Anngular
==========
Anugular is a TypeScript based full-stack web framework for building web and mobile applications. One of the major advantage is that the Angular 8 support for web application that can fit in any screen resolution. Angular application is fully compatible for mobiles, tablets, laptops or desktops. Angular 8 has an excellent user interface library for web developers which contains reusable UI components.

This functionality helps us to create Single Page Applications (SPA). SPA is reactive and fast application. For example, if you have a button in single page and click on the button then the action performs dynamically in the current page without loading the new page from the server. Angular is Typescript based object oriented programming and support features for server side programming as well.

Angular framework is based on four core concepts and they are as follows:

- Components.
- Templates with Data binding and Directives.
- Modules.
- Services and dependency injection

Component
=========
The core of the Angular framework architecture is Angular Component. Angular Component is the building block of every Angular application.
Every angular application is made up of one more Angular Component.
It is basically a plain JavaScript / Typescript class along with a HTML template and an associated name.

The HTML template can access the data from its corresponding JavaScript / Typescript class.
Component's HTML template may include other component using its selector's value (name).
The Angular Component may have an optional CSS Styles associated it and the
HTML template may access the CSS Styles as well.

Template
========

Template is basically a super set of HTML. Template includes all the features of HTML and
provides additional functionality to bind the component data into the HTML and to dynamically generate HTML DOM elements.
The core concept of the template can be categorised into two items and they are as follows:

Data binding

Used to bind the data from the component to the template.
{{ title }}
Here, title is a property in AppComponent and it is bind to template using
Interpolation.

Directives

Used to include logic as well as enable creation of complex HTML DOM elements.
<p *ngIf="canShow">

Modules
==========
Angular Module is basically a collection of related features / functionality. Angular Module groups multiple components and services under a single context.
For example, animations related functionality can be grouped into single module and Angular already provides a module for the animation related functionality, BrowserAnimationModule module.

An Angular application can have any number of modules but only one module can be set as root module, which will bootstrap the application and then call other modules as and when necessary. A module can be configured to access functionality from other module as well. In short, compon

- declarations: option is used to include components into the AppModule module.
- imports: option is used to import other modules into the AppModule module.
- providers: option is used to include the services for the AppModule module.
- bootstrap: option is used to set the root component of the AppModule module.

Services
==========
Services are plain Typescript / JavaScript class providing a very specific functionality. Services will do a single task and do it best. The main purpose of the service is reusability. Instead of writing a functionality inside a component, separating it into a service will make it useable in other component as well.

Also, Services enables the developer to organise the business logic of the application. Basically, component uses services to do its own job. Dependency Injection is used to properly initialise the service in the component so that the component can access the services as and when necessary without any setup.

1. DATA BINDING
==========
Data binding deals with how to bind your data from component to HTML DOM elements (Templates). We can easily interact with application without worrying about how to insert your data. We can make connections in two different ways one way and two-way binding.

One-way data binding
==========
One-way data binding is a one-way interaction between component and its template. If you perform any changes in your component, then it will reflect the HTML elements. It supports the following types:

String interpolation

In general, String interpolation is the process of formatting or manipulating strings. In Angular, Interpolation is used to display data from component to view (DOM). It is

denoted by the expression of {{ }} and also known as mustache syntax.

Let’s create a simple string property in component and bind the data to view.
Add the below code in test.component.ts file as follows:

```
export class TestComponent implements OnInit {
 appName = "My first app in Angular 8";
}
```

```
<h1>{{appName}}</h1>
```

Event binding

Events are actions like mouse click, double click, hover or any keyboard and mouse actions. If a user interacts with an application and performs some actions, then event will be raised. It is denoted by either parenthesis () or on-. We have different ways to bind an event to DOM element. Let’s understand one by one in brief.

```
export class TestComponent {
showData($event: any){
 console.log("button is clicked!");
 if($event) {
 console.log($event.target);
 console.log($event.target.value);
 }
 }
}
```

```
<h2>Event Binding</h2>
<button (click)="showData($event)">Click here</button>
```

alternatively :

```
<h2>Event Binding</h2>
<button on-click = "showData()">Click here</button>
```

Property Binding

Property binding is used to bind the data from property of a component to DOM elements. It is denoted by [].

```
<input [disabled] = "valueDisabled" type="text" placeholder="Enter text">
```

```
 valueDisabled = true;
```

Attribute Binding

Attribute binding in Angular allows you to bind values to standard HTML attributes and custom attributes that don't have corresponding DOM properties.
While Angular usually prefers property binding for interacting with DOM properties, attribute binding can be useful when you need to work with attributes directly.
To perform attribute binding in Angular, you use square brackets [...] in your template, just like with property binding.

Here's how to use attribute binding:

```
<a [href]="url">Visit our website</a>

<button [attr.disabled]="isDisabled">Click me</button>

url = "www.javatpoint.com";
isDisabled = true;
```

Class binding

Class binding is used to bind the data from component to HTML class property. The syntax is as follows:

```
  MyClass = "red";

  MyStyles  = true; // if value is true, blue color will be added to text, if it is false , blur color will not be added to text.


      .red{
            color: red;
      }

      .blue{
            color: blue;
      }


            <!-- Class Binding -->
      <p [class] = "MyClass"> Lorem ipsum dolor sit amet consectetur adipisicing elit.
            Delectus ipsa fugiat sunt quis hic, explicabo nesciunt facilis aperiam ex quia voluptate voluptatibus doloribus possimus.
            Earum nisi voluptatibus vitae dolorum commodi.
      </p>

      <p [class.blue]="MyStyles">Lorem ipsum, dolor sit amet consectetur adipisicing elit.
            Nemo at praesentium dolor deleniti, neque ducimus, ipsa consectetur molestiae assumenda possimus omnis eum iste culpa modi similique porro quae unde delectus?
      </p>
```

Two-way data binding

==========

Two-way data binding is a two-way interaction, data flows in both ways (from component to views and views to component). Simple example is ngModel. If you do any changes in your property (or model) then, it reflects in your view and vice versa. It is the combination of property and event binding.

```
  fullName : string | undefined = "Nani Babu Pallapu";
```

<input type="text" [class]="MyClass" [(ngModel)] = "fullName" /> <!--we need to import FormsModule to use ngModel -->
<h3 [class.blue] = "MyClass" > Hi I am {{fullName}}. Welcome to my home !</h3>

2. Angular-Directives
==========
The Angular 8 directives are used to manipulate the DOM.
By using Angular directives, you can change the appearance, behavior or a layout of a DOM element. It also helps you to extend HTML.

Angular 8 directives can be classified in 3 categories based on how they behave:

Component Directives
Structural Directives
Attribute Directives

1) Component Directives: Component can be used as directives. Every component has Input and Output option to pass between component and its parent HTML elements.
2) Structural Directives: Structural directives start with a * sign. These directives are used to manipulate and change the structure of the DOM elements.
For example, *ngIf directive, *ngSwitch directive, and *ngFor directive.

*ngIf Directive: gIf directive is used to display or hide data in your application based on the condition becomes true or false. We can add this to any tag in your template.
*ngSwitch Directive: The *ngSwitch allows us to Add/Remove DOM Element. It is similar to switch statement of C#.
*ngFor Directive: The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).

3) Attribute Directives: Attribute directives are used to change the look and behavior of the DOM elements. For example: ngClass directive, and ngStyle directive etc.

ngClass Directive: The ngClass directive is used to add or remove CSS classes to an HTML element.
ngStyle Directive: The ngStyle directive facilitates you to modify the style of an HTML element using the expression. You can also use ngStyle directive to dynamically change the style of your HTML element.

Structural Directives example:
==========

```
check : boolean = true;

loggedAs : string  = "user";
```

```
<div>
  <h2>Structural Directives</h2>
  <h3>*ngIf Demo</h3>
  <p *ngIf="check">Welcome to my computer world!</p>

  <h3>*ngIf - Else Demo</h3>
  <div>
```

```
    <p *ngIf="check; else logOut">You are logged into Computer!</p>
    <ng-template #logOut>You are logged out!</ng-template>
  </div>

  <h3>*ngFor Demo </h3>
  <div>
    <h4> Student List </h4>
    <div *ngFor="let student of studentList">
      <ul style="list-style-type:none;">
        <h5>Student {{student.id}} Details</h5>
        <div>
          <li>Student Id:{{student.id}}</li>
          <li>Student Name : {{student.name}}</li>
          <li>Student Branch : {{student.branch}}</li>
          <li>Student Pass/Failed :<div *ngIf="student.pass; else failed"> Passed</div>
          </li>
          <ng-template #failed>
            <p>Failed</p>
          </ng-template>
        </div>
      </ul>
    </div>
  </div>

  <h3>*ngSwitchCase-Demo</h3>
  <div>
    <ul style="list-style-type:none;" [ngSwitch]="loggedAs">
      <li *ngSwitchCase="'admin'"> You logged-in as Admin</li>
      <li *ngSwitchCase="'user'">You logged-in as a User</li>
      <li *ngSwitchCase="'sna'">You logged-in as SNA-team member</li>
      <li *ngSwitchDefault>Unknown user logged-in</li>
    </ul>
  </div>
</div>


Attribute Directives example:
==============================

in css :

.highlight
{
 color: red;
}

in html :
<div>
  <h3 [ngStyle] = "{'color':'red'}">2. Built-in Attribute directives</h3>
  <div>
    <div> <h4>i.ngStyle-directive</h4>
      <p [ngStyle]="{'color': 'green', 'font-size': '14px'}">
        paragraph style is applied using ngStyle
      </p>
    </div>

    <div class="container"> <h4>ii. ngClass-directive</h4>
      <br />
      <div *ngFor="let student of studentList" [ngClass]="{'highlight':student.name === 'divya'}">
        {{ student.name }}
```

```
        </div>
      </div>

      <div> <h4>iii.ngModel-directive</h4>
        <input type="text" [(ngModel)] = "fullName" placeholder="enter your name" id="name">
        <p [ngStyle]="{'color':'dark blue','border':'2px solid blue','margin':'50px','padding':'20px','font-
size':'20px'}">{{fullName}}</p>
      </div>

      <div><h4>iv.Custom-attribute-directive</h4>
        <p appCustomDirectiveExample >Nani Pallapu</p> <!-- 'appCustomDirectiveExample' is custom directive and it
should not be enclosed with brackets-->
      </div>
   </div>
</div>
```

Component Directives
====================

In Angular, you can use the @Input and @Output decorators to pass data into and emit events from components. These decorators are essential for building parent-child communication between components.
Here's how you can use @Input and @Output decorators in Angular components:

Using @Input Decorator:
---

1. @Input Decorator Definition: @Input is used to pass data into a component from its parent component.
Parent Component:

2. In the parent component template or TypeScript file, you can bind data to a child component's input property using property binding.

Example (Parent Component HTML):

```
<app-child [inputData]="parentData"></app-child>
```

Example (Parent Component TypeScript):

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

@Component({
  selector: 'app-parent',
  template: `
    <app-child [inputData]="parentData"></app-child>
  `,
})
export class ParentComponent {
  parentData = 'Data from parent';
}
```

In this example, the inputData property in the ChildComponent is decorated with @Input(), allowing it to receive data from the parent component.

Using @Output Decorator:
---

1. @Output Decorator Definition: @Output is used to emit events from a child component to its parent component.

2. Child Component: In the child component TypeScript file, you define an output property using the @Output decorator. You also create an EventEmitter to emit events.

Example (Child Component TypeScript):

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

@Component({
  selector: 'app-child',
  template: '<button (click)="emitEvent()">Emit Event</button>',
})
export class ChildComponent {
  @Output() myEvent = new EventEmitter<string>();

  emitEvent() {
    this.myEvent.emit('Event data from child');
  }
}
```

In this example, we've created an EventEmitter named myEvent and emit an event with data when the button is clicked.

3.Parent Component: In the parent component HTML template, you can listen for the event emitted by the child component and handle it using an event binding.

Example (Parent Component HTML):

```
<app-child (myEvent)="handleEvent($event)"></app-child>
```

Example (Parent Component TypeScript):

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

@Component({
  selector: 'app-parent',
  template: `
    <app-child (myEvent)="handleEvent($event)"></app-child>
    <p>{{ eventData }}</p>
  `,
})
export class ParentComponent {
  eventData: string;

  handleEvent(data: string) {
    this.eventData = data;
  }
}
```

In this example, the parent component listens for the myEvent event emitted by the child component and handles it in the handleEvent method.

These are the basic concepts of using @Input and @Output decorators in Angular components to facilitate

communication between parent and child components

3. Angular Pipes

Pipes are referred as filters. It helps to transform data and manage data within interpolation, denoted by {{ | }}. It accepts data, arrays, integers and strings as inputs which are separated by '|' symbol.

Built-in Pipes

- Lowercasepipe
- Uppercasepipe
- Datepipe
- Currencypipe
- Jsonpipe
- Percentpipe
- Decimalpipe
- Slicepipe

app.component.ts

```
  title = 'angular-pipes-examples';

  fullName : string = 'Nani Babu';

  todayDate  = new Date();

  sampleJsonData = {
    name : 'Nani Babu',
    age : '25',
    job : 'Software Engineer',
    address : {
      village : 'uppaluru',
      city : 'Vijayawada',
      district : 'Krishna District',
      state : 'Andhra Pradesh',
      pincode : '521151'
    }
  }

  days = ['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday'];
```

squareroot.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'squareroot' // it should be used in template file
})
export class SquarerootPipe implements PipeTransform {

  transform(value: number): number {
    return Math.sqrt(value);
  }
```

```
}
```

digitscount.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'digitscount'
})
export class DigitscountPipe implements PipeTransform {

  transform(value: number): number {
    return value.toString().length;
  }

}
```

app.component.html

```
<div [ngStyle]="{'text-align':'center'}">
  <h2>{{title}}</h2>

  <h3>1.Uppercase Pipe</h3>
  <p>{{fullName | uppercase}}</p><br>

  <h3>2.Lowercase Pipe</h3>
  <p>{{fullName | lowercase}}</p><br>

  <h3>3.Currency Pipe</h3>
  <p>{{256.66 | currency:"USD"}}</p> <!-- $256.66 -->
  <p>{{256.66 | currency:"INR" }}</p> <!-- ₹256.66 --> <br>

  <h3>4.Date pipe</h3>
  <p>{{todayDate}}</p> <!-- Mon Sep 18 2023 23:04:28 GMT+0530 (India Standard Time)  -->
  <p>{{todayDate | date }}</p> <!-- Sep 18, 2023  -->
  <p>{{todayDate | date:'d/M/y' }}</p> <!-- 18/9/2023 -->
  <p>{{todayDate | date:'shortDate' }}</p> <!-- 9/18/23  -->
  <p>{{todayDate | date:'fullDate' }}</p> <!-- Monday, September 18, 2023  -->
  <p>{{todayDate | date:'h:mm' }}</p> <br><!-- 11:05  -->

  <h3>5.Decimal Pipe</h3>
  <p>{{ 454.78787814 | number: '3.4-6' }}</p> <br>

  <h3>6.Json Pipe</h3>
  <p>{{sampleJsonData | json}}</p><br>

  <h3>7.Percent Pipe</h3>
  <p>{{00.54565 | percent }}</p><br>

  <h3>8.Slice Pipe</h3>
  <p>{{days | slice:2:4 }}</p><br> <!-- Wednesday,Thursday , it's printing based on index position in Array  -->

  <h3>9.Custom Pipes</h3>
  <p>{{625 | squareroot }}</p> <!-- 25 –>
  <p>{{625 | digitscount }}</p> <!– 3 digts -->
</div>
```

4. SERVICES AND DEPENDENCY INJECTION

Services :

Services provides specific functionality in an Angular application. In a given Angular application, there may be one or more services can be used. Similarly, an Angular component may depend on one or more services.

Dependency Injection :

Angular services may depend on another services to work properly. Dependency resolution is one of the complex and time consuming activity in developing any application. To reduce the complexity, Angular provides Dependency Injection pattern as one of the core concept.

An Angular service is plain Typescript class having one or more methods (functionality) along with @Injectable decorator. It enables the normal Typescript class to be used as service in Angular application.

```
import { Injectable } from '@angular/core';
@Injectable()
export class DebugService {
 constructor() { }
}
```

Here, @Injectable decorator converts a plain Typescript class into Angular service.

Observables

In Angular, observables are a core part of the RxJS (Reactive Extensions for JavaScript) library, which is used to manage asynchronous operations and handle data streams.

Observables are a powerful way to work with data and events in a reactive, non-blocking manner.

Here are some key concepts related to observables in Angular:

Observable:

An observable is a representation of a stream of data or events that can be observed over time. It can emit multiple values over time, including asynchronous events like HTTP requests, user inputs, and more.

Observer:

An observer is an object that listens to the emitted values from an observable. It defines methods like next, error, and complete to handle the data emitted by the observable.

Subscription:

A subscription is the mechanism used to connect an observer to an observable.

It represents the ongoing execution of an observable and can be used to cancel or unsubscribe from the observable when it's no longer needed. This helps prevent memory leaks.

HTTP , Observables and RxJS

---

1. HTTP Get request from EmployeeService.
2. Receive the observables and cast it into an Employee Array.
3. Subscribe to the observables from EmployeeList and EmployeeDetails.
4. Assign the Employee Array to the local variables.

RxJs : Reactive Extension for JavaScript and External Library to work with Observables.

```
export class StudentService implements OnInit {

  // data location or you can use website url.
  private _url : string = "/assets/data/student.json";


  constructor(private http : HttpClient) {

   }
  ngOnInit() {

  }

 //1. we are making request using 'this.http.get()' by passing 'this._url' argument to fetch data from website(or file).
 //2. Receive the observables and cast it into an Student Array.
 getEmployees() : Observable<Student[]>{
    return this.http.get<Student[]>(this._url);
  }
}



export class StudentListComponent implements OnInit {

  studentList!: Student[];
  show : boolean = true
  isDisable : boolean = false;

  constructor(private studentService : StudentService) {

  }

// 3. Subscribe to the observables from EmployeeList.
// 4. Assign the Employee Array data(data) to the local variables(this.studentList).
  ngOnInit() {
    this.studentService.getEmployees().subscribe(data => this.studentList = data)
  }
```

```
  showStudents(){
    this.show = !this.show;
  }
}
```

ANGULAR ROUTING
==========

1. Generate a project with routing option.
2. Generate a departmentList and employeeList components.
3. Configure the routes.
4. Add buttons and use directives to navigate.

app.routing.module.ts file

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { EmployeeListComponent } from './employee-list/employee-list.component';
import { DepartmentsComponent } from './departments/departments.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

const routes: Routes = [
  {path : '', redirectTo : '/departments', pathMatch:'full'}, // renders if given path is empty
  {path : 'employees', component : EmployeeListComponent}, // renders employee component
  {path : 'departments', component : DepartmentsComponent}, // renders departments component
  {path : "**", component : PageNotFoundComponent} // // renders page-not-found component if given url does not match
above paths
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {

 }

 export const routingComponents = [EmployeeListComponent, DepartmentsComponent, PageNotFoundComponent];
```

app.component.html file

```
<router-outlet></router-outlet>
<!-- routers will be staarted here-->

<nav>
  <a routerLink = "/employees" routerLinkActive = "activating">Employee-List</a>
  <a routerLink = "/departments" routerLinkActive = "activating" >Department-List</a>
</nav>
```

RouterParameters
==========

employee-list component files

```
  public employees!: any[];
  constructor(private router : Router) {

  }
  ngOnInit() {
    this.employees = [{ "id": 1, "name": "Nani", "department": "JAVA", "package": "10LPA" },
    { "id": 2, "name": "Pinky", "department": "PYTHON", "package": "12LPA" },
    { "id": 3, "name": "divya", "department": "DEVOPS", "package": "14LPA" },
    { "id": 4, "name": "rajesh", "department": ".NET", "package": "18LPA"}
    ];
  }

  onSelect(employee : any){
    this.router.navigate(['/employees',employee.id])   // passing the parameter id
  }

  <div>
    <ul  style="list-style-type: none;" (click) = "onSelect(employee)" *ngFor="let employee of employees">
        <li>{{employee.id}}.{{employee.name}}</li>
    </ul>
</div>
```

employee-details component files

```
  public employees! : any[];
  public employeeId! : number;
  constructor(private route : ActivatedRoute){

  }

  ngOnInit() {

    this.employees = [{ "id": 1, "name": "Nani", "department": "JAVA", "package": "10LPA" },
    { "id": 2, "name": "Pinky", "department": "PYTHON", "package": "12LPA" },
    { "id": 3, "name": "divya", "department": "DEVOPS", "package": "14LPA" },
    { "id": 4, "name": "rajesh", "department": ".NET", "package": "18LPA"}
    ];

    let id = parseInt(this.route.snapshot.paramMap.get('id')!);
    this.employeeId = id;
  }



<h3>Employee-Details</h3>
<div>
    <ul style="list-style-type: none;"  *ngFor="let employee of employees" >
        <li *ngIf="employee.id == employeeId">{{employee.id}} {{employee.name}}  {{employee.department}}
{{employee.package}}</li>
    </ul>
</div>
```

ParamMap Observables

```
<h3>Employee-Details</h3>
<div>
   <ul style="list-style-type: none;"  *ngFor="let employee of employees" >
      <li *ngIf="employee.id == employeeId">{{employee.id}} {{employee.name}}  {{employee.department}}
{{employee.package}}</li>
   </ul>
   <a (click)="goPrevious()">Previous</a>
   <a (click)="goNext()">Next</a>
   <br><br>
</div>
```

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap, Router } from '@angular/router';

@Component({
  selector: 'app-employee-details',
  templateUrl: './employee-details.component.html',
  styleUrls: ['./employee-details.component.scss']
})
export class EmployeeDetailsComponent implements OnInit{

  public employees! : any[];
  public employeeId! : number;
  constructor(private activatedRoute : ActivatedRoute, private router : Router){

  }

  ngOnInit() {

    this.employees = [{ "id": 1, "name": "Nani", "department": "JAVA", "package": "10LPA" },
    { "id": 2, "name": "Pinky", "department": "PYTHON", "package": "12LPA" },
    { "id": 3, "name": "divya", "department": "DEVOPS", "package": "14LPA" },
    { "id": 4, "name": "rajesh", "department": ".NET", "package": "18LPA"}
    ];

    /*

    //When we navigate back to the same component , Angular simply reuses the component. At that time angular does
not call ngOnit() method.
    // that is why we use paramMap observables instead of snapshot.

    let id = parseInt(this.activatedRoute.snapshot.paramMap.get('id')!);
    this.employeeId = id;

    */

    this.activatedRoute.paramMap.subscribe((params: ParamMap) =>{
      let id = parseInt(params.get('id')!);
      this.employeeId = id;
    });
```

```
  }

  goPrevious(){
    let previousId = this.employeeId-1;
    this.router.navigate(['/employees',previousId]);
  }

  goNext(){
    let nextId = this.employeeId+1;
    this.router.navigate(['/employees', nextId]);
  }

}
```

Child Routes
==========
app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { EmployeeListComponent } from './employee-list/employee-list.component';
import { DepartmentsComponent } from './departments/departments.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
import { EmployeeDetailsComponent } from './employee-details/employee-details.component';
import { EmployeeOverviewComponent } from './employee-overview/employee-overview.component';
import { EmployeeContactComponent } from './employee-contact/employee-contact.component';

const routes: Routes = [
  { path: '', redirectTo: '/departments', pathMatch: 'full' }, // renders if given path is empty

  { path: 'employees', component: EmployeeListComponent }, // renders employee component

  // renders employee-details component
  {

    path: 'employees/:id',
    component: EmployeeDetailsComponent,
    // child routes
    children :[
      {path : 'employee-overview', component : EmployeeOverviewComponent},
      {path : 'employee-contact', component : EmployeeContactComponent}
    ]

  },
  { path: 'departments', component: DepartmentsComponent }, // renders departments component
  { path: "**", component: PageNotFoundComponent } // // renders page-not-found component if given url does not match
above paths
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {
```

```
}

export const routingComponents = [EmployeeListComponent, DepartmentsComponent, PageNotFoundComponent,
EmployeeOverviewComponent, EmployeeContactComponent];
```

employee.details.component.ts

```
  // optional route parameters, it will fetch the employeeId and navigate back to employees
  goBack(){
    let selectedId = this.employeeId ? this.employeeId : null;
    this.router.navigate(['/employees',{id:selectedId}]);
  }

 // child routings
  showOverview(){
    this.router.navigate(['employee-overview'],{relativeTo: this.activatedRoute});
  }

  showContact(){
    this.router.navigate(['employee-contact'],{relativeTo : this.activatedRoute});
  }
```

employee.list.component.ts

```
  ngOnInit() {
    this.employees = [{ "id": 1, "name": "Nani", "department": "JAVA", "package": "10LPA" },
    { "id": 2, "name": "Pinky", "department": "PYTHON", "package": "12LPA" },
    { "id": 3, "name": "divya", "department": "DEVOPS", "package": "14LPA" },
    { "id": 4, "name": "rajesh", "department": ".NET", "package": "18LPA"}
    ];

    // optional route parameters
    this.activatedRoute.paramMap.subscribe((params : ParamMap)=>{
      let id = parseInt(params.get('id')!);
      this.selectedId = id;

    });
  }

  onSelect(employee : any){
    this.router.navigate(['/employees',employee.id])   // passing the parameter id
  }

  // optional route parameters
  isSelected(employee : any){
    return employee.id == this.selectedId;
  }
```

employee.list.component.html

```
  <p>employee-list works!</p>
<div>
    <ul  style="list-style-type: none;" (click) = "onSelect(employee)" [class.selected]="isSelected(employee)" *ngFor="let
employee of employees">
        <li>{{employee.id}}.{{employee.name}}</li>
    </ul>
</div>
```

employee-details.component.html

```
        <router-outlet></router-outlet> <!-- Manditory for child routes template -->


        <a (click)="showOverview()">Employee-Overview</a>
    <a (click)="showContact()">Employee-Contact</a>

    <a (click)="goBack()">Back</a> <!-- optional router parameters , it will take back to employee list and highlight
selected employee -->
    <br><br><br><br>
```

ANGULAR-FORMS
==========

Forms are used to handle user input data. Angular 8 supports two types of forms. They are Template driven forms and Reactive forms. This section explains about Angular 8 forms in detail.

How does it work?
---

1. The component template contains html to collect user data
2. The component Class handles data-binding.
3. Collected data is sent to the server through Services.

Template -------> Class -------> Service -------> Server
Collect Data Bind Data Send data Receive data.

Template driven forms(TDF)
==========
Template driven forms is created using directives in the template. It is mainly used for creating a simple form application. Let's understand how to create template driven forms in brief.

--> We heavily rely on two-way data binding. We don't have to keep tract of input field values and react to change the input values.Angular takes care of that with ngModel directive.
--> Two Data-Binding with ngModel.
--> Bulky Html code and minimal component code.
--> Angular also provides the ngForm directive which along with ngModel directive automatically tracks the forms and forms elements state and validity.

--> Drawback of TDF approach is Unit-Testing is a challange that means forms validation logic can not be unit-tested. Only way to test the logic is to run into test with a browser.

--> Another drawback is Readablity decreases with complex forms and validations.

--> Suitable for only simple applications. For complex forms and validations where unit-testing is necessary we should go with Reactive Forms.

Data-Binding in Template Driven Forms There are 3 directives

i. ngForm - export the form data

ii. ngModel - track of each element data

iii.ngModelGroup - In Angular, ngModelGroup is a directive used to group together multiple <input> elements that use the ngModel directive for two-way data binding.

This grouping can be helpful when you have a form with nested structures or when you want to manage related form controls in a structured manner.

Syntax:

[(ngModel)]="person.name" --> it will fetch predefined values from object and update them with user defined values(modified values)

name="age" --> it will give name to json object for that particular value.

```
<form #myForm="ngForm">
  <div ngModelGroup="person">
    <input name="name" [(ngModel)]="person.name" placeholder="Name">
    <input name="age" [(ngModel)]="person.age" placeholder="Age">
  </div>
</form>
```

Validation and displaying Error messages

```
//required #name="ngModel" [class.is-invalid]="name.invalid && name.touched" --> it will give validation that means name should not be empty once it is touched
   <div class="form-group">
     <label>Name</label>
     <input type="text" required #name="ngModel" [class.is-invalid]="name.invalid && name.touched" class="form-control" name="userName" placeholder="enter name" [(ngModel)]="student.name">
     <small [ngStyle]="{'color':'red'}" [class.d-none]="name.valid || name.untouched">Name is required</small><!-- it means : don't show this error message if it is valid or touched -->
   </div>

//#phone="ngModel" pattern="^\d{10}$" [class.is-invalid]="phone.invalid && phone.touched" –> it will give pattern matching validation numbers should not empty or exceed 10 numbers.
   <div class="form-group">
     <label>Phone Number</label>
     <input type="number"  #phone="ngModel" pattern="^\d{10}$" [class.is-invalid]="phone.invalid && phone.touched" class="form-control"  name="phone" placeholder="enter phone number" [(ngModel)]="student.phone">
     <small [ngStyle]="{'color':'red'}" [class.d-none]="phone.valid || phone.untouched">Invalid Phone Number</small> <!-- it means : don't show this error message if it is valid or touched –>

   </div><br>

     <div class="form-group">
     <label>Phone Number</label>
     <input type="number" required  #phone="ngModel" pattern="^\d{10}$" [class.is-invalid]="phone.invalid &&
```

```
phone.touched" class="form-control"  name="phone" placeholder="enter phone number" [(ngModel)]="student.phone">
    <!-- <small [ngStyle]="{'color':'red'}" [class.d-none]="phone.valid || phone.untouched">Invalid phone number</small> it means : don't show this error message if it is valid or touched -->
    <div *ngIf="phone.errors && (phone.invalid || phone.untouched)"> <!-- displayed invidual errors based on error type -->
      <small class="text-danger" *ngIf="phone.errors?.['required']">phone number is required</small>
      <small class="text-danger" *ngIf="phone.errors?.['pattern']">phone number must be 10 digits</small>
    </div>
  </div><br>



    validation and displaying for select tags:
    ----------------------------------

    -->We added custom-validation for selectors, #topic reference variable has been passed to validateTopic(topic.value) and if topicError=true and topic is touched, then we apply class.is-invalid and display error message.
    -->If topicHasError is false and topic is untouched, we are applying validation through class.d-none class.

    <div class="form-group">
    <select (blur)="validateTopic(topic.value)" (change)="validateTopic(topic.value)" #topic="ngModel" [class.is-invalid]="topicHasError && topic.touched" class="custom-select"  name="course" [(ngModel)]="student.topic">
      <option  value="default">I am interested in </option>
      <option *ngFor="let topic of topics" >{{topic}}</option>
    </select>
    <div>
      <small class="text-danger" [class.d-none]="!topicHasError || topic.untouched">please choose a topic</small>
    </div>
  </div><br>


    export class AppComponent {
      title = 'Template Driven Forms';

      topicHasError: boolean = true;

      public topics = ['Python', 'Java', 'Devops'];

      public student = new Student('NaniBabu', 'nanipallapu369@gmail.com', 9392590089, 'default', 'morning', true);

      validateTopic(value: string) {
          if (value == 'default') {
            this.topicHasError = true;
          } else {
            this.topicHasError = false;
          }
      }
    }


    form validation
    ---------------

    --> studentForm.form.valid will be set as true, if form has an error or invalid then Disable the submit the button.
    --> studentForm.form.valid will be set as false, if form don't have an error or is not invalid then Enable the submit the button.

      <form #studentForm="ngForm" studentForm.form.valid>
```

```
            <div class="form-group">
              <label>Name</label>
              <input type="text" required #name="ngModel" [class.is-invalid]="name.invalid && name.touched" class="form-
control" name="userName" placeholder="enter name" [(ngModel)]="student.name">
              <small [ngStyle]="{'color':'red'}" [class.d-none]="name.valid || name.untouched">Name is required</small><!--
it means : don't show this error message if it is valid or touched -->
            </div>

            <div class="form-group">
              <select (blur)="validateTopic(topic.value)" (change)="validateTopic(topic.value)" #topic="ngModel" [class.is-
invalid]="topicHasError && topic.touched" class="custom-select"  name="course" [(ngModel)]="student.topic">
                    <option  value="default">I am interested in </option>
                    <option *ngFor="let topic of topics" >{{topic}}</option>
              </select>
            <div>
      <small class="text-danger" [class.d-none]="!topicHasError || topic.untouched">please choose a topic</small>


    </div>
        <div>
            <!-- disable the Submit button if studentForm has an error or topic has error -->
            <button [disabled]="studentForm.form.invalid || topicHasError" class="btn btn-primary" type="submit">Submit</
button>
            </div>
        </form>
```

Submitting form

---

1. novalidate - this will prevent browser validation for kicking in when we click on submit button

2. Bind to (ngSubmit) event which gets which gets emmitted when Submit button is clicked.

3. create a service class and go to the service class and import HttpClient and inject it in the constructor.

4. In Service class , crate a property called blank _url and create a enroll(student : Student) method with argument Student to make the post request.

5. With in the body of enroll() method, we make our post request this.http.post<any>(this._url, student).

6. The post request will return response as observable from service class and we need to subscribe to the observable in app.component.ts

7. create a server_dummy_folder folder and initialize the package.json file(npm init –yes --> this command will create package.json file)

8. install dependencies(throgh this command --> npm install --save express body-parser cors)
   express –> webserver
   body-parser --> middle ware to handle form data
   cors --> is package to make a request across different ports.
9. With in server_dummy_folder create a file called server.js and add below configurations.

```
var express = require("express")
var cors = require('cors')
var app = express()

app.use(cors());

var bodyParser = require("body-parser");
```

```
app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());
var HTTP_PORT = 8008

app.listen(HTTP_PORT, () => {
    console.log("Server running on port %PORT%".replace("%PORT%", HTTP_PORT))
});

app.get("/", (req, res, next) => {
    res.json({ "message": "Ok" })
});


app.post("/enroll", function(req, res) {

    console.log(req.body);
    res.status(200).send({"Message":"Data received"});
});
```

10. run this command --> node server
    it will show servering is running

11. paste this public url : string = 'http://localhost:8008/enroll'; in service component file.

12. Fill the form and submit it. open the console check it and open the server terminal , you will be able to see response data(student data).

Note: you can disable through the form *ngIf once you submit the form.

TEMPLATE-DRIVEN-FORMS EXAMPLE :
==========

index.html file for bootstrap:

```
<!doctype html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>FormsATemplateDrivenForms</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet"
    integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN"
crossorigin="anonymous">
</head>

<body>

  <app-root></app-root>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.min.js"
    integrity="sha384-BBtl+eGJRgqQAUMxJ7pMwbEyER4l1g+O15P+16Ep7Q9Q+zqX6gSbd85u4mG4QzX+"
```

```
        crossorigin="anonymous"></script>

    </body>

    </html>
```

app.component.html:

```
    <div [ngStyle]="{'text-align':'center'}">
      <h3>{{title}}</h3>
    </div>

    <!-- template reference variable and assing 'ngForm' string, so that ngForm directive exports itself as the string
ngForm -->
    <!-- 'name' attribute should be used if we want to display with in json format-->
    <div class="container-fluid">
      <h4>Student Enrollment Form</h4>

      <form #studentForm="ngForm" *ngIf ="!submitted; else displayMsg" (ngSubmit)="onSubmitt()"
studentForm.form.valid novalidate>


        <!-- user defined values in json format -->
        {{studentForm.value | json}}

        <hr /> <!-- horizantal line for separation-->

        <!-- it will show predefined values in json format because in .ts file user has already assingned predefined
values through constructor -->
        {{student | json}}

        <hr /> <!-- horizantal line for separation-->

        <!-- checking the form is valid or invalid -->
        <!-- {{studentForm.form.valid}} -->


        <div class="form-group">
          <label>Name</label>
          <input type="text" required #name="ngModel" [class.is-invalid]="name.invalid && name.touched"
class="form-control" name="userName" placeholder="enter name" [(ngModel)]="student.name">
          <small [ngStyle]="{'color':'red'}" [class.d-none]="name.valid || name.untouched">Name is required</
small><!-- it means : don't show this error message if it is valid or touched –>
        </div>

        <div class="form-group">
          <label>Email</label>
          <input type="email" class="form-control" name="userEmail" placeholder="enter
email" [(ngModel)]="student.mail">
        </div><br>

        <div class="form-group">
          <label>Phone Number</label>
          <input type="number" required  #phone="ngModel" pattern="^\d{10}$" [class.is-invalid]="phone.invalid
&& phone.touched" class="form-control"  name="phone" placeholder="enter phone
number" [(ngModel)]="student.phone">
          <!-- <small [ngStyle]="{'color':'red'}" [class.d-none]="phone.valid || phone.untouched">Invalid phone
number</small> it means : don't show this error message if it is valid or touched -->
```

```
                    <div *ngIf="phone.errors && (phone.invalid || phone.untouched)">
                        <small class="text-danger" *ngIf="phone.errors?.['required']">phone number is required</small>
                        <small class="text-danger" *ngIf="phone.errors?.['pattern']">phone number must be 10 digits</small>
                    </div>
                  </div><br>

                  <div class="form-group">
                    <select (blur)="validateTopic(topic.value)" (change)="validateTopic(topic.value)"
#topic="ngModel" [class.is-invalid]="topicHasError && topic.touched" class="custom-select"
name="course" [(ngModel)]="student.topic">
                        <option  value="default">I am interested in </option>
                        <option *ngFor="let topic of topics" >{{topic}}</option>
                    </select>
                    <div>
                        <small class="text-danger" [class.d-none]="!topicHasError || topic.untouched">please choose a
topic</small>
                    </div>
                  </div><br>

                  <div class="form-group">
                    <div class="form-check">
                        <input class="form-check-input" type="radio" name="timePreference"
value="morning" [(ngModel)]="student.time">
                        <label class="form-check-label">Morning(9AM-12PM)</label>
                    </div>
                  </div>
                  <div class="form-check">
                    <input class="form-check-input" type="radio" name="timePreference"
value="evening" [(ngModel)]="student.time">
                    <label class="form-check-label">Evening(6PM-9PM)</label>
                  </div>

                  <div class="forom-check-mb3">
                    <input class="form-check-input" type="checkbox" name="subscribe" [(ngModel)]="student.subscribe">
                    <label class="form-check-label">Subscribe Promotional Offers</label>
                  </div><br>

                  <div>
                    <!-- disable the button if studentForm has an error or topic has error -->
                    <button [disabled]="studentForm.form.invalid || topicHasError" class="btn btn-primary"
type="submit">Submit</button>
                  </div>
              </form>

              <ng-template #displayMsg><h3 [ngStyle]="{'color':'darkgreen'}">Form has been submitted successfully!</
h3>.</ng-template>


            </div>
```

app.component.ts:

```
import { Component } from '@angular/core';
import { Student } from './student';
import { StudentService } from './student.service';
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'Template Driven Forms';

  topicHasError: boolean = true;

  public submitted : boolean = false;

  public topics = ['Python', 'Java', 'Devops'];

  public student = new Student('NaniBabu', 'nanipallapu369@gmail.com', 9392590089, 'default', 'morning', true);

  constructor(private studentService : StudentService){

  }

  validateTopic(value: string) {
       if (value == 'default') {
         this.topicHasError = true;
       } else {
         this.topicHasError = false;
       }
  }

  onSubmitt(){
       console.log(this.student);
       this.submitted = true;
       this.studentService.enroll(this.student).subscribe(
         data => console.log('Printed Successfully', data),
         error => console.log('error occured!', error)
       );

  }
}
```

student.ts class file :

```
export class Student {

       constructor(
              public name : string,
              public mail : string,
              public phone : number,
              public topic : string,
              public time  : string,
              public subscribe : boolean

       ){

       }
}
```

student.service.ts:

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse, HttpResponse } from '@angular/common/http';
import { Student } from './student';

@Injectable({
  providedIn: 'root'
})
export class StudentService {

  public url : string = 'http://localhost:8008/enroll';
  constructor(private httpClient : HttpClient) {

   }

   enroll(student : Student){
       return this.httpClient.post<any>(this.url, student);
   }
}
```

REACTIVE FORMS:
==========

Reactive Forms is created inside component class so it is also referred as model driven forms.

Every form control will have an object in the component and this provides greater

control and flexibility in the form programming. Reactive Form is based on structured

To enable reactive forms, first we need to import ReactiveFormsModule in app.module.ts.

--> Points to remember :

1. Code and logic reside in component class.
2. No two-way binding.
3. Well suited for complex scenarios.
4. Dynamic form fileds.
5. Custom validation.
6. Dynamic validation.
7. Unit testing is possible.

--> What we are going to do is :

1. CLI generated project.
2. Add the form HTML.

3. Create the form module.
4. Manage the form control values.
5. FormBuilder services.
6. Validation - Simple , Custom, Cross-field and Dynamic.
7. Dynamic form controls.
8. Submitting form data.

1. creating Form module and Nesting Form Groups and Manage the Form control values steps below.

--> import ReactiveFormsModule in import section app.module.ts file.

app.component.ts.

```
import { Component } from '@angular/core';
import { FormGroup } from '@angular/forms'
import { FormControl } from '@angular/forms'


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'Reactive Form Example';
  submitted : boolean = false;


  // creating form module

  registerationUserForm  = new FormGroup({
    userName : new FormControl('nani pallapu'),
    password : new FormControl(''),
    confirmPassword : new FormControl(''),

    //Nesting FormGroups
    address : new FormGroup({
      city : new FormControl(''),
      state : new FormControl(''),
      postalCode : new FormControl('')
    })

  });

  // this method is used to disable the submit button and recieving registerationUserForm and print it on console.
  onSubmitt(registerationUserForm : FormGroup){
    this.submitted = !this.submitted;
    alert('form has been submitted!');
    console.log(registerationUserForm);
  }

  // this method is used to pre-fill registerationUserForm gettting Data from API
  // data has to maintain structure of form group if we use setValue

  loadData(){

    this.registerationUserForm.setValue({
      userName : 'Nani Babu Pallapu',
```

```
      password : 'Hyderabad@369',
      confirmPassword : 'Hyderabad@369',
      address : {
        city : 'Eluru',
        state : 'Andhra Pradesh',
        postalCode : '543001'
      }
    });
  }


  // this method is used to pre-fill registerationUserForm gettting Data from API
  // data don't have to maintain structure of form group if we use patchValue

  // loadData(){
  //   this.registerationUserForm.patchValue({
  //     userName : 'Nani Babu Pallapu',
  //     password : 'Hyderabad@369',
  //     confirmPassword : 'Hyderabad@369',
  //   });
  // }

}
```

app.component.html

```
<h2 [ngStyle]="{'text-align':'center'}">{{title}}</h2>

<div class="container-fluid">
  <form [formGroup]="registerationUserForm"> <!-- Binding the registerationUserForm to formGroup directive -->

    <!-- formControlName="userName" userName is what we have declared in ts file for FormControl  -->
    <div class="form-group">
      <label>Username : </label>
      <input formControlName="userName" type="text" placeholder="enter username" class="form-control">
    </div>

    <div class="form-group">
      <label>Password : </label>
      <input formControlName="password" type="password" placeholder="enter passwrod" class="form-control">
    </div>

    <div class="form-group">
      <label>Confirm Password : </label>
      <input formControlName="confirmPassword" type="password" placeholder="confirm your password" class="form-
control">
    </div>

    <div formGroupName="address">
      <div class="form-group">
        <label>City : </label>
        <input formControlName="city" type="text" placeholder="enter city" class="form-control">
      </div>

      <div class="form-group">
        <label>State : </label>
        <input formControlName="state" type="text" placeholder="enter state" class="form-control">
```

```
    </div>

    <div class="form-group">
      <label>Postal Code : </label>
      <input formControlName="postalCode" type="password" placeholder="enter postal code" class="form-control">
    </div>
  </div>

  <div>
    <button class="btn btn-primary" (click)="onSubmitt(registerationUserForm)" [disabled]="submitted"
type="submit">Register</button>

    <!--  this button is used to pre-fill registerationUserForm gettting Data from API -->
    <button class="btn btn-secondary" (click)="loadData()" type="button">Load API Data</button>
  </div>


 </form>

 <hr /> <!– horizantal line –>
 <div>
  {{registerationUserForm.value | json }}
 </div>
</div>
```

2. FormBuilder services and simple validation steps below:

--> Creating multiple Form Controls instance manually can become very repeatative.
--> To overcome this, Angular provides FormBuilder Service which intend provides methods to handle generating form controls.
--> We will still create Form Control instances , but with lesser code.

Reactive FORM Example
==========

app.component.scss

```
input[type=text] {
    width:300px;
}
input[type=email] {
    width:300px;
}

input[type=password] {
    width:300px;
}

.form-group {
    display: flex;
    flex-direction: row;
    align-items: center;
}
```

```
label{
        float: left;
        width: 150px;
        text-align: right;
        margin-right: 0.5em;

        white-space: nowrap;
        overflow: hidden;
        text-overflow: ellipsis;
        -o-text-overflow: ellipsis;
}

  input{
        margin: 20px;
        margin-right: 50px;
        margin-left: 100px;
        margin-bottom: 10px;
        margin-top: 10px;
        border: 0.5px solid lightgray;

  }

  .btn-primary{
        width: fit-content;
        text-align: center;
        padding-left: 250px;
        padding-right: 250px;
  }

  .btn-secondary{
        margin: 20px;
  }

  .addStyles{
        margin-left: 260px;
        padding-left: 20px;
        margin-right: 10px;
  }

  .addMails{
        border: none;
        background-color: transparent;
        font-size: 250%;
        font-weight: bold;
        color: grey;
        margin-left: -50px;
        margin-bottom: 10px;

  }

  .hide{
        display: none;
  }
  .addMails:hover + .hide{
        display: block;
        color: gray;
        font-weight: 500;
  }

  #changeStyle{
```

```
        margin-top: -10px;
        margin-left: 260px;
        padding-left: 20px;
        margin-right: 10px;
        margin-bottom: 20px;
    }
```

app.component.html

```
        <h2 [ngStyle]="{'text-align':'center'}">{{title}}</h2>

        <div class="container-fluid">
          <form [formGroup]="registerationUserForm" (ngSubmit)="onSubmit()" > <!-- Binding the
registerationUserForm to formGroup directive -->

            <!-- formControlName="userName" userName is what we have declared in ts file for FormControl  -->
            <!-- registerationUserForm.get('userName')? is replaced by username in ts file by assing this long value to
variable.-->
            <div class="form-group">
              <label>Username : </label>
              <input [class.is-invalid]="userName?.invalid && userName?.touched" formControlName="userName"
type="text"
                  placeholder="enter username" class="form-control">
              <small [class.d-none]="userName?.valid || userName?.untouched" class="text-danger">username is
required*</small>
              <!-- don't show this if username is valid and untouched-->
            </div>


            <div class="form-group">
              <label>Email : </label>
              <input [class.is-invalid]="eMail?.invalid && eMail?.touched" type="email"  formControlName="mail"
class="form-control" placeholder="enter email address">
              <button type="button" class="addMails" (click)="addAlternativeEmails()">+</button>
              <small class="hide">Add Alternative Emails</small>

              <small [class.d-none]="eMail?.valid || eMail?.untouched" class="text-danger">Email is required</small>
            </div>


              <!-- formArrayName is a directive, it will help us to keep track of FormArray values -->
              <!-- index is used to iterate and insert each input tag into the array based on array -->
              <div formArrayName="alternativeMails" *ngFor="let alterMail of alterEmails.controls; let i=index">
                  <input type="email" id="changeStyle" placeholder="enter alternative email {{i+1}}" class="form-
control" [formControlName]="i">
              </div>


              <div class="form-group">
                  <input  class="addStyles" formControlName="subscribing" type="checkbox">
                  <small>subscribe for promotional offers</small>
              </div>
```

```
                <div class="form-group">
                  <label>Password : </label>
                  <input [class.is-invalid]="password?.invalid && password?.touched" formControlName="password"
type="password"
                        placeholder="enter passwrod" class="form-control">

                  <div *ngIf="password?.invalid && password?.touched">
                        <div>
                          <small *ngIf="password?.errors?.['required']" class="text-danger">Password is required</small>
                        </div>
                        <div>
                          <small *ngIf="confirmPass?.errors?.['PasswordValidator'] || password?.errors?.['maxlength']"
                              class="text-danger">Password must be between 4 to 15 characters </small>
                        </div>
                  </div>

                </div>

                <!-- registerationUserForm.errors?.['misMatch'] -> it will check, password, confirmPassword both are
matchied or misMatched based on validators. -->
                <div class="form-group">
                  <label>Confirm Password : </label>
                  <input
                        [class.is-invalid]="(confirmPass?.invalid && confirmPass?.touched) || registerationUserForm.errors?.
['misMatch']"
                        formControlName="confirmPassword" type="password" placeholder="confirm passwrod" class="form-
control">

                  <div *ngIf="(confirmPass?.invalid && confirmPass?.touched) || registerationUserForm.errors?.
['misMatch']">
                        <div>
                          <small *ngIf="confirmPass?.errors?.['required']" class="text-danger">confirmPass is required</
small>
                        </div>

                        <div>
                          <!-- PasswordValidator custom validation from password-validators.ts -->
                          <small *ngIf="confirmPass?.errors?.['PasswordValidator']" class="text-danger">Password must be
between 4 to 15
                              characters </small>

                        </div>
                        <div>
                          <small *ngIf="registerationUserForm.errors?.['misMatch']" class="text-danger">confirmPassword did
not match
                              password</small>

                        </div>
                  </div>
                </div>

                <div formGroupName="address">
                  <div class="form-group">
                        <label>City : </label>
                        <input formControlName="city" type="text" placeholder="enter city" class="form-control">
                  </div>

                  <div class="form-group">
```

```
                    <label>State : </label>
                    <input formControlName="state" type="text" placeholder="enter state" class="form-control">
                  </div>

                  <div class="form-group">
                    <label>Postal Code : </label>
                    <input formControlName="postalCode" type="password" placeholder="enter postal code"
class="form-control">
                  </div>
                </div>

                <div>
                  <button class="btn btn-primary" [disabled]="submitted"
                    type="submit">Register</button>

                  <!--  this button is used to pre-fill registerationUserForm gettting Data from API -->
                  <button class="btn btn-secondary" (click)="loadData()" type="button">Load API Data</button>
                </div>


              </form>

              <hr /> <!-- horizantal line -->
              <div>
                {{registerationUserForm.value | json }}
              </div>
            </div>
            <br>
            <br>
            <br>
            <br>
            <br>
            <br>



                    app.component.ts
            ---------------------------------------------------------------------------------------------

                    import { Component, OnInit } from '@angular/core';
            import { FormBuilder, FormGroup, Validators, FormArray } from '@angular/forms'
            import { FormControl, } from '@angular/forms'
            import { PasswordValidator } from './password-validator';
            import { PasswordMatchingValidator } from './password-matching-validator';
            import { RegistrationService } from './registration.service';


            @Component({
              selector: 'app-root',
              templateUrl: './app.component.html',
              styleUrls: ['./app.component.scss']
            })
            export class AppComponent implements OnInit {
              title = 'Reactive Form Example';
              submitted : boolean = false;

              registerationUserForm! : FormGroup;
```

```
//   this.registerationUserForm.get('userName'); is replaced by username
// it will be set to class binding in html form.
get userName(){
  return this.registerationUserForm.get('userName');
}

get eMail(){
  return this.registerationUserForm.get('mail');
}

get alterEmails(){
  return this.registerationUserForm.get('alternativeMails') as FormArray;
}


// addAlternativeEmails() a method that can be called dinamyically to insert FormControl into the FormArray
addAlternativeEmails(){
  this.alterEmails.push(this.formBuilder.control(''));
}

// it will be set to class binding in html form.
get password(){
  return this.registerationUserForm.get('password');
}

// it will be set to class binding in html form.
get confirmPass(){
  return this.registerationUserForm.get('confirmPassword');
}




  constructor(private formBuilder : FormBuilder, private registerationService : RegistrationService){

  }

  //   this.registerationUserForm.get('userName'); is replaced by username

  ngOnInit() {
       this.registerationUserForm = this.formBuilder.group ( {
         userName : ['nani pallapu', Validators.required], // adding simple validation
         mail : [''],
         subscribing : [false],
         password : ['', [Validators.required,Validators.maxLength(15), PasswordValidator]], // adding multiple
validations in array
         confirmPassword : ['',[Validators.required, PasswordValidator]], // custom validation for confirmPassword.
It will make sure length should be between 4 to 8
         address : this.formBuilder.group({
              city : [''],
              state : [''],
              postalCode : ['']
         }),

         // Dynamic Form Control - for this we use FormArray
         alternativeMails : this.formBuilder.array([])
       },

       // Cross Field Validation.
```

```
        { validator : PasswordMatchingValidator}

        );


        //Conditional Validation.
        this.registerationUserForm.get('subscribing')?.valueChanges
        .subscribe(checkedValue  =>{
          const email = this.registerationUserForm.get('mail');

          // if subscribing is clicked, we are setting validators for email , otherwise clearing the validators.
          if(checkedValue){
              email?.setValidators(Validators.required);
          } else{
              email?.clearValidators;
          }

          email?.updateValueAndValidity(); // finally , we are updating it to ensure correct status is reflected.
        });

    }
    /*
      registerationUserForm = this.formBuilder.group ( {
          userName : ['nani pallapu', Validators.required], // adding simple validation
          email : [''],
          password : ['', [Validators.required,Validators.maxLength(15), PasswordValidator]], // adding multiple
validations in array
          confirmPassword : ['',[Validators.required, PasswordValidator]], // custom validation for confirmPassword.
It will make sure length should be between 4 to 8
          address : this.formBuilder.group({
            city : [''],
            state : [''],
            postalCode : ['']
          })
      }, { validator : PasswordMatchingValidator}

      );

    */






    /*
      // creating form module

      registerationUserForm  = new FormGroup({
          userName : new FormControl('nani pallapu'), //'nani pallapu' is default value.
          password : new FormControl(''),
          confirmPassword : new FormControl(''),

          //Nesting FormGroups
          address : new FormGroup({
            city : new FormControl(''),
            state : new FormControl(''),
            postalCode : new FormControl('')
          })
```

```
  });

*/

  // this method is used to disable the submit button and recieving registerationUserForm and print it on console.
  onSubmit(){
        this.submitted = !this.submitted;
        alert('form has been submitted!');
        console.log(this.registerationUserForm);

        this.registerationService.register(this.registerationUserForm.value).subscribe(
          response => console.log("Success!"),
          error => console.log("Error Occured")
        );
  }

  // this method is used to pre-fill registerationUserForm gettting Data from API
  // data has to maintain structure of form group if we use setValue

  loadData(){

        this.registerationUserForm.setValue({
          userName : 'Nani Babu Pallapu',
          password : 'Hyderabad@369',
          confirmPassword : 'Hyderabad@369',
          address : {
                city : 'Eluru',
                state : 'Andhra Pradesh',
                postalCode : '543001'
          }
        });
  }

 /*
  // this method is used to pre-fill registerationUserForm gettting Data from API
  // data don't have to maintain structure of form group if we use patchValue

  loadData(){
        this.registerationUserForm.patchValue({
          userName : 'Nani Babu Pallapu',
          password : 'Hyderabad@369',
          confirmPassword : 'Hyderabad@369',
        });
  }

*/

}


register.service.ts
-------------------



import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
```

```
})
export class RegistrationService{

  public url : string = 'http://localhost:8008/enroll';

  constructor( private httpClient : HttpClient) {

  }

  register(userData : any){
      return this.httpClient.post<any>(this.url, userData)
  }

}
```

password-matching-validator.ts

```
import { AbstractControl, Validators } from "@angular/forms";

// this funcation is used to validate password and confirmPassword both are same or not.
// instead of passing single FormControl , We are passing FormGroup because we need to validate password and confirmPassword.
export function PasswordMatchingValidator(control : AbstractControl): { [key : string]: boolean } | null {

    const password = control.get('password'); // fetching from password FormControl password
    const confirmedPassword = control.get('confirmPassword')//fetching from confirmPassword FormControl password

    // it is used to make sure password is entered or not. and touched the confirmedPasswod or not.
    if(password?.pristine || confirmedPassword?.pristine){
        return null;
    }
    // if password and confirmedPassword don't matchi, we are setting misMatch value as true using ternary operator, otherwise it will be set to null.
    return password && confirmedPassword && password.value != confirmedPassword.value ? { 'misMatch':true} : null
}
```

password-validator.ts

```
import { AbstractControl } from "@angular/forms";

export function PasswordValidator(control : AbstractControl){

    // custom validation for confirmPassword. It will make sure length should be between 4 to 8
        if(control && control.value && (control.value === null || control.value ===" || (control.value.length >=4 && control.value.length <=15))){
            return null;
        }
        return {
            PasswordValidator : true
        };
}
```