**Angular**

**Def:-Angular is a framework that use to build client side application and it is specially great building single page application,**

**Where parts of the view get refresh asynchronously without reload the entire page**

**Components**

* **Basic building blocks of angular application.**
* **It can be defined using @Component decorator.**
* **Every component consist of three parts.**
* the template which loads the view for the component, a stylesheet which defines the look and feel for the component, and a class that contains the business logic for the component.
* Command to create component is  **ng generate component test Or ng g c test**

**Modules**

Angular application just a collection of many individual modules.

Every module represents feature area in our application.

A module is a place where we can group components, directives, services, and pipes.

Module decides whether the components, directives, etc can be used by other modules, by exporting or hiding these elements.

 Every module is defined with a @NgModule decorator.

By default, modules are of two types:

* Root Module
* Feature Module

Every application can have only one root module whereas, it can have one or more feature modules.

A root module imports **BrowserModule**, whereas a feature module imports **CommonModule**.

In the application that we created before, one can see that the root module is defined inside **app.module.ts**

To create a feature module, run the following command: **ng g m test-module**

As one can see, **CommonModule** is imported since this is a feature module.

**Directives**

**Def:-**  Directives are defined as classes that can add additional behavior to the elements in the template or modify existing behavior.

* It can be declared with a **@Directive** decorator.
* Every directive has its own behavior and can be imported into various components of an application.

**When to use a directive?**

* Consider an application, where multiple components need to have similar functionalities. The norm thing to do is by adding this functionality individually to every component but, this task is tedious to perform. In such a situation, one can create a **directive** having the required functionality and then, import the directive to components which require this functionality.
* Basically directives are used to manipulate the DOM, for example adding/removing the element from DOM or changing the appearance of the DOM elements.

**Why to use a directive?**

 Use Angular's built-in directives to manage forms, lists, styles, and what users see.

**Types of directives**

1. **Component directives**
2. **Attribute directives**
3. **Structural Directives**
4. **Component Directive**

These form the main class in directives. **Instead** of @Directive decorator we use **@Component** decorator to declare these directives. These directives have a view, a stylesheet and a selector property.

1. **Attribute directives**

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components.

Many NgModules such as the [RouterModule](https://angular.io/guide/router" \o "Routing and Navigation) and the [FormsModule](https://angular.io/guide/forms" \o "Forms) define their own attribute directives. The most common attribute directives are as follows:

1. **ngClass:-** Adds and removes a set of CSS classes.
2. **ngStyle:-** Adds and removes a set of HTML styles.
3. **ngModel:-** Adds two-way data binding to an HTML form element.

**ngClass**

**Why do we use ngClass?**

The ng-class directive dynamically binds one or more CSS classes to an HTML element.

The value of the ng-class directive can be a string, an object, or an array. If it is a string, it should contain one or more, space-separated class names.

**To add or remove a single class, use**[**class binding**](https://angular.io/guide/class-binding)**rather than [NgClass](https://angular.io/api/common/NgClass).**

**Binding to single class**

Ex:-

HTML File

<h1 [class.onSale]="onSale">{{title}}</h1>

TS File

title = 'hello-angular';

onSale=true;

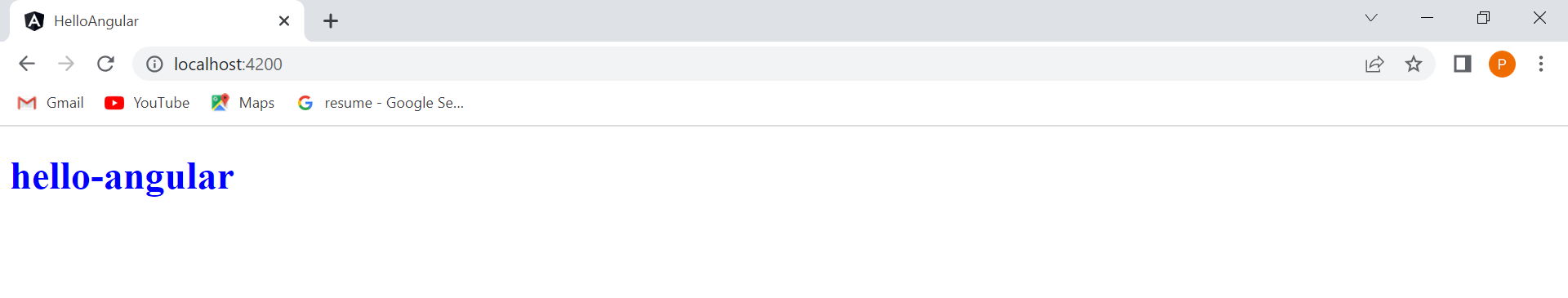
CSS File

.onSale{

    color:blue;

}

**OUT PUT**



**Binding to multiple class**

[class]="classExpression"

The expression can be one of:

* A space-delimited string of class names.
* An object with class names as the keys and truthy or falsy expressions as the values.
* An array of class names.

**Example for string value**

HTML File

<h1 [class]="classExpression">{{title}}</h1>

TS File

export class AppComponent {

  title = 'hello-angular';

  classExpression="blueColor italicStyle";

}

CSS File

.blueColor{

    color:blue;

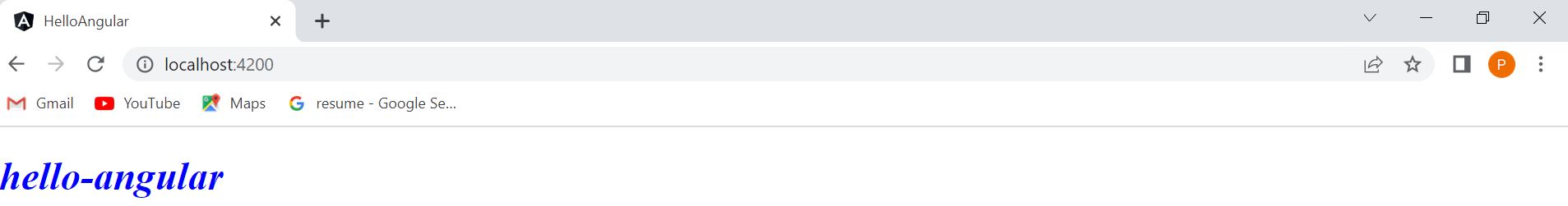
}

.italicStyle{

    font-style: italic;

}

**OUTPUT**



**Example for Object value**

HTML File

<h1 [class]="classExpression">{{title}}</h1>

TS File

export class AppComponent {

  title = 'hello-angular';

  classExpression={

    blueColor:false,

    italicStyle:true

  };

}

CSS File

.blueColor{

    color:blue;

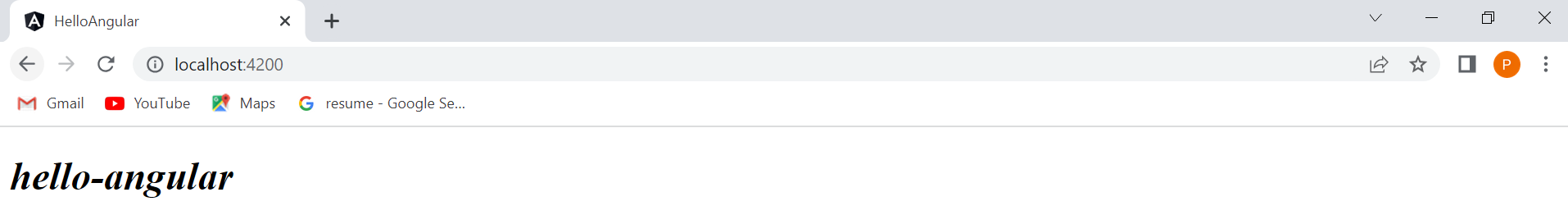
}

.italicStyle{

    font-style: italic;

}

**OUTPUT**



**Example for Array value**

HTML File

<h1 [class]="classExpression">{{title}}</h1>

TS File

export class AppComponent {

  title = 'hello-angular';

  classExpression=['blueColor','italicStyle'];

}

.blueColor{

    color:blue;

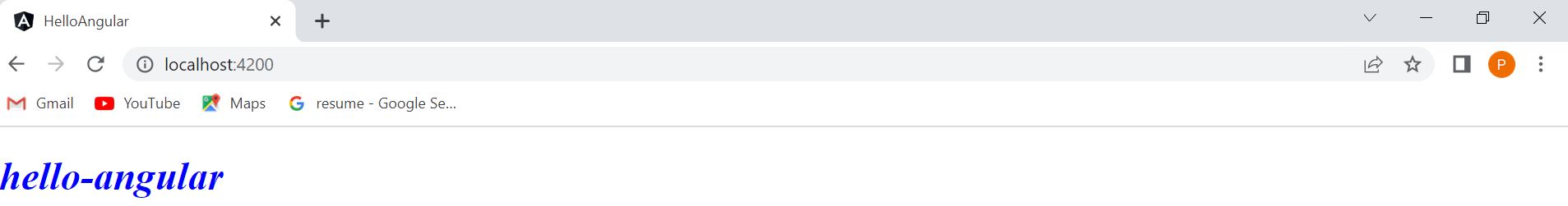
}

.italicStyle{

    font-style: italic;

}

**OUTPUT**



**Example for NgClass**

**Method also we can write to ngClass**

**1)**

<div [ngClass]="setCurrentClasses()">This div is initially saveable, unchanged, and special.</div>

TS File

 canSave=true;

  isUnchanged = true;

  isSpecial = false;

  currentClasses:  Record<string, boolean> = {};;

 setCurrentClasses(){

  this.currentClasses =  {

    saveable: this.canSave,

    modified: !this.isUnchanged,

    special:  this.isSpecial

  };

  return this.currentClasses;

 }

CSS File

.saveable{

background-color: green;

}

.modified{

color: yellow;

}

.special{

    font-style: italic;

}

div{

    color:white;

}

**2)**

HTML File

<div [ngClass]="currentClasses">This div is initially saveable, unchanged, and special.</div>

TS File

 canSave=true;

  isUnchanged = true;

  isSpecial = false;

  currentClasses =  {

    saveable: this.canSave,

    modified: !this.isUnchanged,

    special:  this.isSpecial

  };

CSS File

.saveable{

background-color: green;

}

.modified{

color: yellow;

}

.special{

    font-style: italic;

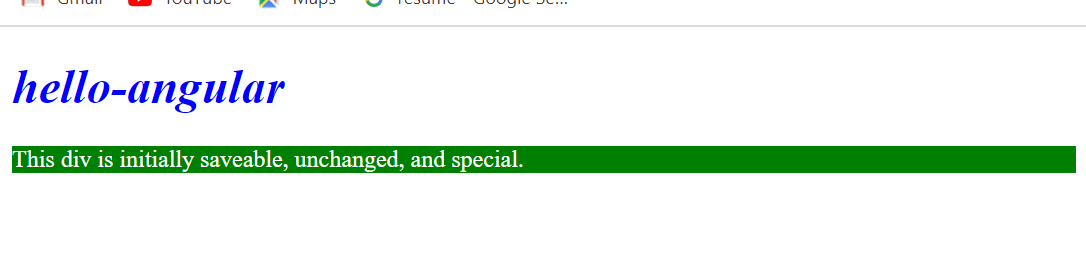
}

div{

    color:white;

}

**OUTPUT**



**NgStyle**

Use [NgStyle](https://angular.io/api/common/NgStyle) to set multiple inline styles simultaneously, based on the state of the component.

 The NgStyle attribute **is used to change the appearance and behavior of the element**.

**Note that binding an array to [**[**style**](https://angular.io/api/animations/style)**] is not supported.**

<div [ngStyle]="ngStyleExample">Example for ngStyle</div>

 ngStyleExample={

  color:'blue',

  fontStyle:'italic'

 }

**Assign method to ngStyle**

<div [ngStyle]="setCurrentStyles()">Example for ngStyle</div>

 setCurrentStyles(){

 let ngStyleExample={

    color:'blue',

    fontStyle:'italic'

   }

  return ngStyleExample;

 }

**3) Structural Directive**

These directives are generally used to manipulate DOM elements.

Every structural directive has a ‘ \* ’ sign before them.

Ng-template tag its basically has container

1. \*ngIf directive

In html file

<p>structural-directive works!</p>

<h1>NgIf Directive</h1>

<div \*ngIf="isDisplay">

    <h4>If Block</h4>

</div>

<h1>NgIf and else Example</h1>

<div \*ngIf="isDisplay; else elseBlock">

    <h4>If Block</h4>

</div>

<ng-template #elseBlock>

    <h4>Else Block</h4>

</ng-template>

<h1>Another Syntax to display if and else block</h1>

<div \*ngIf="isDisplay; then ifBlock; else elseBlock">

</div>

<ng-template #ifBlock>

    <h4>If Block</h4>

</ng-template>

<ng-template #elseBlock>

    <h4>ElseBlock</h4>

</ng-template>

In ts file

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

@Component({

  selector: 'app-structural-directive',

  templateUrl: './structural-directive.component.html',

  styleUrls: ['./structural-directive.component.css']

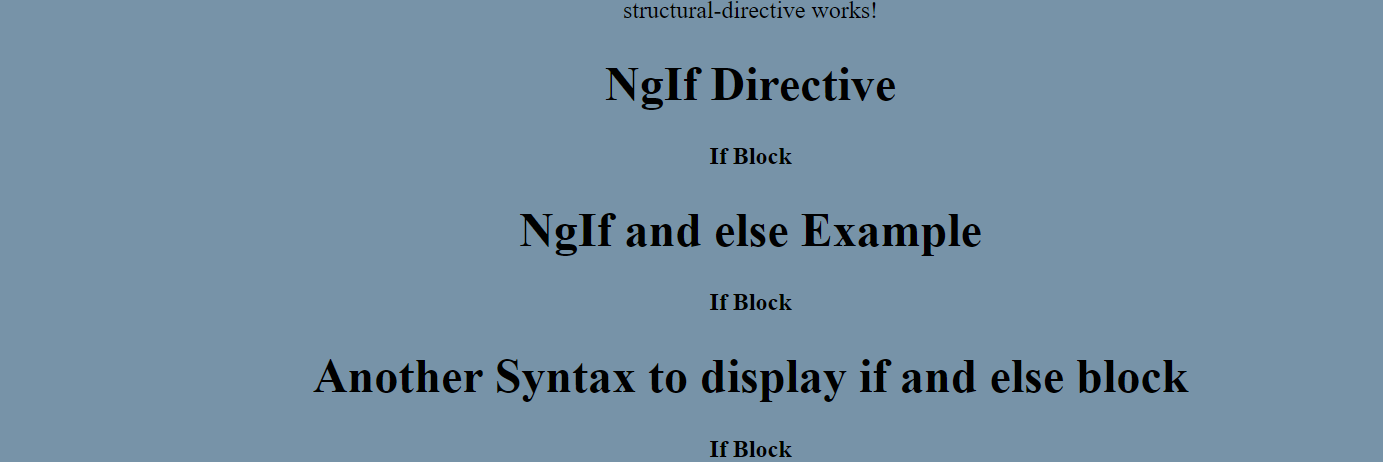
})

export class StructuralDirectiveComponent {

isDisplay=true;

}

Out put



1. **\*ngSwitch**

**It is used when we want to compare against multiple values**

<h1>Example for ngSwitch</h1>

<div [ngSwitch]="color">

    <p \*ngSwitchCase="'red'">you picked red color</p>

    <p \*ngSwitchCase="'blue'">you picked blue color</p>

    <p \*ngSwitchCase="'green'">you picked green color</p>

</div>

**Out put**



1. **\*ngFor**

**To render the list of elements**

<h1>Example for ngFor</h1>

<div \*ngFor="let color of colors, index as i, first as f, last as l, odd as o, even as e">

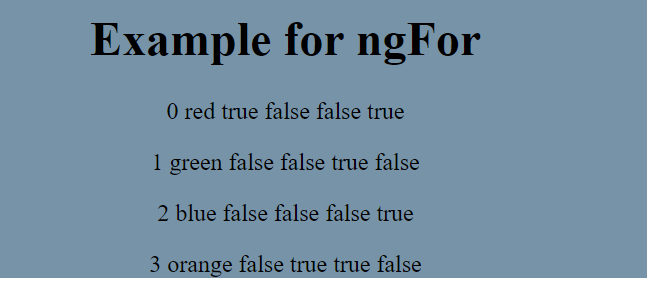
    <p>

        {{i+" "}}{{color+" "}}{{f+" "}}{{l+" "}}{{o+" "}}{{e}}

    </p>

</div>

**Output**



**Data Binding**

**Def: A process that allows applications to display data values to a user and respond to user actions.**

**User actions include clicks, touches, keystrokes and so on.**

**Communication between typescript code to template.**

**Types of data binding**

1. **String interpolation {{data}}**
2. **Property binding [property]=”data”**
3. **Event Binding (event)=”expression”**
4. **Two-way-binding [(ngModel)]=”data”**

**String interpolation**

**Def: Bind the data from class to template**

**Why: Some time we want to change the value dynamically and display value to user.**

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

@Component({

  selector: 'app-test',

  template: `<p>Wel Come

   <span style="color:red">{{name}}</span> to Angular tutorial</p>`,

  styleUrls: ['./test.component.css']

})

export class TestComponent {

name="Pooja";

}

**Output**



**Using string interpolation we can do the following operation.**

**1. Evaluate the expression and display the result to user.**

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

@Component({

  selector: 'app-test',

  template: `<p>After added two numbers the result is: {{2+2}}</p>`,

  styleUrls: ['./test.component.css']

})

export class TestComponent {

}

**Output**

After added two numbers the result is: 4

**2.String concatenation**

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

@Component({

  selector: 'app-test',

  template: `<p>{{'Wel'+'Come '}}</p>

   <p>{{'Wel Come'+name}}`,

  styleUrls: ['./test.component.css']

})

export class TestComponent {

name="Pooja";

}

**Output**

Wel Come

Wel ComePooja

**3.javaScript properties and methods**

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

@Component({

  selector: 'app-test',

  template: `<p>{{name.length}}</p>

   <p>{{name.toUpperCase()}}

   <h2>{{greaterUser()}}</h2>`,

  styleUrls: ['./test.component.css']

})

export class TestComponent {

name="Pooja";

greaterUser(){

  return "Hello "+this.name;

}

}

**Output:**

5

POOJA

**Hello Pooja**

**Using interpolation we can not able perform the following operation**

**1.assigning the expression to variable**

**2.Access gloable variable such as window, screen and son on**

**For example to find out the current page url we get error**

1.import { Component } from '@angular/core';

@Component({

  selector: 'app-test',

  template: `<h3>{{window.location.href}}</h3>`,

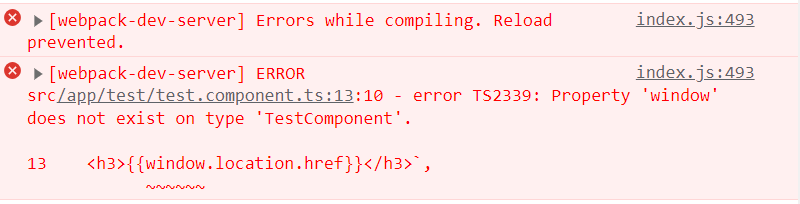
  styleUrls: ['./test.component.css']

})

export class TestComponent {

}

**Error**



2.import { Component } from '@angular/core';

@Component({

  selector: 'app-test',

  template: `<h3>{{siteUrl}}</h3>`,

  styleUrls: ['./test.component.css']

})

export class TestComponent {

name="Pooja";

greaterUser(){

  return "Hello "+this.name;

}

siteUrl=window.location.href;

}

**Output:**

### http://localhost:4200/

**Property Binding**

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

@Component({

  selector: 'app-test',

  template: `<input type="text" value="Pooja">`,

  styleUrls: ['./test.component.css']

})

export class TestComponent {}

**Difference between HTML attribute and DOM property**

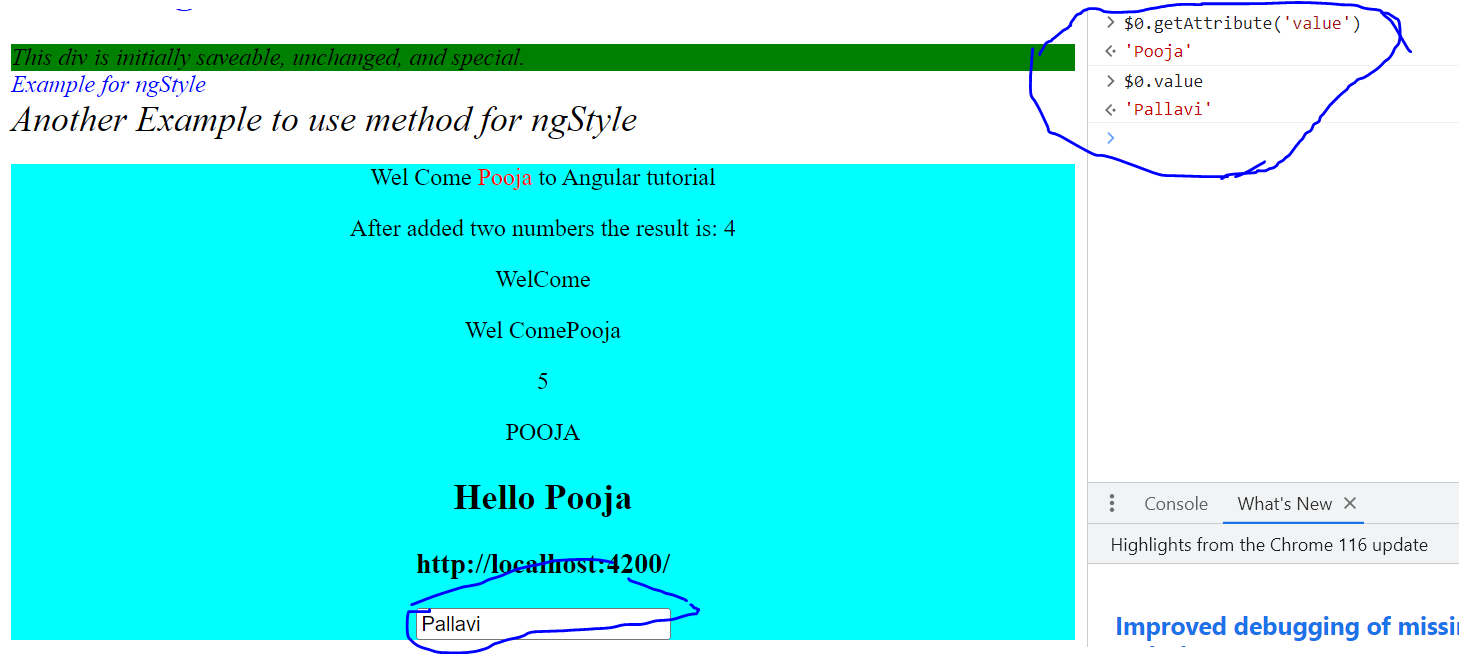
**In console type $0.getAttribute(‘value’) then enter we see the output that is value ($0 represents the current element)**

**Then type $0.value we get the value of input filed** 

**If we change value in input field**

**Then write the same command $0.getAttribute(‘value’) we get same value but if we write command $0.value we get what ever the data written in DOM .**

**So, attribute didn’t change but value property change.**



1. **Attributes and properties are not the same**
2. **Attributes are defined by the html**
3. **Properties are defined by the DOM**
4. **Attributes are initialize DOM properties and then they are done. Attributes value cannot change once they are initialized.**
5. **Property values however can change.**

**We can also use interpolation to bind the id of element.**

**If interpolation is there Why do we need property binding ? because interpolation works only for string values.**

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

@Component({

  selector: 'app-test',

  template: `

   <input type="text" value="Pooja"><br><br>

   <input [id]="myId" type="text" value="Pooja"><br><br>

   <input id={{myId}} value="Pallavi">`,

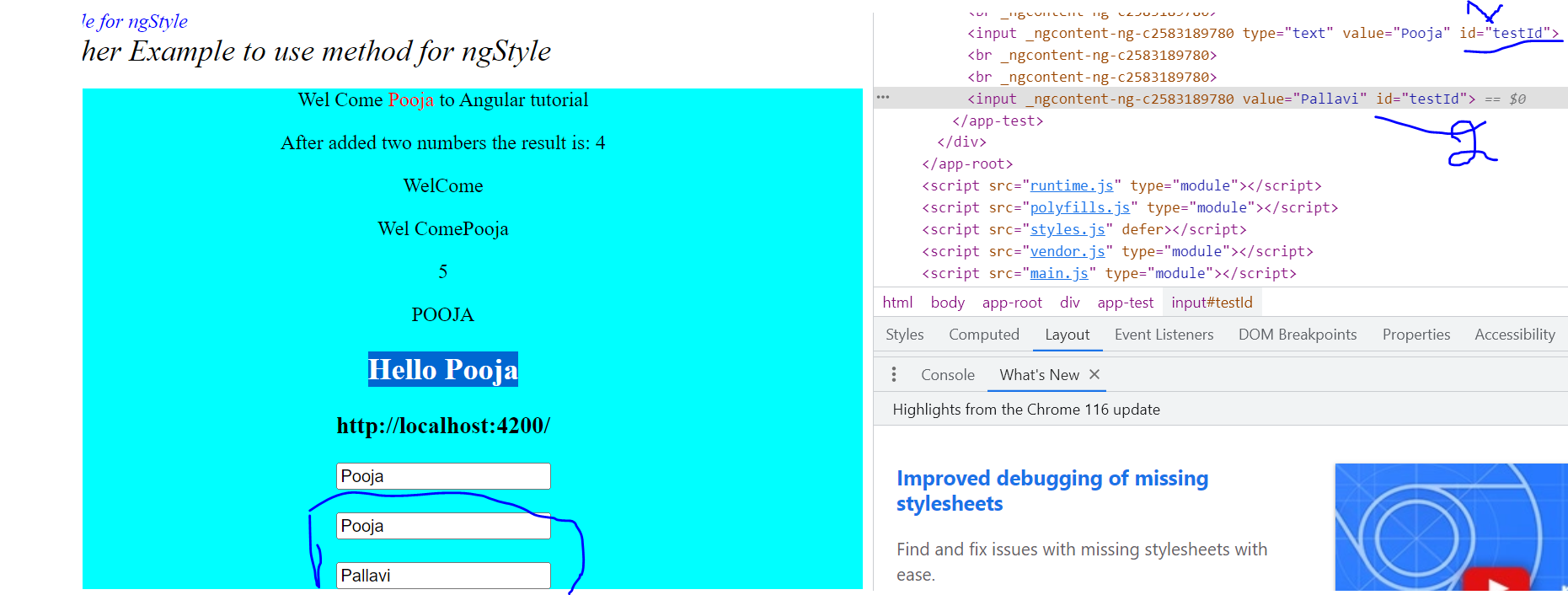
  styleUrls: ['./test.component.css']

})

export class TestComponent {

public myId= "testId";

}



**There are some html property need to bind Boolean values**

**If we add the disabled property to false still the input field is in disable mode.**

**If we use interpolation also its not work**

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

@Component({

  selector: 'app-test',

  template: `

   <input type="text" value="Pooja"><br><br>

   <input [id]="myId" type="text" value="Pooja"><br><br>

   <input disabled="false" id={{myId}} value="Pallavi">

   <input disabled="{{false}}" id={{myId}} value="Pallavi">`,

  styleUrls: ['./test.component.css']

})

export class TestComponent {

public myId= "testId";

}



**For this situation we use property binding**

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

@Component({

  selector: 'app-test',

  template: `

   <input type="text" value="Pooja"><br><br>

   <input [id]="myId" type="text" value="Pooja"><br><br>

   <input [disabled]="false" id={{myId}} value="Pallavi">`,

  styleUrls: ['./test.component.css']

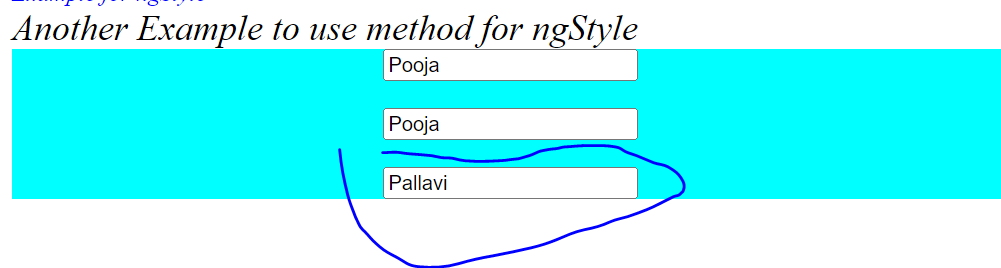
})

export class TestComponent {

public myId= "testId";

}

**Output**



**Another syntax for use of property binding is**

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

@Component({

  selector: 'app-test',

  template: `

   <input type="text" value="Pooja"><br><br>

   <input [id]="myId" type="text" value="Pooja"><br><br>

   <input bind-disabled="false" id={{myId}} value="Pallavi">`,

  styleUrls: ['./test.component.css']

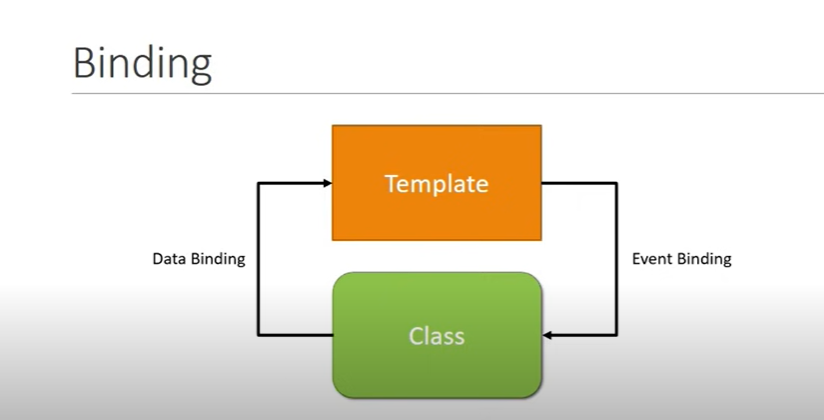
})

export class TestComponent {

public myId= "testId";

}

**Event Binding**



**Some times we need to transfer the data from template to class**

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

@Component({

  selector: 'app-test',

  template: `<button (click)="onClick()">Greeting</button>

   <h2>{{greeting}}</h2>`,

  styleUrls: ['./test.component.css']

})

export class TestComponent {

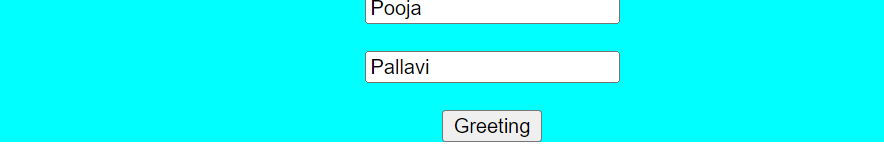
onClick(){

this.greeting = "Wel Come to angular tutorial";

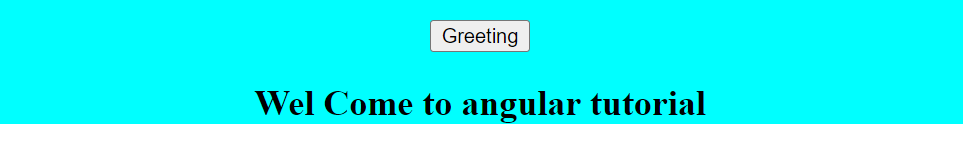
}

}

**Before button click output**



**After button click output**



**If we need the information about the event then we pass the event as parameter**

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

@Component({

  selector: 'app-test',

  template: `

   <button (click)="onClick($event)">Greeting</button>

   <h2>{{greeting}}</h2>`,

  styleUrls: ['./test.component.css']

})

export class TestComponent {

public greeting="";

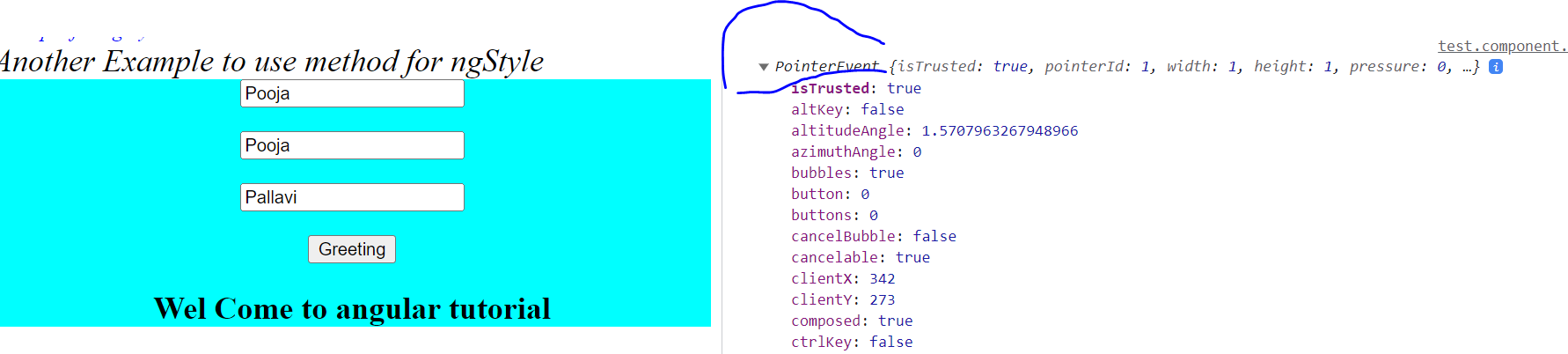
onClick(event:any){

this.greeting = "Wel Come to angular tutorial";

console.log(event);

}

}



**We can also add template statement to event**

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

@Component({

  selector: 'app-test',

  template: `

   <button (click)="greeting='Wel Come to Pooja'" >Greet</button>

   <h2>{{greeting}}</h2>`,

  styleUrls: ['./test.component.css']

})

export class TestComponent {

public greeting="";

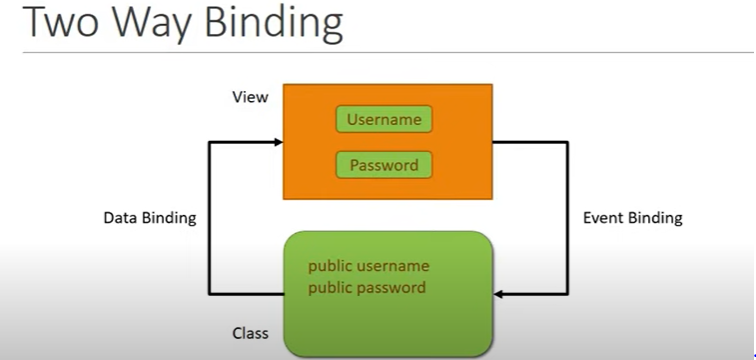
}



**Two way Data Binding**

**When we work with form input we need transfer data from template to class and class to template.**

**In this situation model and view always synch with.**



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

@Component({

  selector: 'app-test',

  template: `<input [(ngModel)]="name" type="text">

   {{name}}`,

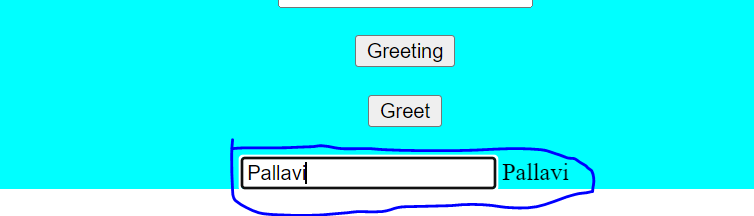
  styleUrls: ['./test.component.css']

})

export class TestComponent {

public name=””;

}



**Template Reference variable**

**To access the DOM elements and there property we use template reference variable**

**Reference element to refer the html element and all of its DOM property.**

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

@Component({

  selector: 'app-test',

  template: `

 <input #myInput type="text">

 <button (click)=onClick(myInput.value)>Log</button>

   `,

  styleUrls: ['./test.component.css']

})

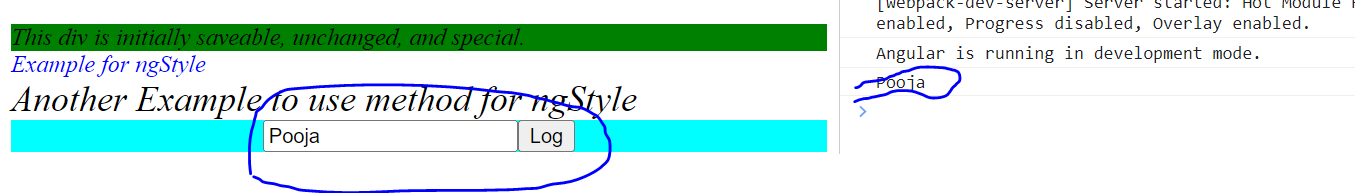
export class TestComponent {

onClick(myInput:any){

console.log(myInput);

}

}



**If we directly pass reference as a parameter we get the input element**

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

@Component({

  selector: 'app-test',

  template: `

 <input #myInput type="text">

 <button (click)=onClick(myInput.value,myInput)>Log</button>

   `,

  styleUrls: ['./test.component.css']

})

export class TestComponent {

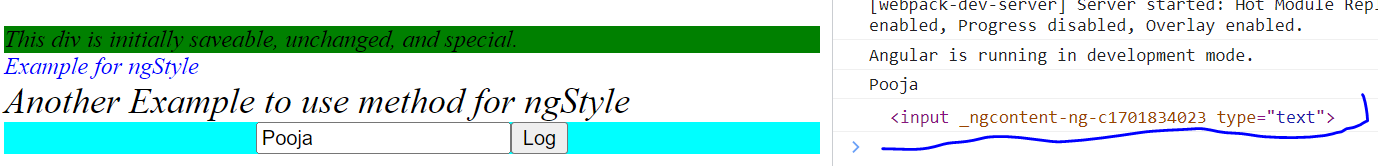
onClick(myInput:any,myInputReference:any){

console.log(myInput);

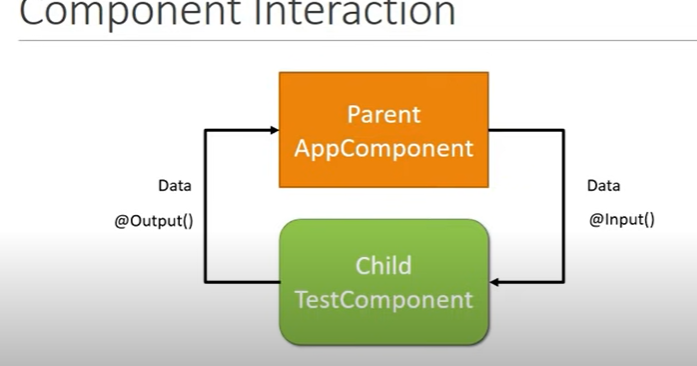
console.log(myInputReference);

}

}



**Component Interaction**



**Transfer data from one component to another if the component have parent child relation**

**Parent html page**

<h1>Parent Component</h1>

<input #inputValue type="text" value="Enter parent Data">

<button style="margin: 10px;" (click)="onClick(inputValue.value)">Click me to send Data</button>

<div style="background-color: antiquewhite;">

  <app-child [parentData]="name"></app-child>

</div>

**Parent ts file**

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  name = 'ComponentInteraction';

  onClick(value:any){

    this.name=value;

  }

}

**Child html page**

<p>child works!</p>

<h2>Revied data from parent {{parentData}}</h2>

**Child ts file**

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

@Component({

  selector: 'app-child',

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

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

})

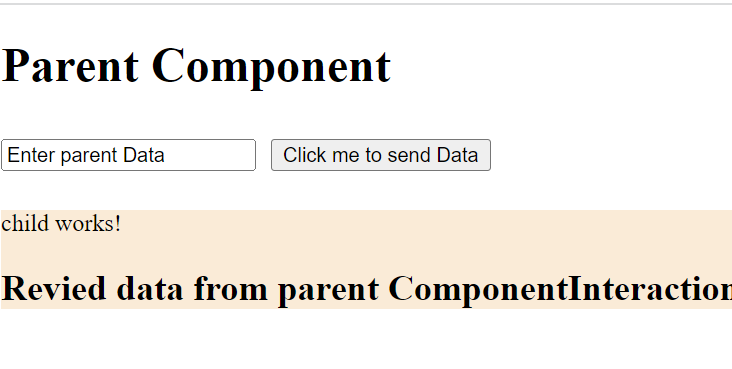
export class ChildComponent {

@Input() parentData:any;

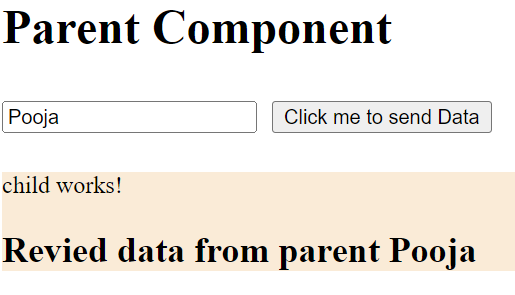
}

**Out put**

**Before click the button**



**After button click**



**If we want to display the different name that the parent component uses for that we alias name**

**Parent component html**

<h1>Parent Component</h1>

<input #inputValue type="text" value="Enter parent Data">

<button style="margin: 10px;" (click)="onClick(inputValue.value)">Click me to send Data</button>

<div style="background-color: antiquewhite;">

  <app-child [parentData]="name"></app-child>

</div>

**Parent ts file**

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  name = 'ComponentInteraction';

  onClick(value:any){

    this.name=value;

  }

}

**Child html file**

<p>child works!</p>

<h2>Revied data from parent {{name}}</h2>

**Child ts file**

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

@Component({

  selector: 'app-child',

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

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

})

export class ChildComponent {

@Input('parentData') name:any;

}

**Send data from child to parent**

**Parent html**

<h1>Parent Component</h1>

<h2>{{message}}</h2>

<div style="background-color: antiquewhite;">

  <app-child (childEvent)="message=$event">app-child>

</div>

**Parent ts file**

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  message="";

}

**Child html**

<p>child works!</p>

<button (click)="tiggerEvent()">click send data to parent</button>

**Child ts**

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

@Component({

  selector: 'app-child',

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

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

})

export class ChildComponent {

@Output() childEvent = new EventEmitter();

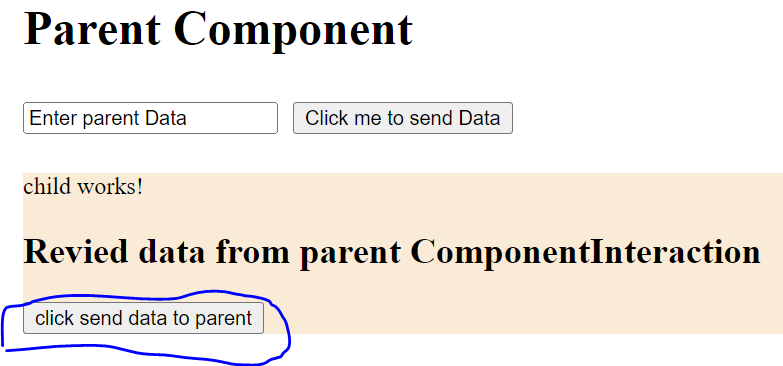
tiggerEvent(){

  this.childEvent.emit("Hello Pooja!");

}

}

**Out put**



**Pipes**

**Pipes allows us to transform data before displaying them in the view**

**It transform the data only for the view not change the value of the property in the class.**

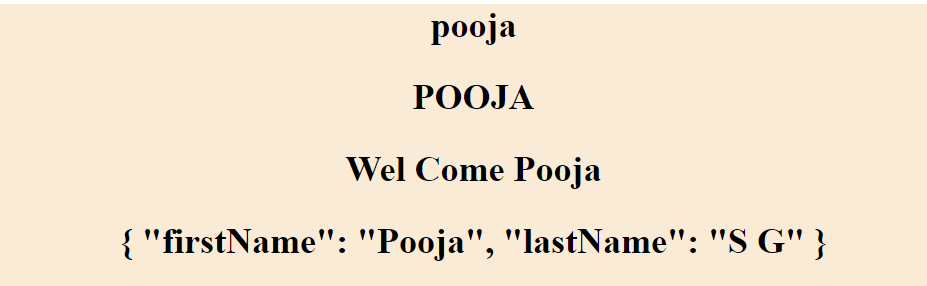
<h2>{{name| lowercase}}</h2>

<h2>{{name|uppercase}}</h2>

<h2>{{message|titlecase}}</h2>

<h2>{{details|json}}</h2>

**Output**



<!-- //number pipe -->

    <h2>{{5.678|number:'1.2-3'}}</h2>

      <!-- first number represets the number of interger

      second number represets the minimum decimal value

      third number represents the maximum decimal value -->

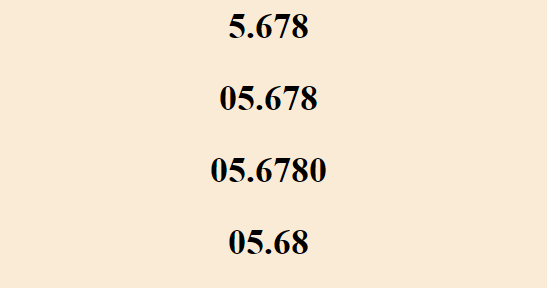
    <h2>{{5.678|number:'2.2-3'}}</h2>

    <!-- if not have minimum values its add the zero to its -->

    <h2>{{5.678|number:'2.4-5'}}</h2>

    <!-- if the max number represents less then it round up the number-->

    <h2>{{5.678|number:'2.1-2'}}</h2>



 <!--slice pipe-->

    <h2>{{data|slice:3}}</h2>

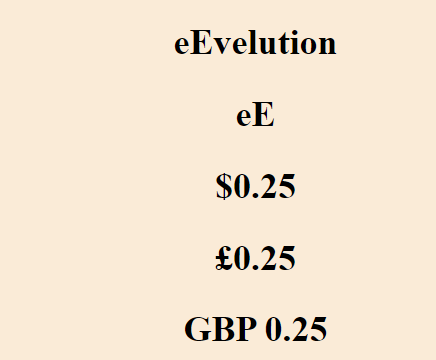
    <h2>{{data|slice:3:5}}</h2>

    <!--currency pipe-->

    <h2>{{0.25|currency}}</h2>

    <h2>{{0.25|currency:'GBP'}}</h2>

    <h2>{{0.25|currency:'GBP ':'code'}}</h2>



<!--Date pipe-->

    <h2>{{date}}</h2>

    <h2>{{date|date}}</h2>

    <h2>{{date|date:'short'}}</h2>

    <h2>{{date|date:'shortTime'}}</h2>

    <h2>{{date|date:'shortDate'}}</h2>

    <h2>{{date|date:'medium'}}</h2>

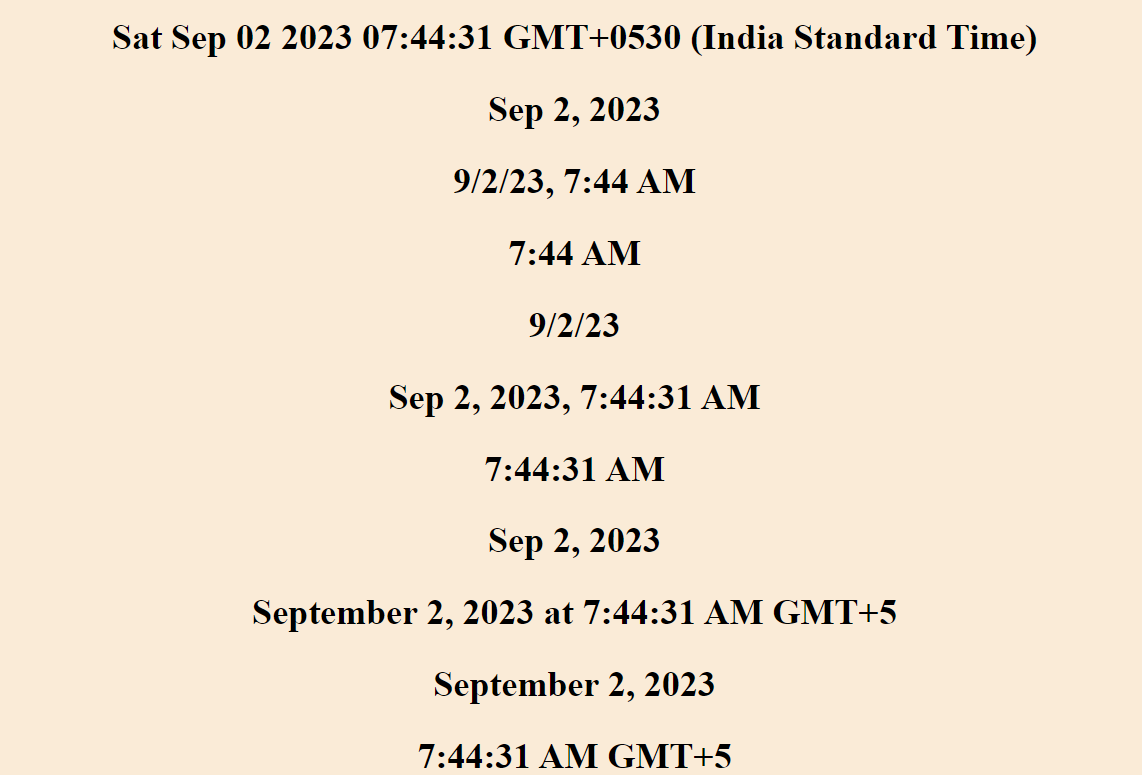
    <h2>{{date|date:'mediumTime'}}</h2>

    <h2>{{date|date:'mediumDate'}}</h2>

    <h2>{{date|date:'long'}}</h2>

    <h2>{{date|date:'longDate'}}</h2>

    <h2>{{date|date:'longTime'}}</h2>



**Services**

**Services are class with specific purpose**

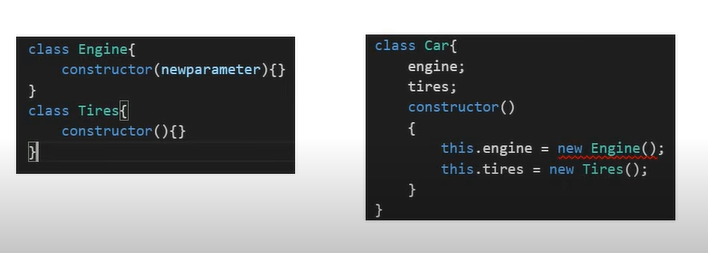
1. **Share data, It is used to transfer the data from one component to another component.**
2. **Also use to implement the logic. For example: employee enter the date of birth we need to calculate there age ,we write the logic to do that but this doesn’t need view logic and reusable code which can be independent of any individual component.**
3. **External interaction. For example to connect database**
4. **Naming convention - .service.ts**

**Dependency injection is not used in the following situation and its draw back are**

1. **Code is not flexible any time the dependencies change the car class needs to be change as well**
2. **This code is not suitable for testing**

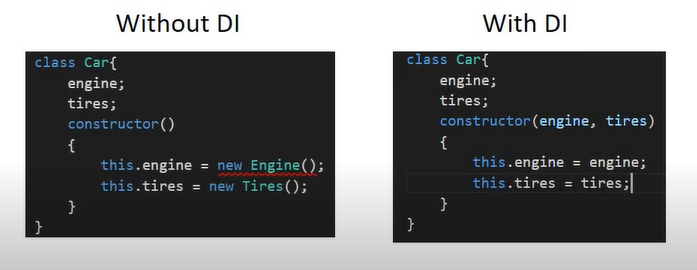
**Any time you instantiate the new car the same type of engine and same type of tiers to be tested.**

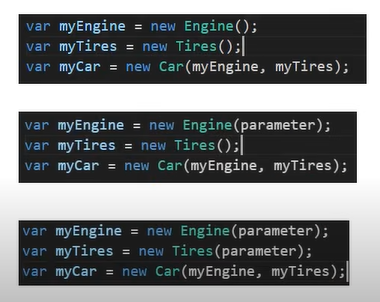
**If we want to test with petrol or diesel engine its not work.**



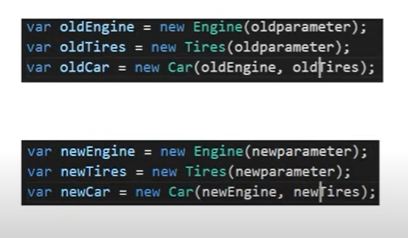
**Dependency injection is a coding pattern in which class receives its dependencies from external sources rather the creating them itself.**

**Below code is now flexible even is we change in dependencies car class not affect.**





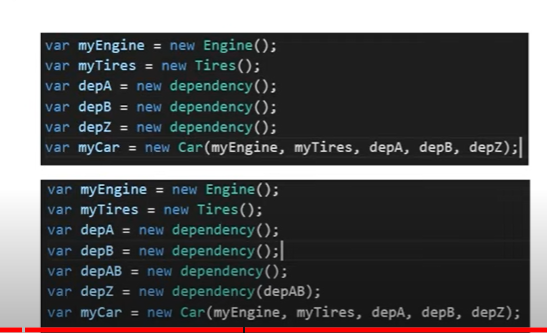
**For testing also the same**



**Here is another draw back**

**With DI we create a car passing dependencies as parameter but here we have to create parameter first then pass those parameter but**

**Here we use only two dependencies so here easy to create only two parameter but if we have more than hundred of parameter its difficult to create parameter**



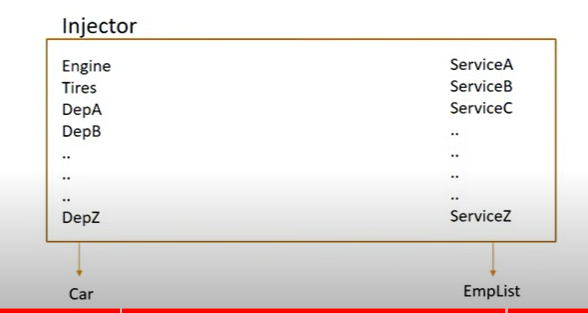
**To overcome this draw back angular dependency injection as frame work**

**DI frame work something we called as injector**

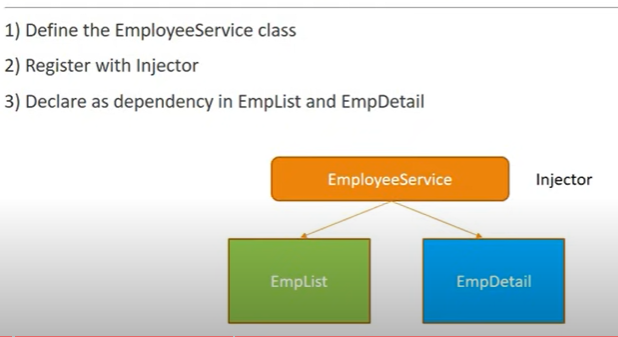
**Injector is basically container of all the dependencies like engine ,tiers so on.**

**If we want car as per car injector provide car for you**

**Framework will manages all the dependencies**



**Example:-**



**Example:**

**Create service using ng g s employee**

**In service ts file**

**Injectable decorator tells angular this service might its self have injected dependency.**

**Inject service into other service injectable decorator must required**

import { Injectable } from '@angular/core';

@Injectable({

  providedIn: 'root'

})

export class EmployeeServiceService {

  constructor() { }

  employeeDetails:any=[];

  getEmployee(){

   return this.employeeDetails=[

      {id:'1',name:'Pooja',age:'20'},

      {id:'2',name:'Pallavi',age:'25'},

      {id:'3',name:'Chandrika',age:'24'},

      {id:'4',name:'gowri',age:'22'},

      {id:'5',name:'vikas',age:'21'}

    ]

  }

}

**Component ts file**

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

import { EmployeeServiceService } from '../employee-service.service';

@Component({

  selector: 'app-employee-list',

  templateUrl: './employee-list.component.html',

  styleUrls: ['./employee-list.component.css']

})

export class EmployeeListComponent {

  employeeDetails:any=[];

  constructor(private employeeService:EmployeeServiceService){}

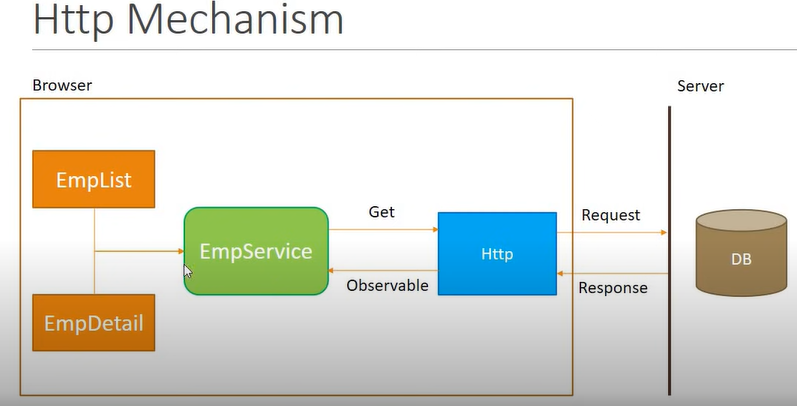
  ngOnInit(){

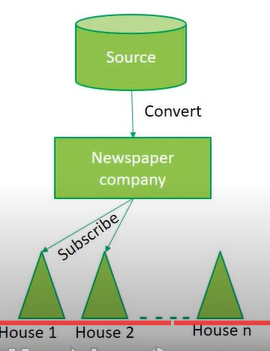
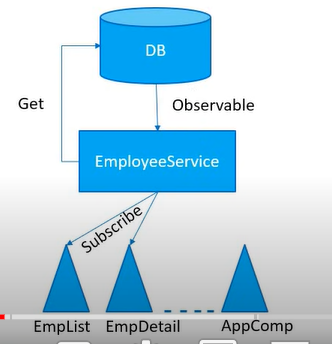
    this.employeeDetails=this.employeeService.getEmployee();

  }

}

**HTTP and Observable**



**Here the example for observable that is news paper company**

**The company provide the paper to house those who are subscribe to them.**

**Observable**

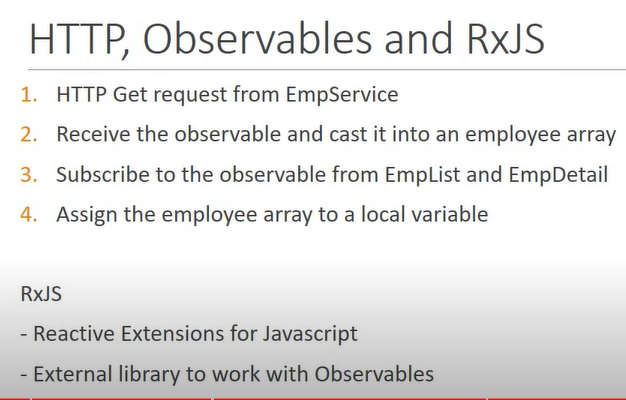
observables as an interface to handle a variety of common asynchronous operations. For example: The HTTP module uses observables to handle AJAX requests and responses. The Router and Forms modules use observables to listen for and respond to user-input events.

1. Observable streams the data, it provide the data in the form packets and its not wait for entire data available.

2. Observable is not native to JS but it is provided by the another JS library i.e,RxJS=>reactive extension library for JS

3. RXJS as two parts one is observable i.e.,which stream the data Another is observer which is going to use the data

4. Observable is only provide the data if some one use the data.



**Employee interface**

export interface IEmployee{

    id:number,

    name:string,

    age:number

}

**Employee service**

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

import { Injectable } from '@angular/core';

import { Observable } from 'rxjs';

import { IEmployee } from './employee';

@Injectable({

  providedIn: 'root'

})

export class EmployeesService {

  constructor(private http:HttpClient) { }

 private url="/assets/data/employee.json";

  getEmployees():Observable<IEmployee[]>{

    return this.http.get<IEmployee[]>(this.url);

  }

}

**Employee list ts file**

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

import { EmployeesService } from '../employees.service';

@Component({

  selector: 'app-employee-list',

  templateUrl: './employee-list.component.html',

  styleUrls: ['./employee-list.component.css']

})

export class EmployeeListComponent {

  employees:any=[];

  constructor(private employeeService:EmployeesService){}

  ngOnInit(){

    this.employeeService.getEmployees().subscribe(data=>this.employees=data);

  }

}

**In asset folder we have file as employee.json**

[

    {"id":1,"name":"pooja","age":25},

    {"id":2,"name":"pallavi","age":22},

    {"id":3,"name":"mamatha","age":20},

    {"id":4,"name":"radha","age":18},

    {"id":5,"name":"manjula","age":25},

    {"id":6,"name":"kavya","age":27},

    {"id":7,"name":"ramya","age":18}

  ]

**HTTP ERROR HANDLING**