**Angular Framework**

**It is framework based on HTML, CSS and JS and it is used to develop the web applications that run inside the browser.**

**Why to use the angular framework?**

**It gives our application a clear structure.**

**It has the re-usable codes.**

**Make our application more testable.**

**Angular follows component-based design for developing the web application**

**Components**

**A Component is group of data, html template, logic for the area of the screen which user see. E.g navbar, header, footer, rating system, course section everything is considered as the component.**

**Module**

**A Module is group of related component.**

**Dependency Injection**



**Ng serve**

**Ng Serve will use webpack tool to bundle all the files into single .js, .jpg, .png, .css,… files.**

**Property Binding Syntax (Only for known properties of elements)**

**@Component({**

**template = `<img [src]= “imageUrl/>”` 🡪 property binding**

**})**

**Export class CoursesComponent{**

**imageUrl = “”**

**}**

**Attribute Binding(Those attribute not known to elements)**

**Use attr.attribute-name**

**@Component({**

**template = `<table>**

**<tr>**

**<td [attr.colSpan]=""></td>🡪 attribute binding**

**</tr>**

**</table>`**

**})**

**Export class CoursesComponent{}**

**Class Binding**

**Use: class.class-name**

**@Component({**

**Template = `<button [class.activeClass]]= “isActive”> Save </button>`**

**})**

**Export class CourseComponent{**

**isActive: Boolean = true/false; 🡪 if true class will be applied or else not.**

**}**

**Style Binding**

**Use: [style.style-property]**

**@Component({**

**Template = `<h1 [style.backgroundColor]]= “red” > Save </h1>`**

**})**

**Export class CourseComponent{}**

**Event Binding**

**Use = (click)= “function-name()”**

**@Component({**

**Template = `<button (click)= “onSave()”> Save </button>`**

**})**

**Export class CourseComponent{**

**onSave(){}**

**}**

**Event Filtering**

**Filter key code from key events.**

@Component({

Template = `<input type="text"(keyup.enter)="onKeyDownEnter()"/>`

})

Export class CourseComponent{

onKeyDownEnter(){}

}

**Template Variable**

**Use: #variable-name in element to declare the template variable.**

**Example:**

**@Component({**

**Template: `<input #email (key.enter)= “onKeyEnter(email.value)”`>**

**})**

**Export class CourseComponent{**

**onKeyEnter(emailValue: string){**

**console.log(emailValue);**

**}**

**}**

**Two Way Binding (Component to View & View to Component)**

**Method 1:**

@Component({

Template = `

<input type="text" [value]='email'

(key.enter)="email= '$event.target.value';onKeyEnter()"/>

`

})

Export class CourseComponent{

Email: string;

}

**Method 2:**

**ngModel is the directive used to do two-way binding.**

**<input [(ngModel)]="email" (keyup.enter)="onKeyEnter()"/>**

**The change in input field will be detected in the variable email of the component class.**

**PIPES**

**It is used to format the data.**

**Build-in Pipes: Uppercase, LowerCase, Currency, Decimal, Percentage**

**Example:**

**@Component({**

**selector: 'courses',**

**template: `{{course.title | uppercase}}<br>**

**{{course.students | number}}**

**`**

**})**

**Course.title will pass through uppercase pipe and result will be in uppercase format.**

**Course.students value will be displayed in number format.**

**Decimal Format:**

**number: ‘no\_of\_digits.min\_decimal – max\_decimal’**

**Example:**

**Number: 4.975**

{{course.rating | number:'1.2-2'}}

**// means number displayed as 1 digit and minimum, maximum 2 decimal places i.e. 4.97**

**Currency Format:**

**Example:**

**Currency: ‘country code’: ‘apply symbol’: ‘decimal format’**

**{{ course.price | currency:'INR':true: '3.2-2'}}**

**Custom Pipe**

**Make a file pipe-name.pipe.ts**

**Exmaple: pipe-name is summary.**

**Import Pipe and PipeTransform**

**In Pipe decorator add field name and mention pipe-name.**

**Implement PipeTransform Interface and implement transform function.**

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

@Pipe({

    name: 'summary'

})

export class SummaryPipe implements PipeTransform{

    transform(value: any, limit?: number) {

        if(!value)

            return null;

            let actualLimit = (limit)?limit: 50;

        return value.substr(0, actualLimit) + "...";

    }

}

**Lastly within modules add new Pipe Class name in declarations array.**

**Directives**

**Directives are classes that additional behavior to our elements.**

**Directives are of two types:**

**Structural: Modify structure of the DOM**

**Attributes: Modify attributes of the elements.**

1. **Ng-if**

**Approach 1:**

<div \*ngIf="courses.length > 0">

    List Of Courses

</div>

<div \*ngIf="courses.length == 0">

    No Courses Yet

</div>

**Approach 2:**

<div \*ngIf="courses.length > 0; else noCourses">

    List Of Courses

</div>

<ng-template #noCourses>

    No Courses Yet

</ng-template>

**Approach 3:**

<div \*ngIf="courses.length > 0;then courseList else noCourses"></div>

<ng-template #courseList>

    List Of Courses

</ng-template>

<ng-template #noCourses>

    No Courses Yet

</ng-template>

**Hidden Property**

<div [hidden]="courses.length == 0">

    List Of Courses

</div>

<div [hidden]="courses.length > 0">

    No Courses Yet

</div>

**When we use hidden property the element exist in DOM but it is just hidden and while using ngIf the element is removed from DOM.**

**ngFor**

**Example 1:**

**html file**

<ul>

    <li \*ngFor="let course of courses">

       {{course.id}} - {{course.name}}

    </li>

</ul>

**Example 2:**

**html file: exported values aliased to local variables.**

<ul>

    <li \*ngFor="let course of courses; index as i; even as isEven">

       {{i}} - {{course.name}} - <span \*ngIf="isEven;">even</span>

    </li>

</ul>

**Class file**

courses = [

    {id: 1, name: 'Course1'},

    {id: 2, name: 'Course2'},

    {id: 3, name: 'Course3'}

  ];

The following exported values can be aliased to local variables:

* $implicit: T: The value of the individual items in the iterable ([ngForOf](https://angular.io/api/common/NgForOf)).
* [ngForOf](https://angular.io/api/common/NgForOf): [NgIterable](https://angular.io/api/core/NgIterable)<T>: The value of the iterable expression. Useful when the expression is more complex then a property access, for example when using the async pipe (userStreams | [async](https://angular.io/api/common/AsyncPipe)).
* index: number: The index of the current item in the iterable.
* count: number: The length of the iterable.
* first: boolean: True when the item is the first item in the iterable.
* last: boolean: True when the item is the last item in the iterable.
* even: boolean: True when the item has an even index in the iterable.
* odd: boolean: True when the item has an odd index in the iterable.

**ngFor to Change Detection**

<ul>

    <li \*ngFor="let course of courses">

       {{course.id}} - {{course.name}}

       <button class="btn btn-primary" (click)="onChange(course)">Update</button>

    </li>

</ul>

**Class file**

  onChange(course: courseType){

    course.name = "updated";

  }

**ngFor trackBy**

**Track By is used to increase the application performance.**

<ul>

    <li \*ngFor="let movie of movies">

      {{ movie.title }} - {{movie.director}}

    </li>

  </ul>

The Angular creates a li element for each movie. So if there are n number of movies, the angular inserts the n number of li nodes into the DOM

But the data will not remain constant. The user will add a new movie, delete a movie, sort the list in a different order, or simply refresh the movie from the back end. This will force the angular to render the template again.

The easiest way to achieve that is to remove the entire list and render the DOM again. But this is inefficient and if the list is large it is a very expensive process.

To avoid that the Angular uses the object identity to track the elements in the collection to the DOM nodes. Hence when you add an item or remove an item, the Angular will track it and update only the modified items in the DOM.

But if you refresh the entire list from the back end, it will replace the objects in the movie collection with the new objects. Even if the movies are the same, Angular will not be able to detect as the object references have changed. Hence it considers them new and renders them again after destroying the old ones.

We can solve this problem by providing a function to the trackBy option that returns a unique id for each item. The ngFor will use the unique id returned by the trackBy function to track the items. Hence even if we refresh the data from the back end, the unique id will remain the same and the list will not be rendered again.

**The Leading Astrik**

**When we use leading(\*) angular rewrites the element within the <ng-template>**

**Example:**

<div \*ngIf="courses.length>0; else noCourses">

    List of Courses

</div>

**Converts to**

<ng-template [ngIf]="courses.length > 0">

    <div>

        List Of courses

    </div>

</ng-template>

<ng-template [ngIf]="!(courses.length>0)">

    <div>

        No Courses

    </div>

</ng-template>

**ngClass**

**Adds or removes the css class on the html element**

**Class can be assigned using string, array, objects**

<some-element [[ngClass](https://angular.io/api/common/NgClass)]="'first second'">...</some-element> <some-element [[ngClass](https://angular.io/api/common/NgClass)]="['first', 'second']">...</some-element> <some-element [[ngClass](https://angular.io/api/common/NgClass)]="{'first': true, 'second': true, 'third': false}">...</some-element>

In Object if value is true then class will be applied or else not.

**Ng Style**

An attribute directive that updates styles for the containing HTML element. Sets one or more style properties, specified as colon-separated key-value pairs. The key is a style name, with an optional .<unit> suffix (such as 'top.px', 'font-style.em'). The value is an expression to be evaluated. The resulting non-null value, expressed in the given unit, is assigned to the given style property. If the result of evaluation is null, the corresponding style is removed.

Example:

[ngStyle]="{

        'backgroundColor':canSave? 'blue':'green',

        'color': canSave? 'white': 'black',

'width.px':80

    }"

**Creating Custom Decorators**

Hostlistners:

It is decorator that declares a DOM event to listen for, and provides a handler method to run when that event occurs.

@HostListener('focus') onFocus(){

    console.log("on focus");

  }

ElementRef:

Service defined in angular that gives access to DOM object.

Input:

Decorator that marks a class field as an input property and supplies configuration metadata. The input property is bound to a DOM property in the template. During change detection, Angular automatically updates the data property with the DOM property's value.

Custom Decorators:

Html file:

<input type="text" [appInputFormat]="'uppercase'">

Directive.ts

import { Directive, HostListener, ElementRef, Input } from '@angular/core';

@Directive({

  selector: '[appInputFormat]'

})

export class InputFormatDirective {

  @Input('appInputFormat') format: string = "";

  constructor(private el: ElementRef) {

   }

  @HostListener('blur') onBlue(){

    let value: string = this.el.nativeElement.value;

    if(this.format === "lowercase")

      this.el.nativeElement.value = value.toLowerCase();

    else

      this.el.nativeElement.value  = value.toUpperCase();

      console.log('on blur');

  }

}

**Template Driven Forms**

Template-driven forms:

Apply directive to input field

Good for simple forms

Simple Validation

Easier to create

Less Code

**NgForm (Directive)**

Selector-name: ngForm

Creates a top-level [FormGroup](https://angular.io/api/forms/FormGroup) instance and binds it to a form to track aggregate form value and validation status.

As soon as you import the [FormsModule](https://angular.io/api/forms/FormsModule), this directive becomes active by default on all <form> tags. You don't need to add a special selector.

You optionally export the directive into a local template variable using [ngForm](https://angular.io/api/forms/NgForm) as the key (ex: #myForm="[ngForm](https://angular.io/api/forms/NgForm)").

To register child controls with the form, use [NgModel](https://angular.io/api/forms/NgModel) with a name attribute. You may use [NgModelGroup](https://angular.io/api/forms/NgModelGroup) to create sub-groups within the form.

Example:

<form #employeeForm="ngForm">

    <div class="form-group">

        <label for="employeeId">Employee ID</label>

        <input type="text" class="form-control" id="employeeId"

        [(ngModel)]="employeeInput.id" name="employeeId"  required>

    </div>

    <p>

        {{employeeInput | json}}

    </p>

</form>

**Handling Error Message / Validation**

The template reference variable (#name) is set to "[ngModel](https://angular.io/api/forms/NgModel)" because that is the value of the [NgModel.exportAs](https://angular.io/api/core/Directive" \l "exportAs) property. This property tells Angular how to link a reference variable to a directive.

**Drop DownList**

If we have to pass object({}) as value to the option we must use [ngValue] directive rather than traditional value attribute

**ngModelGroup**

If we have complex form then we can create the sub-groups in the form using ngModelGroup and it will too create the Form Group object for that group.

ngForm has the submit property and ngModelForm doesn’t has it because ot doesn’t makes sense to submit a part of form

**Reactive Form Groups**

Form Group

firstname

lastname

FormControl

oup

FormControl

**Services**

Service is a component that holds business logic. Service component objects are created by Angular at runtime and injected in the other components (UI Component and Directive)

Write a class with method/methods and inside methods we can write any type of logic i.e BL, CrossCutting, REST API]

e.g class IncrementCounterService{

}

To register service in @NgModule providers:[ IncrementCounterService]. 🡪 Injector Service who creates the objects (only when any component asks for service/ any component is dependent on the service) and injects the object in the required components.

Service Component

Business Logic Service

HTTP Service/ Rest API

Cross Cutting Concerns (Common Code in every method),

Service

Where components will ask injector service to inject service object

Answer: Service objects are injected in constructor of component

**There is only one object created for the Injector Service.**

**Tree Shaking:**

**HTTP Services / REST API**

Get Method

To get the data from the server.

POST Method:

To create the data on the server

DELETE Method:

To delete the data on the server.

PUT Method

To update the data on the server

PATCH

To update only certain property of the object on the server.

**Handling Errors:**

Errors are of two types: Expected and Unexpected Error.

Expected Errors: Bad Request (400), Not Found (404)

Unexpected Errors: Server Down,

**Routing and Navigation**

Routing allows us to navigate from one part of application to another part of application or in terms of angular one part of view to another part of view in the page.

<router-outlet>

It is the directive which tells angular where to display view

Setting routes

The order of the routes matters.

In redirectTo you have to mention url and pathMatch means to match the full URL.

const appRoute: Routes = [

  {path:'', redirectTo: 'Home', pathMatch: 'full'},

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

  {path:'About', component: AboutComponent},

  {path:'Contact', component: ContactComponent},

  {path:'Courses', component: CoursesComponent}

]

{path:”\*\*” , component: ErrorComponent}

“\*\*” 🡪 this means wild card route. It matches every route. If we mention any route which is not existing then app will navigate to this route.

We should mention wild card route always at last.

**Absolute Path and Relative Path**

Absolute Path we have use (‘/’) before specifying the path and the mentioned path is appended after localhost/ root path.

Relative Path we do not specify (‘/’) and it appends the mentioned path to the existing path.

**Marking Active Link**

Use [routerLinkActive]=” ’class-name’, ‘class-name’, … ” to style active link

        <li class="nav-item" [routerLinkActive]="'active'">

          <a class="nav-link" routerLink="Home">Home</a>

        </li>

**Dynamic Routing**

{path:'Courses/Course/:id', component: CourseComponent},

We have to use (:parameter-name) for dynamic routing. The value id will be provided dynamically.

Course.ts

  constructor(private \_activatedRoute: ActivatedRoute,private \_service: CourseService){}

  ngOnInit(){

    this.\_courseId = this.\_activatedRoute.snapshot.paramMap.get('id');

    this.courses = this.\_service.courses.filter(ele => ele.id.toString() == this.\_courseId);

  }

Course.html

<div \*ngFor="let course of courses">

    Id: {{course.id}}

    Name: {{course.name}}

</div>

Here,

ActivatedRoute Service gives us current path and using it we can get dynamic parameter value.

**If value of our route parameter is going to change over the time it is better to use observable.**

  ngOnInit(){

    // this.\_courseId = this.\_activatedRoute.snapshot.paramMap.get('id');

    // this.courses = this.\_service.courses.filter(ele => ele.id.toString() == this.\_courseId);

    this.\_activatedRoute.paramMap.subscribe(

      (response)=>{

        this.\_courseId = response.get('id');

        this.courses = this.\_service.courses.filter(ele=>ele.id.toString()==this.\_courseId);

      }

    )

  }

**Navigate To page using function in Class**

Html file

<a (click)="navigateToHome()">Go To Home Page</a>

typescript

  constructor(private route: Router, private activatedRoute: ActivatedRoute){  }

  navigateToHome(){

    this.route.navigate(['Home'],{relativeTo: this.activatedRoute});

    //this.route.navigateByUrl('Home');

  }

**Query Parameter**

Query Parameter are the optional parameters which we pass to a route. These query parameters are added to end of the URL separated by (‘?’).

Route vs Query Parameter

Route parameters are defined in the path but query parameter are optional parameters.

**Child Component**

1. We can pass data from component class to view template using property binding, string interpolation and event binding.
2. We can also pass the data form parent component to child component and vice-versa using @input and @output decorators.

**@Input decorator**

To send the data from child component( component class) to the parent component(html template).

We declare the child component variables with @input decorator

Example:

Courses component 🡪 parent component

Filter component 🡪 child component

We can also write type-alias within the input decorator.

Child Component Class ( Filter component)

export class FilterComponent {

  constructor(){

  }

  ngOnInit(){

  }

  @Input('total') all:number = 0;

  @Input() free: number = 0;

  @Input() premium: number = 0;

}

Parent Component template ( Course component)

<div>

    <app-search></app-search>

    <app-filter

        [total]="getTotalCourses()"

        [free]="getTotalFreeCourses()"

        [premium]="getTotalPremiumCourses()"></app-filter>

</div>

**@Output Decorator**

Child Component

Ts file

  selectedRadioButtonValue: string  = "All";

  @Output()

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

  OnRadioButtonSelectionChanged(){

    this.filterRadioButtonSelectionChanged.emit(this.selectedRadioButtonValue);

    console.log(this.selectedRadioButtonValue);

  }

Html File

    <label>

        <input type="radio" name="filter" value="All"

        [(ngModel)]="selectedRadioButtonValue"

        (change)="OnRadioButtonSelectionChanged()"

        >

        <span>{{ 'All Courses('+ all +')' }}</span><br>

    </label>

Parent Component

Ts File

  courseCountRadioButton: string = "All";

  onFilterRadioButtonChanged(data: string){

    this.courseCountRadioButton = data;

    console.log(data);

  }

Html file

<app-filter> (filterRadioButtonSelectionChanged)="onFilterRadioButtonChanged($event)"></app-filter>

**“SPA” Application**

**Patterns to study:**

**Dependency Injection Pattern**

**Factory Pattern**

**SOLID Principles**