When we want to start developing any web-based application, we always find that a large number of UI or interfaces are very much form dependent. This is very much true for any enterprise-type application also, since most of the interfaces of that application are a large form which contains multiple tabs, dialogs, grid, etc. And also these types of forms always contain non-trivial business validation logic. Since in a component-based framework, we need to split a form into multiple small and reusable pieces of code. These components can be used across the entire application which basically provides many architectural benefits including flexibility and design changes.

Angular 8 deals with the forms of objects through ngModel. The instance two-way data binding technique of ng-model in the Angular framework is really a life-saver, because it provides automatic sync between the form and the view model objects. For using the forms module in Angular, we need to inject the FormModule in our application module.

Types of Angular Forms

As a modern full-fledged UI Framework, Angular has its own full-fledged libraries for developing complex form-based UI. The current Angular framework has two types of form-building strategies as,

1. Template Driven Form
2. Model Drive Form or Reactive Form

Both the above technology belongs to the @angular/forms packages and are totally based on the form-control classes. But in spite of that, both the techniques are different from each other in respect to their own philosophy, programming style, and technique.

In the below section, we will discuss in detail items related to the above two types of Angular Forms.

Template Driven Forms

In Angular Framework, Template-driven are those forms where we can write logic, validations, controls, etc. in the HTML template part of the component. Basically, the Template is totally responsible for setting the form elements in the UI, for implementing the validation using form control, etc. With the help of Template, we can also provide the Form Group within the HTML template. Template-driven forms are perfect for the simple scenario-based interface where we can easily use the two-way data binding of Angular. For Template-driven form, Angular provides some form-specific directives which we can use to bind the form input data without model variable. Due to this form-specific directive, we can add extra functionality and behavior to a plain HTML form. In the end, the Template itself takes care of binding the values with the model and the form validation.

Benefits of Template Driven Forms

In Angular Framework, we can obtain some benefit for using Template-Driven form as below,

* It is much easier to use
* This technique works perfectly in simple scenarios
* It totally depends on two-way data binding techniques i.e. ngModel syntax.
* It requires a minimum of code in the component part since most of the work is done in the HTML template part.
* It automatically tracks the form element and its control.
* Despite the above benefits, it has some drawbacks like –
* Template-driven form techniques fail when we want to design some complex form in the UI section
* We can’t perform any Unit Testing based on the Template Driven Form.

Model-Driven Forms

Model-Driven form technique is often called a Reactive Form technique in Angular. It provides a model-driven approach to the UI to handling form inputs towards the component variable. Using this technique, we can create and update a simple form of control, progress to using multiple controls within a form group to validate the form control values, etc.

Model-driven or Reactive forms normally use an explicit and immutable approach to manage the state of a form control at any given point of time. If any change occurrs in the form control, then it will return a new state through which Angular can maintain the integrity of the model variables between changes. Basically, reactive forms are normally based on observable streams. So, every form's input controls and values provide the input value as a stream which is accessed by the reactive form synchronously. With the help of reactive forms, we can perform a straightforward process to perform unit testing on the form controls. This can be done very easily since we can be assured that form data is consistent and predictable whenever requested.

Reactive forms or Model-driven forms are different compared to template-driven forms. Reactive forms always provide more predictability with synchronous access towards the data model variable, provide immutability with observables operators and also keep track of changes using observable streams.

Benefits of Model-Driven Forms

In Angular Framework, we can obtain some benefit for using Model-Driven form or Reactive Form techniques as below,

* In Reactive Forms, form definition including logic related coding mainly maintained within the TypeScript part of the component. Since using this technique, we create form controls programmatically using FormGroup or FormBuilder class. In HTML template, HTML form tags are only used to put a reference of TypeScript based form-control class.
* It provides us programmatic and full control of the form value updates and form validations.
* In this technique, we can create a dynamic structure-based form at run time.
* We can implement custom form validation.
* Since the entire form-based part is in typescript class or component, it is much easier to write unit tests in reactive forms.
* Despite the above benefits, it has some drawback like –
* This technique requires much more coding, especially in the TypeScript part.
* It is a little bit complex to understand and maintain the code.

Template-Driven Form vs Reactive Form

The comparison between Template-Driven Form and Reactive Forms (Model-Driven Forms) are as below,

|  |  |
| --- | --- |
| Template-Driven Form | Reactive Form |
| Template-Driven Form is less explicit, and it is mainly created by Directives. | Reactive Form is more explicit and normally created within the Component class. |
| It supports the unstructured data model | It always supports the structured data model. |
| It uses directives for implementing Form validations | It uses the function for implementing Form Validations |
| When form control value changes, it provides an asynchronous mechanism to update form controls. | When form control value changes, it provides synchronous mechanism to update form controls. |

Form Controls

FormControl class is mainly used to assign any form related fields in the Angular component. This class is also used in the FormBuilder class method. It is used mainly for ease of access. With the help of references of the FormControl in place of FormControl class, we can gain access to the inputs in the template without using the Form itself. Similarly, we can use any instance of FormControl to access its parent group by using its root property. When we define a form of control, it requires two properties: an initial value and a list of validators.

Reactive Form Validation

Angular Framework provides many validators to validate the form control input values in any application. These validations can be imported along with the related dependencies for procedural forms. In general, the common practice for using Form validations are using .valid and .untouched to determine if we need to raise an error message. As an inbuilt validator, we can use hasError() method on the form element to validate the data.

Reactive Form Custom Validation

As per the built-in validators, it is very useful if we can create our own custom validator for our purpose. Angular Framework always allows us to do just that, with minimal effort. A simple function takes the FormControl instance and returns null if everything is fine. If the test fails, it returns an object with an arbitrarily named property. In this case, we need to use the property name .hasError() for the test.

1. <div [hidden]="!password.hasError('hasSpecialChars')">
2. Your password must have Special Characters like @,#,$, etc!
3. </div>

Demo 1 - Template Driven Form

Now, in this demo, we will demonstrate how to define a template-driven form in Angular.

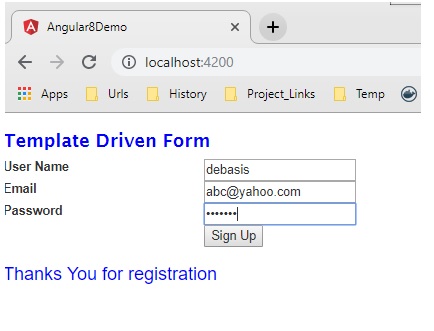
**app.component.ts**

1. **import** { Component, OnInit } from '@angular/core';
2. **import** { NgForm } from '@angular/forms';
4. @Component({
5. selector: 'app-root',
6. templateUrl: './app.component.html',
7. styleUrls : ['./custom.css']
8. })
9. **export** **class** AppComponent **implements** OnInit {
10. **private** formData: any = {};
11. **private** showMessage: **boolean** = **false**;
13. constructor() {
14. }
16. ngOnInit(): **void** {
17. }
19. registerUser(formdata: NgForm) {
20. **this**.formData = formdata.value;
21. **this**.showMessage = **true**;
22. }
23. }

**app.component.html**

1. <h2>Template Driven Form</h2>
2. <div>
3. <form #signupForm="ngForm" (ngSubmit)="registerUser(signupForm)">
4. <table style="width:60%;" cellpadding="5" cellspacing="5">
5. <tr>
6. <td style="width :40%;">
7. <label **for**="username">User Name</label>
8. </td>
9. <td style="width :60%;">
10. <input type="text" name="username" id="username" [(ngModel)]="username" required>
11. </td>
12. </tr>
13. <tr>
14. <td style="width :40%;">
15. <label **for**="email">Email</label>
16. </td>
17. <td style="width :60%;">
18. <input type="text" name="email" id="email" [(ngModel)]="email" required>
19. </td>
20. </tr>
21. <tr>
22. <td style="width :40%;">
23. <label **for**="password">Password</label>
24. </td>
25. <td style="width :60%;">
26. <input type="password" name="password" id="password" [(ngModel)]="password" required>
27. </td>
28. </tr>
29. <tr>
30. <td style="width :40%;"></td>
31. <td style="width :60%;">
32. <button type="submit">Sign Up</button>
33. </td>
34. </tr>
35. </table>
36. </form>
37. <div \*ngIf="showMessage">
38. <h3>Thanks You {{formData.username}} **for** registration</h3>
39. </div>
40. </div>

Now check the output in the browser,



Demo 2 - Model-Driven Form

Now in this demo, we will discuss how to develop a Reactive Form in Angular 8 Framework. For that, we first need to develop a login form component as below –

**app.component.ts**

1. **import** { Component, OnInit, ViewChild } from '@angular/core';
2. **import** { Validators, FormBuilder, FormControl, FormGroup  } from '@angular/forms';
4. @Component({
5. selector: 'app-root',
6. templateUrl: './app.component.html',
7. styleUrls : ['./custom.css']
8. })
9. **export** **class** AppComponent **implements** OnInit {
10. **private** formData: any = {};
12. username = **new** FormControl('', [
13. Validators.required,
14. Validators.minLength(5)
15. ]);
17. password = **new** FormControl('', [
18. Validators.required,
19. hasExclamationMark
20. ]);
22. loginForm: FormGroup = **this**.builder.group({
23. username: **this**.username,
24. password: **this**.password
25. });
27. **private** showMessage: **boolean** = **false**;
29. constructor(**private** builder: FormBuilder) {
30. }
32. ngOnInit(): **void** {
33. }
35. registerUser() {
36. **this**.formData = **this**.loginForm.value;
37. **this**.showMessage = **true**;
38. }
39. }
41. **function** hasExclamationMark(input: FormControl) {
42. **const** hasExclamation = input.value.indexOf('!') >= 0;
43. **return** hasExclamation ? **null** : { needsExclamation: **true** };
44. }

**app.component.html**

1. <h2>Reactive Form Module</h2>
2. <div>
3. <form [formGroup]="loginForm" (ngSubmit)="registerUser()">
4. <table style="width:60%;" cellpadding="5" cellspacing="5">
5. <tr>
6. <td style="width :40%;">
7. <label **for**="username">User Name</label>
8. </td>
9. <td style="width :60%;">
10. <input type="text" name="username" id="username" [formControl]="username">
11. <div [hidden]="username.valid || username.untouched" **class**="error">
12. <div [hidden]="!username.hasError('minlength')">
13. Username can not be shorter than 5 characters.
14. </div>
15. <div [hidden]="!username.hasError('required')">
16. Username is required.
17. </div>
18. </div>
19. </td>
20. </tr>
21. <tr>
22. <td style="width :40%;">
23. <label **for**="password">Password</label>
24. </td>
25. <td style="width :60%;">
26. <input type="password" name="password" id="password" [formControl]="password">
27. <div [hidden]="password.valid || password.untouched" **class**="error">
28. <div [hidden]="!password.hasError('required')">
29. The password is required.
30. </div>
31. <div [hidden]="!password.hasError('needsExclamation')">
32. Your password must have an exclamation mark!
33. </div>
34. </div>
35. </td>
36. </tr>
37. <tr>
38. <td style="width :40%;"></td>
39. <td style="width :60%;">
40. <button type="submit" [disabled]="!loginForm.valid">Log In</button>
41. </td>
42. </tr>
43. </table>
44. </form>
45. <div \*ngIf="showMessage">
46. <h3>Thanks You {{formData.username}} **for** registration</h3>
47. </div>
48. </div>

Now, for using the reactive form we need to inject ReactiveFormModule in our app.module.ts file as below –

1. **import** { BrowserModule } from '@angular/platform-browser';
2. **import** { NgModule, NO\_ERRORS\_SCHEMA } from '@angular/core';
3. **import** { FormsModule, ReactiveFormsModule } from '@angular/forms';
5. **import** { AppComponent } from './app.component';
7. @NgModule({
8. declarations: [
9. AppComponent
10. ],
11. imports: [
12. BrowserModule, FormsModule, ReactiveFormsModule
13. ],
14. providers: [],
15. bootstrap: [AppComponent],
16. schemas: [NO\_ERRORS\_SCHEMA]
17. })
18. **export** **class** AppModule { }

Now check the output in the browser,

