Angular (also referred to as "Angular 2+") is a [TypeScript](https://en.wikipedia.org/wiki/TypeScript)-based,  [open-source](https://en.wikipedia.org/wiki/Free_and_open-source_software), Single page [web application framework](https://en.wikipedia.org/wiki/Web_framework) led by the Angular Team at [Google](https://en.wikipedia.org/wiki/Google) and by a community of individuals and corporations.

Angular Js comes in 2010, Latest version is Angular-15 and every six-month Google release updated version (May/Nov). Angular version 3 was never released.

Opinionated (use specific design pattern) and it’s maintained by google.

Use typescript and Very efficient CLI command line interface.

The Angular CLI (Command Line Interface) is a command-line tool that helps you to create, build, and manage Angular projects.

Installation – on terminal-1-**npm install -g @angular/cli**

**2-ng new my-angular-app 3-cd my-angular-app 4-ng serve 5-http://localhost:4200.**

Ex-: ng new project-name, ng serve, ng build, ng generate module module-name, or ng g m m-name, ng generate component component-name, ng generate service service-name, ng test, ng help.

**<span>Name:{{name}}</span> name is view model variable.**

The double curly brace syntax, "{{ }}", is used to bind expressions to string interpolation, are **placeholders** in a template that are replaced with actual values at runtime.

**Template literals,** on the other hand, are a new feature in ECMAScript 6 (ES6) that allows for string interpolation and multiline strings. They are enclosed in backticks (`) instead of single or double quotes and can include placeholders using the ${variable\_name} syntax. When the string is evaluated, the placeholders are replaced with their corresponding values.

const firstName = 'John';

const lastName = 'Doe';

const message = `Hello, ${firstName} ${lastName}!`;

**String interpolation** is a technique used to embed expressions or variables inside a string, allowing for dynamic string generation.

const name = "John";

const age = 30;

const message = `My name is ${name} and I am ${age} years old.`;

console.log(message);

The ${name} and ${age} placeholders are replaced with the values of the name and age variables, respectively, resulting in the string "My name is John and I am 30 years old.

@Component({selector: 'app-root',

template:`<h2>{{mangay}}</h2>`,

styleUrls: ['./app.component.css']})

export class AppComponent {mangay="list of course";}

//list of course

Difference between Angular and AngularJS:

1-Architecture: Angular is a complete rewrite of AngularJS with a different architecture. Angular uses a component-based architecture where components represent the building blocks of an application, whereas AngularJS uses a directive-based or MVC architecture.

2-Language: Angular is written in TypeScript, which is a superset of JavaScript, whereas AngularJS is written in plain JavaScript.

3-Performance: Angular is faster than AngularJS due to the use of the **new Change Detection mechanism** and **Ahead-Of-Time (AOT)** compilation, which means that the application is pre-compiled during the build process, making it faster at runtime.

4-Directives: In AngularJS, directives are used to create reusable components, whereas in Angular, components replace directives as the primary way to create reusable code.

5-Dependency Injection: Dependency injection in Angular is more powerful and flexible than in AngularJS. It allows for better testing and modularization of an application.

6-Mobile support: Angular has better support for mobile devices as it has a built-in mobile toolkit, whereas AngularJS doesn't have this feature.

**Change Detection mechanism** in Angular is a powerful feature that detects changes in the data and updates the view accordingly. Whenever there is a change in the component's data, Angular triggers a change detection cycle, which updates the view with the latest data.

In Angular, the Change Detection mechanism uses a unidirectional data flow, which means that the data flows from the component to the view. The component is the source of truth, and the view is a projection of the component's data.

Angular provides two strategies for Change Detection: Default and OnPush.

Default Change Detection: This is the default strategy used by Angular. In this strategy, Angular checks for changes in all components and their child components every time there is a change in the data. This can be resource-intensive and can impact the performance of the application, especially if the application has a large number of components.

OnPush Change Detection: This strategy is an optimization of the default strategy. In this strategy, Angular only checks for changes in components if their input properties have changed or if an event has been triggered. This strategy is more performant than the default strategy as it reduces the number of change detection cycles.

To use the OnPush strategy, you need to annotate the component with the **ChangeDetectionStrategy.OnPush** decorator. This tells Angular to use the OnPush strategy for the component.

**History**

1-AngularJS (version 1.x): k/a Angular Js developed by google in 2010 on MVC (Modular view Controller) architecture

2-Angular 2-2016-major changes comes in, built from scratch, component-based architecture, The new version of Angular also introduced a new language called TypeScript, which added optional static typing and other features to JavaScript.

3-were up to 2.3 then 4 came out bcz of @angular/router library gets updated 3.3 and other library @angular/core/compiler/ http were at 2.3 version. To avoid conflict, google releases 4.

4-Angular 4-2017

5-Angular 5 introduced a new HttpClient

HttpClient is a built-in module to send HTTP requests and receive responses from a server.

6-Angular 6 introduced improved support for Progressive Web Apps (PWAs).

PWA uses modern web technologies in app to provide a native app-like experience to users.

A native app is a software application designed and developed specifically for a particular platform or operating system (OS), such as iOS or Android. Its helps to get High performance, Rich user experience, Offline capabilities, better security, and reliability. (Longer development time and higher development costs.)

7-Angular 7-2019

8-Angular 8-2019-

9-Angular 9-Ivy, a new rendering engine that improved performance and reduced bundle size.

10-Angular 10-new Date Range Picker component and better support for Web Workers.

11-Angular 11 introduced faster compilation times

12-Angular 12 improved support for ESLint and a new CSS parsing engine.

Superset and preprocessor are terms commonly used in the context of programming languages and technologies. They refer to different concepts:

**Superset:** A superset is a programming language that includes all the features and syntax of another language, along with additional features and enhancements. In other words, a superset extends the capabilities of the base language. The term is often used when discussing languages that are designed to be compatible with an existing language while adding extra functionality.

Ex: TypeScript is a superset of JavaScript. TypeScript includes all the features of JavaScript but also introduces static typing, interfaces, and other features do not present in JavaScript. This means that any valid JavaScript code is also valid TypeScript code, but TypeScript adds its own features on top of JavaScript.

**Preprocessor:** A preprocessor is a tool that processes source code before it's compiled or interpreted. It performs transformations on the code based on directives or commands embedded in the code itself. These transformations can include text replacement, conditionals, and other operations that modify the code before it's passed to the compiler or interpreter.

Ex: Sass and Less are preprocessor languages for CSS. They introduce features like variables, nesting, and mixins that make writing and managing CSS code more efficient. The preprocessor processes these enhanced CSS-like languages into standard CSS that browsers can understand.

In summary, a superset extends an existing programming language with additional features, while a preprocessor is a tool that processes source code before compilation or interpretation, often enhancing the code with features not directly supported by the target language.

**Why Ts..TypeScript**

Ts is Superset of JS means any valid JS Code is valid Ts code.

TypeScript is just a JS with additional features like strong and static typing which makes it predictable and easier to debug.

-Ts Concept classes, interfaces, constructor, access modifier (public and private properties)

-Catch error on compile time and can access additional Ts tools.

Its complier IVY rendering engine who transpile the Ts code into JS code so that browser can understand.

1-Ts installation **npm i -g typescript**

2-To check the version **tsc --version (**4.9.5)

3-**code main.ts** or file Name

When we do ng serve or any CLI cmd, CLI calls Ts complier under the hood to transpile all codes.

Data Types in Ts-

1-Number, 2-Boolean, 3-String, 4-Array, 5-Object, 6-enum, 7-Tuple 8-Void, 9-Null,10-Undefined,

11-Union, 12-Intersection.

6- enum- define a set of named constants means automatically assign number to the set values.

enum WeekDays {

Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday}

Then Monday it at 0 tueseday at 1 same as array index

let today = WeekDays.Wednesday;

console.log(today); // Output: 2

enum Color {Red,Green,Blue}

let chosenColor: Color = Color.Green;

console.log(chosenColor); // Output: Green

7-Tuple: Represents an array with a fixed number of elements of different types.

let person: [string, number] = ["John", 30];

8-Void: Represents the absence of a value, typically used as the return type of functions that don't return a value.

11-Union: Represents a value that can be of one of multiple types.

let value: string | number = "Hello";

value = 42;

12- Intersection: Represents a value that has multiple types combined.

interface Nameable {name: string;}

interface Ageable {age: number;}

let personInfo: Nameable & Ageable = {name: "Bob", age: 30 };

-In TypeScript, an assertion is a way to tell the compiler that you know the type of a value better than it does.

**Interface:** to define a contract or shape of an object as we can assign data type to any variable like let a:string=”today”, but in case of object where 2 value properties are there Object{value1:property1, value2:property2} here we can assign any data type to Object as string or other so with the help of interface we can assign data type to Object.

first, we have to defined interface. Like name would be string as use as string only…

interface Person {

name: string;

age: number;

greet: (name: string) => void;}

const person: Person = {

name: "John",

age: 30,

greet: (name: string) => {

console.log(`Hello, ${name}! My name is ${this.name}.`);}};

For custom object we need to explicitly allocate memory to it by initialising it using new operator

Let person=new Person ();

Here person is a class and Person is an object, so an object is an instance of the class. My name is Ashish is an object I am a human is class.

**Cohesion:** things are related should be part of one-unit, means all related code and functionality put together within a module or class.

**Constructor:** every class can have constructor which is basically a method that is called when we create an instance or object of that class.

**Access Modifier:** keywords that control the visibility and accessibility of class members.

public, private, protected.

public: accessed from anywhere, both inside and outside the class.

private: accessed within the class in which they are defined, it is not accessible from outside the class, including its subclasses. This ensures that the member is encapsulated and cannot be directly manipulated from external code. Accessible only within the defining class.

protected: accessed within the class in which they are defined and in any subclasses that inherit from the class. Accessible within the defining class and its subclasses.

class MyClass {

private privateProperty: number = 42;

private privateMethod() {// Implementation}}

const instance = new MyClass();

instance.privateProperty; // Error: Property 'privateProperty' is private

instance.privateMethod(); // Error: Property 'privateMethod' is private

class BaseClass {

protected protectedProperty: string = "hello";

protected protectedMethod() {// Implementation}}

class SubClass extends BaseClass {

useProtected() {

console.log(this.protectedProperty); // Accessible here

this.protectedMethod(); // Accessible here}}

const instance = new BaseClass();

instance.protectedProperty; // Error: Property 'protectedProperty' is protected

instance.protectedMethod(); // Error: Property 'protectedMethod' is protected

A Class Component having data/variables/properties and function.

Import {} from ‘./’ for same folder

**Built-in modules in Angular:** 1-FormsModule, 2-HttpClientModule, 3-RouterModule, 4-BrowserAnimationsModule, 5-MatIconModule (Material design Icon Module), 6-MatButtonModule, 7-MatDialogModule, etc.

**MatDialogModule -** provides a dialog component that can be used to display modal dialogs and pop-ups in an Angular application, making it easy to create interactive UI components.

**Module-**a angular module should have-

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

import { ReactiveFormsModule } from '@angular/forms';

import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { ChildComponent } from './child/child.component';

@NgModule({

declarations: [AppComponent, ChildComponent],

imports: [BrowserModule,AppRoutingModule,ReactiveFormsModule],

providers: [],

bootstrap: [AppComponent]})

export class AppModule { }

Inside a Modules we have imported modules, exports-to export,

@NgModule decorator({declaration: [], imports:[], providers:[], bootstrap:[]}),

export class AppModule{}

Providers to creator of services, bootstrap to browse all UI on browser, declaration to declare like components, directives, pipes etc

Bootstrapping is the process of starting up an Angular application. The bootstrapping process involves creating an instance of the root component and rendering it within the browser.

The bootstrapping process starts with the main.ts file, which is the entry point for the application. The main.ts file contains the code to bootstrap the root module of the application.

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule)

.catch(err => console.error(err));

the platformBrowserDynamic().bootstrapModule(AppModule) method is used to bootstrap the AppModule, which is the root module of the application. This method tells Angular to start the application and render the AppComponent within the browser.

**1-**Component-2-Services-3-Pipes-4-Diretctives,5-Data-binding,6-Metadata,7-Tempaltes 8-Dependcny Injection

**Template –tells how the component should view means HTML file.** <h1>{{title}}</h1>

<p>{{description}}</p>

**Services** -is a reusable piece of logic that can be used across multiple components. Services are used to encapsulate and share common functionality throughout an application, making it easier to maintain and test the application. (Reuse, Encapsulated just like function).

**Directives**-extend the **power of html attributes or element.** Directives provide a way to add custom behavior to an element, or to **manipulate the** **DOM** in a specific way.

Ex-\*ngFor and \*ngIf and can be customs.

@Component({

selector: 'app-root',

template:`<h2>{{mangay}}</h2>

<ul><li \*ngFor="let jaangaye of courses">{{jaangaye}}</li></ul>`,

styleUrls: ['./app.component.css']})

export class AppComponent {

mangay="list of course";

courses=["course1","course2","course3"];}

//list of Cousre

Course1

course2

course3

3 types of Directives in Angular

1- Component Directives: is associated with a **template**. It creates and controls a portion of the DOM and can interact with data and services.

2- Structural Directives: to manipulate the structure of the DOM. They add or remove html **elements** from the DOM based on conditions, start with asterisk, such as \*ngIf, \*ngswitch and \*ngFor.

3- Attribute Directives: used to change the appearance or behavior of an element. They can be used to add styling, or to bind events to elements.

**ngContent:** directive is used in Angular's component templates to define "content projection" points. Content projection allows you to pass content from a component's parent template into the component's template.

**ngContainer** is also a structural directive in Angular, similar to ngContent. Like ngContent, ngContainer doesn't add or remove elements from the DOM directly. It's often used in conjunction with other structural directives to help control the layout and rendering of elements. The ngContainer directive, represented by the <ng-container> HTML tag, is used to group elements in a template. It acts as a lightweight container for content without introducing additional elements into the rendered output. One of the primary use cases for ngContainer is when you need to apply structural directives (like ngIf or ngFor) to multiple elements without adding extra wrappers to the DOM. Like React Fragment.

Both ngtemplate and ngConatiner are structural directives and both not added in dom, both as react fragment.

ngTemplate Vs ngContainer- ngtemplate require template variable and render content basis of other structural directives.

**Dependency Injection (DI)** is a software design pattern where we can use the one functionality or service of an object to another component, means other component is dependent on first component for functionality and first component is injected to other component.

A 25-dollar term for 5 cent concepts.

DI means injecting or providing the dependency of a class into its constructor.

Dependency=Service

Injection= to add or injected

Whenever angular file install it will automatically generate 6 files-app.moudule.ts , app.component.ts and app.component.html, app.component.spec.ts and app.component.css, app-routing.module.ts

While new component generated 4 files creteated-child.componet.ts/.html/.css/spec.ts

**Data-Binding-** Communication or synchronization of data between the component and the template. The communication between **app.component.ts** and **app.component.html** file It allows you to bind a component's property to a view, so that whenever the component's data changes, the view is automatically updated. Angular supports two-way data binding, which means that changes made in the view are reflected in the component and vice versa.

While @Input() and @Output() decorators are used for communication in between parent and child components and its templates.

The main techniques for data binding in Angular are:

Interpolation

Property Binding

Event Binding

Two-Way Binding

**Folder/File Structure:**

1-.angular -it is not generated initially comes when after the project run in vs editor after that it will appear in project. Start generated after Angular-13.

2-.vscode-vs editor dependency not related to project (ng serve launch)

3-node\_modules-conatianer of all npm or node packages where all of the third-party libraries and dependencies used by the project are stored and their reference will be created in package.json and package-lock.json

node\_modules folder- Stored all the third-party libraries the app may depend upon. Can become quite large, as each package and its dependencies are stored in their own subfolder. As a result, the node\_modules folder is usually excluded from source control, and it's generated from scratch whenever you clone the project repository. If you want to delete, you can delete this folder while share to some one. The other person should install the npm packages than it will get restore just we need only package.json and package-lock.json as their reference are already their.

This folder purely for development not deployed in the server.

**Src folder:** app, assets, favicon.ico, index.html, main.ts, styles.css, where all the sources are there.

**4 app folder:** core of your application, including the components, modules, services, models, and other code that make up your application.

4.1 app.component.ts: TypeScript file that defines the component's logic.

4.2 app.component.html: HTML file that defines the component's template (view).

4.3 app.module.ts: defines the components, services, and other code that belong to the feature.

4.4 app-routing.module.ts: for routing

4.5 app.component.spec.ts: for testing

4.6 app. component.css: for css

**5-assets**: This folder contains all the static assets for your application, such as images, fonts, and other media.

6-**environments**: for environment setup such as development, staging, and production.’

7-fevicon icon-title-bar logo image (favorite icon)

8-core folder: folder contains code that is used across your application.

9-shared folder: shared across multiple features of your application, such as common components, services, and pipes.

10-styles folder: Global stylesheets for your application- Sass, SCSS, Less or CSS files.

**11-index.html:** This is the main HTML file for your application, and it's the file that is served when the user visits your application. <body> <app-root></app-root></body>

**12-main.ts:** This is the main TypeScript file for your application, and it's where you bootstrap the main component of your application. platformBrowserDynamic().bootstrapModule(AppModule)

13-app.module.ts: This is the root module file for your application, and it defines the components, services, and other code that belong to your application as a whole. Like @Decorators({ Declaration, Import, Provider, bootstrap})

14-15-pollyfils.ts and browserlistrc files makes angular to runs in all browsers or older version like safari, chrome, edge etc. bcz we use typescript and browser don’t understand typescript they u/s only html n js.

TypeScript is the superset of JS means additional features of js and we write typescript in separate file being generated n browser convert it into normal js file just like preprocessor in css.

The Angular engine is named "Ivy Rendering Engine based on AOT.

Just-in-Time (JIT) compilation is a process in which Angular applications are compiled as they are executed by the browser. This contrasts with Ahead-of-Time (AOT) compilation, in which Angular applications are compiled (pre-compiled) in advance before they are run in the browser.

**Test.ts and karma.conf.js** both files for testing used by tester and use “**jasmine**” frame to test angular projects. Karma is test runner for Js code.

**Editorconfiq**-set the rule for the project in a group and each member are bound to follow the basic rules for that project which defines coding styles for a project and all the developer use same setting in their editor.

**angular.json** file contains angular configuration files, a set of properties that control various aspects of the build process. Where we can use specific library from node\_modules like using bootstrap in styles.

"styles": ["src/styles.css",

"node\_modules/bootstrap/dist/css/bootstrap.min.css"],

gitignore- excluding files and folder from git repository. Versioning n managing your source code.

**Readme-** provides information about a project, provide a quick overview of the project, including its purpose, how to install and use it, and any other relevant information.

tsconfig.app.json, tsconfig.json, tsconfig.spec.json, files for typescript

tsconfig.json- compile ts code into js code to browser can u/s

"target": "ES2022",

"tslib": "^2.3.0" and “zone.js": "~0.12.0" “~” tilt will search from patch only and the upper level, of patch will get install (major and minor will be fixed) and “^”teirat will search upper level of both from minor changes 2nd part and patch/bug.

**Ex-"tslib": "^2.3.0" teirat** will see first the upper level of minor changes if he finds then install above the minor changes of 3 means 4 will get install otherwise next will search to patch or revision above 0 will get installed.

14.2.5=14 is major,2 is minor changes and 5 is patch after bug resolve release.

If no tilt, no teirat then will install exact version.

dependencies =dependencies for production w/o app will not run.

devDependencies -dependency for development and can run the app.

pacakge.json- contains the libraries that the your app is dependent upon.

6 files- name, version, scripts, private, dependencies, devDependencies.

The package-lock.json file is used to ensure that all installations of the project use the same dependencies and versions, making the installation process more deterministic and reliable. This is important to ensure that the project works consistently across different machines and environments.

package.json vs package-lock.json

Both package.json and package-lock.json are files used in Node.js projects to manage dependencies and their versions.

all the libraries needed or **required** for the project will be in package.json its not necessary that these library with the same version must have in your system while in lock.json file contains all the dependencies which is already(lock) installed in your system

basic requirement -package.json

all node modules installed libraries reference are into lock.json

With the help of package.json and package-lock.json files we can go to any version, new or old to work with by changing the version numbers and the same version will get into update in package-lock.json file.

In the context of Node.js and the npm package manager, there are two types of dependencies that can be specified in a package.json file: dependencies and devDependencies.

**dependencies:** **For Run..**These are the dependencies that are required for the project to **run in production**. This means that the application cannot function without these dependencies being installed. dependencies are installed when you run npm install without the --dev flag.

11 major references for angular libraries and if we import other libraries then its reference will store here. ex Bootstrap

npm install –save package name or npm install that would be installed as a dependency.

npm install –save dev package name or npm i s d package name for dev dependency installation.

"dependencies": {

"@angular/animations": "^15.1.0",

"@angular/common": "^15.1.0",

"@angular/compiler": "^15.1.0",

"@angular/core": "^15.1.0",

"@angular/forms": "^15.1.0",

"@angular/platform-browser": "^15.1.0",

"@angular/platform-browser-dynamic": "^15.1.0",

"@angular/router": "^15.1.0",

"bootstrap": "^5.2.3",

"rxjs": "~7.8.0",

"tslib": "^2.3.0",

"zone.js": "~0.12.0"},

-@ are core packages of angular and others are support pakages like "rxjs": for http apis to connect front end to backend

**devDependencies:** **for Dev..**These are the dependencies that are only needed during the development and testing process. This means that the application can still function without these dependencies being installed. devDependencies are installed when you run npm install with the --dev flag.

One advantage of separating dependencies and devDependencies is that it allows for a faster installation process and reduces the size of the deployed application. In many cases, production-level dependencies may have a large size, but they are critical to the functioning of the application, so they must be included.

CLI-compile ts project into js project

|  |  |
| --- | --- |
| **Modules** | **Component** |
| -Organise bundle of related codes help to break up the application into smaller, manageable parts  -set of encapsulated code meant to communicate with external app.  -a module can contain one or more components,  Group of related Components. | Reusable codes related to UI.  - component must belong to a module |

**Flow:**index.html=>main.ts=>app.module.ts=>app.component.ts=>app.component.html=> index.html

In app.componet.ts we have selector: 'app-root', direct app to find the whatever written in templateUrl: './app.component.html', show at app-root.

Modules is the group of declaration-group of components, import- created Modules, provider-services, pipes come in, and export.

Bootstrap provider bootstrap come only in root app.module.ts

First module file will load then app.component file and then finally your app will get load.

**Decorator** is type script function start with @and end with parenthesis(argument) and don’t put semicolon; at the end, (ex-@name(argument)) and used to modify or **extend the behavior of an Angular component or module** and they provide a way to add metadata to your code.

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'my-angualr-app';}

Import decorator from core then define as @decorator({argument metadata as value and property and }) to tell how component will work.

In this example, the @Component decorator is applied to the AppComponent class to declare it as an Angular component. The @Component decorator takes an **object literal** as an argument, which contains the component's metadata, such as the selector, template, and styleUrls.

Built-in decorators, such as @Component(), @NgModule(), @Directive(), @Pipe(),@Injecatble() to define services, @Input(), @Output() etc.

The @Input() decorator is used to pass data from a parent component to a child component. The @Input() decorator allows you to bind a property in the child component and receive its value from the parent component. The value is passed to the child component as an input property vice versa @Output().

The @Output() decorator in Angular is used to pass data from a child component to a parent component. The @Output() decorator allows you to bind an event in the child component and raise it from the child component to the parent component.

HostElement-where we apply custom directive or hostlistner directive.

Hostlistner -help to handle event on host element.

[Home.component.html](http://Home.component.html) is host element and Its logic in comes in [home.component.ts](http://home.component.ts) file

myclick() is just function in [home.component.ts](http://home.component.ts) and we attached this function with hostlistener decorator

@Hostlistener(`click`) myClick(){}

Hostlistner click event will apply over all home component myClick event.

Ex-@Hostlistener(`click`) myClick(){alret(“clicked”} will show clicked whenever we click on home page.

The **@HostListener** decorator is used to attach event listeners to the host element of a component. The @HostListener decorator allows you to listen to events that are triggered on the host element, such as clicks, mouse events, keyboard events, and more.

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

@Component({selector: 'app-component', template: ` <p>Click the host element to see the event in the console.</p>`})

export class AppComponent {@HostListener('click', ['$event']) onClick(event: MouseEvent) {

console.log('The host element was clicked!', event);}}

pass an event as argument into the attached function and when the event fire call that associated function to do dom manipulation, ex. making cursor active at any place, seeing map and holding cursor at one place make that place active, making mail read unread bold.

Means if we do any activity on non-clickable element then browser will listen that activity and it become active.

Host listener is core library part and fire only below mentioned function.

Takes 3 arguments-1-by what event should triggered,2-an event start with $ sign,3-new function.

-Wokes only just below mentioned function

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'my-angualr-app';

@HostListener(`click`,[`$event`])

show(){alert("hello");}

}

In Java Script:

const button = document.querySelector("button");

button.addEventListener("click", function(event) {

console.log("Button was clicked!");

});

**Components:** Components are the building blocks of an Angular application. They define a reusable piece of UI and the logic associated with that UI. Each component has its **own template,** styles, and logic, and can interact with other components to create a complete application.

It’s a subset of directory and always associated with a template.3 part-

1-Component Class: defines the properties and behavior of the component.

2-Template: This is an HTML template that defines the structure and appearance of the component's UI.

3=Metadata: This is additional information about the component

Component**=**html template + data (properties) + logic(functions)up

**1-Create Component Then 2-Register it in Module then 3-Add an element in html.**

A component has encapsulated data, Html template/view, and logic.

**Component=data + html view + function**

Class component having data/variables/properties and function.

**Selector**- is a string that is used to identify a component in a template or html. Ways to use-

1-Element selector- <app-student>

2-Class selector: `.app-my-component` ex <div class=”root”></div>(@Component({

selector: '.root',

3-[Attribute selector]: [app-my-component]. ex <div root></div> (selector: '[root}',

**Template: in @Component decorator**

To render the html code Can be tempalteUrl- for external file or template for inline.

If we use both tempalteUrl and template in one decorator, then tempalteUrl will have highest priority (will apply only one). While in styleUrls and styles, whichever comes in last/bottom in @Component decorator has highest priority (apply only one other will be completely ignored).

Use of backtick for multiline template.

**@Component decorator** styleUrl for external, and styles internal and style tag for inline styling.

styleUrls: ['./app.component.css']

styles:[`h1{background-color:red}`]

**@Component decorator** -preserveWhitespaces: true for white space in between button or any..

**ViewProviders** is a feature used to provide a specific set of services and values that are specific to a particular component or directive, and it is not available for content children.

**Component View: Content Children:**

<my-parent></ my-parent> <my-parent><my-child></ my-child></ my-parent>

<my-child></ my-child>

At component level we can provide a service in 2 ways-

Via providers array

Via viewproviders array

-To call any components constructor we have to create object for that constructor first.

-In module we can use data from other component with the help of Provider and only module have Provider but if we want to use a component or class or service in particular component then we have to use viewProviders in that component as we don’t have providers in components.

And we can restrict the usage of service or component by using that particular service or component if only that particular component uses.

@Component({

selector: 'app-root',

template: '<h1>{{title}}</h1>',

viewProviders: [MyService]})

export class AppComponent {

title: string;

constructor(private myService: MyService) {

this.title = myService.getTitle();}}

**providers Vs viewProviders-**providers sharing instances of services across a component and all its child components while viewProviders is used for creating instances of services that are specific to a component and its view children, with each instance having its own narrow scope.

**Singleton**-Single instance of given object exist in memory and pass this single instance to all your components. (Service shared in as a instances).

**Encapsulation: viewEncapsulation-**allows you to keep the internal state like template, css, and logic of a component private and hide it from the outside world and are all encapsulated within the component.

Graphical user interface, text, application, chat or text message

Description automatically generated

Three types of encapsulations: viewEncapsulation

**1-Emulated-(ensapsulation-yes)**Defalut behaviour. No shadow dom and style encapsulation, css will not go from parent to child or vice-versa. Style b**elongs to that particular component only.**

**2-Native/Shadow: (ensapsulation-yes, but share with child)** Create separate shadow dom for that particular component, no ngcontent attribute created,Shadow Dom, and style encapsulated means parent to child css will go only in one direction, downside, parent to child, child to child 1.

**3-None:** **(ensapsulation-no)**No shadow Dom, No encapsulation means any direction data will go and all styles become global style will apply in all the components.

Parent component take other component inside and other component will become child.

**Note:** Every component has its own css file, so every component has its own unique ngcontent attribute created by angular like a component has unique attribute number specific to all this component only and all the element and class pertain to this component has same attribute number like ngcontent-ngaC20. Except global.css file don’t have attribute so that the style applied here available to all components.

**ng-Content-**angular structural directive allows us to add/insert dynamic content in a fixed view template. (Refer)

**ngContent:** directive is used in Angular's component templates to define "content projection" points. Content projection allows you to pass content from a component's parent template into the component's template.

Based on Transclusion concept of Angular1 and k/a Content Projection which allows us to insert shadow dom in your component. Means if you want to insert html element or other component in a component and we can achieve content projection by using <ng-content></ng-content>

For multiple content and we want a particular set of elements or tag comes in a particular place then use **select.** <ng-content select=`h1`></ng-content>

**ng-Container-** ng-container is a structural directive in Angular that allows you to group elements together without adding an extra element to the DOM. It is a logical container that can be used to group nodes but is not rendered in dom tree as a node. Just like react fragment. Means all the content will show into the dom but not the ng-container. (Refer)

**ngContainer** is also a structural directive in Angular, similar to ngContent. Like ngContent, ngContainer doesn't add or remove elements from the DOM directly. It's often used in conjunction with other structural directives to help control the layout and rendering of elements. The ngContainer directive, represented by the <ng-container> HTML tag, is used to group elements in a template. It acts as a lightweight container for content without introducing additional elements into the rendered output. One of the primary use cases for ngContainer is when you need to apply structural directives (like ngIf or ngFor) to multiple elements without adding extra wrappers to the DOM. Like React Fragment.

**ng-Container Vs ng-template-both are structural directives.**

-both are not added in dom, not creating any extra node -both are same as React fragment.

-bcz we cannot use 2 structural directives in one element, to use 2 structural directives we have to make 2 divs which cause dom heavy, to solve this use ng-container and div.

In ng-template content render over dom basis of template variable(#xyz), if we will not put any conditional directive then it will not render the content over dom means to render ng-template we need other directives whereas in ng-Container did not need any other directives or template variables to render the content over dom. Ex.

<ng-container \*ngIf="isLoggedIn; then welcome else notLoggedIn"></ng-container>

<ng-template #welcome>

<p>Welcome, {{username}}!</p>

</ng-template>

<ng-template #notLoggedIn>

<p>Please log in to access this page.</p>

</ng-template>

**Shadow DOM** stands for "Shadow Document Object Model". It is a web standard that allows developers to create encapsulated and isolated DOM subtrees within an HTML document. These subtrees, called "shadow trees," are separate from the main DOM of the document and are rendered independently.

The purpose of Shadow DOM is to enable developers to create reusable components that are encapsulated and have their own internal structure, styles, and behavior. It is particularly useful for creating web components, which are reusable and self-contained UI elements that can be used across different web pages and applications.

With Shadow DOM, developers can create custom HTML elements that have their own encapsulated styles, scripts, and behaviors. The shadow tree of a custom element is hidden from the rest of the document, which prevents style conflicts and provides a clean separation between the component and the rest of the page.

**Shadow DOM** : Shadow DOM stands for "Shadow Document Object Model" is a web standard that provides a way to **encapsulate and isolate** the content and styles of web components from the rest of the page. It allows developers to create components with their own DOM tree and CSS styles that are separate from the main document's DOM tree and styles.

With Shadow DOM, a web component can have its own internal DOM tree and CSS styles that are hidden from the rest of the page. This makes it possible to create reusable and modular components that can be used in different parts of a web application without conflicting with the styles or functionality of other components or the page itself.

To create a Shadow DOM, a developer defines a new DOM subtree and attaches it to an existing element using the Element.attachShadow() method. The Shadow DOM subtree is then populated with elements and styles that are only visible within the component itself.

It is a web standard that allows developers to create encapsulated and isolated DOM subtrees within an HTML document. These subtrees, called "shadow trees," are separate from the main DOM of the document and are rendered independently.

With Shadow DOM, developers can create custom HTML elements that have their own encapsulated styles, scripts, and behaviors. The shadow tree of a custom element is hidden from the rest of the document, which prevents style conflicts and provides a clean separation between the component and the rest of the page.

**DOM Vs Virtual DOM Vs Shadow Dom**

DOM: The Document Object Model (DOM) is a programming interface that allows developers to access and manipulate the content, structure, and style of web pages. The DOM represents the page as a hierarchical tree structure where each node in the tree corresponds to an element, attribute, or text node in the HTML document.

Shadow DOM: The Shadow DOM is a web standard that provides a way to encapsulate and isolate the content and styles of web components from the rest of the page. It allows developers to create components with their own DOM tree and CSS styles that are separate from the main document's DOM tree and styles.

Virtual DOM: The Virtual DOM is a programming concept that provides a way to update the content and style of web pages more efficiently. It works by creating a lightweight copy of the DOM and keeping it in memory. When a change is made to the page, the Virtual DOM is updated instead of directly manipulating the real DOM. Then, the Virtual DOM is compared with the real DOM to identify the minimal set of changes needed to update the page, and those changes are applied to the real DOM.

The DOM is the underlying structure of a web page, while the Shadow DOM provides a way to isolate and encapsulate components within that structure. The Virtual DOM is a programming concept that provides a more efficient way to update the DOM.

……………………………………………………………………………………………………………………………………………………

**Providers:** @NgModule and @Component decorators are having providers.

-Parrent to child communication with the help of @Input decorator and or with the help of input property.

**a: Parrent to child communication with the help of @Input decorator:**

1-In the parent component, define the property that you want to pass to the child component using the @Input() decorator.

export class ParentComponent {

parentData = 'Hello from parent';}

2-Use the child component in the parent component template and bind the input property using property binding

<app-child [mydata]="parentdata"></app-child>

3-Define the input property in the child component using the @Input decorator.

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

@Component({

selector: 'app-child',

template: '<p>{{myData}}</p>'})

export class ChildComponent {

@Input() myData: string;}

Or we can pass it into child html file as below: and use templteurl

<p>child works!value of Child Componet : {{mydata}}</p>;

The error "Property 'data' has no initializer and is not definitely assigned in the constructor" typically occurs when you declare a property in a TypeScript class but do not initialize it in the constructor or anywhere else in the class. In strict mode, which is enabled by default in newer versions of TypeScript, all non-nullable properties must be initialized in the constructor or through a property initializer.

To fix this error, you can do one of the following:

1. Initialize the property in the constructor:
2. import { Component, Input } from '@angular/core';
3. @Component({
4. selector: 'app-child',
5. template: '<p>{{data}}</p>'})
6. export class ChildComponent {
7. @Input() data: string;
8. constructor() {
9. this.data = '';}}

**Ex.1**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']})

export class AppComponent {

title = 'my-angular-app2';

parrentdata= "Some data to pass to the child component";}

2.

<h1>heloooooooooooooo</h1>

<app-child [data]='parrentdata'></app-child>

3.

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

@Component({

selector: 'app-child',

templateUrl: './child.component.html'})

export class ChildComponent{

@Input() data:string;

constructor(){

this.data=''}}

4.

<p>child works!{{data}}</p>

**Template reference variables** allow you to easily reference specific elements or components in your code. A template reference variable is a way to give a name to a specific element or component instance in the template, which can then be used to reference that element or component in the code. To create a template reference variable, you simply add a hashtag (#) followed by the variable name to the element or component you want to reference.

#Pcomponent and #Pcomponent2 are the Template reference variables in place of input.value and (keyup) is the event. (normally event.target.value)

**b: Parrent to Child data transfer with the help of input property:**

<div>

<h1>Parrent Cpmponent</h1>

Parrent Cpmponent : <input type="text" #Pcomponent (keyup)="0"/>

Parrent Cpmponent2 : <input type="text" #Pcomponent2 (keyup)="0"/>

<app-child [Pdata]="Pcomponent.value"></app-child>

<app-child [Pdata]="Pcomponent2.value"></app-child>

</div>

H

<p>Child Component</p>

value of child component: {{Pdata}};

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

@Component({

selector: 'app-child',

templateUrl: './child.component.html',

styleUrls: ['./child.component.css'],

inputs:['Pdata']})

export class ChildComponent {Pdata:any;}

Result//

Graphical user interface, text, application

Description automatically generated

**Mix all for inputs property and @Input decorator example:**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']})

export class AppComponent {

title = 'my-angular-app3';

parrentdata= "Prarrent to child data transfer";

constructor(){

this.parrentdata=''};}

<div>

<h1>Parrent Cpmponent</h1>

Parrent Cpmponent : <input type="text" #Pcomponent (keyup)="0"/>

Parrent Cpmponent2 : <input type="text" #Pcomponent2 (keyup)="0"/>

<app-child [Pdata]="Pcomponent.value"></app-child>

<app-child [Pdata]="Pcomponent2.value"></app-child>

<app-child [Cdata]= 'parrentdata'></app-child>

</div>

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

@Component({

selector: 'app-child',

templateUrl: './child.component.html',

styleUrls: ['./child.component.css'],

inputs:['Pdata']})

export class ChildComponent {

Pdata:any;

@Input() Cdata:string;

constructor(){

this.Cdata=''}}

<p>Child Component</p>

value of child component: {{Pdata}};

value of parrent to child data tarnsfer: {{Cdata}}

Graphical user interface

Description automatically generated with medium confidence

**Child to Parent Data Transfer:**

1-Make Child component

2-Use Child component in parent component

3-Send function from parent component

4-Call function in child component

5-Get data in Parent component

In Angular, you can pass data from a child component to its parent component using the @Output decorator and EventEmitter.

1-In the child component, declare an @Output property and create an EventEmitter object:

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

@Component({

selector: 'app-child',

template: `

<button (click)="sendMessage()">Send Message</button>})

export class ChildComponent {

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

sendMessage() {

this.messageEvent.emit('Hello from child component!');}}

2-In the parent component, use the child component and bind to the messageEvent output property:

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

@Component({

selector: 'app-parent',

template: `

<app-child (messageEvent)="receiveMessage($event)"></app-child>

<p>{{ message }}</p>`})

export class ParentComponent {

message: string;

receiveMessage($event) {

this.message = $event;}}

In this example, when the button in the child component is clicked, the sendMessage() method is called and emits an event with the message "Hello from child component!" using the messageEvent output property.

In the parent component, the messageEvent output property is bound to the receiveMessage() method using event binding. The receiveMessage() method sets the message property of the parent component to the event payload.

When the child component emits the message event, it triggers the receiveMessage() method in the parent component, which updates the message property. The updated message property is then displayed in the parent component's template using string interpolation.

**Ex-2-**1-Define a function in child html and send it to child .ts

<p>child works!</p>

<input type="text" #box />

<button (click)="updateDataEvent.emit(box.value)">Update Data</button>

2- In the child component, declare an @Output property and create an EventEmitter object and bind created function with new event emitter.

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

@Component({

selector: 'app-child',

templateUrl: './child.component.html',

styleUrls: ['./child.component.css']})

export class ChildComponent{

@Output() updateDataEvent= new EventEmitter<string>();}

3-In parent html bind function with event

<!-- <h1>hi</h1> -->

<p>{{mesaage}}</p>

<app-child (updateDataEvent)= "recivedata($event)"></app-child>

4-Call function in the parent ts component and get the data.

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']})

export class AppComponent {

title = 'my-angular-app4';

mesaage="type";

recivedata(item:string){

this.mesaage=item;};}

-Declare a @Output () property, create instance of event emitter object and raise an event in child component to emit data of child component to the parent component with the help of event binding.

**Directives-(means for template only)**

**Directives**-extend the **power of html attributes or element.** Directives provide a way to add custom behavior to an element, or to **manipulate** the DOM in a specific way.

Ex- ngFor and ngIf and can be custom directives.

3-types of Directives in Angular

**1-Component Directives**: is associated with a **template**. It creates and controls a portion of the DOM and can interact with data and services.

It allows you to define a custom **component** with a template and logic, which can be used throughout your application.

To create a component directive in Angular, you need to use the @Component decorator. This decorator provides metadata about the component, such as its selector (which is used to reference the component in templates), its template or templateUrl (which defines the component's HTML), and its styles or styleUrls (which define the component's CSS).

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

@Component({

selector: 'app-hello-world',

template: '<h1>Hello, World!</h1>',

styles: ['h1 { color: blue; }']})

export class HelloWorldComponent {

// component logic goes here}

In this example, we have created a component directive called HelloWorldComponent with a selector of app-hello-world. The component's template consists of an <h1> element with the text "Hello, World!", and its styles consist of a CSS rule that sets the color of the <h1> element to blue.

Once you have created a component directive, you can use it in your application by including its selector in your templates. For example:

<app-hello-world></app-hello-world>

2- Structural Directives: to manipulate the structure of the DOM. They add or remove html **elements** from the DOM based on conditions, start with asterisk, ex- \*ngIf, ngswitch and \*ngFor.

**ngContent:** directive is used in Angular's component templates to define "content projection" points. Content projection allows you to pass content from a component's parent template into the component's template.

**ngContainer** is also a structural directive in Angular, similar to ngContent. Like ngContent, ngContainer doesn't add or remove elements from the DOM directly. It's often used in conjunction with other structural directives to help control the layout and rendering of elements. The ngContainer directive, represented by the <ng-container> HTML tag, is used to group elements in a template. It acts as a lightweight container for content without introducing additional elements into the rendered output. One of the primary use cases for ngContainer is when you need to apply structural directives (like ngIf or ngFor) to multiple elements without adding extra wrappers to the DOM. Like React Fragment.

Both ngtemplate and ngConatiner are structural directives and both not added in dom, both as react fragment.

ngTemplate Vs ngContainer- ngtemplate require template variable and render content basis of other structural directives which Container not required.

**\*ngIf –** The \*ngIf directive is used to conditionally **add or remove** an element from the DOM based on a Boolean expression. For example, you might use ngIf to display a message only if a certain condition is true. Ex. <p \*ngIf="isLoggedIn">Welcome, {{username}}!</p>

<div \*ngIf="loggedIn">

<p>Welcome, {{userName}}!</p></div>

Just like the hidden property of the element the only difference b/w ngIf and hidden property, if ngIf condition false then the element removed from the Dom where as in hidden its just hidden in dom which occupy the dom space.

**ngIf-else** if the condition evaluated by ngIf is false. We have to **create template variable.**

<p \*ngIf="isLoggedIn; else notLoggedIn">Welcome, {{username}}!</p>

<ng-template #notLoggedIn>

<p>Please log in to access this page.</p>

</ng-template>

**ngIf-then-else-**then keyword is used to specify the template to display when the condition is true, and the else keyword is used to specify the template to display when the condition is false.

<ng-container \*ngIf="isLoggedIn; then welcome else notLoggedIn"></ng-container>

<ng-template #welcome>

<p>Welcome, {{username}}!</p>

</ng-template>

<ng-template #notLoggedIn>

<p>Please log in to access this page.</p>

</ng-template>

**ngFor** - The ngFor directive is used to loop over a collection of items and create a new instance of a template for each item in the collection.

**Syntax \*ngfor=”let value of collection”;**

value is variable name of your choosing.

collection is a property on your component which holds a collection it may be array or object.

App.component.ts

export class AppComponent {

title = 'my-angular-app4';

Students:any[]=[{'name':'Rahul'},{'name':'Shnakra'},{'name':'Ganesh'},{'name':'Mahesh'}]}

app.componet.html

<ul>

<li \*ngFor="let kuchbhi of Students">{{kuchbhi.name}}</li>

<li \*ngFor="let kuchbhi of Students ; let i=index">{{i+1}}-{{kuchbhi.name}} with index no start with 1</li>

<li \*ngFor="let kuchbhi of Students; let f=first;let l=last; let ev=even; let od=odd;">{{kuchbhi.name}}/{{f}},{{l}}-{{ev}}/{{od}} </li>

<li \*ngFor="let keyValuePair of Students | keyvalue"></li>

</ul>

Local variables used in \*ngfor : index, even, odd, first, last.

**TrackBy with \*ngfor-** When you use \*ngFor with large lists, it can cause performance issues because Angular needs to re-render or re-construct the entire list every time a change is made in the dom tree. Bcz by default angular track them based on the object identity which means that reference of that object in the memory. So, if you redownloaded course with id1, every time you redownload that course, that course is going to be different object in the memory location even though the content of those object is going be equal.

To improve performance, you can use the trackBy option with \*ngFor it will not re-render over dom.

The trackBy option tells Angular how to track changes in the collection, and it can significantly improve the performance of the application. It will reuse the existing DOM elements as much as possible, rather than creating new ones, which can greatly reduce the rendering time.

<ul>

<li \*ngFor="let item of items; trackBy: trackByFn">{{item.name}}</li>

</ul>

The trackByFn function takes two arguments: the index of the current item and the current item itself, and it returns a **unique identifier** for the item. This identifier is used by Angular to track changes in the collection.

trackByFn(index: number, item: any) {

return item.id;}

trackBy function is use for performance optimisation and will take 2 argument first is index and second is current item, and we can return the **unique identifier** as a return argument.

Sytax: TrackByStudentID(index:number,student:any):string{return student.studentID;}

**Grouping with \*ngfor-**to group data and render it in a nested structure.

**Note:** Leading asterisk means all the directives which start with asterisk (\*) like \*ngFor, \*ngIf, \*ngSwitchCase Angular rewrite that block using ng-template.

**ngSwitch -** Compare the value of property against multiple values. The ngSwitch directive is used to conditionally display different content based on the value of an expression. For example, you might use ngSwitch to display different messages based on the status of an order.

ngSwitch: no asterisk

\*ngSwitchCase: with asterisk

\*ngSwitchDefalult: with asterisk

app.component.html

<div [ngSwitch]="name">

<div \*ngSwitchCase="'Ram'">Hello Ram</div>

<div \*ngSwitchCase="'Sita'">Hello Sita</div>

<div \*ngSwitchCase="'Mohan'">Hello Mohan</div>

<div \*ngSwitchCase="'Shyam'">Hello Shyam</div>

<div \*ngSwitchCase="'Ram'">Hello Ram</div>

<div \*ngSwitchDefault>No data Match</div>

</div>

app.component.ts

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']})

export class AppComponent {

title = 'my-angular-app4';

name="Mohan";}

// Hello Mohan

-no duplicate case should be written otherwise it will get print twice bcz in switch statement there is no break. ex put Ram in name variable and write second case of Ram will print twice.

-Switch cases are case sensitive Ram not ram and should be in ="'Ram'"> **“ ’ ’ ”**

Print at the same place.

<div>

Enter your Name:<input type="text" #txtname (keyup)="0"/>

<div [ngSwitch]="txtname.value">

<div \*ngSwitchCase="'Ram'">Hello Ram</div>

<div \*ngSwitchCase="'Sita'">Hello Sita</div>

<div \*ngSwitchCase="'Mohan'">Hello Mohan</div>

<div \*ngSwitchCase="'Ram'">Hello Ram</div>

<div \*ngSwitchCase="'Shyam'">Hello Shyam</div>

<div \*ngSwitchDefault>No data Match</div></div</div>

**3- Attribute Directives:** used to modify the behavior or appearance of HTML elements in the DOM by applying custom attributes. They can be used to add styling, or to bind events to elements. ngStyle, ngModel, ngClass. etc

**[ngStyle]-**the ngStyle directive provides a flexible and powerful way to apply dynamic styles to your Angular application, allowing you to create rich, interactive user interfaces. (inline)

<p [ngStyle]="{ color: myColor }">This based on the value of 'myColor'.</p>

**[ngClass]-**allows you to add or remove CSS classes to an HTML element based on conditions or expressions in your component. (external)

**Create Custom Directive-** with the help of @Directive Decorator to add custom behavior to HTML elements.

-ng g d dname

Ex-1

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

@Directive({

selector: '[appInputFormat]'

})

export class InputFormatDirective {

constructor(private el:ElementRef) {

el.nativeElement.style.backgroundColor="green"}}

then pass that directive selector in html.

<h2 appInputFormat>Pipes</h2>

****

Ex-2

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

@Directive({

selector: '[appInputFormat]'})

export class InputFormatDirective {

constructor(private el:ElementRef) {

el.nativeElement.style.backgroundColor="green"   }

@HostListener('click') onClick() {

this.el.nativeElement.style.backgroundColor = 'yellow';}}

then pass that directive selector in html.

<h2 appInputFormat>Pipes</h2>

when we click on the page pipes turned yellow.

**Data binding**

Data binding in Angular refers to the synchronization of data between the component (ts file) and the view (html content).

Component to view- ts file to html-one way.

View to component=html to ts-one way.

Ts file to html and vice versa is two-way binding

Diagram

Description automatically generated

1-Interpolation: {{property name}}

2-Property binding: [property name] =’expression’

**Note**: We bind the property of a dom element like src to the property of the component, not an attribute of an html.

3-Atribute binding: <td [atri.colspan]=”myColSpan” align=”center”>Records</td>

4- Event binding: to bind a view element event to a component method. To use event binding, you wrap the element event in parentheses ( ) and set the value to the component method. (click)="handleClick()"

@Component({

selector: 'app-root',

template:`<h2>{{mangay}}</h2> string interpolation

<img src=”{{imageUrl}}”/>

<img [src]=”imageUrl”/> property binding

styleUrls: ['./app.component.css']})

export class AppComponent { mangay="list of course";

imageUrl=”<http://xyz.com>”;

**Two-way binding:** Update and display property on the same time.

Import ngModule in app.module.

Import {FormsModule} from ‘@angular/forms’ and put FormsModule into import in app.module.

Two-way binding is a combination of property binding and event binding that allows you to bind a view element property to a component property and update both values when either change. To use two-way binding, you use the ngModel directive and wrap the element property and component property in parentheses [(ngModel)]="componentProperty".

App.component.html<input [(ngModel)]='roja'>

Enterefd your name:{{roja}}

App.component.ts

roja:string="Ram";

**Pipes**

Pipes (|): are a way to transform data before it is displayed in the view to desired output.

-pipes are used for data transformation and formatting.

{{‘Test’ | uppercase}} //TEST

App.component.html evendate="15 march";

App.component.ts

<p>The event will take place on {{ evendate | date:'medium' }}</p>

Diagram

Description automatically generated

Parameterized Pipes: we can any numbers of parameter using (:).

Chaining Pipes: we can use multiple pipes with the same data.

- You can chain pipes by using the pipe operator | between two or more pipes. The output of the 6first pipe is passed as input to the second pipe and so on.

data | pipe1: parameter1:parameter2 | pipe2:parameter1:parameter2 | pipe3

<p> {{ dob | date: 'dd/mm/yyyy'| uppercase }}</p>

**Pipes with String:** uppercase | lowercase | titlecase | Slice: starting index number | Slice: starting index number: end index number.

Date|shortdate|longdate|fulldate

<p> {{ c| percent}}</p><p> {{ c| percent:'2.4-8'}}</p>

Percent format:’2.4-8’(digit .dot then slash-)

<p> {{ c| number}}</p><p> {{ c| number:'2.4-8'}}</p>

decimal format: |number ’2.4-8’(digit .dot then slash-), for decimal use number pipe.

Here 2 is the integer number digits comes before decimal then dot 4 is min digit and 8 is max digit comes after decimal.

<p> {{ c| currency:'INR'}}</p><p> {{ c| currency:'INR':'code'}}</p>

**Json pipes:** | to convert data into json otherwise only object will show.

**Custom Pipes: have to apply @Pipe decorator to class which we can import from core library.**

Create pipe in terminal **ng g pipe pipename**

Create pipe in terminal **ng g pipe pipename --flat to make pipe in same folder**

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

@Pipe({name: 'myypipe'})

export class MyypipePipe implements PipeTransform {

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

return null;}}

And import into app.module file and registered into declaration array

import { MypipePipe } from './mypipe.pipe';

@NgModule({

declarations: [AppComponent,

MypipePipe, ],

**Routing-** a-Static Routing and b-dynamic routing.

**a-Static Routing**

**1-Give path-**In App.routing.module.ts**-** Create path in routing file in const routes: Routes = [ ]; and add which component you want to go with {path: ‘keyword’, component:componentName}

-After adding component that component will get imported automatically.

const routes: Routes = [

{path:'student',component:StudentComponent},

{path:'studentdetails',component:StudentdetailsComponent}];

**2**-Add Router Outlet-Then go to app.component.html and add router outlet element. This directive tells Angular where to render the content of each route. **<router-outlet></router-outlet>**

**3-Create Link**-Then create link with the help of anchor tag and now the components are routed.

**<a [routerLink]="['/student']">STUDENT</a> or <a routerLink="student">STUDENT 1</a>**

<a [routerLink]="['/student']">STUDENT 1</a

<a [routerLink]="['/studentdetails']">Details 1</a>

<router-outlet></router-outlet>

**-routerLinkActive=” active current**” is a directive in Angular that is used to add a CSS class to an element when the link associated with it becomes active.

<li routerLinkActive="active">

<a routerLink="/home">Home</a></li>

<li routerLinkActive="active bold"></li>

<li routerLinkActive="active current bold"></li>

**Child Route:** nested route

{path:”a ”, children:[ path:”a/details”, component:” “]}

<a [routerLink]="['/a/details']">Child Route</a>

**Or**

<button routerLink="/a/details/{{a.details}}">Show details</button>

**b-Dynamic Routing-**

all the above steps would be the same only we have to create a function to make this routing dynamic or server-side.

**4-**Create a function(mala) which will get triggered on any event(click) in app.component.html

<button (click)="mala()">Dynamic Routing</button>

**5-**define that function in app.component.ts file after importing Router from @a/router

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

import { Router } from '@angular/router';

export class AppComponent {

constructor(private router:Router){}

mala(){this.router.navigate(['student']);} }

In app.component.ts file import router from a/router and use this router into constructor(private router:Router){} and navigate through this router to particular component.

Now if you click that button you will navigate to student component.

**Redirection-**with the help of redirectTo

const routes: Routes = [

{ path: 'user/:id', redirectTo: '/profile/:id' }];

**Passing Fragment to route:**(also k/a a hash fragment) is a piece of the URL that comes after the "#" character. e.g., locatlhost:4200/Products#service

In the context of URLs, the fragment identifier is used to point to a specific section or element within a web page.

const routes: Routes = [

{ path: 'blog', component: BlogComponent, fragment: 'comments' },];

To access the fragment in your component, use the "ActivatedRoute" service like this:

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

import { ActivatedRoute } from '@angular/router';

@Component({

selector: 'app-blog',

templateUrl: './blog.component.html',

styleUrls: ['./blog.component.css']})

export class BlogComponent implements OnInit {

fragment: string;

constructor(private route: ActivatedRoute) { }

ngOnInit() {

this.route.fragment.subscribe(fragment => {

this.fragment = fragment;

});}}

**Wildcard route-**is a special route that matches any URL path that **does not** match any of the defined routes in your application. You can define a wildcard route using the \*\* path segment.

**Note:** always define in the last route o/w all other routes will directed to wildcard route.

Ex-page not found. Make a component and give any message in that component html file and define route path in app.routing.module.ts

{path:'\*\*',component:PagenotfoundComponent}];

If any routing happens apart from given routes, then it will go to render page not found component.

**Why Route parameter are Observable**: Route parameters in Angular are observable because they can change over time. When a user navigates to a different route or updates the current URL parameters, the route parameters may change, and the Angular application needs to be aware of these changes to update the corresponding components or services.

Observables in Angular provide a way to handle asynchronous data streams, which is exactly what happens with route parameters. Whenever a user navigates to a route with parameters, Angular creates an observable stream that emits values whenever the parameters change.

By **subscribing** to the route parameters observable, a component or service can react to changes in the route parameters and update its internal state accordingly. This makes it easy to keep track of the current state of the application and respond to user interactions in real-time.

**Observable:** are used to handle asynchronous data and perform asynchronous operations and whoever **subscribe** that observable will get notified and get that data. Basically, handle asyn operation operations like handling user input, making HTTP requests, and handling events.

- observables can be created using the RxJS library,

- can be subscribed to using the **subscribe ()** method, which allows you to receive data emitted by the observable and handle errors.

**Use of Bootstrap in Angular-**

1-insatll bootstrap by npm install bootstrap --save

2- go to angular.json file and add bootstrap into styles by

"styles": ["src/styles.css",

"node\_modules/bootstrap/dist/css/bootstrap.min.css"],

**Route Guard (refer notes)**

Is a mechanism that allow to control access to certain routes in app based on certain condition.

Its is used for authentication and authorization.

5 types of Route Guard in angular

**1-CanActivate:** This guard is used to determine if a user is allowed to navigate to a particular route. It returns a boolean or an Observable/Promise that resolves to a boolean. If the result is true, the user is allowed to navigate to the route. If it's false, the user is redirected to another route or shown an error message.

CanActivate method takes 2 parameters:

a-ActivatedRouteSnapshot b-RouterStateSnapshot

and returns Boolean Value / Observable Boolean Value / Promise Boolean Value

**2-CanDeactivate:** This guard is used to determine if a user is allowed to leave a particular route and allow to prevent a user from leaving a route with unsaved changes means ask permission before leaving that page.

It's applied to the current route and returns a boolean or an Observable/Promise that resolves to a boolean. If the result is true, the user is allowed to leave the route. If it's false, the user is shown a confirmation dialog or redirected to another route.

CanDeActivate method takes 3 parameters:

a-Component where you want to use this guard.

b-current route means ActivatedRouteSnapshot

c-current state means RouterStateSnapshot

**3-CanActivateChild:** This guard is similar to CanActivate, but it's used to protect child routes. It's applied to the parent route and checks if any of its child routes can be activated.

**4-Canload:**

**5-Resolve:** This guard is used to fetch data for a route before it's activated. It returns an Observable/Promise that resolves to the data that the route needs. Once the data is available, the route is activated, and the data is passed to the component.

Implementation of Route Guard:

1-Create a route guard as a service.

2-Implement route guard interface in that service.

3-Implement the route guard method in that service.

4-Provide guard service to app.module.ts as in provider array.

5-Use that route guard on the route which you want to guard.

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

import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree } from '@angular/router';

import { Observable } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class MyGuard implements CanActivate {

canActivate(

next: ActivatedRouteSnapshot,

state: RouterStateSnapshot): boolean | Observable<boolean> | Promise<boolean> {

// Logic to decide whether the user is allowed to access the route or not

// Return true if the user is allowed to access the route, false otherwise

return true;}}

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

import { Routes, RouterModule } from '@angular/router';

import { MyComponent } from './my.component';

import { MyGuard } from './my.guard';

const routes: Routes = [

{ path: 'my-path', component: MyComponent, canActivate: [MyGuard] }];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]})

export class AppRoutingModule { }

**Service**

It allows you to create reusable code that can be shared across multiple components or modules in your application.

If we use service in module level provider, then it become singleton object as it creates single object and if in different components it become multiple objects.

**Singleton**-Single instance of given object exist in memory and pass this single instance to all your components. (Service shared in as a instances).

Service is injected into application using dependency injection mechanism.

At component level we can provide a service in 2 ways-

1-Via providers array

2-Via viewproviders array

-To call any components constructor we have to create object for that constructor first.

-In module we can use data from other component with the help of Provider and only module have Provider but if we want to use a component or class or service in particular component then we have to use viewProviders in that component as we don’t have providers in components.

And we can restrict the usage of service or component by using that particular service or component if only that particular component uses.

@Component({

selector: 'app-root',

template: '<h1>{{title}}</h1>',

viewProviders: [MyService]})

export class AppComponent {

title: string;

constructor(private myService: MyService) {

this.title = myService.getTitle();}}

**providers Vs viewProviders-**providers sharing instances of services across a component and all its child components while viewProviders is used for creating instances of services that are specific to a component and its view children, with each instance having its own narrow scope.

**How to create Service:**

1-create a new file using **ng g service servicename** and 2 files createdmyservice.service.ts and myservice.service.spec.ts

2-New ts file automatically import Injectable decorator from core.

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

@Injectable({ providedIn: 'root'})

export class MyserviceService {

constructor() { }}

**4-**Define the functionality of your service by adding methods and properties to the service class and export it and import wherever you want

export class MyserviceService {

private data: any[];

constructor() {

this.data = [];}

addData(item: any) {

this.data.push(item);}

getData() {

return this.data;}}

**5-**Import/Use the service in your components by injecting it in the constructor and pass as an parameter in constructor

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

import { MyService } from './my-service.service';

@Component({

selector: 'app-my-component',

template: `{{ data }}`,

providers:[Myservice]

})

export class MyComponent {

data: any[];

constructor(private myService: MyService) {

this.data = this.myService.getData();}}

6-Add it in as provider on component file or module file depend on use case in @ component we have to make providers property.

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']})

Module file:

providers: [],

bootstrap: [AppComponent]})

export class AppModule { }

7-inject in constructor argument through di constructor (private\_example:Example){}

8-using the service with the referencing name given in constructor like \_example.somemethod

9-bind the value in html file

Inshort

1-create service and add as a parameter in the constructor of the component class where you want that service.

2-registerd service as a provider in that component or module use case.

When angular is going to create an instance of this component, first it will instantiate its dependencies and then it will inject those dependencies into the constructor of its class. This is called as **Dependency Injection (DI).**

**Dependency Injection (DI)** is a software design pattern where we can use the one functionality or service of an object to another component, means other component is dependent on first component for functionality and first component is injected to other component.

A 25-dollar term for 5 cent concepts.

DI means injecting or providing the dependency of a class into its constructor.

Dependency=Service

Injection= to add or injected

**@Injectable decorator:**

**Note:** It is not necessary to use injectable decorator to make service, Injectable decorator inside any service is not meant to make service. We can use service w/o injectable decorator bcz service is just like function of component class and implement the functionality. We use injectable decorator inside service to use/inject other service/ dependency in that service. (Nested service) Means to inject one service to another service.

**provideIn:** comes in angular-6.

As we know that Service load first then Module load then component load over browser then app will start.

In app.module all the import load first then app module will load.

Service generated by angular/cli will automatically be created with @Injectable decorator along with a new property called provideIn. ProviedIn instruct the application where to provide the service.

-allows a service to be provided at a specific level of the application's injector hierarchy.

-By default, services in Angular are provided at the root level of the application injector hierarchy. This means that the service is available to the entire application, including all components and modules. However, sometimes you may want to limit the scope of a service to a specific module or component. This is where the provideIn feature comes in.

-It helps to optimize application's performance by reducing the number of services that are loaded at runtime as we have to import specific service into app.module to provide service in all the modules and components, and service will be used in all components. What if we need to use service in specific components in that case, we have to use provideIn and load when that particular component get in use only.

provideIn als k/a tree shakable provider and meant for eliminate unused code.

**Tree Shakeable Provider (TSP) : provideIn :** it’s a feature in angular toremove unused code during the build process, which can help to reduce the size of the final application bundle.

Tree shaking is a process in which the compiler analyzes the application's code and identifies which parts of the code are actually used at runtime. Any code that is not used (e.g., functions, variables, or entire modules) is removed from the final application bundle.

In order for tree shaking to work effectively, your code must be structured in a way that allows the compiler to determine which parts of the code are actually used. This means that:

1-All of your code must be written in a modular format. This means that each module should be self-contained and should only include the code that is required to perform its specific tasks.

2-All of your code must be written using ES6 (ECMAScript 2015) syntax. This is because ES6 provides a set of features that make it easier for the compiler to analyze and optimize your code.

3-You must avoid using dynamic imports. Dynamic imports make it difficult for the compiler to determine which parts of your code are actually used, since the code that is imported is not known until runtime.

When you use the Angular CLI to build your application, tree shaking is automatically enabled by default, but we can configure manually also to optimize your application's performance with the help of **Webpack.**

**Webpack** is build in automation tool, it gets all our scripts and stylesheets, combine them, puts them in a bundle and then minifies that bundle for optimization. Will do

1-Code splitting:

2-Tree shaking:

3-Asset optimization: webpack can optimize assets such as images and fonts by compressing them and generating different versions for different browsers.

Below in the element inspect all the bundles are injected into scripts.

<script src="runtime.js" type="module"></script>

<script src="polyfills.js" type="module"></script>

<script src="styles.js" defer=""></script>

<script src="vendor.js" type="module"></script>

<script src="main.js" type="module"></script>

**Hot Module Replacement/ Reloading (HMR)** is a feature of webpack whenever the source files is modified web pack automatically refreshes your browser.

npm install @angularclass/hmr --save-dev

**Forms**

Angular forms are a key feature of the Angular framework, which allows developers to create and manage user input in web applications. Angular provides two types of forms: template-driven forms and reactive forms.

Template-driven forms are created using Angular directives within the HTML template of a component. These forms rely on two-way data binding and can be created quickly, but they have limited functionality and can be difficult to manage as the form grows in complexity.

Reactive forms, on the other hand, are created programmatically using TypeScript classes that define the form model and validation rules. Reactive forms provide more control and flexibility over the form's behavior but require more code to implement.

Both types of forms allow developers to:

-Bind input fields to component properties,

-Validate user input,

-Track user interaction with the form

-Display error messages to users

To use either type of form in an Angular application, you must import the FormsModule or ReactiveFormsModule from @angular/forms and add them to the imports array in your application module.

2 different ways to collect and validate data from user.

**1-Template-driven forms**, you define the form controls in the template and bind them to properties in the component class, means everything that we are going to use in an app is defined into the template.

**NgForm** is a built-in directive that is used **to register formcontrols** with a form and to track the state and validity of the form. When a form is submitted, the NgForm directive can be used to access the values of the formcontrols and to perform any required actions.

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

Form registered with #regForm variable and assigned to the ngForm where all the inputs control are grouped and with Register function which we call in ts.file, we can get all the value of control.

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

import { Router } from '@angular/router';

import { NgForm } from '@angular/forms';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']})

export class AppComponent {

title = 'my-angular-app4';

mesaage="type";

Register(regForm:NgForm){console.log(regForm);}}

We are exporting the ngForm value to the local variable regForm. We are exporting local variable just to use some attached properties:

regForm.value-gives object containing all the values of the field on the form.

regForm.valid-give validity of the value true for valid value else false.

regForm.touched-gives ture and false. On touched of controls.

When a user interacts with a form control, such as by clicking on it or entering a value, the control is considered "touched".

**ngModel** is a built-in directive in Angular that is used for two-way data binding between a form control and a component property.

Prerequisite we need to 1-import FormsModule in app.module.ts from ‘@angular/forms’

Then register it into imports array.

import { FormsModule } from '@angular/forms';

@NgModule({

imports: [FormsModule],...})

2-Add form controls to your template ngForm and ng Model

3-Handle form submission in your component with function.

ngForm create the control grouping of all the element where ngModel is written and ngModel should have name property and register all the inputs in the ngForm.

Advantages: easy to use, simple scenarios, two-way binding using [(NgMode)l] syntax, minimum component code, Automatic track of the form and its data, etc.

Disadvantages: unit testing, not suited for complex scenarios.

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

<input type="text" name="firstname" ngModel>

<input type="text" name="lastname" ngModel>

<input type="text" name="Email" ngModel>

<button type="submit">Register</button>

</form>

Ts file call function

Register(regForm:any){

var firstname=regForm.controls.firstname.value;

console.log(firstname);

console.log(regForm);}}

Text

Description automatically generated

**Validation in Template-driven forms**:

ng-touched: controls have been visited.

ng-untouched: controls have not been visited.

ng-dirty: controls value has been changed.

ng-pristine: controls value has not been changed.

ng-valid: control values are valid.

ng-invalid: control values are invalid.

Ex.input.ng-invalid.ng-touched{border-color:red;}

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

<input type="text" name="firstname"  ngModel>

<input type="text" name="lastname"  ngModel>

<input type="text" name="Email" required ngModel #email="ngModel">

<span \*ngIf="email.touched && !email.valid">Pls enter email</span>

<button type="submit" [disabled]="!regForm.valid">Register</button>

</form>

**2-Reactive forms: or Model driven forms-** at component level ts.file are responsible for user interaction.

Reactive forms are based on the reactive programming style, which emphasizes declarative and functional programming. Reactive forms allow you to create and manipulate form controls programmatically in the component class. Reactive forms are more powerful and flexible than template-driven forms, but they require more setup and configuration.

Prerequisite Need to import ReactiveFormsModule in app.module.ts

we need to create Model using inbuilt classes formGroup and formControl and then we need to bind the model to HTML form.

Advantages: Flexible, for complex scenarios, no data binding is done, more code and less html, less work on template more work on component, dynamic work easy, unit testing is easier.

If we want to detect any changes occurs in any variable its very difficult to detect by template driven form but in reactive form any variable changes, we can detect easily.

Disadvantages: more codes.

Reactive Forms in Angular is a method for creating and managing forms in a reactive way using FormControl, FormGroup, and FormArray classes, the ts file (form are manage at component level) responsible for handling the user interaction/validation. First, we have to create Model using inbuilt classes like FormGroup and FormControl and then we need to bind that model to the html form.

1-import ReactiveFormsModule in app.module.ts

import { ReactiveFormsModule } from '@angular/forms';

@NgModule({

imports: [ BrowserModule, ReactiveFormsModule ],

...})

export class AppModule { }

2-Import FormGroup, FormControl, FormBuilder, NgForm from `@angular/forms`; in component

import{FormControl,FormGroup,FormBuilder,NgForm} from '@angular/forms';

3-Define FormGroup in component

FormGroup is an AbstractControl class take 4 parameters an object {property: value} out of which 1-2 are mandatory 2 (3-4 )are optional.

1-property or key and should be string only

2-value is an AbstractControl Class.

3-validator?:validationFn

4-asyncValidator?:AsyncValidatorFn

Ex-**FormGroup( { key=string:AbstarctControl Class()}**

**validator?:validationFn,**

**asyncValidator?:AsyncValidatorFn)**

export class SignupformCopomnentComponent {

Sform= new FormGroup(

{username:new FormControl(),

password:new FormControl(),

});}

<form [formGroup]="Sform">

<input type="text" formControlName="username">

<input type="text" formControlName="password">

</form>

**Note-AbstractControl class** in Object Oriented programing we have a concept of Inheritance and if you have multiple classes that should have some common behavior and properties, instead of implementing this common behavior and common properties in multiple places, we define them once in a parent or base class, and then have those other classes inherit these properties and behavior from their base class.

So, in Angular, AbstractControl is the base class for FormControl and FormGroup. All the properties that are common between these two classes are defined in the AbstractControl class. So, these classes simply inherent these properties from their parent.

So back here, as the first argument here, we need to pass an object, and these objects should have one or more key value pairs, keys should be string and values should be AbstractControl objects. So, we add an object, here we need two key value pairs. One for the username, and one for password. So, username, that's a string, and the value here, should be an instance of a class that derives from AbstractControl. So, we can either add a FormControl object here, or if you're dealing with a complex form that includes subgroups, we need to set this to a FormGroup object. And this FormGroup will then have one or more FormControl objects, okay? So, here we need a FormControl,

1-import ReactiveFormsModule in app.module.ts

import { ReactiveFormsModule } from '@angular/forms';

@NgModule({

imports: [ BrowserModule, ReactiveFormsModule ],

...})

export class AppModule { }

2-Define a FormGroup object in your app.component.ts

import{FormControl,FormGroup,FormBuilder,NgForm} from '@angular/forms';

Now on app.component.ts we first need to add form classes and then we need to define a function to detect if any html element changed or not.

ngOnlint(){this.form=new FormGroup({ firstname: new FormControl(“”),

lastname: new FormControl(“”),

email: new FormControl(“”),})}

FormControl- track and validate the value of controls. (for Individual input value in the form)

FormGroup: collection of FormControl which track the validity and state of the group of FormControl instance. (for Individual complete form value in the form)

FomrBuilder: to develop the forms along with their initial value and their validation.

3-define the form controls with their initial values.

export class AppComponent {

title = 'Reactive Form';

signupForm:FormGroup;

FirstName:string="";

LastName:string="";

Email:string="";

Password:string="";

constructor(private frmbuilder:FormBuilder){

this.signupForm=frmbuilder.group({

fname:new FormControl(),

lname:new FormControl(),

emailname:new FormControl(),

passwprdname:new FormControl(), });};

4-

Html file

<form [formGroup]='signupForm' (ngSubmit)="PostData(signupForm)">

<input type="text" formControlName='fname'><br>

<input type="text" formControlName='lname'><br>

<input type="text" formControlName='emailname'><br>

<input type="text" formControlName='passwprdname'><br><br><br>

<input type="submit" value="PostData">

</form>

**Validation in Reactive forms**:

1-Import Validators from form.

import { FormGroup, FormControl, Validators } from '@angular/forms';

constructor(private frmbuilder:FormBuilder){

this.signupForm=frmbuilder.group({

fname:new FormControl(),

lname:new FormControl(),

emailname:new FormControl(),

passwprdname:new FormControl(),   }); };

The first argument of new FormControl() is FormControl initial value represented in ‘ ’ and 2nd is validation for more validation in one FormControl we can use compose Validators.

constructor(private frmbuilder:FormBuilder){

this.signupForm=frmbuilder.group({

fname: ['',Validators.required],

lname: ['',Validators.required,Validators.maxLength(10)],

emailname: ['',Validators.required,Validators.email],

passwprdname: ['',Validators.required],  })

**Getvalue and reset value in Reactive Forms:**

PostData(signupForm:any){this.FirstName=this.signupForm.get('fname')?.value;

const nick=this.signupForm.get('lname')?.value;

console.log(nick);}

**To Reset:**

<input type="reset" value="reset">

<button type="reset">Reset again</button>

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

reset(){this.signupForm.reset();}

resetForm(){this.signupForm.reset();}

to reset particular value

resetForm(){this.signupForm.reset({fname:'Mohan',});}

**setvalue() and patchvalue() in Reactive forms:** both methods sets the value in the form control of FormGroup.

setValue() sets the value in each and every form control of FormGroup. We **cannot omit** any form control in setvalue() but when we want to assign only few from controls of the formGroup then we need to use patchValue().

It is necessary to mention all the form control in setValue() otherwise it will through error.

But when we use patchValue()then its not necessary to mention all the form controls.

**Note:** the setValue() method will throw an error if you try to set the value of a disabled control or a control that doesn't exist in the form group. In such cases, you should use the patchValue() method instead, which is similar to setValue() but will only update the values of controls that are present in the provided object.

**Note: Get, Post, Delete, Put and Patch method of http request.**

**Where Put and Patch method update the sent**

<button type="button" (click)="filldata()">Set Value fill data automatically</button><br>

filldata(){this.signupForm.setValue({

'fname': 'Shiv',

'lname':'Sadan',

'emailname':'ss@gmail.com',

'passwprdname':'1234',})}

By clicking Set Value fill data automatically data will get filled in input box automatically. All forms controls to be needed.

By clicking Patch Value fill data automatically data will get filled in input box automatically w/o all form control.

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text

Description automatically generated

**valuechanges and statuschange:** are the properties of FormControl, FormArray and FormGroup classes.

**valuechanges and statuschange** property both returns an Observable instance that emits events whenever the value of the control(s) changes and we can **subscribe** them to get the data.

**valuechanges** property help to get the data whenever changes the input at a time. We can see that it returns Observable of any type, and we can subscribe valueChanges to get the data.

If we subscribe valueChanges of a FormControl /FormArray /FromGroup instance, we get latest value of that control/ those array/ form controls whenever there is any change of the form.

**statuschange:** is same as valuechanges butwhen the status is changes AFTER validation check.

Text

Description automatically generated with medium confidence

**valuechanges:** for particular form control

ngOnInit(){ this.signupForm.get('fname')?.valueChanges.subscribe(uname=>{console.log('fname changed'+ uname);})}

**valuechanges:** for form group control

ngOnInit(){

this.signupForm.get('fname')?.valueChanges.subscribe(uname=>{console.log('fname changed'+ uname);});

this.signupForm.get('lname')?.valueChanges.subscribe(uname=>{console.log('lname changed'+ uname);});

this.signupForm.get('emailname')?.valueChanges.subscribe(uname=>{console.log('emailname changed'+ uname);});

this.signupForm.get('passwprdname')?.valueChanges.subscribe(uname=>{console.log('passwprdname'+ uname);});

}

Graphical user interface, application

Description automatically generated

**statuschanges:** for particular form control

ngOnInit(){

this.signupForm.get('fname')?.statusChanges.subscribe(uname=>{console.log('fname changed'+ uname);});

A picture containing background pattern

Description automatically generated

**Asynchronous validation:**

Asynchronous validators in Angular are functions that are used to validate a form control based on an asynchronous operation, such as a server request. These validators are defined as part of the form control's validation rules and are **executed** asynchronously when the control's value changes.

To define an asynchronous validator in Angular, you need to create a function that returns a Promise or an Observable. The function should take the form control as an argument and should resolve with null if the validation is successful or an error object if the validation fails. Here is an example of an asynchronous validator that checks if a username is available:

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

import { FormControl } from '@angular/forms';

import { Observable } from 'rxjs/Observable';

@Injectable({ providedIn: 'root' })

export class UsernameValidator {

static checkUsername(control: FormControl): Promise<any> | Observable<any> {

const username = control.value;

// simulate server request

return new Observable(observer => {

setTimeout(() => {

if (username === 'john') {

observer.next({ usernameTaken: true });

} else {

observer.next(null);}

observer.complete();

}, 2000);});}}

In this example, the checkUsername function returns an Observable that simulates a server request to check if the username is available. If the username is taken, the validator returns an error object with a usernameTaken key. Otherwise, it returns null to indicate that the validation was successful.

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

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

import { UsernameValidator } from './username.validator';

@Component({

selector: 'app-root',

template: `

<form [formGroup]="form">

<input type="text" formControlName="username">

<div \*ngIf="form.get('username').hasError('usernameTaken')">

Username is already taken </div>

</form>`})

export class AppComponent {

form: FormGroup;

constructor(private fb: FormBuilder) {

this.form = this.fb.group({

username: ['', Validators.required, UsernameValidator.checkUsername]

});}}

In this example, the checkUsername validator is added to the username control's validation rules using the Validators.required function. When the user types in the username input, the validator is executed asynchronously and displays an error message if the username is taken.

**Ex-2-**Async validator is very similar to creating a sync validator the only difference is that async validator must return either a promise or an observable and note that angular does not provide any built-in async validators. We use async validator when we need to send http request to the server to check if the data is valid, so when we send an http request to the server the server might take some time in sending the response so we need to wait for that time and when the data is available then only it should be validated for that we use async validator.

Use case in reactive form, what we want is in this email control to restrict user to use procademy.gmail.com in email address input so when the user tries to register with this email address, we want to restrict him. For that we are going to create a custom validator and this custom validator is going to be an async custom validator.

Go to app component class and create a method as we know that a validator is nothing but a method, let's call this method emailNotAllowed and this method is going to receive a parameter (control: Form Control) control and type formControl. So, on whichever formControl we will use this validator emailNotAllowed we will receive that formControl as an argument, and it will be stored inside this control parameter while defining.

When we create an async validator it must return either a promise or an observable, so here we also need to specify the type of the value which this method is going to return so it is either going to return a promise of any type or it is going to return an observable of any type and to use this observable we also need to import it from rxjs library.

Now inside this method let's create a variable (const liya) (or this.control.value) and assign it into a new promise because here we are going to create an instance of this promise and, promise takes a callback function and this callback function receives two parameters the resolve callback function and reject callback function.

emailNotAllowed(control:FormControl):Promise<any>|Observable<any>{

const liya=new Promise((resolve,reject)=>{

setTimeout(()=>{

if(control.value==='pro@gmail.com'){

resolve({emailnotallowedsofar:true})

}else{resolve(null)}},5000);

return Response;})

}

Inside the body of this callback function let's use setTimeout function so basically here we want to return some data after a time interval just to simulate the asynchronous behavior, the first argument of the set timeout function is a callback function and then the second argument is the time interval.

So for time interval 5000 milliseconds now inside the callback function of the set timeout let's use if statement, so here we want to check if this control has a value equal to procademy gmail.com, if control.value is equal to procademy gmail.com in that case we want to return a validation error so for that let's use resolve method and when this promise will be resolved we want to return an error code, so let's create that error code emailnotallowed:true , so this is the error code which we want to return if in the email field we enter this value procademy gmail.com otherwise if the user enter any other value apart from this email address in that case we want to make this promise resolve with the value **null** so again in the else part let's call this resolve method again and from here we want to return null so if the user enters this email address in the email field this email not allowed validator will return this error code and you can see this error code in the error object of email control but if the user has entered any other value apart from this email in that case this null will be returned and the error object of the email control will be assigned with this value null, let's return this promise from this emailnotallowed method so let's say return response.

In this way we have created an async validator now let's use this validator

email: new FormControl['', [Validators.required, Validators.email], emailNotAllowed]});

Use this validator on the email control, now on this email control we are already using this required validator and email validator, and this required, and email validators are sync validators now when we want to use an async validator on a control we can pass it as the third argument of this form control constructor.

so for this form control constructor the first argument is this value null the second argument is this array inside which we are specifying the sync validators and then for the third argument we can specify an async validator this.emailNotAllowed.

Let's go to the web page let me open developer console, go to element tab expand this form and let's go to email control so here we have this email control now on this email control currently we have these classes ng untouched ng pristine and ng invalid now let's go ahead and let's enter some email field here so let me select this email xyz@gmail.com, you will notice this ng pending you know css class added on this email control and after five seconds that ng pending changed to ng valid.

If we enter procademy gmail.com so again you will notice here this ng pending has been added on this email control and after five seconds it has changed to ng invalid so since this email not allowed is an async validator it is going to return this email not allowed error code or this value null after 5000 milliseconds so for 5000 milliseconds we are seeing the ng pending on the email field and after five seconds when the control is validated it is either displaying ng valid or ng invalid based on the value which we are entering in this email control.

if we are entering this email address in that case that ng pending after five seconds it will change to ng invalid because this validator this email not allowed validator it will not allow user to use this email address in the email field okay let's see this one more time so let me refresh the page let's go to this email control so it is inside this form group and here we have this email control so currently you will notice it has this ng untouched ng pristine and ng invalid now let me go ahead and enter an email address apart from proacademy gmail.com as soon as i enter this email for 5 seconds you will see ng pending and after 5 seconds it will change to ng valid because this email address is valid now if i go ahead and enter procademy gmail.com again for the 5 seconds you will see ng pending and after five seconds it will change to ng invalid because in the email field this email address is not allowed.

**Nested Form Groups: with the help of FormGroupName**

mport { Component } from '@angular/core';

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({

selector: 'app-my-form',

template: `

<form [formGroup]="myForm" (ngSubmit)="onSubmit()">

<fieldset formGroupName="personalInfo">

<legend>Personal Information</legend>

<label for="firstName">First Name:</label>

<input type="text" id="firstName" formControlName="firstName">

<label for="lastName">Last Name:</label>

<input type="text" id="lastName" formControlName="lastName">

</fieldset>

<fieldset formGroupName="address">

<legend>Address</legend>

<label for="street">Street:</label>

<input type="text" id="street" formControlName="street">

<label for="city">City:</label>

<input type="text" id="city" formControlName="city">

<label for="state">State:</label>

<input type="text" id="state" formControlName="state">

<label for="zip">Zip Code:</label>

<input type="text" id="zip" formControlName="zip">

</fieldset>

<button type="submit">Submit</button>

</form>

`

})

export class MyFormComponent {

myForm: FormGroup;

constructor(private fb: FormBuilder) {

this.myForm = this.fb.group({

personalInfo: this.fb.group({

firstName: ['', Validators.required],

lastName: ['', Validators.required]

}),

address: this.fb.group({

street: [''],

city: [''],

state: [''],

zip: ['']})});}

onSubmit() {

console.log(this.myForm.value);}}

In this example, we create a parent form group called myForm and two nested form groups called personalInfo and address. Each nested form group contains its own set of form controls. We can reference the nested form groups using the formGroupName attribute in the HTML template

**FormArray in Reactive Forms:** is a class that represents an array of form controls or form groups. It is used to manage a dynamic set of form controls and allows you to add or remove form controls at runtime.

Add/remove element or DOM interaction in reactive form using FormArray.

Set of controls which are group together. It may be group of FormsControls, FormGroup, FormArray, or combination of all.

-it tracks the value and validity state of array of formcontrol. (Check both value and status at the same time).

First have to import {FormArray} from ‘@angular/form’

import{FormControl,FormGroup,FormBuilder,NgForm,Validators,FormArray} from '@angular/forms';

ngOnInit(){

const arr=new FormArray([

new FormControl('RAM'),

new FormControl('SHYAM'),]);

console.log(arr.value);

console.log(arr.status);}

ngOnInit(){

const arr=new FormArray([

new FormControl('RAM'),

new FormControl('SHYAM'),

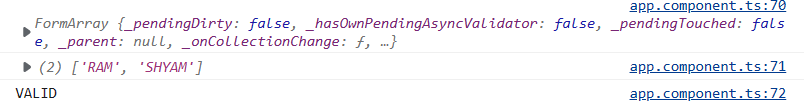
]);

console.log(arr);

console.log(arr.value);

console.log(arr.status);}

check variable arr showed FormArray, its value is in array[RAM,SHYAM] and status showing valid:



.

[(1) Angular 10 Tutorial #69 - Reactive Forms - FormArray | Angular 10 Tutorial For Beginners - YouTube](https://www.youtube.com/watch?v=9BJa6u_Gs1w)

**Nested FormArray in reactive forms:**

**Notes:** Template driven vs Reactive form: -Template driven form are controlled by html file bcz of ngForm and ngModel are there only while reactive from is controlled by ts file.

Reactive form is managed at component level.

-If we want to detect any changes occurs in any variable its very difficult to detect by template driven form but in reactive form any variable changes, we can detect easily.

**-AbstractControl class** in Object Oriented programing we have a concept of Inheritance and if you have multiple classes that should have some common behavior and properties, instead of implementing this common behavior and common properties in multiple places, we define them once in a parent or base class, and then have those other classes inherit these properties and behavior from their base class.

So, in Angular, AbstractControl is the base class for FormControl and FormGroup. All the properties that are common between these two classes are actually defined in the AbstractControl class. So, these classes simply inherent these properties from their parent.

So, in total an abstract base class that provides a common set of properties and methods for working with form controls and form groups. It is the superclass of FormControl, FormGroup, and FormArray.

Here are some of the key properties and methods of the AbstractControl class:

**value:** Gets or sets the value of the control.

**status:** Gets the validation status of the control (e.g., 'VALID', 'INVALID', 'PENDING').

**valid:** Gets whether the control is valid.

**invalid:** Gets whether the control is invalid.

**touched:** Gets whether the control has been touched.

**untouched:** Gets whether the control has not been touched.

**dirty:** Gets whether the control has been changed.

**pristine:** Gets whether the control has not been changed.

**errors:** Gets any validation errors associated with the control.

**valueChanges:** An Observable that emits the current value of the control whenever it changes.

**statusChanges:** An Observable that emits the current validation status of the control whenever it changes.

**setValidators():** Adds or replaces the validators for the control.

**setAsyncValidators():** Adds or replaces the async validators for the control.

**markAsTouched():** Marks the control as touched.

**markAsUntouched():** Marks the control as untouched.

**markAsDirty():** Marks the control as dirty.

**markAsPristine():** Marks the control as pristine.

**updateValueAndValidity():** Recalculates the value and validation status of the control.

**Error Handling:**

**Global error handling** in Angular is the process of intercepting and handling errors that occur throughout an Angular application.

To implement global error handling, you can use inbuild Angular **ErrorHandler class** which provide a method called **handleError.**

1-Create a class that implements the ErrorHandler interface:

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

@Injectable()

export class AppErrorHandle implements ErrorHandler {

handleError(error: any): void {

// Handle the error

console.error(error);

}}

2-Register the ErrorHandler class in the app.module.ts file:

import { NgModule, ErrorHandler } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { AppErrorHandle } from './app-error-handle';

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule],

providers: [

{ provide: ErrorHandler, useClass: AppErrorHandle }

],

bootstrap: [AppComponent]

})

export class AppModule { }

Here in app.module.ts providers we should pass an object with 2 properties 1-provide with value of ErrorHandler, and 2-useClass: AppErrorHandle.

**Miscellaneous**

Service load first then Module load then component load over browser then app will start.

In app.module all the import load first then app module will load.

**Onlint:** Online linting is a process of checking the code for errors and style violations using an online tool or service. Linting helps to identify issues in the code before it is executed, ensuring that the code is of good quality and conforms to best practices. By using an online linting tool, you can quickly identify issues and improve the overall quality of your code.

Types of Onlint:

**1-TSLint-** static analysis tool for typescript code that check ts code for readability, maintainability, functionality errors. for TypeScript that provides rules for static analysis of code. It checks the code for common errors and style violations, such as missing semicolons, unused variables, and improper indentation.

**2-ESLint-**used with JavaScript and TypeScript. It is highly configurable and provides many options for customizing the linting rules.

**3-Prettier-**is a code formatter that automatically formats the code based on predefined rules.

**4-Stylelint-**linter for CSS and Sass that checks for syntax errors, style violations, and browser compatibility issues.

**The <base>** tag is useful when you want to specify a base URL for all the links in a web page, rather than having to specify it individually for each link. By using this tag, all relative URLs in the document will be resolved relative to the specified base URL.

**<base href="/">-**the base URL is set to "/", which means that all relative URLs in the web page will be resolved relative to the root directory of the website.

**Single Page Application (SPA)-**Single page is downloaded from the server as the user navigate from one page to another only the content of the target page is downloaded.

**Life Cycle Hooks:** Components have life cycle hooks; these are the special methods that we can add to our component and Angular will automatically call these methods at specific time during the life cycle of the component to perform certain actions.

<https://www.youtube.com/watch?v=jFk9-zV27BE> https://www.youtube.com/watch?v=0Ywak7Ppszw

Life cycle stages: 1-Create Component, 2-Render Component, 3-Creates and Render Children Component, 4-Destroy a Component.

Most used lifecycle hooks in Angular:

**ngOnInit()-**is a lifecycle hook in Angular that is called once the component is initialized, after its inputs have been bound for the first time.

It's important to note that ngOnInit() is called only once, when the component is first initialized. If you need to perform logic each time the component's inputs change, you can use the ngOnChanges() lifecycle hook instead.

**ngOnChanges:** This is called whenever an input property of the component changes. The hook provides information about the changes through the SimpleChanges object.

**ngDoCheck:** This is called during every change detection cycle and is used to detect changes that Angular doesn't detect automatically.

**ngAfterViewInit:** This is called after the component's view has been initialized. This is the ideal place to put logic that needs to interact with the view such as manipulating the DOM or initializing third-party libraries.

**ngAfterContentInit:** This is called after the content of the component has been initialized. This is the ideal place to put initialization logic that interacts with projected content such as initializing child components or manipulating the content.

**ngOnDestroy:** called when the component is about to be destroyed. This is the ideal place to put cleanup logic such as **unsubscribing** from observables, releasing resources, or cancelling timers.

Ctrl key + `(backtick)=Open terminal

**DOM Vs HTML**

**DOM=**itsmodel of object that represent the structure of the document its essentially a tree of objects in memory.

**Html=markup language that represent dom in text**

Whenever browser parses an html document it creates tree of objects in memory that they refer to as dom. With the help of Js we can manipulate don programmatically.

Most of the attributes of html elements have one to one mapping to the properties of DOM objects, but few exceptions are there like colspan is html attribute while text context is dom attribute.

**Safe traversal operator or Optional chaining operator (?)-**to ignore null or undefined value.

const person = {

name: 'John',

age: 30,

address: {

street: '123 Main St',

city: 'Anytown',

state: 'CA',

zip: '12345'}};

const street = person.address.street; wiil show error

const street = person.address?.street;

In this example, the address property of the person object is optional, and may be null or undefined. The safe traversal operator is used in the second line to access the street property of the address object only if it is not null or undefined. If the address object is null or undefined, the value of street will also be null or undefined, without causing an error.

**ElementRef -** is a class that provides access to the DOM element of a component.

When a component is rendered on the screen, it creates a view that consists of the component's template and its associated styles. The view is then attached to a DOM element in the browser's Document Object Model (DOM). The ElementRef class provides a reference to this DOM element, allowing you to interact with it directly from within your component.

import from @angular/core package and pass it into constructor as a parameter like service.

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

@Component({

selector: 'my-component',

template: '<div>Hello, {{name}}!</div>'})

export class MyComponent {

constructor(private elementRef: ElementRef) {

this.elementRef.nativeElement.style.backgroundColor = 'yellow';}

name = 'World';}

**Reactive programming** which allows us to manipulate **stream of data** using functions,focuses on reacting to changes in data, instead of explicitly defining the steps to transform the data.

**RxJS:** is a js library that allow us to work with asynchronous data stream.

**RxJS** (Reactive Extensions for JavaScript) is a library for reactive programming using Observables, which allows you to work with asynchronous data streams.

RxJS is widely used in front-end web development, particularly with frameworks such as Angular, React, and Vue. It can also be used in server-side JavaScript environments such as Node.js.

Some of the key concepts in RxJS include **Observables, Subscriptions, Operators, and Subjects.** **Observables** are the sources of data streams, while **subscriptions** allow you to subscribe to and handle the emitted data. **Operators** provide a way to transform and manipulate the data in the streams, while **Subjects** is a special type of observable for cross component communication to be multicasted to many observers

https://www.youtube.com/watch?v=V4iMyVnQPqM

Observables and Promise are used to handle asynchronous data and perform asynchronous operations.

A Promise provides all data at once, whereas Observable **streams the data** in packets means share data in chunks do not wait for the complete data.

A Promise will return the data even if there is no one to use it. However, Observable will return the data only when some one subscribes to it.

A Promise is native to JS. An observable is not a feature of JS or Angular but is provided by a 3rd party lib Rxjs.

An observable is a function which converts an ordinary stream of data into an Observable stream of data. Observable is a wrapper around ordinary stream of data. An Observer is something which uses the Observable data in order to use the data of Observable the Observer must subscribe to the Observer.

rxjs has two main players: **1-observable** which is the stream of data and **2-observer** which is going to use that data now in order to make this observer use the data emitted by this observable the observer has to subscribe to that observable we can also say that an observer is the subscriber of that observable.

In order to use this observable, we need to import rxjs library in our angular application and this rxjs library is installed automatically when we create a new angular project.

This observable constructor we need to pass a callback function and this callback function will receive an argument which will be the observer and this argument will be injected by rxjs library.

new observables have observable constructor where we need to pass a callback function and this callback function will receive an argument which will be the

**observer** and this observer argument will be injected byrxjs library,and this observer is nothing but thesubscriber which is waiting for the data**.**

Inside this callback function let's first log a message using console.log and let's say observable starts, then let's emit some data and to emit the data on this observer we can call next method and it will emit some data let's emit the string value one from here in the same way let's also emit other values so let me copy this line maybe five times

export class ObservableComponent {

myObservable = new Observable(observer => {

observer.next(1);

observer.next(2);

observer.next(3);

});

ngOnit(){

this.myObservable.subscribe((val)=>{console.log(val);})

}}

The subscribe method takes 3 optional parameters and these parameters are callback functions. 1-first callback function is **next,**

2-callback function is **error** and,

3-callback function is **complete**

and all these three parameters are optional.

the next parameter is a callback function which gets executed every time when the observer call next method and this next method will return some value.

Next method in the subscribe will get call by 3 times.

subscribing to this my observable this next callback function will be called 3 times because this observable is going to emit 3 data it is going to stream these 3 data.

let's specify a callback function using this arrow function syntax and this callback function is going to receive the data which the observable has returned or emitted, so in this val parameter we have the value which the observable has emitted now how do we want to use that value, let's log that value in the console.

//Output will be 1,2,3 data will be streamed here one by one.

**JSON Web Token (JWT)**

Install- npm install @auth0/angular-jwt --save

import { JwtHelperService } from '@auth0/angular-jwt';

const token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...';

const jwtHelper = new JwtHelperService();

const decodedToken = jwtHelper.decodeToken(token);

console.log(decodedToken);

import { JwtHelperService } from '@auth0/angular-jwt';

const token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...';

const jwtHelper = new JwtHelperService();

const isTokenExpired = jwtHelper.isTokenExpired(token);

if (isTokenExpired) {

console.log('Token is expired');} else {

console.log('Token is valid');}

We have some inbuild classes and methods in Jwt library to use for.

Like JwtHelperService is class having methods like decodeToken();, isTokenExpired();,

tokenNotExpired();,getTokenExpiredDate();, urlBase64Decode(); etc.

import { JwtHelperService } from '@auth0/angular-jwt';

Inject the JwtHelperService in the constructor of your component:

constructor(private jwtHelper: JwtHelperService) {}

const token = localStorage.getItem('token');

const decodedToken = this.jwtHelper.decodeToken(token);

console.log(decodedToken);

const token = localStorage.getItem('token');

const isExpired = this.jwtHelper.isTokenExpired(token);

console.log(isExpired);

const token = localStorage.getItem('token');

const expirationDate = this.jwtHelper.getTokenExpirationDate(token);

console.log(expirationDate);

**tokenNotExpired()** in-built method from the angular2-jwt library, which is used for verifying the expiration status of JSON Web Tokens (JWTs) in Angular applications.

**Local Storage:** is a feature of modern web browsers that allows web applications to store data locally on the user's computer and we can use with localStorage object. **Key-value pair.**

Local Storage Methods:

1-setItem (); to set a key-value pair in local storage. The key parameter is a string that represents the key for the data, and the value parameter is a string that represents the data to store.

setItem(key: string, value: string): void

localStorage.setItem('username', 'john.doe');

2-getItem(); to retrieve a value from local storage based on its key. If the key is not found, this method returns null. getItem(key: string): string | null

const username = localStorage.getItem('username');

**3-removeItem ();** to remove a key-value pair from local storage based on its key.

removeItem(key: string): void;

localStorage.removeItem('username');

4**-clear ();** to clear all key-value pairs from local storage.

localStorage.clear();

**Note:** localStorage object can only store data as strings, so you may need to use methods like JSON.stringify() and JSON.parse() to store and retrieve complex data types like objects and arrays.

The amount of data you can store in local storage is limited (usually around 5-10MB,chrome-5mb and safari-10mb), so it's important to use it judiciously and avoid storing sensitive information.

**Redux:** Store, Action, Reducer function

**Installation: npm install @ngrx/store @ngrx/effects @ngrx/entity**

**1-Or** npm install redux ng2-redux --save

2-import the NgRedux module and the root reducer (which you will create later) in your app module and add in import array in app.module.

import { NgReduxModule, NgRedux } from 'ng2-redux';

3-create store with the help of configureStore method

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

import { BrowserModule } from '@angular/platform-browser';

import { NgReduxModule, NgRedux } from 'ng2-redux';

import { rootReducer } from './reducers';

import { AppComponent } from './app.component';

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule, NgReduxModule],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule {

constructor(ngRedux: NgRedux<any>) {

ngRedux.configureStore(rootReducer, {});

}

}

Here, the configureStore method is used to create the Redux store and connect it to the root reducer. The second parameter to this method is the initial state of the store, which is an empty object in this case.

4-Need to create your root reducer file called reducers.ts: take curr state and action as parameter

**rootReducer(state, action) {return new State;}**

-If in case we have more reducer then we must combine them all in single rootReducer function with the help of combineReducer function and pass as argument in configureStore Function.

import { combineReducers } from 'redux';

// import your reducers here

import { counterReducer } from './counter.reducer';

const rootReducer = combineReducers({

// add your reducers to this object

counter: counterReducer});

export default rootReducer;

5-Finally Use the Redux store in components. To do this, you need to inject the NgRedux service into your component's constructor and use the **select method** to retrieve the parts of the store that your component needs.

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

import { NgRedux, select } from 'ng2-redux';

@Component({

selector: 'app-counter',

template: `

<div>

<h1>Counter: {{ counter }}</h1>

<button (click)="increment()">Increment</button>

<button (click)="decrement()">Decrement</button>

</div>

`})

export class CounterComponent {

@select('counter') counter$: Observable<number>;

constructor(private ngRedux: NgRedux<any>) {}

increment() {

this.ngRedux.dispatch({ type: 'INCREMENT' });}

decrement() {

this.ngRedux.dispatch({ type: 'DECREMENT' });}}

Here, the **@select decorator();** is used to retrieve the counter property from the store and make it available as an observable on the component's counter$ property. The dispatch method is used to dispatch actions to the store.

**Testing:**

The most popular testing frameworks for Angular are Jasmine and Karma.

Jasmine is a **behavior-driven development (BDD)** framework for testing JavaScript code. It provides a set of functions for writing unit tests and allows developers to write tests in a readable and expressive manner. Jasmine provides features like test suites, test specs, test expectations, and spies for testing asynchronous code.

In BDD, the focus is on the behavior of the system being developed rather than the implementation details.

Having support Spy Objects to target an element we want to test.

**Karma** is a test runner that allows you to execute your Jasmine tests in a browser or a headless browser environment. Karma automates the process of running your tests in **different browsers** and provides features like test results reporting and continuous integration support.

Jasmine and Karma are both testing frameworks commonly used in Angular applications.

In an Angular application, you can use Jasmine and Karma together to write and execute tests for your components, services, and other Angular artifacts. You can use Karma to launch a browser or a headless browser, and then Jasmine to write the actual test code. By using these two frameworks, you can ensure that your Angular application is thoroughly tested and free of bugs before deploying it to production.

Angular provides the **TestBed** utility for **setting up and configuring the testing environment**. It also includes a number of testing utilities and services, such as ComponentFixture, which provides access to a component instance and its template, and HttpClientTestingModule, which provides a mock HTTP client for testing HTTP requests and responses.

In addition to Jasmine and Karma, developers can also use Protractor, a testing framework specifically designed for Angular applications. Protractor is an end-to-end testing framework that uses Selenium WebDriver to interact with the application in a browser environment.

**AAA=Arrange Act Assert pattern**

Arrange: Setting up the test environment, creating the testing module using TestBed.configureTestingModule, and getting an instance of the service to be tested.

Act: Invoking the calculateSum method of the service with provided input values.

Assert: Checking if the result of the calculation matches the expected value (15 in this case).

import { TestBed } from '@angular/core/testing';

import { MyService } from './my.service';

describe('MyService', () => {

let service: MyService;

beforeEach(() => {

TestBed.configureTestingModule({});

service = TestBed.inject(MyService);

});

it('should calculate the sum correctly', () => {

// Arrange

const num1 = 5;

const num2 = 10;

// Act

const result = service.calculateSum(num1, num2);

// Assert

expect(result).toBe(15);

});

});

**Zone.js Vs Rxjs:** both are the JS library to manageasynchronous operations.

Zone.js is a JavaScript library that is commonly used in Angular applications to help manage asynchronous operations, such as event listeners, timers, and AJAX (async JS+xml) requests and keep track of asynchronous operations and keep the user interface in sync with the state of the application.

Zone.js provides a way to hook into these operations and track their progress, allowing Angular to know when an asynchronous operation has started and finished. This is important because Angular uses a concept called **change detection** to keep the user interface up to date with the state of the application. If an asynchronous operation updates the state of the application but Angular is not aware of it, the user interface may not reflect the correct state.

Zone.js works by creating a zone around an asynchronous operation. A zone is essentially a wrapper around a JavaScript function that intercepts calls to certain asynchronous operations and allows you to track them. When an asynchronous operation is initiated within a zone, Zone.js will notify Angular that a new task has started. Angular will then perform change detection once the task has completed to update the user interface.

On the other hand, **RxJS** is a library for reactive programming. It provides a way to work with asynchronous data streams using observable sequences. RxJS can be used to handle asynchronous operations such as AJAX requests, but it is not designed specifically for this purpose. RxJS provides a powerful **set of operators** for manipulating and transforming data streams, as well as tools for handling errors, multicasting data, and more. It can be used in a wide variety of applications beyond Angular.

In summary Zone.js is focused on managing asynchronous operations in the browser, while RxJS is focused on working with reactive data streams.

**Firebase:** Firebase is a popular backend service provided by Google, which offers various services such as authentication, database, storage, hosting, and more.

**npm install -g firebase-tools**

**firebase login**

**firebase init**

**npm install firebase @angular/fire**

import { AngularFireModule } from '@angular/fire';

import { environment } from 'src/environments/environment';

@NgModule({

imports: [

AngularFireModule.initializeApp(environment.firebaseConfig),],...})

export class AppModule { }

**Interview Questions:**

**1. What is Angular-** Angular is an open-source web application development framework created by Google. It is used to build frontend, single-page applications that run on JavaScript.

**2. What are the technologies used in Angular?**

Angular uses TypeScript, which is a superscript of JavaScript. So, any valid JavaScript is a valid TypeScript. (Static Type, Interface)

**3. What are the technologies used in Angular?**

Ts is Superset of JS means any valid JS Code is valid Ts code.

TypeScript is just a JS with additional features like strong and static typing which makes it predictable and easier to debug.

Static typing, Interface-Structure of object, Cohesion, Constructor, Access Modifier: public, private, protected.

**4. How does an Angular application work?**

The working of Angular is based on its components. So, the working of the Angular application starts with the configuration file ANGULAR.JSON. The builder refers to this file to find the paths, configurations, and the main file. Now the process finally starts. Next comes the MAIN.TS file that acts as the entry point for the configuration file. It basically helps in creating the browser environment that enables it to run the application. Now, the bootstrapping of the Angular application is done through the APP.MODULE.TS. Now the app component that gets bootstrapped is stored in the APP.COMPONENT.TS file. Now the INDEX.HTML file is called and is used to ask Angular to load the application component. After the component is loaded, the content gets displayed from the APP.COMPONENT.HTML file.

**5. What is metadata?**

Using metadata is how we tell Angular how to process a class. When we use a component, it acts as a class unless we tell Angular that it’s a component, and we do this with the help of metadata. Metadata is attached in TypeScript using a decorator. Decorators are functions that know the configuration of classes and how they are supposed to work.

6. What is the difference between constructor and ngOnInit?

|  |  |  |
| --- | --- | --- |
| Basis | Constructor | ngOnInit |
| Usage | A Constructor should be used to set up Dependency Injection, Initialization of class fields, etc. | ngOnInit is used to write the work code that executes as soon as the class is instantiated. |

**7. What is the purpose of the async pipe?**

**Alternate of subscribe is async pipe.**

The main purpose is to simplify the process of subscribing to and displaying the values emitted by Observable streams or Promises directly in the **template**, while automatically managing the subscription lifecycle.

**8. What is the purpose of ngFor and ngIf directive?**

**\*ngFor-** directive is used to loop over a collection of items and create new instance of a template for each item in the collection.

Syntax \*ngfor=”let value of collection”;

TrackBy with \*ngfor- When you use \*ngFor with large lists,

Structural Directives: to manipulate the structure of the DOM. They add or remove html elements from the DOM based on conditions, start with asterisk, such as \*ngIf, \*ngswitch and \*ngFor.

**\*ngIf -**The \*ngIf directive is used to conditionally add or remove an element from the DOM based on a Boolean expression. For example, you might use ngIf to display a message only if a certain condition is true. Ex. <p \*ngIf="isLoggedIn">Welcome, {{username}}!</p>

Just like the hidden property of the element the only difference b/w ngIf and hidden property, if ngIf condition false then the element removed from the Dom whereas in hidden its just hidden in dom which occupy the dom space.

ngIf-else if the condition evaluated by ngIf is false. We have to create template variable.

ngIf-then-else.

**9. What happens if you use script tag inside template?**

Angular's core philosophy revolves around separating concerns and maintaining a clear distinction between the view (template) and the component logic. Mixing script tags within templates goes against this separation and can lead to maintenance challenges and security risks like Limited Execution, Potential Security risk, Invalid Angular syntax.

If we use script tag inside template, Angular marks the value as unsafe and automatically initiates the process of sanitizing it. This eradicates(remove) the script tag but the content is kept safe, i.e. the text element. This entire process results in eliminating the risk of injection attacks.

**10. What is Template Statement/Variable/Expression:**

**Template statements-** The methods or properties in Angular that are used in HTML in response to user events are called template statements.

**Template expressions-** is an expression that is represented in double curly braces ‘{{ }}’ and produces a value. Also, k/a String Interpolation for one way data binding.

**Template reference variables** allow you to easily reference specific elements or components in your code. A template reference variable is a way to give a name to a specific element or component instance in the template, which can then be used to reference that element or component in the code. To create a template reference variable, you simply add a hashtag (#) followed by the variable name to the element or component you want to reference.

**11. Difference between Angular and AngularJS:**

1-Architecture: Angular is a complete rewrite of AngularJS with a different architecture. Angular uses a component-based architecture where components represent the building blocks of an application, whereas AngularJS uses a directive-based or MVC architecture.

2-Language: Angular is written in TypeScript, which is a superset of JavaScript, whereas AngularJS is written in plain JavaScript.

3-Performance: Angular is faster than AngularJS due to the use of the new Change Detection mechanism and Ahead-Of-Time (AOT) compilation, which means that the application is pre-compiled during the build process, making it faster at runtime.

4-Directives: In AngularJS, directives are used to create reusable components, whereas in Angular, components replace directives as the primary way to create reusable code.

5-Dependency Injection: Dependency injection in Angular is more powerful and flexible than in AngularJS. It allows for better testing and modularization of an application.

6-Mobile support: Angular has better support for mobile devices as it has a built-in mobile toolkit, whereas AngularJS doesn't have this feature.

**12. What is Data Binding-**Communication or synchronization of data between the component and the template. The communication between app.component.ts and app.component.html file It allows you to bind a component's property to a view, so that whenever the component's data changes, the view is automatically updated. Angular also supports two-way data binding, which means that changes made in the view are reflected in the component and vice versa. The main techniques for data binding in Angular are:

1-String Interpolation 2-Property Binding 3-Attribute Binding

4-Class Binding 5-Style Binding 6-Event Binding

7-Event Filtering 8-Two-Way Binding 9-Template Reference Variables

One way data binding, the changes in the state affect the view from component to view template. On the contrary it is also possible that the change in the view affects the state by changing it from view template to component.

Two-way Data Binding, the changes in the view can lead to change in the model. Similarly, any changes in the model can change the view from component to view template.

**13. Explain how custom elements work internally?**

Custom Elements, also known as Web Components, are a set of web platform APIs that allow you to define your own reusable components with their own encapsulated HTML, CSS, and JavaScript.

Creating a custom element class: As a first step, build a custom element class using the createCustomElement() function provided by Angular. The function converts an Angular component (including its dependencies) to a custom element. The NgElementConstructor interface is implemented through this conversion which, in turn, creates a constructor class that is used for producing a self-bootstrapping instance.

Registering element class with browser: The customElements.define() function is used to register the configured constructor, and its associated custom-element tag with the browser’s CustomElementRegistry.

The custom element is added to DOM similar to a built-in HTML.

Browser instantiates component-based class: Once the custom element is added to DOM, an instance of the registered class is created by the browser and added to the DOM.

Data binding and Change detection: In the final step, the created instance enables data binding and change detection. The template content is rendered using the component and DOM data.

**14. What is HttpClient and its benefits?**

Front-end applications communicate with backend services over HTTP protocol using either XMLHttpRequest interface or the fetch () API. Angular provides a simplified client HTTP API known as HttpClient which is based on top of XMLHttpRequest interface.

import { HttpClientModule } from '@angular/common/http';

**Advantages:**

Contains testability features.

Provides typed request and response objects.

Intercept request and response.

Supports Observable APIs

Supports streamlined error handling.

**Example:**

Import HttpClient into root module:

import { HttpClientModule } from '@angular/common/http';

@NgModule({

imports: [BrowserModule,

// import HttpClientModule after BrowserModule.

HttpClientModule,],})

export class AppModule {}

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

import { HttpClient } from '@angular/common/http';

const userProfileUrl: string = 'assets/data/profile.json';

@Injectable()

export class UserProfileService {

constructor(private http: HttpClient) { }

getUserProfile() {return this.http.get(this.userProfileUrl);}}

fetchUserProfile() {

this.userProfileService.getUserProfile()

.subscribe((data: User) => this.user = {

id: data['userId'],

name: data['firstName'],

city:  data['city']}); }

**15. Router Outlet is a directive that is used to render the component associated with the current route. It acts as a placeholder where the router dynamically renders the component that matches the current route configuration.**

**16. Services** -is a reusable piece of logic that can be used across multiple components. Services are used to encapsulate and share common functionality throughout an application, making it easier to maintain and test the application. (Reuse, Encapsulated just like function).

**17. What is the difference between jQuery and Angular?**

**jQuery is js library, whereas Angular is JS frontend framework.**

**Unlike jQuery, Angular offers two-way data binding, validation, support RESTful API.**

**Both similar in expression consist of variables, operators, and literals.**

**18. Dependency Injection (DI) -**we can use one functionality or service of an object to another component, means other component is dependent on first component for functionality and first component is injected to other component.

DI means injecting or providing the dependency of a class into its constructor.

**19. Describe Angular authentication and authorization.**

The login details of a user are given to an authenticate API available on the server. Once the credentials are validated by the server, it returns a JSON web token (JWT), which includes attributes and the data of the current user. Further, the user is easily identified using JWT, and this process is known as authentication.

After logging in, users have various types and levels of access—some can access everything, while others may have restrictions from some resources. Authorization determines the access level of these users.

**20. What is the digest cycle process in Angular? Comes in Angular Js**

In contrast, in modern Angular:

**Change Detection mechanism and Zones:** Angular uses a more sophisticated change detection mechanism that operates within the context of JavaScript "zones."

Change detection is triggered automatically by various asynchronous events, such as user interactions, timers, and HTTP requests.

**Unidirectional Data Flow:** Modern Angular primarily follows a unidirectional data flow, where changes are propagated from the component's class to the view.

**OnPush Change Detection:** Angular also supports change detection strategies like OnPush, where components are only checked for changes if their input properties change.

**21. What are the distinct types of Angular filters?**

Filters are placed after the pipe symbol ( | ) while used in expressions.

**Currency**, **filter**, **date**, **lowercase**, **uppercase**, **orderBy, json**, **number**, **limitTo, etc.**

**22. What is schematic?**

A schematic is a template-based code generation tool that helps you automate repetitive tasks and accelerate your development workflow. Schematics are built on top of the Angular CLI and provide a standardized way to generate and modify files in an Angular application. Schematics are written in TypeScript and use the Angular DevKit to interact with the file system and manipulate files.

To use a schematic, you first need to install it using the Angular CLI. You can then run the schematic using the **ng generate command** followed by the schematic name and any necessary options or arguments. For example, to generate a new component, you would run ng generate component my-component.

**Rule in Schematic** refers to a set function that takes a Tree, transforms it, and finally returns a new Tree.

**23. What is multicasting in Angular?**

**T**he process of sharing a single observable data stream among multiple subscribers.

In Angular, when we are using the HttpClient module to communicate with a backend service and fetch some data, after fetching the data, we can broadcast it to multiple subscribers, all in one execution. This task of responding with data to multiple subscribers is called multicasting. It is specifically useful when we have multiple parts of our applications waiting for some data. To use multicasting, we need to use an RxJS subject. As observables are unicast, they do not allow multiple subscribers. However, subjects do allow multiple subscribers and are multicast.

**Ways to multicast:**

Using the share() operator.

Using the Subject class and next() method.

Using the BehaviorSubject class.

**24. What will happen if you do not supply handler or callback function for observer** –

whether you're dealing with Observables in RxJS or event listeners in plain JavaScript, not supplying a handler means that you're not providing any code to execute when an event occurs or when values are emitted. As a result, the event or Observable will have no visible effect on your application.

import { Observable } from 'rxjs';

  const observable = new Observable(observer => {

    observer.next('Value 1');

    observer.next('Value 2');

    observer.complete(); });

  // Subscribing without a handler

  const subscription = observable.subscribe();

-your observable will not do anything when it emits data. This is because the observer is responsible for defining the behavior that should occur when the observable emits data.

An observer consists of 3 functions: next(), error(), and complete(). These functions define how the observer should react to different events that can occur when an observable emits data. These are call back functions.

If you do not supply a handler for the next() function, your observer will not do anything when the observable emits data. This means that any data emitted by the observable will be lost, and your application will not be able to react to it.

Similarly, if you do not supply handlers for the error() and complete() functions, your observer will not be able to handle errors or know when the observable has completed. This can lead to unexpected behavior in your application and make it difficult to debug errors.

In case of event listener in plain Js: In this case, if the button is clicked, the event will still be triggered, but since there's no handler provided, nothing will happen in response to the click.

 const button = document.getElementById('myButton');

// Attaching an event listener without a handler

button.addEventListener('click', /\* no handler function here \*/);

**25. Differences between Angular expressions and JavaScript expressions (operator (=+/? && ))**

Angular and JavaScript expressions are similar in syntax, but angular expressions are scoped locally while in JavaScript, are scoped against the global window object, and we can use directives in angular not in js.

**26. Server-side rendering (SSR) -** involves generating the HTML, CSS, and JavaScript for a web page on the server, and then sending this pre-rendered content to the client for display in the browser.

Angular Universal is a package for enabling server-side rendering in Angular applications.

Angular Universal also help in SEO, SSR, Improved performance, Faster time content, Universal Compatibility, Async Rendering and architecture.

**ng add @nguniversal/express-engine**

**27. Difference between interpolated content and the content assigned to the innerHTML property of a DOM element?**

Both used to dynamically update the content of a web page.

Interpolated content is created using curly braces {{}} and is used to dynamically display data values in the HTML template of an Angular component. Interpolated content is a one-way binding, meaning that changes to the data value will be reflected in the HTML template, but changes made to the HTML template will not affect the data value.

The innerHTML property of a DOM element is used to dynamically update the content of a DOM element using JavaScript. The innerHTML property allows you to set the HTML content of an element, including any HTML tags and content. Unlike interpolated content, the innerHTML property is a two-way binding, meaning that changes made to the HTML content will also update the data value.

<div id="myDiv">Initial content</div>

const myDiv = document.getElementById('myDiv');

myDiv.innerHTML = '<strong>New content</strong>';

-Interpolated content is typically used to display data values in the HTML template, while the innerHTML property is used to dynamically update the content of a DOM element.

-Interpolated content is a one-way binding, while the innerHTML property is a two-way binding.

**28. What are HttpInterceptors in Angular?**

HttpInterceptors are part of the @angular/common/HTTP module and are used to intercept and transform HTTP requests and responses. They provide a way to modify or handle requests and responses globally before they are sent to the server or received by the client. Interceptors are useful for adding authorization headers, handling errors, or logging requests and responses.

To create an interceptor need to implement the HttpInterceptor interface, which contains two methods: **intercept() and handleError().**

The intercept() method is called for each HTTP request, and it allows you to modify the request headers or body before it is sent to the server. The handleError() method is called if an error occurs during the request or response

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

import { HttpInterceptor, HttpHandler, HttpRequest } from '@angular/common/http';

@Injectable()

export class AuthInterceptor implements HttpInterceptor {

intercept(request: HttpRequest<any>, next: HttpHandler) {

// Add authorization header to the request

const token = localStorage.getItem('token');

if (token) {

request = request.clone({

setHeaders: {

Authorization: `Bearer ${token}`}});}

return next.handle(request);}}

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

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

import { AuthInterceptor } from './auth.interceptor';

@NgModule({

imports: [HttpClientModule],

providers: [{provide: HTTP\_INTERCEPTORS,

useClass: AuthInterceptor,

multi: true}]})

export class AppModule { }

**29. What is ngcc- It is a tool provided by the Angular team to help developers transition from AngularJS (Angular 1.x) to Angular (Angular 2+).**

The ngcc(Angular Compatibility Compiler) is a tool used in Angular to upgrade node\_module, compiled with non-ivy ngc into ivy compliant format.

**30. What is folding-**In Angular, it might be possible while generating the code that some of the non-exported members are folded. This is called Folding, i.e the process in which the evaluation of an expression is done by the collector and result is recorded in the .metadata.json, is known as Folding.

**31. What is the State function?**

The State function in Angular declares an animation state within a trigger attached to an element.

import { trigger, state, style, transition, animate } from '@angular/animations';

export const myAnimation = trigger('myAnimation', [

state('inactive', style({

backgroundColor: '#eee',

transform: 'scale(1)'})),

**32. What is Style function-**The Style function in Angular is used to declare a key/value object that contains CSS properties/styles and are used for an animation.

**33. NgZone-** is a service that provides a way to execute code outside or inside the Angular zone.

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

@Component({

selector: 'app-example',

template: `

<button (click)="onClick()">Click Me</button>`})

export class ExampleComponent {

constructor(private ngZone: NgZone) {}

onClick() {

this.ngZone.runOutsideAngular(() => {

// run code outside the Angular zone

console.log('Running code outside the Angular zone');});}}

By default, Angular runs all code inside the zone, which means that any changes made by that code will trigger change detection and update the view. However, in some cases, you may want to run code outside the zone to improve performance or prevent change detection from being triggered.

**34. What is NoopZone**-helps Angular figure out and know when to trigger the change detection.

NoopZone is a class in Angular that represents a zone that does nothing. It is used to optimize performance in situations where you don't need to run code inside the Angular zone.

When Angular is initialized, it creates a default zone that tracks and manages change detection. This default zone is used to run all code that is executed by the application. However, in some cases, you may not need to run code inside the Angular zone. For example, if you are using a third-party library that updates the DOM directly, you can improve performance by running that code outside the zone.

To achieve this, you can use the NoopZone class to create a zone that does nothing. Any code that is executed inside this zone will not trigger change detection or update the view.

import { Component, NgZone, NoopZone } from '@angular/core';

@Component({

selector: 'app-example',

template: `

<button (click)="onClick()">Click Me</button>

`})

export class ExampleComponent {

constructor(private ngZone: NgZone) {}

onClick() {

const noopZone = new NoopZone();

this.ngZone.runOutsideAngular(() => {

noopZone.run(() => {

// run code outside the Angular zone

console.log('Running code outside the Angular zone');});});}}

**35. What is Bazel tool?**

Bazel is a powerful build tool developed and massively used by Google and it can keep track of the dependencies between different packages and build targets. In Angular8, you can build your CLI application with Bazel. **Note:** The Angular framework itself is built with Bazel.

**36. What is platform in Angular?**

A platform is the context in which an Angular application runs. The most common platform for Angular applications is a web browser. The runtime-platform is provided by the @angular/platform-\* packages and these packages allow applications that make use of @angular/core and @angular/common to execute in different environments. i.e, Angular can be used as platform-independent framework in different environments, For example,

1-While running in the browser, it uses platform-browser package.

2-When SSR(server-side rendering ) is used, it uses platform-server package for providing web server implementation.

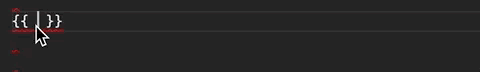
**36. What is DOM sanitizer?**

DomSanitizer is used to help preventing Cross Site Scripting Security bugs (XSS) by sanitizing values to be safe to use in the different DOM contexts.

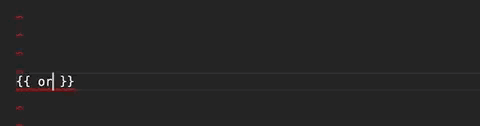
**37. Internalization:** Internationalization, also known as **i18n,** refers to the process of adapting an application to support multiple languages and cultures.

**38. Explain the features provided by Angular Language Service -same feature Typescript have.**

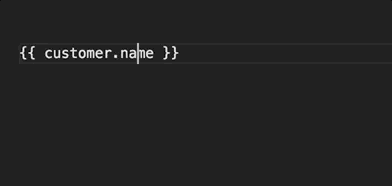
**Autocompletion:** Autocompletion can speed up your development time by providing you with contextual possibilities and hints as you type with in an interpolation and elements.

[](https://github.com/sudheerj/angular-interview-questions/blob/master/images/language-completion.gif)

**Error checking:** It can also warn you of mistakes in your code.

[](https://github.com/sudheerj/angular-interview-questions/blob/master/images/language-error.gif)

**Navigation:** Navigation allows you to hover a component, directive, module and then click and press F12 to go directly to its definition.

[](https://github.com/sudheerj/angular-interview-questions/blob/master/images/language-navigation.gif)

**Subjects In RxJs:**

In RxJS, a Subject is a special type of observable that allows values to be multicasted to multiple observers. It acts as both an observer and an observable, meaning you can subscribe to it like any other observable, and you can also push values into it by calling its next() method.

Here are the key characteristics of a Subject in RxJS:

Multicasting: A Subject can multicast values to multiple subscribers. This means that each subscriber receives the same values emitted by the Subject.

Hot Observable: A Subject is a hot observable, which means it starts emitting values as soon as it's created, regardless of whether there are any subscribers.

Imperative API: Unlike other observables in RxJS, which are primarily declarative, a Subject has an imperative API. You can manually push values into the Subject using the next(), error(), and complete() methods.

Multiple Observers: You can subscribe multiple observers to a Subject, and all of them will receive the same sequence of values emitted by the Subject.

Bridge between Observable and Observer: A Subject can be thought of as a bridge between observables and observers. It can be subscribed to like an observable, and it can also act as an observer by receiving values from other observables and emitting them to its subscribers.

There are different types of Subject in RxJS:

Subject: A basic implementation of a Subject.

BehaviorSubject: A variant of Subject that remembers the last emitted value and emits it to new subscribers immediately upon subscription.

ReplaySubject: A variant of Subject that emits a specified number of previously emitted values to new subscribers.

AsyncSubject: A variant of Subject that emits only the last value emitted by the source observable (and only when the source observable completes).

Subject is a powerful tool in RxJS for implementing event handling, data sharing, and multicasting scenarios, especially in scenarios where you want multiple observers to receive the same values emitted by an observable. However, you should be mindful of its hot observable nature and ensure that it's used appropriately in your application to prevent unintended side effects.