**Angular 18**

Go to console enter the command :-

**npm –v** (to check the version of node js )

To install the Angular CLI, open a terminal window and run the following command:

**npm install -g @angular/cli@17 ~ for version 17**

**npm install -g @angular/cli ~ for latest 18 or19**

next enter **ng v** = To get the Angular CLI version and other info.

* Open the folder where you have to create new project >
* Open command prompt on that folder and enter command
* Command for Creating new project :- **ng new < project-name >** > Press Enter key
* Select the CSS Framework
* Enter - No for server side rendering for new.

**ng serve –o =** for open project on browser.

**Angular Components :-**

* It is a class that controls a part of the user interface (UI) and its associated behaviour.
* To create a new component – **ng generate component** **<component\_name >**
* Short command - **ng g c <name>**

Component – **Selector** = selector is unique name for components. Whenever we have to use components then we can use with html tag of selector.

* Command for Run the project - **ng serve**
* Stop the compilation **– CTRL + C**
* We can use coponents in app.components.html by using component selector (in the form of Html tags**)**
* **First impost the component in app.component.html in imports**
* **To install Bootstrap – npn i bootstrap**
* Next add reference of link in **styles** of **angular.json –**
* **./node\_module/bootstrap/dist/css/bootstrap.min.css**

**Routing (navigating) in Angular :- ( in app.routes.ts )**

* **Routing** in Angular refers to the mechanism that allows you to navigate between different views or pages in a single-page application (SPA). Angular's routing module provides the functionality to map URLs to components, allowing for navigation and rendering of views based on the URL.
* **Routes** are a mapping of URL paths to components.
* **RouterLink** is an Angular directive used to create links between different views (components).
* **RouterOutlet** is a directive that acts as a placeholder where the routed component will be displayed.

Export const routes : Routes = [

{

Path : “user-page”,

Components: **UserComponent** (component-name)

},

{

Path : admin,

Components : AdminComponents

}]

**It will automatically imports the urls at top**

* **To enable routing in app.component.html we have to add <router-outlet> </router-outlet>**
* For navigation link in alternating of a href we have to use **routeLink=”/user-page” (component name)**
* **Using Constructor : (using ts file )**
* **Constructor(private router : Router){**
* **}**

**Navigate(){**

**This.router.navigateByUrl(“<component:selector>”}**

**}**

* **Via routerLink (via html) :- [ routerLink ] =”[/component]”**

**Data Binding :-**

1. **One Way Data Binding :-**

**I. Via Interpolation {{data}} =**

* **Interpolation** refers to **embedding expressions into marked up text**. By default, interpolation uses the **double curly braces** **{{ and }}** as delimiters.
* first create a variable add value that is to bind.
* And then where you have to bind data just add {{ data }} value will be bind.
* Ex- in app.ts = in class
* name ; string = “Jaydip”
* <h2> Hello {{Name }}</h2>
* That will show output Hello Jaydip

1. **Property Binding –**

* we can bind data in any html tag property.
* Ex - <p [class] = “clname”> This is Property Binding </p>
* We have to bind using [] square brackets.

1. **Event binding :-**

* In event binding first we have to create an function.
* Attach to the html element like button , p , span, heading.
* Ex – greet (){

Alert(“ Hello, Welcome to my website “);

} // app.components.ts

<Button class=”btn btn-danger” (click) = “greet()”>Show message<button> //in app.components.html

1. **Two Way Binding :-**

* We can bind the data using [(ngModel)]

1. **Using Signal = signal() =**

**Directives :-**

1. **Structural Directive :-**
2. **\*ngIf** – this directive is used to insert or remove the element.

* We can remove or add html element on button click , dropdownselect or checkbox select.
* <div \*ngIf="isLoggedIn">Welcome, user! </div>
* <button (click)="toggleLogin()">Toggle Login Status</button>
* Ex- isLoggedIn: boolean = false;

toggleLogin() {

this.isLoggedIn = !this.isLoggedIn;

* }
* Jyala hide karayach aahe tyala \*ngIf vapara ani jyacha event var hide karayacha aahe ( button,checkbox or dropdown ) tyala [(ngModel )] vapara.
* To use the \*ngIf and \*ngFor we have to import the commonModule in component.ts file.

1. **\*ngFor –** It only needs an Array for itreration.

* **Syntax :**- <li \*ngFor=”let city of cityArray”>**{{city}}</**li>
* First create array which we have to itrerate.
* Then add the syntax to the element that which we use for itreration.
* Ex – we can use in <li> , cards <option > for multiple option in dropdown list.
* We can use \*ngIf inside \*ngFor for comparison.

1. **Attribute Directive :-**
2. **[ngClass] :-** The ngClass directive allows you to dynamically add or remove CSS classes on an HTML element based on a condition or expression.

<div [ngClass]="'bgcolor'">This is highlighted.</div>

In .ts = { bgcolor :string = “bg-primary”}

1. **[ngStyle] :-** The ngStyle directive allows you to dynamically apply inline styles to an element based on a component property or expression.

**Ex-** <div [ngStyle]="{ 'color': textColor, 'font-size': fontSize }"> This text has dynamic styles. </div>

Class{

textColor = 'blue';

fontSize = '20px';

}

* **Control flow Statement :-**

1. **@if :-**

@if (istrue == true){

<div class=”bg-primary”>

Hello Jaydip </div>

@if(istrue){

<div class=”bg-primary”>

Hello Jaydip </div>

}

@else{

<div class=”bg-warning”>

Hello Jaydip </div>

}

1. **@for :- @for(item of items; track $index) { }**
2. **@switch :-**

Daynumber :string = “”;

< input type= “text” [(ngModel)]=”daynumber”;

@switch(daynumber){

@case(‘1’){

<span> Monday </span>

}

@case(‘2’){

<span>Tuesday </span>

}

@case(‘3’){

<span> Wednesday </span>

}

@default{

<span> No day Selected </span>

}

}

* **Pipe in Angular :- (we need to import this every pipes)**

Angular provides built-in pipes for typical data transformations.

Datepipe, uppercasepipe, lowercasepipe, jsonpipe, etc.

Using pipe – (currdate | date : ‘dd-mm-yy’)

you can define a custom pipe by implementing a TypeScript class with the @Pipe decorator. A pipe must have two things:

* A name, specified in the pipe decorator
* A method named **transform t**hat performs the value transformation.

The TypeScript class should additionally **implement** the **PipeTransform interface** to ensure that it satisfies the type signature for a pipe.

* **Template Form Validation :**

**#hash-property =** it is a variable created in html. We can store the [(ngModel)] value.

* We can use control statement @if , @else.
* Ex - @if (variablename.errors ?. [‘required’]){
* This is required field.
* }

**Reactive form validation :-**

* Reactive forms are defined in the component class using **FormGroup,** **FormControl,** and **FormArray**. Validation is applied directly in the component class using Angular’s **Validators** class.

Ex - myForm = new FormGroup({

name: new FormControl('', [Validators.required, Validators.minLength(3)]), }); }

* **Get API using HttpClient :-**
* First create a **Service of HttpClient (**is .ts file**)**

In app.config provider section - add ProvideHttpClient;

* **Ex** - Constructor(private http :HttpClient){}

We can create object of httpClient using inject

* **Ex –** http = inject(HttpClient);
* **Api Call :-**

# use subscribe for getting and storing data in variable.

**Syntax : -** (**GET API**)

MethodName(){

this.http.get(“< - - API Url - - > ”).subscribe((res:any)=>{

})

**Syntax : -** (**POST API**)

MethodName(){

this.http.post(“< - - API Url - - > ”, <..Object..>).subscribe((res:any)=>{

})

* **Services in Angular : -**

**services** are classes that are used to provide specific functionality that can be shared across different components of an application.

* Used to store most commonly data in file. Used for commonly used urls, api calls.
* Ex – Used for API calling.

**2 Ways to us Services :-**

* **With inject()**
* Ex – http = inject(“HttpClient”)
* **With Constructor () in .ts file**
* Ex – constructor(private Studentservice : servicesdemo ) {
* }
* using dependency injection in constructor.

* **Reusable Components :-**

**reusable components** are components that can be used in multiple parts of your application without needing to rewrite the same code.

**Input and Output Properties**: Components should expose @Input and @Output properties for dynamic data binding and communication with parent or child components.

* @Input: For receiving data from a parent component. (Parent to child)
* @Output: For emitting events or notifying the parent component of changes. (Child to Parent )
* First we have to create separate folder for reusable components.
* Then create component that we have to use in multiple components or files.
* We can use component in multiple components with component selector <selector name > </selector name>
* **@input ()**= also Known as **Property decorator**.
* @inport () msg :string =” ”; // for passing message in child component.
* <app-Alert [message]=”Message from input ”> </app-Alert>

Ex - <div class="row">

<div class="col-6">

<app-reusable-progress [progressvalue]="70" ></app-reusable-progress>

</div>

</div>

**@output -**

* @output () btnclick = new EventEmitter<any> ();
* Onclick(){

this.btnclick.emit(“Hello jay”);

* }
* We can send the data in emit function and add (btnclick)=getdata($event) in html
* Then use in html button
* Ex - <button (btnclick)=”ondelete(item.id)” class=”btn btn-primary”>
* <my-button (btnclick)=getdata($event)>

**Component Life Cycle Events :-**

ngOnInit () :- Automatically call function after component load.

* **ngtemplate :-** <ngtemplate #ngdemo>
* **ng-container : -** we are calling api and api is taking 5 sec to load then we can show loading animation on that tableusing ng container.
* use #property in else of \*ngIf
* Instead of showing data table we can render the loading container.
* Check notes.
* **ViewChild() :-** ViewChild is a decorator used to access and manipulate a child component, directive, or DOM element within a parent component's template. It allows the parent component to reference and interact with the child elements or components directly.
* he reference to the child element/component will only be available after the view has been initialized, which is why you typically access it in ngAfterViewInit() lifecycle hook.
* **Decorators :-**
* **Decorators** are special functions that allow you to add metadata to a class, method, property, or parameter. These decorators provide a way for Angular to understand how to process and instantiate components, services, directives, pipes, etc.
* **@components, @ngmodule, @injectable, @Directive, @Pipe, etc**

Class Decorators - Component, injectable, pipe, directive,module.

Property Decortors – Input, Output, hostBinding

Method Decorators – hostListner

Parameter Decorators – inject

* **Guards in Angular :-**

In Angular, guards are used to control access to different parts of an application, specifically for managing navigation to and from routes. They allow you to implement logic that determines whether a user can access a route or not, and can be used for various purposes such as authentication, authorization, or checking unsaved changes.

* **CanActivate**: -  Used to prevent unauthorized users from accessing a route.

 This guard runs before the route is activated.

 Example: Checking if a user is logged in before allowing access to a route.

**CanActivateChild**:

* Similar to **CanActivate**, but it specifically applies to child routes.
* Used when you want to guard a parent route, but need the same logic for its child routes.

**CanDeactivate**:

* Used to prevent users from leaving a route if certain conditions aren't met, like unsaved changes in a form.
* This guard checks whether the user can navigate away from the current route.
* **Class & Interface :-**
* **Class** is created to define the data types of the objects or variable.
* It avoids to define the any type.
* **Interface -** They are used for type-checking and to ensure that objects conform to a particular structure, without providing implementation.

#Whenever you are creating a variable and bind it to the form then create –**Class**

**#**whenever creating an variable to hold array then create a **– Interface**

**Class** file madhe class create kara > variable with types - sname : string

Create constructor and initialize the values. This,sname = “”

Then we can create objects of of that class.

**ViewClild():**

@viewChild(‘newModel’) model : ElementRef | undefiled;

<div class=”model” #newModel > </div>

**ASP .Net**