Creating dynamic forms

- `FormArray` is an alternative to `FormGroup` for managing any number of unnamed controls.
- You can dynamically insert and remove controls from form array instances.

Import the FormArray class.

```
import { FormArray } from '@angular/forms';
```

Define a FormArray control.

```
profileForm = this.fb.group({
  firstName: ['', Validators.required],
  lastName: [''],
  address: this.fb.group({
    street: [''],
    city: [''],
    state: [''],
    zip: ['']
  }),
  aliases: this.fb.array([
    this.fb.control('')
  ])
});
```

> Note: `Validators.required` sets the `firstName` field to be a required field.

Access the FormArray control

A `getter` provides access to the aliases in the form array instance compared to repeating the `profileForm.get()` method to get each instance.

```
get aliases() {
  return this.profileForm.get('aliases') as FormArray;
}
```

The `FormArray.push()` method inserts the control as a new item in the array.

```
addAlias() {
  this.aliases.push(this.fb.control(''));
}
```

Display the form array in the template.

```
<div formArrayName="aliases">
  <h2>Aliases</h2>
  <button (click)="addAlias()" type="button">+ Add another alias</button>

  <div *ngFor="let alias of aliases.controls; let i=index">
    <!-- The repeated alias template -->
    <label for="alias-{{ i }}">Alias:</label>
    <input id="alias-{{ i }}" type="text" [formControlName]="i">
  </div>
</div>
```

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.
- You can then inspect the control's state by exporting `ngModel` to a local template variable.

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

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

```
  <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>
```

- 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. 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. Pass these in as the third argument when you instantiate a `FormControl`.

Built-in validator functions

- 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.

```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    name: new FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
    ]),
    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 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.

Validators in template file

```
<input type="text" 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.

Create a custom directive:

```
ng generate directive forbiddenName
```

Define code for custome validator:

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

Adding custom validators to reactive forms

In reactive forms, add a custom validator by passing the function directly to the `FormControl`.

```
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`.

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

  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.

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

Further Readings

- Building Dynamic Forms
  - Live Example
- YouTube Video: Angular Reactive Forms in 10 Minutes
  - Live Example
- Form Validation Example