ANGULAR

Angular is a JavaScript Binding framework which binds the HTML UI and Javascript Model. This helps you to reduce your effort on writing those lengthy lines of code for binding.
Adding to it,it also helps you to build SPA by using the concept of routing. It also has a lot of other features like HTTP , DI , Input output because of which you do not need other frameworks.

Angular is an open-source front-end web framework. It is one of the most popular JavaScript frameworks that is mainly maintained by Google. It provides a platform for easy development of Mobile and Desktop web applications using HTML and TypeScript.
It integrates powerful features like declarative templates, end to end tooling, dependency injection.
Angular is typically used for the development of SPA which stands for Single Page Applications.

What are Single Page Applications (SPA)?

SPA stands for Single page application. Single page applications are applications where the main UI gets loaded once and then the needed UI or data is loaded rather than making a full post back.
SPA or single page applications are those applications where the main page is loaded once and the other pages get loaded on demand.
A Single Page Application method is speedier, resulting in a more consistent user experience.
Angular provides a set of ready-to-use modules that simplify the development of single page applications.
Advantages of using Angular framework are listed below

- It supports two-way data-binding
- It follows MVC pattern architecture
- It supports static template and Angular template
- You can add a custom directive
- It also supports RESTful services
- Validations are supported
- Client and server communication is facilitated
- Support for dependency injection
- Has strong features like Event Handlers, Animation, etc.

Explain the basic components involved in Angular
There are 7 important pillars in Angular Architecture.
1. Template:- The HTML view of Angular.
2. Component:- Binds the View and Model.
3. Modules:- Groups components logically.
4. Bindings :- Defines how view and component communicate.
5. Directive :- Changes the HTML DOM behaviour.
6. Services :- Helps to share common logic across the project.
7. DI :- Dependency injection helps to inject instance across constructor

TypeScript is open-source and it is developed and maintained by Microsoft. The TypeScript language provides several benefits are as follows while developing angular applications
1- Intelligence support while writing code   2- Auto-completion Facility
3- Code navigation   4- Strong Typing
5- It also supports features like classes, interfaces, and inheritance as our traditional programming languages such as C#, Java, or C#.

For setting up Angular Application   (node -v and npm –v)

Node.js is an open-source cross-platform javascript run-time environment)

NPM is node.js package manager for javascript programming language. It is automatically installed when we install node.js

Typescript -   npm install –g typescript
Angular CLI (It is a tool that allows us to create a project, build and run the project in the development server directly using the command line command) - ng v

```
npm install -g @angular/cli
Create ang App-    ng new project-name
Cmd-
Ng new appName --create-application=false     ->create workspace but not application
Cd appName
Ng g application webAppName
Ng g library myLib
 Ng g c home --skip-tests=true
Ng g c login
Ng g c login/logout
Ng g c signup -> create comp. At root level
ng update @angular/cli @angular/core
```

---

node_modules Folder-This folder contains the packages (folders of the packages) which are installed into your application when you created a project in angular. If you installed any third-party packages then also their folders are going to be stored within this node_modules folder. npm install

Only private dependencies are required at production level.

Src folder- This is the source folder of our angular application. That means this is the place where we need to put all our application source code. So, every component, service class, modules, everything we need to put in this folder only.

Assets folder where you can store the static assets like images, icons, etc.

Environments folder is basically used to set up different environments. These two files are as follows:

1. environment.prod.ts. This file for the production environment
2. environment.ts. This file for the development environment.

Favicon.icon: *It is the icon file that displays on the browser.*

Index.html *file contains HTML code with the head and body section. It is the starting point of your application or you can say that this is where our angular app bootstraps.*

Polyfills.ts File is basically used for browser-related configuration. In angular, you write the code using typescript language. The Polyfills.ts file is used by the compiler to compile your Typescript code to a specific JavaScript method so that it can be parsed by different browsers.

Angular.json- It contains all the configuration of Angular Project. It contains the configurations such as what is the project name, what is the root directory as source folder (src) name which contains all the components, services, directives, pipes, what is the starting point of your angular application (index.html file), What is the starting point of typescript file (main.ts), etc.

how an Angular application works:

Configuration File (`angular.json`): *Every Angular application includes a `angular.json` file. This file contains all the configuration settings for the application.*
During the build process, Angular uses this file to find the entry point of the app.

Entry Point (`main.ts`): *The `main.ts` file acts as the entry point of the Angular application.*
It creates a browser environment for the app to run.
It calls the `bootstrapModule()` function to bootstrap the application.

Bootstrapping the App Module:*Code used in `main.ts`:*

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

platformBrowserDynamic().bootstrapModule(AppModule);
```

`AppModule` is bootstrapped here, which is defined in `app.module.ts`.

App Module (`app.module.ts`): *`AppModule` is the root module of the application.*
It contains declarations of all components used in the application.
It specifies the root component to bootstrap, typically `AppComponent`.

Root Component (`app.component.ts`):
`AppComponent` is the root component of the application.
This component is responsible for interacting with the webpage and supplying data.

Component Structure: *Each component in Angular has:*

Selector: *Used to insert the component in HTML*

***Template / TemplateURL**: Contains the HTML structure.*
***StyleURLs**: Contains styles specific to the component.*

HTML Entry (`index.html`):
Angular calls the `index.html` file after bootstrapping.
This file contains a custom HTML tag, typically `<app-root>`, which corresponds to the selector of `AppComponent`.
***Rendering**:*
The selector `<app-root>` triggers Angular to render the `AppComponent`.
Angular then processes the template and displays the content in the browser.

Browserslist file is currently used by autoprefixer to adjust CSS to support the specified browsers.

test.ts file is the configuration file of Karma which is used for setting the test environment. Within this file, the tester will write the unit test cases for testing the project.

karma.config.js file is used to store the setting of karma i.e. test cases. It has a configuration for writing unit tests. Karma is the test runner and it uses jasmine as a framework. Both tester and framework are developed by the angular team for writing unit tests.

package-lock.json is automatically generated for those operations where npm modifies either the node_modules tree or package.json. In other words, the package.lock.json is generated after an npm install.

Package.json: file is mandatory for every npm project. It contains basic information regarding the project (name, description, license, etc), commands which can be used, dependencies – these are packages required by the application to work correctly, and dependencies – again the list of packages which are required for application however only during the development phase.

README is the file that contains a description of the project which we would like to give to the end-users so that they can start using our application in a great manner. This file contains the basic documentation for your project, also contains some pre-filled CLI command information.

Tsconfig.app.json: This file is used during the compilation of your application and it contains the configuration information about how your application should be compiled.

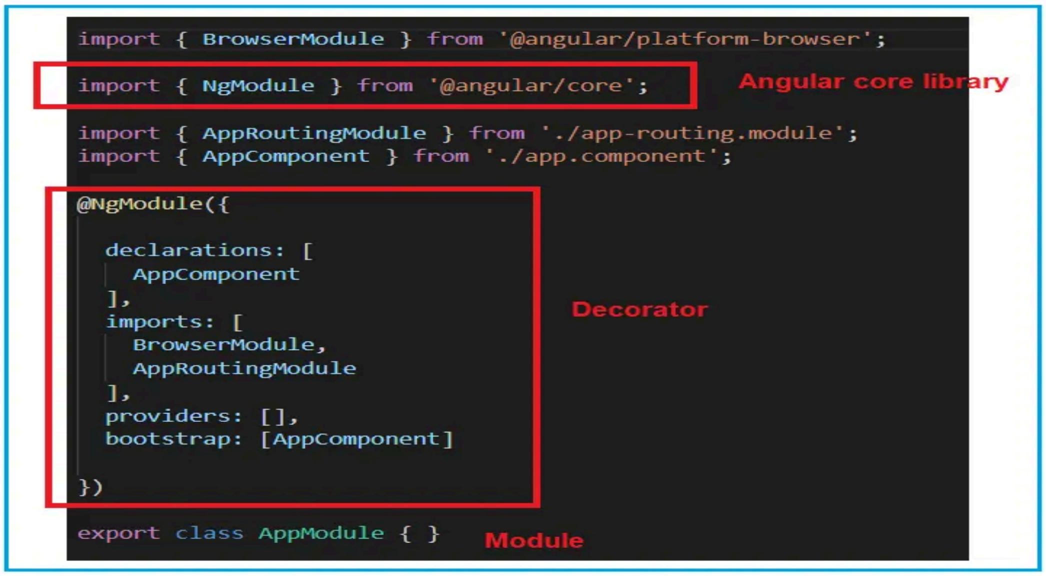
Tsconfig.json: This file contains the configurations for typescripts. If there is a tsconfig file in a directory, that means that the directory is a root directory of a typescript project, moreover, it is also used to define different typescript compilation-related options.

Tsconfig.spec.json: This file is used for testing purposes in the node.js environment. It also helps in maintaining the details for testing.
Tslint.json: This is a tool useful for static analysis that checks our TypeScript code for readability, maintainability, and functionality errors.

Module- is a mechanism to group components, services, directives, pipes. So, the grouping of related components, services, pipes, and directives is nothing but a module in angular. AppModule (app.module.ts file) is the default module or root module of our angular application.

By default, modules are of two types: 1- root module imports BrowserModule, whereas a 2- feature module imports CommonModule.

```
ng generate module modulename
ng g module modulename
```



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

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]

})
export class AppModule { }
```

Declarations array is an array of components, directives, and pipes. Whenever you add a new component, first the component needs to be imported and then a reference of that component must be included in the declarations array.

Imports section, we need to include other modules (Except @NgModule). By default, it includes two modules i.e. BrowserModule (Exports required infrastructure for all Angular apps) and AppRoutingModule.

Providers Array (providers): We need to include the services in the provider's section. Whenever you create any service for your application, first you need to include a reference of that service in the provider's section and then only you can use that service in your application.

Bootstrap Array (bootstrap): This section is basically used to bootstrap your angular application i.e. which component will execute first. At the moment we have only one component i.e. AppComponent and this is the component that is going to be executed when our application runs.

Decorators provide metadata to [CPVM] classes, property, value, method, etc. and decorators are going to be invoked at runtime. The name of the decorator starts with @ symbol followed by brackets and arguments. Meta-data or annotations are also termed as decorators.

Built-in decorators are available in angular. All built-in decorators are imported from @angular/core library. Some of them are as follows:

1. @NgModule to define a module.
2. @Component to define components.
3. @Injectable to define services.
4. @Input and @Output to define properties

Types of decorators: CPPM

1. **Class Decorators** are applied at the class level and define the overall purpose of the class. They are used to declare whether a class is a component, module, or service. These decorators allow us to provide metadata without writing extra code inside the class. Examples of class decorators are `@Component`, which defines a component, and `@NgModule`, which defines a module.

2. **Property Decorators** are used to decorate specific properties within a class. They are commonly used in component communication. Examples include `@Input`, which allows a parent component to pass data to a child component, and `@Output`, which enables the child component to send data to the parent. These decorators are used inside the class on its properties.

3. **Parameter Decorators** are used for parameters within class constructors. A common example is `@Inject`, which is used to manually specify a dependency that should be injected into the constructor.

4. **Method Decorators** are used to add functionality to methods defined inside a class. One popular example is `@HostListener`, which allows a method to respond to events like clicks, mouse hovers, or key presses. This helps the component interact with the DOM in a more structured way.

Component is the basic building block of the application. It is responsible for controlling a specific part of the user interface and binds the data (model) with the view (HTML). Components are defined using the `@Component` decorator. Each component typically consists of four files:

TypeScript file where the logic is written (`<component-name>.component.ts`),

HTML template file that defines the layout (`<component-name>.component.html`),

CSS file for styling (`<component-name>.component.css`), and

Testing file (`<component-name>.component.spec.ts`) used for unit testing.

The main purpose of using components is to make the code reusable and easier to test.

A component has three main parts: the template, the class, and the styles. The template is written in HTML and includes Angular-specific syntax like bindings and directives. It can be written in two ways: as an inline template directly inside the component using the `template` property, or as a separate file linked using the `templateUrl` property.

Data Binding means to bind the data (Components filed) with the View (HTML Content). Data binding defines how the view and component communicate with each other.

Data Binding is a process that creates a connection to communicate and synchronize between the user interface and the data.

There are two types of Data binding

1- In One-way binding, data flows in a single direction:

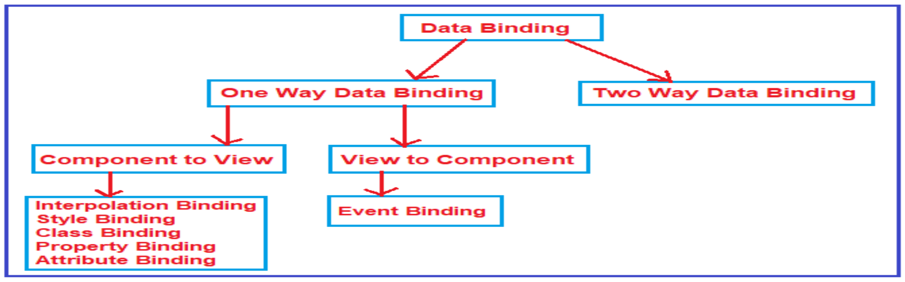
From Component to View: The view updates automatically when the component's data changes.

Example: `{{ title }}` or `[property]="value"`

From View to Component: The component gets updated based on user input via events.

Example: `(event)="handlerFunction($event)"`

2- In Two-way binding, data flows in both directions: Any change in the component updates the view, and Any change in the view (user input) updates the component's property.
Achieved using `[(ngModel)]` directive. Example: `<input [(ngModel)]="username">`



A- **<u>Component to View</u>**

1. Attribute Binding: Used to bind the **attributes** of an HTML element (like `colspan`, `aria-label`, etc.) to the component's properties.
Angular does **not** bind attributes directly like properties, so we must use the `attr.` prefix.

Syntax: `<td [attr.colspan]="columnSpan"></td>`

2. Property Binding: Binds the **DOM properties** of HTML elements to the component's model properties. It updates the element property whenever the component property changes.

Syntax: `[property]="expression"`

Examples: `<button [disabled]="isDisabled">Click Me</button>`

`<span [innerHTML]="title"></span>`

`<img [src]="imagePath" />`

Note: **Property Binding** and **Interpolation** both bind data **from component to view** (i.e., one direction).

3. Class Binding: Used to **add or remove CSS classes** on an element dynamically. Supports **conditional** class application.

Examples:

`<div [class.active]="isActive"></div>`

`<div [class]="className"></div>`
`isActive` or `className` are component properties that control the CSS class.

4. Style Binding: Used to set or update the **inline styles** of an element dynamically. You can bind to a single style or multiple styles.

Examples: `<div [style.color]="textColor"></div>`

`<div [style.font-size.px]="fontSize"></div>`

5. String Interpolation: Uses **double curly braces `{{ }}`** to display component data in the template.It evaluates the expression and inserts the result into the DOM.
Interpolation is a technique that allows the user to bind a value to a UI element.

Syntax: `{{ propertyName }}`

Example: `<p>{{ title }}</p>`

`<button disabled="{{ isDisabled }}">Click Me</button>`

Note: While interpolation looks like string substitution, it's actually evaluating an expression in the component.

DOM stands for Document Object Model. When a browser loads a web page, then the browser creates the Document Object Model (DOM) for that page.

B- View to Component-
Event binding This allows the component to respond to user interactions such as button clicks, key presses, mouse movements, and other DOM events. When an event occurs in the view (HTML), the component gets notified and can take action.

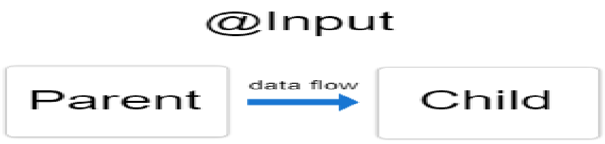
Event binding listens for changes or actions performed by the user on an HTML element.
The syntax for event binding uses **parentheses `()`** around the target event name, followed by an assignment to a **template statement** (usually a method from the component).

Syntax: `(eventName)="methodName()"`

Angular Component Input Properties- Component Input Properties are used to pass data from a parent component to a child component. This is achieved using the `@Input()` decorator in the child component. It allows the child to receive values from the parent, helping in component communication and data sharing in a structured way.

To use `@Input()`, you first need to import it from `@angular/core`. Once a property in the child component is decorated with `@Input()`, that property can receive data from the parent component through property binding.

When it comes to **sharing data between components in Angular**, here are the common ways:

- **Parent to Child**: Use the `@Input()` decorator to send data down to the child 

component.
- **Child to Parent**: Use the `@Output()` decorator along with `EventEmitter` to send data or events back to the parent component.
- **Child to Parent (using ViewChild)**: Use `@ViewChild()` to directly access a child component's properties or methods from the parent.
- **Unrelated Components**: Use a **shared service** to act as a bridge and share data between components that don't have a direct relationship.

Sometimes, in the component, we need direct access to elements or child components defined in the view. This is where **ViewChild and ViewChildren** come in.

`ViewChild` is used to get a reference to a **single element, directive, or component** from the template. It allows the component to directly interact with that element—for example, calling a method of a child component or manipulating a DOM element.

`ViewChildren` is similar but used when you want to access a **collection of elements or components** instead of just one.

So, **`ViewChild`** gives access to a **single instance**, while **`ViewChildren`** gives access to **multiple instances**. Both are helpful when you want to programmatically control elements defined in your template from your component class.

`ContentChild` and **`ContentChildren`** are used to access **projected content** that is passed from a parent component into a child component using **content projection** (typically via `<ng-content>`).

When a parent component uses content projection to insert HTML or another component into a child component's template, Angular provides `ContentChild` and `ContentChildren` to allow the child component to interact with that projected content.

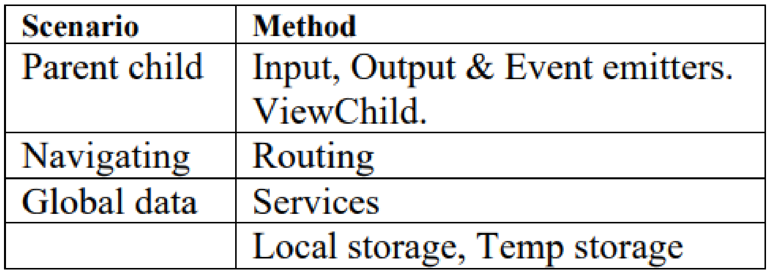
- `ContentChild` is used to get a reference to a **single element, directive, or component** that has been projected into the child component.
- `ContentChildren` is used when you want to get a **collection** of elements, directives, or components from the projected content.

In short, both are useful when you want the **child component to interact with what has been passed into it from the parent**, and they differ by how many projected elements they can reference:

- `ContentChild` → one item
- `ContentChildren` → multiple items

This is different from `ViewChild`/`ViewChildren`, which access elements inside the component's own view (template), not projected content from the outside.

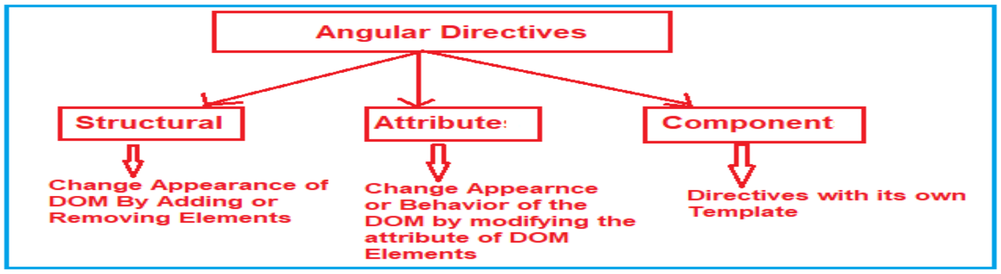
<u>Pass data between components</u>



Scenario	Method
Parent child	Input, Output & Event emitters. ViewChild.
Navigating	Routing
Global data	Services
	Local storage, Temp storage

<u>Directives</u> are used to change the appearance, behavior, or layout of elements in the DOM (Document Object Model). They are like custom HTML attributes that allow you to extend HTML's capabilities and create your own application-specific behavior. A directive is simply a class that is decorated with the `@Directive` decorator.

There are three main types of directives in Angular: **Structural**, **Attribute**, and **Component** directives. **(SAC)**


Angular Directives
Structural
Attribute
Component
Change Appearance of DOM By Adding or Removing Elements
Change Apperance or Behavior of the DOM by modifying the attribute of DOM Elements
Directives with its own Template

Structural directives change the layout of the DOM by adding or removing elements. They are responsible for shaping or reshaping the DOM structure. These directives are prefixed with an asterisk (`*`). Common structural directives include:

`*ngIf`: Adds or removes an element from the DOM based on a boolean condition
*<div *ngIf="isValid"><b>The Data is valid.</b></div>*
`*ngFor`: Iterates over a list or array to generate elements dynamically.
*<li *ngFor="let item of items">{{ item }}</li>*
`*ngSwitch`: Switches between multiple views based on a matching expression.

Attribute directives -change the appearance or behavior of an existing element. They don't add or remove elements from the DOM but alter how elements look or behave. Some common attribute directives include:
`NgStyle`: Used to set multiple inline styles dynamically.
<button [ngStyle]="{'color':'red', 'font-weight': 'bold', 'font-size.px':20}">Click Me</button>

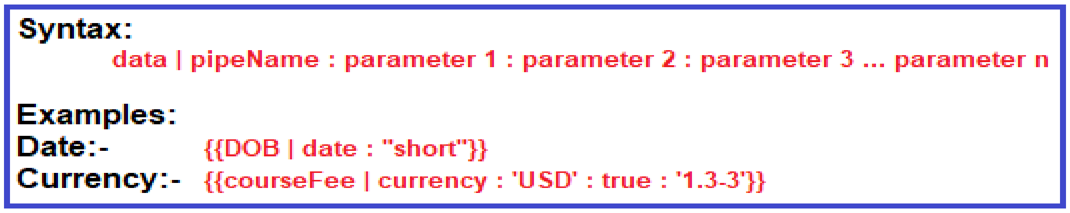
`NgClass`: Used to add or remove CSS classes dynamically based on conditions.
<h3 [ngClass]="'one'">Angular ngClass with String</h3>

Component Directive is a special type of directive that includes a template, styles and logic. It's the most commonly used directive in Angular. A component directive is created using the `@Component` decorator and is responsible for controlling a section of the user interface.

Pipes (|)- Pipe/filter takes the raw data as input and then transforms the raw data into some desired format. Pipes are simple functions to use in template expressions to accept an input value and return a transformed value. <td>{{student.Name | uppercase}}</td>

Example- lowercase, uppercase, titlecase, decimal, date, percent, currency etc.
two types i.e. Built-in Pipes and Custom Pipes

We can pass any number of parameters to the pipe using a colon (:) and when we do so, it is called Angular Parameterized Pipes.

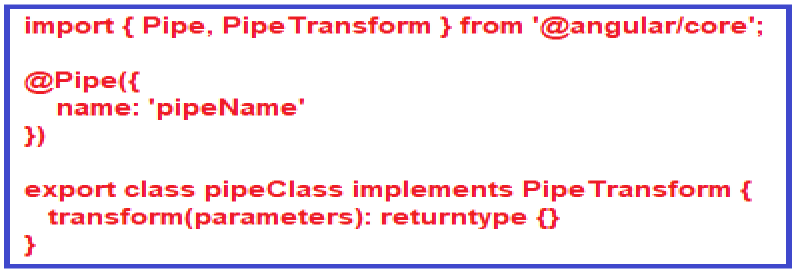


Syntax:
data | pipeName : parameter 1 : parameter 2 : parameter 3 ... parameter n

Examples:
Date:- {{DOB | date : "short"}}
Currency:- {{courseFee | currency : 'USD' : true : '1.3-3'}}

How to create a Custom Pipe? ng g pipe MyTitle –flat

In order to create a custom pipe in angular, you have to apply the @Pipe decorator to a class which you can import from the Angular Core Library. The @Pipe decorator allows you to define the pipe name that you will use within the template expressions



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

@Pipe({
   name: 'pipeName'
})

export class pipeClass implements PipeTransform {
  transform(parameters): returntype {}
}
```

Pure pipes are the pipes that execute when it detects a pure change in the input value. A pure change is when the change detection cycle detects a change to either a primitive input value (such as String, Number, Boolean) or object reference (such as Date, Array, Function, or Object).
Impure pipes are the pipes that execute when it detects an impure change in the input value. An impure change is when the change detection cycle detects a change to composite objects, such as adding an element to the existing array.

What is Eager and Lazy Loading?

The application module i.e. AppModule is loaded eagerly before application starts. Whereas there is something called feature modules which holds all the features of each module which can be loaded in two ways namely eagerly or lazily or even preloaded in some cases.

Eager loading: To load a feature module eagerly we need to import it into an application module using imports metadata of @NgModule decorator. Eager loading is highly used in small size applications. In eager loading, all the feature modules will be loaded before the application starts.This is the reason the subsequent request to the application will always be faster.

Lazy loading: refers to the concept of loading modules **on demand**, rather than loading everything at the start. This means only the necessary HTML, CSS, and JavaScript files are loaded when required, which improves the application's performance and reduces the initial load time. Lazy loading is especially useful as the application grows and includes multiple feature modules. Instead of loading all modules upfront, Angular loads them **only when the user navigates to a route that needs them**. This helps make the application start faster and reduces unnecessary resource usage.

To implement lazy loading, the feature module should **not be imported** in the main `AppModule`. Instead, it should be configured in the routing file using the `loadChildren` property.

To implement lazy loading in Angular, you need to follow three main steps:

1. **Divide your project into separate modules** – Each major feature of your application should have its own module. For example, you might have `StudentsModule`, `TeachersModule`, etc.

2. **Create separate routing files for each module** – In the main routing configuration (`app-routing.module.ts`), use the `loadChildren` property to specify which module should be loaded lazily.

   { path: 'students', loadChildren: () => import('./students/students.module').then(m => m.StudentsModule) }

3. **Use `RouterModule.forRoot()` and `RouterModule.forChild()` properly** – In the main app routing module, use `RouterModule.forRoot()` to configure the root routes. Inside each feature module, use `RouterModule.forChild()` to define the child routes specific to that module.

Routing is a mechanism which is used for navigating between pages & displaying appropriate components or pages on the browser. In order to implement routing we need to use the routing module of angular
Create routing collection which defines the URL path and which component to load for that URL

```
const routes: Routes = [
  {path:'studentLink', component:StudentComponent},
  {path:'studentdetailsLink',component:StudentdetailComponent}];
```

Later this routing collection is loaded using the router module using “forroot” or “forchild”.

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
```

Then we can use router-link directive on html anchor tag or in TS we can navigate using router.navigate

```
<a [routerLink] = "['/studentLink']" >Student</a> <br/>
<a [routerLink] = "['/studentdetailsLink']" >student details</a>
```

Router Outlet is a dynamic component that the router uses to display views based on router navigation. RouterOutlet is one of the router directives <router-outlet> Router-outlet is a reserved tag in which the UI loads when routing happens.

With the help of **routerLink** directive, you can link to routes of your application right from the HTML Template.

RouterLink is a directive used with anchor (`<a>`) tags to enable navigation between different routes defined in your application. It allows you to link to specific paths in the routing configuration directly from your HTML template, making navigation seamless without reloading the page.

The `routerLink` directive takes a string that matches a defined route path. When the user clicks the link, Angular's Router handles the navigation internally. This ensures the routing is managed by Angular itself rather than making a regular HTTP request.

HTML template: <a routerLink="/home">Home Page of our website</a>

Server Side: GetStudent() { this.router.navigate(['/studentLink']);}

Redirecting Routes : When an Angular application starts, it typically navigates to an **empty route (`''`)** by default. To provide a better user experience, you might want to **redirect** this empty path to a specific **default route** — like a login or dashboard page.

Angular allows you to achieve this using the `redirectTo` property in the route configuration.

```ts
const routes: Routes = [
  { path: '', redirectTo: '/login', pathMatch: 'full' },
  { path: 'login', component: LoginComponent },
  { path: 'dashboard', component: DashboardComponent }
];
```

Wildcard Route in Angular uses a path defined as `**`. It is a catch-all route that matches any URL that is not matched by earlier route definitions.

*{ path: '**', component: PageNotFoundComponent }*

This is typically used for:

- Redirecting users to a "Page Not Found" component
- Handling unknown or invalid routes

Route Order: **Static routes** (like `'home'`, `'login'`) should come first. **The empty path (`''`)** for default redirection should come after static routes. **The wildcard route (`'**'`)** should always come last. If the wildcard route is placed earlier, it will catch all requests — preventing the rest of the routes from being accessed.

factory() is a function which works similar to the service() but is much more powerful and flexible. factory() are design patterns which help in creating Objects.

Services in Angular are classes that contain business logic, reusable data, or functions, and are meant to be shared across multiple components in an application. They are a vital part of Angular's architecture and are commonly used to:

Fetch data from APIs
Share logic between components
Manage application-wide state
Handle utility functions (e.g., validation, formatting)

An Angular service typically includes:

An **exported class**
Decorated with **`@Injectable()`**
Imported from `@angular/core`

Creating a Service (CLI Command): ng generate service Student

Angular uses a **Dependency Injection (DI)** system to inject services into components

Where to Call a Service? — Constructor vs ngOnInit

- **Constructor** is used to inject dependencies (like services), but **should not contain any logic that performs heavy tasks**.

- **ngOnInit** is the right place to:
  Call APIs via service methods
  Run logic that should happen after the component is initialized

Use the constructor to initialize and inject. Use `ngOnInit()` for calling time-consuming tasks like API calls.

Angular provides three different ways to create and register a service with the DI system:

Factory: A function that returns the service object. Useful for dynamic service creation

Service: The most common way — an Angular class decorated with `@Injectable()`

Provider: Allows advanced configuration of service behavior using `useClass`, `useValue`, `useFactory`, or `useExisting`

Service Scope Options

- `providedIn: 'root'` (default) → Available throughout the app.
- `providedIn: SomeModule` → Limited to a specific module.
- Register manually inside the `providers` array of a component/module.

Dependency Injection, **Dependency Injection (DI)** is a design pattern used to supply objects or services that a class needs, instead of creating them directly inside the class. Angular supports and uses DI to increase modularity, flexibility, and maintainability of code.
Dependency: A service or object that a component or class needs to function (e.g., a logging service, data service).

Injection: The act of passing this dependency to the class that needs it.
constructor(private myService: MyService) {} // dependency injected

Why Use Dependency Injection?

- Promotes **loose coupling** between classes
- Makes **unit testing** easier
- Supports **reusability and scalability**
- Centralizes the way objects are created and managed

Angular creates and injects services automatically using the **constructor of a class**

Angular lifecycle hooks are nothing but callback functions, which angular invokes when a certain event occurs during the component's life cycle.

These hooks give you the ability to execute custom logic at different stages of the component's life, such as when it is created, updated, or destroyed.

Angular lifecycle events can be grouped into two categories:

1. **The sequence of events when the component is loaded for the first time**.
2. **The sequence of events that fires on every change detection cycle** after the component is loaded.

First-time Sequence of Events (When the Component Is Loaded for the First Time):

1. **Constructor**: constructor is not an Angular-specific event but rather a TypeScript class feature. It is the first method that gets called when an object of the class is created. In Angular, the constructor is invoked before Angular-specific lifecycle hooks are triggered. This is a good place to inject dependencies.
2. **ngOnChanges**: This hook is triggered when any **data-bound input property** changes. It runs before `ngOnInit`. This is useful for reacting to changes in input properties.
3. **ngOnInit**: This method is called once the component's **input properties** are set, and it is invoked after the first change detection. It's typically used to initialize properties or fetch data that the component requires.
4. **ngDoCheck**: This lifecycle hook is called whenever **Angular's change detection** runs. It allows the developer to implement custom change detection logic. It is invoked after `ngOnInit` and after every change detection cycle.
5. **ngAfterContentInit**: This method is called after Angular **projects external content** (via `<ng-content>`) into the component's view. It's fired only once after the first check of the projected content.
6. **ngAfterContentChecked**: This hook is called after Angular checks the projected content. It is fired after `ngAfterContentInit`, and any time content changes are detected.
7. **ngAfterViewInit**: This lifecycle hook is called after Angular initializes the component's **views** and any **child views** or **projected content**. This is where you can perform initialization that requires access to the component's view.
8. **ngAfterViewChecked**: This hook is triggered after Angular's default change detection and the change detection for the projected content have completed. It's useful for detecting changes that require a re-rendering of the view.

Sequence of Events During Change Detection (After Component Initialization): Once the component is initialized, the lifecycle hooks will continue to fire during every change detection cycle. If no input property has changed, **ngOnChanges** will not be called again. However, these hooks will run on each change detection cycle:

9. **ngOnChanges**: This will be called again whenever there are changes in the input properties.
10. **ngDoCheck**: Called whenever Angular runs change detection. It provides an opportunity to detect changes manually.
11. **ngAfterContentChecked**: Called after the content projection has been checked.
12. **ngAfterViewChecked**: This hook is called after the view has been checked for changes.

Clean-up Phase (When Component is Unloaded):

13. **ngOnDestroy**: This is the **cleanup** phase. It is invoked just before Angular destroys the component or directive. It's where you should unsubscribe from observables, clean up timers, or perform other cleanup tasks to prevent memory leaks

What are the lifecycle hooks for components and directives?

An Angular component has a discrete life-cycle which contains different phases as it transits through birth till death. In order to gain better control of these phases, we can hook into them using the following:

- constructor: It is invoked when a component or directive is created by calling new on the class.
- ngOnChanges: It is invoked whenever there is a change or update in any of the input properties of the component.
- ngOnInit: It is invoked every time a given component is initialized. This hook is only once called in its lifetime after the first ngOnChanges.
- ngDoCheck: It is invoked whenever the change detector of the given component is called. This allows you to implement your own change detection algorithm for the provided component.
- ngOnDestroy: It is invoked right before the component is destroyed by Angular. You can use this hook in order to unsubscribe observables and detach event handlers for avoiding any kind of memory leaks.

Dirty checking refers to the process by which Angular detects changes in component data and updates the view accordingly. This is part of Angular's **change detection mechanism**.

During this process, Angular compares the current values of component properties with their previous values. If it finds any difference, it updates the DOM to reflect the new data. This check happens in every **change detection cycle**.

The term **"dirty"** comes from the idea that once a value has changed, it is considered "dirty," and Angular needs to clean (update) it by syncing the view.

In summary, dirty checking means:

- Angular checks all bound properties.
- It compares current values with previous ones.
- If any change is detected, Angular updates the UI.

provider in Angular is a way to tell the **Dependency Injection (DI)** system how to create or deliver a service or dependency.

It defines **how Angular should supply a value** for a given dependency in your application. A provider can:

- Create and return a new instance of a service.
- Use a specific value, class, or factory function.

Providers are registered using:

- The `@Injectable()` decorator in services.
- The `providers` array in modules or components.
- Or globally with `providedIn: 'root'`.

In short:

- A **provider** tells Angular how to create a service.
- It helps manage **service injection** throughout your app.

What is the difference between a service() and a factory()?

both `service()` and `factory()` are used to create reusable pieces of logic or data that can be injected into other parts of the application. However, they differ in how they are implemented and used.

`service()`

- A `service()` is a **constructor function**.
- Angular uses the `new` keyword to instantiate the service.
- It is simpler and works well when your service mainly holds data or functions.
- You define methods on `this` inside the service.

```
app.service('MyService', function() {
  this.greet = function() {
    return "Hello from Service";
  };
});
```

`factory()`

- A `factory()` is a **function that returns an object**.
- It provides more flexibility because you can define private variables, closures, and custom logic before returning the object.
- It's useful when you want more control over how the service is created or what it returns.

```
app.factory('MyFactory', function() {
  return {
    greet: function() {
      return "Hello from Factory";
    }
  };
});
```

Key Difference:

- `service()` uses `this` and is instantiated with `new`.
- `factory()` returns an object and is more flexible for complex logic.

.Difference between a provider, a service and a factory

Provider	Service	Factory
A provider is a method using which you can pass a portion of your application into app.config	A service is a method that is used to create a service instantiated with the 'new' keyword.	It is a method that is used for creating and configuring services. Here you create an object, add properties to it and then return the same object and pass the factory method into your controller.

$scope: A service injected into controllers and directives to share data and logic between the view and controller.

scope: A property in directive definitions used to define the scope behavior (inherited, isolated, etc.).

AOT stands for **Ahead-of-Time compiler**. It is a compilation method in Angular that **pre-compiles application components and templates during the build process**, before the browser downloads and runs the code.

Benefits of AOT Compilation:

1. **Faster rendering:**
   Components are compiled during the build phase, so the browser loads executable code immediately, reducing startup time.
2. **Smaller Angular framework size:**
   AOT removes the need to include the Angular compiler in the bundle (which is required in JIT - Just-in-Time compilation), resulting in smaller application size.
3. **Early error detection:**
   Errors in templates and bindings are caught at build time, rather than runtime, improving application reliability.

4. **Enhanced security:**
   Since the templates are pre-compiled, it reduces the risk of injection attacks via template expressions.
5. **Tree-shaking friendly:**
   Unused code (like unused directives and components) can be easily removed during the bundling process using tools like Webpack, further reducing the app size.

AOT Works:

- Converts Angular HTML and TypeScript code into efficient JavaScript code **during the build phase**.
- Embeds the templates as JavaScript code into the components.
- Generates highly optimized and executable code.

Angular expressions are code snippets that are usually placed in binding such as {{ expression }} similar to JavaScript. These expressions are used to bind application data to HTML.   Syntax: {{ expression }}

Transpiling refers to the process of conversion of the source code from one programming language to another. In Angular, generally, this conversion is done from TypeScript to JavaScript. It is an implicit process and happens internally.

HTTP Interceptors are a powerful feature provided by Angular's `HttpClient` module. They allow developers to **intercept and manipulate HTTP requests and responses.** We can inject some code into http process pipeline to modify or process the outgoing/incoming http request/response.

Ideal for tasks like: Adding **authorization tokens (JWT), Error handling** and custom messaging, **Logging and monitoring** HTTP traffic, Setting common headers or parameters

Events are directives like `ng-click`, `ng-keyup`, `ng-mouseover`, etc., are used to bind event listeners to HTML elements. These help customize the behavior of elements based on user actions.. Few of the events supported by Angular are listed below:
ng-click   ng-copy   ng-cut   ng-dblclick ng-keydown   ng-keypress
ng-keyup   ng-mousedown   ng-mouseenter   ng-mouseleave
ng-mousemove   ng-mouseover   ng-mouseup   ng-blur

List some tools for testing angular applications?

Karma,   Angular Mocks,   Mocha,   Browserify,   Sion

Observables- A sequence of items that arrive asynchronously over time.
HTTP call - single item
Single item- HTTP response

observable updates in the latest version of Angular?

- Faster Builds
- Angular ESLint Updates
- Internet Explorer Updates
- Webpack 5 Support
- Improved Logging and Reporting
- Updated Language Service Preview
- Updated Hot Module Replacement (HMR) Support

RxJS - Reactive extensions for JS, It’s JS library that uses observables to work with Reactive Programming that deals with asynchronous data calls, call backs

Subscribing- .subscribe() is basically a method on the Observable type. The Observable type is a utility that asynchronously or synchronously streams data to a variety of components or services that have subscribed to the observable.

Subscribe takes 3 methods as parameters each are functions:

- next: For each item being emitted by the observable perform this function
- error: If somewhere in the stream an error is found, do this method provider
- complete: Once all items are complete from the stream, do this method

Describe how you will set, get and clear cookies?

For using cookies in Angular, you need to include a module called ngCookies angular-cookies.js.
To set Cookies – For setting the cookies in a key-value format ‘put’ method is used.

```
cookie.set('nameOfCookie',"cookieValue");
```

To get Cookies – For retrieving the cookies ‘get’ method is used.

```
cookie.get(‘nameOfCookie’);
```

To clear Cookies – For removing cookies the 'remove’ method is used.

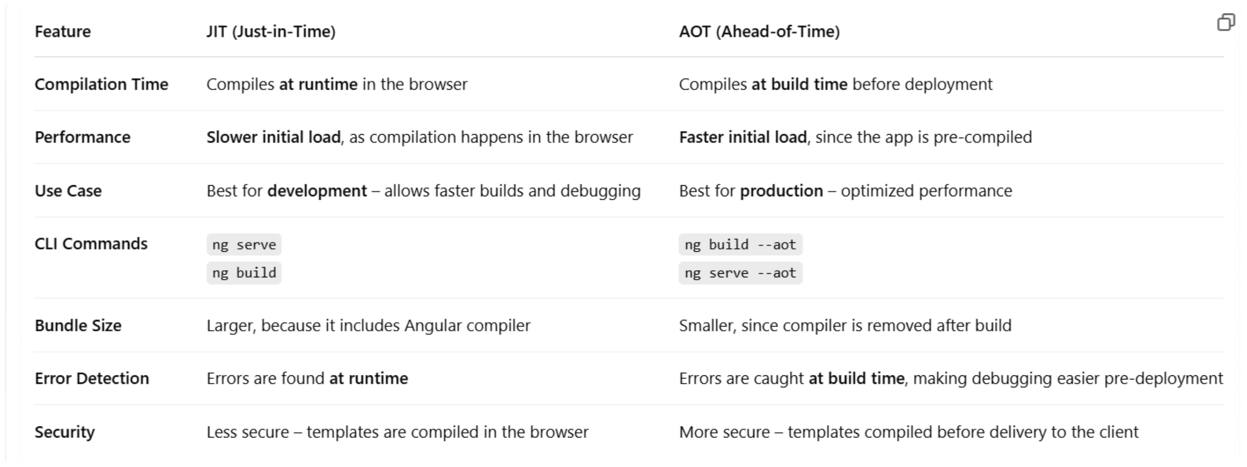
```
cookie.delete(‘nameOfCookie’);
```

Key differences between observables and promises are

Observables	Promises
Emit multiple values over a period of time.	Emit a single value at a time.
Are lazy: they’re not executed until we subscribe to them using the subscribe() method.	Are not lazy: execute immediately after creation.

Have subscriptions that are cancellable using the unsubscribe() method, which stops the listener from receiving further values.	Are not cancellable.
Provide the map for forEach, filter, reduce, retry, and retryWhen operators.	Don’t provide any operations.

List the Differences Between **Just-In-Time (JIT) Compilation and Ahead-Of-Time (AOT) Compilation?**



Feature	JIT (Just-in-Time)	AOT (Ahead-of-Time)
Compilation Time	Compiles **at runtime** in the browser	Compiles **at build time** before deployment
Performance	**Slower initial load**, as compilation happens in the browser	**Faster initial load**, since the app is pre-compiled
Use Case	Best for **development** – allows faster builds and debugging	Best for **production** – optimized performance

CLI Commands	`ng serve`
	`ng build`
		`ng build --aot`
		`ng serve --aot`

Bundle Size	Larger, because it includes Angular compiler	Smaller, since compiler is removed after build
Error Detection	Errors are found **at runtime**	Errors are caught **at build time**, making debugging easier pre-deployment
Security	Less secure – templates are compiled in the browser	More secure – templates compiled before delivery to the client

Angular Material is a user interface component package that enables professionals to create a uniform, appealing, and fully functioning websites, web pages, and web apps. It does this by adhering to contemporary web design concepts such as gentle degradation and browser probability.

What is the purpose of FormBuilder?

The FormBuilder is a syntactic sugar that speeds up the creation of FormControl, FormGroup, and FormArray objects. It cuts down on the amount of boilerplate code required to create complex forms.
When dealing with several forms, manually creating multiple form control instances can get tedious. The FormBuilder service provides easy-to-use control generation methods.
Follow the steps below to use the FormBuilder service:

- Import the FormBuilder class to your project.
- FormBuilder service should be injected.
- Create the contents of the form.

To import the FormBuilder into your project use the following command in the typescript file:

```
import { FormBuilder } from '@angular/forms';
```

Feature	Subject	BehaviorSubject
Holds current value?	❌ No	✅ Yes
Requires default?	❌ No	✅ Yes
Emits on subscribe?	❌ No (if nothing emitted after subscribe)	✅ Immediately emits last value
Use case	Events, fire-and-forget streams	State management, form values, shared data

Subject vs BehaviorSubject

1. Subject (Rx.Subject)

- Does **not hold a current value**.
- Subscribers will **only receive values emitted *after* they subscribe**.
- If a value is emitted before the subscription, it is **lost** for that subscriber.

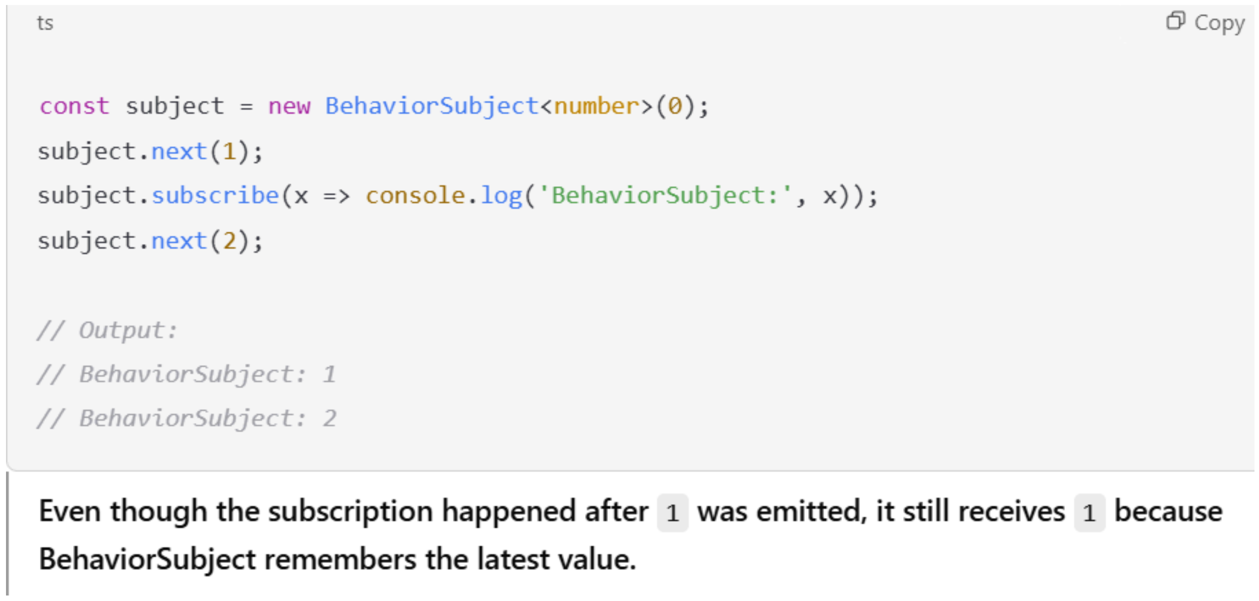
```ts
const subject = new Subject<number>();
subject.next(1);
subject.subscribe(x => console.log('Subject:', x));
subject.next(2);

// Output:
// Subject: 2
```

> `1` is not received because the subscription happened after it was emitted.

2. BehaviorSubject (Rx.BehaviorSubject)

- **Holds a current value**.
- Requires an **initial value**.

- Subscribers receive the **latest value immediately upon subscription**, and all future values as well.



ts

Copy

```ts
const subject = new BehaviorSubject<number>(0);
subject.next(1);
subject.subscribe(x => console.log('BehaviorSubject:', x));
subject.next(2);

// Output:
// BehaviorSubject: 1
// BehaviorSubject: 2
```

> Even though the subscription happened after `1` was emitted, it still receives `1` because BehaviorSubject remembers the latest value.

RxJS Observables	Promises
Observables are used to run asynchronously, and we get the return value multiple times.	Promises are used to run asynchronously, and we get the return value only once.
Observables are lazy.	Promises are not lazy.
Observables can be canceled.	Promises cannot be canceled.
Observables provide multiple future values.	Promises provide a single future value.

Guard is a special type of service in Angular used to control access to routes. It determines whether or not a user can navigate to a particular route. Always place guards on protected routes in your `app-routing.module.ts`

`ng g guard helpers/auth : This generates a guard file like auth.guard.ts`

⚙️ Types of Guards in Angular

Guard Type	Purpose
`CanActivate`	Determines if a route can be activated
`CanActivateChild`	Checks if child routes can be activated
`CanDeactivate`	Confirms navigation away from a route
`CanLoad`	Prevents lazy-loaded module from loading
`Resolve`	Pre-fetches data before route activates

🔑 Basic Example of Auth Guard (JWT Check)

```ts
// auth.guard.ts
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(private router: Router) {}

  canActivate(): boolean {
    const token = localStorage.getItem('jwtToken'); // Or use a proper AuthService
    if (token) {
      return true;
    } else {
      this.router.navigate(['/login']);
      return false;
    }
  }
}
```

🧭 Using the Guard in Routing

```ts
// app-routing.module.ts
const routes: Routes = [
  { path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] },
  { path: 'login', component: LoginComponent },
];
```

Interceptors are part of Angular's **HttpClient module** that allow you to intercept and modify **HTTP requests** and **responses** globally.

ng g interceptor interceptors/auth : This creates: `auth.interceptor.ts` under `src/app/interceptors/`.

Cleaner code, reusable logic, centralized control

You can use them to:

- Attach JWT tokens to requests
- Log requests/responses
- Handle global error messages
- Set common headers

There are five ways to share data between components:

- Parent to child component
- Child to parent component
- Sharing data between sibling components
- Sharing data using ViewChild property
- Sharing data between not related components

1. Parent to Child Component

Use the @Input() decorator in the child component to receive data.

```
// child.component.ts
@Input('childToMaster') masterName: string;

<!-- parent.component.html -->
<app-child [childToMaster]="product.productName"></app-child>
```

2. Child to Parent Component

Use @Output() and EventEmitter in the child to send data back to the parent.

```
// child.component.ts
@Output() childToParent = new EventEmitter<string>();
```

```
sendToParent(name: string) {
  this.childToParent.emit(name);
}

<!-- parent.component.html -->
<app-child
  [childToMaster]="product.productName"
  (childToParent)="childToParent($event)">
</app-child>


// parent.component.ts
childToParent(name: string) {
  this.product.productName = name;
}
```

3. Sibling to Sibling Component

Use the parent component as a mediator:

1. Child A → Parent (via `@Output`)
2. Parent → Child B (via `@Input`)

This ensures decoupling and clean architecture.

4. Using `@ViewChild()`

Gives parent access to child component's methods or properties.

```
// parent.component.ts
@ViewChild(AppChildComponent) child;

ngAfterViewInit() {
  this.product.productName = this.child.masterName;
}
```

- Make sure to access the child only after view init (`ngAfterViewInit`).

5. Sharing Between Unrelated Components

Use a Shared Data Service with `BehaviorSubject`.

shared-data.service.ts

```
@Injectable({ providedIn: 'root' })
export class SharedDataService {
  private messageSource = new BehaviorSubject<string>('default message');
  currentMessage = this.messageSource.asObservable();

  changeMessage(message: string) {
    this.messageSource.next(message);
  }
}
```

Component 1 (Sender)

```
constructor(private sharedDataService: SharedDataService) {}
this.sharedDataService.changeMessage("New Value");
```

Component 2 (Receiver)

```
selectedMessage: string;

ngOnInit() {
  this.sharedDataService.currentMessage.subscribe(message => {
    this.selectedMessage = message;
  });
}
```