Angular 11 (part 3)

By

Anand Kulkarni

anand.kulkarni1@zensar.com

Contents

Building UI Forms

Angular provides us 3 ways to create forms:

1) Template Driven Forms
2) Model Driven Forms (MDF)
3) MDF using FormBuilder

To work on UI forms, we import respective module in app.module.ts:

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

imports: [BrowserModule, FormsModule] *// For 'Template Driven Forms'*

imports: [BrowserModule, ReactiveFormsModule] *// For 'Model Driven Forms'*

Template Driven Forms

In template driven forms, you can build forms by writing templates in the Angular template syntax with the form-specific directives.

```
<form #productForm="ngForm" (ngSubmit)="onSubmit(productForm.value)">
  <div class="form-group">
                    <input type="text" name="name" class="form-control"
placeholder="Name" ngModel>
  </div>   <div class="form-group">
                    <input type="text" name="quantity" class="form-control"
placeholder="Quantity" ngModel>
  </div>
<button type="submit" class="btn btn-primary">ADD PRODUCT</button>
</form>
```

Validation on Template Driven Forms

Angular applies suitable validation class depending upon the state of a form control.

Sr. No.	Form control's state	Class if true	Class if false
1.	Form control has been visited	ng-touched	ng-untouched
2.	Form control's value have been changed	ng-dirty	ng-pristine
3.	Form control's value is valid	ng-valid	ng-invalid

Validation of Template Driven Form

We can implement template driven form validation using different validation classes like touched, untouched, dirty, pristine, valid, invalid etc. Below is the code snippet to display error message in case of invalid input.

```
<div class="form-group col-md-3">
        <label>Name</label>
        <input type="text" name="name" required class="form-control" #nameRef
                        #nameModelRef="ngModel" ngModel>
        <div [hidden]="nameModelRef.valid || nameModelRef.pristine" class="alert
alert-danger">
                Please enter the name
        </div>
</div>
```

Handling multiple errors with Template Driven Form

Sometimes an HTML element has multiple validations attached. For example, required, minlength, maxlength, pattern matching etc. To handle such element, we have written error handling code below:

```
<input type="text" name="name" required minlength="3" class="form-control"
#nameRef #nameModelRef="ngModel" ngModel>
<div *ngIf="nameModelRef.errors && nameModelRef.touched" class="alert alert-
danger">
        <div [hidden]="!nameModelRef.errors.required" >     Please enter
your name     </div>
        <div [hidden]="!nameModelRef.errors.minlength"> Name length
should be minimum 3                characters </div>
</div>
```

Submitting a valid template driven form

If you want Angular to take care of your form validations then first you need to disable HTML 5 validation messages using ‘novalidate’ attribute as follows:

```
<form #productForm="ngForm" (ngsubmit)="onSubmit(productForm.value)”
novalidate>
```

The second step is to disable form ‘Submit’ button until user enters all field values as valid:

```
<button [disabled]=“!productForm.form.valid” type=“submit” class="btn btn-
primary">Submit</button>
```

Model Driven Forms (MDF)

- In model driven forms, a form is represented by a model and this model is responsible to handle all user interactions with the form.
- We create a model using 2 classes:
  1. ***FormControl:*** Every HTML element can be represented as an instance of FormControl class.
  2. ***FormGroup:*** FormGroup is a collection of FormControl objects. Note that every HTML form is a valid instance of FormGroup class.
- Steps to build a form using model driven approach:
  1. Generate a form structure in component class using FormControl & FormGroup classes.
  2. Create a simple HTML form matching the structure of component class.
  3. Bind HTML form with form structure described into component class.

MDF Step 1: Generate a form structure

```
@Component({
	selector: '<product-mdf-form></product-mdf-form>',
	templateUrl: './product-mdf-form.html'
})
export class ProductMdfFormComponent {
	productForm = new FormGroup({
		name: new FormControl(),
		quantity: new FormControl(50),
		price: new FormControl()
	});

	onSubmit() {
			console.log(this.productForm.value);
	}
}
```

MDF Step 2 & 3: Create & bind the HTML form

```
<form [formGroup]="productForm" (ngSubmit)="onSubmit()">
  <div class="form-group row">
      <label>Name</label>
      input type="text" formControlName="name" class="form-control">
  </div>
  <div class="form-group row">
      <label>Quantity</label>
      <input type="text" formControlName="quantity" class="form-control">
  </div>
  <div class="form-group row">
      <label>Price</label>
      <input type="text" formControlName="price" class="form-control">
  </div>
  <button   type="submit" class="btn btn-primary">ADD PRODUCT</button>
</form>
```

MDF Validation

1. Angular provides a class called ‘Validator’ to validate an MDF form.
2. Every FormControl object constructor takes second argument of validator array itself. For example:

name = new FormControl(null, [Validators.required, Validators.minlength(3)]);

Component class in MDF Validation

```
@Component({
   selector: '<product-mdf-form></product-mdf-form>',
   templateUrl: './product-mdf-form.html',
   styles: [`input.ng-invalid {border-left: 5px solid red}   input.ng-valid {border-left: 
5px solid green}`]
})
export class ProductMdfFormComponent {
   productForm = new FormGroup({
      name: new FormControl(null, [Validators.required, Validators.minLength(3), 
Validators.maxLength(8)]),
      quantity: new FormControl(50),    price: new FormControl()
   }); }
```

HTML form in MDF Validation

```
<div class="col-md-3">
    <label>Name</label>
    <input type="text" formControlName="name" class="form-control">
    <div *ngIf="productForm.controls['name'].hasError('required')" class="alert alert-
    danger">
                    Please enter product name
    </div>
    <div *ngIf="productForm.controls['name'].hasError('minlength')" class="alert
    alert-danger">
                    Please enter atleast 3 characters
    </div>
    <div *ngIf="productForm.controls['name'].hasError('maxlength')" class="alert
    alert-danger">
                    Name cannot exceed 8 characters
    </div>
</div>
```

MDF with FormBuilder

- If your form has dozens of fields then generating form structure using FormControl class will make your code messy. To avoid this, we use FormBuilder.
- FormBuilder is just a syntactical improvement in creating MDF.
- In order to introduce FormBuilder, you only need to change your component class. The HTML form will remain unchanged i.e. it is same as we described in traditional MDF.

Using FormBuilder in Component

```
import { ReactiveFormsModule, FormGroup, FormBuilder, Validators } from
'@angular/forms';
@Component({
selector: '<product-formbuilder-form></product-formbuilder-form>',
templateUrl: './product-mdf-form.html',
styles: [`input.ng-invalid {border-left: 5px solid red}   input.ng-valid {border-left: 5px
solid green}`]
})
export class ProductFormBuilderComponent {
      productForm: FormGroup;
    constructor(private formBuilder: FormBuilder) {
            this.productForm = this.formBuilder.group({
            name: ['Anand', [Validators.required, Validators.minLength(3),
            Validators.maxLength(8)]],
            quantity: [], price: [] });
    }
}
```

Services

- Angular components mainly takes care of view. Sometimes, we need a component that focuses on only business logic without view associated. Such a component is called as ‘Service’.
- Once service is defined, it can be accessed by several components.
- Service is a singletone class i.e. Angular creates a single instance of your service class.

Introducing a Service

There are 3 steps to introduce a service in an angular application:

1. Create a service class.
2. Register a service.
3. Declare service dependency.

Creating a Service

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

@Injectable()
export class ProductService {
        products = [
                {id: 1, name: 'Chair', price: 3000, quantity: 25},
                {id: 1, name: 'Pen', price: 200, quantity: 60}
        ];

        getProducts() {
                return this.products;
        }
}
```

Registration & declaring a Service

```
import { ProductService } from './products.service';

@Component({
            selector: 'sample',
            templateUrl: './sample.html',
            providers: [ProductService]                        //Service registration
})
export class SampleComponent {
            constructor(productService: ProductService) {          //Declaring service
dependency
                        console.log("Products = ", productService.getProducts());
            }
}
```

When a service is registered into any component, all its children components can access this service.

Thank you!!