UNIT – III
COMPONENTS, PIPES AND FORMS

Components

- Angular components are the building blocks of Angular applications.
- It encapsulate a part of the user interface, including its template, styles, and behavior.
- Each component represents a reusable piece of UI functionality and can be composed together to create complex applications.
- Components in Angular follow the principles of encapsulation, reusability, and maintainability.

Component Structure

Structure of an Angular component consists of three main parts:

Template

- The template defines the HTML markup of the component's view.
- It contains placeholders and Angular directives that are replaced with dynamic data and logic during runtime.

Styles

- The styles define the component's visual appearance
- It includes CSS rules and stylesheets.
- Styles can be defined using inline styles, external CSS files, or CSS preprocessors like Sass or Less.

TypeScriptCode

- The TypeScript code defines the component's behavior and logic.
- It includes properties, methods that control how the component interacts with the DOM
- It handles user input, and responds to changes in its state or props.

Create route:

To create a route for multiple pages

app.routes.ts

```
import { provideRouter, Routes } from '@angular/router';
import { Login } from './login/login';
import { Swap } from './swap/swap';
export const routes: Routes = [{path:"login",component:Login},
   {path:"swap",component:Swap},
{ path: '', redirectTo: 'login', pathMatch: 'full' }];
export const appProviders = [provideRouter(routes)];
```

App.html

```
<ul ><li ><a  href="/login">Login program</a>
</li>
<li >
<a  href="/swap" >Swapping Program</a>
</li>
</ul>
<div>
<router-outlet></router-outlet>
</div>
```

login.html:

```
<h2>Login</h2>
<form #loginForm="ngForm" (ngSubmit)="onSubmit()">
<label>Email</label>
<input type="email" name="email" [(ngModel)]="user.email" required />
<label>Password</label>
<input type="password" name="password" [(ngModel)]="user.password" required >
<button type="submit" [disabled]="!loginForm.valid">Login</button>
</form>
<P>{{res}}</P>
</div>
```

login.ts

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { FormsModule } from '@angular/forms';
@Component({
  selector: 'app-root',
  imports: [RouterOutlet,FormsModule],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  user = {
    email: '',
    password: ''
  };
    res:string='';
  onSubmit() {
    if (this.user.email === 'anjac' && this.user.password === 'mca') {
      this.res="VALID USER"
    } else {
       this.res="INVALID USER"
    }
  }
  protected title = 'login';
}
```

swap.html

```
<p class="lead text-center fw-bold">Swapping Any Data Type</p>
<form #formItems="ngForm" (ngSubmit)="onSubmit(formItems)">
<label for="num1" class="form-label">Enter The First Value</label>
<input type="text" name="num1" class="form-control" ngModel required>
<label for="num2" class="form-label">Enter The Second Value</label>
<input type="text" name="num2" class="form-control" ngModel required>
<select class="form-select mt-4" name="sel" ngModel >
<option selected disabled>Select Data Type</option>
<option value='0'>Number</option>
<option value='1'>String</option>
</select>
<button type="submit" class="btn btn-outline-success">
<b>SWAPPING</b>
</button>
</div>
</form>
<p>Before Swapping : <span>{{firstValue + ',' + secondValue}}</span></p>
<p>After Swapping : <span>{{ f1 + ',' + f2 }} </span></p>
```

swap.ts:

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
@Component({
  selector: 'app-swap',
  imports: [FormsModule,CommonModule],
  templateUrl: './swap.html',
  styleUrl: './swap.css'
})
export class Swap {
 protected title = 'swapping';
firstValue: string = ''
```

```
  secondValue: string = ''
  f1: string = ''
  f2: string = ''
  selectDataType: any = ''
  onSubmit(form: any) {
    this.firstValue = form.value.num1
    this.secondValue = form.value.num2;
    this.selectDataType=form.value.sel;
    switch (this.selectDataType) {
      case '0':
        let a:number=Number(this.firstValue);let b:number=Number(this.secondValue);
        a=a+b;
        b=a-b;
        a=a-b;
        this.f1=a.toString();
        this.f2=b.toString();
        break
      case '1':
        [this.f2,this.f1]=[this.firstValue,this.secondValue];
        break
    }
  }
}
```

****

Component Style

1. External Stylesheets

- Define component-specific styles in a separate CSS file (e.g., my-component.component.css).
- Reference this file in the @Component decorator using the styleUrls property, which accepts an array of strings:

Example

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.css'] // Reference external stylesheet
})
export class MyComponent {     // ...   }
```

2. Inline Styles

- It define styles directly within the @Component decorator using the styles property.
- This property also accepts an array of strings, where each string contains CSS rules.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-my-component',
  template: `<h1 style="color: blue;">Hello from ANJAC</h1>`,
  styles: [ `  h1 {   font-size: 24px;  } `]
})
export class MyComponent {
  // ...
}
```

3. Template Inline Styles

- It apply styles directly to HTML elements within the component's template using the style attribute, similar to standard HTML:

```
<p style="font-weight: bold; color: green;">ANJAC.</p>
```

****

View Encapsulation

- In Angular, View encapsulation is a mechanism that allows to control how styles are applied to components.
- It ensures that the styles defined in one component do not affect other parts of the application.
- Ithas three typesof view encapsulation: Emulated, Shadow DOM, and None.

i. ViewEncapsulation.Emulated

- Emulated is the default encapsulation mode in Angular.
- It emulates the behavior of the Shadow DOM by adding unique attributes to the component's CSS selectors and the corresponding HTML elements.
- It ensures that the styles are scoped to the component and do not leak out to other components.
- Emulated encapsulation is suitable for most use cases as it provides a good balance between isolation and performance.

ii. ViewEncapsulation.ShadowDom

- Shadow DOM uses the browser's native Shadow DOM API to encapsulate the component's styles.
- It creates a ShadowRoot for the component's host element, ensuring that the styles are completely isolated from the rest of the application
- Shadow DOM encapsulation provides true isolation but may not be supported in all browsers.

iii. ViewEncapsulation.None

- In this mode, Angular does not apply any encapsulation.
- The styles defined in the component are globally applied and can affect any HTML element within the application.
- None encapsulation should be used with caution as it can lead to style conflicts across the application.

Example

App.ts

```
import { Component, ViewEncapsulation } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { Aboutus } from './aboutus/aboutus';
import { Home } from './home/home';
@Component({
  selector: 'app-root',
  imports:[Home,Aboutus],
  templateUrl: './app.html',
  styleUrl: './app.css',
 //encapsulation:ViewEncapsulation.None
 //encapsulation:ViewEncapsulation.Emulated
 encapsulation:ViewEncapsulation.ShadowDom
})
export class App {
  protected title = 'view_encap';
}
```

App.html

```
<h1>{{title}}</h1>
<p>I am a paragraph in green</p>
<app-home></app-home>
<app-aboutus></app-aboutus>
```

App.css

```
p  {color: green;}
```

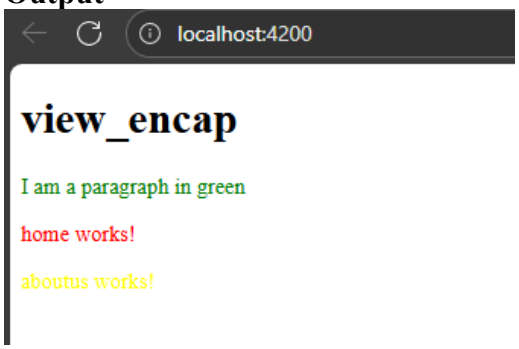
home.css

```
p
{
   color: red;
}
```

Aboutus.css

```
p
{
   color: yellow;
}
```

Output



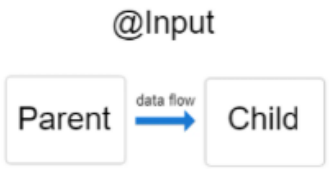
*****

Input & Output directives

- In Angular, components often need to communicate with each other.
- The @Input() and @Output() decorators are used to achieve this communication.
- These properties help in creating modular, reusable components by establishing a structured way of sharing data.

@Input() property

- @Input() property is writable
- It is used to pass data from a parent component to a child component.



- **Declaring @Input() :** @Input() propertyName: type;

Example of @Input()

Step 1 : Step 1: Define an @Input() property in the Child Component

Child-Component.ts

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-child',
  template: `<h3>Welcome, {{ name }}!</h3>`
})
export class ChildComponent {
  @Input() name!: string;
}
```

Step 2: Pass data from the Parent Component

Parent.ts

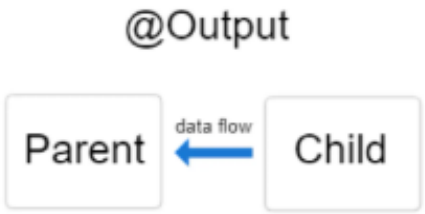
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-parent',
  template: `<app-child [name]="userName"></app-child>`
})
export class ParentComponent {
  userName: string = AK';
}
```

Output : Welcome AK

@Output() property

- @Output() property is observed

- It is used to send events or data from a child component to a parent component.



- **Declaring @Output:** @Output() propertyName = new EventEmitter<type>();

Example of @Output() in Angular
Step 1: Define an @Output() Event Emitter in the Child Component
child-component.ts

```
import { Component, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'app-child',
  template: `<button (click)="sendMessage()">Click Me</button>`
})
export class ChildComponent {
  @Output() messageEvent = new EventEmitter<string>();
    sendMessage() {
    this.messageEvent.emit('Welcome MCA From Child!');
  }
}
```

Step 2: Handle the Event in the Parent Component

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-parent',
  template: `<app-child (messageEvent)="receiveMessage($event)"></app-child>
            <p>{{ message }}</p>`
})
export class ParentComponent {
  message: string = '';
  receiveMessage(event: string) {
    this.message = event;
  }
}
```

Output
(Button is clicked)
Welcome MCA From Child!

****

Pipes

- The pipes that transform data directly in the template.
- It takes in data as input and transform it into the desired output.
- Pipes are used to transform data within template expressions for display purposes.

Built in Pipes
There are built in pipes are used to provides a convenient way to format data such as, dates, currencies, numbers, and strings without modifying the component data.
String

- It has several Methods
- UpperCasePipe: It transforms text to uppercase
- LowerCasePipe: It transforms text to lowercase
- SlicePipe: It slice a string or array and return a new sub string or sub array.
- TitleCasePipe: It capitalizes the first letter of each word.

Date and Time

- It formats a date value according to the rules

Number Formatting

- It has several methods
- CurrencyPipe: Formats a number as currency.
- DecimalPipe: Formats a number as decimal.
- PercentPipe: Formats a number as a percentage.

JSON:

- It displaying object values as a JSON string)

Example :

App.Component.cs

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})
export class AppComponent  {
  name = 'anja college';
  rollno="SF466";
  const currentDate: Date = new Date();
  const indianrupee = 466.10;
  const mark=0.78;
  const m1=0.1234;
}
```

App.Component.ts

```
<p>Original Text: {{ name }}</p>
<p>Uppercase Text: {{ name | uppercase }}</p>
<p>Lowercase Text: {{ rollno | lowercase }}</p>
<p>Title Case Text: {{ rollno | titlecase }}</p>
<p>Chained Transformation: {{ name | uppercase | slice:0:6}}</p>
<p>Formatted Date: {{ currentDate | date:'fullDate' }}</p>
<p>{{ 234.567 | currency:'USD' }}</p>
<p>{{ 2234.56 | currency:'EUR':'symbol':'1.0-0' }}</p>
<p>{{ indianrupee | currency:'EUR':'symbol':'1.0-0' }}</p>
<p>{{ indianrupee | currency:'USD':'symbol':'1.0-0' }}</p>
<p>{{ mark | percent }}</p>
<p>{{ m1 | percent:'1.0-2' }}</p>
```

Output

Original Text: anja college
Uppercase Text: ANJA COLLEGE
Lowercase Text: sf466
Title Case Text: Sf466
Chained Transformation: ANJA C
Formatted Date: Saturday, August 16, 2025
$234.57
€2,235
€466
$466
78%
12.34%

Example 2 :

App.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})
export class AppComponent  {
  myProfile = {
    name: 'AK',
    city: 'Sivakasi',
    hobbies: ['read','code','test']
  };
}
```

App.component.html

```
<div>
  <h2>Object Data </h2>
  <p>{{ myProfile }}</p>
  <h2>JSON Formatted Data</h2>
  <pre>{{ myProfile | json }}</pre>
  <pre>{{ myProfile["name"] | lowercase }}</pre>
</div>
```

Output

```
Object Data
[object Object]
JSON Formatted Data
{
  "name": "AK",
  "city": "Sivakasi",
  "hobbies": [
    "read",
    "code",
    "test"
  ]
}
Ak
```

Creating User Defined Pipe and Custom Pipe

- Creating a custom pipe in Angular involves defining a class that implements the PipeTransform interface and is decorated with the @Pipe decorator.
- It allows to transform data within the templates.

Steps to create a custom pipe

1. **Generate the pipe**

- Generate the pipe using the ng generate pipe <pipe-name>
- The generated pipe class will automatically implement the PipeTransform interface from @angular/core.
- This interface requires the implementation of a transform method.

2. Define the transform method

- The transform method is where the custom transformation logic resides.
- It takes the input value as its first argument and can accept additional parameters to customize the transformation.
- It should return the transformed value.

3. Use the custom pipe in templates

- Apply the pipe to an expression using the pipe symbol (|) followed by the pipe's name and any optional arguments.

Example 1 :

Filter.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'filter'
})
export class FilterPipe implements PipeTransform {
  transform(items: any[], searchTerm: any, searchBy: string) {
    if (!searchTerm) {
      return items;
    }
    return items.filter(item => {
      const currentItem = item[searchBy];
      return currentItem.toString().toLowerCase().includes(searchTerm.trim().toLowerCase());
    });
  }
}
```

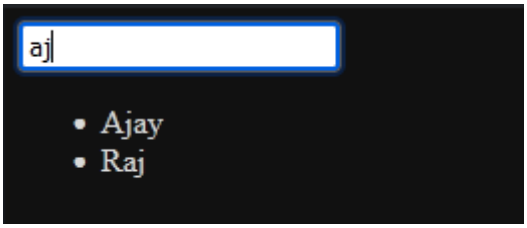
app.component.ts

```
import { Component } from '@angular/core';
export interface Students {
  name: string;
}
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})
export class AppComponent  {
  stud: Students[] = [];
  searchTerm: string;
  names = ['Alagukumar', 'Sabari', 'Ajay', 'Raj', 'Kumar', 'Vel'];
  constructor() {
    this.names.forEach((e) => this.stud.push({
      name: e
    }));
  }
}
```

App.Component.html

```
<div>
  <input [(ngModel)]="searchTerm" placeholder="Filter by Name">
  <ul>
    <li *ngFor="let person of stud | filter: searchTerm: 'name'">
      {{person.name}}</li>
  </ul>
</div>
```

Output



Example 2:

round-number.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
   name: 'roundNumber',
   standalone: true,
})
export class RoundNumberPipe implements PipeTransform {
   transform(value: number | string): number {
      return Math.round(Number(value));
   }
}
```

app.component.ts

```
import { Component } from '@angular/core';
import { RoundNumberPipe } from './round-number.pipe';
@Component({
   selector: 'app-root',
   standalone: true,
   imports: [RoundNumberPipe],
   templateUrl: './app.component.html',
   styleUrl: './app.component.scss',
})
export class AppComponent {
   title = UserDefined-pipes';
   val:Number=4.64;
}
```

app.component.html

```
<div>
   <p>
      {{3.4 | roundNumber}}
   </p>
   <hr>
   <p>
      {{ val | roundNumber}}
   </p>
</div>
```

Output

3

5

*****

Forms

- Forms are used to collect the information from the user
- Forms are a powerful way to handle user input and data binding.
- Angular has two types of forms called Template-Driven Forms and reactive forms.

Template-Driven Forms

- It managed primarily within the HTML template using directives like ngModel.
- It is simpler for basic forms with less complex validation.
- It relies on two-way data binding with ngModel.
- The validation attributes (e.g., required, minlength, pattern) are added directly to HTML input elements.

Example :

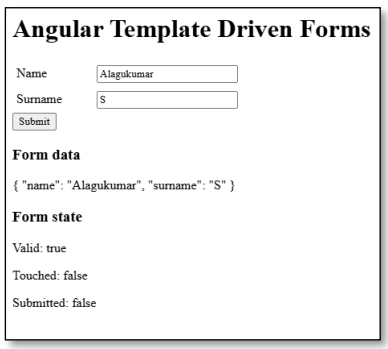
App.component.html

```
<h1>Angular Template Driven Forms</h1>
<form #userForm="ngForm">
          <div>
                    <label for="name">Name</label>
                    <input id="name"
         name="name"
         type="text"
         required
         [(ngModel)]="user.name" />
  </div>
  <div>
    <label for="surname">Surname</label>
    <input id="surname"
         name="surname"
         type="text"
         required
         [(ngModel)]="user.surname" />
  </div>
  <button type="button" (click)="submit()">Submit</button>
</form>
<h3>Form data</h3>
{{user | json}}
<h3>Form state</h3>
<p>Valid: {{userForm.valid}}</p>
<p>Touched:  {{userForm.touched}}</p>
<p>Submitted: {{userForm.submitted}}</p>
```

App.component.cs

```
import { Component } from "@angular/core";
@Component({
  selector: "my-app",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  user: User = {
    name: "Alagukumar",
    surname: "S"
  };
  submit() {
    alert(`Submited with: ${this.user.name} ${this.user.surname}`);
  }
}
export interface User {
  name: string;
  surname: string;
}
```

Output



*****

Data Validation

- In Angular template-driven forms, data validation is achieved using directives and validation classes.
- Angular provides built-in validation attributes and classes to manage form validation effectively.

Validation Classes in Angular

- The validators class, comes under the @angular/forms
- Validator class ensuring that user input in both Reactive Forms and Template-driven Forms meets specified criteria.
- Template-Driven Forms are forms where the validation is applied directly in the HTML template using Angular's built-in directives, like required, minlength, and maxlength.
- Template driven form uses ngModel directive for two-way data binding and needs less code than Reactive Forms.
- In Angular Reactive Forms, form validation is handled within component class using the FormControl, FormGroup, and FormArray classes, along with built-in or user defined validators.

Validators Methods

- **Validators.required**
  It ensures a control has a non-empty value
- **Validators.minLength(length)**
  It validates that the control's value meets a minimum length.
- **Validators.maxLength(length)**
  It validates that the control's value does not exceed a maximum length.
- **Validators.pattern(regex)**
  It validates the control's value against a regular expression.
- **Validators.email**
  It validates that the control's value is a valid email address format.
- **Validators.min(value)**
  It validates that the numeric value is greater than or equal to a specified minimum.
- **Validators.max(value)**
  It validates that the numeric value is less than or equal to a specified maximum.
- **Validators.compose(validators)**
  It combines multiple validator functions into a single validator.

Validation States

- valid: It indicates that the control or form's value meets all defined validation rules.
- invalid: It indicates that the control or form's value does not meet one or more defined validation rules.
- pristine: It indicates that the control or form has not been modified by the user.
- dirty: It indicates that the control or form has been modified by the user.
- untouched: It indicates that the user has not yet interacted with (e.g., focused and blurred) the control.
- touched: It indicates that the user has interacted with (e.g., focused and blurred) the control.
- submitted: It indicates that the form has been submitted.

Example:

App.Component.ts

```
import { Component } from '@angular/core';
import { NgForm } from '@angular/forms';
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
```

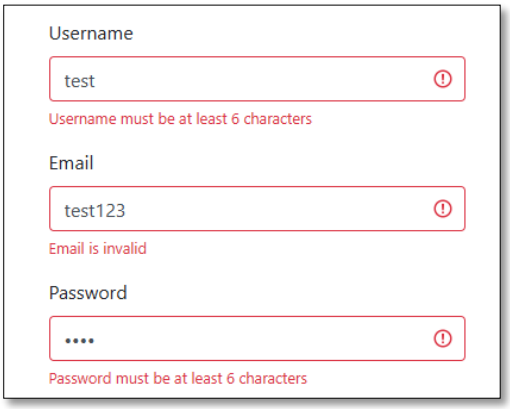
```
export class AppComponent {
  form = {
    username: '',
    email: '',
    password: '',
    confirmPassword: ''
  };
  onSubmit(): void {
    console.log(JSON.stringify(this.form, null, 2));
  }
  onReset(form: NgForm): void {
    form.reset();
  }
}
```

App.componenet.html

```
<div class="register-form">
  <form
    name="form"
    #f="ngForm"
    (ngSubmit)="f.form.valid && onSubmit()"
    novalidate
    [appMatchPassword]="['password', 'confirmPassword']"
  >
    <div class="form-group">
      <label>Username</label>
      <input
        type="text"
        class="form-control"
        name="username"
        [(ngModel)]="form.username"
        required
        minlength="6"
        maxlength="20"
        #username="ngModel"
        [ngClass]="{ 'is-invalid': f.submitted && username.errors }"
      />
      <div *ngIf="f.submitted && username.errors" class="invalid-feedback">
        <div *ngIf="username.errors['required']">Username is required</div>
        <div *ngIf="username.errors['minlength']">
          Username must be at least 6 characters
        </div>
        <div *ngIf="username.errors['maxlength']">
          Username must be at most 20 characters
        </div>
      </div>
    </div>

    <div class="form-group">
      <label>Email</label>
      <input
        type="email"
        class="form-control"
        name="email"
        [(ngModel)]="form.email"
        required
        email
```

```
      #email="ngModel"
      [ngClass]="{ 'is-invalid': f.submitted && email.errors }"
    />
    <div *ngIf="f.submitted && email.errors" class="invalid-feedback">
      <div *ngIf="email.errors['required']">Email is required</div>
      <div *ngIf="email.errors['email']">Email is invalid</div>
    </div>
  </div>

  <div class="form-group">
    <label>Password</label>
    <input
      type="password"
      class="form-control"
      name="password"
      [(ngModel)]="form.password"
      required
      minlength="6"
      maxlength="40"
      #password="ngModel"
      [ngClass]="{ 'is-invalid': f.submitted && password.errors }"
    />
    <div *ngIf="f.submitted && password.errors" class="invalid-feedback">
      <div *ngIf="password.errors['required']">Password is required</div>
      <div *ngIf="password.errors['minlength']">
        Password must be at least 6 characters
      </div>
      <div *ngIf="password.errors['maxlength']">
        Username must not exceed 40 characters
      </div>
    </div>
  </div>
  <div class="form-group">
    <button type="submit" class="btn btn-primary">Register</button>
    <button
      type="button"
      (click)="onReset(f)"
      class="btn btn-warning float-right"  >
      Reset
    </button>
  </div>
 </form>
</div>
```

Output



*****

Reactive Forms

- Reactive Forms use a model-driven approach
- Reactive Forms allows to create forms programmatically using TypeScript
- It provides a robust way to handle dynamic validation, user inputs, and error messages.
- In reactive forms, the model is created in the component class, and the form is updated by the component itself,

Components of Reactive forms

FormGroup

- A collection of FormControl instances.
- It represents the entire form and manages its values and validations.

FormControl

- It represents an individual input field in the form, such as text boxes or checkboxes.

FormArray

- A collection of FormControl or FormGroup instances
- It allows you to manage an array of inputs dynamically.

Validators

- Functions that enforce validation rules on form controls, such as required, minlength, or pattern.
- These components work together to provide a more flexible way of handling forms, giving developers the ability to dynamically change form values, validations, and reactions based on user input.

FormBuilder

- The FormBuilder is a service that simplifies the process of creating and managing forms.
- It provides a cleaner and more concise way to define form controls, groups, and arrays, reducing code.

Create Reactive Forms

It contains following steps

- Import ReactiveFormsModule
- Create Form Model in component class using FormGroup, FormControl & FormArrays
- Create the HTML Form resembling the Form Model.
- Bind the HTML Form to the Form Model

App.html:

```
<div class="container mt-4" style="max-width: 500px;">
  <h2 class="mb-3">Student Registration Form</h2>
  <form [formGroup]="studentForm" (ngSubmit)="onSubmit()">
    <!-- Email -->
    <div class="mb-3">
      <label>Email</label>
      <input type="email" class="form-control" formControlName="email" />
      <div class="text-danger" *ngIf="studentForm.get('email')?.touched &&
studentForm.get('email')?.invalid">
        <small *ngIf="studentForm.get('email')?.errors?.['required']">Email is required</small>
        <small *ngIf="studentForm.get('email')?.errors?.['email']">Enter a valid email</small>
      </div>
    </div>

    <!-- Age -->
    <div class="mb-3">
      <label>Age</label>
      <input type="number" class="form-control" formControlName="age" />
      <div class="text-danger" *ngIf="studentForm.get('age')?.touched &&
studentForm.get('age')?.invalid">
        <small *ngIf="studentForm.get('age')?.errors?.['required']">Age is required</small>
        <small *ngIf="studentForm.get('age')?.errors?.['min']">Minimum age is 18</small>
```

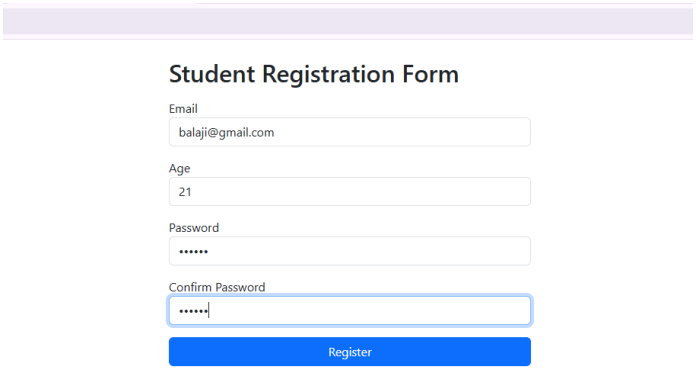
```
      <small *ngIf="studentForm.get('age')?.errors?.['max']">Maximum age is 60</small>
     </div>
   </div>

   <!-- Password -->
   <div class="mb-3">
     <label>Password</label>
     <input type="password" class="form-control" formControlName="password" />
     <div class="text-danger" *ngIf="studentForm.get('password')?.touched &&
studentForm.get('password')?.invalid">
       <small *ngIf="studentForm.get('password')?.errors?.['required']">Password is
required</small>
       <small *ngIf="studentForm.get('password')?.errors?.['minlength']">At least 6
characters</small>
     </div>
   </div>

   <!-- Confirm Password -->
   <div class="mb-3">
     <label>Confirm Password</label>
     <input type="password" class="form-control" formControlName="confirmPassword" />
     <div class="text-danger" *ngIf="studentForm.errors?.['mismatch'] &&
studentForm.get('confirmPassword')?.touched">
       <small>Passwords do not match</small>
     </div>
   </div>
   <!-- Submit -->
   <button type="submit" class="btn btn-primary w-100" [disabled]="studentForm.invalid">
     Register
   </button>
  </form>
</div>
```

app.ts:

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { ReactiveFormsModule,FormBuilder,FormGroup,Validators  } from
'@angular/forms';
import { RouterOutlet } from '@angular/router';
@Component({
  selector: 'app-root',
  imports: [ReactiveFormsModule,CommonModule],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
  protected title = 'reactiveform';
  studentForm: FormGroup;
  constructor(private fb: FormBuilder) {
    this.studentForm = this.fb.group(
      {
        email: ['', [Validators.required, Validators.email]],
        age: ['', [Validators.required, Validators.min(18), Validators.max(60)]],
        password: ['', [Validators.required, Validators.minLength(6)]],
        confirmPassword: ['', Validators.required],
      },
```

```
      { validators: this.passwordMatchValidator }
    );
  }
  // Custom Validator for matching passwords
  passwordMatchValidator(form: FormGroup) {
    const password = form.get('password')?.value;
    const confirmPassword = form.get('confirmPassword')?.value;
    return password === confirmPassword ? null : { mismatch: true };
  }
  onSubmit() {
    if (this.studentForm.valid) {
      console.log('Student Registered:', this.studentForm.value);
      alert('Student Registration Successful!');
      this.studentForm.reset();
    } else {
      this.studentForm.markAllAsTouched();
    }
  }
}
```

Output:



*****

Dynamic Forms

- Dynamic forms allows to create forms that can adapt to changing data models or user inputs.
- It is particularly useful when the structure of the form is not fixed and needs to be generated dynamically based on metadata or configuration.

Adding and Removing elements

Steps to Create a Dynamic Form in Angular

1. Set up your Angular project: Ensure you have Angular CLI installed and create a new project.
2. Import Reactive Forms Module: Add ReactiveFormsModule to your AppModule.
3. Define a Data Model: Create a structure to hold the form field configurations.
4. Generate Form Controls Dynamically: Use FormBuilder to create controls based on the data model.
5. Bind the Form to the Template: Use Angular's template syntax to render the form dynamically.

Steps to Create Dynamic Forms

1. Define Metadata

- Create a JSON or object structure to describe the form fields, types, and validations.

```
const formMetadata = [
  { label: 'Name', type: 'text', name: 'name', required: true },
  { label: 'Email', type: 'email', name: 'email', required: true },
  { label: 'Age', type: 'number', name: 'age', required: false }
];
```

2. Generate Form Controls

- Use the metadata to dynamically create FormControl instances.

```
const formGroup = new FormGroup({});
formMetadata.forEach(field => {
  formGroup.addControl(
    field.name,
    new FormControl('', field.required ? Validators.required : null)
  );
});
```

3. Bind to Template

- Use Angular's template syntax to render the form dynamically

```
<form [formGroup]="formGroup">
  <div *ngFor="let field of formMetadata">
    <label>{{ field.label }}</label>
    <input [type]="field.type" [formControlName]="field.name" />
  </div>
  <button type="submit" [disabled]="formGroup.invalid">Submit</button>
</form>
```

4. Handle Form Submission

- Process the form data in the component.

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

Example :

App.html:

```
<form [formGroup]="form" class="container py-5">
  <div class="d-flex justify-content-between align-items-center mb-3">
    <h4 class="mb-0">Dynamic Form Table</h4>
  </div>
  <div formArrayName="items" class="table-responsive">
    <table class="table table-bordered table-striped align-middle text-center">
      <thead class="table-dark">
        <tr>
          <th>Name</th>
          <th>Age</th>
          <th>Action</th>
        </tr>
      </thead>
      <tbody>
        <tr
          *ngFor="let item of items.controls; let i = index"
          [formGroupName]="i">
        <!-- <td>{{i}}</td> -->
          <td>
            <input
              type="text"
              class="form-control"
              placeholder="Enter name"
              formControlName="name"

            />
             <div
              class="text-danger small mt-3"
              *ngIf="item.get('name')?.invalid && item.get('name')?.touched"
            >
```

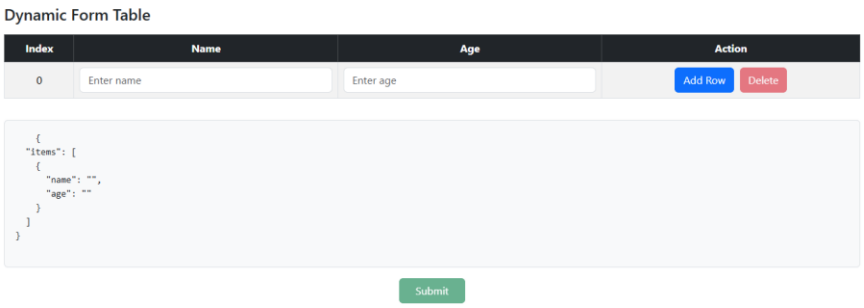
```
            <span *ngIf="item.get('name')?.errors?.['required']">
              * Name is required
            </span>
          </div>
          </td>
         <td>
          <input
            type="number"
            class="form-control"
            placeholder="Enter age"
            formControlName="age"
          />
          <div
            class="text-danger small mt-1"
            *ngIf="item.get('age')?.invalid && item.get('age')?.touched"
          >
            <span *ngIf="item.get('age')?.errors?.['required']">
              * Age is required
            </span>
            <span *ngIf="item.get('age')?.errors?.['min']">
              Age must be greater than 0
            </span>
            <span *ngIf="item.get('age')?.errors?.['max']">
              Age must be less than or equal to 120
            </span>
          </div>
         </td>
         <td class="text-center px-4 py-2 justify-content-between">
  <button class="btn btn-primary" (click)="addItem()">Add Row</button> 
  <button type="button" [disabled]="items.length === 1" class="btn btn-danger"
(click)="deleteItem(i)">
    Delete
  </button>
</td>

       </tr>
     </tbody>
   </table>
  </div>

  <pre class="bg-light p-3 mt-3 border rounded">
    {{ form.value | json }}
  </pre>

  <div class="text-center mt-3">
    <button [disabled]="form.invalid" type="submit" class="btn btn-success px-4"
(click)="submit()">
      Submit
    </button>
  </div>
</form>
```

App.ts:

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { FormsModule,FormArray,FormGroup,FormBuilder, ReactiveFormsModule,
Validators } from '@angular/forms';
```

```
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet,CommonModule,FormsModule,ReactiveFormsModule ],
  templateUrl: './app.html',
  styleUrl: './app.css'
})
export class App {
form: any;
  constructor(private fb: FormBuilder) {
    this.form = this.fb.group({
      items: this.fb.array([
        this.createItem()
      ]),
    });
  }
createItem(): FormGroup {
    return this.fb.group({
      name: ['', Validators.required],
      age: ['',[Validators.required, Validators.min(1), Validators.max(120)]
      ]
    });
  }
  get items() {
    return this.form.get('items') as FormArray;
  }
  deleteItem(index: number) {
    this.items.removeAt(index);
  }
  addItem() {
    this.items.push(
      this.createItem()
    );
  }
  submit() {
     if (this.form.valid) {
      console.log('Form Submitted:', this.form.value);
    } else {
      this.form.markAllAsTouched();
    }
  }
}
```

Output:



*****