12. ngIf Directive

→ Structural Directives can be used to add or remove HTML elements from the DOM

→ The 3 Common built in Structural directives are

ngIf
ngSwitch
ngFor

→ ngIf and ngSwitch can be used to conditionally render the HTML elements

→ ngFor is used to render the list of elements

```
<h2 *ngIf="true">
    CodeEvolution
</h2>
```

→ Let's Create a property in the class

⇒ displayName = false;

```
<h2 *ngIf = "displayName">
    Code Evolution.
</h2>
```

→ To implement else block for ngIf Directive, we use ng-template reference variable.

```
<h2 *ngIf = "displayName; else elseBlock">
    Code Evolution>
</h2>
<ng-template #elseBlock>
    <h2>
    4 Name is Hidden
    </h2>
</ng-template>
```

→ ng-Template can be used as container to add or remove other HTML elements from the DOM; whenever we are using ngIf.

→

We can also use ngIf apart from inline Template

<div *ngIf="displayName; then thenBlock; else elseBlock">

#thenBlock >

13. ngSwitch Directive

→ ngSwitch can be used to render HTML element instead of executing some logic.

→ We basically use ngSwitch to compare with lot of values.

In the Class

→ public color = "red"

In the Template

```
<div [ngSwitch] = "color" >

<div *ngSwitchCase = "red" > You Selected Red Color </div>

"

"

<div *ngSwitchDefault > Pick Again </div>

</div>
```

14. ngFor Directive

→ ngFor Directive can be used to render list of elements

Example Lets declare a array of variables in the class

```
Public colors = ["red", "blue", "green", "yellow"];
```

& in the HTML Template

```
<div *ngFor="let color of colors">
<h2>{{color}}</h2>
</div>
```

To Access the index during each iteration

```
<div *ngFor="let color of colors; index as i">
<h2>{{i}} {{color}}</h2>
</div>
```

My we also have, ~~first~~
last

15. Component Interaction

→ For the last few examples we are working on Test Component, Test Component is nested inside App Component.

→ Here App Component is the parent Component and Test Component is the child Component.

→ In Angular Applications we come to a scenario where Components need to interact with each other.

→ i.e The parent Component might send some data to the child Component, child Component might also send some data to the parent Component.

Components interact with each other using @Input and @Output decorator.

→ Using @Input decorator child Accept input from the parent, @Output decorator child when send out the events to indicate something.

example :— Send name from AppComponent to Test Component

p In the Test Component we will display name.

Uy we Send the Message from Child Component to ie Test Component to App Component

Sending data from ~~Child~~ Parent Component to ~~Parent~~ Child Component.

→

Example In the AppComponent class declare the variable

```
export class AppComponent
{
    title = 'app';
    public name = 'Vishnas';
}
```

p In the App Component.html, where we bind the Test Component ie

<app-test [ParentData] = "name"> </app-test>

→ In the Test Component.ts, declare the same variable that we are getting from Parent data ie (49)

Export TestComponent implements OnInit {

public parentData;

}

→ Here we need to provide information to the Test Component, that this is not a normal variable and the we are receiving from parent property.

The way we do is using @Input decorator

@Input() public parentData;

↓ In the Template

<h2> {{ "Hello " + parentData }} </h2>

need to

→ Sometimes we have different name than that is coming from parent data, Here we use Alias name

@Input ('parentData') public name;

Child Component to Parent Component :-

In the case of parent to child Component interaction, Parent Component HTML have child Component selector, so that we can easily bind the data from parent to child.

whereas in the Child Component HTML we dont have Parent Component Selector.

The way a child Component sends the data to the parent Component is using events.

Create an Event by using EventEmitter class, so that we can send an event to the parent. ie (51)

@Output() public childEvent = new EventEmitter();

To able to send the child Event to the parent we use @Output

→ lets fire the event on button click

<button> (click) = "fireEvent()" > </button>

fireEvent()

{ this.childEvent.emit("Hey Codevolution");

}

Here event is emitted in the child Component

→ To capture it in the parent Component, bind the custom event. ie

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

declare the variable message in the appcomponent.html & display it :

public String message :

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

16. Pipes

→ Pipes allows us ~~to~~ transform the data before displaying it to the view

Pipes on String Type

Public name = 'CodeEvolution';

<h2>{{ name | lowercase }}</h2>
| uppercase
| titlecase
| slice:3:5 → not including
↓ starts from index 3
(index start with 0

Public Person = {
"firstname": "John",
"lastname": "Doe"
}

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

Pipes on Numbers :-

→ Number takes one argument in a specified format

<h2> {{ 5.678 | number : '1.2-3' }} </h2>
(1 → Min no of integer digit)

<h2> {{ 5.678 | number : '3.4-5' }} </h2>

<h2> {{ 5.678 | number : '3.1-2' }} </h2>
(3 → Minimum no of decimal ; 2 → Max no of Decimal)

o/p : 5.678

o/p : 005.06780

o/p : 005.68

Percent pipe :- & Currency pipe :-

<h2> {{ 0.25 | percent }} </h2>

17 - Services

=> Why Services are required

Example In the app.component.html, we have an employee list Component i.e

<employee-list> </employee-list>

In the Employee-list Compohents.ts

we have an Array of employees

Export class employeeListComponent implements onInit {

public employees = [

{ "id": 1, "name": "Andrew", "age": 30 },

];

→ In the employee list Component. To render name of employees

```
<h2> Employee List </h2>
<ul *ngFor = "let employee of employees">
  <li> {{ employee.name }} </li>
</ul>
```

→ let create Another Component i.e employee-detail-Component & display all the details like name, age and Salary.

Here we have to copy the same employee list. & Code is duplicated.

→

→ Here we are voilating the principles ie

→ Do not Repeat yourself (DRY)

→ Single Responsibility principle ie

Component class should have only one Responsibility

ie To Control the view logic.

in our case it is also responsible for Creating the data ie employee-list.

→ The Solution is Service. ie

→ A Class with specific purpose

→ To share the data Across multiple Components.

→ Also to Implement business logic

→ To connect with database

→ we Implement Services by using Dependency Injection.

18. Dependency Injection :-

→ Can be implemented in 3 ways

1. Code without DI & its drawbacks
2. DI as design pattern
3. DI as a framework

1. Code without DI :-

```
Class Engine
{
  Constructor(new parameter){}
}

Class Type
{
  Type Constructor(){}
}
```

```
Class Car
{
  engine;
  type;
  Constructor()
  {
    this.engine = engine();
    this.type = new Tyre();
  }
```

Example

Here the draw back is, Engine has a parameter then Car doesn't able to create the object

→ To create an object of Car class we need to pass the parameter of engine type.

Why is the case with Tyres

→ Here the Code is not flexible because of the dependencies changed, the dependent has to be changed.

→ Also we get the same kind of object every time.

2 DI As Design pattern :-

DI is a Coding pattern in which class receives its dependencies from external resources sources rather than Creating them for itself.

The solution for the Code is DI Drawback

```
Class Car
{
    engine;
    tyres;

    public Car (engine, tyres)
    {
        this.engine = engine;
        this.tyre = tyre;
    }
```

→ The Problem with Dependency Injection as Design pattern is, as a Developer we have to Manually Create the Dependency objects & Dependent objects

→ If the number of object increases, Maintainance will be difficult for Developer.

The Solution is DI as a framework

3rd. DI As a framework

→ DI framework has something called 'injector', which registers all the dependencies

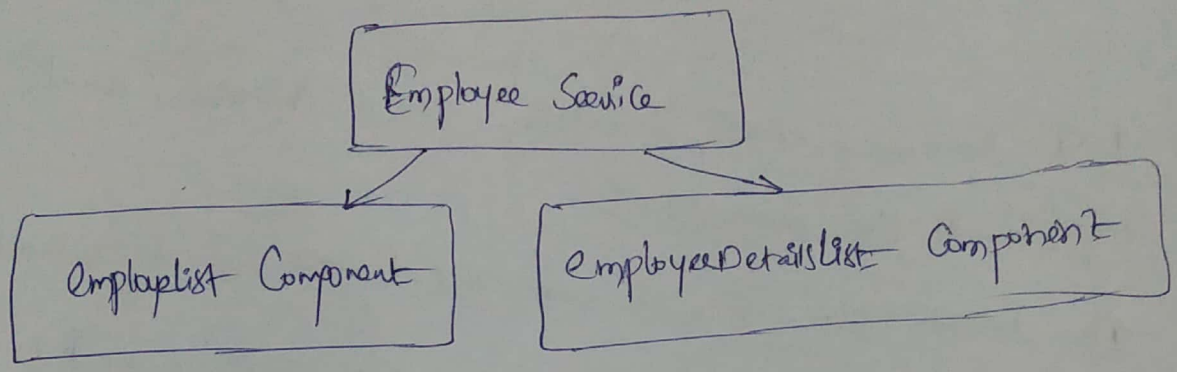
→ The injector acts like a container for all the dependencies.

→ DI framework manages all the dependencies & makes the developer job more simple.

→ In our example employee list component depends on service A, B, C and so on

↓ we register all the dependencies of Services with injector

→ ie when the employee list component is initialized all the dependencies will be initialized Automatically ie in our case employeelist.



* ① Define the Employee Service class
② Register with Injector
③ Declare as a Dependency in Emplist and EmpDetail.

19. Using a Service

→ Define the Employee Service

To generate the Service, in the Integrated Terminal, within the project folder, run the command

ng g s employee

↑ it generates employee.service.ts

→ The basic Template for Service is

```
@Injectable()
export class EmployeeService
{
  Constructor() { }
}
```

And this Service is responsible to provide employee data

```
getEmployees()
{

  return
    [{ "id": 1, "name": "Andrew", "age": 30}

    ];
}
```

To use this Service, in the Component declare as an array

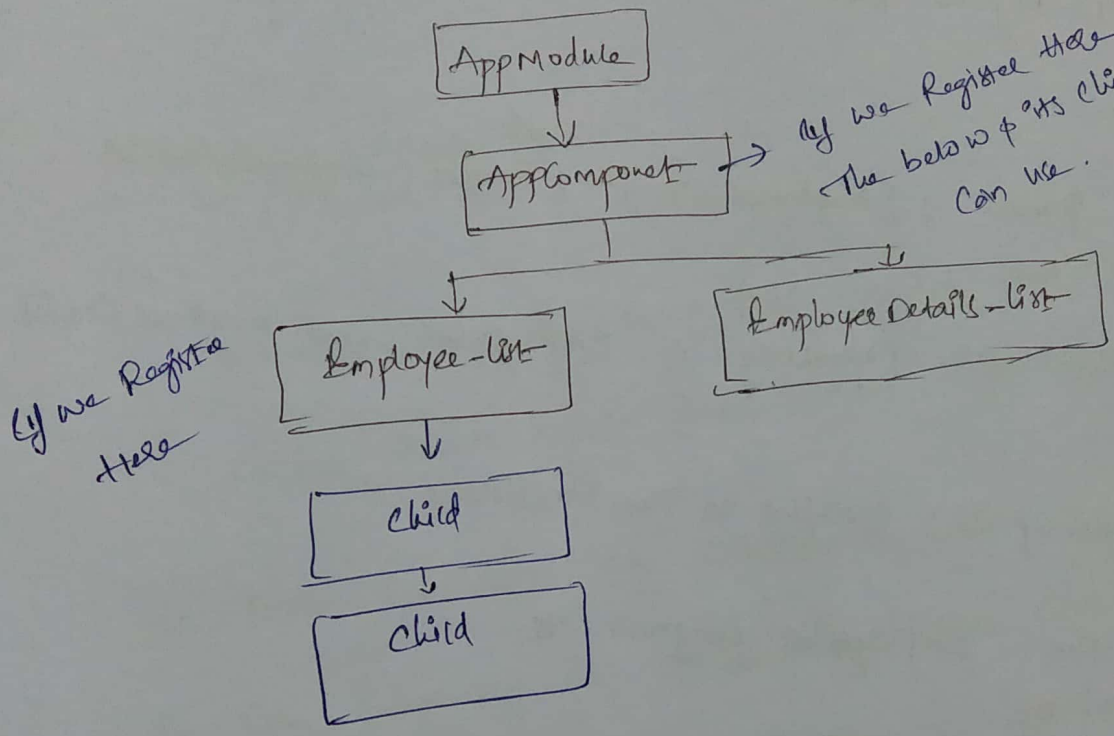
Employee-list.Component.ts

```
public employees = [];
```

4 Register the Service with Angular Injector

If we dont see Register the Service as like Regular Class in Angular.

→ There are multiple places where we can register the Service & is important because Angular has hierarchical Dependency injection System.



→ Each Module is a feature Area in our Application & it can grow.

→ It is better to register the Service in App Module.

→ If we register the Service in App module,

Components under the App Module can use the Service

→ To Register the Service, we use the providers Metadata.

ie

providers : [EmployeeService] also import employee Service.

If Declare as Dependency in Employee list and Employee Detail.

Dependency is Specified in the Constructor ie

in the EmployeeList-Component.ts

Constructor (private _employeeService : EmployeeService)

{

}

And we are going to make use of employeeService.

By in ngOnInit()

{

}

→ It is a lifecycle

ngOnInit() Method and get called when a Component has been initialized

{

this.employees = this.employeeService.getEmployees();

}

Injectable Decorator :-

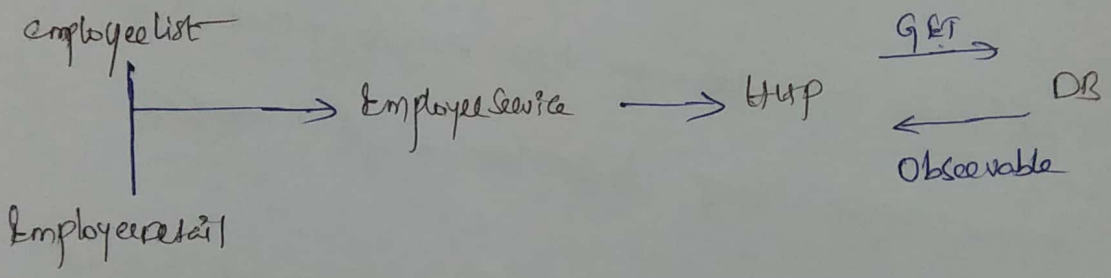
@Injectable() decorator tells Angular that this Service might itself have injected dependencies

we have not used any @Injector in Component because if the Component use @Component uses tells the Angular that it may or may not use @Injector in the Service we dont use any @Component.

20. HTTP and Observables

→ In a real world application we need to fetch the data in the service from web server / web service

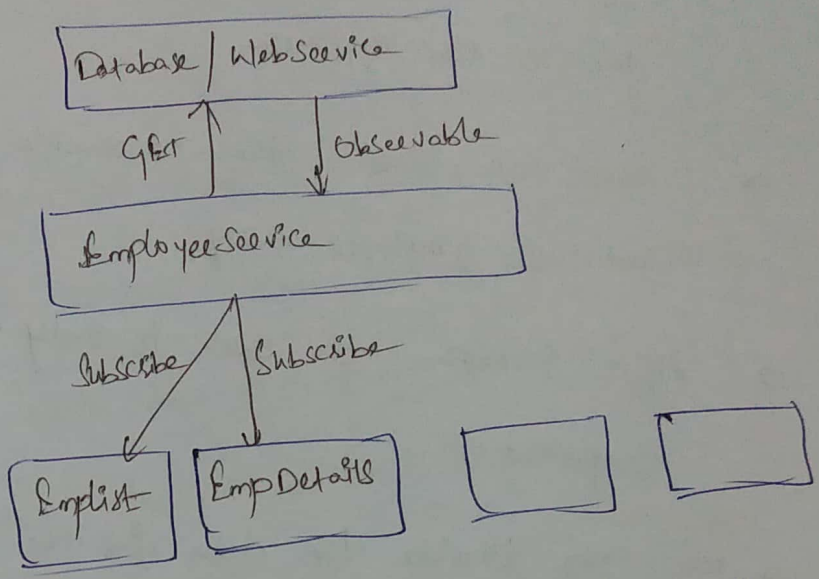
→



(the response we get back from the HTTP is Observable

→ Employee Component needs to cast an Observable into an Array of employees & then return the same to employeelist & employee Detail Component.

Observable :-



→ Observable is nothing but sequence of items that arrive asynchronously over time

→ HTTP call → Single item is called instead of sequence of but from items/calls.

→ Single item is nothing but Http Response.

→ An Observable is nothing but HttpResponse that arrives Asynchronously.

→ The observable is not in format that we can readily use in our Application.

→ Once we receive the observable we need to convert into an employee Array

→ After conversion, the data is ready to provide for the Component.

→ We only provide the data for the Components that Subscribe the Service.

21. Fetch Data using Http

Steps

1. Http Get Request from Employee Service
2. Receive the Observable and cast into an Employee Array
3. Subscribe to the observable from the Component
4. Assign the employee Array to a local Variable

→ Till Angular 4 we are using Http Module & from Angular 5 we are using HttpClient Module

In the App Module

'import HttpClient Module & add in the 'imports

→ By importing HttpClient Module. We are also registering the HttpService with Angulars 'injector.

→ To use the HttpService in Employee class, declare the dependency in the Employee Service Constructor.

```
private url:String = "                ";
Constructor (private http : HttpClient)
{
}

getEmployees()
{
return this.http.get(this.url);

}
```

→ Here get() returns an observable, for our application, Observable needs to cast into format that represents an Array of Employee.

For that lets create an Employee Interface:

→ export interface IEmployee

{

id: number;

name: String;

age: number

}

Then in the Service

getEmployees(): Observable<IEmployee[]>

{

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

}

// Subscribe the Observable from the Component

In the Components onInit()

ngonit() {

this.employeeService.getEmployees()

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

}

22. Http Error Handling

→ As we know that an Observable returns an Http Call & we need to handle the errors returned by Observable & here we make use of Catch operator.

→ We need to import Catch operator & Catch operator takes method name as an argument

```
getEmployees() : Observable<Employee[]>
{
    return this.http.get<Employee[]>(this.url)
                .Catch(this.errorHandler);
}

errorHandler(error: HttpErrorResponse)
{
}
```

→ It is important that User needs to be informed when any exception is thrown

→ if not user ends up staring at blank screen.

→ errorHandler here throws out the errorMessage and Any Component that has subscribed to the observable can make use of it and display the error message to the user. in the view.

```
errorHandler ( error : HttpErrorResponse)
{
  return Observable.throw (error.message || "Server Error");
}
```

// Part Two

Second part is to display the error message in the Component that has subscribed the observable.

In the employee list Component

```
ngOnInit(){
  this.employeeService.getEmployees()
  .subscribe(data => this.employees = data},
  error => this.errorMessage = error);
```

<h3>{{errMsg}}</h3>

is do the Property binding