ANGULAR 17 (nov 2023)

>> New Updates:

1. new Logo ,new website(angular.dev now , previously angular.io).

2. By default standalone(removed module , ngModule), Signals improved implementation with less boilerplate code.

3. Initial> AngularJs: failed to support Typescript , ssr need heavy boilerplate, performance lg due to digest cycle dependency to track changes (as had to look all the ocmponents to track changed one), had two way data binding.

Now> Angular : loaded with features like Typescript support, oops support, improved performance , less boiler plate code, features like components, directives, pipes , Http services, dependency injection etc introduced. Improved two way data binding breaking into [one way binding, two way binding, event binding] which ultimately improved performance by providing more explicit control and unncessary checks avoiding.

Major improvement of performance by: (integration of Zone.js) and (On push strategy introduction)

>>>>>>>>>>>>>>>> **Question:** Why angular versions have 3 parts: eg: 17.1.2 >>>>>>>>>>>>>>>>

Sol> Because these three parts denote(major version=>17 , minor version=>1, patch version=>2 . Release frequency 6month.

>>>>>>>>>>>>>>>> **Question:** What is requirement/role of Nodejs in angualr. >>>>>>>>>>>>>>>>

Sol> NodeJs(built on chrome's js engine V8, developed by Ryan Dahl in 2009) being server side runtime environment, open source cross platform for developing server-side and networking applications. Why need for angular:

> It activates/spins dev server on local system.

> Installs npm by default and its role in project creation installation of packages in angular , creates package.json

> To install angular cli we need npm. For bundling, TS compilation etc. Install/Uninstall packages.

4. To install:

>> ng new myprojectName

>> cd myprojectName

>> ng serve //to run project on port 4200 by default

Main Building blocks of Angular application (from v17 onwards)

>> Modules (optional now): To be used for features Module , lazy loading of modules we can still create and use . It is a class decorated by **@NgModule** , taking single metadata object and its properties describing module. To create module run: " **ng g m moduleName ".**

>> Components: A class with template taht deals with the view of application and contains the core logic for the page. It is decorated by **@Component** , stores metadata. To create component run: " **ng g c componentName ".**

>> **Templates:** view of a page.

>> **Metadata:** data about data, say extra information , data about anything.

>> **Data Binding:** Communication between model(component.ts) and view(.html).

>> **Directives:** Killer feature(a js class declared as @directive) of angular. Used to extend the power of HTML(say directly run loop , switch on html) attribute and to change the appearance or behaviour(say use conditional view like if right in html) of a DOM element. 3 types of directives are (**Component Directive, Structural Directive, Attributes Directive**).

>> **Services:** A function or an object taht can be used to share data and behaviour across application. Uses Injectors to make data avaikabale across application. To create service run: " **ng g s serviceName ".**

>> **Dependency Injection:** Injecting any dependency . Eg: Services are dependency as have functionality ,data needed by components so components acts as injector that injects these services.

Component

Components are the most basic user interface building block of an Angular application. An angular application contains a tree of angular components.
It is a subset of directives and always associated with a template. Unlike other directives, you can only create one instance of a component per element in a template.

A component is nothing more than a simple TypeScript class in which you can create your own methods and properties according to your needs, which is used to bind with a UI (HTML page or cshtml page) of our application.
The most important feature of any Angular application is the component that controls View or the template that we use. In general, we write all the application logic for the view that is assigned to this component.

Therefore, an Angular application is just a tree of such Components, when each Component is processed, it recursively processes its children Components. At the root of this tree is the top level component, the main component.

Now, angular-cli has a command to create your component. However, the app component that is created by default will always remain the parent component and the next components created will form the child components. Let's create your corner component.
create a components using the following command in the integrated terminal:

ng g c demo
• **ng** calls the angular CLI
• **g** is the abbreviation of generate (note, you can use the whole word generate if you wish)
• **c** is the abbreviation of component (note, you can use the whole word component if you wish).
component is the type of element that is going to be generated (others include directive, pipeline, service, class, security, interface, enumeration and module)
• and then the **name of the component**
When you run the above command on the command line, you will receive the following output

Now, if we check the structure of the file, we will get the new demo folder created in the src/app folder.
The following files are created in the demo folder:

• demo.component.css: the css file is created for the new component.
• demo.component.html: the html file was created.
• demo.component.spec.ts: can be used for unit tests.
• demo.component.ts: here we can define the module, properties, etc.

Decorators

- **Decorators** are a feature of TypeScript and are implemented as functions. The name of the decorator starts with @ symbol following by brackets and arguments, since decorators are just functions in TypeScript.

- Decorators are simply functions that return functions. These functions supply metadata to Angular about a particular class, property, value, or method…
- **Decorators are invoked at runtime.**

- Decorators allow you to execute functions. For example @Component executes the Component function imported from Angular 17.

Common Decorators

- @NgModule() to define modules…
- @Component() to define comments…
- @Injectable() to define services…
- @Input() and @Output() to define properties…that send and receive data from the dom.

There are many built-in decorators available in Angular…and many properties on each decorator

Type of Decorators

- **Class decorators**, e.g. @Component and @NgModule
- **Property decorators** for properties inside classes, e.g. @Input and @Output
- **Method decorators** for methods inside classes, e.g. @HostListener
- **Parameter decorators** for parameters inside class constructors, e.g. @Inject

Each decorator has a unique role

Selector

The **selector** attribute allows us to define how Angular is identified when the component is used in HTML. It tell Angular to create and insert an instance of this component where it finds the selector tag in the Parent HTML file in your angular app.

The selector accepts a string value, which is the CSS selector that Angular will use to identify the element. We will use it in html to place this component where we want. If you see the index.html file, you will find **<app-root></app-root>** component inside the body.

The component selector is a CSS selector that identifies how Angular finds this particular component in any HTML page. In general, we use element selectors **<app-root></app-root>**), but it can be any CSS selector, from a CSS class to an attribute as well.

The component is applied to the **<app-root></app-root>** tag in index.html. If index.html does not have that tag, Angular will fail at startup. You can check where the angular application will be played.

DIRECTIVES:

CommonModule provides several common directives and pipes, including *ngIf, *ngFor, ngSwitch, and others.

>> STRUCTURAL DIRECTIVES:

@for

The @for block repeatedly renders content of a block for each item in a collection.

Syntax

```
@for (item of items; track item.name) {
  <li> {{ item.name }} </li>
} @empty {
  <li> There are no items. </li>
}
```

Inside @for contents, several implicit variables are always available:

Variable	Meaning
$count	Number of items in a collection iterated over
$index	Index of the current row
$first	Whether the current row is the first row
$last	Whether the current row is the last row
$even	Whether the current row index is even
$odd	Whether the current row index is odd

>>>> Only show even index data, along with index number: i.e use of $even, $index:

```
<!-- {{employees}} -->
@for(emp of employees;track emp.id; let i=$index,e=$even){
<ul>
    @if(e){
        <li>#{{i}} -  {{emp.id}} - {{emp.name}}</li>
    }

</ul>
}@empty{
    No Data Found
}
```

Output:

#0 - 100 - Chandan Kumar

#2 - 102 - Soni Kumari

#4 - 104 - Rahul Kumar sharma

>>> Showing use of $first $last etc:

```
<!-- {{employees}} -->
@for(emp of employees;track emp.id; let i=$index,e=$even,o=$odd,f=$first,l=$last){
<ul>
    @if(f){
        <li style="background-color: blue;">#{{i}} -  {{emp.id}} - {{emp.name}}</li>
    }
    @if(!f && !l){
       <li> #{{i}} -  {{emp.id}} - {{emp.name}}</li>
    }
    @if(l){
        <li style="background-color: red;">#{{i}} -  {{emp.id}} - {{emp.name}}</li>
    }

</ul>
}@empty{
    No Data Found
```

>>>>

Attribute Directive

he attribute directive changes the appearance or behavior of a DOM element. These directives look like regular HTML attributes in templates. The *ngModel* directive which is used for two-way is an example of an attribute directive. Some of the other attribute directives are listed below:

•**NgStyle:** Based on the component state, dynamic styles can be set by using *NgStyle*. Many inline styles can be set simultaneously by binding to *NgStyle*.

•**NgClass:** It controls the appearance of elements by adding and removing CSS classes dynamically.

Example

ngStyle

The **NgStyle** directive allow you to set a given DOM elements style properties.

One way to set styles is by using the NgStyle directive and assigning it an *object* , like so:

<div [ngStyle]="{ 'background-color':'green' }">Hello Sahosoft

This sets the background color of the div to green.

>> Dynamic and static use of ngStyle:

```
<div style="background-color: aqua;">Hello Sahosoft</div>
<div [ngStyle]="{'background-color' : mycountry=='India' ? 'green':'red'}">Hello World !!</d
```

ngClass

•**NgClass:** It controls the appearance of elements by adding and removing CSS classes dynamically.

- ✓ ngClass with string
- ✓ ngClass with array
- ✓ ngClass with object

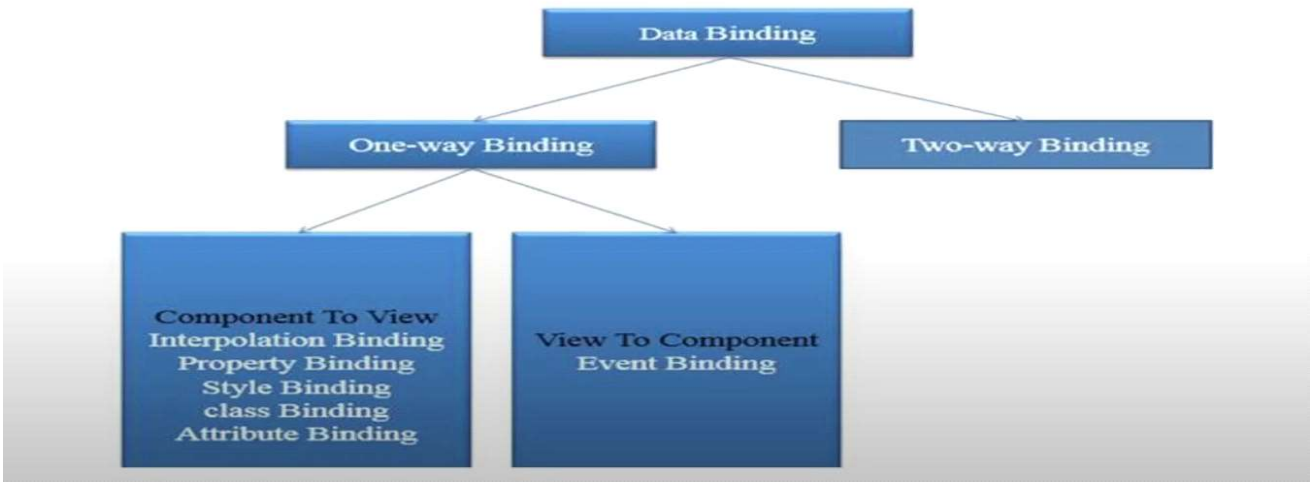
```
<!-- 1. ngClass with String -->

<div class="one">Hello Sahosoft</div>

<div [ngClass]="'one three'">Hello Sahosoft</div>

<!-- 1. ngClass with Array -->
<div [ngClass]="['two','four']">Hello World</div>

  <!-- 1. ngClass with object -->

  <div [ngClass]="{'one':false,'three':true}">Hello World</div>

  <div [ngClass]="{'two':mycountry=='USA'}">Hello</div>
```

DATA BINDING

Data Binding

Data Binding means to bind the View (Html content) with Controller's (Component's) field. That is whenever we display dynamic data on a view (HTML) from Component, data binding is used.

Data Binding basically means interacting with data. So, we can say that the interaction between templates and business logic is called data binding.



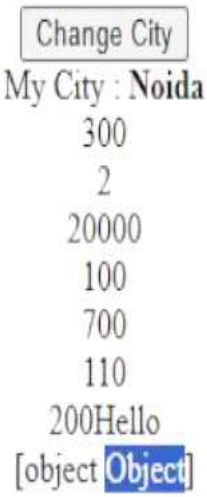
interpolation Binding

Interpolation is a technique that allows the user to bind a value to a UI element.
Interpolation binds the data one-way. This means that when value of the field bound using interpolation changes, it is updated in the page as well. It cannot change the *value* of the field

The format for defining interpolation in a template is: **{{ propertyName }}**

Component ----------{{Value}}----------→ Dom

```
<button (click)="changecity()">Change City</button> <br/>

My City : <b>{{city}}</b><br/>

{{num1+num2}} <br/>
{{num1/num2}} <br/>
{{num1*num2}} <br/>
{{num1-num2}} <br/>
{{num1+500}} <br/>
{{50+60}} <br/>
{{num1+"Hello"}} <br/>

{{company}}
```



//cannot bind object directly. Can bind with dot(.)

ERROR HANDLING IN INTERPOLATION:

```
<!-- Name : {{company && company.name}}<br/>
City : {{company && company.city}}<br/>
State : {{company && company.state}}<br/>
Country : {{company && company.country}} -->

Name : {{company?company.name: 'No data'}}<br/>
City : {{company?company.city : 'No data'}}<br/>
State : {{company?company.state:'No data'}}<br/>
Country : {{company?company.country:'No data'}}
```

Rarely_Use: ATTRIBUTE BINDING : one way binding (component to view binding)

i.e can say value of attribute directly....eg: colspan is attribute not property. as colspan,rowspan element DOM me nhi

aate...

JIS Element ka Property na Bane i.e generate na ho unka attribute bind krna chahiye as atttribute binding gives old value .

Done by : [attr.colspan]="colFromTsVariableName"

>>> if not used attr. then it becomes property binding. and in this case throw error.

Purpose: Used to set attributes of HTML elements (not necessarily related to CSS).

Syntax: [attr.attributeName].

Use Case: You can bind HTML attributes such as src, href, aria-*, etc., which can affect how elements behave or are rendered.

Attribute Binding

With attribute binding, we can set the value of an attribute directly. The thing to note is that you must use the attribute binding only when there is no element property there to bind.
Attribute binding is used to bind attribute of an element with the field of a component.

```
<td [attr.colspan]="myColSpan" align="center">  Employee's Records   </td>
```

Eg: say while creating table of rows with two col. each but first row has only 1 col. so here we will need to use attribute binding for colspan use:

Binding here colspan means value of colspan should be coing form .ts not made static like(colspan="2")...so we need to bind.

Always_Use: PROPERTY BINDING : one way binding (component to view binding)

Property Binding

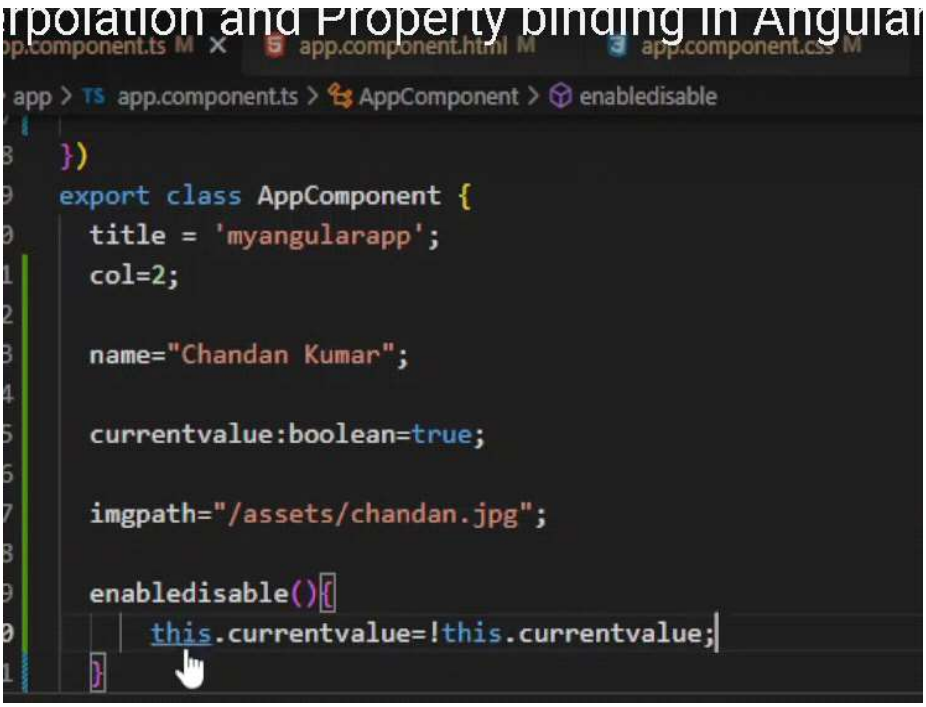
- ✓ It is used to bind values of component/model properties to the HTML element.
- ✓ Depending on the values, it will change the existing behavior of the HTML element.
- ✓ **Syntax** [property] ='expression'
- ✓ In property binding, there is source and target. For this example, we can define it as [innerHTML] = ' firstname'. Here, innerHTML is a target that is a property of span tag and source is a component property i.e. firstname

eg:

```
<h2>Property Binding</h2>
<span [innerHTML]="name"></span>

<h2>Interpolation</h2>
<span innerHTML="{{name}}"></span>

<h2>Property Binding</h2>

<img [src]="imgpath" width="400px" alt="Chandan kumar image">

<h2>Interpolation</h2>
<img src="{{imgpath}}" width="400px" alt="Chandan kumar image">
```

BOth working same but has slight difference . Always do property binding ehile dealing with property not interpolation as **interpolation sometimes fails to bind some property. Eg: When working with BOOLEAN**

```
<button (click)="enabledisable()">Enable Disable</button>

<h2>Property Binding</h2>
<button [disabled]="currentvalue">Click me</button>

<h2>Interpolation</h2>
<button disabled="{{currentvalue}}">Click me</button>
```



Here even on clicking Enable Disable button only , button with property bindig acts correctly , btn with interpolation fails to get the updated value .

STYLE BINDING: one way binding (component to view binding)

Adding/Removing style on element at run time dynamically.

Purpose: Used to dynamically set CSS styles of an element.

Syntax: [style.property] or [ngStyle].

Use Case: You can bind CSS properties directly to Angular component properties or expressions.

- **Example:**

html Copy code

```
<div [style.color]="isRed ? 'red' : 'blue'">This text changes color</div>
```

Or using `ngStyle`:

html Copy code

```
<div [ngStyle]="{'color': isRed ? 'red' : 'blue', 'font-size': fontSize + 'px'}">St
```

```
<button style="background-color: red;">Click me</button><br/>
<button style="background-color: red;" [style.font-weight]="'bold'">Click me</button><br/>
<button style="background-color: red;" [style.font-weight]="isbold">Click me</button><br/>
<button style="background-color: red;" [style.font-size.px]="fontsize">Click me</button><br/>
```

CLASS BINDING : one way binding (component to view binding)

Purpose: To conditionally add or remove CSS classes based on a component's state. Can give multiple css classes in string form, array form , object form. Or apply class conditionally.

Syntax: The basic syntax for class binding is [ngClass], which can take an object, an array, or a string as its value.

```
<h3>Object Syntax</h3>
<div [ngClass]="{'active': isActive, 'disabled': isDisabled}">
  This div uses object syntax for class binding.
</div>

<h3>Array Syntax</h3>
<div [ngClass]="[isActive ? 'active' : '', isDisabled ? 'disabled' : '']">
  This div uses array syntax for class binding.
</div>

<h3>String Syntax</h3>
<div [ngClass]="activeClass">
  This div uses string syntax for class binding.
</div>
```

IMP. DIFFERENCES

Feature	Style Binding	Attribute Binding	Property Binding
Purpose	Dynamically applies CSS styles.	Dynamically sets HTML attributes.	Dynamically sets DOM properties.
Syntax	`[style.property]` or `[ngStyle]`	`[attr.attributeName]`	`[propertyName]`
Use Case	To modify visual styles like color, size, etc.	To set HTML attributes like `src`, `href`, `disabled`, etc.	To set properties of DOM elements like `checked`, `value`, etc.
Common Properties	`color`, `background-color`, `font-size`, etc.	`src`, `href`, `alt`, `title`, etc.	`value`, `checked`, `disabled`, `innerHTML`, etc.
Data Type	Typically string values (CSS values).	Can be various types (strings, numbers).	Can be various types (strings, booleans, numbers).
Change Detection	Affects styles at render time based on conditions.	Affects attributes at render time based on conditions.	Affects properties at render time based on conditions.
Binding Context	Directly modifies the styles of an element.	Modifies the underlying HTML attributes of an element.	Modifies the properties of the DOM element in Angular.
Reactiveness	Reflects changes in component properties immediately in the view.	Reflects changes in component properties immediately in the view.	Reflects changes in component properties immediately in the view.

>>>>>>>>>>>>>>>>>>

Feature	Style Binding	ngStyle	ngClass
Purpose	Dynamically binds individual CSS properties.	Dynamically sets multiple inline CSS styles using an object or array.	Dynamically adds or removes CSS classes based on conditions.
Syntax	`[style.property]`	`[ngStyle]`	`[ngClass]`
Use Case	To bind a specific CSS property to a component property or expression.	To apply multiple CSS styles conditionally from an object or array.	To conditionally apply or remove one or more CSS classes based on component properties or expressions.
Common Properties	`color`, `background-color`, `font-size`, etc.	Any CSS property, e.g., `{'color': 'red', 'font-size': '16px'}`.	Class names, e.g., `{'active': isActive, 'disabled': isDisabled}`.
Data Type	String value (e.g., color name, pixel value).	Object (key-value pairs for CSS styles) or array of strings.	String, object, or array (for class names).
Reactiveness	Immediately reflects changes to the specific CSS property.	Reflects changes to any of the properties defined in the object or array.	Reflects changes to class names based on the component's boolean expressions.
Example	`<div [style.color]="isRed ? 'red' : 'blue'">Colored Text</div>`	`<div [ngStyle]="{'color': isRed ? 'red' : 'blue', 'font-size': fontSize + 'px'}">Styled Text</div>`	`<div [ngClass]="{'active': isActive, 'disabled': isDisabled}">Class Based Text</div>`

EVENT BINDING: Two way binding (component to view & view to Component binding)

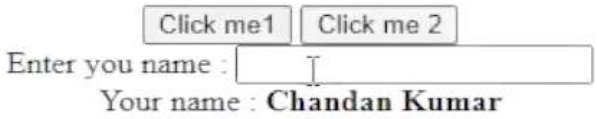
Purpose: Any occurance is event**.** To respond to user actions like clicks, key presses, mouse movements, etc.

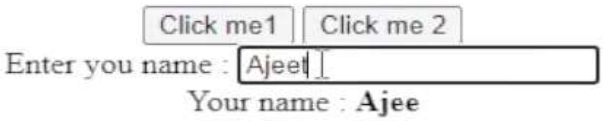
Syntax: The syntax for event binding is (eventName)="expression", where eventName is the name of the event to listen for, and expression is the action to take (usually a method in the component).

```
export class AppComponent {
  title = 'myangularapp';

  name="Chandan Kumar";

  onclick(val:any){
    this.name=val;
  }

  onchange(){
    console.log("Click me 1 fired");
  }
```

```
<button (click)="onchange()">Click me1</button>
<button on-click="onchange2()">Click me 2</button>

<br/>

<!-- <input type="text" (keyup)="keyup()" (keydown)="keydown()"> -->

<!-- <input type="text" on-keyup="keyup()" on-keydown="keydown()"> -->

Enter you name : <input type="text" #txtname (keyup)="onclick(txtname.value)"><br/>
Your name : <b>{{name}}</b>
```

Here event can be binded in cannonical form: eg: (click)="onchange2()" can be written in **cannonical form: on-click="onchange2()"**



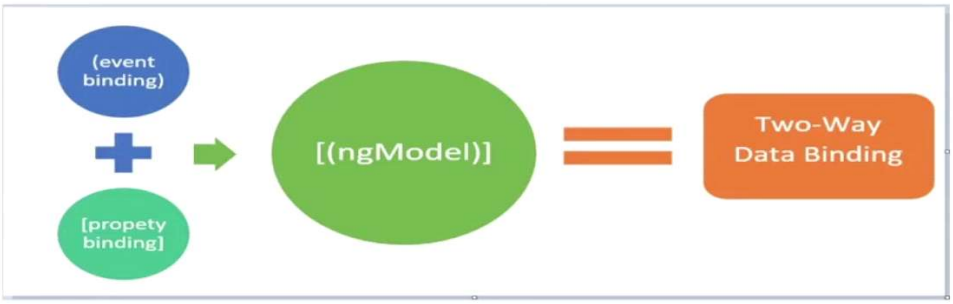


on key press: change of data binded:

TWO WAY DATA BINDING

(Mixture of Property binding and event binding) written in ngModel and avilable in Forms library.

The most popular and widely used data binding mechanism is two-way binding in the angular framework. Basically two-way binding mainly used in the input type field or any form element where user type or provide any value or change any control value in the one side, and on the other side, the same automatically updated in to the controller variables and vice versa.



In simple words, two-way data binding is a combination of both Property Binding and Event Binding.

```
<input [value]='data1' (input)='data = $event.target.value'>
```

Binding using [(ngModel)] directive

ngModel directive which combines the square brackets of property binding with the parentheses of event binding in a single notation.

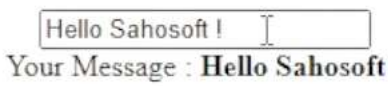
```
<input [(ngModel)] ='data'>
```

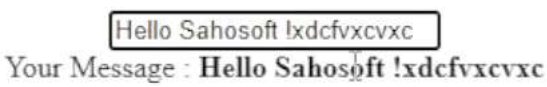
To use inshorter form now we do it using : ngModel , to use ngModel we need **to import FormsModule** in import metadata in component decorator.

Eg:

```
_message="Hello Sahosoft !";

onchange(val:any){
  this._message=val;
}
```

```
<input type="text" [value]="_message" #txtdata (keyup)="onchange(txtdata.value)"><br/>

Your Message : <b>{{_message}}</b>
```





>>>>>>>>> Other modern updated way todo so is using ngModel:

AS: **[] property binnding + () event biding = [(ngModel)] ngModel 2way binding.**

Now need not to write change event or anything all pre-written:

```
<input type="text" [(ngModel)]="_message"><br/>

Your Message : <b>{{_message}}</b>
```

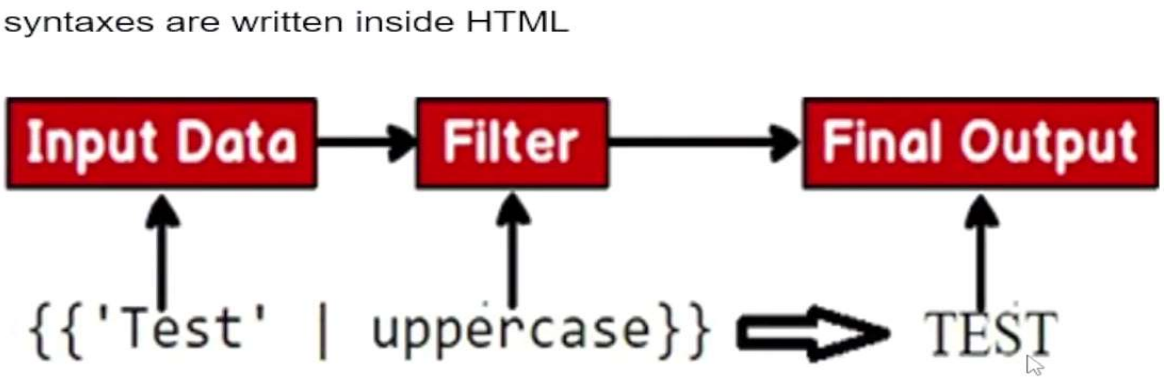
PIPES

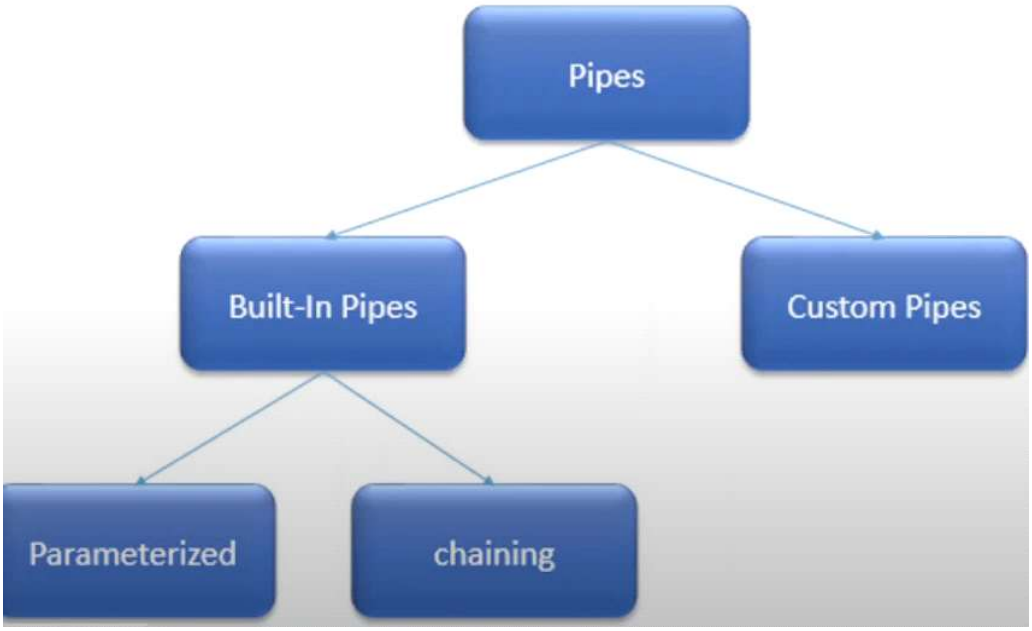
Pipes in Angular

Angular Pipes can be used to transform data to desired output.
Pipes takes in a data input and transforms data to a different output.
using the Pipe operator (|), we can apply the Pipe's features to any of the property in our Angular project.

Pipes (|) in Angular are used to transform the data before displaying it in a browser. Angular provides a lot of built-in pipes to translate the data before displaying it into the browser and as we know, Angular lets us extend its feature, we can even create custom pipes in Angular.

In angularJs PIPES were known as 'Filters'





Angular provides some built-in pipes:

1. Lowercase
2. Uppercase
3. Titlecase
4. Slice
5. Json Pipe
6. Decimalpipe
7. Percent Pipe
8. Currency Pipe
9. Date Pipe
10. **async Pipe**

Examples: using below data:

```
title = 'Hello Sahosoft Technologies..';

company={
  name:'Sahosoft technologies',
  city:'Noida',
  state:'UP',
  country:'India'
}

students:[
  {id:101,name:'Chandan Kumar',gender:'Male'},
  {id:102,name:'Ajeet Kumar SIngh',gender:'Male'},
  {id:103,name:'Soni Kumari',gender:'Female'},
  {id:104,name:'Reeta Kumari pandey',gender:'Female'},
  {id:105,name:'Mohan Kumar SHarma',gender:'Male'},
  {id:106,name:'Sohan Kumar',gender:'Male'}
]
```

1. **Built-in Non-Parameterized pipes**:

```
<h3>{{title}}</h3>
<h3>{{title | lowercase}}</h3>
<h3>{{title | uppercase}}</h3>
<h3>{{title | titlecase}}</h3>
```

2. **Parameterized Pipes**(that takes some parameter): **{{title | slice:fromIndex:ToIndex}}**

3. JSON Pipe: **<h3>{{company | json}} </h3>**

{ "name": "Sahosoft technologies", "city": "Noida", "state": "UP", "country": "India" }

4. **Binding Array form data To table**: using for loop with new directive format @for:

```
<tr>
    <th>Stdent ID</th>
    <th>Student Name</th>
    <th>Student Gender</th>
</tr>

@for(student of students;track student.id){
    <tr>
        <td>{{student.id}}</td>
        <td>{{student.name | uppercase}}</td>
        <td>{{student.gender}}</td>

    </tr>
}
```

Stdent ID	Student Name	Student Gender
101	CHANDAN KUMAR	Male
102	AJEET KUMAR SINGH	Male
103	SONI KUMARI	Female
104	REETA KUMARI PANDEY	Female
105	MOHAN KUMAR SHARMA	Male

5.

Decimal Pipes in Angular

DecimalPipe 12.20
This pipe is used for transformation of decimal numbers.
The first argument is a format string of the form

"{minIntegerDigits}. {minFractionDigits}-{maxFractionDigits}",
minIntegerDigits : Minimum number of integer digits. Default is 1.
minFractionDigits : Minimum number of fraction digits. Default is 0.
maxFractionDigits : Maximum number of fraction digits. Default is 3.
Now find some sample examples.
1. Using default format:

minIntegerDigits = 1
minFractionDigits = 0
maxFractionDigits = 3

Eg: <h3> {{12.89765446 | number }} </h3> ==> output: 12.898

<h3> {{12.89765446 | number : '2.3-5' }} </h3> ==> // min fraction digit set to 3 and maximum fracction digit set to 5 so, output: 12.89765

<h3> {{12.89765446 | number : '3.2-3' }} </h3> ==> // minIntegerDigit set to 3 , minFraction digit set to 2 and maxFraction digit set to 3 so, output: 012.898

6. Currency Pipe : changes format of currency(say symbol) not the value of currency. If symbol of currency not given bydefault gives USD.

Currency Pipes in Angular

Formats a number as currency.
CurrencyPipe is an angular Pipe API that formats a number as currency using locale rules. It belongs to **CommonModule**. CurrencyPipe uses currency keyword with pipe operator to format a number into currency format. Find the syntax.

number_expression | currency[:currencyCode[:symbolDisplay[:digitInfo]]]
Find the description.
number_expression : An angular expression that will give output a number.
currency : A pipe keyword that is used with pipe operator. It formats a number into currency format.
currencyCode : This is the currency code such as **INR** for Indian rupee, **USD** for US dollar. Default is **USD**.
symbolDisplay : Default is **false**. But if we assign **true** then it will display currency symbol such as **$** for dollar.
digitInfo: It defines a currency format. We have described the use of **digitInfo** in DecimalPipe section. It is used with following syntax.
{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

Important >>> If wants to format the currenccy into decimal format . Need to pass decimalformat as argument at 3rd place along with 'true/false' as 2nd argument. Here true means : apply the mentioned symbol, false means do not add symbol instead write symbol name string as it is passed as 1st argument of currency pipe.

```
<h3>{{num | currency}}</h3>

<h3>{{num | currency : 'USD'}}</h3>
<h3>{{num | currency : 'INR'}}</h3>
<h3>{{num | currency :'GBP'}}</h3>

<h3>{{num | currency : 'INR':true:'2.3-5'}}</h3>
```

$12.20

$12.20

₹12.20

£12.20

₹12.200

>> if gave false: {{num | currency : 'INR':false:'2.3-5'}} ==> output: INR12.200

7. Date Pipe: Format date in any format , as by defaault date comes in Indian Standard Time.

Eg:- today = new Date();

<h3> {{ today }} </h3> ==> Outputs: Tue Oct 15 2024 14:30:15 GMT+0530 (India Standard Time)

<h3> {{ today | date }} </h3> ==> Outputs: Oct 15, 2024, 2:30:15 PM

<h3> {{ today | date : 'longDate' }} </h3> ==> Outputs: October 15, 2024

Year: {{ today | date: 'yyyy' }} ==> Outputs: 2024

Full Date: {{ today | date: 'fullDate' }} ==> Outputs: Tuesday, September 3, 2024

Short Date: {{ today | date: 'shortDate' }}</p ==> Outputs: 9/3/24

Medium Date: {{ today | date: 'mediumDate' }} ==> Outputs: Sep 3, 2024

Custom Format: {{ today | date: 'dd/MM/yyyy' }} ==> Outputs: 03/09/2024

Custom Format: {{ today | date: 'dd-MM-yyyy' }} ==> Outputs: 03-09-2024

Time: {{ today | date: 'h:mm:ss a' }} ==> Outputs: 12:05:08 AM

Custom Time: {{ today | date: 'HH:mm:ss Z' }} ==> Outputs: 00:05:08 +0530

8. Chaining Pipes: Chaining two or pipes together.

Eg: <h3>{{ today | date : 'fullDate' | uppercase }}</h3> ==> Outputs: TUESDAY, SEPTEMBER 3, 2024

9. Custom Pipes:

Custom Pipes in Angular

To create a pipe in Angular 17, you have to apply the @Pipe decorator to class, which we can import from the core Angular library.
The @Pipe decorator allows you to define the pipe name that you'll use within template expressions.

Syntax

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({ name: 'Pipename' })
export class Pipeclass implements PipeTransform {
 transform(parameters): returntype { } }
```

Note -Transform() method will decide the input types, the number of arguments, and its types and output type of our custom pipe.

>>>>>>>>>To generate custom pipe run cmd to generate file => welcome.pipe.ts: **ng g p welcome**

app > TS welcome.pipe.ts > WelcomePipe

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

@Pipe({
  name: 'abc',
  standalone: true
})
export class WelcomePipe implements PipeTransform {

  transform(value: any, ...args: any[]): any {

    // debugger;
    return 'Hello : '+value;
  }

}
```

now import this pipe in component decorator , in imports metadata to use this pipe as: {{name | welcome}} ==> Output:

SERVICES

Service

Service is a mechanism used to share the functionality b/w the components.

- ✓ An angular service is simply a function that allows us to access it's defined properties and methods.
- ✓ it also helps keep our coding organized.
- ✓ You will most likely run into a scenario in which you need to use the same code across multiple components.
- ✓ you will create a single reusable data service and inject it into the component that need it.
- ✓ It easy to unit test component with a mock service.

Create a Service

We can create a custom service as per our requirement. For creating a service, we need to follow the below steps.

Step 1: Create the Service File

Creating a Service with the Angular-CLI

This is fairly easy, as it generates the scaffolding for you. In the command prompt, simply type:

ng generate service myservicename

ng g s myservicename

Step 2: Import the Injectable Member (it will be by default added when creating service through CLI cmd)

At the top of your new services file, add:

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

Step 3: Add the Injectable Decorator (it will be by default added when creating service through CLI cmd)
In step 2, we imported the Injectable member from the Angular Core library. Now we need to add the Injectable() decorator:

```
@Injectable()
```

Note: As with all other Angular decorators, we proceed the name with an @ symbol, and we do not add a semi-colon; at the end

Step 4: Export the Services Class
And finally, we create the class that contains the logic of our service:

```
export class ExampleService {
  // This is where your methods and properties go, for example:
 someMethod()
 {       return 'Hey!';    }
}
```

Step 5: Import the Service to your Component
Choose a component file and at the top, you must include the service member (line 2 below):

```
import { Component } from '@angular/core';
import { ExampleService } from './example.service';
```

Step 6: Add it as a Provider
Now you must add it to the providers array in the Component decorator metadata if want to use this service on component level (line 4 below):

```
@Component({
  selector: 'my-app',
 template: '<h1>{{ title }}</h1>',
 providers: [ExampleService]
})
```

Step 7: Include it through dependency injection
In the constructor arguments of the component class, we include it through dependency injection:

```
constructor (private _exampleService: ExampleService) { }
```

Step 8: Using the Service
Now we can access the service's methods and properties by referencing the _exampleService.

For example:

```
ngOnInit() {
 this.title = this._exampleService.someMethod();
}
```

>>>>>>>>>Question: What is use of decorator @Injectable. Will service run without it.>>>>>>>>>>>>>>>>>>>>>>>>
Sol. Yes that service will run untill it is not a nested service and ServiceClass Name is provided as '**providers' metadata** inside component decorator we need to use this service.

WHen there are nested service i.e use of one service iniside another then we surely need @Injectable({providedIn:'root'}).

>>>>>>>>>>>>>Question: What is Singleton in SERVICE and non-singleton in service.>>>>>>>>>>>>>>>>>>>>>>>>
Sol. SINGLETON in SERVICE: When any serivce 'providedIn' is of 'root' level say module level then that services no matter in how many components gets used creates/uses its single reference object that is changes is always synced accross

applictaion. From angular 17 by default it is singleton.

Whereas when that 'providedIn' is not passed say tried to create without Injectatable and of 'component' level (by providing serviceName as 'providers' metadata inside Component decorator of particular component)then various different objeccts are created at different components to use that service , this is Non-Singleton service say Component level in service.

```
@Component({
  selector: 'app-comp2',
  standalone: true,
  imports: [],
  templateUrl: './comp2.component.html',
  styleUrl: './comp2.component.css',
  providers:[NumlistService]
})
```

>>>>>>>>>>>>>>>>>>Question: What is **Tree Shaking in SERVICE** .>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Sol. These are the providers say step in build process that removes unused code , unused services from the code base. This process is Tree Shaking.

Made by default in Angular7 . This feature is gets implemented by 'providedIn' providers automatically.

How it auto Tracks now unused services>>Now since 'providedIn' tracks where , in which all components does the object of the services is made thus made easy to track which all services are getting used and which are unused , if unused shake them off , do not let them go with build codebase.

So 'providedIn' are also known as 'Tree Shakable Providers'

Tree Shaking

Tree Shakeable Providers are a way to define services and other things to be used by Angular's dependency injection system in a way that can improve the performance of an Angular application.

Tree shaking

Tree shaking is a step in a build process that removes unused code from a code base. Removing unused code can be thought as "tree shaking".

By using tree shaking, we can make sure our application only includes the code that is needed for our application to run.

For example,

Suppose we have a utility library that has functions a (), b () and c ().

In our application we import and use function a () and c () but do not use b (). We would expect that the code for b () to not be bundled and deployed to our users. Tree shaking is the mechanism to remove function b () from our deployed production code that we send to our user's browsers.

ROUTING

Routing is mechanism for navigating between page and displaying appropriated component/page on browser.

Angular Router is an official Angular routing library, written and maintained by the Angular Core Team.
It's a JavaScript router implementation that's designed to work with Angular and is packaged as @angular/router.

First of all, Angular Router takes care of the duties of a JavaScript router:
it activates all required Angular components to compose a page when a user navigates to a certain URL
it lets users navigate from one page to another without page reload
it updates the browser's history so the user can use the back and forward buttons when navigating back and forth between pages.

Angular Router allows us to:

- ✓ redirect a URL to another URL
- ✓ resolve data before a page is displayed
- ✓ run scripts when a page is activated or deactivated
- ✓ lazy load parts of our application.

HOW ROUTER WORKS IN ANGULAR17

When a user navigates to a page, Angular Router performs the following steps in order:
Every time a link is clicked or the browser URL changes, Angular router makes sure your application reacts accordingly.
To accomplish that, Angular router performs the following 7 steps in order:

Step 1 - Parse the URL

In step 1 of the routing process, Angular router takes the browser URL and parses it as a URL tree.
A URL tree is a data structure that will later help Angular router identify the router state tree in step 3

To parse the URL, Angular uses the following conventions:

- ✓ / - slashes divide URL segments
- ✓ () - parentheses specify **secondary routes**
- ✓ : - a colon specifies a **named router outlet**
- ✓ ; - a semicolon specifies a **matrix parameter**
- ✓ ? - a question mark separates the **query string parameters**
- ✓ # - a hashtag specifies the **fragment**
- ✓ // - a double slash separates multiple secondary routes

Step 2 – Redirect

Before Angular router uses the URL tree to create a router state, it checks to see if any redirects should be applied. **There are 2 kinds of redirect:**

local redirect: when redirectTo does not start with a slash. replaces a single URL segment

Example: { path: 'one', redirectTo: 'two' }

absolute redirect: when redirectTo starts with a slash. replaces the entire URL

Example: { path: 'one', redirectTo: '/two' }

Only one redirect is applied!

Step 3 - Identify the router state

Angular router traverses the URL tree and matches the URL segments against the paths configured in the router configuration.
If a URL segment matches the path of a route, the route's child routes are matched against the remaining URL segments until all URL segments are matched.
If no complete match is found, the router backtracks to find a match in the next sibling route

Step 4 - Guard - run guards

At the moment, *any* user can navigate *anywhere* in the application *anytime*. That's not always the right thing to do.
Perhaps the user is not authorized to navigate to the target component.
Maybe the user must login (*authenticate*) first.
Maybe you should fetch some data before you display the target component.
You might want to save pending changes before leaving a component.
You might ask the user if it's OK to discard pending changes rather than save them.
You can add *guards* to the route configuration to handle these scenarios.

Step 5 - Resolve - run resolvers
it resolves the required data for the router state.
Step 6- Activate
it activates the Angular components to display the page.
Step 7 – Manage
Finally, when the new router state has been displayed to the screen, Angular router listens for URL changes and state changes.
it manages navigation and repeats the process when a new URL is requested.

Router outlet

Router outlet is a dynamic component that the router uses to displays views based on router navigations.

Router outlet is a *Routing component.* An Angular component with a RouterOutlet that displays views based on router navigations.
The role of **<router-outlet>** is to mark where the router displays a view. (This is the location where Angular will insert the component we want to route to on the view)

The <router-outlet> tells the router where to display routed views.
The RouterOutlet is one of the router directives that became available to the AppComponent because AppModule imports AppRoutingModule which exported RouterModule.

Can say it as placeholder where we load our content component dynamically.

Router Link

```
<nav [ngClass] = "'menu'">
  <a routerLink="/home" routerLinkActive="active-link">Home</a> |
  <a routerLink="/add-book" routerLinkActive="active-link">Add Book</a> |
  <a routerLink="/manage-book" routerLinkActive="active-link">Manage Book</a>
</nav>
<router-outlet></router-outlet>
```

HOW TO set page Title DYNAMICALLY WITH PAGE ROUTING

```
app.routes.ts > routes > title
import { Routes } from '@angular/router';
import { DashboardComponent } from './dashboard/dashboard.component';
import { AboutusComponent } from './aboutus/aboutus.component';
import { ContactusComponent } from './contactus/contactus.component';

export const routes: Routes = [
    {path:'home',component:DashboardComponent,title:'Home page'},
    {path:'aboutus',component:AboutusComponent,title:'About US'},
    {path:'contactus',component:ContactusComponent,title:'Contact US'}
];
```

WILDCARD ROUTING(404 page fro not existing url)

Wildcard route to intercept invalid URLs and handle them gracefully.

A *wildcard* route has a path consisting of two asterisks (* *).
It matches *every* URL, the router will select *this* route if it can't match a route earlier in the configuration.
A wildcard route can navigate to a custom "404 Not Found" component or redirect to an existing route

{ path: '**', component: PageNotFoundComponent }
If the entire router configuration is processed and there is no match, router navigation fails and an error is logged.

Create a component for 404 page and then pass that component with routing
{path:'**',component:PageNotFoundComponent, title:'Page Not Found'}

HOW TO set BASE URL IN ROUTING IN ANGULAR17

i.e setting page at base url(index.ts): http://localhost:4000

```
export const routes: Routes = [
    {path:'',redirectTo:'home',pathMatch:'full'},
    {path:'home',component:DashboardComponent,title:'Home page'},
```

i.e now it will redirects to 'http://localhost:4000/home' if enters only base url 'http://localhost:4000' . Otherwise would have redirected to wildcard url i.e 'Page Not Found'.
This Path can be put anywhere not fixed position, but generally we put at top.
OR IF WANT that on hitting Base Url : 'redirects to home ' but do not show /hoem endpoint then do:

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

HOW TO set NAMED and MULTPLE ROUTES or (AUXILIARY ROUTES) IN

The Router outlet is a placeholder that gets filled dynamically by Angular, depending on the current router.
we'll see advanced uses of the <router-outlet> component such as named, multiple outlets and auxiliary routing.

How to Create a Named Router Outlet?

We can create a named Router outlet using the *name* property of the <router-outlet> component:

```
<router-outlet name="outlet1"></router-outlet>
```

>>>>>>>>>QUESTION: What will happen if we put multiple router-outlets in a component. Will multiple places components will start get loading ? Give cases when do we need to use Multiple router-outlets say named router-outlets and how? >>>

Sol:

You can have multiple outlets in the same template:

```
<router-outlet></router-outlet>
<router-outlet name="sidebar"></router-outlet>
```

The unnamed outlet is the primary outlet.
Except for the primary outlet, all other outlets must have a name.

>> **Case When write Multiple router-outlet not named ones**: Then in that case last router-outlet will be treated as primary router-outlet and wll load components in that router outlet only.

>> **Case When to use Multiple Named Router-Outlets**: a named router outlet is a powerful feature that allows you to define multiple outlets for rendering different components based on routing. This feature is especially useful in complex applications where you want to display different views simultaneously, such as in a multi-step form, dashboard layout, or when implementing modal dialogs.
>> Eg: in modules case: each module different module load....
>> Least used as not seo friendly.

>> Case How to create named and parameterized routes:

Create RouteLinks as:

```
<a [routerLink]="[{outlets:{outlet2:['aboutusnew']}}]">About US new</a>
<a [routerLink]="[{outlets:{outlet2:['aboutusnew,501']}}]">About US new 2</a>
```

Define Named Outlet:

```
<router-outlet name="outlet2"></router-outlet>
```

Define outlet name inside router endpoint:

Here these marked two are routes endpoints which will be loaded in named Outlets not in primary outlet.
>>>> Second marked route is also known as Parameterized Route since it takes /:id as param:

```
export const routes: Routes = [
    // {path:'',redirectTo:'home',pathMatch:'full'},
    {path:'',component:DashboardComponent},

    {path:'home',component:DashboardComponent,title:'Home page'},
    {path:'aboutus',component:AboutusComponent,title:'About US'},
    {path:'contactus',component:ContactusComponent,title:'Contact US'},
    {path:'aboutusnew',component:AboutusComponent,outlet:'outlet2'},
    {path:'aboutusnew/:id',component:AboutusComponent,outlet:'outlet2'},
    {path:'**',component:PagenotfoundComponent,title:'page not found'},
```

ROUTE GUARDS

Route Guards

Being web application developers, we are aware how a Server checks the permissions for the user on the navigation and returns the result Telling whether a user can access the resource or not.
In the same way, we need to achieve this on client-side. Route Guards are the solution for that. We can use the route guards to control whether the user can navigate the route or not.
Angular's route guards are interfaces which can tell the router whether or not it should allow navigation to a requested route. They make this decision by looking for a true or false return value from a class which implements the given guard interface.
In the Angular application in which authentication and authorization is required to navigate a route, the role of Angular route guard comes into the picture.

There are five different types of guards and each of them is called in a particular sequence.

The router's behavior is modified differently depending on which guard is used.

There are 5 types of Route Guards we use in Angular.

CanActivate
This helps to decide whether the route can be activated or not.1
CanActivateChild
This helps to decide whether child routes can be activated or not.
CanDeactivate
This helps to decide whether the route can be deactivated or not. Generally, this is used to warn a user if they are navigating from the component.
Resolve
This performs route data retrieval before any activation of the route.
CanLoad
Checks to see if the user can navigate to the module which is lazily loaded.

In Short:

- ✓ CanActivate guard (e.g. it checks route access).
- ✓ CanActivateChild guard (checks child route access).
- ✓ CanDeactivate guard (prompt for unsaved changes).
- ✓ Resolve guard (pre-fetching route data).
- ✓ CanLoad guard (check before loading feature module).

We can use the Angular CLI command
ng g g [guard name]

? Which interfaces would you like to implement?
>(*) CanActivate
( ) CanActivateChild
( ) CanMatch

AUTH GUARD

Auth – canActivate guard

Being web application developers, we are aware how a Server checks the permissions for the user on the navigation and returns the result telling whether a user can access the resource or not. In the same way, we need to achieve this on client-side. Route Guards are the solution for that. We can use the route guards to control whether the user can navigate the route or not.

Auth – canActivate guard

1. ***CanActivate***
   This helps to decide whether the route can be activated or not. It use to checks to see if a user can visit a route.
2. ***CanActivateChild***
   This helps to decide whether child routes can be activated or not. It use to checks to see if a user can visit a routes child.
3. ***CanDeactivate***
   This helps to decide whether the route can be deactivated or not. Generally, this is used to warn a user if they are navigating from the component. It use to checks to see if a user can exit a route.
4. ***Resolve***
   This performs route data retrieval before any activation of the route. It use to perform route data retrieval before route activation.
5. ***CanLoad***
   Checks to see if the user can navigate to the module which is lazily loaded. It use to checks to see if a user can route to a module that lazy loaded

Generally, in our application, we have a need where we want to check if the user is logged in or not. Ir that case, we can decide if we want to activate that route for that user or not.
Angular provides CanActivate Route for that.

ng g g [guard name]

IMPLEMENTING AUTHGUARD

We use guard in angular say: to preserve bandwidth usagee by non authenticated user to load a component..i.e untill not authenticated do not load component , preserving bandwidth of network and load only if is logged in. Like say we have a sessionStorage value that tells user is logged in or not then based on that we apply a canActivate Authguard for the same.

So first we generate a guard by running: ng g g auth and tell to create canActivate Guard.

TS auth.guard.ts U | TS app.component.ts M | TS app.routes.ts M | <> login.compo

app > TS auth.guard.ts > ...

```ts
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';

export const authGuard: CanActivateFn = (route, state) : boolean => {
  //create router obj
  const _router= inject(Router);
  let isloggedIn = sessionStorage.getItem('isloggedIn')
  if(isloggedIn==='false'){
    alert("Please login first, redirecting to Login Page!!")
    _router.navigate(['login'])
    return false;
  }
  return true;
};
```

Since our login logic in Login.ts is :

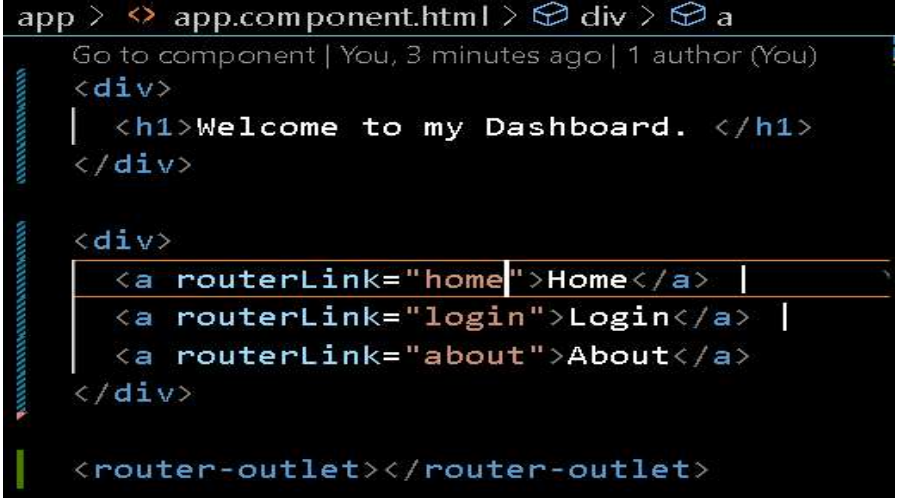
```ts
// login method:
login(userid:string,password:string) : void{
  if(userid==='atul' && password==='1234'){
    //create session storage
    sessionStorage.setItem("isloggedIn","true");
  }
  else{
    sessionStorage.setItem("isloggedIn","false");
  }
}
```

NOW implement this Router Guard to all the routes which needs to be protected: Somewhat like Middleware

You, 10 seconds ago | 1 author (You)

```ts
import { Routes } from '@angular/router';
import { DashboardComponent } from './dashboard/dashboard.component';
import { LoginComponent } from './login/login.component';
import { authGuard } from './auth.guard';

export const routes: Routes = [
    {path:'',component:DashboardComponent},
    {path:'home',component:DashboardComponent,title:'Home',canActivate:[authGuard]},
    {path:'login',component:LoginComponent,title:'Login Page'}
];
```

Here is my app.component.ts:



HOW TO set ROUTER LINK ACTIVE

RouterLinkActive

The **RouterLinkActive** is a directive for **adding** or **removing** classes from an HTML element that is bound to a **RouterLink**. Using this directive, we can toggle CSS classes for active Router Links based on the current **RouterState**. The main use case of this directive is to **highlight** which route is currently active. You can either make the font bold or apply some background color.

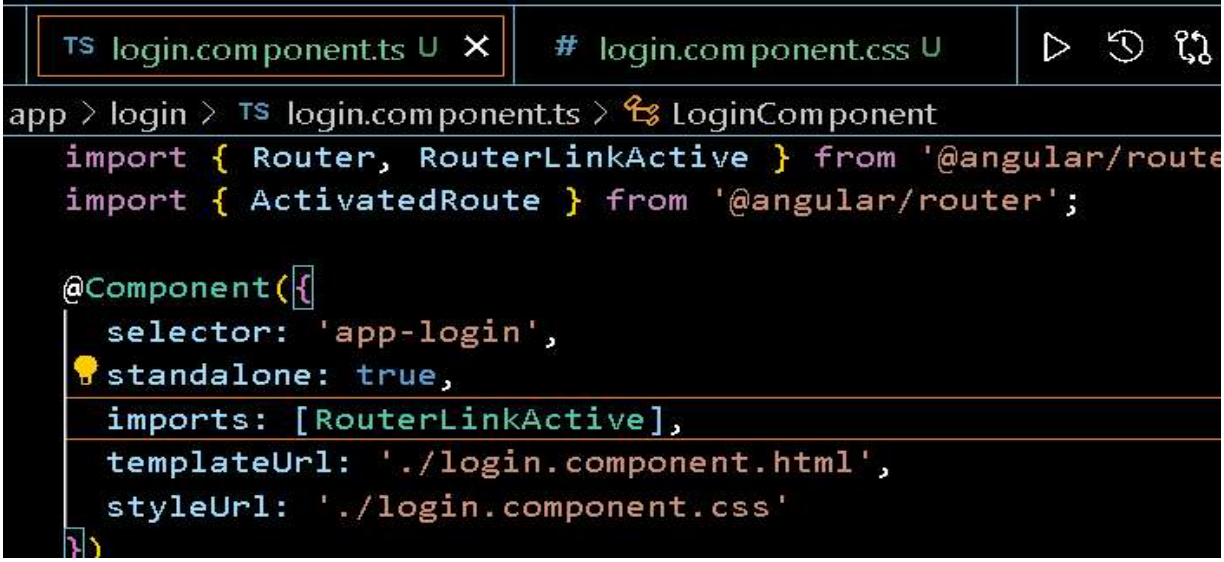
The **RouterLinkActive** Directive is applied along with the **RouterLink** directive. The right-hand side of **RouterLinkActive** contains a **Template expression**. The template expression must contain a **space-delimited** string of **CSS classes**, which will be applied to the element when the route is active.

Steps:

>> Add **routerLinkActive="active"** to all links, where **"active and home"** are css style classes.

```
<div>
  <a routerLink="home" routerLinkActive="active home">Home</a> |
  <a routerLink="login" routerLinkActive="active home">Login</a> |
  <a routerLink="about" routerLinkActive="active home">About</a> |
  <a routerLink="logout" routerLinkActive="active home">Logout</a>
</div>
```

>> import **RouterLinkActive** inside componnet where you are using it. In previous versions it will work without importing as well.



SESSION MANAGEMENT

Different kind of storage space is available for our data on client as well as server side, we can choose any of them according to our need and level of transparency. We use HTML5 Web Storage to store data in angular.

What is HTML Web Storage?

With web storage, web applications can store data locally within the user's browser

Before HTML5, application data had to be stored in cookies, included in every server request. Web storage is more secure, and large amounts of data can be stored locally, without affecting website performance.

This web storage of HTML5 has 2 api storage(session storage, local storage)

COOKIES MANAGEMENT

A cookie is a small piece of data sent from a website and stored on the user's machine by the user's web browsers while the user is browsing.

Cookies are small packages of information that are typically stored in your browsers and websites tend to use cookies for multiple things.

How to install a cookie

npm install ngx-cookie-service --save

cookie methods are

1. **Check** – This method is used to check the cookie existing or not.
2. **Get** - This method returns the value of given cookie name.
3. **GetAll** - This method returns a value object with all the cookies
4. **Set** – This method is used to set the cookies with a name.
5. **Delete** – This method used to delete the cookie with the given name
6. **deleteAll** – This method is used to delete all the cookies

DIFFERENCES :

Feature	Cookies	Session Storage	Local Storage
Data Storage Limit	About 4KB per cookie	About 5-10MB (depends on the browser)	About 5-10MB (depends on the browser)
Data Lifetime	Set expiration date; can persist for a long time	Cleared when the page session ends (i.e., when the tab is closed)	Persists until explicitly deleted by the user or application
Accessibility	Accessible to the server and client-side JavaScript	Only accessible to the client-side JavaScript	Only accessible to the client-side JavaScript
Domain & Path Scope	Sent with every HTTP request; can be scoped to a domain/path	Scoped to the origin (protocol + hostname + port)	Scoped to the origin (protocol + hostname + port)
Security	Can be made HttpOnly (not accessible to JavaScript); can be marked as Secure (sent only over HTTPS)	Accessible only to the originating page	Accessible only to the originating page
Synchronous/Asynchronous	Synchronous (blocking)	Synchronous (blocking)	Synchronous (blocking)
Use Cases	Storing authentication tokens, user preferences, tracking sessions	Storing temporary data for a session (e.g., form data, user selections)	Storing user preferences, application state, caching data for offline use
Storage Management	Manually managed; can be set to expire and needs to be deleted when no longer needed	Automatically cleared when the session ends	Manually managed; needs explicit deletion when no longer needed

Accessibility by JavaScript	Controlled: Can be marked as HttpOnly (not accessible via JavaScript)	Accessible: Read and write via JavaScript	Accessible: Read and write via JavaScript
Data Sent to Server	Automatically: Sent with every HTTP request; can be set to Secure (only sent over HTTPS)	Manually: Not sent automatically; needs explicit handling	Manually: Not sent automatically; needs explicit handling
Vulnerability to XSS	Medium: If not marked HttpOnly, cookies can be accessed via XSS. Sensitive data (like session tokens) can be compromised.	High: Accessible by JavaScript; if XSS occurs, an attacker can read/write any data stored in session storage.	High: Accessible by JavaScript; similar to session storage, any data stored can be accessed via XSS.

FORMS

Angular provides 2 different ways to collect and validate the data from a user.

1. Template-driven forms
2. **Model-driven forms (Reactive forms)**

Template Driven Forms

Template driven forms are simple forms which can be used to develop forms. These are called template-driven as everything that we are going to use in an application is defined into the template that we are defining along with the component.

>>>>>>>>>>For using Template Driven Forms(Less controll, fails for complex bussiness logic, dynamic validation, unit testing not that good in this form)

Prerequisite

We need to import **FormsModule** in an Application module file (**i.e.app. module.ts**).

Template Driven Forms features

- Easy to use.
- Suitable for simple scenarios and fails for complex scenarios.
- Similar to Angular 1.0.
- Two way data binding (using [(NgModel)] syntax).
- Minimal component code.
- Automatic track of the form and its data.
- Unit testing is another challenge.

>>>>>>>>>>Uses **ngForm** can be said collecction of controls (holds data) , everything depends on it for a form to be template driven:

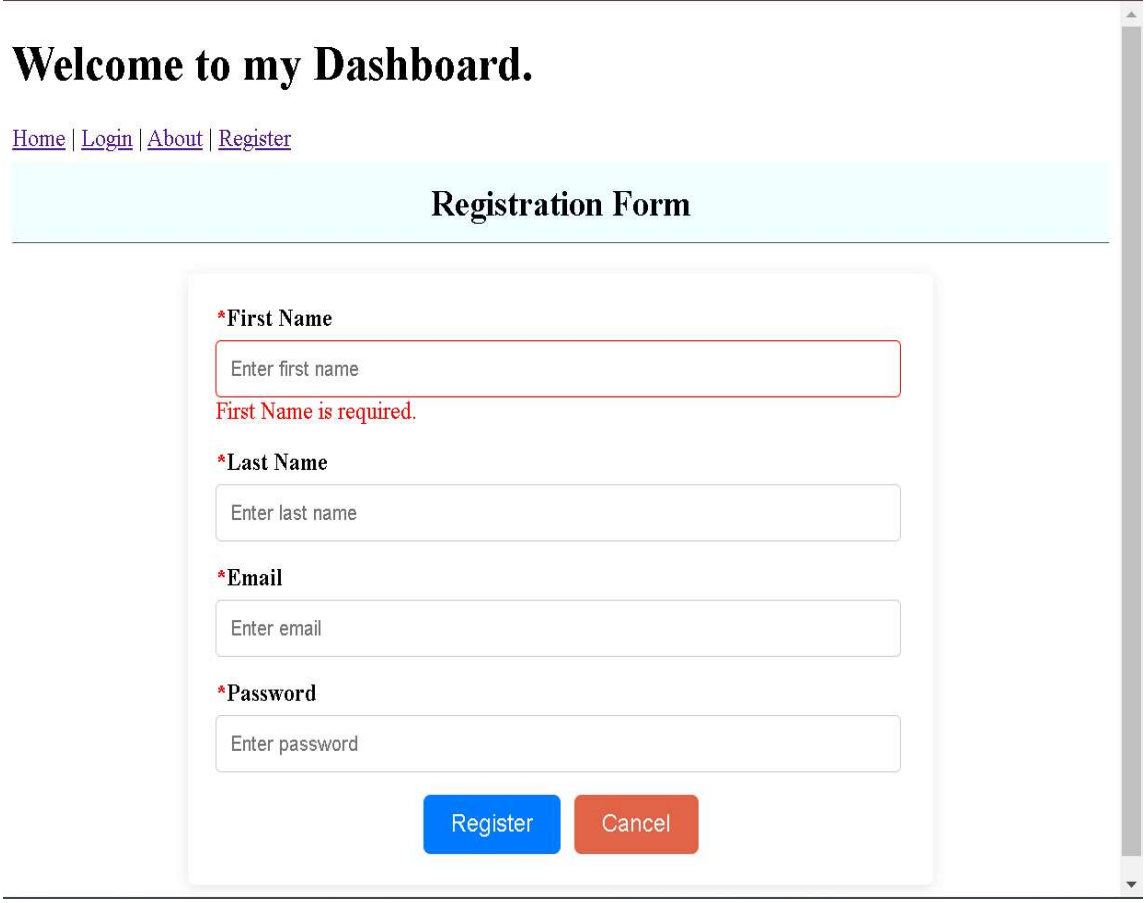
Eg:

template-driven-form.component.ts:

```ts
import { Component } from '@angular/core';
import { FormsModule, NgForm } from '@angular/forms';

@Component({
  selector: 'app-template-driven-form',
  standalone: true,
  imports: [FormsModule,CommonModule],
  templateUrl: './template-driven-form.component.html',
  styleUrl: './template-driven-form.component.css'
})
export class TemplateDrivenFormComponent {
  // Method to handle form submission
  register(form: NgForm) : void {
    if (form.valid) {
      // form.valid : worksa with required keyword
      console.log('Form Submitted!',form.value );
    } else {
      console.log('Form is invalid');
    }
  }
}
```

template-driven-form.html:

```html
<h2>Registration Form</h2>
<!-- using template refernce variable to chck for errors and #regForm template reference variable holding values of form data. -->
<div class="form-container">
    <form #regdata='ngForm' (ngSubmit)="register(regdata)">
        <div class="form-group">
            <label for="firstName"><span class="error">*</span>First Name</label>
            <input type="text" id="firstName" name="firstName" placeholder="Enter first name" ngModel
            required
                #fname='ngModel' />/#firstName
            <span *ngIf='fname.touched && !fname.valid' class="error"> First Name is required.</span>
        </div>/.form-group

        <div class="form-group">
            <label for="password"><span class="error">*</span>Password</label>
            <input type="password" id="password" name="password" placeholder="Enter password" required
            ngModel
                #password='ngModel' />/#password
            <span class="error" *ngIf='password.touched && !password.valid'>Password is required.</span>
        </div>/.form-group

        <div class="form-actions">
            <button type="submit" [disabled]="regdata.invalid">Register</button>
            <button type="button" class="button2">Cancel</button>
        </div>/.form-actions
    </form>
</div>/.form-container
```



NOTE: which ever inputs will have, **ngModel** only their data will be available in **ngForm** . If remove ngModel from the input then for that input data won't be available in ngForm. Then this data is passed to **ngSubmit** function using ngForm **template reference variable(here #regForm)**.

Also need to **import FormsModule** else gives error.

Two main functionalities offered by NgForm and NgModel are the permission to retrieving all the values of the control associated with the form and then retrieving the overall state of controls in the form.
To expose ngForm in the application, we have used the following code snippet.

```
<form #regForm='ngForm' (ngSubmit)="Register(regForm)" >
```

In this, we are exporting the ngForm value in the local variable “regform”. Now, the question arises, whether or not we need to use this local variable. Well, the answer is no. We are exporting ngForm in the local variable just to use some of the properties of the form and these properties are -

1. regForm.Value: It gives the object containing all the values of the field in the form.
2. regForm.Valid: This gives us the value indicating if the form is valid or not if it is valid value is true else value is false.
3. regForm.touched: It returns true or false when one of the field in the form is entered and touched.

Reactive Forms

In a model-driven approach, the model which is created in the .ts file is responsible for handling all the user interactions/validations.

For this, first, we need to create the Model using Angular's inbuilt classes like **formGroup** and **formControl** and then we need to bind that model to the HTML form.

This approach uses the Reactive forms for developing the forms which favor the explicit management of data between the UI and the Model.

With this approach, we create the tree of Angular form controls and bind them in the native form controls. As we can create the form controls directly in the component, it makes it a bit easier to push the data between the data models and the UI elements.

Prerequisite

we need to import **ReactiveFormsModule** in our **app.module.ts** file.

Reactive Forms Features

- More flexible, but needs a lot of practice
- Handles any complex scenarios
- More component code and less HTML markup
- Easier unit testing

NOTE: **Here major roel play by .ts file. ALl validation , dynamic data control, patching form data etc is done from .ts**

FormControl

It tracks the value of the controls and validates the individual control in the form.
In Reactive forms we initialize FormControl object to use form functionality into our component , which engaged with our html form .
And when we update our form control's value than it will directly be reflected to our FormControl object that we created previously .

FormGroup

Tracks the validity and state of the group of FormControl Instance or moreover, we can say the formgroup to be a collection of FormControls.
like :- Validations , name of input control etc ...

FormBuilder

This helps us to develop the forms along with their initial value and there validations.
Angular has a new helper Class called FormBuilder. FormBuilder allows us to explicitly declare forms in our components. This allows us to also explicitly list each form control's validators.

eg:

p > reactiveform > TS reactiveform.component.ts > ReactiveformComponent

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

@Component({
  selector: 'app-reactiveform',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './reactiveform.component.html',
  styleUrl: './reactiveform.component.css'
})
export class ReactiveformComponent implements OnInit {

  //then create constructor: with dependency injection so as new keyword is not required much
  constructor(private _fb:FormBuilder){}

  // first will create a FormGroup here,
  // to avoid overuse of new keyword as memory releasing will be tough, use formBUilder dependency injection and creating formGroup
  oject by the FormBuilder instance :
  regForm: FormGroup = this._fb.group({
    fname: ['', Validators.required],
    lname: ['', Validators.required],
    email: ['', [Validators.required, Validators.email]], // Email validation
    password: ['', [Validators.required, Validators.minLength(6)]] // Password validation
  });

  ngOnInit() : void{}
```

```ts
  //register method:
  register() : void {
    console.log(this.regForm.value);
    if (this.regForm.valid) {
      console.log("Valid form");
    } else {
      console.log("Invalid form");
      this.regForm.markAllAsTouched();
    }
  }

}
```

.html:

```
pp > reactiveform > reactiveform.component.html > div.form-container > form > div.form-group > span.error
<h2>Reactive Registration Form</h2>
<!-- using property binding to bind formGroup created in .ts eg: regForm ...it will hold all form data. since
regForm is a formgroup availbale in .ts so no need to pass with event binding... -->
<div class="form-container">
    <form [formGroup]="regForm" (ngSubmit)="register()">
        <div class="form-group">
            <label for="firstName"><span class="error">*</span>First Name</label>
            <input type="text" id="firstName" name="firstName" placeholder="Enter first name"
            formControlName='fname'
                required />/#firstName
            <span *ngIf="regForm.controls['fname'].touched && regForm.controls['fname'].invalid" class="error">
            First Name is required.</span>
        </div>/.form-group

        <div class="form-group">
            <label for="password"><span class="error">*</span>Password</label>
            <input type="password" id="password" name="password" placeholder="Enter password" required
                formControlName='password' />/#password
            <span class="error"
                *ngIf="regForm.controls['password'].touched && regForm.controls['password'].invalid">Password is
                required.</span>/.error
        </div>/.form-group

        <div class="form-actions">
            <button type="submit" [disabled]="regForm.invalid">Register</button>
            <button type="button" class="button2">Cancel</button>
        </div>/.form-actions
    </form>
```

>>>>>>>>>>>>>Validators.compose: making validation fast.

```
// used Validators.compose to compose multiple validators together , make approach fast...
regForm: FormGroup = this._fb.group({
  fname: ['', Validators.compose([Validators.required, Validators.minLength(5), Validators.maxLength(20)])],
  lname: ['', Validators.compose([Validators.required, Validators.minLength(5), Validators.maxLength(20)])],
  email: ['', Validators.compose([Validators.required, Validators.email])],
  password: ['', Validators.compose([Validators.required, Validators.minLength(5), Validators.maxLength(20)])]
});
```

>>>>>> Now providing different error message based on these validators:

```
        <div class="form-group">
            <label for="firstName"><span class="error">*</span>First Name</label>
            <input type="text" id="firstName" name="firstName" placeholder="Enter first name" formControlName='fname'
                required />/#firstName
            <div *ngIf="regForm.controls['fname'].touched && regForm.controls['fname'].invalid" class="error">
                <!-- different error message based on different validators -->
                <span *ngIf="regForm.controls['fname'].hasError('required')">
                    First Name is required!
                </span>
                <span *ngIf="regForm.controls['fname'].hasError('minlength')">
                    First Name should be at least 5 characters.
                </span>
                <span *ngIf="regForm.controls['fname'].hasError('maxlength')">
                    First Name should not exceed 20 characters.
                </span>
            </div>/.error
        </div>/.form-group
```

GET AND RESET VALUE OF A FORMGROUP IN REACTIVE FORM

Eg: resetting full(whole form: **way1**: simply set button type="reset", way2: adding a click event with def. this.regForm.reset()) / half(eg: only captcha,password) form using CANCEL BUTTON:

get value & reset in Reactive Forms

- To get the value of form control named as name after form submit, we need to write the code as below.

```
this.userForm.get('name').value
```

- If we want to get all values of the form then we can write code as given below.

```
this.userForm.value
```

- To reset the form we need to call reset() method on FormGroup instance. Suppose userForm is the instance of FormGroup then we create a method as given below.

```
resetForm() { this.userForm.reset(); }
```

We can call the above method using a button as following.

```
<button type="button" (click) = "resetForm()">Reset</button>
```

- If we want to reset form with some default values, then assign the default values with form control name of the form.

```
resetForm() { userForm.reset({ name: 'Mahesh', age: 20 });}
```

>>>>Say if want to get the value of a controller , say want to **get the value of name input** we use **formGroup's (regForm) -> method get()** eg: this.regForm.get('fname').value; **in .ts file.....**

```
//register method:
register() : void {
 console.log(this.regForm.value);
 if (this.regForm.valid) {
   console.log("Valid form");
   console.log("string format, single control value of fname: ",this.regForm.get('fname')!.value)
 } else {
```

Getting certain field values and RESETTING HALF FORM VALUE:

i.e letting form fields remains same only reset certain sensitive fields. What we can do:

>>>>> use formGroup's reset function and pass all the fields that you want tp remain hold values what typed (for that use get value) ...and leave the field that you wnat to reset eg: password left.

Now what will happen is once click reset a event will trigger and the event method will get all typed values and set in form while those fields not inside .reset({})

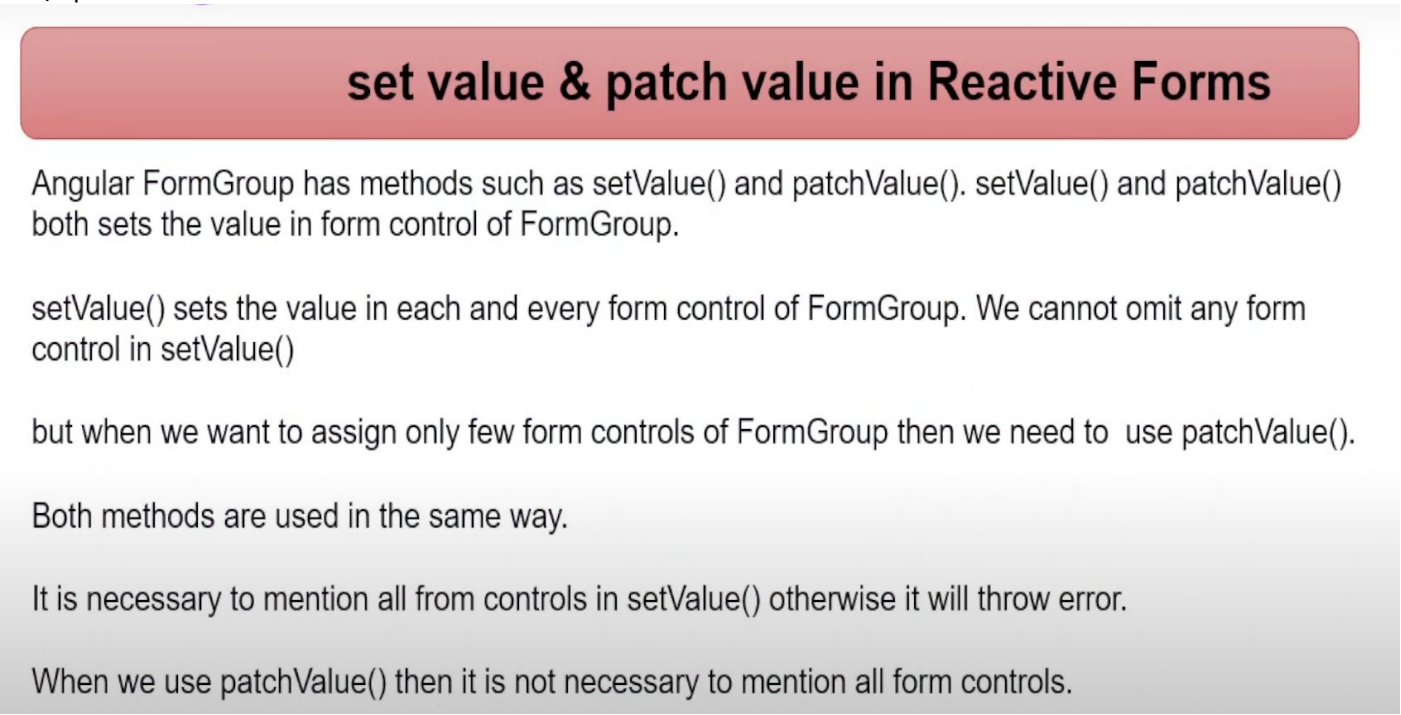
gets empty.

```
<button (click)="resetFields()" class="button2">Reset</button>
```

```
// reset all the fields except one declared here: say let rest data form same only reset sensitive fields to blank:
resetFields() : void{
  // give here fields that you do not want to reset:
  this.regForm.reset({
    fname:this.regForm.get('fname')!.value,
    lname:this.regForm.get('lname')!.value,
    email:this.regForm.get('email')!.value
  });
}
```

SET COMPLETE AND SET PARTIAL FORMGROUP FIELDS WITH VALUES

Say filling whole form using formGroup with certain values by default or filling partial form using formGroup. Eg: Edit/update Form....




```
<button (click)="updateFullForm()" class="button2">Update Full</button>
<button (click)="updatePartialForm()" class="button2" style="margin-top:5px">Update Partial</button>
```

>>>> Filling Full Form: uses: **.setValue({})**

>> Have to give all foermGroup fields else error:

```
// update all form fields ...
updateFullForm() : void{
  this.regForm.setValue({
    //give all controlls here...refers to .ts declared formGroup definition...no matter declared field binded
    in form or not if using .setValue , have to give all fields...
    // Useful to bind form with some values: helpful say in editing,updating form/profile etc:
    fname:"Rahul",
    lname:"Sharma",
    email:"rkhul@gmail.com",
    password:"hellopassword"
  })
}
```

>>>> Filling Partial Form say putting patch: uses: **.patchValue({})**

```
// update Partial form fields ...
updatePartialForm() : void{
  this.regForm.patchValue({
    //give few controlls here...refers to .ts declared formGroup definition.
    //Useful to bind form with some values: helpful say in prefilling some form fields say : user name , id
    etc...
    email:"rkhul@gmail.com",
  })
}
```

VALUE CHNAGES AND STATUS CHANGES in form

NOTE: Do inside ngOnInit

VALUE CHANGES DETECTION:

>> PARTIAL FORM FIELDS VALUE CHANGES DETECTION:

Getting updated changed values of a form control to a variable (SIMILAR TO SETSTATE of react that return each value typed...say keeps field value changes uptodate). For that we need to >>get the .valueChanges using formGroup for that particular controller . >> then: subscribe to that change and store in a variable.

>> ALL FORM FIELDS VALUE CHANGES DETECTION:

Getting updated changed values of whole formGroup in {...} form (SIMILAR TO SETSTATE of react in a object form that return each value typed...say keeps field value changes uptodate). For that we need to >>get the .valueChanges using FormGroup name. >> then: subscribe to that change and store in a variable.

SNAPSHOT:

```
ngOnInit() : void{

  // detecting Partial form fields value chnages:::: say doing setState of react
  this.regForm.get('fname')?.valueChanges.subscribe(newVal=>console.log(newVal));

  // detecting whole formGroup fields value chnages:::: say doing setState of react
  this.regForm.valueChanges.subscribe(newVal=>console.log("whole form changes: ",newVal));
}
```

STATUS CHANGES DETECTION:

>> PARTIAL FORM FIELDS STATUS CHANGES DETECTION:

Getting updated changed VALID or INVALID STATUS of a form control to a variable. (BASED ON VALIDATORS : valid or invalid current status is set and displayed). For that we need to >>get the .statusChanges using formGroup for that particular controller . >> then: subscribe to that change and store in a variable.

>> ALL FORM FIELDS STATUS CHANGES DETECTION:

Getting updated changed VALID or INVALID STATUS of a whole form gorup or deciding which all formcontrols status is required in ({})... . (BASED ON VALIDATORS : valid or invalid current status is set and displayed). For that we need to >> get the .statusChanges using FormGroup. >> then: subscribe to that change and store in a variable.

```
// detecting Partial form fields status changes: (valid or invalid curr status based on validators of that Formcontrol)
this.regForm.get('fname')?.statusChanges.subscribe(newVal=>{console.log(newVal)});

// detecting whole formGroup fields status changes: (valid or invalid curr status based on validators of that Formcontrol)
this.regForm.statusChanges.subscribe(newVal=>{console.log("whole form status changes: ",newVal)});
```

FORM ARRAY IN REACTIVE FORMS

i.e used when need to generate form control in runtime. say generate or create aform control dynamically in array form. Like in Experience or Education section of form exists "ADD MORE" button which dynamically adds a controller to form.

Form Array in Reactive Forms

It is a class that tracks the value and validity state of array of FormControl, FormGroup and FormArray. FormArray aggregates the values of each child FormControl into an array.

Form arrays allow you to create new form controls dynamically.
setValue() and patchValue() are also the methods of FormArray class.
setValue(): Sets the value of the FormArray. We need to pass an array that must match the structure of the control. If our array does not match the structure of the control, setValue() will throw error.

patchValue(): Patches the value of FormArray. We need to pass an array that should match the structure of the control. If the array that we pass, does not match the structure of the control completely, then patchValue() will patch only those fields which are matching with the structure of the control and rest will be as it is.

Creating a FormArray in Angular's reactive forms is typically done inside the ngOnInit lifecycle hook. This is because ngOnInit is called once the component is initialized, making it the perfect place to set up your form structure, including any dynamic controls you might need.

Nested form arrays can come in handy when you have a group of form controls but you're not sure how many there will be.
Form arrays allow you to create new form controls dynamically.
Access the array: You can access the associated FormArray using the get method on the parent FormGroup

Ex: this.form.get('cities').
Get the value: the value property is always synced and available on the FormArray.
Set the value: You can set an initial value for each child control when instantiating the FormArray, or you can set the value programmatically later using the FormArray's setValue or patchValue methods.
Listen to value: If you want to listen to changes in the value of the array, you can subscribe to the FormArray's valueChanges event. You can also listen to its statusChanges event to be notified when the validation status is re-calculated.
Add new controls: You can add new controls to the FormArray dynamically by calling its push method
Ex: this.form.get('cities').push(new FormControl());

Step 1: Import Necessary Modules

Make sure you have the necessary Angular modules imported.

typescript Copy code

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

Step 2: Create the Component

Example: Dynamic Form Array Component

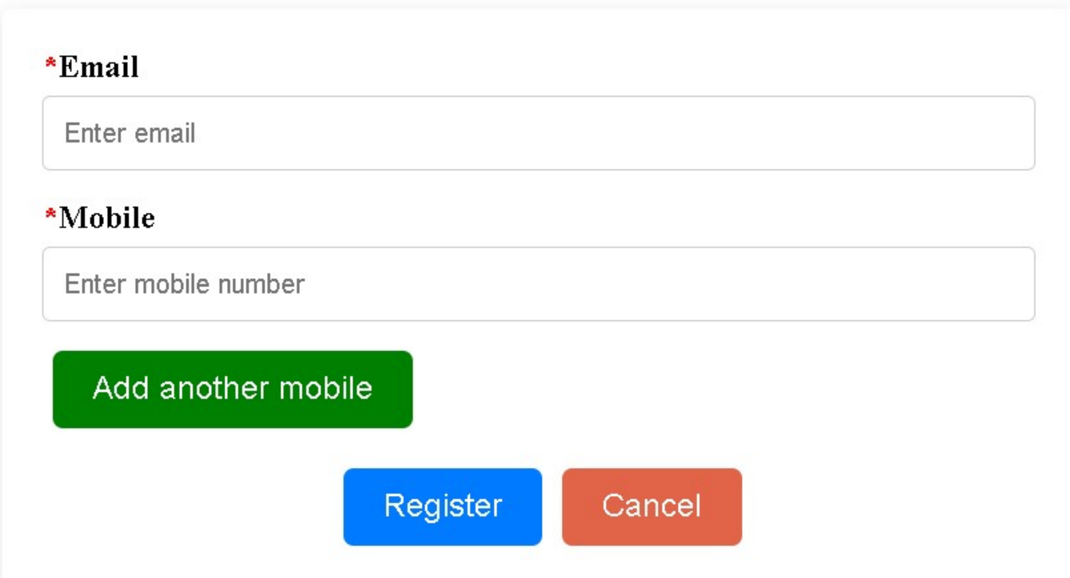
```
<!-- Dynamic Form Controller goes here: -->
<div formArrayName="mobiles">
    <div *ngFor="let mobileno of mobiles.controls; index as idx" class="form-group">
        <label for="mobileno"><span class="error">*</span>Mobile</label>
        <input type="text" name="mobileno" placeholder="Enter mobile number" required [formControlName]="idx" />
        <!-- Show error only if the mobile input is touched and invalid -->
        <!-- Show error messages based on validation -->
        <span class="error" *ngIf="mobileno.touched && mobileno.invalid">
            A valid Mobile is required.
        </span>/.error
        <span class="error" *ngIf="mobileno.errors?.['invalidMobile'] && mobileno.touched">
            Mobile must start with 6, 7, 8, or 9 and be exactly 10 digits.
        </span>/.error
        <!-- Show delete button only for dynamically added controls -->
        <button *ngIf="idx > 0" (click)="deletemobileidx(idx)" style="margin:5px">Delete</button>
    </div>/.form-group

    <!-- Add button to add a new control dynamically -->
    <button type="button" (click)="addmoreControlidx()" style="background-color: green;"
        [disabled]="mobiles.invalid">Add another mobile</button>
</div>
```

>>> .ts:

```
pp > reactive-form-array > TS reactive-form-array.component.ts > ...
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { AbstractControl, FormArray, FormBuilder, ReactiveFormsModule, Validators } from '@angular/forms';


// Custom validator function for mobile number
function mobileNumberValidator(control: AbstractControl): { [key: string]: boolean } | null {
  const value = control.value;
  // Check if the input is a valid number (starting with 6-9 and 10 digits long)
  const isValidNumber = /^[6-9]\d{9}$/.test(value);
  return !isValidNumber ? { invalidMobile: true } : null;
}

@Component({
  selector: 'app-reactive-form-array',
  standalone: true,
  imports: [CommonModule,ReactiveFormsModule],
  templateUrl: './reactive-form-array.component.html',
  styleUrl: './reactive-form-array.component.css'
})
```

```
export class ReactiveFormArrayComponent {

  constructor(private _fb:FormBuilder) {}

// creating form group:
  regForm!:any;

  ngOnInit() : void{
    // Initialize the form with an empty FormArray for mobiles
    this.regForm = this._fb.group({
      email: ['', [Validators.required, Validators.email]],
      mobiles: this._fb.array([
        this._fb.control('', [Validators.required, mobileNumberValidator])
      ])
    });
  }


  // DYNAMIC FORM CONTROLLERS METHOD:====
  // Method to delete a mobile control
  deletemobileidx(idx:number) : void{
    this.regForm.get('mobiles').removeAt(idx)
  }
```

```
// to add another control....simply push new FormControl in FormArray
addmoreControlidx() : void{
  this.mobiles.push(this._fb.control('', [Validators.required, mobileNumberValidator]));
}

// Getter for accessing the mobiles FormArray
get mobiles(): FormArray {
  return this.regForm.get('mobiles') as FormArray;
}

submitForm() : void {
  console.log(this.regForm.value);
  if (this.regForm.valid) {
    console.log("Valid form: ",this.regForm.value );
  }
  else {
    // Mark all controls as touched to show validation messages
    this.regForm.markAllAsTouched(); // Mark all fields as touched to show validation errors
  }
}
```



>>>

ARC Tutorials

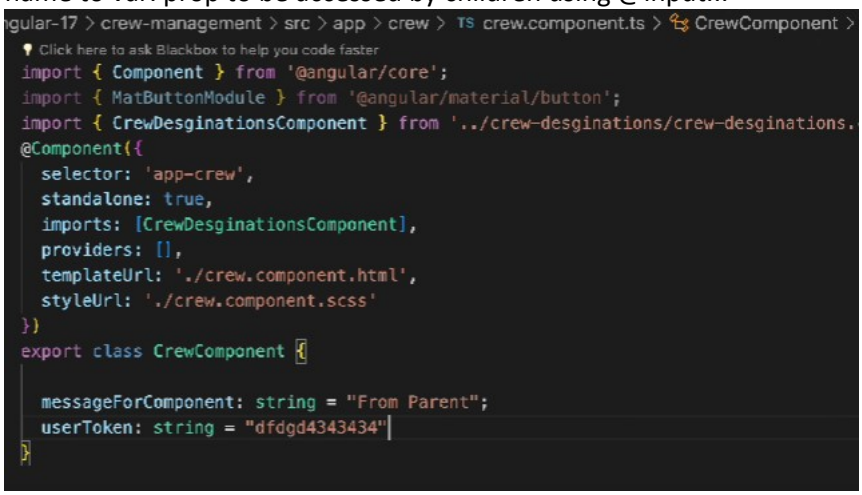
Angular 17 - Component Communications

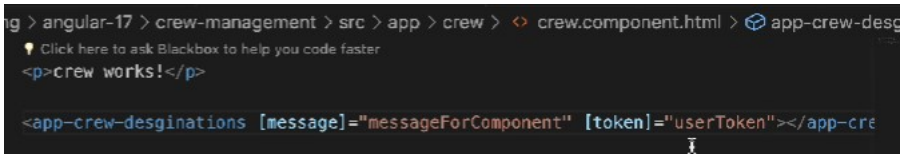
- Component communication in Angular is crucial for building modular and scalable applications.
- Two commonly used mechanisms for communication between components are @Input and @Output

ARC Tutorials

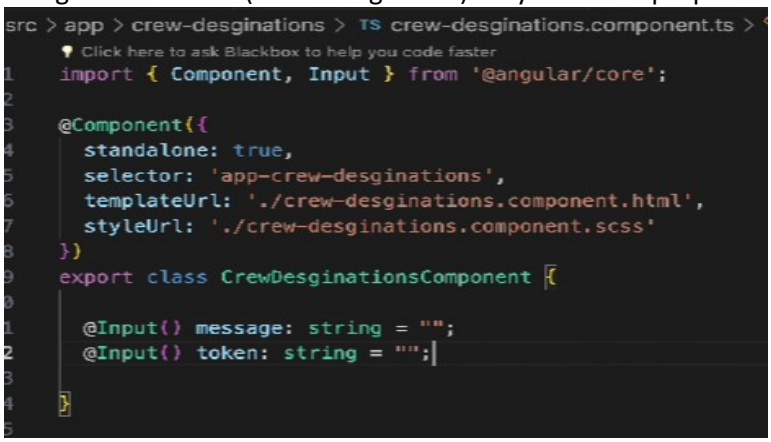
- @Input -> receiving inputs to the component
- @Output -> Sending data from component to parent

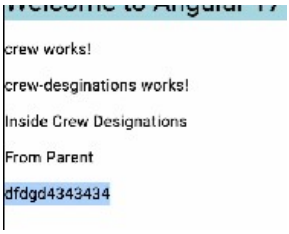
Eg: INPUT: inside parent passing [message] as prop. to child (crew-designation) from parent(crew):... giving some custom name to var. prop to be accessed by children using @input...

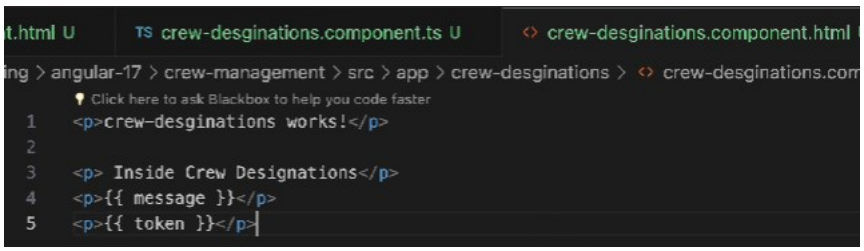




>>> go INSIDE Child (crew-designation) : say to access props from parent using @input we can define(by default "") and use:







>>>>>>>>>>>>>>>>>>>>>>>>>>>@OUTPUT from child (crew-designation) to parent(crew):>>>>>>>>>>>>>>>>>>

Say some action happens on button then notify it to parent uses event binding and @output to emit this event to parent from child.

```
<p>crew-desginations works!</p>

<p> Inside Crew Designations</p>
<p>{{ message }}</p>
<p>{{ token }}</p>

<button (click)="sendMessage();">Send Message to Parent</button>
```

```
export class CrewDesginationsComponent {

  @Input() message: string = "";
  @Input() token: string = "";

  @Output() messageEvent = new EventEmitter<string>();

  sendMessage(){
    this.messageEvent.emit("Hello from Child to Parent");
  }
```

i.e whenever click event will happen it wil take the message and event is called and emits the message to parent form child :

```
    standalone: true,
    imports: [CrewDesginationsComponent],
    providers: [],
    templateUrl: './crew.component.html',
    styleUrl: './crew.component.scss'
  })
  export class CrewComponent {

    messageForComponent: string = "From Parent";
    userToken: string = "dfdgd4343434";

    receivedMessage: string = "";

    receingMessage(message: string): void {
      this.receivedMessage = message;
    }
  }
```

```
<p>crew works!</p>

<app-crew-desginations [message]="messageForComponent" [token]="userToken"
    (messageEvent)="receingMessage($event)"
></app-crew-desginations>

<p>{{ receviedMessage }}</p>
```

crew works!

crew-desginations works!

Inside Crew Designations

From Parent

dfdgd4343434

Send Message to Parent

Hello from Child to Parent

OBSERVABLES

RxJS Observables

- **Observables**
  - Observables emits data over a period of time
  - Observables needs to be subscribed else it won't do anything on its own!
  - **An Observer has 3 methods namely -> next, complete and error**
  - *Angular uses Observables extensively under the hood for Routing, Forms and almost all features*
  - A sample observable is defined as follows

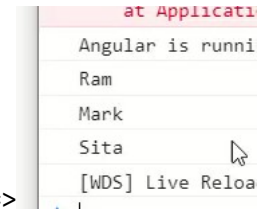
```
agents = new Observable(
   function subscribe(subscriber){
     try{
       subscriber.next("Ram");
       subscriber.next("Mark");
     } catch(e){
       subscriber.error(e);
     }
   }
);
```

>>>> First define variable of type Observable in the component, then inside ngoninit we create new Observable and assign it to that created variable. This new Observable will have function reciving observer and every observer has 3methods: (next values to pass using .next() i.e what values it will emit can emit multiple values using multiple .next()....,
next will have error using .error(), complete using .complete()).

>>>Now we need to subscribe this newly created Observable variable because wothhout subscribe it is useless. So subscribe the data it is returing and use it.

=>

=>



RxJS Of Operator

- **Of Operator**
  - Make observable from a string or array or an object
  - **Where to use it?**
    - Whenever we want to pass a variable which has to be Observable instead of Array or String, we use Of Operator
  - **Hands-On Examples**
    - viewers: Observable<string[]> = of(['welcome', 'to', 'arc', 'tutorials']);

```
studentList = ['Mark', 'Ram', 'Sita', 'Lisa'];
students: Observable<string[]> = of(this.studentList);

studentNames: Observable<string> = of('Ram');

studentObj = {
  id: 10,
  name: 'Ram'
}

student$: Observable<any> = of(this.studentObj);
```

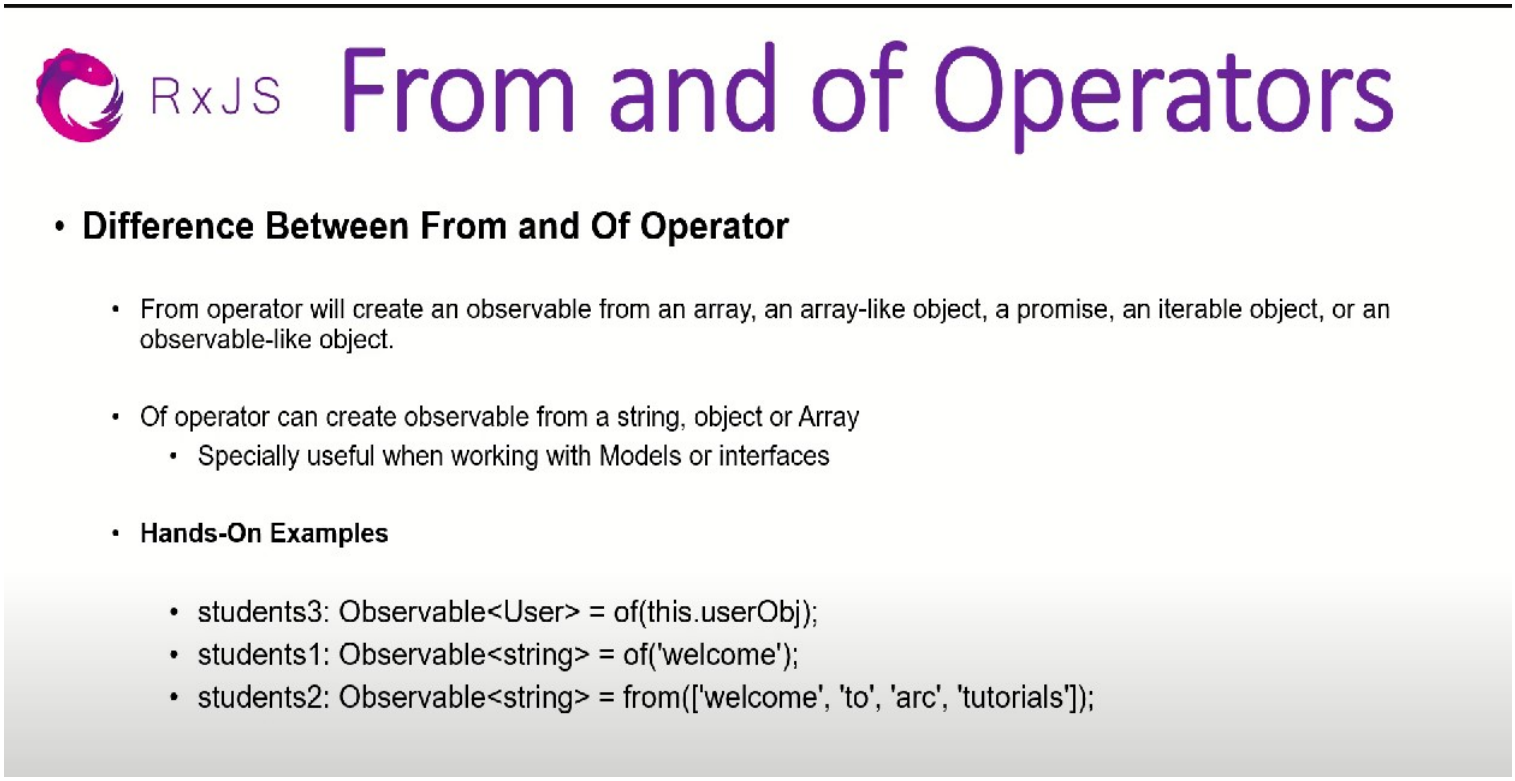
```
ngOnInit(): void {

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

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

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

>>>>>>>>>

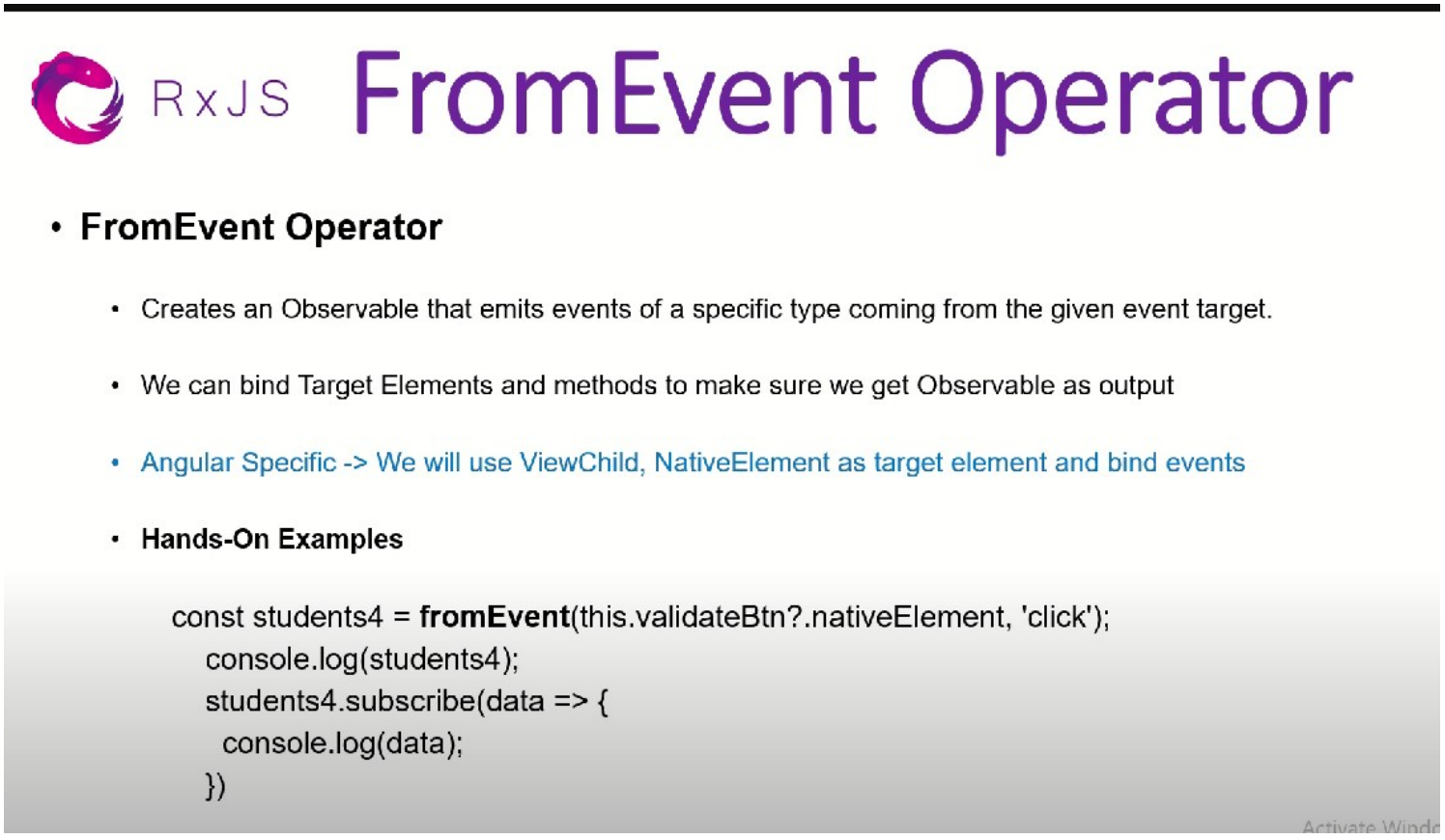
RxJS From Operator

- **From Operator**
  - From operator will create an observable from an array, an array-like object, a promise, an iterable object, or an observable-like object.
  - Remember it will always take Array or Array like
  - Lot of people will confuse “from” operator and “of” operator – **I will explain difference in next episode in detail**
  - **Hands-On Examples**
    - students2: Observable<string> = from(['welcome', 'to', 'arc', 'tutorials']);

>>>> It gives one by one while of returns whole array/string obj.



>>>>create observable from event:



>> in .html:

```
<button #validate (click)="rxJsEventObservable()">
    Click For Observable
</button>
```

>> in .ts

create viewChild decorator , since have to receive only one value else would have used view Children

```
@ViewChild('validate')
validate: ElementRef;
```

now use it inside method to convert it into observable:

```
rxJsEventObservable(){
  const btnObservable$ = fromEvent(this.validate?.nativeElement, 'click');

  btnObservable$.subscribe(data => {
    console.log(data);
  })
}
```