



Angular Forms

.NET

Reactive forms provide a model-driven approach to handling form inputs whose values change over time.

[HTTPS://ANGULAR.IO/GUIDE/REACTIVE-FORMS](https://angular.io/guide/reactive-forms)

Angular Forms - Overview

<https://angular.io/start/start-forms#forms-in-angular>

<https://angular.io/api/forms/FormBuilder>

<https://angular.io/guide/forms-overview>

Angular provides two different form types: **reactive** and **template-driven**. Both capture user input **events** from the view (template), validate the user input, create a **form model** and data model to update, and provide a way to track changes.

- Reactive forms are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, use **Reactive Forms**.
- Template-driven forms are useful for adding a simple form to an app but don't scale as well as **Reactive Forms**. If you have very basic form requirements and logic that can be managed solely in the template, use **Template-Driven Forms**.

Reactive and **Template-Driven Forms** both use a **form model** to track value changes between Angular forms and form input elements.

The example shows how one of the four **form model** types, **FormControl**, is defined and created.

```
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('');
}
```

Form Model Classes

<https://angular.io/guide/forms-overview#testing>

Reactive and ***template-driven*** forms differ in how form-control instances are created and managed.

Both form types are built using these four base classes:

Class Name	Details
FormControl	tracks the value and validation status of an individual form control.
FormGroup	tracks the values and status for a collection of form controls.
FormArray	tracks the values and status for an array of form controls.
ControlValueAccessor	creates a bridge between Angular FormControl instances and native DOM elements.

Reactive (Model-Driven) Forms

<https://angular.io/start/start-forms#forms-in-angular>

<https://angular.io/api/forms/FormBuilder>

<https://angular.io/guide/forms-overview>

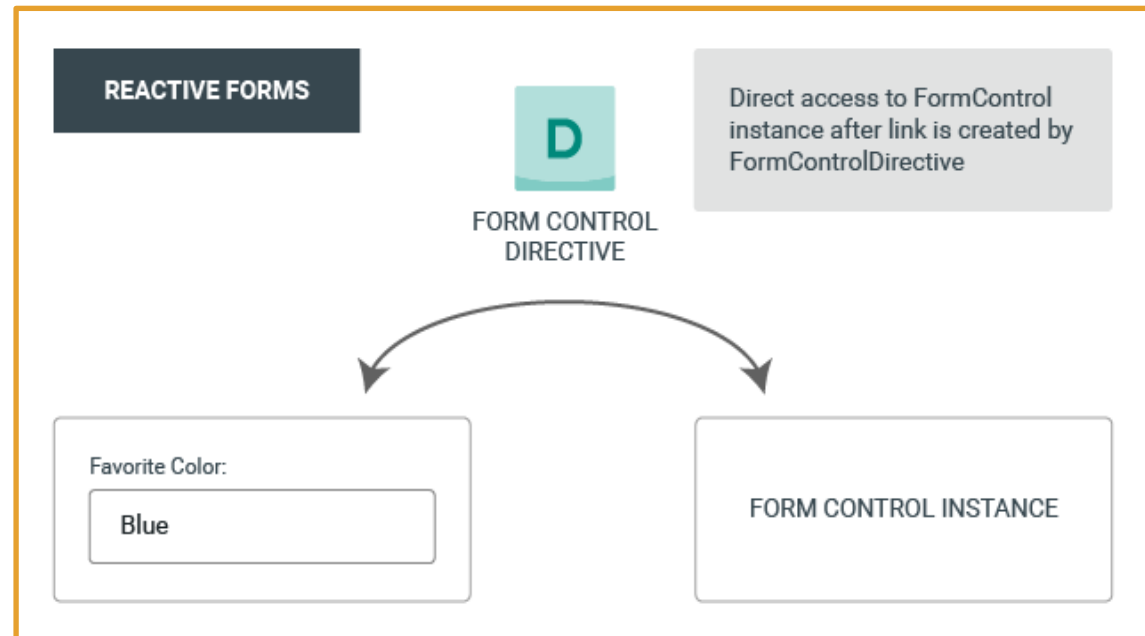
Reactive forms are built around [observable](#) streams, where form inputs and values are provided as streams of input values. Reactive forms provide a model-driven approach to handling form inputs whose values change over time.

There are two parts to an **Angular Reactive form**:

- The objects in **Component** to store and manage the form.
- The visualization of the form in the HTML **template**.

The **ReactiveFormsModule** provides the **FormBuilder** service.

The **form model** is the “source of truth” and provides the value and status of the form element at a given point in time.



Reactive Form Setup (1 / 2)

<https://angular.io/guide/reactive-forms#adding-a-basic-form-control>
<https://codecraft.tv/courses/angular/forms/model-driven/>

1. Import **ReactiveFormsModule** to **app.module.ts**. Also add it to **imports[]** in **NgModule()**:
 - `import { ReactiveFormsModule } from '@angular/forms';`
2. Generate the new component that will have the form:
 - `ng generate component [ComponentName]`.
3. Import **FormControl** and **FormGroup** into the new component:
 - `import { FormControl, FormGroup } from '@angular/forms';`
4. Create an instance of **FormGroup** in your component class to represent the form itself.
5. Create a **FormGroup** object and create **FormControl** properties matching the **<input>** elements of the form in the **.html** template.

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

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

```
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(''),
  });
}
```


Reactive Form Setup (2/2)

<https://angular.io/guide/reactive-forms#adding-a-basic-form-control>
<https://codecraft.tv/courses/angular/forms/model-driven/>

7. In the HTML template, add the *FormControl* to the `<form>` with:
 - `[formGroup]="<formGroupName>"`
8. Add the *formControlName* to each `<input>` of the form with:
 - `[formControlName]="<inputName>"`
9. Add an action to complete when the form is submitted.
10. Add the submit button to the bottom of the form.
11. Add form validation as needed.

Now, whatever input you place in the input field will be transferred to the *FormGroup's FormControl*s on the component.

```
<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>
```

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

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

Reactive Forms – Changing Values

<https://angular.io/guide/reactive-forms#replacing-a-form-control-value>

There are two ways to update the model value:

- 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.
- 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'  
        }  
    });  
}
```


Reactive Form Validation

<https://angular.io/guide/forms-overview#form-validation>

<https://angular.io/guide/reactive-forms#validating-form-input>

<https://angular.io/guide/form-validation#validating-input-in-reactive-forms>

Form validation is used to ensure that user input is complete and correct. Reactive Forms include a set of validator functions for common use cases. These functions receive a control to validate against and return an error object or a null value based on the validation check.

```
import { Validators } from '@angular/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.

When giving user inputted values to **FormControl** properties, use the **.setValue()** function to set them. This function will validate the value against your preset **FormControl** validations.

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

```
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)  
  });  
}
```

Reactive Form Validation

<https://angular.io/guide/forms-overview#form-validation>

<https://angular.io/guide/form-validation>

<https://angular.io/guide/reactive-forms#validating-form-input>

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 an 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**.
- Angular only runs async validators if all sync validators pass.

```
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)
});
```

```
<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>
```

Reactive Form Validation

<https://angular.io/api/forms/Validators>

<https://angular.io/guide/reactive-forms#validating-form-input>

Validator	Description
Min(min:number)	requires the control's value to be greater than or equal to the provided number. [Validators.min(3)]
Max(max:number)	requires the control's value to be less than or equal to the provided number. [Validators.max(15)]
Required()	requires the control have a non-empty value. [Validators.required]
requiredTrue()	requires the control's value be true. This validator is commonly used for required checkboxes. [Validators.requiredTrue]
Email()	Requires the control's value pass an email validation test. Tests the value using a regular expression pattern for common use cases. Use Validators.pattern() to validate the value against a different pattern. [Validators.email]

Reactive Form Validation

<https://angular.io/api/forms/Validators>

<https://angular.io/guide/reactive-forms#validating-form-input>

Validator	Description
<code>minLength()</code>	Requires the length of the control's value to be greater than or equal to the provided minimum length. Use only for types with a numeric length property. <code>[Validators.Minlength(3)]</code>
<code>maxLength()</code>	Requires the length of the control's value to be less than or equal to the provided maximum length. Use only for types with a numeric length property. <code>[Validators.Maxlength(15)]</code>
<code>Pattern()</code>	Requires the control's value to match a regex pattern. <code>[Validators.Pattern('[a-zA-Z]*)']</code>

Reactive Form Validation

<https://angular.io/guide/form-validation#control-status-css-classes>
<https://docs.angularjs.org/api/ng/directive/ngModel>

Angular maintains the state of control properties in the form control element as CSS **classes**. Use these classes to style form control elements according to the state of the form.

State syntax	Meaning
.ng-valid	The value in the field is within range.
.ng-invalid	The value in the field is not within range.
.ng-pending	\$asyncValidator(s) are unfulfilled
.ng-pristine	The form has not yet been modified by the user.
.ng-dirty	If the user changes the value in the field.
.ng-untouched	The field before any user interaction
.ng-touched	When the user blurs the form control element.
.ng-submitted	Only valid for the form. The form has been submitted.

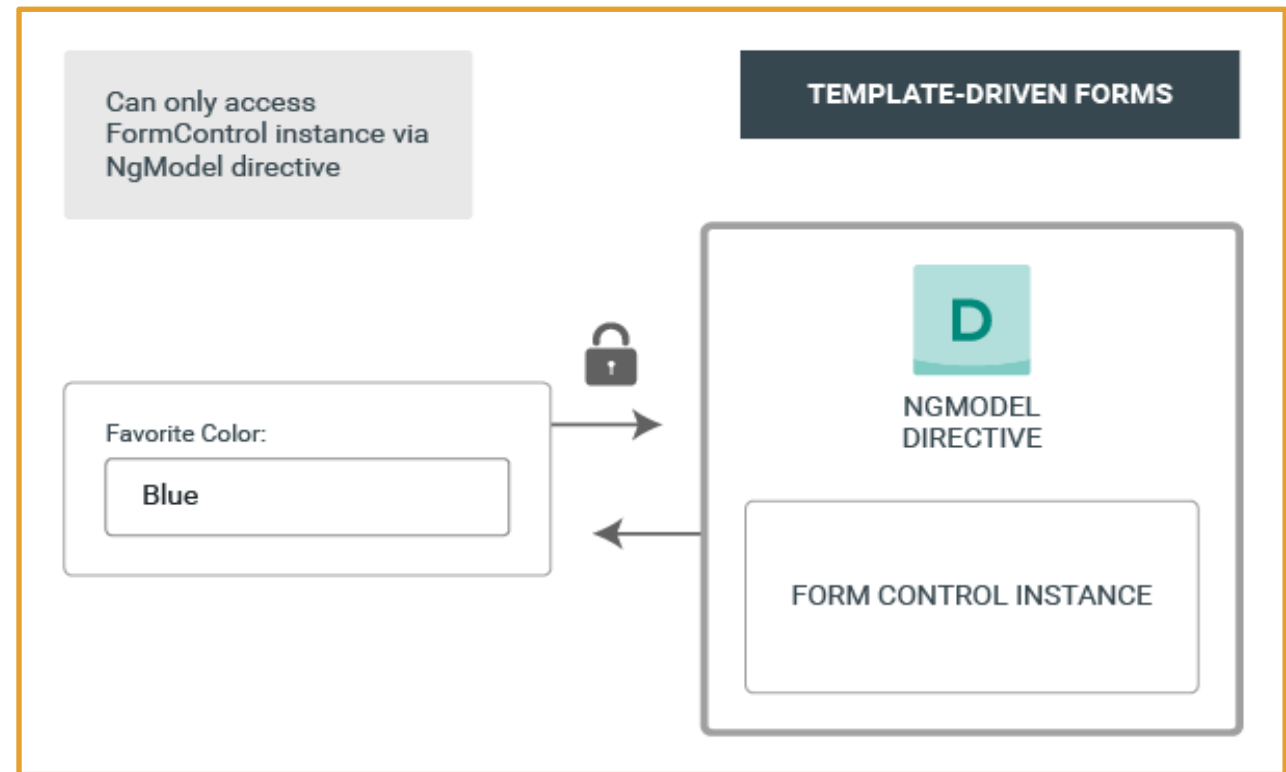
Template Driven Forms

<https://angular.io/start/start-forms#forms-in-angular>
<https://angular.io/api/forms/FormBuilder>
<https://angular.io/guide/forms-overview>

You can build almost any form with an Angular template.

You can lay out the controls creatively, bind them to data, specify validation rules, and display validation errors.

Angular makes the process easy by handling many of the repetitive, boilerplate tasks you'd otherwise code yourself.



Template Driven Forms

<https://angular.io/start/start-forms#forms-in-angular>

<https://angular.io/api/forms/FormBuilder>

<https://angular.io/guide/forms-overview>

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

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


Template-Driven Forms Validation

<https://angular.io/guide/form-validation#validating-input-in-template-driven-forms>

`#name="ngModel"` exports the elements status to the local scope where the `<div>` elements below can use Structural Directives to change their display status.

To prevent the validator from displaying errors before the user has a chance to edit the form, you should first check for either the dirty or touched states in a control.

```
<input type="text" 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">
  <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>
```

Template Driven Forms Setup (1/2)

<https://angular.io/guide/forms#introduction-to-template-driven-forms>

1. Make sure **NgModule** has been imported into **app.module.ts**:
 - `import { NgModule } from '@angular/core';`
2. Create a new class with all the fields that this form will help populate:
 - `ng generate class <className>.`
3. Create a new component with:
 - `ng generate component <componentName>`
4. **import** the Class (from step 2).
5. Create an instance of the class in the component.

```
ng generate class Hero
```

```
export class Player {  
  constructor(Name: string, Wins: number = 0,  
    Losses: number = 0, Id?: number) { }  
}
```

```
ng generate component HeroForm
```

```
import { Player } from '../Player';
```

```
model = new Player('null');
```

Template Driven Forms Setup (2/2)

<https://angular.io/guide/forms#introduction-to-template-driven-forms>

6. Add **@ngModel** to the forms **<input>** elements that correspond to the fields in the Component model (step 4) with:
 - **`[(ngModel)]=model.Name`**
7. If needed, add the **component selector** to a parent view template (.html)
8. Add a check value below your input element text box to see what you are entering live.
9. Enter values into the text box to see the model field value change.

```
<app-login-form></app-login-form>
```

```
<input [(ngModel)]=model.  
</div>  
the name is {{model.Name}}
```

Template Reference Variables

<https://angular.io/guide/template-reference-variables>

The **#myName** in an element in the **.html** file.

```
<input #phone placeholder="phone number" />

<!-- phone refers to the input element; pass
      its `value` to an event handler -->
<button (click)="callPhone(phone.value)">
  Call your muhthah! She misses you!
</button>
```