

# Angular Template Driven vs. Reactive Forms



Lakindu Hewawasam

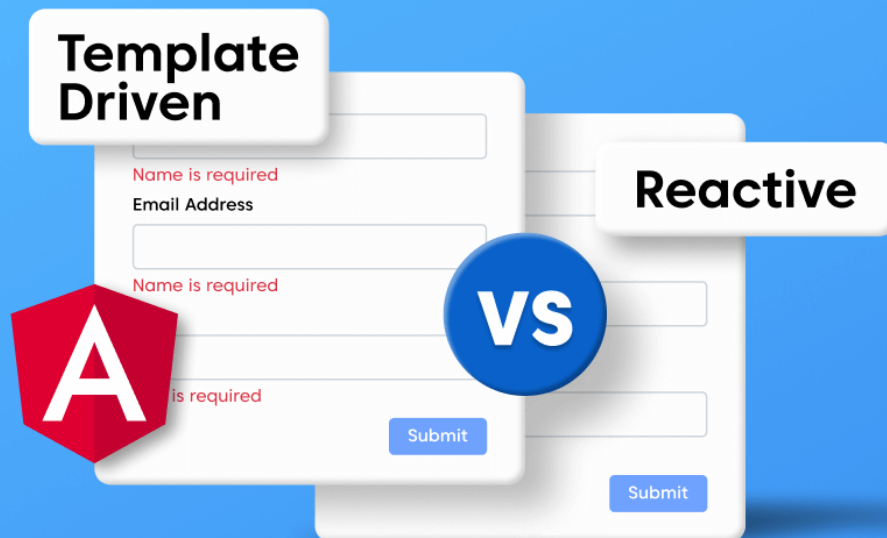


15 min read



Nov 19, 2024

Updated



Almost all enterprise applications are heavily form-driven. Some apps have massive forms that span multiple steps and dialogs and integrate complex validation logic. If developers were to build these complex forms from scratch, they would have to:

- Keep track of the state of the complex, lengthy forms.
- Compute the validity of form sections (with multiple-step forms, this is highly complex).
- Display relevant error messages based on the validation errors. For example, you may have a field with three to five validations on it. For each validation error, you must ensure that the correct error message is displayed, so the user is aware of the error.

These challenges show that building forms for enterprise apps from scratch is inefficient, time-consuming, and error-prone.

Angular provides two modules for creating and managing forms for complex apps. This article will explore the two modules readily available, provide in-depth

comparisons to determine when to use each module, and finally look at a demonstration on managing forms using the two modules.

## Template-Driven Forms in Angular

A template-driven form is the simplest way to build a form in Angular. It uses Angular's two-way data-binding directive (**ngModel**) to create and manage the underlying form instance.

Additionally, as the name suggests, a template form is mainly driven by the view component. So, it uses directives placed in HTML rather than TypeScript or JavaScript to manage the form. A template-driven form is **asynchronous** due to the use of "directives" because the creation of form controls is delegated to the declared directives (IoC).



Syncfusion Angular component suite is the only suite you will ever need to develop an Angular application faster.

Explore Now

## Pros

Using template-driven forms has several advantages:

- Since the entire form gets managed in the view template, it increases the initial simplicity of the form, as you do not need to manage multiple files for one form.
- It is HTML-based. Therefore, you can add validators to your input fields that you are already using with vanilla HTML.

## Cons

However, using template-driven forms has minor drawbacks, too. A few common disadvantages are:

- As the form keeps growing, the view templates get cluttered, making the code difficult to read. Thus, it gets harder to maintain, resulting in a less scalable approach.
- The complex validation code becomes coupled with the template. So, it is difficult to write unit tests for the validation logic. The code becomes less testable and more susceptible to bugs.

- It isn't easy to write complex validation code directly in the template. This makes it challenging to use custom validators for input fields as it requires you to create custom directives for a custom validator.
- Template-driven forms are asynchronous; it takes a second or two to recreate the form. Therefore, accessing the form values in the component class will sometimes result in inaccurate values.

## When to Use a Template-Driven Form

It is essential to know when to use a template-driven form in your Angular app.

You may use a template-driven form if:

- Your form is small to medium scale (around 10 controls with no steps).
- Your form utilizes the validation attributes integrated with HTML without complex validation.
- Your form has basic requirements that can be managed within the template with minimal lines of code.

## Demonstration

This example assumes you have already created a new Angular project with Bootstrap for CSS.



Find the right property to fit your requirement by exploring the complete documentation for Syncfusion's Angular components.

Read Now

## Improving the Modules

To use template-driven forms, Angular requires the **FormsModule** to be imported into your **AppModule** (*app.module.ts*). You can see the import in the following code example.

```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent,
  ],
```

 Copy

```
imports: [  
  BrowserModule,  
  // import the FormsModule to enable Template Driven Forms  
  FormsModule  
],  
providers: [],  
bootstrap: [AppComponent]  
}))  
export class AppModule { }
```

This import will create the directives required for template-driven forms to be available throughout the app.

## Designing the Form

Afterward, declare a form in any view template. I am creating a component (**ng g c TemplateDriven**) to implement the form for demonstration purposes.

In the new component, create a sample form that contains three fields: Name, Email, and Age. The following is the code for the form and its expected output.

```
<div class="row justify-content-sm-center">  
  <div class="col-sm-6">  
    <form>  
      <div class="form-group">
```

[Copy](#)

```
<label>Name</label>
<input class="form-control" placeholder="Provide your name" />
</div>
<div class="form-group">
  <label>Email Address</label>
  <input class="form-control" placeholder="Provide your email address" />
</div>
<div class="form-group">
  <label>Age</label>
  <input class="form-control" placeholder="Provide your age" />
</div>
<div class="form-group">
  <div style="display: flex; margin-top: 10px;">
    <button
      type="submit"
      class="btn btn-primary"
      style="margin-left: auto; width: 30%;"
    >
      Submit
    </button>
  </div>
</div>
</form></div>
</div>
```



Template Driven Form

Name

Email Address

Age

Submit

Expected output for initial form

## Adding the Template-Driven Directives to the Form

As shown in the previous code snippet, the fields get wrapped in the `<form>` `</form>` tags. With the help of the `FormsModule`, bind Angular-specific directives to the form to create the template-driven form.

Then, define the structure of the form by declaring its controls. Ensure that the form outputs an object containing the keys as the field name and the value entered.

Place the directive `ngModel` (provided by `FormsModule`) on the input field that you want to mark as a control. Add the HTML `name` attribute to define the the key that the form uses for the data model. Refer to the following code.



```
<!-- placing the ngModel directive and the NAME attribute (for data structure de
<input class="form-control" placeholder="Provide your name" ngModel name="name"
<input class="form-control" placeholder="Provide your email address" ngModel nam
<input class="form-control" placeholder="Provide your age" ngModel name="age"/>
```

By doing so, Angular will register key-value pairs and create the JavaScript representation of the form (data model).

## Adding Validations to the Fields

Make sure that your input fields get adequately validated on the front end.

Template-driven forms allow you to add validations easily by binding the native HTML validators to the input fields.

For our form, let's make all the fields as required and add property input types as shown in the following code.



```
<!-- Adding the HTML Validators to the Form Controls -->
<input class="form-control" placeholder="Provide your name" ngModel name="name"
<input class="form-control" placeholder="Provide your email address" ngModel nam
<input class="form-control" placeholder="Provide your age" ngModel name="age" re
```



Be amazed exploring what kind of application you can develop using Syncfusion Angular components.

Try Now

## Submitting the Form and Computing Validation Errors

To submit the form, we require two things:

- A submit event.
- Access to the data on the form.

In Angular, the **ngSubmit** directive declares the submit event for the form.

Therefore, when the Submit button is clicked, the event handler for **ngSubmit()** is executed. The callback for the event is declared in the component class. For the event, we have to pass the form values. To do this, create a local reference to the form of type **ngForm**. Refer to the following code.

### Template



```
<form #templateDrivenForm="ngForm" (ngSubmit)="handleFormSubmit(templateDrivenFo
</form>
```

## Component Class



```
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-template-driven',
  templateUrl: './template-driven.component.html',
  styleUrls: ['./template-driven.component.css']
})
export class TemplateDrivenComponent implements OnInit {
  constructor() { }
  ngOnInit(): void {
  }
  handleFormSubmit(form: NgForm): void {
    // value will print the JavaScript Object of the Form Values.
    console.log(form.value);
  }
}
```

Once we submit the form, you can see the form values logged in the console.

Template Driven Form

Name

Lakindu Hewawasam

Email Address

lakindu.development@gmail.com

Age

19

Submit

Elements

Console

Recorder

Sources

1

Filter

Default levels

1 issue

1

template-driven.component.ts:18

```

{name: "Lakindu Hewawasam", emailAddress: "lakindu.development@gmail.com",
age: 19}
age: 19
emailAddress: "lakindu.development@gmail.com"
name: "Lakindu Hewawasam"
[[Prototype]]: Object

```

Displaying the submitted result

## Displaying Validation Errors

To add validation messages, use the local reference, check for individual control errors, and disable the submit action if the form has an error present.

The output shows the final form template and validation messages.

Copy

```

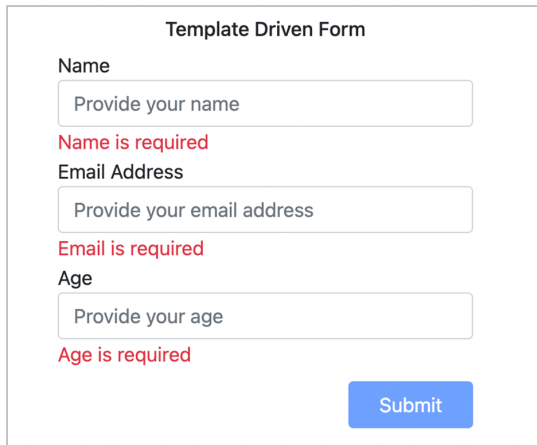
<div class="row justify-content-sm-center"><div class="col-sm-6"><form #template
  Name is required
</span></div>

<div class="form-group"><label>Email Address</label><input class="form-control"
  Email is required
</span></div>

<div class="form-group"><label>Age</label><input class="form-control" placeholder="Age"
  Age is required
</span></div>

```

```
<div class="form-group"><div style="display: flex; margin-top: 10px;"><buttontyp  
[disabled]="templateDrivenForm.invalid"  
  
Submit  
</button></div></div></form></div>  
</div>
```



Template Driven Form

Name  
Provide your name  
Name is required

Email Address  
Provide your email address  
Email is required

Age  
Provide your age  
Age is required

Submit

Completed template-driven form

## Reactive Forms in Angular

The second way to develop forms in Angular is to use reactive forms. Reactive forms utilize the component class to programmatically declare the form controls and the required validators synchronously.

We then bind the controls to the input fields in the HTML template. Reactive forms use classes such as **FormGroup** and **FormControl** to define groups and controls wired to the template.

## Pros

Reactive forms advantages:

- The business logic (validations) and control declaration are isolated from the view template. The view template is solely responsible for the look and feel of the form. This increases scalability, readability, and maintainability.
- As the validation logic gets declared in the component class, unit tests are easily written for the logic to ensure the code is bug-free.
- Reactive forms take away the complexity of writing custom validators. They allow you to define custom validator functions and bind them to the form during declaration.
- Due to its synchronous nature, developers can access and update any form control (child or parent) readily available in the code.

## Cons

Reactive forms drawbacks:

- They are complex to use for first-time developers. The Reactive Forms Module requires an in-depth understanding of the API, creating a high learning curve.
- There is no way to disable inputs using the **disabled** attribute on an HTML control unless we manually disable it using the **FormControl** programmatically.

## When to Use a Reactive Form

If the learning time and added use of the API are not a bother to you, you should consider using a reactive form in these cases:

- If your form has a lot of controls with many groups (children) that require complex validation logic.
- If you have validation code reused in multiple forms across your app. Reactive forms will help reuse the validation logic to avoid code duplication, thus making your code scalable.



- If you want your form data model to be immutable. Reactive forms return a new state when a change occurs in the form.
- To include unit tests for your validation code. As the validators are decoupled from the template, it increases its testability.

## Demonstration

If you want to incorporate reactive forms in your app, let's look at an example to help you get started. This example assumes you already created a new Angular project with Bootstrap for CSS.

## Importing the Module

Angular requires the **ReactiveFormsModule** to enable reactive forms for an Angular app. We can add it to the **AppModule** (*app.module.ts*) by importing the **ReactiveFormsModule**.

```
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
```



```
    AppComponent,  
  ],  
  imports: [  
    BrowserModule,  
    // Import the ReactiveFormsModule to use Reactive Forms in the application.  
    ReactiveFormsModule  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
export class AppModule { }
```

This import ensures that the classes required to construct reactive forms are available throughout the app.

## Defining the Form Controls

The controls are defined in the Component class in a reactive form, while the layout is defined in the template. Let's convert the template-driven form we created earlier to a reactive form. To do so, we use a new component (**ng g c ReactiveForms**). The Reactive Forms Module offers the **FormGroup** and **FormControl** classes to build the form. Additionally, it provides built-in validator functions to test for required fields, regex patterns, and more.

You can see the sample form structure with validators and the submit callback in the following code.



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

@Component({
  selector: 'app-reactive-form',
  templateUrl: './reactive-form.component.html',
  styleUrls: ['./reactive-form.component.css']
})
export class ReactiveFormComponent implements OnInit {
  sampleForm!: FormGroup;
  ngOnInit(): void {
    // create an instance of form group// for the object passed, the key identifies the form group
    this.sampleForm = new FormGroup({ 'name': new FormControl('', [Validators.required]),
    'emailAddress': new FormControl('', [Validators.required, Validators.email])
    'age': new FormControl('', [Validators.required, Validators.pattern('^[0-9])
```

## Configuring the Template



After configuring the structure, validators, and submit events, define the template

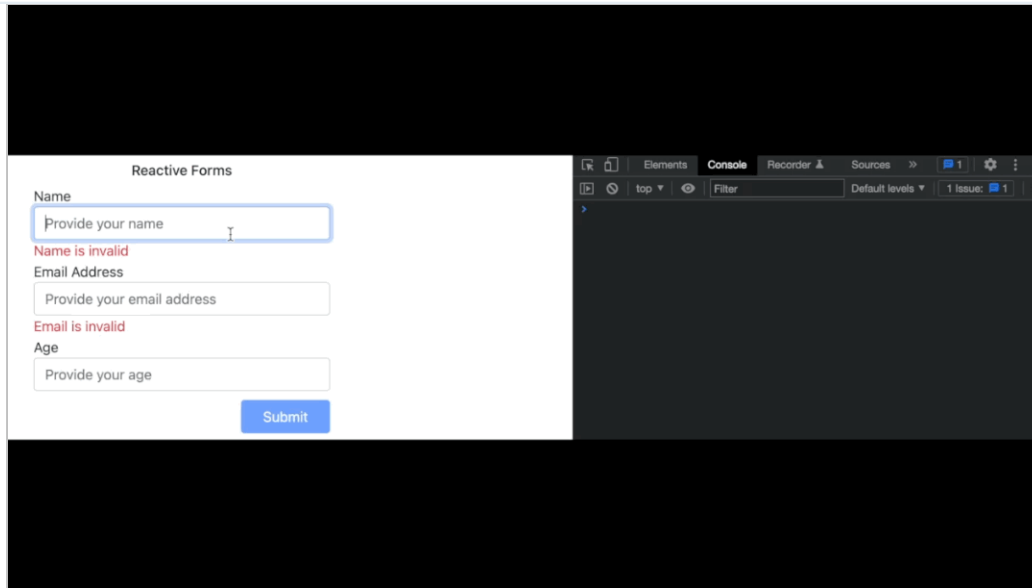


```
<div class="row justify-content-sm-center"><div class="col-sm-6"><form [formGroup]
  Name is invalid
</span></div><div class="form-group"><label>Email Address</label><input cl
Email is invalid
</span></div><div class="form-group"><label>Age</label><input class="form-
Age is invalid
</span></div><div class="form-group"><div style="display: flex; margin-top
  [disabled]="sampleForm.invalid"
>
Submit
</button></div></div></form></div>
</div>
```

This code binds the initialized form as the form group on the **<form>** tag and binds the controls by the **formControlName** directive. The value placed here is the value declared during initialization.

## Output

After adding the code, the reactive form should function the same as before, but this time, with cleaner and scalable code.



Reactive form in action

## Resource

The code used in this article is accessible in this [GitHub repository](#).



Harness the power of Syncfusion's feature-rich and powerful Angular UI components.

[Try It Free](#)

## Conclusion

Angular takes away the added complexity of building forms by providing two modules: template-driven and reactive forms. There is no best way to create a form. You can use either of the two modules based on your use case.

I recommend using a template-driven form to build simple forms with minimal validation. But if you require granular control of input fields with asynchronous and complex validators, reactive forms are the ones for you.

I hope you found this article helpful.

Thank you for reading.

Syncfusion's [Angular UI component](#) library is the only suite you will ever need to build an app. It contains over 65 high-performance, lightweight, modular, and responsive UI components in a single package.

For existing customers, the newest Essential Studio® version is available for download from the [License and Downloads](#) page. If you are not yet a Syncfusion

customer, you can try our 30-day [free trial](#) to check out the available features. Also, check out our demos on [GitHub](#).

For questions, you can contact us through our [support forums](#), [support portal](#), or [feedback portal](#). We are always happy to assist you!

## Related blogs

- [Implementing Route Protection in Angular using CanActivate](#)
- [Efficiently Bind Data to the Angular Data Grid and Perform CRUD Operations Using GraphQL](#)
- [Easy Steps to Host an Angular App in GitHub Pages](#)
- [Top 10 Angular Component Libraries for 2022](#)



MEET THE AUTHOR

## Lakindu Hewawasam

I am a Software Engineer with a passion for building great applications. I love experimenting with new technologies and putting it to good use by building products that all of us can use everyday.