Angular Architecture

What Is Angular?

Angular is an open-source, JavaScript framework written in TypeScript. Google maintains it, and its primary purpose is to develop single-page applications. As a framework, Angular has clear advantages while also providing a standard structure for developers to work with. It enables users to create large applications in a maintainable manner.

Different Angular Versions of Angular:

Angular was developed in 2009,
the original Angular, called Angular 1 and eventually known as AngularJS. Then came Angulars 2, 3, 4, 5, until finally, the current version, Angular 12, released on 12/05/2021.

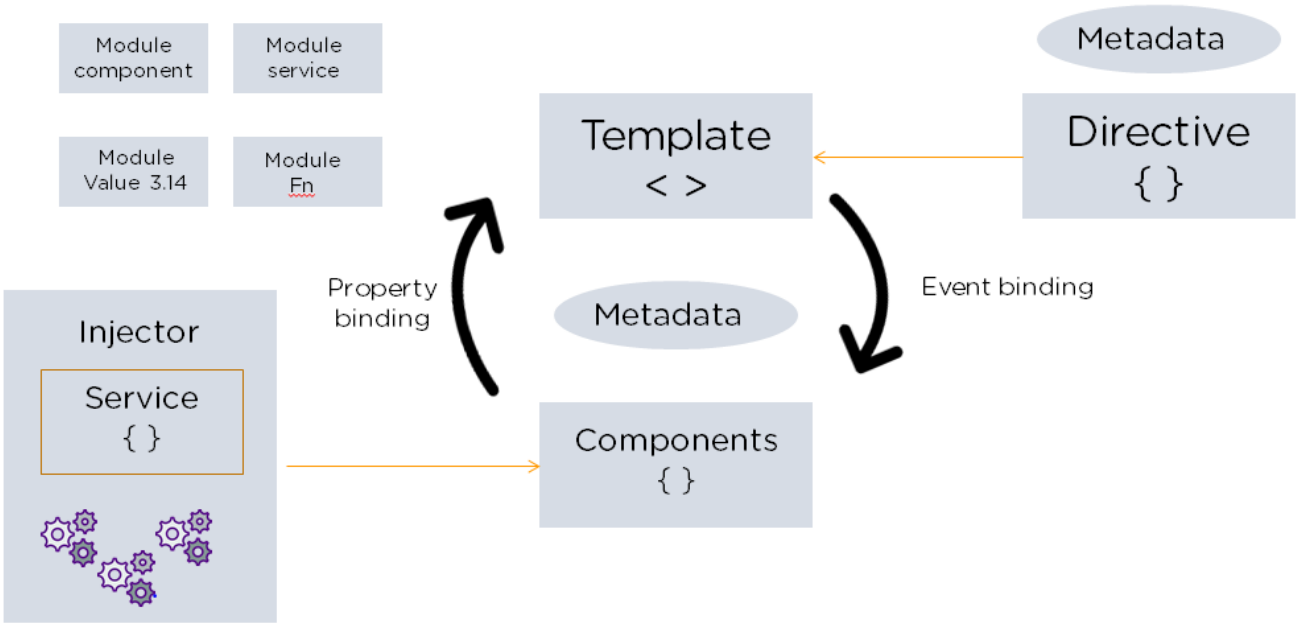
Each subsequent Angular version improves on its predecessor, fixing bugs, addressing issues, and accommodating increasing complexity of current platforms.

Features of Angular

1. Document Object Model
2.TypeScript
3. Data Binding
4. Testing

Angular Architecture

Angular is a full-fledged model-view-controller (MVC) framework. It provides clear guidance on how the application should be structured and offers bi-directional data flow while providing real DOM



thebuilding blocks of an Angular application:

1. Modules
An Angular app has a root module, named AppModule, which provides the bootstrap mechanism to launch the application.

2. Components
Each component in the application defines a class that holds the application logic and data. A component generally defines a part of the user interface (UI).

3. Templates
The Angular template combines the Angular markup with HTML to modify HTML elements before they are displayed. There are two types of data binding:

4.Event binding: Lets your app respond to user input in the target environment by updating your application data.
5.Property binding: Enables users to interpolate values that are computed from your application data into the HTML.

5. Metadata
Metadata tells Angular how to process a class. It is used to decorate the class so that it can configure the expected behavior of a class.

6. Services
When you have data or logic that isn't associated with the view but has to be shared across components, a service class is created. The class is always associated with the @Injectible decorator.

7. Dependency Injection
This feature lets you keep your component classes crisp and efficient. It does not fetch data from a server, validate the user input, or log directly to the console. Instead, it delegates such tasks to the services.

Advantages of Angular

1. Custom Components
Angular enables users to build their own components that can pack functionality along with rendering logic into reusable pieces. It also plays well with web components.

2. Data Binding
Angular enables users to effortlessly move data from JavaScript code to the view, and react to user events without having to write any code manually.

3. Dependency Injection
Angular enables users to write modular services and inject them wherever they are needed. This improves the testability and reusability of the same services.

4. Testing
Tests are first-class tools, and Angular has been built from the ground up with testability in mind. You will have the ability to test every part of your application—which is highly recommended.

5. Comprehensive
Angular is a full-fledged framework and provides out-of-the-box solutions for server communication, routing within your application, and more.

6. Browser Compatibility
Angular is cross-platform and compatible with multiple browsers. An Angular application can typically run on all browsers (Eg: Chrome, Firefox) and OSes, such as Windows, macOS, and Linux.

Limitations of Angular

1. **Steep Learning Curve**

The basic components of Angular that all users should know include directives, modules, decorators, components, services, dependency injection, pipes, and templates.

2. Limited SEO Options

Angular offers limited SEO options and poor accessibility to search engine crawlers.

3. Migration

One of the reasons why companies do not frequently use Angular is the difficulty in porting legacy js/jquery-based code to angular style architecture. Also, each new release can be trouble some to upgrade, and several of them are not backward-compatible.

4. Verbose and Complex

A common issue in the Angular community is the verbosity of the framework. It is also fairly complex compared to other front-end tools.

The Differences Between Angular and AngularJS?

Angular is the catch-all term for every version of the framework (1-12), while AngularJS is the initial Angular version, renamed. Although it's over ten years old, AngularJS isn't obsolete; it still finds lots of use developing smaller web applications.

	AngularJS	Angular
Architecture	Supports mode-view component design	Uses directives and components
Language	JavaScript	Microsoft's TypeScript
Mobile capability	No mobile browser support	Supported by all popular mobile browsers
Structure	Not as manageable as Angular, but ideal for small applications	Easier to build and maintain large applications
Routing	Uses $routeprovider.when() for routing configuration	Uses @Route Config{(…)} for routing configuration
Performance	Not as fast as Angular	Faster than AngularJS

angular install

npm install -g @angular/cli

to create new project

ng new Project name

cd > project name

to run the project

ng serve

for angular 10

npm install -g @angular/cli@10.0.0

File and Folder Structure

package.json :- contains the details about the project

node_packages :- contains project libs.

src_folder :- we will the code.. in src..

app :- a component a component is a bunch of file
css :- for css design
html :- design
spec.ts:- for testing logig program in type script
module :- in a single module more than one component

assets

public folder:- to access images, files etc

environments

we can set the environment testing,production

add the config according to the requirement

index.html--this is the first file to execute the file

main.ts file connect with html and angular and this is the bridge between the angular and html

style.css

we can write the globally css

package-lock.json

full details about the packages

angular.json :- config of your file

tsconfig file:-- type script config file

tsconfigapp.json:- for coding file

tsconfigspec.json:- for testing config file

broweserlisttc :- details about the browser file

karma.config :- for testing config file

polifill.ts:- it will load all the lib for runing the index.html file

Data Binding

Data binding automatically keeps your page up-to-date based on your application's state. You use data binding to specify things such as the source of an image, the state of a button, or data for a particular user.

Property binding: Property binding is a one-way mechanism that lets you set the property of a view element. It involves updating the value of a property in the component and binding it to an element in the view template. Property binding uses the [] syntax for data binding. An example is setting the disabled state of a button.

```
// component.html
  <button [disabled]="buttonDisabled"></button>
HTML
  // component.ts
  @Component({
    templateUrl: 'component.html',
    selector: 'app-component',
  })
  export class Component {
    buttonDisabled = true;
  }
```

Event binding: This data binding type is when information flows from the view to the component when an event is triggered. The view sends the data from an event like the click of a button to be used to update the component. It is the exact opposite of property binding, where the data goes from the component to the view.

```
// component.html
  <p>My name is {{name}}</p>
  <button (click)="updateName()">Update button</button>
// component.ts
  @Component({
    templateUrl: 'component.html',
    selector: 'app-component',
  })
```

```
export class Component {
  name = 'Ramu';

  updateName() {
    this.name = 'Rama Krishna';
  }
}
```

Two-way data binding

Property binding + Data Binding == Two -way data Binding

Two-way data binding refers to sharing data between a component class and its template. If you change data in one place, it will automatically reflate at the other end.

For example, if you change the value of the input box, then it will also update the value of the attached property in a component class.

```
<input [(ngModel)]="courses.name" placeholder="name"/>
```

ngModel Directive

The Angular uses the ngModel directive to achieve the two-way binding on HTML Form elements. It binds to a form element like input, select, selectarea. etc.

How to use ngModel

The ngModel directive is not part of the Angular Core library. It is part of the FormsModule library. You need to import the FormsModule package into your Angular module.

The ngModel directive with [()] syntax (also known as banana box syntax) syncs values from the UI to a property and vice-versa.
So, whenever the user changes the value on UI, the corresponding property value will get automatically updated.

[()] = [] + () where [] binds attribute, and () binds an event.

addcom.html

```
<input type="text" [(ngModel)]="data" placeholder="Enter Name">
<h3> Entered data is  {{data}}</h3>
```

appmodule.ts

```
import { FormsModule } from '@angular/forms';

 imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
```

appcomp.ts

```
export class AppComponent {
  title = 'testAngular';
  buttonDisabled=false;
   data = "Ram and Syam";
}
```

Attribute Binding

Attribute binding is to set the value of attribute directly.
Attribute binding is performed in the same way as property binding with difference that in attribute binding we need to prefix attribute name with attr. keyword

is a techique that allows the user to bind attribute of elements from componment to view (templeate)

the data flow is only one-way i.e from component to view

can be used for any existing properties or custom attributes

syntax for defining attribute binding is
[attr.attribute_name]="expression"

component.ts

```
isrc="C:\\Users\\WELCOME\\Downloads\\\i1.JPG"
```

html

```
333333333
<img [attr.src]="isrc" width="50" height="50">
<table [border]="bdr" [attr.height]="h" [width]="w">
  <tr>
    <td [attr.colspan]="clspn"> A + B </td>
  </tr>
  <tr>
    <td> C </td>
          <td> D </td>
  </tr>
</table>
```

//add comp.ts

```
h = 300;
w = 200;
bdr = 5;
clspn = 2;
```

Style Binding

It is very easy to give the CSS styles to HTML elements using style binding
tyle binding is used to set a style of a view element
We can set the inline styles of an HTML element using the style binding in angular
You can also add styles conditionally to an element, hence creating a dynamically styled element.

```
export class ExampleComponent {

  fontSize: number = 80;
}

 fontSize: number = 80;
  fontColor: string = "red";
  IsBold: boolean = true;
  IsItalic: boolean = true
  textCenter: string = "center";


  MultipleInlineStyle() {
    let myStyleClass = {
      'font-weight': this.IsBold ? 'bold' : 'normal',
      'font-style': this.IsItalic ? 'italic' : 'normal',
      'font-size.px': this.fontSize,
      'color': this.fontColor,
      'text-align': this.textCenter
    };
    return myStyleClass;
  }

<h1  [ngStyle]="MultipleInlineStyle()">Welcome to Angular</h1>

<div [style.color]="status=='error' ? 'red': 'blue'"
     [style.text-align] = "'center'"
     [style.font-size.px]="'80'" >
 Anguler made easy
</div>
```

.html

```
<h1 [style.font-size.px]="fontSize">Welcome to Angular</h1>
```

Class Binding
ngClass is more powerful. It allows you to bind a string of classes, an array of strings.

Class binding is used to set a class property of a view element.
We can add and remove the CSS class names from an element's class attribute with class binding.

.ts

```
.text-info{
  color: rgb(28, 120, 195);
}
.text-danger{
  color: rgb(231, 56, 56);
}
.text-bold{
  font-weight: bold;
}

  public isInfo: boolean = true;
```

.html

```
<p class="text-info">chennai</p>
<p [class]="text-info">delhi</p>
<p [class.text-info]="isInfo">mumbai</p>
```

Angular directive
A directive modifies the DOM by changing the appearance, behavior, or layout of DOM elements
Directives just like Component are one of the core building blocks in the Angular framework to build applications.
a directive is a TypeScript based function that executes whenever Angular compiler identified it within the DOM element.
Directives are used to provide or generate new HTML based syntax which will extend the power of the UI in an Angular Application

Why Directives Required?
Reusability – In an Angular application, the directive is a self-sufficient part of the UI. As a developer, we can reuse the directive across the different parts of the application. This is very much useful in any large-scale applications where multiple systems need the same functional elements like search box, date control, etc.
Readability – Directive provides much more readability for the developers to understand the production functionality and data flow.
Maintainability – One of the main use of directive in any application is the maintainability. We can easily decouple the directive from the application and replace the old one with a new one directives.

In-Build Structural Directive

Angular provides below mentioned built-in directives which can be used within a component to change the elements structure or design.

- ngIf
- ngFor
- ngSwitch

ngIf

.html

```
<div>
  <input type="button" value="{{caption}}" (click)="changeData()"/>
  <br />
  <h2 *ngIf="showInfo"><span>Demonstrate of Structural Directives - *ngIf</span></h2>
</div>
```

.ts

```
showColor: boolean = false;
showInfo: boolean = false;

public changeData(): void {
  this.showInfo = !this.showInfo;
  if (this.showInfo) {
     this.caption = 'Hide Text';
  }
  else {
     this.caption = 'Show Text';
  }
```

ngswitch

.ts

```
studentList: Array<any> = new Array<any>();
 this.studentList = [
          { SrlNo: 1, Name: 'Rajib Basak', Course: 'Bsc(Hons)', Grade: 'A' },
          { SrlNo: 2, Name: 'Rajib Basak1', Course: 'BA', Grade: 'B' },
          { SrlNo: 3, Name: 'Rajib Basak2', Course: 'BCom', Grade: 'A' },
          { SrlNo: 4, Name: 'Rajib Basak3', Course: 'Bsc-Hons', Grade: 'C' },
          { SrlNo: 5, Name: 'Rajib Basak4', Course: 'MBA', Grade: 'B' },
          { SrlNo: 6, Name: 'Rajib Basak5', Course: 'MSc', Grade: 'B' },
          { SrlNo: 7, Name: 'Rajib Basak6', Course: 'MBA', Grade: 'A' },
          { SrlNo: 8, Name: 'Rajib Basak7', Course: 'MSc.', Grade: 'C' },
          { SrlNo: 9, Name: 'Rajib Basak8', Course: 'MA', Grade: 'D' },
          { SrlNo: 10, Name: 'Rajib Basak9', Course: 'B.Tech', Grade: 'A' }
       ];
```

ngFor

```
 Movies: any[] = [
   {
    "name": "Avengers: Endgame"
   },
   {
    "name": "Good Boys"
   },
   {
    "name": "Playmobil: The Movie"
   },
   {
    "name": "Aquarela"
   },
   {
    "name": "Aladdin"
   },
   {
    "name": "Downton Abbey"
   }
 ];

<ul>
    <li *ngFor="let movie of Movies">
      {{ movie.name }}
    </li>
  </ul>
```

nested ngFor

```
<ul *ngFor="let collection of industryType">
    <li>{{ collection.type }}</li>
    <ul>
      <li *ngFor="let hollywood of collection.movies">
        {{ hollywood.name }}
      </li>
    </ul>
  </ul>

industryType: any[] = [
   {
    'type': 'Bollywood',
    'movies': [
      {
       "name": "Gully Boy"
```

```
      },
      {
        "name": "Manikarnika"
      },
      {
        "name": "Kalank"
      }
    ]
  },
  {
    'type': 'Hollywood',
    'movies': [
      {
        "name": "Avengers: Endgame"
      },
      {
        "name": "Good Boys"
      },
      {
        "name": "Playmobil"
      },
      {
        "name": "Aquarela"
      }
    ]
  }
];
```

NgFor Track By

The trackBy method helps in keeping track of collection elements in Angular. By default, Angular doesn't track if any item is being added or removed in the collection. Angular offers trackBy feature it can be used with *ngFor directive.

```
employees: any[];

  constructor() {
    this.employees = [
      {
        code: 'emp101', name: 'Tom', gender: 'Male',
        annualSalary: 5500, dateOfBirth: '25/6/1988'
      },
      {
        code: 'emp102', name: 'Alex', gender: 'Male',
        annualSalary: 5700.95, dateOfBirth: '9/6/1982'
      },
      {
```

```
            code: 'emp103', name: 'Mike', gender: 'Male',
            annualSalary: 5900, dateOfBirth: '12/8/1979'
        },
        {
            code: 'emp104', name: 'Mary', gender: 'Female',
            annualSalary: 6500.826, dateOfBirth: '14/10/1980'
        },
    ];
}

getEmployees(): void {
    this.employees = [
        {
            code: 'emp101', name: 'Tom', gender: 'Male',
            annualSalary: 5500, dateOfBirth: '25/6/1988'
        },
        {
            code: 'emp102', name: 'Alex', gender: 'Male',
            annualSalary: 5700.95, dateOfBirth: '9/6/1982'
        },
        {
            code: 'emp103', name: 'Mike', gender: 'Male',
            annualSalary: 5900, dateOfBirth: '12/8/1979'
        },
        {
            code: 'emp104', name: 'Mary', gender: 'Female',
            annualSalary: 6500.826, dateOfBirth: '14/10/1980'
        },
        {
            code: 'emp105', name: 'Nancy', gender: 'Female',
            annualSalary: 6700.826, dateOfBirth: '15/12/1982'
        },
    ];
}

<table>
    <thead>
        <tr>
            <th>Code</th>
            <th>Name</th>
            <th>Gender</th>
            <th>Annual Salary</th>
            <th>Date of Birth</th>
        </tr>
    </thead>
    <tbody>
```

```
      <tr *ngFor='let employee of employees'>
        <td>{{employee.code}}</td>
        <td>{{employee.name}}</td>
        <td>{{employee.gender}}</td>
        <td>{{employee.annualSalary}}</td>
        <td>{{employee.dateOfBirth}}</td>
      </tr>
      <tr *ngIf="!employees || employees.length==0">
        <td colspan="5">
          No employees to display
        </td>
      </tr>
    </tbody>
</table>
<br />
<button (click)='getEmployees()'>Refresh Employees</button>
```

Component styles

Angular applications are styled with standard CSS. That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.

Additionally, Angular can bundle component styles with components, enabling a more modular design than regular stylesheets.

Component Inline Style

```
@Component({
  selector: 'app-test1',
  templateUrl: './test1.component.html',
  styles: [
    `p { color:blue}`
  ],
})
```

Component External Style

```
@Component({
  selector: 'app-test2',
  templateUrl: './test2.component.html',
  styleUrls: ['./test2.component.css'],
})
```

You can specify both Component inline & Component External style together as shown below

```
@Component({
```

```
  selector: 'app-test2',
  templateUrl: './test2.component.html',
  styles:[`p {color:yellow}`],
  styleUrls: ['./test2.component.css'],
})
```

test2.component.html

```
<style>
  p {
    color: purple;
  }
  </style>

<p>
  test2 works!
</p>
```

Template Inline Style using link tag

You can add the external style sheets using the the link tag as shown below. The path must be relative to the index.html

```
<link rel="stylesheet" href="assets/css/morestyles.css">
<p>
  test2 works!
</p>
```

Working with pipes

Pipes are very useful when it comes to managing data within interpolation "{{ | }}". In Angular, pipes were referred to as filters now they are famous by the name of Pipes.

Angular offers lots of built-in Pipes to deal with specific problems while developing Angular application.

In order to transform data, we use the character |.

```
{{ i will become uppercase one day | uppercase }}
```

Pipes accept date, arrays, strings, and integers as inputs. Inputs are separated with |. Later, the inputs will be converted in the desired format before displaying them in the browser.

app.component.ts

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

@Component({
```

```
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  convertText: string = "I Am Being Managed By Pipes";
}
```

TypeScript
You will find the following code segment in the app.component.html file.
app.component.html

```
<p> Unformatted date : {{toDate }} </p>
            <p> Formatted date : {{toDate | date}} </p>`

 toDate: Date = new Date();
```

Passing arguments to pipes

We can also pass optional arguments to the pipe. The arguments are added to the pipe using a colon (:) sign followed by the value of the argument. If there are multiple arguments separate each of them with the colon (:).

```
  <p> Formatted date : {{toDate | date:'medium'}} </p>`
```

Chaining Pipes
Pipes can be chained together to make use of multiple pipes in one expression.

```
<p> Formatted date : {{toDate | date:'medium'| upercase}} </p>`
```

DatePipe
The Date pipe formats the date according to locale rules. The syntax of the date pipe is as shown below

Syntax

date_expression | date[:format]

Where

date_expression is a date object or a number

date is the name of the pipe

format is the date and time format string which indicates the format in which date/time components are displayed.

Component	format	Example
Year	y	2016
Year	yy	16
Month	M	9
Month	M	99
Month	MMM	Nov
Month	MMMM	November
Day	d	9
Day	dd	09
hour	j	9
hour	jj	09
hour	h	9 AM
hour	hh	09 AM
hour24	H	13
hour24	HH	13
minute	m	9
minute	mm	09
second	s	9
second	ss	99
Time zone	z	Pacific Standard time
Time zone	Z	GMT-8:00
Time zone	a	PM

Component	format	Example
era	G	AD
era	GGGG	Anno Domini

Format argument also supports some predefined commonly used formats

Format Name	Equivalent Format strng	Example (for en-US)
medium	yMMMdjms	Sep 3, 2010, 12:05:08 PM
short	yMdjm	9/3/2010, 12:05 PM
fullDate	yMMMMEEEEd	Friday, September 3, 2010
longDate	yMMMMd	September 3, 2010
mediumDate	yMMMd	Sep 3, 2010
shortDate	yMd	9/3/2010
mediumTime	Jms	12:05:08 PM
shortTime	Jm	12:05 PM

UpperCasePipe & LowerCasePipe

msg: string= 'Welcome to Angular';

```
<p>Uppercase :{{msg | uppercase}} </p>
<p>Lowercase :{{msg | lowercase}} </p>`
```

SlicePipe
Creates a new List or String containing a subset (slice) of the string or array.

Syntax

array_or_string_expression | slice:start[:end]
array_or_string_expression is the string to slice

slice is the name of the pipe
start is the start position/index from where the slicing will start

end is the ending index/position in the array/string

```
<p>Example 1 :{{msg | slice:11:20}} </p>
        <p>Example 2 :{{msg | slice:-9}} </p>`

  msg: string= 'Welcome to Angular ';
```

DecimalPipe / NumberPipe

The Decimal Pipe is used to Format a number as Text. This pipe will format the number according to locale rules.
Syntax
number_expression | number[:digitInfo]

Where

number_expression is the number you want to format
number is the name of the pipe
digitInfo is a string which has the following format

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

Where
minIntegerDigits is the minimum number of integer digits to use. Defaults to 1.
minFractionDigits is the minimum number of digits after fraction. Defaults to 0.
maxFractionDigits is the maximum number of digits after fraction. Defaults to 3.

```
<p> Unformatted :{{num}}</p>
        <p> Formatted :{{num | number}}</p>
        <p> Formatted :{{num | number:'3.1-2'}}</p>
        <p> Formatted :{{num | number:'7.1-5'}} </p>`

 num: number= 1245879542.14554;
```

PercentePipe
Formats the given number as a percentage according to locale rules.

Syntax
number_expression | percent[:digitInfo]
number_expression is the number you want to format
percent is the name of the pipe

CurrencyPipe
Formats a number as currency using locale rules.
number_expression | currency[:currencyCode[:symbolDisplay[:digitInfo]]]

Where

number_expression currency to format a number as currency.
Currency is the name of the pipe
currencyCode is the ISO 4217 currency code, such as USD for the US dollar and EUR for the euro.
symbolDisplay is a boolean indicating whether to use the currency symbol or code. Use true to display symbol and false to use code
digitInfo is similar to the one used in decimal pipe

```
<p>Unformatted :{{cur}} </p>
        <p>Example 1 :{{cur | currency }} </p>
        <p>Example 2 :{{cur | currency:'INR':true:'4.2-2'}} </p>
```

cur: number= 175;

Angular Built-in Pipes
Async Pipe

The Async pipe is considered the best practice when you are getting data in the form of observables. The async pipe subscribes to an Observable/Promise automatically and returns the transmitted values

Currency Pipe
Date Pipe
Slice Pipe
Decimal Pipe
Json Pipe
Percent Pipe
LowerCase Pipe
UpperCase Pipe

custom pipe

Create custom pipes to encapsulate transformations that are not provided with the built-in pipes. Then, use your custom pipe in template expressions, the same way you use built-in pipes—to transform input values to output values for display.

Marking a class as a pipe
create pipe
ng generate pipe pipes/age

the pipe file will create in app folder

.ts File

```
 courseList = [
```

```
    {
      courseid: "1",
      coursename: "Java programming",
      author: "Franc",
      dob:new Date('03/07/1976')
    },
    {
      courseid: "2",
      coursename: "Learn Typescript programming",
      author: "John",
      dob:new Date('12/2/1920')
    }
  ];

  ngOnInit() {
    // Convert String obect to JSON
    this.stringJson = JSON.stringify(this.courseList);
    console.log("String json object :", this.stringJson);
    console.log("Type :", typeof this.stringJson);

    // ConvertjSON to an object
    this.stringObject = JSON.parse(this.stringJson);
    console.log("JSON object -", this.stringObject);
  }
}
```

generate new pipe by using this command

ng generate pipe pipes/age

in age.pipe.ts

```
transform(value: any): unknown {

    let cyear :any=new Date().getFullYear();
    let byear:any=new Date(value).getFullYear();
    let userAge=cyear-byear;
    return userAge ;

}
```

Appcomponent HTML

```
<table border="2">
    <tr *ngFor="let item of stringObject">
      <td>{{item.courseid}}</td>
      <td>{{item.coursename}}</td>
```

```
    <td>{{item.author}}</td>
    <td>{{item.dob}}</td>
    <td>{{item.dob | age}}</td>
  </tr>
</table>
```

pipe and change detection

for changes to data-bound values in a change detection process that runs after every DOM event: every keystroke, mouse move, timer tick, and server response.

.ts file

```
heroes: any[] = [];
  canFly = true;
  constructor() { this.reset(); }
  addHero(name: string) {
    name = name.trim();
    if (!name) { return; }
    let hero = {name, canFly: this.canFly};
    this.heroes.push(hero);
  }
```

.html file

```
New hero:
<input type="text" #box
   (keyup.enter)="addHero(box.value); box.value=""
   placeholder="hero name">
<button (click)="reset()">Reset</button>
<div *ngFor="let hero of heroes">{{hero.name}}</div>
```

Example 2

.html

```
<h3>
   Change detection is triggered at:
   <span [textContent]="time | date:'hh:mm:ss:SSS'"></span>
</h3>
<button (click)="0">Trigger Change Detection</button>
```

.ts file

```
 get time() {
   return Date.now();
}
```

Pure and impure pipes

A pure pipe is only called when Angular detects a change in the value or the parameters passed to a pipe. An impure pipe is called for every change detection cycle no matter whether the value or parameter(s) changes.

Pure pipes in Angular (which is also default) are executed only when Angular detects a pure change to the input value. A pure change is either a change to a primitive input value (such as String, Number, Boolean, or Symbol), or a changed object reference (such as Date, Array, Function, or Object).

A pure pipe must use a pure function. A pure function always return the same output for the same input.

Impure pipes in Angular
If you want pipe to be executed after a change within an object, such as a change to an element of an array, you need to define your pipe as impure to detect impure changes.

An impure pipe is called for every change detection cycle which could slow down your app drastically so be very careful with implementing an impure pipe in Angular.

custom pipe
Custom pipe class (filter.pipe.ts)

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name:'filter'
})
export class FilterPipe implements PipeTransform{
  transform(value: any, name: string) {
    if(name === ''){
      return value;
    }
    return value.filter((user) => user.firstName.startsWith(name));
  }
}
```

Component Class (app.component.ts)

```
{
 nameString = '';
  users = [{
      firstName: 'John',
      lastName: 'Doe',
      dept: 'Finance',
      salary: 5000,
      doj: new Date('2015-12-11')
```

```
    },
    {
      firstName: 'Amy',
      lastName: 'Watson',
      dept: 'HR',
      salary: 8000,
      doj: new Date('2013-07-23')
    },
    {
      firstName: 'Shrishti',
      lastName: 'Sharma',
      dept: 'IT',
      salary: 10000,
      doj: new Date('2019-10-20')
    }
  ]
}
```

Template (app.component.html)

```
 <h1>User Details</h1>
  <span>Search </span><input type="text" [(ngModel)]="nameString">
  <br/><br/>
  <table class="table table-sm table-striped m-t-4">
    <thead class="thead-dark">
      <tr>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Department</th>
      <th>Salary</th>
      <th>Joining Date</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let user of users | filter:nameString">
      <td>{{user.firstName}}</td>
      <td>{{user.lastName}}</td>
      <td>{{user.dept}}</td>
      <td>{{user.salary}}</td>
      <td>{{user.doj | date:'dd/MM/yyyy'}}</td>
    </tr>
  </tbody>
  </table>
</div>

<button type="button" class="btn btn-success m-3" (click)="addUser()">Add User</button>
```

```
<button type="button" class="btn btn-success" (click)="reset()">Reset</button>
```

Adding pipe declaration to the AppModule

```
addUser(){
  this.users.push({
    firstName: 'Alia',
    lastName: 'Bajaj',
    dept: 'Finance',
    salary: 6000,
    doj: new Date('2016-09-16')
  })
}

reset(){
  this.users = this.users.slice()
}
```

Angular Routing

Creating Routing

Routing is used to navigate from one component to another component in angular

1 config the routs
2 adding router outlet
3 add routing links in template

create new Project
ng new routeExample
after that
create components
1) ng g c header
2) ng g c about
3) ng g c contact

import router module in appmodule.ts

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

const routes: Routes = [

  {
    path:'',component:HomeComponent
  },
  {
```

```
  path:'about',component:AboutComponent
 },
 {
  path:'contact',component:ContactComponent
 }

];

 imports: [
  BrowserModule,
  AppRoutingModule,
  RouterModule.forRoot(routes)
 ],
```

wild card routs --> is used for any error will it displays the page.

```
 {
path:'**',component:ErrorComponentComponent
 }
```

by using nav bar

by using head.component.html

```
<ul>
  <li>
  <a href="/about">About</a>
  </li>
  <li>
  <a href="/contact">Contact</a>
  </li>
```

with out page refresh use routerlink insted of href

```
<ul>
  <li>
  <a routerLink="/about">About</a>
  </li>
  <li>
  <a routerLink="/contact">Contact</a>
  </li>
  </ul>
```

child routs / Nested Routs

in any angular applications some routs may only viewed between another routs

```
const routes: Routes = [

  {
    path:'',component:HomeComponent
  },
  {
    path:'about',component:AboutComponent,
childern:
[{
path:'aboutchild1':component:aboutComp1
},
{
path:'aboutchild2':component:aboutComp2
}]
},
  {
    path:'contact',component:ContactComponent
  }

];
```

Parameterised Routs

We can pass the aruguments in the link url

```
[
{
path:'user/id',component:usercomponent
}
]
```

Id	Empname	Salary
101	Ravi	4500
102	Rani	5241
103	Deepa	8745

```
<a [routerLink]="['/user',u.id]" routerLinkActive="active">

Name:-hp lap top
```

Batter 1.2v
Color :red
Hdd :2Tb
Ram :16 Gb
Category:laptop

Name:-watch
Batter 1.2v
Color :red
Company:titan
Category:watch

localhost:4200/user/103 // url looks like

Query Params

Based on the query

```
<li>
      <a routerLink="/products" routerLinkActive="Active"
[queryParams]="{'category':'men'}">Products</a>
      </li>
```

route gaurds
protecting the routs

canDeactive-->away from the current route
CanActive-->Gaurd navigation to a route
CanActiveChild-->to a child route
canLoad-->to a feature module loaded asynchronously
Resolve-->perform route data retrival before route activation

three steps to use a route guard

1)build the route gaurd
2)register the guard with angular dependency injection system
3)tie the guard to aroute

1)create service file

emp-desc-services.ts

export class empdesc implements canDeactive implements CanDeactive <>interface
{

```
candeactive(Componment:empcommopment):boolean{
if(component.emp.dirty)
{
return confirm('are you sure want to delete')
}
}
```

2)in appmodule.ts

import service in appmodule
in the providers also give service name

3)

```
[
{
path:'user',component:usercomponent,

}
]
```

canDeactive[servicename]

Simple Root Module
We use an Angular Module as the starting point for our applications. When we decorate a class with @NgModule we are telling Angular important information about this Angular Module's role.

```
// app.module.ts
@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent],
})
export class AppModule { }
```

 # Bootstrapping our App
The example below shows how we will bootstrap AppModule using the browser platform. Again, by convention, we name the AppModule file app.module.ts.

```
// main.ts
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

 # Declaring
We create custom components, directives and pipes in our application. We can tell Angular that this module will use these by declaring the in the @NgModule declarations property.

```
// app.module.ts
@NgModule({
  imports: [BrowserModule],
  declarations: [
    AppComponent,
    VehicleListComponent,
    VehicleSelectionDirective,
    VehicleSortingPipe
  ],
  bootstrap: [AppComponent],
})
export class AppModule { }
```

Imports

We imported BrowserModule, which itself imports the CommonModule which contains the common built-in directives such as NgIf and NgFor. By importing BrowserModule, we in turn get what CommonModule has to offer too.

What if we want to use form functionality such as ngModel or http? We can import those modules, too. This makes these features available in or AppModule.

```
// app.module.ts
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule
  ],
```

 # Providers

We often have services that we want to share within our app. How do we allow all of our components to use a service? We use Angular Modules.

We can add add providers to the application's root dependency injector in our AppModule. Since our AppModule is our root module, this will make the services available everywhere in the application.

```
providers: [
    LoggerService,
    VehicleService,
    UserProfileService
  ],
```

Ahead-of-time (AOT) compilation

An Angular application consists mainly of components and their HTML templates. Because the components and templates provided by Angular cannot be understood by the browser directly, Angular applications require a compilation process before they can run in a browser.

The Angular ahead-of-time (AOT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase before the browser downloads and runs that code. Compiling your application during the build process provides a faster rendering in the browser.

Here are some reasons you might want to use AOT.

Faster rendering With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the application first.

Fewer asynchronous requests The compiler inlines external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.

Smaller Angular framework download size There's no need to download the Angular compiler if the application is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.

Detect template errors earlier The AOT compiler detects and reports template binding errors during the build step before users can see them.

Better security AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

Choosing a compiler
Angular offers two ways to compile your application:

Just-in-Time (JIT), which compiles your application in the browser at runtime. This was the default until Angular 8.
Ahead-of-Time (AOT), which compiles your application and libraries at build time. This is the default since Angular 9.

Comparison between Ahead of Time (AOT) and Just in Time (JIT) –

JIT AOT
JIT downloads the compiler and compiles code exactly before Displaying in the browser.
Loading in JIT is slower than the AOT because it needs to compile your application at runtime.
JIT is more suitable for development mode.
Bundle size is higher compare to AOT.
You can run your app in JIT with this command:
ng build

AOT has already complied with the code while building your application, so it doesn’t have to compile at runtime.

Loading in AOT is much quicker than the JIT because it already has compiled your code at build time.
AOT is much suitable in the case of Production mode.

Bundle size optimized in AOT, in results AOT bundle size is half the size of JIT bundles.

ng build --prod
You can catch template binding error at display time.You can catch the template error at building your application.

What are Feature Modules?

A feature module is an ordinary Angular module for all intents and purposes, except the fact that isn't the root module. Basically - it's just an additional class with the @NgModule decorator and registered metadata. The feature module partitions areas of the application and collaborates with the root module (and with further feature modules).

The main aim for feature modules is delimiting the functionality that focuses on particular internal business inside a dedicated module, in order to achieve modularity. In addition, it restricts the responsibilities of the root module and assists to keep it thin. Another advantage - it enables to define multiple directives with an identical selector, which means avoiding from directive conflicts.

There are five types of feature modules: Domain, Routed, Routing, Service and Widget. Each of them concentrates and provides a particular type of utilities.

creating feature module

mg g m car
ng g c car/automatic
import carmodule in app module

What is an Angular Service

Service is a piece of reusable code with a focused purpose. A code that you will use in many components across your application

Our components need to access the data. You can write data access code in each component, but that is very inefficient and breaks the rule of single responsibility. The Component must focus on presenting data to the user. The task of getting data from the back-end server must be delegated to some other class. We call such a class a Service class

What Angular Services are used for
- ✓ Features that are independent of components such a logging services
- ✓ Share logic or data across components
- ✓ Encapsulate external interactions like data access

Advantageous of Angular Service

- ✓ Services are easier to test.
- ✓ They are easier to Debug.
- ✓ We can reuse the service at many places.

Create a new file under the folder src/app and call it **product.ts**

Product Angular Service

Next, let us build an Angular Service, which returns the list of products.

ng g s product

Invoking the ProductService

The Next step is to invoke the ProductService from the component. Open the **app.componet.ts**

product service

product.service.ts
Step-1

```
export class ProductService {
 constructor() { }
 public  getProducts() {
  let products:Product[];
  products=[
     new Product(1,'Memory Card',500),
     new Product(2,'Pen Drive',750),
     new Product(3,'Power Bank',100)
  ]
  return products;
}
}
```

product.ts

```
export class Product {
   constructor(productID:number,   name: string ,   price:number) {
      this.productID=productID;
      this.name=name;
      this.price=price;
   }
   productID:number ;
   name: string ;
   price:number;
}
```

Step-2
product.ts

app.component.ts

```
import { Product } from './product';
const product_service= new InjectionToken<ProductService>("product_service");
products:Product[]=[];
 productService;
 constructor(){
  this._produService=new _produService();
 }

 getProducts() {
  this.products=this._produService.getProducts();
 }
```

Step-3
app.component.ts

```
<div class="container">
 <h1 class="heading"><strong>Services </strong>Demo</h1>
 <button type="button" (click)="getProducts()">Get Products</button>
  <div class='table-responsive'>
      <table class='table'>
        <thead>
          <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Price</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let product of products;">
            <td>{{product.productID}}</td>
            <td>{{product.name}}</td>
            <td>{{product.price}}</td>
          </tr>
        </tbody>
      </table>
    </div>
</div>
```

Angular Dependency Injection

Dependency Injection (DI) is a technique in which a class receives its dependencies from external sources rather than creating them itself.

```
constructor(private _produService:ProductService ){
   //this._produService=new _produService();
 }
```

It creates & maintains the Dependencies and injects them into the Components, Directives, or Services.

A ReflectiveDependency injection container used for instantiating objects and resolving dependencies. Deprecated: from v5 - slow and brings in a lot of code, Use Injector. create instead.

Providers
Angular Providers allows us to register classes, functions, or values (dependencies) with the Angular Dependency Injection system.
The Providers are registered using the token. The tokens are used to locate the provider.

We can create three types of the token. Type Token, string token & Injection Token.

Provide
The first property is Provide holds the Token or DI Token. The Injector uses the token to locate the provider in the Providers array. The Token can be either a type, a string or an instance of InjectionToken.

Provider
The second property is the Provider definition object. It tells Angular how to create the instance of the dependency. The Angular can create the instance of the dependency in four different ways.
1) It can create a dependency from the existing service class (useClass).
2) It can inject a value, array, or object (useValue).
3) It can use a factory function, which returns the instance of service class or value (useFactory).
4) It can return the instance from an already existing token (useExisting).

DI Token
The Injector maintains an internal collection of token-provider in the Providers array. The token acts as a key to that collection & Injector use that Token (key) to locate the Provider.
The DI Token can be either type, a string or an instance of InjectionToken.

Type Token
the type being injected is used as the token.
providers :[{ provide: ProductService, useClass: ProductService }]

String token
Instead of using a type, we can use a string literal to register the dependency. This is useful in scenarios where the dependency is a value or object etc, which is not represented by a class.

```
{provide:'PRODUCT_SERVICE', useClass: ProductService },
{provide:'USE_FAKE', useValue: true },
{provide:'APIURL', useValue: 'http://SomeEndPoint.com/api' },
```

You can then use the Inject the dependency using the @Inject method

```
class ProductComponent {
  constructor(@Inject('PRODUCTSERVICE') private prdService:ProductService,
@Inject('APIURL') private apiURL:string ) {
}
```

Injection Token

The Problem with the string tokens is that two developers can use the same string token at a different part of the app. You also do not have any control over the third-party modules, which may use the same token. If the token is reused, the last to register overwrites all previously registered tokens.

```
export const API_URL= new InjectionToken<string>('');
```

Reactive Extension for JavaScript.(RxJS)

what is reactive programming?

reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change

RxJS is a library for reactive programming using observables that makes it easier to compose asynchronous or call back-based code

reactive programming is just different way of building software applications

what is Rxjs?

is a library for composing asynchronous and event based programs by using observable sequence

RxJS consists of 3 main things

--observable

data types,observers,schedulers,subjects

-operators

array methods

--map

--filter

--reduce

-helps us in data stream

4 observable

-- its a data stream
ecom web site order status?

subscriber
it will listen to the observable for data changes /updates

ex...
youtube chanel
observable
data is posted every single day on my channel
scubribe
you got notified everytime i post a video
there is a change in data

Rxjs important Concepts

these concepts inRxJS which slove async event management are

observable:- represents the idea of an invokable collection of future values or events

observer:- is a collection of callbacks that knows how to listen to values delivered by the observable

Subscription: represents the execution of an observable is primarily useful for cancelling the execution

operators: are pure functions that enable a functional programming style of dealing with collections with operations like map,filter,concat,reduce,etc

subject: is the equivalent to an eventemitter and the only way of multicasting a value or event to multiple observers

schedulers : are centralized dispatchers to control concurrency allowing us to coordinate when computation happens eg settimeout

Observables

are emits data over a period of time

needs to be subscribed else it won't do anything on its own

has 3 methods namely->next,complete and error

```
persons:Observable<string>;

persons=new Observable(function(observer)
{
try
{
observer.next("ram")
observer.next("ravi")
observer.next("sita")
observer.next("Geetha")
observer.next("john")
}catch(e){observer.error(e);}
}
);
this.persons.subscribe(data=>{
```

console.log(data)}

Operators

are very important part of RxJS

RxJS library provides a lot of useful operators which helps us write clean code and reduce lot of effect in writing custom logic(time interval) which leads many bus

an operator is a pure function

pure function will also return same value when passed with same input value
Ex->a+b
an operator takes in observable as input and output will also be an observable

types of operators

Creation
Mathematical
Join or Combination
Transformation
Filtering
Utility
Conditional
Multicasting
Error handling

of operator

make observable from a string or array or an object

it used we want to pass a variable which has to be observable instead of an array or string we use of operator

studnm=['ram','ravi','deepa','sita']

```
students:Observable<String[]> =of(this.studnm)
studobj={
  id:102,
  name:'ravi'
}
studentObj:Observable<any>=of(this.studobj)
btnClick()
{
const source = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
const subscribe = source.subscribe(val => console.log(val));
const subscribe1 = this.students.subscribe(val => console.log(val));
const StudObjj = this.studentObj.subscribe(val => console.log(val));
}
```

Operator from

will create an observable from an array, an array like object, a promise an iterable object or an observable like object

it will always take array or array like objects

```
studnm=['ram','sita','gopal']
names:Observable>String>=from(this.studnm)

this.studnm.subscribe(data=>{console.log(data);
}
)
```

fromEvent()

creats an observable that emits events of a specific type comming from the given event target

we can bind target elements and methods to make sure we get observable as output

```
const clicks = fromEvent(document, 'click');
const source = interval(1000);
const subscribe = source.subscribe(val => console.log(val));
```

```
clicks.subscribe(x => console.log(x));
```

interval()

RxJS interval() operator is a creation operator used to create an observable that emits a sequence of integers every time for the given time interval on a specified SchedulerLike. It emits incremented numbers periodically in time.

```
const source = interval(1000);
const subscribe = source.subscribe(val => console.log(val));
```

debounceOperator

RxJS debounceTime() operator is a filtering operator that emits a value from the source Observable only after completing a particular period without another source emission.

Ts file

```
this.searchForm.get('name').valueChanges
.pipe(
  debounceTime(2000)
  )
  .subscribe((data: any) =>
      {
    console.log(data);
  }
    );
  }
```

Html file

```
<div [formGroup]="searchForm">
  <input type="text" formControlName="name" (keyup)="readValue()">
  </div>
```

take operator

emits only the first count values emitted by the source observables

how may values to except
take(2)

takewhile()

emitted by the source observable so long as each value statisfies the given predicate and then completes as soon as the predicate is not statisifed

ts file

```
this.searchForm.get('name').valueChanges
.pipe(
  take(2),   //take values

 takeWhile((v:any)=>this.checkCondition(v)),
  debounceTime(2000)
```

```
  )
  .subscribe((data: any) =>
      {
    console.log(data);
  }
    );
  }

checkCondition(value:any)

{
  console.log("value",value)
  return value.length > 5 ? false : true
}
```

.html file

```
<div [formGroup]="searchForm">
  <input type="text" formControlName="name" (keyup)="readValue()">
  </div>
```

takeLast

emits the n values from the source as specified the count argument

```
mobiles=['samsung','nokia','vu','readmi','realmi']
mobileObj:observable<string>=from(this.mobiles)
   this.mobileObj.pipe(
      takeLast(2),
    )
    .subscribe(data=>
    {
      console.log(data)
    }
    )
```

First operator

it emits first value from observable

last Operator

it emits last value from observable

elementAt
operator will give single value from the source observable based upon the index given

it is an array index

filter operator
filter operator will give filter the values from source observable based on the predicate function given

```
all_nums = of(1, 6, 5, 10, 9, 20, 40);

let final_val = this.all_nums.pipe(filter(a => a % 2 === 0));
final_val.subscribe(x => console.log("The filtered elements are "+x));

const example = this.all_nums.pipe(filter(num => num % 2 === 0));
const subscribe = example.subscribe(val => console.log(`Even number: ${val}`));
```

distinct
---
operator will give all the values from the source observables that are distinct when compared with the previous values

```
this.mobileObj.pipe(
  distinct())
.subscribe(data=>console.log(data))

  }
    );
```

skip operator
will give back an observable that will skip the first occurance of count items taken as input

```
this.mobileObj.pipe(
```

```
  distinct(),
  skip(2)
  )
.subscribe(data=>console.log(data))
  }
    );
  }
```

count operator

will give back an observable that will skip the first occurance of count items taken as input

count()

max Operator

Min operator

Http Service
Most front-end applications need to communicate with a server over the HTTP protocol, to download or upload data and access other back-end services. Angular provides a client HTTP API for Angular applications, the HttpClient service class in @angular/common/http.

The HTTP client service offers the following major features.
- The ability to request typed response objects.
- Streamlined error handling.
- Testability features.
- Request and response interception.

Prerequisites
Before working with the HttpClientModule, you should have a basic understanding of the following:
- TypeScript programming
- Usage of the HTTP protocol
- Angular app-design fundamentals, as described in Angular Concepts
- Observable techniques and operators. See the Observables guide.

creating new angular project
- ng new angular-httpclient-app
- cd angular-httpclient-app
- npm install bootstrap

change angular.json

```
"styles": [
          "node_modules/bootstrap/dist/css/bootstrap.min.css",
          "src/styles.css"
        ]
```

Generate components for Angular
- ng g c employee-create
- ng g c employee-edit
- ng g c employee-list

Configure JSON Server in Angular
- npm i json-server --save
- mkdir server
- cd server
- npx json-server --watch db.json

You can check your local db.json file on this URL http://localhost:3000/employees.

Enable Routing Service in Angular

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';


import { EmployeeCreateComponent } from './employee-create/employee-create.component';
import { EmployeeEditComponent } from './employee-edit/employee-edit.component';
import { EmployeesListComponent } from './employees-list/employees-list.component';

const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: 'create-employee' },
  { path: 'create-employee', component: EmployeeCreateComponent },
  { path: 'employees-list', component: EmployeesListComponent },
  { path: 'employee-edit/:id', component: EmployeeEditComponent }
];

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

export class AppRoutingModule { }
```

Import HttpClient API

to access the external server to fetch the data using the RESTful API in Angular with HttpClient service

In order to use HttpClient API to make the communication with Http remote server, you must set up this service in your Angular app.

in app.module.ts

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    HttpClientModule
  ]
})
```

In order to create CRUD operations using RESTful API in Angular, we need to generate employee.ts class and rest-api.service.ts files.

ng g m shared/Employee
in shared/employee.ts

```
export class Employee {
  id: string;
  name: string;
  email: string;
  phone: number;
}
```

Next, generate RestApiService class:

- ng g s shared/rest-api
- shared/rest-api.service.ts

```
import { ErrorHandler, Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Employee } from '../shared/employee';
import { Observable, throwError } from 'rxjs';
import { retry, catchError } from 'rxjs/operators';

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

  apiURL = 'http://localhost:3000';
  constructor(private http:HttpClient) {}
    httpOptions = {
      headers: new HttpHeaders({
        'Content-Type': 'application/json'
      })


   }
   getEmployees(): Observable<Employee> {
    return this.http.get<Employee>(this.apiURL + '/employees')
    .pipe(
      retry(1),
      catchError(this.handleError)

    )
  }

  createEmployee(employee: { name: string; email: string; phone: number; }):
Observable<Employee> {
    return this.http.post<Employee>(this.apiURL + '/employees', JSON.stringify(employee),
this.httpOptions)
    .pipe(
```

```
    retry(1),
    catchError(this.handleError)
   )
  }

  handleError(error:any) {
    let errorMessage = '';
    if(error.error instanceof ErrorEvent) {
      // Get client-side error
      errorMessage = error.error.message;
    } else {
      // Get server-side error
      errorMessage = `Error Code: ${error.status}\nMessage: ${error.message}`;
    }
    window.alert(errorMessage);
    return throwError(errorMessage);
  }
}
```

Access HttpClient from Angular Component

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { EmployeeCreateComponent } from './employee-create/employee-create.component';
import { EmployeeEditComponent } from './employee-edit/employee-edit.component';
import { EmployeeListComponent } from './employee-list/employee-list.component';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule } from '@angular/forms';


@NgModule({
  declarations: [
    AppComponent,
    EmployeeCreateComponent,
    EmployeeEditComponent,
    EmployeeListComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    FormsModule
```

```
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Create Data using Angular
employee-create.component.html

```
<div class="container custom-container">
  <div class="col-md-12">
    <h3 class="mb-3 text-center">Create Employee</h3>

    <div class="form-group">
      <input type="text" [(ngModel)]="employeeDetails.name" class="form-control"
placeholder="Name">
    </div>

    <div class="form-group">
      <input type="text" [(ngModel)]="employeeDetails.email" class="form-control"
placeholder="Email">
    </div>

    <div class="form-group">
      <input type="text" [(ngModel)]="employeeDetails.phone" class="form-control"
placeholder="Phone">
    </div>

    <div class="form-group">
      <button class="btn btn-success btn-lg btn-block" (click)="addEmployee()">Create
Employee</button>
    </div>

  </div>
</div>
```

Go to employee-create.component.ts

```
import { Component, OnInit, Input } from '@angular/core';
import { Router } from '@angular/router';
import { RestApiService } from "../shared/rest-api.service";

@Component({
  selector: 'app-employee-create',
  templateUrl: './employee-create.component.html',
  styleUrls: ['./employee-create.component.css']
```

```
})
export class EmployeeCreateComponent implements OnInit {

  @Input() employeeDetails = { name: '', email: '', phone: 0 }

  constructor(
    public restApi: RestApiService,
    public router: Router
  ) { }

  ngOnInit() { }

  addEmployee() {
    this.restApi.createEmployee(this.employeeDetails).subscribe((data: {}) => {
      this.router.navigate(['/employees-list'])
    })
  }

}
```

employees-list.component.html

```
<div class="container custom-container-2">

  <!-- Show it when there is no employee -->
  <div class="no-data text-center" *ngIf="Employee.length == 0">
    <p>There is no employee added yet!</p>
    <button class="btn btn-outline-primary" routerLink="/create-employee">Add Empoyee</button>
  </div>

  <!-- Employees list table, it hides when there is no employee -->
  <div *ngIf="Employee.length !== 0">
    <h3 class="mb-3 text-center">Employees List</h3>

    <div class="col-md-12">
      <table class="table table-bordered">
        <thead>
          <tr>
            <th scope="col">User Id</th>
            <th scope="col">Name</th>
            <th scope="col">Email</th>
            <th scope="col">Phone</th>
            <th scope="col">Action</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let employee of Employee">
            <td>{{employee.id}}</td>
```

```
          <td>{{employee.name}}</td>
          <td>{{employee.email}}</td>
          <td>{{employee.phone}}</td>
          <td>
            <span class="edit" routerLink="/employee-edit/{{employee.id}}">Edit</span>
            <span class="delete" (click)="deleteEmployee(employee.id)">Delete</span>
          </td>
        </tr>
      </tbody>
    </table>
  </div>

 </div>

</div>
```

employees-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { RestApiService } from "../shared/rest-api.service";

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

 Employee: any = [];

 constructor(public restApi: RestApiService) { }

 ngOnInit(): void {
  this.loadEmployees()
 }

 loadEmployees() {
  return this.restApi.getEmployees().subscribe((data: {}) => {
   this.Employee = data;
  })
 }

}
```