Angular Forms

- Form is a container that comprises of a set of elements like button, textbox, checkbox, radio etc.
- Form provide as UI from where user can input, edit, delete or view data.
- It provides an interface for interacting with data.
- In Angular you can configure 2 type of forms
    - **Template Driven Forms**
    - **Model Driven Form / Reactive Forms**

Template Driven Forms

- A template driven form configures and handles all interactions at View Level (HTML).
- Configuration of a form and its manipulation both are handled in HTML template.
- Very optimized controller level interaction. All interactions are managed in View.
- It reduces the number of requests to component.
- It can improve the page load time.
- It is good for forms designed in “in-line” technique.
- Template driven form is heavy on page.
- It is slow in handling interactions.
- It renders slow.
- It is hard to test and extend form.
- Separation issues. Not loosely coupled.

- You can use template driven forms when you are designing as UI that doesn’t require regular extension. And form doesn’t require dynamic changes in UI.

Configuring Template Driven Form:

- Angular provides the following directives 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 method that are used to configure and handle a form control like, textbox, checkbox, radio, listbox etc.
- The library for “NgForm and NgModel” is “@angular/forms”.
- The module is “FormsModule”

Syntax:

Configure Form:

<form #formName=”ngForm”>

</form>

- NgForm provide set of attributes
    - value
    - pristine
    - dirty
    - valid
    - invalid
    - submitted etc.

Configure a Form Element:

- NgModel is used to make a static form element into dynamic.

Syntax:

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

txtName.value

txtName.valid

txtName.dirty

txtName.invalid etc.

- The form data is submitted to controller by using “Submit” event.
- The form object “frmRegister.value” can send the details of all elements and their values.

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 implements
OnInit {

  constructor() { }

  ngOnInit(): void {
  }
  onSubmit(obj){
    alert('Name:' + obj.txtName);
  }
```

}

TemplateForm.component.html

```
<div class="container-fluid">
  <h2>Register Product</h2>
  <form #frmRegister="ngForm"
(submit)="onSubmit(frmRegister.value)">
    <dl>
      <dt>Name</dt>
      <dd>
        <input name="txtName" ngModel
#txtName="ngModel" type="text">
      </dd>
      <dt>Price</dt>
      <dd>
        <input type="text" name="txtPrice" ngModel
#txtPrice="ngModel">
      </dd>
      <dt>Shipped To</dt>
      <dd>
        <select name="shippedTo" ngModel
#shippedTo="ngModel">
```

```
            <option>Delhi</option>
            <option>Hyd</option>
          </select>
        </dd>
    </dl>
    <button type="submit">Submit</button>
</form>
<h2>Details [Control Reference]</h2>
<dl>
    <dt>Name</dt>
    <dd>{{txtName.value}}</dd>
    <dt>Price</dt>
    <dd>{{txtPrice.value}}</dd>
    <dt>Shipped To</dt>
    <dd>{{shippedTo.value}}</dd>
</dl>
<h2>Details [Form Reference]</h2>
<dl>
    <dt>Name</dt>
    <dd>{{frmRegister.value.txtName}}</dd>
```

```
    <dt>Price</dt>
    <dd>{{frmRegister.value.txtPrice}}</dd>
  </dl>
</div>
```

Validating Template Driven Form

- Validation is the process of verifying user input.
- Validation is required to ensure that contradictory and unauthorized data is not get stored into database.
- Angular handles validation client side.
- Angular provides a set of validation services with pre-defined functionality.
- Angular validation services are classified into 2 types
    - Form State Validation Services
    - Input State Validation Services

Form State Validation Services:

- A set of validation services used to validation a form.
- Form state services will validate all controls in a form.
- Form state is not individually verifying every control. It verifies all controls at the same time.

- Form state validation services can be accessed by using the form reference. [NgForm]

Service	Property	Type	Description
NgPristine	pristine	boolean	- It returns true if form is not modified. - Form is loaded but no modifications identified. [true]
NgDirty	dirty	boolean	- It returns true if any element value in the form is modified.
NgValid	valid	boolean	- It returns true if all forms elements have valid data.
NgInValid	invalid	boolean	- It returns true if any one element in a form is having invalid data.
NgSubmitt	submitt	boolea	- It returns true

ed	ed	n	on form submit.

Syntax:

formName.pristine

formName.dirty

formName.submitted etc..

Ex:

Templateform.component.html

```
<div class="container-fluid">
  <div class="row">
    <div class="col-3">
      <h2>Register</h2>
      <form [ngClass]="{'valid-style':frmRegister.valid,
'invalid-style':frmRegister.invalid}"
#frmRegister="ngForm">
      <div class="form-group">
         <label>User Name</label>
         <div>
```

```
            <input required minlength="4" maxlength="10"
name="UserName" ngModel #UserName="ngModel"
type="text" class="form-control">
        </div>
    </div>
    <div class="form-group">
      <label>Mobile</label>
      <div>
          <input required pattern="\+91[0-9]{10}"
type="text" class="form-control" ngModel
#Mobile="ngModel" name="Mobile">
      </div>
    </div>
    <div class="form-group">
     <div class="btn-group">
         <button [disabled]="frmRegister.invalid"
class="btn btn-primary">Register</button>
         <button *ngIf="frmRegister.dirty" class="btn
btn-success">Save</button>
      </div>
    </div>
   </form>
```

```
</div>
<div class="col-9">
  <h2>Validation State</h2>
  <table class="table table-hover">
    <thead>
        <tr>
          <td>Pristine</td>
          <td>{{frmRegister.pristine}}</td>
        </tr>
        <tr>
           <td>Dirty</td>
           <td>{{frmRegister.dirty}}</td>
         </tr>
         <tr>
           <td>Valid</td>
           <td>{{frmRegister.valid}}</td>
         </tr>
         <tr>
           <td>Invalid</td>
           <td>{{frmRegister.invalid}}</td>
```

```
            </tr>
            <tr>
              <td>Submitted</td>
              <td>{{frmRegister.submitted}}</td>
            </tr>
        </thead>
      </table>
    </div>
  </div>
</div>
```

Templateform.component.css

```
button:disabled {
    cursor: not-allowed;
}
.valid-style {
    background-color: rgb(199, 252, 199);
}
.invalid-style {
    background-color: rgb(252, 190, 190);
}
```

```
form {
   padding: 20px;
}
```

Input State Validation Services

- These are the validation services provided by Angular to verify the validation state of every element individually.
- You can access with reference of element name.

Service	Property	Type	Description
NgPristine	pristine	boolean	- It returns true if specific from element value is not modified.
NgDirty	dirty	boolean	- It returns true if specific form element value is modified
NgValid	valid	boolean	- It returns true if specific element have valid data.

NgInValid	invalid	boolean	- It returns true if specific element is having invalid data.
NgTouched	touched	boolean	- It returns true if any specific element is touched. [Gets focus]
NgUnTouched	untouched	boolean	- It returns true if any specific element is untouched.
NgErrors	errors	object	- It is an error object that can collect all errors of element. It can individually verify every validation error.

Angular Provides Validation CSS classes to defined effects dynamically by verifying the validation state:

- .ng-valid
- .ng-invalid
- .ng-pristine
- .ng-dirty
- .ng-touched
- .ng-untouched

input.ng-valid { }

form.ng-valid { }

Ex:

Inputvalidation.component.ts

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


@Component({
  selector: 'app-inputvalidation',
  templateUrl: './inputvalidation.component.html',
  styleUrls: ['./inputvalidation.component.css']
})
export class InputvalidationComponent{
  showCityError = true;
  showEvenError = false;
```

```
  onCityChange(val) {
    if(val=='notcity') {
      this.showCityError = true;
    } else {
      this.showCityError = false;
    }
  }
  VerifyEven(val) {
     if(val % 2 == 0) {
       this.showEvenError = false;
     } else {
       this.showEvenError = true;
     }
  }
}
```

Inputvalidation.component.html

```
<div class="container-fluid">
 <div class="form-register">
   <h2>Register</h2>
   <form #frmRegister="ngForm">
```

```
    <div class="form-group">
        <label>User Name</label>
        <div>
            <input ngModel #txtName="ngModel"
name="txtName" type="text" 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>
        </div>
    </div>
    <div class="form-group">
        <label>Mobile</label>
        <div>
```

```
            <input type="text" name="txtMobile" ngModel
#txtMobile="ngModel" class="form-control" required
pattern="\+91[0-9]{10}">
                <div class="text-danger"
*ngIf="txtMobile.touched && txtMobile.invalid">
                    <span
*ngIf="txtMobile.errors.required">Mobile
Required</span>
                    <span
*ngIf="txtMobile.errors.pattern">Invalid
Mobile</span>
                </div>
            </div>
        </div>
        <div class="form-group">
            <label>Select Your City</label>
            <div>
                <select (change)="onCityChange(lstCity.value)"
name="lstCity" ngModel #lstCity="ngModel"
class="form-control">
                    <option value="notcity">Select City</option>
                    <option value="Hyd">Hyd</option>
```

```
            <option value="Delhi">Delhi</option>
            <option value="Mumbai">Mumbai</option>
          </select>
          <span *ngIf="showCityError" class="text-danger">Please Select Your City</span>
        </div>
      </div>
      <div class="form-group">
        <label>Enter Even number</label>
        <div>
          <input (blur)="VerifyEven(txtEven.value)" type="text" name="txtEven" ngModel #txtEven="ngModel" class="form-control">
          <span class="text-danger" *ngIf="showEvenError">Not an Even Number</span>
        </div>
      </div>
      <div class="form-group">
        <button class="btn btn-primary btn-block">Register</button>
      </div>
```

```
    </form>
 </div>
</div>
```

Inputvalidation.component.css

```
.form-register {
  width: 300px;
  padding:20px;
  margin:auto;
  justify-content: center;
  align-items: center;
}
input.ng-invalid{
    border:1px solid red;
    box-shadow: 2px 3px 4px red;
}
input.ng-valid{
    border:1px solid green;
    box-shadow: 2px 3px 4px green;
}
```

Model Driven Form / Reactive Form

- Reactive forms provide a model driven approach
- They are bound to model. So that any change in model will update the view.
- All controls and elements are configured at application logic level. (controller)
- Easy to extend and loosely coupled.
- Easy to test.
- Clean separation of functionality and presentation.
- Reactive forms are asynchronous, they allow to submit only a specific portion of form, instead of submitting entire form.
- They support partial updates.
- They use AJAX.
- You can dynamically add or remove controls from form.
- Form can change according to state and situation.
- If regular extensions are required and dynamic manipulations are required then go with Reactive forms.
- If you are using multiple files for a component then the initial load time will increase.
- The library required for configuring and manipulating reactive forms “@angular/forms”
- The Modules required to configure forms and controls

- ReactiveFormModule
- FormsModule

Configure a Form Control

- The form elements like textbox, checkbox, radios, dropdown etc are configured using “FormControl” base.

Syntax:

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

- You have to bind the control with UI element by using property “formControl”

  Syntax:

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

- You can dynamically set value or update value into form control by using following functions
  - setValue()
  - patchValue()

Syntax:

this.elementName.setValue(someValue);

Ex:

- Import **ReactiveFormsModule** in “app.module.ts”

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{
  txtName = new FormControl('');
  lstCities = new FormControl('');

  UpdateClick() {
    this.txtName.setValue('Samsung TV');
    this.lstCities.setValue('Hyd');
  }
```

```
}
```

Reactivedemo.component.html

```
<div class="container-fluid">
  <div class="row">
     <div class="col-3">
       <h2>Register Product</h2>
       <div class="form-group">
          <label>Name</label>
          <div>
             <input [formControl]="txtName" type="text"
class="form-control">
          </div>
       </div>
       <div class="form-group">
          <label>Select City</label>
          <div>
             <select [formControl]="lstCities" class="form-
control">
                <option>Delhi</option>
                <option>Hyd</option>
             </select>
```

```
        </div>
      </div>
      <div class="form-group">
        <button (click)="UpdateClick()" class="btn btn-
primary btn-block">Update Details</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>
```

Configure Forms and Nested Forms with Controls

- 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 methods 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{
  frmRegister = new FormGroup({
     Name: new FormControl(''),
     Price: new FormControl(''),
     frmDetails: new FormGroup({
       City: new FormControl(''),
       Instock: new FormControl('')
     })
  });
  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 type="text"
formControlName="Name" class="form-control">
                </dd>
                <dt>Price</dt>
                <dd>
                   <input type="text"
formControlName="Price" 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 (click)="UpdatePartial()" class="btn
btn-primary  btn-block">Update Details</button>
            </div>
         </fieldset>
```

```
      </form>
    </div>
    <div class="col-9">
      <h3>Product Details</h3>
      <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?"Availabl
e":"Out of Stock"}}
        </dd>
      </dl>
    </div>
  </div>
</div>
```

Form Builder in Reactive Approach

- Form builder is a **service** provided by Angular to configure forms and its elements dynamically.
- FormBuilder uses singleton pattern.
- Memory is allocated for first request to form the same memory is used across multiple requests.
- FormBuilder is the base for configuring forms and its controls, it provides the following methods
    - group()
    - control()
    - array()

Form Builder Method	**Description**
group()	It configures a form group with set of elements. It dynamically creates <form> element. It can be used to create nested forms.
control()	It configures form element like <input>, <select>, <option>, <textarea> etc.
array()	It configures a collection of form controls. It allows to add or remove controls dynamically.

- The properties that are used to bind with UI elements
    - formGroup : Parent Form
    - formGroupName : Child Form
    - formControlName : Form elements
- **FormBuilder** is a member of “@angular/forms”

Syntax:

```
constructor(private fb: FormBuilder) { }

parentForm = this.fb.group({
   controlName: [‘value‘, validators],
   controlName: [‘value’, validation],
   childForm: this.fb.group({
        controlName: [‘value’, validation]
     })
});

<form [formGroup]=”parentForm”>
    <div formGroupName=”childForm”>
       <input type=”text” formControlName=”controlName”>
    </div>
```

```
  </form>
```

Ex:

Builderdemo.component.ts

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

@Component({
  selector: 'app-builderdemo',
  templateUrl: './builderdemo.component.html',
  styleUrls: ['./builderdemo.component.css']
})
export class BuilderdemoComponent implements
OnInit {

  constructor(private fb: FormBuilder) { }

  frmRegister = this.fb.group({
    Name: [''],
    Price: [''],
    frmDetails: this.fb.group({
```

```
    City: [''],
    InStock: ['']
   })
  });

  ngOnInit(): void {
  }

}
```

Builderdemo.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 type="text"
formControlName="Name" class="form-control">
            </dd>
            <dt>Price</dt>
            <dd>
                <input type="text"
formControlName="Price" 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 type="checkbox"
formControlName="InStock">  Yes
                </dd>
              </dl>
            </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)?"Availa
ble":"Out  of Stock"}}
        </dd>
      </dl>
   </div>
  </div>
</div>
```

Form Array and Form Control

- Form Array allows to add or remove controls dynamically.
- It is configure by using “array()” of FormBuilder service.
- It represents a simple typescript array and make use of all array function
    - push()
    - unshift()
    - pop()
    - shift()
    - removeAt() etc.
- Form “control()” is used to allocate memory for a control dynamically.

Syntax:

formBuilderObject.array([formbuilder.control(), formbuilder.control()])

Ex:

builderdemo.component.ts

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

@Component({
  selector: 'app-builderdemo',
  templateUrl: './builderdemo.component.html',
  styleUrls: ['./builderdemo.component.css']
})
export class BuilderdemoComponent implements OnInit {

  constructor(private fb: FormBuilder) { }

  frmRegister = this.fb.group({
```

```
    Name: [''],
    Price: [''],
    frmDetails: this.fb.group({
      City: [''],
      InStock: ['']
    }),
    newControls: this.fb.array([this.fb.control('')])
  });

  // Accessor for NewControls
  get newControls(){
    return this.frmRegister.get('newControls') as
FormArray;
  }

  AddPhoto() {
    this.newControls.push(this.fb.control(''));
  }

  RemovePhoto(i) {
```

```
    this.newControls.removeAt(i);
  }

  ngOnInit(): void {
  }

}
```

Builderdemo.component.html

```
<div class="container-fluid">
  <div class="row">
    <div class="col-4">
      <h2>Register Product</h2>
      <form [formGroup]="frmRegister">
        <fieldset>
          <legend>Basic Info</legend>
          <dl>
            <dt>Name</dt>
            <dd>
              <input type="text"
formControlName="Name" class="form-control">
```

```
            </dd>
            <dt>Price</dt>
            <dd>
               <input type="text"
formControlName="Price" 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 type="checkbox"
formControlName="InStock">  Yes
                </dd>
            </dl>
        </div>
        <div>
            <h2>Upload Photo
                <button (click)="AddPhoto()" class="btn
btn-link">Add More</button>
            </h2>
            <div *ngFor="let item of
newControls.controls; let i = index" style="margin-
top: 20px;">
                <div class="form-inline">
                    <div><input type="file"
formControlName="i"></div>
                    <div><button (click)="RemovePhoto(i)"
class="btn btn-link">Remove</button></div>
                </div>
            </div>
        </div>
```

```
      </fieldset>
    </form>
  </div>
  <div class="col-8">
    <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)?"Avai
lable":"Out  of Stock"}}
      </dd>
    </dl>
  </div>
 </div>
```

Validating Input in Reactive Forms

- In a reactive form component class is “Source of Truth”
- We configure and manipulate controls in component class.
- Instead of adding validator through attribute in template. You can configure them in a controller class.
- Angular will call the validator functions whenever the value changes.
- Angular uses pre-defined validator functions for verifying the value in class.
- It verifies the input value with the validator defined in class and returns boolean true or false.
- The built-in validator functions of Angular are defined in “Validators” class.
- You can also create custom validators.
- The commonly used validator functions are:
    - min()
    - max()
    - required()
    - requiredTrue()
    - email()
    - minlength()

- maxlength()
- pattern()
- nullValidator()
- compose()
- composeAsync() etc.

FAQ: What are Sync and Async Validators?

- **Sync Validators** are Synchronous functions that take a control instance (object) and immediately return a set of validation errors or null.
- **Async Validators** are asynchronous functions that take a control instance and return a observable which emits the result later as per the situation.
- **Angular by default uses “Async” validators.**

Syntax:

public txtName = new FormControl(“value”, [Validators])

<input validator>

- The validator functions are defined in “Validators” base class of “angular/forms”

Ex:

ReactiveValidation.component.ts

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

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



@Component({
  selector: 'app-reactivevalidation',
  templateUrl: './reactivevalidation.component.html',
  styleUrls: ['./reactivevalidation.component.css']
})
export class ReactivevalidationComponent implements
OnInit {

  frmRegister: FormGroup;
  submitted = false;


  constructor(private fb: FormBuilder) { }


  ngOnInit(): void {
    this.frmRegister = this.fb.group({
      UserName: ['', [Validators.required,
Validators.minLength(4)]],
```

```
      Mobile: ['', [Validators.required,
Validators.pattern(/\+91[0-9]{10}/)]],
      Email: ['', [Validators.required, Validators.email]]
    });
  }
  get frm() {
    return this.frmRegister.controls;
  }
  OnSubmit() {
    this.submitted = true;
    if(this.frmRegister.invalid) {
      return;
    }
    alert('Registered Successfully.');
  }
}
```

ReactiveValidation.component.html

```
<div class="container-fluid">
  <h2>Register USer</h2>
  <form [formGroup]="frmRegister"
(ngSubmit)="OnSubmit()" >
```

```
    <div class="form-group">
        <label>User Name</label>
        <div>
            <input type="text"
formControlName="UserName" class="form-control">
            <div *ngIf="submitted &&
frm.UserName.errors" class="text-danger">
                <span
*ngIf="frm.UserName.errors.required">User Name
Required</span>
                <span
*ngIf="frm.UserName.errors.minlength">Name too
short..</span>
            </div>
        </div>
    </div>
    <div class="form-group">
        <label>Mobile</label>
        <div>
            <input type="text" formControlName="Mobile"
class="form-control">
```

```
        <div *ngIf="submitted && frm.Mobile.errors"
class="text-danger">
            <span
*ngIf="frm.Mobile.errors.required">Mobile
Required</span>
            <span
*ngIf="frm.Mobile.errors.pattern">Invalid
Mobile</span>
        </div>
      </div>
   </div>
   <div class="form-group">
      <label>Email</label>
      <div>
        <input type="text" formControlName="Email"
class="form-control">
        <div *ngIf="submitted && frm.Email.errors"
class="text-danger">
            <span
*ngIf="frm.Email.errors.required">Email
Required</span>
```

```
            <span *ngIf="frm.Email.errors.email">Invalid
Email..</span>
          </div>
        </div>
    </div>
    <div class="form-group">
        <button class="btn btn-primary btn-
block">Register</button>
    </div>
  </form>
</div>
```

Angular Routing