Angular

How Angular project is executed?

When we run an Angular project, the main.ts file gets executed. From main.ts the Anguler comes to know about the AppModule. From here, it goes to app.module.ts file. There, it comes to know about AppComponent. SO it goes to app.component.ts file. From here, it comes to know about the app-root which is in the selector property of the component and also know about the view from the templateUrl property. After this the index.html file will be loaded in the browser where the <app-root></app-root> in index.html will be replaced by the content in the template provided by app component.

When we start the application by hitting the ng-serve command, it rebuilds the project then it will create javascript bundles automatically to the index.html file. So, the script files are injected into the index.html by the angular cli.

Component

Component is a TypeScript class decorated with @Component decorator and it contains methods & properties which can use in html. The AppComponent (Root Component)is the main component in any Angular application. The root component can have several nested components and at the end we will have a complete application consisting of different components.

We can have components for header, navbar, main body, sidebar, footer,etc. We can also have components for home, about, contact present in the navbar.

```
@Component({
selector: 'app-component-overview',
templateUrl: './component-overview.component.html',
styleUrls: ['./component-overview.component.css'] })
export class AppComponet{
title='AngularApp';
message = 'Angular is the app of apps';
}
```

So a component has :

A Class, that contains the code required for the view template i.e. contains the UI logic

A View template, defines the user interface. It contains the Html, directives and data binding.

A decorator, it adds meta data to the class, making it a component.

In order to use a component, we need to export it, and register it in app module.

We can use the value in selector property as an html tag.

Template property of Component:

With template property, we can write html in the component file. It won't need the html file.

```
@Component({
selector: 'app-component-overview',
template: '<div><p>This web takes cookies</p></div>',
styleUrls: ['./component-overview.component.css'] })
export class AppComponet{
title='AngularApp';
message = 'Angular is the app of apps';
}
```

In templateUrl property, we specify the path to a html file to use as a view file. While for template property we specify the html inside a string. For multiline html code using template property, user``(tactiks). Its good practice to write the html code in separate file, if we have 3 or more lines of html code.

Disadvantage: We know about errors in html at runtime only. We are mixing typescript code with html code.

Styles property of Component:

With style property, we can add css style in the component file. No need for css file.

```
@Component({
selector: 'app-component-overview',
template: '<div><p>This web takes cookies</p></div>',
styles: ["div{margin: 10px 0px; padding: 10px 20px;}",
"p{font-size:40px}"]})
export class AppComponet{
title='AngularApp';
message = 'Angular is the app of apps';
}
```

It is similar to template property and has similar disadvantages.

Selectors in Angular:

There are different selector in Angular:

1. We can use a selector like an **HTML tag**.
   We do so by using this syntax.
   Selector : ‘app-nav’,
   Example: <app-nav></app-nav>
2. We can use a selector like a **HTML attribute**.
   We do so by using this syntax: Write the value between []
   Selector : ‘[app-nav]’,
   Example: <div app-nav></div>
3. We can use a selector like a **CSS class**.
   We can do so by using this syntax: Add a dot(.) before the value.
   Selector: ‘.app-nav’,
   Example : <div class=”app-nav”></div>

Data binding:

It allows us to communicate between a component class and its corresponding view template or vice versa.

One-way data binding:

It is when we can access a component class property in its corresponding view template Or when we can access a value form view template in corresponding component class property. I.E only in one direction.

Two-way data binding:

Binds data from component to view template and view template to component class. This is a combination of property binding and event binding i.e. change in component will be seen in view and change in view will be seen in component. In Both Directions.

Property binding is done by [].

Event binding is done by ().

So for two way binding, [()]. This is banana box syntax. Then we use a directive called ngModel and assign this property. To use ngModel you need to import FormModules.

<input type=”text” [(ngModel)] = “searchValue”>

searchValue is a property in the component/

String Interpolation:

Used to bind data from component class to view template. I.E. data flows from component to view. It is used to achieve one way data binding. Using string interpolation we can use property or methods of component class.

We use {{}} to perform string interpolation in html file. Inside {{}}, we can write any typescript code. We can also call typescript method using string interpolation. Example: <h1> {{ getName() }} </h2>

We use Angular to dynamically render data in the webpage. When we write static content in html, angular isn't need. Also, if the same static data is used in multiple parts, then it is prone to have issue like not rendering consistent data. To solve this, we can create a property in th component where we can define the data we want to render.

Inside component:

export class HeaderComponent{

    slogan: string = "Hello world."

}

In Html:

<h1>{{slogan}}</h1>

We using slogan property in the component to render the value in the view dynamically using data binding.

Property binding:

With property binding, we are binding the properties of html elements with the property or method ofc component class.

In component class,

export class HeaderComponent{

    source : string = "/assets/shopping.jpg';

}

In Html,

<img [src]="source" height="240" width="320">

Here data flows from component to view.

Event binding:

Binds webpage events to a components property or method. Using event binging we can pass data from view to component.

In view,

```
<div class="search-div">
    <span><b>Search:</b></span>
    <input type="text" (input)="changeSearchValue($event)">
</div>
```

The $event stores all the event data related to the particular event.

In Component,

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

  constructor() { }

  ngOnInit(): void {
  }

  searchValue :string = ""

  changeSearchValue(eventData : Event){
    this.searchValue = (<HTMLInputElement>eventData.target).value
  }
}
```

So whenever user enter the value in the search box, the input event will happen. And when the input methods happens, it'll call the changeSearchValue($event) method. Then we pass the event data to the method. From the event data, we get the user provided value.

Directives:

Directives are simply an instruction to the DOM. Used to tell Dom what to add to the webpage and what not. Also tell DOM how to render html elements.

Components:

Are kind of one such instruction in DOM. Using component we tell DOM what to add to the webpage. We can use directive as a css class or html element. But we mostly use directives as html attribute.

Custom directive:

```
@Directive({
  Selector: '[changeDivGreen]'
})
Export class ChangeDivGreen{
}
```

Types of directives:

1. **Structural directives:**

   Changes the view of a webpages by adding or removing DOM elements from a webpage.

   Note: Before using any structural derivative in Angular, we need to star with *. Example: *ngFor, *ngIf,*ngSwitch

2. **Attribute directives:**

   Used like an attribute on a existing webpage element to change its look and behaviour. Example: [ngStyle], [ngClass],[ngModel]

Built-in Directives:

1. **ngFor:**
   It is a structural directive. That means, ngFor manipulates the DOM by adding or removing elements from the DOM. It is used to repeat a portion of HTML template once per each item from an itterable list.

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

  constructor() { }

  ngOnInit(): void {
  }

  products = [
    {id:1, name:"Watch",price:"1200",color:'black',available:'available'},
    {id:2, name:"TV",price:"100000",color:'black',available:'not available'},
    {id:3, name:"Phone",price:"120000",color:'blue',available:'available'},
    {id:4, name:"Laptop",price:"87000",color:'midnight black',available:'not
available'},
    {id:5, name:"Book",price:"1450",color:'black',available:'available'}
  ]

}
```

we can assign multiple variable in ngFor like below.

```
<div *ngFor="let item of products; let i = index">
    <p>{{i}}</p>
    <div class="product-container">
        <div>
            <div class="name-container">
```

```
                <h6>{{item.name}}</h6>
            </div>
            <div class="detailContainer">
                <div class = "details">{{item.available}}</div>
                <div class = "details"><b>Price:</b>{{item.price}}</div>
                <div class = "details"><b>Color:</b>{{item.color}}</div>
            </div>

        </div>
        <div class="options">
            <button class="btn btn-success">Show Details</button>
            <button class="btn btn-warning">Buy Now</button>
        </div>
    </div>
</div>
```

2. **ngStyle:**

   It is an attribute directive. ngStyle changes the look and behavior of an HTML element. The ngStyle directive is used to set a CSS style dynamically for an HTML element based on a given typescript expression.

```
<div class="detailContainer">
                <div class = "details" [ngStyle]="{color: item.available
==='available' ? 'Green' : 'Red'}">{{item.available}}</div>
                <div class = "details"><b>Price:</b>{{item.price}}</div>
                <div class = "details"><b>Color:</b>{{item.color}}</div>
            </div>
```

   ngStyle is used as html attribute to change the color of available text. We put ngStyle within [] for property binding with the value in component. Here, text color is set up dynamically.

3. **ngIf:**

   It is an structural directive. ngIf is used to add or remove element from a webpage based on a given condition. If the condition assigned to ngIf returns true, it will add the element on which it is used to the webpage. Otherwise, if the condition returns false, it will remove that element from the webpage.

```
<div class="search-div">
    <span><b>Search:</b></span>
    <!-- <input type="text" (input)="changeSearchValue($event)"> -->
    <input type="text" [(ngModel)]="searchValue">
    <span *ngIf="searchValue != '' ">You searched for "{{searchValue}}"</span>
</div>
```

   Another Example:

```
 <div class="options">
```

```
            <button class="btn btn-success">Show Details</button>
            <button class="btn btn-warning" *ngIf="item.available ==='available';
else notifyMe">Buy Now</button>
            <ng-template #notifyMe>
                <button class="btn btn-danger">Notify me</button>
            </ng-template>
        </div>
```

Here we are using else to render different button when item is not available. For using else, we have to provide the name of ng-template to get render, which we have provided by using '#' sign.

4. **ngClass:**
   It is an attribute directive. The ngClass directive is used to add a CSS class dynamically to a webpage element.

```
@Component({
  selector: 'app-notification',
  // <div class="alert alert-success" [hidden]="displayNotifaication">
  template : `<div class="alert alert-success" [ngClass]="{fadeOut:
displayNotification}">
                  This website uses cookies
                  <div class="close">
                  <button class="btn" (click)="closeNotification()">X</button>
                </div>
              </div>
             `,
  styles : ["div{margin:10px 0px; padding: 10px 20px; text-align:center;}",
            "p{font-size:14px;}",
            ".close{floar:right;margin-top:-15px;}",
            ".fadeOut{visibility:hidden; opacity:0; transition: visibility 0s 2s,
opacity 2s linear;}"
          ]
})
export class NotificationComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

  displayNotification : boolean = false;

  closeNotification(){
    this.displayNotification = true;
  }
}
```

Child Component:

Custom event binding?

We can pass data from component class to view template and vice versa using property binding, string interpolation and event binding. We can also pass data from parent component to child component and vice versa. We user @Input and @Output decorator for that.

Custom Property binding:

We can pass data from parent component to child component using @Input decorator. We can also call it custom property binding because here we bind the custom properties of child component class with the property or method of parent component class.

Example:

Products is parent component and Filter is child component. Below shows the concept of @Input() where data is sent from parent to child component.

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

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

  constructor() { }

  ngOnInit(): void {
  }

  products = [
    {id:1, name:"Watch",price:"1200",color:'black',available:'available'},
    {id:2, name:"TV",price:"100000",color:'black',available:'not available'},
    {id:3, name:"Phone",price:"120000",color:'blue',available:'available'},
    {id:4, name:"Laptop",price:"87000",color:'midnight black',available:'not
available'},
    {id:5, name:"Book",price:"1450",color:'black',available:'available'}
  ]

  getTotalProducts(){
    return this.products.length
  }
  getTotalAvailableProducts(){
    return this.products.filter(item =>
      item.available === 'available').length
  }

  getTotalUnAvailableProducts(){
    return this.products.filter( item => item.available === 'not
available').length
  }
}
```

This is HTML file for Products.

```
<app-filter
    [total] = "getTotalProducts()"
    [available] = "getTotalAvailableProducts()"
    [unavailable]="getTotalUnAvailableProducts()"
    ></app-filter>

<div *ngFor="let item of products; let i = index">
    <!-- <p>{{i}}</p> -->
    <div class="product-container" [ngStyle]="{backgroundColor: item.available
==='available' ? '#90EE90' : '#FFCCCB'}">
        <div>
            <div class="name-container">
                <h6>{{item.name | uppercase}}</h6>
            </div>
            <div class="detailContainer">
                <div class = "details" [ngStyle]="{color: item.available
==='available' ? 'Green' : 'Red'}">{{item.available }} </div>
                <div class = "details"><b>Price:</b>{{item.price}}</div>
                <div class = "details"><b>Color:</b>{{item.color}}</div>
            </div>

        </div>
        <div class="options">
            <button class="btn btn-success">Show Details</button>
            <button class="btn btn-warning" *ngIf="item.available ==='available';
else notifyMe">Buy Now</button>
            <ng-template #notifyMe>
                <button class="btn btn-danger">Notify me</button>
            </ng-template>
        </div>
    </div>
```

```
</div>
```

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

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

  constructor() { }

  ngOnInit(): void {
  }

  @Input(‘total’) all : number = 0;
  @Input() available : number = 0;
  @Input() unavailable : number = 0;
}
```

This is HTML for filter.

```
<div class ='filter-container'>
    <span>Filer :</span>

    <input type="radio" name="filter" value="All">
    <span>{{'All Products('+all+')'}}</span>

    <input type="radio" name="filter" value="Available">
    <span>{{'Available Products('+available+')'}}</span>

    <input type="radio" name="filter" value="Unavailable">
    <span>{{'Unavailable Products('+unavailable+')'}}</span>
</div>
```

Here the data is being passed from parent component (products) to child component (filter). By using methods like getTotalProducts(), we send data to filter component, where it assigns the value of ‘all’ attribute. Similar process are done for available and unavailable. These are decorated with @Input decorator to signify the take the input from parent. We can also use alias to represent the attributes in child, but specifying the name inside @Input(). For this to work, we need to to provide the alias name while binding it in parent component.

Example:

Below shows the concept of @Output() where data is sent from child to parent component.

To create a new property event, we use EventEmitter from @angular/core.

We can not have two structural directives on the same html tag. To resolve this issue we can use ng-container.

For filter component:

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

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

  constructor() { }

  ngOnInit(): void {
  }

  @Input('total') all : number = 0;
  @Input() available : number = 0;
  @Input() unavailable : number = 0;

  selectedRadioButton : string = 'All';

  @Output()
  filterRadioButtonSelectionChanged : EventEmitter<string> = new
EventEmitter<string>();

  onRadioButtonSelectionChange(){
    this.filterRadioButtonSelectionChanged.emit(this.selectedRadioButton);
    console.log(this.selectedRadioButton);
  }
}
```

HTML

```
<div class ='filter-container'>
```

```
    <span>Filter :</span>



    <input type="radio" name="filter" value="All"
[(ngModel)]="selectedRadioButton"
    (change)="onRadioButtonSelectionChange()">
    <span>{{'All Products('+all+')'}}</span>


    <input type="radio" name="filter" value="Available"
[(ngModel)]="selectedRadioButton"
    (change)="onRadioButtonSelectionChange()">
    <span>{{'Available Products('+available+')'}}</span>

    <input type="radio" name="filter" value="Not available"
[(ngModel)]="selectedRadioButton"
    (change)="onRadioButtonSelectionChange()">
    <span>{{'Unavailable Products('+unavailable+')'}}</span>
</div>
```

Products Component:

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

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

  constructor() { }

  ngOnInit(): void {
  }

  products = [
    {id:1, name:"Watch",price:"1200",color:'black',available:'Available'},
    {id:2, name:"TV",price:"100000",color:'black',available:'Not available'},
    {id:3, name:"Phone",price:"120000",color:'blue',available:'Available'},
    {id:4, name:"Laptop",price:"87000",color:'midnight black',available:'Not
available'},
    {id:5, name:"Book",price:"1450",color:'black',available:'Available'}
  ]

  getTotalProducts(){
    return this.products.length
  }
  getTotalAvailableProducts(){
    return this.products.filter(item =>
      item.available === 'available').length
  }

  getTotalUnAvailableProducts(){
    return this.products.filter( item => item.available === 'Not
available').length
  }

  productCountRadioButton : string = 'All';

  onFilterRadioButtonChange(data: string){
    this.productCountRadioButton = data
    console.log(this.productCountRadioButton)
  }

}
```

HTML

```
<app-filter
    [total] = "getTotalProducts()"
    [available] = "getTotalAvailableProducts()"
    [unavailable]="getTotalUnAvailableProducts()"
    (filterRadioButtonSelectionChanged)="onFilterRadioButtonChange($event)"
    ></app-filter>

<ng-container *ngFor="let item of products; let i = index">
        <!-- <p>{{i}}</p> -->
        <div  class="product-container" *ngIf = "productCountRadioButton ===
'All' || productCountRadioButton === item.available">
            <div class="product-container" [ngStyle]="{backgroundColor:
item.available ==='Available' ? '#90EE90' : '#FFCCCB'}">
                <div>
                    <div class="name-container">
                     <h6>{{item.name | uppercase}}</h6>
                    </div>
                    <div class="detailContainer">
```

```
                        <div class = "details" [ngStyle]="{color: item.available
==='Available' ? 'Green' : 'Red'}">{{item.available}} </div>
                        <div class = "details"><b>Price:</b>{{item.price}}</div>
                        <div class = "details"><b>Color:</b>{{item.color}}</div>
                    </div>
                </div>
            </div>
            <div class="options">
                <button class="btn btn-success">Show Details</button>
                <button class="btn btn-warning" *ngIf="item.available
==='Available'; else notifyMe">Buy Now</button>
                <ng-template #notifyMe>
                    <button class="btn btn-danger">Notify me</button>
                </ng-template>
            </div>
        </div>
</ng-container>
```

Here, filter is a child component and product is parent component.

In the product html, we are using app-filter tag to dynamically render the filter html. The filter html renders a radio button. Depending on the radio button clicked, the data is passed to product component, where it renders only those products specified with the options selected in the radio button. In filter html, event binding is used to render total number of items of that category.

```
    <input type="radio" name="filter" value="Not available"
[(ngModel)]="selectedRadioButton"
    (change)="onRadioButtonSelectionChange()">
    <span>{{'Unavailable Products('+unavailable+')'}}</span>
```

Here, when the unavailable button is selected, the value of “Not available” is sent to the parent(product). In the filter component, we have specified an @Output decorator for filterRadioBUttonSelectionChanged variable. It sends a EventEmitter as it returns a data after the event has occurred.

In products html, property binding is used to send data from products to filter to render the number of items in each category. Event binding is used to receive data from the child(filter).

```
<app-filter
    [total] = "getTotalProducts()"
    [available] = "getTotalAvailableProducts()"
    [unavailable]="getTotalUnAvailableProducts()"
    (filterRadioButtonSelectionChanged)="onFilterRadioButtonChange($event)"
    ></app-filter>
```

That value is then used to set the property in product component.

```
 productCountRadioButton : string = 'All';

  onFilterRadioButtonChange(data: string){
    this.productCountRadioButton = data
    console.log(this.productCountRadioButton)
  }
```

Also here, the logic is trying to use both ngFor and ngIf, so we have to use ng-container to be able to use it.

```
<ng-container *ngFor="let item of products; let i = index">
        <!-- <p>{{i}}</p> -->
        <div  class="product-container" *ngIf = "productCountRadioButton === 
'All' || productCountRadioButton === item.available">
```

Template reference variable:

It is a reference to any DOM element, component or a directive in the template. We use # sign to create a template reference variable. Template variables help you use data from one part of a template in another part of the template. Use template variables to perform tasks such as respond to user input or finely tune your application's forms.

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

@Component({
  selector: 'app-temp-ref-demo',
  templateUrl: './temp-ref-demo.component.html',
  styleUrls: ['./temp-ref-demo.component.scss']
})
export class TempRefDemoComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

  sayHello(inputEl: HTMLInputElement){
    alert('Hello'+inputEl.value)
  }
}
```

IN temp-ref-demo html

```
<input type="text" #myVariable (keyup)="0">
<button (click)="sayHello(myVariable)"> Say Hello</button>
<p>{{myVariable.value}}</p>
```

IN container HTML.

```
<h2>{{message}}</h2>
<img [src]="durantImage" height=”240” width=”320”/>

<app-temp-ref-demo></app-temp-ref-demo>
```

Here, the #myVariable is made a template reference variable. On the click of the button it sends the event where the value of the input is sent to container. It stores ta local reference of the variable.

On a component:

```
import { Component, OnInit } from '@angular/core';
import { Players } from './players';

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

  constructor() { }

  ngOnInit(): void {
  }

  selectedPlayer:Players | null = null;

  player: Players[] = [
    {
      playerNo:1, name : "Jalckson",address:"ktm"
    },
    {
      playerNo:2, name : "denis",address:"pok"
    },
    {
      playerNo:1, name : "robman",address:"ktm"
    }
  ]
}
```

```html

<h1>List of Players</h1>

<table id="player-table">
    <thead>
        <tr>
            <th>Player No</th>
            <th>Name</th>
            <th>Address</th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor = "let p of player">
            <td>{{p.playerNo}}</td>
            <td>{{p.name}}</td>
            <td>{{p.address}}</td>
            <button (click)="selectedPlayer = p">Select</button>
        </tr>
    </tbody>
</table>
```

When the button is clicked to select a player, the value of the player object is store in the selectedPlayer variable. In the component below, the players component is implemented, where the player is made a template reference variable on a component. So it stores the reference to that component. The ?.name is a null coalescing technique, used as there is null stored as default.

```html
<app-navbar></app-navbar>
<app-players #player></app-players>
<p>You have selected {{player.selectedPlayer?.name}}</p>
<app-container></app-container>
<app-notification></app-notification>
<app-products></app-products>
```

So we used a method to pass the html element to the component class by passing the local reference variable to that method as a argument.

@ViewChild Decorator:

Sometimes we might access the html element before calling the method. @ViewChild is used to access HTML elements, components or directives from a view template in our component class. In simple terms, we can access the view of a component using viewChild decorator.

@ViewChild decorator, assigns a property with a reference to an Html element or a component or a directive.

In html

```
<div>
    <label>DOB:</label>
    <input type="date" #dobInput (blur)="calculateAge()">
</div>
<div>
    <label>Age:</label>
    <input type="text" #ageInput>
</div>
```

In Component

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

  @ViewChild('dobInput') dateOfBirth : ElementRef | null = null;
  @ViewChild('ageInput') age !: ElementRef ;


  calculateAge(){
    console.log(this.dateOfBirth)
    console.log(this.age)
    let birthYear = new
Date(this.dateOfBirth?.nativeElement.value).getFullYear();
    let currentYear = new Date().getFullYear();

    let calAge = currentYear - birthYear
    console.log(calAge)
    this.age.nativeElement.value = calAge;
  }
}
```

@ViewChild is to access child elements/components from the parent component.

@Input or @Output are to access data bindings in the child component that were set inside the parent component's template.

Content Projection:

We project a content from our parent to child component using ngContent directive.

In child component:

```
<h2>Child Component</h2>
<ng-content></ng-content>
```

In parent component:

```
<h1>Parent Component</h1>
<app-child>
    <p>This is a paragraph</p>
</app-child>
```

When the child component.html is render, the paragraph element in the parent component.html will be projected in place of ng-content. This is called as content projection.

@ContentChild:

We use @ContentChild decorator to access HTML elements, components or directives from the view template of parent component into component class of child component.

```
<h1>Parent Component</h1>
<app-child>
    <p #paragraph>This is a paragraph</p>
</app-child>
```

A property decorated with @ContentChild or @ViewChild only gets initialized just before ngAfterContentInit lifecycle hook.

In app.html(parent):

```
<div class="container">

    <input type="text" #input>
    <button (click)="OnSubmit(input)">Submit</button>
    <app-demo [value]="inputText" *ngIf="destroy">
        <h4 #header4>This is projected content {{inputText}}</h4>
    </app-demo>
    <br><br>
    <button (click)="DestroyComponent()">destroy</button>
</div>
```

In demo component(child)

```
export class DemoComponent implements OnInit , OnChanges,
DoCheck,AfterContentInit,
AfterContentChecked,AfterViewInit,AfterViewChecked,OnDestroy{

  @Input() value :string ="demo"

  constructor() {
    console.log("Constructor called from demo component.")
    // console.log(this.value)
  }

  ngOnInit(): void {
    console.log("NgOnInit called from demo component.")
    // console.log(this.value)
    console.log(this.headerEl)
  }

  @ContentChild('header4') headerEl !: ElementRef

  ngAfterContentInit(){
    console.log("ngAfterContentInit called from demo component.")
    console.log(this.headerEl)

  }
```

The value in log from ngOnInit will be undefined. The value in log from ngAfterContentInit will be the the html elementRef.

View Encapsulation:

It is a concept or behavior in angular, where component CSS styles are encapsulated into the components view and do not effect the rest of the application.

For Example:

We have three components app, comp1 and comp2. All three have button element in there view. Comp1 and comp2 are child of app, that is there tags are present in the app html. We have defined css of app on app.css. This will be implemented only for the button of the app, the buttons of comp1 and comp2 won't be effected by app.css.

So, it is important to ensure css styles specified in one component doesn't override the css rules in another component. To achieve this we use View Encapsulation.

There are three types of view encapsulation:

1. ViewEncapsulation.None
2. ViewEncapsulation.Emulated
3. ViewEncapsulation.ShadowDOM

By default, Angular uses Emulated.

Examples:

For ViewEncapsulation.None

```
@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.scss'],
  encapsulation : ViewEncapsulation.None
})
export class ProductsComponent implements OnInit {
```

When using None, whatever css style we have written for the parent component(app) will be used for the child component(comp1,comp2) as well. In this method, no unique attributes will be used.

Example:

<div class="options"><button class="btn btn-success">Show Details</button><button class="btn btn-warning">Buy Now</button><!--bindings={

  "ng-reflect-ng-if-else": "[object Object]"

}--><!--container--></div>

For ViewEncapsulation.Emulated

This is the default method Angular uses. In this method, each html element has a unique attribute is used. These attributes are added by the Angular tool automatically.

Example:

<button _ngcontent-suu-c18=""> Say Hello</button>

<div _ngcontent-suu-c55="" class="options"><button _ngcontent-suu-c55="" class="btn btn-success">Show Details</button><button _ngcontent-suu-c55="" class="btn btn-warning">Buy Now</button><!--bindings={

  "ng-reflect-ng-if-else": "[object Object]"

}--><!--container--></div>

Using this unique attribute, Angular achieves Encapsulation in Emulated type.

For ViewEncapsulation. ShadowDOM

```
@Component({
 selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.scss'],
  encapsulation : ViewEncapsulation.ShadowDom
})
export class ProductsComponent implements OnInit {
```

When we define the css in styles.css, the css style will be applied globally. That means, it gets applied in all components and all the html elements. Unique attributes will be used.

When we use ShadowDom on a component, that component creates its own DOM. The browser keeps the ShadowDom separate from the main dom. And the rendering of the shadowDom and main DOM happens separately. That's why the feature, state and style of the shadowDOM remains private and does not get effected by the main DOM. This is how we achieve true encapsulation using shadowDOM.

ng-content:

It is used when we wan to insert the content dynamically inside the component that helps to increase component reusability. Using ng-content, we can pass content inside the component selector and when angular parses that content that appears at the place of ng-content.

Example:

In app.component.html:

```
<app-products>
    <h2>Hello from ngContext</h2>
</app-products>
```

In products.component.html:

```
<app-search (searchTextEvent)="onTextSearchEnter($event)"></app-search>

<app-filter
    [total] = "getTotalProducts()"
    [available] = "getTotalAvailableProducts()"
    [unavailable]="getTotalUnAvailableProducts()"
    (filterRadioButtonSelectionChanged)="onFilterRadioButtonChange($event)"
    ></app-filter>

<ng-content></ng-content>

<ng-container *ngFor="let item of products; let i = index">
```

In angular, it replaces the last ng-content with the actual content to be rendered. So if we have multiple ng-content in a single html, the last ng-content will only get replaced. The others (ng-context)will get ignored.

```
<ng-content></ng-content>
<ng-content></ng-content>
<ng-container *ngFor="let item of products; let i = index">
```

If want to pass multiple html elements, we can assign class name in the select attribute for ng-content to render the required element and the right time.

In app.component.html:

```
<app-products>
    <h2 class="titleH2">Hello h2 from ngContext</h2>
    <h4 class="titleH4">Hello h4 from ngContext</h4>

</app-products>
```

In products.component.html:

```
<ng-content select=".titleH2"></ng-content>
<h3>Hello h3 from product component</h3>
<ng-content select=".titleH4"></ng-content>
```

Angular Lifecycle Hooks:

The angular lifecycle hooks are the methods that angular invokes on the directives and components as it creates, changes and destroys them.

When the angular application starts, it first creates and renders the root component. Then, it creates and renders its childrens and their children. It forms a tree of components. Once angular loads the components, it starts the process of rendering the view. To do that, it needs to check the input properties, evaluate the data bindings and expressions, render the projected content, etc. Angular also removes a component from the DOM when its no longer needed. Angular also lets us know when these events happen using lifecycle hooks. For example:

- **ngOnInit** when Angular initializes the component for the first time.
- When a component's input property change, Angular invokes **ngOnChanges**.
- If the component is destroyed, Angular invokes **ngOnDestroy**.

Change detection is the mechanism by which angular keeps the template in sync with the component. Whenever an event happens, angular will run the change detection cycle. X`

Terms:

Projected content:

Projected content is that HTML content which replaces the <ng-content> directive in child component.

Child Component:

```
<ng-content select=".titleH2"></ng-content>
```

Parent Component:

```
<app-products>
    <h2 class="titleH2">Hello h2 from ngContext</h2>
    // This h2 element is the projected content.
</app-products>
```

Input bound properties:

These are those properties of a component class which is decorated with @Input() decorator.

Child component:

```
  @Input() message : string
```

Constructor of a Component:

Lifecycle of a component begins, when Angular creates the component class. First method that gets invoked is class constructor. Constructor is neither a life cycle hook nor it is specific to Angular. It's a JavaScript feature. It is a method which gets invoked, when a class is created. When a constructor is called, at that point, none of the components input properties are updated and available to use. Neither its child components are constructed. Projected contents are also not available. Once Angular instantiates the class, it kick-start the first change detection cycle of the component.

ngOnChanges:

It is executed right at the start, when a new component is created and it also gets executed whenever one of the bound input property changes. Angular invokes ngOnChanges lifecycle hook whenever any data bound input property of the component of directive changes. Input properties are those properties, which we define using the @Input decorator. It is one of the ways by which a parent component communicates with the child component. Initializing the Input properties is the first task that angular carries during the change detection cycle. And if it detects any change in input property, then it raises the ngOnChanges hook. It does so during every change detection cycle. This hook is not raised if change detection does not detect any changes. It is the only hook that takes arguments.

ngOnInit:

Angluar raises the ngOnInIt hook, after it creates the component and updates it input properties. This hook is fired only once and immediately after its creation(during the first change detection). This is a perfect place where you want to add any initialization logic for your component. Here you have access to every input property of the component. You can use them in http get requests to get the data frim the backend server or run some initialization logic, etc. But remember that, by the time ngOnInIt gets called, none of the child components or projected content are available at this point. Hence, any properties we decorate with @ViewChilad, @ViewChildren, @ContentCHild & @ContentChildren will not be available to use.

ngDoCheck:

Angular invokes ngDoCheck hook event during every change detection cycle. This hook is invoked even if there is no change in any of the properties. For example, ngDoCheck will run if you clicked a button on the webpage which does not change anything, but still its an event. Angular invokes ngDoCheck, after the ngOnChanges and ngOnInit hooks. You can use this hook to implement a custom change detection, whenever Angular fails to detect the changes made to input properties. ngDoCheck is also a great method to use, when you want to execute some code on every change detection cycle.

ngAfterContentInit:

It is called after the components projected content has been fully initialized. Angular also updates the properties decorated with the @ContentCHild and @ContentChildren before raising this hook. This hook is raised, even if there is no content to project. The content here refers to the external content injected from the parent component via Content Projection. The Angular Components can include the ng-content element, which acts as a placeholder for the content from the parent. Parent injects the content between the opening & closing selector element. Angular passes this content to the child component.

ngAfterContentChecked:

It is called during every change detection cycle after Angular finishes checking of components projected content. Angular also updates the properties decorated with the @ContentChild and @ContentChildren before raising this hook. Angular calls this hook even if there is no projected content in the compontent.

This hook is very similar to the ngAfterContentInIt hook. Both are called after the external content is initialized, checked & updated. Only difference is that ngAfterContentChecked is raised after every change detection cycle. While ngAfterContentInIt during the first change detection cycle.

ngAfterViewInit:

It is called after the components view and all its child views are fully initialized. Angular also updates the properties decorated with the @ViewChild & @ViewChildren properties before raising this hook. The View here refers to the view template of the current component and all its child components & directives. This hook is called during the first change detection cycle, where angular initializes the view for the first time. At this point all the lifecycle hook methods & change detection of all child components & directives are processed & component is completely ready. This is a component only hook.

ngAfterViewChecked:

this hook is fired after it checks & updates the components views and child views. This event is fired after the ngAfterViewInIt and after that during every change detection cycle. This hook is very similar to the ngAfterViewInIt hook. Both are caked after all the child components & directives are initialized and updated. Only difference is that ngAfterViewChecked is raised during every change detection cycle. This a component only hook.

ngOnDestroy:

If you destroy a component, for example when you placed ngIf on a component, and this ngIf then set to false, at that time, ngIf will remove that component from the DOM, at that time, ngOnDestroy is called. This method is the great place to do some cleanup work, because this is called right before the objects will be destroyed itself by angular. This is the correct place where you would like to Unsubscribe Observables and detach event handlers to avoid memory leaks.

The order of execution in lifecycle hooks:

OnChanges > OnInit > DoCheck > AfterContentInit > AfterContentChecked > **{Child Component} OnChanges > OnInit > DoCheck > AfterContentInit > AfterContentChecked > AfterViewInit > AfterViewChecked >** AfterViewInit > AfterViewChecked > ngDestroy

Examples:

In demo,

```
export class DemoComponent implements OnInit {

  @Input() value :string ="demo"

  constructor() {
    console.log("Constructor called from demo component.")
    console.log(this.value)
  }

  ngOnInit(): void {
    console.log("NgOnInit called from demo component.")
    console.log(this.value)

  }

}
```

In app html,

```
<div class="container">

    <input type="text" #input>
    <button (click)="OnSubmit(input)">Submit</button>
    <app-demo [value]="inputText"></app-demo>
</div>
```

When we use constructor, the value property logs the default value i.e demo. When we use ngOnInit, the value property is updated with the value of property of the inputText element in app.html. So, if we want to initialize any thing, it is best to do it in ngOnInit.

Custom Attribute Directive:

```

@Directive({
    selector : '[setBackground]' // wrapped in [] to signify it is being used as
attribute directive
})
export class SetBackgroundDirective implements OnInit{

    // private element : ElementRef;
    constructor(private element:ElementRef){
        //angular creates a private property named element for this class, and
then we can access the property
        this.element= element
    }

    ngOnInit(){
        this.element.nativeElement.style.backgroundColor = '#C8E6C9'
    }
}
```

```
<div class="container" setBackground>

    <input type="text" #input>
    <button (click)="OnSubmit(input)">Submit</button>
    <app-demo [value]="inputText" *ngIf="destroy">
        <h4 #header4>This is projected content {{inputText}}</h4>
    </app-demo>
    <br><br>
    <button (click)="DestroyComponent()">destroy</button>
</div>
```

Here we use our custom directive, to change the CSS of the html element.

The native element property contains the reference to the underlying DOM object which gives us direct access to the DOM, bypassing the angular. This is not advisable for following reasons:

- Angular keeps the component and the view in sync using templates, data binding and change detection, etc. All of them are bypassed when we update the DOM directly.
- DOM manipulation works only in browser. You will not be able to use the app in other platforms like in a web worker, in server or in a desktop, or in the mobile app, etc. where there is no browser.
- The DOM APIs do not sanitize the data. Hence it is possible to inject a script, thereby, opening our app an easy target for the XSS injection attack.

So we can use renderer to manipulate the DOM without accessing the DOM element directly. So we use Renderer2. So it provides a layer of abstraction between the DOM element and the component core.

```
import { Directive,ElementRef, OnInit, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective implements OnInit{

  constructor(private element:ElementRef, private renderer : Renderer2) {

  }

  ngOnInit(){
    this.renderer.setStyle(this.element.nativeElement,'backgroundColor','#F1948A'
)
  this.renderer.addClass(this.element.nativeElement,'container')
    this.renderer.setAttribute(this.element.nativeElement,'title','This is
example div')

  }
}
```

```html
<div appHighlight>
    <label>Age:</label>
    <input type="text" #ageInput>
</div>
```

@HostListener:

It listens to the DOM event on the host element and it reacts to the event by executing an event handler method.

In app.html

```html
<div appHighlight appHover>
    <label>Age:</label>
    <input type="text" #ageInput>
</div>


<div class="container" setBackground appHover>

    <input type="text" #input>
    <button (click)="OnSubmit(input)">Submit</button>
    <app-demo [value]="inputText" *ngIf="destroy">
        <h4 #header4>This is projected content {{inputText}}</h4>
    </app-demo>
    <br><br>
    <button (click)="DestroyComponent()">destroy</button>
</div>
```

These two div elements are host elements as we have used the appHover directive in them. And appHover has the HostListener.

In hover directive:

```typescript
import { Directive, ElementRef, HostListener, OnInit, Renderer2 } from
'@angular/core';

@Directive({
  selector: '[appHover]'
})
```

```
export class HoverDirective implements OnInit{

  constructor(private element : ElementRef, private renderer : Renderer2 ) { }

  ngOnInit(): void {

  }

  //event handler method
  @HostListener('mouseenter') onMouseHover(){
    this.renderer.setStyle(this.element.nativeElement,'margin','5px 10px')
    this.renderer.setStyle(this.element.nativeElement,'padding','30px 30px')
    this.renderer.setStyle(this.element.nativeElement,'transition','0.5s')
  }

  @HostListener('mouseleave') onMouseOut(){
    this.renderer.setStyle(this.element.nativeElement,'margin','10px 20px')
    this.renderer.setStyle(this.element.nativeElement,'padding','10px 20px')
    this.renderer.setStyle(this.element.nativeElement,'transition','0.5s')
  }
}
```

The onMouseHover() and onMouseOut are eventHandlers. The handle 'mouseenter 'and 'mouseleave' event respectively. So the @Hostlistener will be listening to the 'mouseenter' event, and when this event happens in the host element, it will execute the onMouseHover event handler method.

@HostBinding:

Using host binding, we are binding the property of the directive class with the property of html element. In this way we can set the value of any html element.

Example for component:

```
<div class="container" appBetterhighlight appHover>
    <p>Learn about host binding</p>
</div>
```

```
@Directive({
  selector: '[appBetterhighlight]'
})
export class BetterhighlightDirective {

  constructor(private element: ElementRef, private renderer : Renderer2) { }
  @HostBinding(‘style.backgroundColor’) background : string = ‘transparent’
  @HostBinding(‘style.border’) border: string = ‘none’

  @HostListener(‘mouseenter’) onMouseEnter(){
    this.background = ‘pink’;
    this.border = ‘red 2px solid’;
  }
  @HostListener(‘mouseleave’) onMouseLeave(){
    this.background =’transparent’;
    this.border = ‘none’;
  }

}
```

Binding for directives:

We send data from component to directive. We use @Input() for the variable in the directive.

```
<div class="container" appBetterhighlight appHover [defaultColor]="'yellow'"
[Betterhighlight] = "'orange'" >
    <p>Learn about host binding</p>
</div>
```

```
export class BetterhighlightDirective implements OnInit{

  constructor(private element: ElementRef, private renderer : Renderer2) { }

  @Input() defaultColor: string = 'transparent'
  @Input('Betterhighlight') highlightColor: string = 'pink'

  @Input() title :string = 'This is title';

  @HostBinding('style.backgroundColor') background : string = this.defaultColor
  @HostBinding('style.border') border: string = 'none'

  @HostListener('mouseenter') onMouseEnter(){
    this.background = this.highlightColor;
    this.border = 'red 2px solid';
  }
  @HostListener('mouseleave') onMouseLeave(){
    this.background = this.defaultColor;
    this.border = 'none';
  }

  ngOnInit(){
    this.background = this.defaultColor
  }

}
```

Here, the variables in directive, defaultColor is used in controllers html , and is used as attribute binding. The highlightcolor variable is given an alias name, Betterhighlight. This Betterhighlight is used in component.html to bind it with template.

Angular Forms:

Template driven form approach:

The easiest way to build the angular forms. Logic of the form is placed in the template. It allows to create sophisticated looking forms easily without writing any JS code. Simple basic form based on html.

Model-driven form approach (Reactive forms):

The logic of the form is defined in the component as an object. The model-driven approach has more benefits as it makes the testing of the component easier. The representation of the form is created in the component class where form fields are created as properties of our component class. This form model is then bound to the HTML elements using the special markups. This is easier to test. Complex form with more control, structure of form is define in typescript class.

Strictly typed reactive forms:

This can detect type error at compile time. It is possible starting from Angular 14. You can create a typed form, just by declaring and initializing the form variable together, which should suffice in most of the situations. Angular infers the type of the form, from the initialization code. You also have the option to create a custom type and assign it to Form Variable.

Building blocks of Angular Forms:

1. **Form Control**
   It represents a single input field in an Angular form.
   The FormControl is an object that encapsulates all this information related to the single input element. It Tracks the value and validation status of each of these control.The FormControl is just a class. A FormControl is created for each form field. We can refer them in our component class and inspect its properties and methods. You can use FormControl to set the value of the Form field, find the status of form field like (valid/invalid, pristine/dirty, touched/untouched) etc & add validation rules to it.
2. **Form Group**
   The FormGroup is a collection of Form controls It Tracks the value and validity state of a group of Form control instances. We pass object in a formGroup. Inside the object we specify the form controls as a key-value pair. It provides a wrapper around a collection of FormControl's It encapsulates all the information related to a group of form elements. It Tracks the value and validation status of each of these control. We can use it to check the validity of the elements. set

its values & listen for change events, add and run validations on the group, etc. We create a FormGroup to organize and manage the related elements.

3. **FormArray**
   It is a way to manage the collection of Form Controls in Angular. The controls can be a FormGroup, FormControl or another FormArray. We can group Form Controls in Angular forms in two ways:
   >> Using Form Group and
   >> Using FormArray
   The difference is how they implement it. In formGroup controls becomes a property of the FormGroup. Each control is represented as key-value pair. While in FormArray, the controls become part of an array.
   We pass array in a FormArray. We need to specify elements inside it.
   Advantage: Using formArray we can generate formControls dynamically.
4. **FormRecord**
   FormRecord is intended for situations where we want to add or remove controls dynamically with dynamic keys. FormRecord accepts one generic argument, which describes the type of the controls it contains. This will ensue type safety by restricting all controls in the FormRecord to have the same value type. It allows us to add new controls at runtime with dynamic keys and also keeps the benefit of the type system. Used from angular 14 for types forms.

Aaaa

Example: Template driven forms

```
<div class="form-container">
    <h2>User Information</h2>
    <form (ngSubmit) = 'onSubmit(myForm)' #myForm="ngForm">
        <div class="form-group">
            <label for="firstname">First Name:</label>
            <input type="text" id="firstname" name="firstname" required ngModel>
            <!-- ngMOdel tells this element is a control of the form.
                no need to assign value or add it within banana syntax -->
        </div>
        <div class="form-group">
            <label for="lastname">Last Name:</label>
            <input type="text" id="lastname" name="lastname" required ngModel>
        </div>
        <div class="form-group">
            <label for="email">Email:</label>
            <input type="email" id="email" name="email" required ngModel>
        </div>
        <div class="form-group">
            <label for="country">Country:</label>
            <select id="country" name="country" required ngModel>
                <option value="">Select a country</option>
                <option value="USA">United States</option>
                <option value="West Indies">West Indies</option>
                <option value="Nepal">Nepal</option>
                <option value="India">India</option>
                <option value="China">China</option>
            </select>
        </div>
        <div class="form-group horizontal-radio" >
            <label>Gender:</label><br>
            <input type="radio" id="male" name="gender" value="male" required
ngModel>
            <label for="male">Male</label>
            <input type="radio" id="female" name="gender" value="female" ngModel>
            <label for="female">Female</label>
            <input type="radio" id="other" name="gender" value="other" ngModel>
            <label for="other">Other</label>
        </div>
        <div class="form-group horizontal-checkbox">
            <label>Hobbies<i> (optional)</i>:</label><br>
            <input type="checkbox" id="sports" name="hobbies[]" value="sports"
ngModel>
            <label for="sports">Sports</label>
            <input type="checkbox" id="arts" name="hobbies[]" value="arts"
ngModel>
            <label for="arts">Arts</label>
            <input type="checkbox" id="yoga" name="hobbies[]" value="yoga"
ngModel>
            <label for="yoga">Yoga</label>
        </div>
        <button type="submit" class="submit-btn">Submit</button>
    </form>
</div>
```

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

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

  constructor() { }
  ngOnInit(): void {
  }

  onSubmit(form : NgForm){
    console.log(form);
  }
}
```

Here we are creating a local reference variable, and passing it as a variable from the method. #ngForm = ‘ngForm’. The # ngForm is the local reference variable and the ‘ngForm’ specifies that ngForm is handling the form. The ngModel at the end of input elements, signifies that angular tells the html element is a control of this form. So with this a property named firstname will be create in the controls object.

There is another way to do this using @ViewChild.

```
<form (ngSubmit) = 'onSubmit()' #myForm="ngForm">
```

```

  @ViewChild('myForm') form !: NgForm

  onSubmit(){
    console.log(this.form);
  }
```

Form States **True when?**

Pristine - The user has not modified the form control

Dirty - The user has modified the form control

Touched - The user has interacted with the form control, e.g., by clicking or focusing on it.

Untouched - The form control has not been interacted with by the user.

Valid - The form control's value meets the validation rules defined in the application.

Invalid - The form control's value does not meet the validation rules defined in the application.

Rendering CSS based on form states:

```
input.ng-touched.ng-invalid{
    border: red 1px solid;
}
```

```
        <div class="form-group">
            <label for="firstname">First Name:</label>
            <input type="text" id="firstname" name="firstname" required
[(ngModel)]="firstName" #fname='ngModel'>
            <!-- ngMOdel tells this element is a control of the form.
                no need to assign value or add it within banana syntax -->
            <div>
                <small *ngIf = ' fname.invalid && fname.touched'> First name is
required field</small>
            </div>
            <span *ngIf="fname.touched && fname.valid"> you entered
{{firstName}}</span>
        </div>
        <div class="form-group">
            <label for="lastname">Last Name:</label>
            <input type="text" id="lastname" name="lastname"  required
[(ngModel)]="lastName">
            <div>
                <span *ngIf="lastName"> you entered {{lastName}}</span>
            </div>
        </div>
```

Render data based on the form states:

```
<small *ngIf = ' fname.invalid && fname.touched'> First name is required
field</small>
```

Render data based on with string interpolation using form states:

```
<span *ngIf="fname.touched && fname.valid"> you entered {{firstName}}</span>
```

It can also be done as:

```
<span *ngIf="lastName"> you entered {{lastName}}</span>
```

Form Control in Template driven form:

Form Group is a collection of form controls and it tracks the value at the validity state of group of form control instances. We create form group to organize and manage the related elements.

```html
<div ngModelGroup="personDetails" #personalDetails="ngModelGroup">
            <div class="form-group">
                <label for="firstname">First Name:</label>
                <input type="text" id="firstname" name="firstname" required
[(ngModel)]="firstName" #fname='ngModel'>
                <!-- ngMOdel tells this element is a control of the form.
                    no need to assign value or add it within banana syntax -->
                <!-- <div>
                    <small *ngIf = ' fname.invalid && fname.touched'> First name
is required field</small>
                </div>
                <span *ngIf="fname.touched && fname.valid"> you entered
{{firstName}}</span> -->
            </div>
            <div class="form-group">
                <label for="lastname">Last Name:</label>
                <input type="text" id="lastname" name="lastname"  required
[(ngModel)]="lastName">
                <!-- <div>
                    <span *ngIf="lastName"> you entered {{lastName}}</span>
                </div> -->
            </div>
            <div class="form-group">
                <label for="email">Email:</label>
                <input type="email" id="email" name="email" email required
ngModel>
            </div>
            <div>
                <small *ngIf="personalDetails.invalid &&
personalDetails.touched"> Some of the field does not have valid value</small>
            </div>
        </div>
```

```html
<div ngModelGroup="personDetails">
            <div class="form-group">
                <label for="firstname">First Name:</label>
                <input type="text" id="firstname" name="firstname" required
[(ngModel)]="firstName" #fname='ngModel'>
                <!-- ngMOdel tells this element is a control of the form.
                    no need to assign value or add it within banana syntax -->
                <div>
                    <small *ngIf = ' fname.invalid && fname.touched'> First name
is required field</small>
                </div>
                <span *ngIf="fname.touched && fname.valid"> you entered
{{firstName}}</span>
            </div>
            <div class="form-group">
                <label for="lastname">Last Name:</label>
                <input type="text" id="lastname" name="lastname"  required
[(ngModel)]="lastName">
                <div>
                    <span *ngIf="lastName"> you entered {{lastName}}</span>
                </div>
            </div>
            <div class="form-group">
                <label for="email">Email:</label>
                <input type="email" id="email" name="email" email required
ngModel>
            </div>
        </div>
```

ngModelGroup directive is used to group multiple properties together, as a object.

```
1. value: Object
      1. country: "Nepal"
      2. gender: "male"
      3. hobbies[]: true
      4. personDetails:
            1. email: "dd@gmail.com"
            2. firstname: "dev"
            3. lastname: "kumar"
```

Advantage of grouping form controls together.

We can validate all the controls in a group together. No need to write validation for each form control.

```
<div ngModelGroup="personDetails" #personalDetails="ngModelGroup">
            <div class="form-group">
                <label for="firstname">First Name:</label>
                <input type="text" id="firstname" name="firstname" required
[(ngModel)]="firstName" #fname='ngModel'>
                <!-- ngMOdel tells this element is a control of the form.
                    no need to assign value or add it within banana syntax -->
                <!-- <div>
                    <small *ngIf = ' fname.invalid && fname.touched'> First name
is required field</small>
                </div>
                <span *ngIf="fname.touched && fname.valid"> you entered
{{firstName}}</span> -->
            </div>
            <div class="form-group">
```

```
                <label for="lastname">Last Name:</label>
                <input type="text" id="lastname" name="lastname"  required
[(ngModel)]="lastName">
                <!-- <div>
                    <span *ngIf="lastName"> you entered {{lastName}}</span>
                </div> -->
            </div>
            <div class="form-group">
                <label for="email">Email:</label>
                <input type="email" id="email" name="email" email required
ngModel>
            </div>
            <div>
                <small *ngIf="personalDetails.invalid &&
personalDetails.touched"> Some of the field does not have valid value</small>
            </div>
        </div>
```

SetValue and PatchValue in Template driven forms:

SetValue method should be used if all the form fields are to be filled.

PatchValue method is used when only select few fields are to be filled.

Reactive Forms:

They are forms where we define the structure of the form in the component class. i.e. we create the form model with Form Groups, Form controls and form arrays. We also define the validation rules in the component class. Then we bind it to the HTML form in the template. This is different from the template-driven forms, where we define the logic and controls in the HTML template.

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

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

  constructor() { }

  gender = [
    {id:'1',value:'Male'},
    {id:'2',value:'Female'},
    {id:'3',value:'Other'}
  ]

  ngOnInit(): void {
  }

  contactForm = new FormGroup({
    name : new FormControl('',[Validators.required,Validators.minLength(3)]),
    age : new FormControl(),
    gender : new FormControl('Male',[Validators.required]),
    isMarried : new FormControl('',[Validators.required]),
    country : new FormControl('Nepal',[Validators.required]),
    location : new FormGroup({
      city: new FormControl(),
      street: new FormControl(),
      pincode:new FormControl(),
    })
  })


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

When we create a reactive form parameter in component, it should be of the type FormGroup. Then we can add property of the form inside the FormGroup as an object. The properties should be on the type FormControl. To bind the form with the HTML form we need to add a [formGroup] in the HTML form and assign it the reactive form property defined in the component. We use formControlName directive to bind the input element in the HTML form with the properties in the formGroup defined in the controller and assign it the value of the property. Inside the parenthesis of FormControl, we can assign default values and we can assign validation in it to. If we have more than one validation, then we add them inside a list as shown above. Above, I have assigned gender to Male and country to Nepal by default. We can use ngSubmit method to submit the form. Here we don't need to use local reference variable or viewChild decorator to send the form to the controller, as we already have the form inside the reactive form property.

```
<div class="form-container">
    <h2>User Information</h2>
    <form [formGroup] = 'contactForm' (ngSubmit) = 'onSubmit()'>
```

```
            <div class="form-group">
                <label for="name">Name:</label>
                <input type="text" id="name" name="name" formControlName
="name"  >
            </div>
            <!-- <div *ngIf="contactForm.get('name')?.invalid &&
(contactForm.get('name')?.dirty || contactForm.get('name')?.touched)">
                <span style="color: red;"
*ngIf="contactForm.get('name')?.errors?.required">Name is required.</span>
                <span style="color: red;"
*ngIf="contactForm.get('name')?.errors?.minlength">Name should have a minimum of
3 characters.</span>
            </div> -->
            <div class="form-group">
                <label for="age">Age:</label>
                <input type="text" id="age" name="age"  formControlName="age"  >
            </div>

        <div class="form-group">
            <label for="country">Country:</label>
            <select id="country" name="country" formControlName="country">
                <option value="">Select a country</option>
                <option value="USA">United States</option>
                <option value="West Indies">West Indies</option>
                <option value="Nepal">Nepal</option>
                <option value="India">India</option>
                <option value="China">China</option>
            </select>
        </div>

        <div class="form-group horizontal-radio" >
            <label>Gender:</label><br>
            <span *ngFor="let gen of gender">
                <input type="radio" id="gen.id" name="gender"
value="{{gen.value}}" formControlName="gender" >
                <label for="{{gen.value}}">{{gen.value}}</label>
            </span>
        </div>


        <div class="form-group">
            <label for="isMarried">Married:</label>
            <input type="checkbox" id="isMarried" name="isMarried"
formControlName ="isMarried"  >
        </div>
        <div formGroupName="location">

            <div class="form-group">
                <label for="city">City</label>
                <input type="text" class="form-control" name="city"
formControlName="city" >
            </div>

            <div class="form-group">
                <label for="street">Street</label>
                <input type="text" class="form-control" name="street"
formControlName="street" >
            </div>

            <div class="form-group">
                <label for="pincode">Pin Code</label>
                <input type="text" class="form-control" name="pincode"
formControlName="pincode">
            </div>
        </div>

        <button type="submit" class="submit-btn"
[disabled]="!contactForm.valid">Submit</button>
    </form>
</div>
```

Using the formGroupName directive in html form, we can group element together in a form. They then share the same css styles.

Form Array:

In the example, the form Array has three formControl element. We then use formArrayName directive and assign it to value of the formArray defined in the component.

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

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

  constructor() { }
  gender = [
    {id:'1',value:'Male'},
    {id:'2',value:'Female'},
    {id:'3',value:'Other'}
  ]

  ngOnInit(): void {
  }

  contactForm = new FormGroup({
    name : new FormControl('',[Validators.required,Validators.minLength(3)]),
    age : new FormControl(),
    gender : new FormControl('Male',[Validators.required]),
    isMarried : new FormControl('',[Validators.required]),
    country : new FormControl('Nepal',[Validators.required]),
    skills : new FormArray([
      new FormControl(null,Validators.required)
    ]),
    location : new FormGroup({
      city: new FormControl(),
      street: new FormControl(),
      pincode:new FormControl(),
    })
  })


  get skills() {
    return this.contactForm.get('skills') as FormArray;
  }

  onSubmit(){
    console.log(this.contactForm);
  }
  addSkills(){
    this.skills.push(new FormControl(null,Validators.required))
  }
}
```

```
        <div formArrayName="skills">
            <!-- <ng-container *ngFor = "let skill of
contactForm.controls['skills']">
                <input type="text" placeholder="add skill">
            </ng-container> -->
            <ng-container *ngFor="let skill of skills.controls; let i = index">
                <input type="text" [formControlName]="i" placeholder="Skill {{ i
+ 1 }}">
            </ng-container>
            <button (click)="addSkills()" >Add skills</button>
        </div>
```

Form Validators:

Two types of validators:

Sync validators: runs validations and returns immediately. They either return a list of errors or null if no errors found.

Async validators: returns a Promise or Observable. They either return a list of errors or null if no errors are found.

Built-in validators:

The Angular ReactiveForms Module provides several Built-in validators out of the box. They are required, minlength, maxlength & pattern etc

We can disable validation of form from html as:

```
<form [formGroup]="contactForm" (ngSubmit)="onSubmit()" novalidate>
```

Custom Validator:

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

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

  constructor() { }

  gender = [
    {id:'1',value:'Male'},
    {id:'2',value:'Female'},
    {id:'3',value:'Other'}
  ]
  ngOnInit(): void {
  }

  contactForm = new FormGroup({
    name : new
FormControl('',[Validators.required,Validators.minLength(3),this.noSpaceAllowed])
,
    age : new FormControl(),
    gender : new FormControl('Male',[Validators.required]),
    isMarried : new FormControl('',[Validators.required]),
    country : new FormControl('Nepal',[Validators.required]),
    skills : new FormArray([
      new FormControl(null,Validators.required)
    ]),
    location : new FormGroup({
      city: new FormControl(),
      street: new FormControl(),
      pincode:new FormControl(),
    })
  })


  get skills() {
    return this.contactForm.get('skills') as FormArray;
  }

  onSubmit(){
    console.log(this.contactForm);
  }
  addSkills(){
    this.skills.push(new FormControl(null,Validators.required))
  }

  noSpaceAllowed(control: FormControl) : { [key: string]: boolean } | null{
    if(control.value != null && control.value.indexOf(' ') != -1){
      return {noSpaceAllowed : true}
    }
    return null
  }
}
```

Added custom validation named noSpaceAllowed. It is of FormControl type, and send an object of key-value pair is error occurs else returns null. Its definition is similar to other built-in validators.

Services:

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

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

Services are used for:

- Features that are independent of components such a logging services
- Share logic or data across components
- Encapsulate external interactions like data access

Advantages:

- Services are easier to test and debug.
- We can reuse the service at many places.
- With services, we can communicate across different components.

Service class doesn't require any decorator. It is a simple typescript class.

```
export class EnterService{

    onEnterClicked(title:string){
        alert("you're inside "+title)
    }
}
```

This service can be used in multiple component to perform the same action.

```
export class KathmanduComponent implements OnInit {

  title : string = 'kathmandu'
  constructor() { }

  ngOnInit(): void {
  }

  onEnter(){
    const enterService = new EnterService();
    enterService.onEnterClicked(this.title)
  }

}
```

```
export class LalitpurComponent implements OnInit {

  title :string = 'lalitpur'
  constructor() { }

  ngOnInit(): void {
  }

  onEnter(){
    const enterService = new EnterService();
    enterService.onEnterClicked(this.title)
  }
}
```

Dependency Injection:

It is used to achieve loose coupling. Dependency Injection (DI) is a technique in which a class receives its dependencies from external sources rather than creating them itself.

```
import { Component, OnInit } from '@angular/core';
import { EnterService } from 'src/Services/enter.service';

@Component({
  selector: 'app-kathmandu',
  templateUrl: './kathmandu.component.html',
  styleUrls: ['./kathmandu.component.scss'],
  providers: [EnterService]
})
export class KathmanduComponent implements OnInit {

  title : string = 'kathmandu'
  constructor(private enterService : EnterService) { }

  ngOnInit(): void {
  }

  onEnter(){
    this.enterService.onEnterClicked(this.title)
  }

}
```

Firstly, we create a constructor, where we define a private variable of the type EnterService. Then we add the providers property in the Component Decorator to specify the service being provided.

There are five main players in the Angular Dependency injection Framework.

- **Consumer**:
  The Consumer is the class (Component, Directive, or Service) that needs the Dependency. In the above example, the KathmanduComponent is the Consumer.
- **Dependency:**
  The Service that we want to in our consumer. In the above example the EnteService is the Dependency
- **Injection Token (DI Token):**
  The Injection Token (DI Token) uniquely identifies a Dependency. We use DI Token when we register dependency
- **Providers:**
  The Providers Maintain the list of Dependencies along with their Injection Token. It uses the Injection Token is to identify the Dependency.
- **Injector:**
  Injector holds the Providers and is responsible for resolving the dependencies and injecting the instance of the Dependency to the Consumer. The Injector uses Injection Token to search for Dependency in the Providers. It then creates an instance of the dependency and injects it into the consumer.

Angular Provides an instance of Injector & Provider to every component & directive in the application (Consumers). It also creates an Injector instance at the module level and also at the root of the application. Basically, it creates a Tree of Injectors with parent-child relationship.

The dependencies are registered with the Provider. This is done in the Providers metadata of the Injector.

We can also add the Services to Providers array of the @NgModule. Then they will be available for use in all the components & Services of the application. The ProductService in this case added to the Injector instance at the module level.

Injector reads the dependencies from the constructor of the KathmanduComponent. It then looks for that dependency in the provider. The Provider provides the instance and injector, then injects it into the consumer. If the instance of the Dependency already exists, then it will reuse it. This will make the dependency singleton.

Hierarchical Injection:

If we provide a service on one component, the angular framework will create and inject an instance of that service for that component and all its child component.

If we provide a service in the appModule, the same instance of the service will be available throughout the app i.e. in all components, directives and services.

When we provide a service on AppComponent, the service will be instantiated and injected for AppComponent and all its child ccomponent and their child component. All of them will receive the same instance of that service.

Note:

It we have providers for the same service in both the parent and child, then the child will override the instance of service created by the parent.

```
import { Component, OnInit } from '@angular/core';
import { UserService } from 'src/Services/user.service';

@Component({
  selector: 'app-add-user',
  templateUrl: './add-user.component.html',
  styleUrls: ['./add-user.component.scss'],

})
export class AddUserComponent implements OnInit {

  username:string =''
  status:string= ''
  constructor(private userService : UserService) { }

  ngOnInit(): void {
  }

  AddUser(){
    this.userService.AddNewUser(this.username,this.status);
  }
}
```

Injecting services into another services:

We need to add metadata where we want to inject something. Like @Component decorator. But in service class, they can't attach any metadata. For that we have a special type of metadata called @Injectable(). We use @Injectable() on the service we want to inject something (receiving service), not on the service to be injected(providing service). But it's good practice to write @Injectable() on top of all services in newer versions of Angular.

Create a constructor on the receiving service with parameter of the type of the service to be injected. Decorate the receive service with @Injectable(). And then provide the service.

For example:

The logger service is created. It instance is create at app.component constructor where the provider is given.

The user service is the one where we want to use the logger service. So we create a constructor and instantiate the loggerService. We also provide the userService class with a @Injectable() decorator. Then we are providing the loggerService from the app.component. Since the logger has been provided by the app.component it can be access by the userService to access it methods.

```
export class LoggerService{

    logMessage(name:string, status:string){
        console.log('A new user with username '+name+' wiht status '+status+ ' 
has been added.')
    }
}
```

```
import { Injectable } from "@angular/core"
import { LoggerService } from "./logger.service"

@Injectable()
export class UserService{

    constructor(private logger : LoggerService){

    }

    users =[
        {name: 'Ram',status:'active'},
        {name: 'Shyam',status:'active'},
        {name: 'Hari',status:'inactive'}
    ]

    AddNewUser(name:string,status:string){
        this.users.push({name: name, status: status})
        console.log(this.users)
        this.logger.logMessage(name,status)
    }
}
```

```
import { Component, ElementRef, OnInit, ViewChild } from '@angular/core';
import { LoggerService } from 'src/Services/logger.service';
import { UserService } from 'src/Services/user.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
  providers: [UserService,LoggerService]
})
export class AppComponent implements OnInit{

  title = 'shop';


  users : {name:string,status:string}[]=[]

  constructor(private userService: UserService,private loggerService :
LoggerService){

  }
  ngOnInit(): void {
    this.users = this.userService.users
  }
}
```

Component interaction with services:

When components have to parent child relationship.

In app component:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
  providers: [UserService,LoggerService]
})
export class AppComponent implements OnInit{

  title = 'shop';

  @ViewChild('dobInput') dateOfBirth : ElementRef | null = null;
  @ViewChild('ageInput') age !: ElementRef ;


  users : {name:string,status:string}[]=[]

  constructor(private userService: UserService,private loggerService :
LoggerService){

  }
  ngOnInit(): void {
    this.users = this.userService.users
  }
}
```

```
<div class="container">
    <div>
        <h3>All users</h3>
        <app-all-users></app-all-users>
    </div>
    <div>
        <app-user-detail></app-user-detail>
    </div>
</div>
```

In UserService:

```
import { Injectable } from "@angular/core"
import { LoggerService } from "./logger.service"
import { EventEmitter } from "@angular/core"

@Injectable()
export class UserService{

    constructor(private logger : LoggerService){

    }

    users =[
        {name: 'Ram',status:'active'},
        {name: 'Shyam',status:'active'},
        {name: 'Hari',status:'inactive'}
    ]

    userDetails = [
        {name:'John',job:'teacher',gender:'male',country:'nepal',age:35},
        {name:'ron',job:'wizard',gender:'male',country:'uk',age:40},
        {name:'kon',job:'photgrapher',gender:'female',country:'uk',age:30},
        {name:'son',job:'footballer',gender:'male',country:'korean',age:28},
    ]

    AddNewUser(name:string,status:string){
        this.users.push({name: name, status: status})
        console.log(this.users)
        this.logger.logMessage(name,status)
    }

    OnShowDetailsCLicked = new EventEmitter<{name:string, job:string, 
gender:string, country:string, age:number}>()

    ShowUserDetails(user : {name:string, job:string, gender:string, 
country:string, age:number}){
        this.OnShowDetailsCLicked.emit(user)
    }

}
```

In AllUserComponent:

```
import { Component, OnInit } from '@angular/core';
import { UserService } from 'src/Services/user.service';

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

  constructor(private userService : UserService) { }

  userDetails : {name:string, job:string, gender:string, country:string, 
age:number} [] = []

  ngOnInit(): void {
    this.userDetails = this.userService.userDetails
  }

  ShowDetails(user : {name:string, job:string, gender:string, country:string, 
age:number}){
    this.userService.ShowUserDetails(user)
  }
}
```

```
<div class="container" *ngFor="let user of userDetails">
    <div  class="user-container">
        <div class ="user-name">{{user.name}}</div>
        <div class="user-job">{{user.job}}</div>
        <button (click)="ShowDetails(user)">Details</button>
    </div>
</div>
```

In UserDetails component:

```
import { Component, OnInit } from '@angular/core';
import { UserService } from 'src/Services/user.service';

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

  constructor(private userService : UserService) { }

  user !: {name:string, job:string, gender:string, country:string, age:number}

  ngOnInit(): void {
    this.userService.OnShowDetailsCLicked.subscribe((data: {name:string,
job:string, gender:string, country:string, age:number}) =>{
        this.user = data
    })
  }



}
```

```
<div class="container" *ngIf = "user != undefined">
    <div class="user-detail">
        <p><b> Name: </b> {{user.name}}</p>
        <p><b> Job: </b> {{user.job}}</p>
        <p><b> Gender: </b> {{user.gender}}</p>
        <p><b> Age: </b> {{user.age}}</p>
        <p><b> Country: </b> {{user.country}}</p>

    </div>
</div>
```

Singleton Service:

A singleton service is a service for which only one instance exists in an app. In this tutorial, we will show how to create a singleton service in Angular when the service is in the root module, eagerly loaded module, or lazy loaded module.

There are two ways in which you can create a Singleton Service

Using the root option of the providedIn property. This works irrespective of your service is in an eager module or lazy loaded module. Using the providedIn is the preferred way as it makes the service tree shakeable.

```
@Injectable({
    providedIn: 'root'
  })
  export class AppService {
  ...
  }
```

The second option is to add it in the Providers array of @NgModule. If the NgModule is root module or eagerly loaded module, then the AppService is available as a Singleton service to the entire application. But if the NgModule is lazy-loaded module, then AppService is available only in that Lazy loaded module and not outside of it.

ProvidedIn root, any & platform in Angular:

The providedIn allow us to specify how Angular should provide the dependency in the service class itself instead of in the Angular Module. It also helps to make the service tree shakable i.e. remove the service from the final bundle if the app does not use it.

ProvidedIn root:

Use the ProvidedIn root option, when you want to register the application-level singleton service. The root option registers the service in the Root Module Injector of the Module Injector tree. This will make it available to the entire application. This is irrespective of whether the service is lazy loaded or eagerly loaded. If it is never used it will not be added in the final build (tree shaking).

Lazy loaded service:

Using ProvidedIn root adds it to the Root Module Injector and makes it application-wide singleton. Registering the service in a @NgModule will make it available in that Module only (Singleton within the Module Scope). Using both makes it singleton for the rest of the application, while it creates a separate instance for that Module.

ProvidedIn any:

Use ProvidedIn: any when you want every lazy-loaded module to get its own instance of the service. The eagerly loaded modules always share the instance provided by the Root Module Injector. Hence this will not have any effect on them.

```
@Injectable({
   providedIn: 'any'
})
export class SomeService{
}
```

ProvidedIn platform:

A special singleton platform injector shared by all applications on the page. The platform allows us to add the service to the Providers of the Platform Injector. If you recall, the Platform Injector is the parent of the Root Module Injector in the Module Injector tree. This is useful if you have multiple Angular Apps running on a single page. This is a useful option if you are using Angular Elements, where they can share a single instance of service between them.

```
@Injectable({
   providedIn: 'platform'
})
export class SomeService{
}
```

@Self, @SkipSelf, @Optional, @Host Decorators:

@Self, @SkipSelf, @Optional & @Host are Angular Decorators that configure how the DI Framework should resolve the dependencies. These decorators are called Resolution Modifiers because they modify the behavior of injectors.

When a component asks for Dependency, the DI Framework resolves it in two phases. In the first phase, it starts to look for the Dependency in the current component's ElementInjector. If it does not provide the Dependency, it will look in the Parent Components ElementInjector. The Request bubbles up until it finds an injector that provides the service or reaches the root ElementInjector. If ElementInjector does not satisfy the request, Angular looks for the Dependency in the ModuleInjector hierarchy. If Angular still doesn't find the provider, it throws an error.

@Self

The @Self decorator instructs Angular to look for the dependency only in the local injector. The local injector is the injector that is part of the current component or directive.

@SkipSelf

The @SkipSelf decorator instructs Angular to look for the dependency in the Parent Injector and upwards. It tells Angular not to look for the injector in the local injector, but start from the Parent. You can think of this decorator as the opposite of the @Self. Open the GrandChildComponent again. Add the SkipSelf instead of Self decorator.

@Optional

Optional marks the dependency as Optional. If the dependency is not found, then it returns null instead of throwing an error.

@Host

The definition is a little confusing. So let us try to simplify it a bit.

- The @Host property searches for the dependency inside the component's template only.
- It starts with the current Injector and continues to search in the Injector hierarchy until it reaches the host element of the current component.
- It does not search for the dependency in the Providers of the host element.
- But it does search in the ViewProviders of the host element.
- Module injector is never searched in the case of @Host flag

It looks somewhat similar to @Self. But @Self only checks in the Injector of the current component. But the @Host checks for the dependency in the current template.

ViewProviders:

ViewProviders are similar to Providers except that the dependencies that you define are visible only to its view children. They are not visible to the Content children.

ViewProviders defines the set of injectable services that are visible only to its view DOM children. These services are not visible to views projected via content projection.

We can insert a component (child component) inside another Angular component (Parent Component). The View from that child component is called a View Child (or View Children's) of the Parent Component. Angular Components also allow us to insert (or Project) HTML Content provided by another component. We use the element <ng-content></ng-content> to create a placeholder for the external content. The view created from such a component is known as Content Child (or Content Children).

Providers Vs ViewProviders:

The Services declared in Components Providers are injected to both view children and content children.

But the Services declared in ViewProviders are injected only to view Children.

ViewProviders metadata is available only to Angular Components. Providers are available in NgModule, Component, Pipes, Directives, etc.

Observables:

We use observables to perform asynchronous operations and handle asynchronous data. We can handle asynchronous data in Angular using Promise or Observable.

Next: The observable invokes the next() callback, when it receives an value.

Error: when the error occurs it invokes the error() callback with details of error and subscriber finishes. After an error is emitted, and if there's any remaining value to be emitted then those value won't be emitted after the error, not even complete callback() value.

Completion: When the observable completes it invokes the complete() callback. The complete method doesn't take any parameter or argument. Any value emitted after the complete callback will not get emitted.

```
export class AppComponent implements OnInit{
  myObservable = new Observable((observer) => {
    console.log('Observable starts')
    setTimeout(()=>{observer.next("1")},1000)
    setTimeout(()=>{observer.next("2")},2000)
    setTimeout(()=>{observer.next("3")},3000)
    setTimeout(()=>{observer.error(new Error('Something went wrong! please try
again later'))},3000)
    setTimeout(()=>{observer.next("4")},4000)
    setTimeout(()=>{observer.next("5")},5000)
    setTimeout(() =>{observer.complete()},6000)

  })
  constructor(private userService: UserService,private loggerService :
LoggerService){

  }
  ngOnInit(): void {
    this.users = this.userService.users
    this.myObservable.subscribe((val) => {
      console.log(val)
    },(error) => {
      alert(error.message)
    },()=>{
      alert('Observable has complete emitting all values')
    }
  );
  }
}
```

Promise vs Observable:

A Promise promises us a data even when no one is using the data and provides us the data over a period of time. A promise provides us the data once the complete data is ready. Here, the data can be the actual data we requested for or an error message/object. It returns all the data at once. It provides us a single data. It is native to javascript.

An Observable is a function that convert ordinary stream of data into an observable stream of data. You can think of Observables as a wrapper around the ordinary stream of data. It sends data in chunks/ packets. An Observable streams the data. It provides us multiple values. It only provides you data if someone is using the data. It is not native to javascript. It is provided by RxJs.

RxJs:

RxJs(Reactive Extension library for JavaScript) is a JavaScript library, that allows us to work with asynchronous data stream. It has two main players:

Observable: it is the stream of data.

Observer: who is going to use the data. In order to use the data provided by observable, the observer must subscribe to the observable.

Methods of creating observable:

1. **Create method**

```
//using create method to create observable
  myObservable = Observable.create((observer) =>{
    setTimeout(()=>{observer.next("A")},1000)
    setTimeout(()=>{observer.next("B")},2000)
    setTimeout(()=>{observer.next("C")},3000)
    setTimeout(()=>{observer.error(new Error('Something went wrong! please try again later'))},4000)
    setTimeout(()=>{observer.next("D")},4000)
    setTimeout(()=>{observer.next("E")},5000)
    setTimeout(() =>{observer.complete()},6000)
  })

  constructor(private userService: UserService,private loggerService : LoggerService){

  }
  ngOnInit(): void {
    this.users = this.userService.users
    this.myObservable.subscribe((val) => {
      console.log(val)
    },(error) => {
      alert(error.message)
    },()=>{
      alert('Observable has complete emitting all values')
    }
  );
  }
```

2. **of method:**

```
 array1 = [1,2,6,7,8];
  array2 = ['A','B','C']

  myObservable = of(this.array1,this.array2)

  constructor(private userService: UserService,private loggerService :
LoggerService){

  }
  ngOnInit(): void {
    this.users = this.userService.users
    this.myObservable.subscribe((val) => {
      console.log(val)
    },(error) => {
      alert(error.message)
    },()=>{
      alert('Observable has complete emitting all values')
    }
  );
  }

```

After all the data has been emitted, the of() operator, which is imported from RxJs , also emitts a complete signal. Of operator emits the iterable as it is. It take any number of arguments.

3. **from operator:**
   From operator iterates over the iterable and then emits the value of that iterable one by one. It takes only one argument and that should also be iterable. In from method, we can also pass a promise, to convert that promise into an observable.

Operators of RxJs:

Operators in RxJs are functions that takes an observable as input and transform it into a new observable and return it. We use operators to manipulate the observable data stream.

Pipe method:

The **pipe** method of the Angular Observable is used to chain multiple operators together. We can use the pipe as a standalone method, which helps us to reuse it at multiple places or as an instance method. The pipe method accepts operators such as filter, map, as arguments. Each argument must be separated by a comma. The order of the operators is important because when a user subscribes to an observable, the pipe executes the operators in a sequence in which they are added. There are two ways we can use the pipe. One as an instance of observable and the other way is to use if as standalone method.

Map operators:

To perform any actions on the values emitted by the observable. Map operator applies a given project function to each value emitted by the source Observable and emits the resulting values as an Observable.

```
array1 = [1,2,6,7,8];
  array2 = ['A','B','C']

  // using of operator to create Observable
  // myObservable = of(this.array1,this.array2)

  //using from operator to create Observable
  myObservable = from(this.array1)

  transformedObs = this.myObservable.pipe(map((val) =>{
    return val * 5;
  }))

  constructor(private userService: UserService,private loggerService :
LoggerService){

  }
  ngOnInit(): void {
    this.users = this.userService.users
    this.transformedObs.subscribe((val) => {
      console.log(val)
    },(error) => {
      alert(error.message)
    },()=>{
      alert('Observable has complete emitting all values')
    }
  );
  }
```

Returns :

5

10

30

35

40

Filter operator:

Filter operator filter items from the source observable based on some condition and returns the filtered value as a new observable. To filter data based on certain conditions:

```
 array1 = [1,2,6,7,8];
  array2 = ['A','B','C']

  // using of operator to create Observable
  // myObservable = of(this.array1,this.array2)

  //using from operator to create Observable
  myObservable = from(this.array1)

  transformedObs = this.myObservable.pipe(map((val) =>{
    return val * 5;
  }))

  filteredObs = this.transformedObs.pipe(filter((val)=>{
    return val >= 30
  }))

  constructor(private userService: UserService,private loggerService :
LoggerService){

  }
  ngOnInit(): void {
    this.users = this.userService.users
    this.filteredObs.subscribe((val) => {
      console.log(val)
    },(error) => {
      alert(error.message)
    },()=>{
      alert('Observable has complete emitting all values')
    }
  );
  }
```

Return :

```
30

35

40
```

We can do this actions in the same statement:

```
  array1 = [1,2,6,7,8];
  array2 = ['A','B','C']

  // using of operator to create Observable
  // myObservable = of(this.array1,this.array2)

  //using from operator to create Observable
  myObservable = from(this.array1)

  transformedObs = this.myObservable.pipe(map((val) =>{
    return val * 5;
  }),
    filter((val) =>{
      return val >= 30
    })
  )

  constructor(private userService: UserService,private loggerService :
LoggerService){

  }
  ngOnInit(): void {
    this.users = this.userService.users
    this.transformedObs.subscribe((val) => {
      console.log(val)
    },(error) => {
      alert(error.message)
    },()=>{
      alert('Observable has complete emitting all values')
    }
  );
  }
```

Return:

30

35

40

Subjects in RxJs:

A subject is a special type of observable that allows value to be multicasted to many Observers. Subjects are like EventEmitters. We use subjects for cross component communications.

```
import { EventEmitter, Injectable } from "@angular/core";
import { Subject } from "rxjs";

@Injectable()
export class DataService{

    // dataEmitter = new EventEmitter<string>();

    dataEmitter = new Subject<string>()
    raiseDataEmitterEvent(data:string){
        this.dataEmitter.next(data)
    }
}
```

```
import { Component, OnInit } from '@angular/core';
import { DataService } from 'src/Services/data.service';

@Component({
  selector: 'app-comp1',
  templateUrl: './comp1.component.html',
  styleUrls: ['./comp1.component.scss']
})
export class Comp1Component implements OnInit {

  constructor(private dataService : DataService) { }

  ngOnInit(): void {
  }

  enteredText !: string


  onButtonCLick(){
    // console.log(this.enteredText)
    this.dataService.raiseDataEmitterEvent(this.enteredText)
  }
}
```

```
<div class="container">
    <input type="text" [(ngModel)]="enteredText">
    <button (click)="onButtonCLick()">Click</button>
</div>
```

```
import { Component, OnInit } from '@angular/core';
import { DataService } from 'src/Services/data.service';

@Component({
  selector: 'app-comp2',
  templateUrl: './comp2.component.html',
  styleUrls: ['./comp2.component.scss']
})
export class Comp2Component implements OnInit {

  constructor(private dataService : DataService) { }

  ngOnInit(): void {
    this.dataService.dataEmitter.subscribe( (val) => {
      this.inputText = val;
    })


  }


  inputText !: string


}
```

```
<div class="container">
    <h3> You entered: <i> {{inputText}}</i></h3>
</div>
```

Here data is passed from comp1 to comp2 using subject from RxJs. The data in comp1 is render in comp2. Comp1 and comp2 have no parent child relationship.

Unsubscribe to an Observable:

It is needed to unsubscribe any observable that can run for undefined amount of time, like the interval method of RxJs. The counterSub below is a observer of counterObs observable. We have to use the counterSub to unsubscribe.

```
counterObs = interval(1000)
  counterSub :any;
  ngOnInit(): void {
    this.users = this.userService.users
    this.counterSub = this.counterObs.subscribe((val) => {
      console.log(val)
    },(error) => {
      alert(error.message)
    },()=>{
      alert('Observable has complete emitting all values')
    }
  );

  }


  unsubscribe(){

    this.counterSub.unsubscribe();

  }
```

Http Requests:

Note: You don’t connect angular to a database directly. Because it is a frontend JS framework, and anyone can inspect the angular application and get the connections/credentials of the database if it is present. So we access the database by communicating with the APIs.

Setting Headers & query parameter in HTTP Requests:

We create a header using HttpHeaders class, and to the constructor of the HTTP headers class we pass an anonymous object i.e. {‘myHeaders’ : ‘procademy’ }. There we are using key value pair to store the value as shown in the example. Then we are passing the header through the post request by specifying a third parameter in the post method, where we specify the headers we want to send in that request.

```
  error = new Subject<string>();

constructor(private userService: UserService,private loggerService :
LoggerService,
     private dataService : DataService, private http : HttpClient){

  }

  ngOnInit(): void {
    this.users = this.userService.users
    this.counterSub = this.counterObs.subscribe((val) => {
      console.log(val)
    },(error) => {
      alert(error.message)
    },()=>{
      alert('Observable has complete emitting all values')
    }
  );
  }


  createProduct(products : {pName:string, desc:string, price:string}){
    const headers = new HttpHeaders({'myHeader':'procademy'})
    this.http.post<{name:string}>(
      'https://www.myWeb.com/products.json',
      products,{headers : headers})
      .subscribe((res) => {
      console.log(res)
    }, (err)=>{
      this.error.next(err.message)
    }
    )
  }
```

Another way:

Here, we create a instance of HttpHeaders(), use the set method to add new headers to the headers instance. The HttpHeaders() is immutable. The set instance will return a new instance of the HttpHeaders.

The set method returns a new instance after modifying the given header. If the header already exists then the headers value is replace by the given value in the returned object.

The append method will append a new value to the existing set of values for the header and return a new instance. It doesn’t check if the value exists or not.

```
fetchProduct(){
    const headers = new HttpHeaders().set('context-
type','application/json').set('Access-Control-Allow-Origin','*')
    return  this.http.get<{[name:string] : Product}>(
      'https://www.myWeb.com/products.json',
      {headers : headers})
      .pipe(map((res) =>{
        const products =[]
        for(const name in res){
          if(res.hasOwnProperty(name)){
            products.push({...res[name], id : name})
          }
        }
        return products
      }),catchError((err) => {
        return throwError(err)
      })
    )

  }
```

The append method:

```
 deleteProduct(id:string){

    let headers = new HttpHeaders()
    headers = headers.append('myHeader1','value1')
    headers = headers.append('myHeaders2','value2')
    this.http.delete('https://www.myWeb.com/products/'+id+'.json',{headers:header
s}).subscribe()
  }
```

Request query parameters:

The query parameter, print= pretty will display the data in json format in a pretty way. The query string parameters are immutable. When we call/ use the set method, it return a new instance of the HttpParams().

```
 fetchProduct(){
    const headers = new HttpHeaders().set('context-
type','application/json').set('Access-Control-Allow-Origin','*')

    const params = new HttpParams().set('print','pretty')

    return  this.http.get<{[name:string] : Product}>(
      'https://www.myWeb.com/products.json',
      {headers : headers, params : params})
      .pipe(map((res) =>{
        const products =[]
        for(const name in res){
          if(res.hasOwnProperty(name)){
            products.push({...res[name], id : name})
          }
        }
        return products
      }),catchError((err) => {
        return throwError(err)
      })
    )
```

Routing:

The Router is a separate module in Angular. It is in its own library package, @angular/router. The Angular Router provides the necessary service providers and directives for navigating through application views.

Using Angular Router you can

- Navigate to a specific view by typing a URL in the address bar
- Pass optional parameters (query parameters) to the View
- Bind the clickable elements to the View and load the view when the user performs application tasks
- Handles back and forward buttons of the browser
- Allows you to load the view dynamically
- Protect the routes from unauthorized users using Route Guards

Components of Angular Router

Router

An Angular Router is a service (Angular Router API) that enables navigation from one component to the next component as users perform application tasks like clicking on menus links, and buttons, or clicking on the back/forward button on the browser. We can access the router object and use its methods like navigate() or navigateByUrl(), to navigate to a route

Route

Route tells the Angular Router which view to display when a user clicks a link or pastes a URL into the browser address bar. Every Route consists of a path and a component it is mapped to. The Router object parses and builds the final URL using the Route

Routes

Routes is an array of Route objects our application supports

RouterOutlet

The outerOutlet is a directive (<router-outlet>) that serves as a placeholder, where the Router should display the view

RouterLink

The RouterLink is a directive that binds the HTML element to a Route. Clicking on the HTML element, which is bound to a RouterLink, will result in navigation to the Route. The RouterLink may contain parameters to be passed to the route's component.

RouterLinkActive

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 RouterLinks based on the current RouterState

ActivatedRoute

The ActivatedRoute is an object that represents the currently activated route associated with the loaded Component.

RouterState

The current state of the router includes a tree of the currently activated routes together with convenience methods for traversing the route tree.

RouteLink Parameters array

The Parameters or arguments to the Route. It is an array that you can bind to RouterLink directive or pass it as an argument to the Router.navigate method.

Location strategy in Angular:

HashLocation Strategy:

You can use the HashLocationStrategy by providing the useHash: true in an object as the second argument of the RouterModule.forRoot in the AppModule.

Example:

```
@NgModule({
declarations: [
   AppComponent,HomeComponent,ContactComponent,ProductComponent,ErrorComponent
],
imports: [
   BrowserModule,
   FormsModule,
   HttpModule,
   Hashlocationstrategy RouterModule.forRoot(appRoutes, { useHash: true }
],
providers: [ProductService],
bootstrap: [AppComponent]
})
```

Pros:

- Supported by all browsers.

Cons:

- Produces a URL like http://example.com/#foo
- Will not Support Server-Side Rendering

PathLocation strategy:

It is the default strategy in Angular application. To Configure the strategy, we need to add <base href> in the <head> section of the root page (index.html) of our application. i.e <base href="/">. The Browser uses this element to construct the relative URLs for static resources (images, CSS, scripts) contained in the document.

Pros:

- Produces a clear URL like http://example.com/foo
- Supports Server-Side Rendering

Cons:

- Older browser does not support
- Server Support needed for this to work

It is recommend to use the HTML 5 style (PathLocationStrategy) as your location strategy. Because it produces clean and SEO Friendly URLs that are easier for users to understand and remember. You can take advantage of the server-side rendering, which will make our application load faster, by rendering the pages in the server first before delivering them the client

Use the hash location strategy only if you have to support older browsers.

Pass route parameters:

{ path: 'product/:id', component: ProductDetailComponent }

Child Routes / Nested Routes:

Here the ProductDetailComponent is the sibling of the ProductComponent and not its child.

```
{ path: 'product', component: ProductComponent },
{ path: 'product/:id', component: ProductDetailComponent },
```

To make ProductDetailComponent as the child of the ProductComponent, we need to add the children key to the product route, which is an array of all child routes as shown below.

```

{ path: 'product', component: ProductComponent,
 children: [
   { path: 'detail/:id', component: ProductDetailComponent }
 ],
```

The child route definition is similar to the parent route definition. It has a path and component that Angular Router uses to render when the user navigates to this child route.

In the above example, the parent route path is ‘product’ and the child route is ‘detail/:id’ .This is will match the URL path “/product/detail/id”. When the user navigates to the “/product/detail/id”, the router will start to look for a match in the routes array

It starts off the first URL segment that is ‘product’ and finds the match in the path ‘product’ and instantiates the ProductComponent and displays it in the <router-outlet> directive of its parent component (which is AppComponent). The router then takes the remainder of the URL segment ‘detail/id’ and continues to search for the child routes of Product route. It will match it with the path ‘detail/:id’ and instantiates the ProductDetailComponent and renders it in the <router-outlet> directive present in the ProductComponent.

Query parameters vs Route parameters

The route parameters are required and Angular Router uses them to determine the route. They are part of the route definition.

For Example, when we define the route as shown below, the id is the route parameter.

```
{ path: 'product', component: ProductComponent }
{ path: 'product/:id', component: ProductDetailComponent }
```

The Angular Router will not navigate to the ProductDetailComponent route if the id is not provided. It will navigate to ProductComponent instead. If the product route is not defined, it will result in an error.

However, the query parameters are optional. The missing parameter does not stop angular from navigating to the route. The query parameters are added to the end of the URL Separated by Question Mark(?).

The best practice is to use Route Parameters to identify a specific resource or resources. For example a specific product or group of products.

```
//All Products
/products

//specific Product
/product/:id
```

Use query parameters to sort, filter, paginate, etc. For example sort products based on name, rating, etc. Filter based on price, color, etc.

```
//All Products sorted on rating
/products?sort=rating

//All Products sorted on rating with color=red
/products?sort=rating&color=red

//All Products sorted on rating with color=red & second page
/products?sort=rating&color=red&page=2
```

The Query parameters are not part of the route. Hence you do not define them in the routes array like route parameters. There are two ways in which you can pass a Query Parameter to Route.

- Using routerlink directive
- Using router.navigate method.
- Using the router.navigateByUrl method

Using routerLink Directive in Template:

We use the queryParams property of the routerlink directive to add the query parameter. We add this directive in the template file.

```
<a [routerLink]="['product']" [queryParams]="{ page:2 }">Page 2</a>
```

The router will construct the URL as

```
/product?page=2
```

You can pass multiple Query Parameters as shown below

```
<a [routerLink]="['products']"
                  [queryParams]="{ color:'blue' , sort:'name'}">Products</a>
```

The router will construct the url as :

```
/products?color=blue&sort=name
```

Using router.navigate method in Component:

You can also navigate programmatically using the navigate method of the Router service as shown below.

```
goTo() {
  this.router.navigate(
     ['/products'],
     { queryParams: { page: 2, sort:'name'} }
  );
}
```

The above code will navigate to the following URL

```
/products?page=2&sort=name
```

Using router.navigateByUrl method in the Component:

You can also use the navigateByUrl method of the router. navigateByUrl expects an absolute URL, Hence we need to build our query string programmatically. Also, the navigateByUrl method does not have queryParamsHandling option.

```
this.router.navigateByUrl('product?pageNum=2');
```

Reading query parameter:

Reading the Query parameters is similar to reading the Router Parameter. There are two ways by which you can retrieve the query parameters.

- Subscribing to the queryParamMap or queryParams observable.

- Using queryParamMap or queryParams property of the snapshot property.

Both the above are part of the ActivatedRoute service. Hence we need to inject it into our component class.

We usually retrieve the value of the Query parameter in the ngOninit life cycle hook, when the component is initialized. When the user navigates to the component again, and if the component is already loaded then, Angular does not create the new component but reuses the existing instance. In such circumstances, the ngOnInit method of the component is not called again.

If you are using the Snapshot to read the value of the Query parameters in ngOnInit, then you will be stuck with the values that you read when the component is loaded for the time. This is because Snapshot is not observable. Hence it will not notify you if the value changes.

By subscribing to the paramMap observable (or to the params observable), you will get a notification when the value changes. Hence you can retrieve the latest value of the parameter and update the component accordingly.

Passing data through route:

The parent can communicate with their child using @Input directive. A child can pass data to the Parent using the @Output & EventEmitter. The parent can use the @ViewChild to access the child component. In case of components are unrelated then we can use the Angular Services to Share data between them.

We can also share data between components using the route. Angular can pass data through the route in several ways.

- Using Route Parameter
- The Query Parameters or Query Strings
- Using URL Fragment
- Static data using the data property
- Dynamic data using the state object

RouterLinkActive:

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.

Example:

```
<li><a [routerLink]="['home']" routerLinkActive="active">Home</a></li>

<li><a [routerLink]="['product']" [routerLinkActive]="['active']">Product</a></li>
```

When the user navigates to any of the above routes, the Angular router adds the “active” class to the activated element. And when the user navigates away the class will be removed.

The Angular does this by watching the URL. Whenever the Url matches with the URL of the routerLink directive, it applies the classes defined in the RouterLinkActive directive. When it does not match it will be removed from the element. Using this we can apply different background or foreground color to our navigation links.

You can add multiple classes to the routerLinkActive directive as shown below.

```

<a routerLink="/user/bob" [routerLinkActive]="['class1', 'class2']">Bob</a>

<li><a [routerLink]="['home']" routerLinkActive="active  home">Home</a></li>
<li><a [routerLink]="['product']" [routerLinkActive]="['active','home']">Product</a></li>

```