**Angular Forms**

* Form is a container that comprises of set of elements, which allow interaction with our application.
* Form provides an UI from where user can input, edit, delete, view data.
* HTML form comprises of elements like button, textbox, checkbox, radio, listbox etc.
* Angular makes the static HTML form into dynamic.
* HTML presents the form and Angular makes it interactive to handle client-side interactions.
* Based on where Angular is handling interactions the forms in angular are classified into 2 types
  + **Template Driven Forms**
  + **Model Driven Form / Reactive Form**

**Template Driven Forms**

* A template driven form configures and handles all interactions at View Level (HTML)
* Configuration of a form and its manipulation both handled in HTML template.
* Very optimized controller level interaction. All interactions are at view level.
* It reduces the number of requests to a component.
* It improves the page load time.
* It is good for forms designed in “in-line” technique.
* Template drive form is heavy on page. Slow in handling interactions and rendering.
* Hard to test and extend the form.
* Separation issues. Not loosely coupled.
* You can use template driven forms when you are designing an UI that doesn’t require regular extensions.
* The directives that are used to configure “Form and Form Elements” in template driven approach.
  + NgForm
  + NgModel
* **NgForm:** It provides a set of properties and methods that are used to configure and handle <form> element.
* **NgModel:** It provides a set of properties and methods that are used to configure and handle a form control like button, textbox, checkbox, radio, dropdown list, etc.
* The library for “NgForm and NgModel” is “@angular/forms”
* The module is “FormsModule”

**Configuring Form:**

<form #formName=”ngForm”>

</form>

* NgForm provides set of attributes
  + Value
  + pristine
  + dirty
  + valid
  + invalid
  + etc..

Syntax:

formName.value

formName.pristine

formName.dirty

**Configuring a Form Element**

* “NgModel” is used to make a static form control into dynamic.

<input type=”text” ngModel #txtName=”ngModel” name=”txtName”>

txtName.value

txtName.valid

**Key Note:**

* A Form Reference must implement “NgForm” to handle the form behaviour.   
  **<form #frmRegister=”ngForm”>**  [**ngForm is of type NgForm**]
* NgForm makes the form dynamic.
* NgForm is a member of “**FormsModule**” in “@angular/forms”.
* Form can’t access and submit data of any control without a **Name**. Every control defined in a form must have a **“name”** defined.  
  **<input type=”text” name=”txtName”>**
* Angular can’t recognize any form element dynamically. The static form element must transform into dynamic form element.
* “**NgModel”** is a directive that makes the Static Form element into Dynamic.  
  **<input type=”text” name=”txtName” ngModel>**
* Every dynamic element must have a reference name, which is used as a model name to store its value dynamically.
* Every form element must implement “NgModel”.
* Every control is configured “ngModel”. So that it can have a reference name dynamically.  
  <input type=”text” name=”txtName” ngModel #txtName=”ngModel”>

**Form related properties are derived from “NgForm”**

**Control related properties are derived from “NgModel”**

**Accessing the values from Template Form:**

**Accessing all Values**

* You can use “value” property of “NgForm”.
* The “value” property returns an object that contains collection of key value pairs.
* The key/value pairs refer to Control Name and the Control Value.
* All form control and their values can be accessed by using “formName.value” property.

**Syntax:**  
frmRegister.value 🡪 Returns collection of all form elements and their values as an object

**Accessing any specific value**

* You can access the value of any control by using “value” property of “NgModel”

**Syntax:**

txtName.value 🡪 Returns the value of any specific element.

Ex:

* **Templateform.component.ts**

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

@Component({

selector: 'app-templateform',

templateUrl: './templateform.component.html',

styleUrls: ['./templateform.component.css']

})

export class TemplateformComponent {

public product;

public onFormSubmit(obj){

this.product = obj;

alert(this.product.txtName);

}

}

* **Templateform.component.html**

<div class="container-fluid">

<div class="row">

<div class="col-3">

<h2>Register Product</h2>

<form #frmRegister="ngForm" (submit)="onFormSubmit(frmRegister.value)" >

<div class="form-group">

<label>Name</label>

<div>

<input ngModel #txtName="ngModel" name="txtName" type="text" class="form-control">

</div>

</div>

<div class="form-group">

<label>Price</label>

<div>

<input ngModel #txtPrice="ngModel" name="txtPrice" type="text" class="form-control">

</div>

</div>

<div class="form-group">

<label>Quantity</label>

<div>

<input ngModel #txtQty="ngModel" name="txtQty" type="text" class="form-control">

</div>

</div>

<div class="form-group">

<label>Shipped To</label>

<div>

<select ngModel #lstShippedTo="ngModel" name="lstShippedTo" class="form-control">

<option>Delhi</option>

<option>Hyderabad</option>

</select>

</div>

</div>

<div class="form-group">

<label>In Stock</label>

<div>

<input ngModel #optStock="ngModel" name="optStock" type="checkbox"> Yes

</div>

</div>

<div class="form-group">

<button class="btn btn-primary btn-block">Register</button>

</div>

</form>

</div>

<div class="col-4">

<h2>Product Object</h2>

<pre>

{{frmRegister.value | json}}

</pre>

</div>

<div class="col-5">

<h2>Product Details</h2>

<dl>

<dt>Name</dt>

<dd>{{frmRegister.value.txtName}}</dd>

<dt>Price</dt>

<dd>{{txtPrice.value}}</dd>

<dt>Qty</dt>

<dd>{{txtQty.value}}</dd>

<dt>Total</dt>

<dd>{{txtQty.value \* txtPrice.value}}</dd>

<dt>Shipped To</dt>

<dd>{{lstShippedTo.value}}</dd>

<dt>Stock Status</dt>

<dd>{{(optStock.value==true)?"Available":"Out of Stock"}}</dd>

</dl>

</div>

</div>

</div>

**Validation in Template Driven Forms**

* Validation is the process of verifying user input.
* Validation is required to ensure that contradictory and un-authorized data is not stored into the data source.
* Angular can handle validations client side by using a set of validation services.
* Angular validation services are categorized into 2 groups
  + Form State Validation Services
  + Input State Validation Services

**Form State Validation Services**

* Form state validation services verifies all fields in the form simultaneously at the same time.
* Angular verifies all fields in the forms before submitting and report errors.

|  |  |  |
| --- | --- | --- |
| **Service Name** | **Property** | **Description** |
| NgPristine | pristine | * It returns Boolean true when form is untouched. * All fields loaded but no modification identified. |
| NgDirty | dirty | * It returns Boolean true when form is modified. * At least one field in the form modified then entire form is recognized as dirty. |
| NgValid | valid | * It returns true when all fields in the form are in valid state. |
| NgInvalid | invalid | * It returns true when any one form field state is recognized as invalid. |
| NgSubmitted | submitted | * It returns true on form submit. |

* All angular validation services return Boolean value

Syntax:

formName.propertyName

frmRegister.invalid

frmRegister.pristine

Ex:

**formvalidation.component.html**

<div class="container-fluid">

<div class="row">

<div class="col-3">

<form #frmRegister="ngForm">

<dl>

<dt>Name</dt>

<dd>

<input type="text" name="txtName" ngModel #txtName="ngModel" required minlength="4">

</dd>

<dt>Mobile</dt>

<dd>

<input type="text" name="txtMobile" ngModel #txtMobile="ngModel" required pattern="\+91[0-9]{10}">

</dd>

</dl>

<button [disabled]="frmRegister.invalid" class="btn btn-primary btn-block">Submit</button>

</form>

</div>

<div class="col-9">

<h2>Form State Services</h2>

<table class="table table-hover">

<thead>

<tr>

<th>Service Name</th>

<th>Value</th>

</tr>

</thead>

<tbody>

<tr>

<td>Pristine</td>

<td>{{frmRegister.pristine}}</td>

</tr>

<tr>

<td>Dirty</td>

<td>{{frmRegister.dirty}}</td>

</tr>

<tr>

<td>Invalid</td>

<td>{{frmRegister.invalid}}</td>

</tr>

<tr>

<td>Valid</td>

<td>{{frmRegister.valid}}</td>

</tr>

<tr>

<td>Submited</td>

<td>{{frmRegister.submitted}}</td>

</tr>

</tbody>

</table>

</div>

</div>

</div>

**Input State Validation Services**

* Input state validation is verifying every form element individually.
* You can validate every form field and identify the issues.

|  |  |  |
| --- | --- | --- |
| **Service** | **Property** | **Description** |
| NgPristine | pristine | It returns true when any specific element is not yet modified. |
| NgDirty | dirty | It returns true when the value of form element is modified. |
| NgTouched | touched | It returns true when element gets focus and blurred. |
| NgUntouched | untouched | It returns true if element never touched. |
| NgValid | valid | It returns true if all input validations are valid. |
| NgInvalid | invalid | It returns true if any one validation property returns invalid. |
| NgErrors | errors | It is an object that collects all errors of input field.   * Required * Minlength * Maxlength * Pattern * Email etc. |

Syntax:

txtName.pristine

txtName.invalid

Ex: Form State and Input State validation

**formvalidation.component.html**

<div class="container-fluid">

<form #frmRegister="ngForm">

<dl>

<h2>Register User</h2>

<dt>User Name</dt>

<dd>

<input type="text" name="txtName" ngModel #txtName="ngModel" class="form-control" required>

<span \*ngIf="frmRegister.submitted && txtName.invalid || txtName.touched && txtName.invalid" class="text-danger">Name Required</span>

</dd>

<dt>Mobile</dt>

<dd>

<input type="text" name="txtMobile" ngModel #txtMobile="ngModel" class="form-control" required>

<span \*ngIf="frmRegister.submitted && txtMobile.invalid || txtMobile.touched && txtMobile.invalid" class="text-danger">Mobile Required</span>

</dd>

<button class="btn btn-primary btn-block">Register</button>

</dl>

</form>

</div>

**Note:** You can’t use “invalid or valid” properties for a field that comprises of multiple validations. You can use “errors” object to identify the specific error in input field.

Ex: **Errors object to handle multiple errors**

<div class="container-fluid">

<form #frmRegister="ngForm">

<dl>

<h2>Register User</h2>

<dt>User Name</dt>

<dd>

<input type="text" name="txtName" ngModel #txtName="ngModel" class="form-control" required minlength="4">

<div \*ngIf="txtName.touched && txtName.invalid" class="text-danger">

<span \*ngIf="txtName.errors.required" >Name Required</span>

<span \*ngIf="txtName.errors.minlength">Name too short..</span>

</div>

</dd>

<dt>Mobile</dt>

<dd>

<input type="text" name="txtMobile" ngModel #txtMobile="ngModel" class="form-control" required pattern="\+91\d{10}">

<div \*ngIf="txtMobile.touched && txtMobile.invalid" class="text-danger">

<span \*ngIf="txtMobile.errors.required">Mobile Required</span>

<span \*ngIf="txtMobile.errors.pattern">Invalid Mobile</span>

</div>

</dd>

<button class="btn btn-primary btn-block">Register</button>

</dl>

</form>

</div>

**Can we use required property for radio button, dropdown list, checkbox to handle required validation?**

1. No. You have to define “Custom Validations”

**Custom Validation**

* Every input validation can’t be handled by using HTML validation properties.
* You have to write custom methods to verify the input value as per your requirements and report an error explicitly.

Ex:

Formvalidation.component.ts

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

@Component({

selector: 'app-formvalidation',

templateUrl: './formvalidation.component.html',

styleUrls: ['./formvalidation.component.css']

})

export class FormvalidationComponent{

public displayCityError = false;

public displayEvenError = false;

public VerifyCity(val){

if(val=='nocity') {

this.displayCityError = true;

} else {

this.displayCityError = false;

}

}

public VerifyEven(val) {

if(val % 2 == 0) {

this.displayEvenError = false;

} else {

this.displayEvenError = true;

}

}

}

Formvalidation.component.html

<div class="container-fluid">

<form #frmRegister="ngForm">

<dl>

<h2>Register User</h2>

<dt>User Name</dt>

<dd>

<input type="text" name="txtName" ngModel #txtName="ngModel" class="form-control" required minlength="4">

<div \*ngIf="txtName.touched && txtName.invalid" class="text-danger">

<span \*ngIf="txtName.errors.required" >Name Required</span>

<span \*ngIf="txtName.errors.minlength">Name too short..</span>

</div>

</dd>

<dt>Mobile</dt>

<dd>

<input type="text" name="txtMobile" ngModel #txtMobile="ngModel" class="form-control" required pattern="\+91\d{10}">

<div \*ngIf="txtMobile.touched && txtMobile.invalid" class="text-danger">

<span \*ngIf="txtMobile.errors.required">Mobile Required</span>

<span \*ngIf="txtMobile.errors.pattern">Invalid Mobile</span>

</div>

</dd>

<dt>Gender</dt>

<dd>

<input type="radio" name="optGender" value="none" ngModel #optGender="ngModel"> None

<input type="radio" name="optGender" value="male" ngModel #optGender="ngModel"> Male

<input type="radio" name="optGender" value="female" ngModel #optGender="ngModel"> Female

</dd>

<dt>Select Your City</dt>

<dd>

<select (change)="VerifyCity(lstCities.value)" name="lstCities" ngModel #lstCities="ngModel" class="form-control" >

<option value="nocity">Select City</option>

<option value="Delhi">Delhi</option>

<option value="Hyd">Hyd</option>

</select>

<span \*ngIf="displayCityError" class="text-danger">Please Select Your City</span>

</dd>

<dt>Enter an Even Number</dt>

<dd>

<input (blur)="VerifyEven(txtEven.value)" type="text" class="form-control" name="txtEven" ngModel #txtEven="ngModel">

<span \*ngIf="displayEvenError" class="text-danger">Not an Even Number</span>

</dd>

<button class="btn btn-primary btn-block">Register</button>

</dl>

</form>

</div>

**CSS Effects for Elements using Validation Properties**

Formvalidation.component.css

dl {

width:400px;

margin:auto;

justify-items: center;

align-items: center;

}

.validStyle {

border:2px solid green;

box-shadow: 2px 2px 3px green;

}

.invalidStyle {

border:2px solid red;

box-shadow: 2px 2px 3px red;

}

Formvalidation.component.html

<dd>

<input [ngClass]="{validStyle:txtName.valid, invalidStyle:txtName.invalid}" type="text" name="txtName" ngModel #txtName="ngModel" class="form-control" required minlength="4">

<div \*ngIf="txtName.touched && txtName.invalid" class="text-danger">

<span \*ngIf="txtName.errors.required" >Name Required</span>

<span \*ngIf="txtName.errors.minlength">Name too short..</span>

</div>

</dd>

**Angular Provides CSS classes for Validation**

* Angular provides a set of CSS validation classes.
* These classes can identity the validation state of form or input element and application effects dynamically.

|  |  |
| --- | --- |
| **Class Name** | **Description** |
| .ng-invalid | It contains a set of style attributes that are applied to any input element of form when its state is returned as “invalid”. |
| .ng-valid | It contains a set of style attributes that are applied to any input element of form when its state is returned as “valid”. |
| .ng-pristine | It contains a set of style attributes that are applied to any input element of form when its state is returned as “pristine”. |
| .ng-dirty | It contains a set of style attributes that are applied to any input element of form when its state is returned as “dirty”. |
| .ng-touched | It contains a set of style attributes that are applied to any input element of form when its state is returned as “touched”. |
| .ng-untouched | It contains a set of style attributes that are applied to any input element of form when its state is returned as “untouched”. |

Form CSS Classes are applied by using form selector or any reference for form

**form.ng-valid { }**

**form.ng-pristine{ }**

Input CSS classes are applied by using input selector or any reference for input.

**input.ng-valid { }**

**input.ng-touched { }**

Ex:

**Formvalidation.component.css**

dl {

width:400px;

margin:auto;

justify-items: center;

align-items: center;

}

input.ng-invalid {

border: 1px solid red;

box-shadow: 2px 2px 3px red;

}

input.ng-valid {

border: 1px solid green;

box-shadow: 2px 2px 3px green;

}

form.ng-invalid {

background-color: lightcoral;

}

form.ng-valid {

background-color:lightgreen;

}

**Formvalidation.component.html**

<div class="container-fluid">

<form #frmRegister="ngForm">

<dl>

<h2>Register User</h2>

<dt>User Name</dt>

<dd>

<input type="text" name="txtName" ngModel #txtName="ngModel" class="form-control" required minlength="4">

<div \*ngIf="txtName.touched && txtName.invalid" class="text-danger">

<span \*ngIf="txtName.errors.required" >Name Required</span>

<span \*ngIf="txtName.errors.minlength">Name too short..</span>

</div>

</dd>

<dt>Mobile</dt>

<dd>

<input type="text" name="txtMobile" ngModel #txtMobile="ngModel" class="form-control" required pattern="\+91\d{10}">

<div \*ngIf="txtMobile.touched && txtMobile.invalid" class="text-danger">

<span \*ngIf="txtMobile.errors.required">Mobile Required</span>

<span \*ngIf="txtMobile.errors.pattern">Invalid Mobile</span>

</div>

</dd>

<dt>Gender</dt>

<dd>

<input type="radio" name="optGender" value="none" ngModel #optGender="ngModel"> None

<input type="radio" name="optGender" value="male" ngModel #optGender="ngModel"> Male

<input type="radio" name="optGender" value="female" ngModel #optGender="ngModel"> Female

</dd>

<dt>Select Your City</dt>

<dd>

<select (change)="VerifyCity(lstCities.value)" name="lstCities" ngModel #lstCities="ngModel" class="form-control" >

<option value="nocity">Select City</option>

<option value="Delhi">Delhi</option>

<option value="Hyd">Hyd</option>

</select>

<span \*ngIf="displayCityError" class="text-danger">Please Select Your City</span>

</dd>

<dt>Enter an Even Number</dt>

<dd>

<input (blur)="VerifyEven(txtEven.value)" type="text" class="form-control" name="txtEven" ngModel #txtEven="ngModel">

<span \*ngIf="displayEvenError" class="text-danger">Not an Even Number</span>

</dd>

<button class="btn btn-primary btn-block">Register</button>

</dl>

</form>

</div>

**Reactive Forms or Model Driven Forms**

* Reactive forms provide Model Driven approach.
* They are bound to model so that any change in model will update the view.
* A model driven approach binds the view with data structure.
* Configuration of forms and controls are defined at application logic level. (controller)
* Easy to extend and loosely coupled.
* Easy to test.
* Clean separation of functionality and presentation (Implementation and Design)
* Reactive forms are asynchronous, they allow to submit only a specific portion of form.
* Type support partial updates.
* You can dynamically add or remove controls from form.
* The library required for configuration and implementation of reactive forms is   
  “@angular/forms”
* The classes required for configure forms and controls dynamically
  + ReactiveFormsModule
  + FormsModule

**Configure a Form Control**

* The form elements like textbox, checkbox, radio, dropdown etc. are configured by using “FormControl” base.
* Any form element can be configured by using “**FormControl**”

**Syntax:**

public elementName = new FormControl(“value”, options);

* The form control is bound to any element in the view by using “[formControl]” property. It is the member of “**ReactiveFormsModule**”. Import the module into “app.module.ts”

**Syntax:**

<input type=”text” [formControl]=”elementName”>

* You can dynamically set value or update value into form control by using the functions
  + setValue()
  + patchValue()

Syntax:

this.elementName.setValue(“somevalue”);

EX:

* Go to “app.module.ts”

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

imports: [

ReactiveFormsModule

],

* Add a new component
* **Reactivedemo.component.ts**

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

import { FormControl } from '@angular/forms';

@Component({

selector: 'app-reactivedemo',

templateUrl: './reactivedemo.component.html',

styleUrls: ['./reactivedemo.component.css']

})

export class ReactivedemoComponent {

public txtName = new FormControl('');

public lstCities = new FormControl('');

public UpdateClick() {

this.txtName.setValue('Samsung TV');

this.lstCities.setValue('Hyderabad');

}

}

* **Reactivedemo.component.html**

<div class="container-fluid">

<div class="row">

<div class="col-3">

<h2>Register</h2>

<div class="form-group">

<label>Name</label>

<div>

<input [formControl]="txtName" type="text" class="form-control">

</div>

</div>

<div class="form-group">

<label>Shipped To</label>

<div>

<select [formControl]="lstCities" class="form-control">

<option>Delhi</option>

<option>Hyderabad</option>

</select>

</div>

</div>

<div class="form-group">

<button (click)="UpdateClick()" class="btn btn-info btn-block">Update</button>

</div>

</div>

<div class="col-9">

<h2>Product Details</h2>

<dl>

<dt>Name</dt>

<dd>{{txtName.value}}</dd>

<dt>Shipped To</dt>

<dd>{{lstCities.value}}</dd>

</dl>

</div>

</div>

</div>

Summary: **FormControl - to create a control, [formControl] – to bind with element in UI.**

**Create and Configure Form and Nested Forms**

* You can dynamically create and configure forms.
* It allows to extend the form and make it more asynchronous.
* You can create a form by using “FormGroup” base.
* “FormGroup” is a collection of FormControls.

Syntax:

public parentForm = new FormGroup({

controlName: new FromControl(‘’),

controlName: new FormControl(‘’),

childForm: new FormGroup({

controlName: new FormControl(‘’)

})

})

* To bind a form and nested form you have to use the properties
  + [formGroup] – Parent Form
  + [formGroupName] – Child Form

Syntax:

<form [formGroup]=”parentForm”>

<div [formGroupName]=”childForm”>

</div>

</form>

* If you are defining a control in form group the control is bound to element by using the attribute

“formControlName”

Syntax:

**<input type=”text” formControlName=”controlName”>**

* The method used to set and patch values are
  + setValue()
  + patchValue()

Ex:

**Reactivedemo.component.ts**

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

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

@Component({

selector: 'app-reactivedemo',

templateUrl: './reactivedemo.component.html',

styleUrls: ['./reactivedemo.component.css']

})

export class ReactivedemoComponent {

public frmRegister = new FormGroup({

Name: new FormControl(''),

Price: new FormControl(''),

frmDetails: new FormGroup({

City: new FormControl(''),

InStock: new FormControl('')

})

});

public UpdatePartial() {

this.frmRegister.patchValue({

Name: 'Samsung TV',

frmDetails: {

City: 'Delhi',

InStock: true

}

});

}

}

**Reactivedemo.component.html**

<div class="container-fluid">

<div class="row">

<div class="col-3">

<h2>Register Product</h2>

<form [formGroup]="frmRegister">

<fieldset>

<legend>Basic Info</legend>

<dl>

<dt>Name</dt>

<dd>

<input formControlName="Name" class="form-control" type="text">

</dd>

<dt>Price</dt>

<dd>

<input formControlName="Price" type="text" class="form-control">

</dd>

</dl>

</fieldset>

<fieldset>

<legend>Stock Details</legend>

<div formGroupName="frmDetails">

<dl>

<dt>City</dt>

<dd>

<select formControlName="City" class="form-control">

<option>Delhi</option>

<option>Hyd</option>

</select>

</dd>

<dt>In Stock</dt>

<dd>

<input formControlName="InStock" type="checkbox">

</dd>

</dl>

<button class="btn btn-primary btn-block" (click)="UpdatePartial()">Update Details</button>

</div>

</fieldset>

</form>

</div>

<div class="col-9">

<h2>Product Details</h2>

<dl>

<dt>Name</dt>

<dd>{{frmRegister.value.Name}}</dd>

<dt>Price</dt>

<dd>{{frmRegister.value.Price}}</dd>

<dt>City</dt>

<dd>

{{frmRegister.value.frmDetails.City}}

</dd>

<dt>Stock</dt>

<dd>

{{(frmRegister.value.frmDetails.InStock)==true?"Available":"Out of Stock"}}

</dd>

</dl>

</div>

</div>

</div>