Angular

Angular is a popular open-source framework developed by Google for building dynamic, single-page web applications (SPAs). Angular is a framework which provides complete developer platform for Single Page Application and Progressive Web Applications. It uses TypeScript, which is a superset of JavaScript, and provides a powerful set of tools for developing complex applications with a focus on maintainability, scalability, and performance.

Single Page Application: Every time a new route triggers, it won't load whole page, this feature allows to create whole project in one HTML called SPA.

Progressive Web App: These apps work similar in different platforms.

Framework: Framework contains all the packages needed for the application development. (Unlike library which is pre written code used for specific purpose.)

Key Concepts of Angular:

1. **Component-based Architecture**:
   - A component consists of an HTML template for the UI, a TypeScript class for handling logic, and metadata that defines how the component should behave.
   - Angular applications are built using components, which are self-contained building blocks with their own logic and templates.
2. **Modules**:
   - Angular applications are organized into modules. A module is a container that groups related components, services, and other code.
   - The root module, AppModule, bootstraps the Angular application.
3. **Templates and Data Binding**:
   - Angular uses templates to define the view, and it supports powerful data-binding mechanisms.
   - **Interpolation**: Binding data from the component to the template using {{ }}.
   - **Property Binding**: Setting properties of DOM elements with [property]="expression".
   - **Event Binding**: Handling user actions with (event)="handler()".
   - **Two-way Binding**: Synchronizing data between the component and the UI using [(ngModel)].
4. **Services and Dependency Injection**:
   - Angular promotes the use of services to share logic between components. Services are typically used for things like fetching data from APIs.
   - Dependency injection (DI) is a key feature, allowing Angular to efficiently provide instances of services where needed.
5. **Routing**:
   - Angular provides a powerful routing system to build single-page applications. The Angular Router maps URLs to components and helps manage navigation within the app without reloading the page.
6. **Directives**:
   - Directives are special markers in the DOM that tell Angular to attach a specific behavior to an element. Angular provides built-in directives like

*ngIf and *ngFor, and you can create custom directives for reusable behaviors.

7. **Forms**:
   - Angular has two approaches to managing forms:
     - **Template-driven forms**: Simple and easy to use, but less scalable.
     - **Reactive forms**: More powerful and suitable for complex scenarios, offering better control over form validation and dynamic form creation.
8. **Pipes**:
   - Pipes are used to transform data in templates. For example, the | date pipe formats a date, and the | uppercase pipe converts text to uppercase. You can also create custom pipes.
9. **Testing**:
   - Angular encourages a test-driven development approach. It comes with built-in tools for unit testing components and services using frameworks like Jasmine and Karma.
10. **Performance Optimization**:
    - Angular provides various techniques for optimizing performance, including Ahead-of-Time (AOT) compilation, lazy loading of modules, and change detection strategies.

Advantages of Angular:

- **Structured Code**: Angular's architecture promotes clean, maintainable, and testable code.
- **TypeScript**: Leveraging TypeScript allows for better tooling, refactoring, and catching errors at compile time.
- **Large Ecosystem**: Angular has a vast ecosystem of tools, libraries, and extensions.
- **Support from Google**: Being developed and maintained by Google ensures regular updates, strong community support, and stability.

Getting Started with Angular:

To start building Angular applications, you typically use the Angular CLI (Command Line Interface), which provides a powerful set of tools to scaffold projects, generate components, and run development servers.

1. **Install Angular CLI**:

```
npm install -g @angular/cli
```

2. **Create a New Angular Application**:

```
ng new my-angular-app
cd my-angular-app
ng serve
```

3. **Develop**: You can now start building your application by creating components, services, and other features using the Angular CLI and writing code in TypeScript.

   Angular is widely used in the industry for building large-scale applications, and its comprehensive framework helps developers manage complex applications with ease.

Advantages of Angular:

1. **Component-based Architecture**: Promotes reusability, modularity, and organized code structure.
2. **Two-way Data Binding**: Simplifies synchronization between the model and the view.
3. **TypeScript Support**: Provides static typing, better tooling, and early error detection.
4. **Dependency Injection**: Enhances modularity and testability by decoupling components from their dependencies.
5. **Powerful CLI**: Automates tasks like project setup, scaffolding, testing, and building applications.
6. **Built-in Routing and State Management**: Supports advanced navigation and state handling in single-page applications.
7. **Comprehensive Documentation and Community**: Offers extensive resources and community support for problem-solving and learning.
8. **Performance Optimizations**: Features like Ahead-of-Time (AOT) compilation and lazy loading improve app performance.
9. **Cross-platform Development**: Enables building web, mobile (via Ionic), and desktop applications.
10. **Enterprise-ready**: Suited for large-scale, complex applications due to its robustness and scalability.

Disadvantages of Angular:

1. **Steep Learning Curve**: Complex concepts can be overwhelming for beginners.
2. **Verbose Code**: Requires more boilerplate code compared to other frameworks.
3. **Complexity**: Overkill for small or simple applications due to its extensive features.
4. **Performance Issues**: May struggle with very large applications if not optimized properly.
5. **Frequent Updates and Breaking Changes**: Updates often introduce breaking changes that require codebase adjustments.
6. **Heavy Framework**: Larger bundle size can affect initial load times compared to lighter frameworks.
7. **Complex Integration with Legacy Systems**: Challenging to integrate with non-modern JavaScript frameworks or legacy systems.
8. **Verbose Testing**: Writing tests, especially for complex components, can be time-consuming and complicated.
9. **Opinionated Structure**: Enforces specific conventions and architecture, limiting flexibility for developers who prefer more freedom.

Q: Difference Between Angular and AngularJS

Category	Angular	AngularJS
Creator	Google	Google
Language supported	JavaScript and Typescript	JavaScript
Mobile Development friendly	Compatible for mobile-development	Not compatible
Architecture	It uses components and directives.	Support model-view-controller (MVC) and model-view-view-model (MVVM) architectures.
Testing	Supports unit testing with Karma	Testing is done through third-party applications
CLI	Comes with Angular CLI	No support for CLI
Dependency Injection	Uses hierarchal dependency injection	Does not use dependency injection. Uses directives
Performance	Supports server-side rendering which offers a speedy performance	Overall performance is slow as compared to Angular
Example	Gmail and Upwork	Netflix and Lego

Q: Difference Between React & Angular:

Field	React	Angular
	It is a **JavaScript library**.	Angular is **a framework.**
Written	React.js written in JavaScript.	Written in Microsoft's Typescript language, which is a superset of ECMA Script 6 (ES6).
Dependency Injection	React.js Does not use the DI concept.	Angular Hierarchical DI system used.
Routing	Routing is not easy in React JS.	Routing is comparatively easy as compare to React JS.
Scalability	It is highly scalable.	It is less scalable than React JS.
Data Binding	It supports Uni-directional data binding that is one way data binding.	It supports Bi-directional data binding that is two data binding.
DOM	It has virtual DOM.	It has regular DOM.
Testing	It supports Unit Testing.	It supports both Unit testing and Integration testing.
Used as	React.js is a JavaScript library. As it indicates react js updates only the virtual DOM is present and the data flow is always in a single direction.	Angular is a framework. Angular updates the Real DOM and the data flow is ensured in the architecture in both directions.

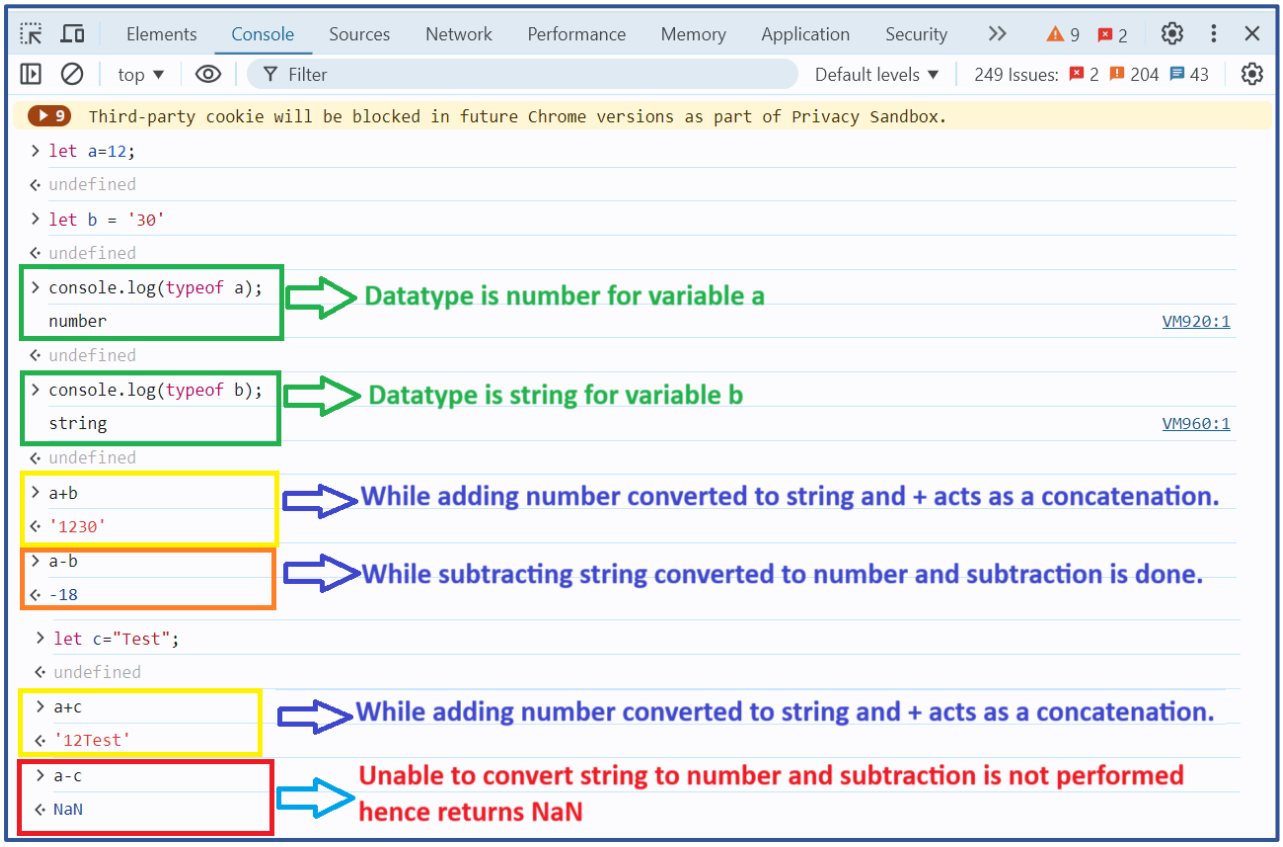
Released	It was released in 2013.	It was released in 2010.
Architecture	React.js is more simplified as it follows MVC ie., Model View Control. This like angular includes features such as navigation but this can be achieved only with certain libraries like Redux and Flux. Needs more configuration and integration.	The architecture of angular on the other hand is a bit complex as it follows MVVM models ie., Model View-ViewModel. This includes lots of tools and other features required for navigation, routing, and various other functionalities.
Performance	React.js holds JSX hence the usage of HTML codes and syntax is enabled. But this doesn't make react js a subset of HTML. This is purely JavaScript-based.	Angular, on the other, is a mere subset of HTML.
Preference	React.js is preferred when the dynamic content needed is intensive. As react js holds more straightforward programming and since it is reliable many apps such as Instagram, Facebook, and Twitter still prefer to react js over angular.	Angular is platform-independent and hence is compatible to work in any platform. Hence, the HTML app which is compatible with all the browsers can prefer angular. One major app which uses angular is YouTube.

Q: Difference Between TypeScript and JavaScript

- TypeScript is known as an Object-oriented programming language whereas JavaScript is a prototype-based language.
- TypeScript has a feature known as Static typing but JavaScript does not support this feature.
- TypeScript supports Interfaces but JavaScript does not.

Feature	**TypeScript**	**JavaScript**
	Developed by Microsoft	Developed by Google
Typing	Provides static typing.	Dynamically typed.
Tooling	Comes with IDEs and code editors.	Limited built-in tooling.
Syntax	Similar to JavaScript, with additional features like static typing.	Standard JavaScript syntax.
Compatibility	Backward compatible with JavaScript.	Cannot run TypeScript in JavaScript files.
Debugging	Stronger typing can help identify errors.	May require more debugging and testing.
Type	Object Oriented Programming Language.	Prototype Based Language.

Learning curve	Can take time to learn additional features.	Standard JavaScript syntax is familiar.
Example	Let a=20; A='Mahesh' //Invalid	Let a=20; A='Mahesh' //Valid
Execution	Browser doesn't understand TypeScript.	Browser understands JavaScript, html, css.
	Needs to be converted to JavaScript before it reaches to browser.	Browser will directly understand the code.

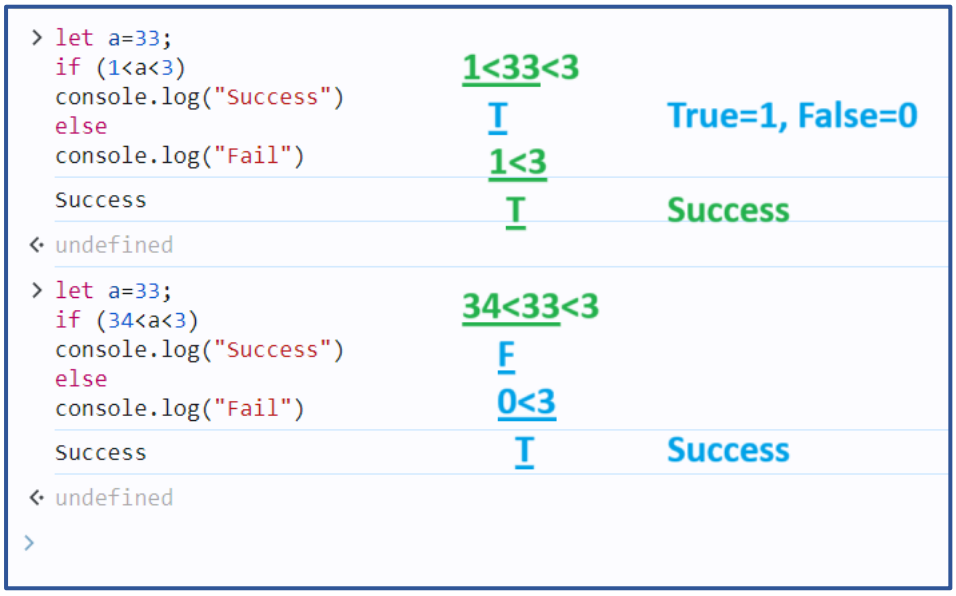


This is the official bug reported over angular website also, mentioning Angular is dynamically typed, meaning that the datatype will change according to the value. And that will create big problem as like below

Ex:

```
let a=33;
if (1<a<3)
console.log("Success")
else
console.log("Fail")
```

Success


let a=33;
if (1<a<3)
console.log("Success")
else
console.log("Fail")
Success
undefined
1<33<3
T
1<3
T
True=1, False=0
Success
let a=33;
if (34<a<3)
console.log("Success")
else
console.log("Fail")
Success
undefined
34<33<3
F
0<3
T
Success

Basics of TypeScript:

TypeScript is an open-source programming language developed by Microsoft that builds on JavaScript by adding static types. TypeScript is a superset of JavaScript, meaning any valid JavaScript code is also valid TypeScript code. However, TypeScript adds additional syntax to allow developers to specify types (such as string, number, boolean, etc.) for variables, function parameters, and return values.

To install TypeScript, you typically use npm (Node Package Manager), which comes with Node.js. (npm cannot be separately installed it comes with Node.js)

1. Install Node.js

- **Node.js**: If you haven't installed Node.js yet, download and install it from nodejs.org. Installing Node.js will also install **npm (Node Package Manager).**
- **To update** npm use: **npm install -g npm**

2. Install TypeScript Globally

- Open your terminal (Command Prompt, PowerShell, or a terminal in your code editor) and run the following command to install TypeScript globally: **npm install -g typescript@latest**

This installs the TypeScript compiler globally on system, so you can use it from any directory.

3. Verify the Installation

- To verify that TypeScript is installed correctly, run the following command in your terminal: **tsc –version**
- To Update Typescript use : **npm install -g typescript**

```
C:\Users\mrmah>npm install -g typescript

added 1 package in 4s
npm notice
npm notice New minor version of npm available! 10.5.0 -> 10.8.3
npm notice Changelog: https://github.com/npm/cli/releases/tag/v10.8.3
npm notice Run npm install -g npm@10.8.3 to update!
npm notice

C:\Users\mrmah>tsc --v
Version 5.5.4
```

4. Compile TypeScript Code

- Once installed, you can compile a TypeScript file (.ts) into JavaScript by using the tsc command. For example: **tsc example.ts**

Optional: Initialize a TypeScript Project

- If you are starting a new TypeScript project, you can initialize it with a tsconfig.json file by running: **tsc --init**

This file helps configure TypeScript options such as the target ECMAScript version, module resolution, and more. Once installed and configured, ready to start developing in TypeScript!

Note:

Npm : Node Package Manager

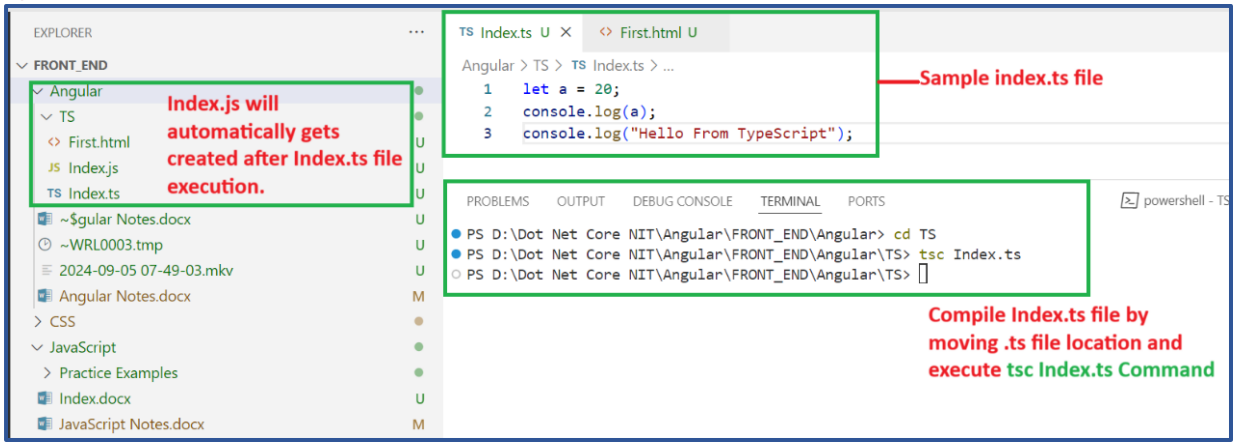
-g : To install it globally in system. If not mentioned it will install in current folder.

@latest/version no: Will install particular version or latest version.

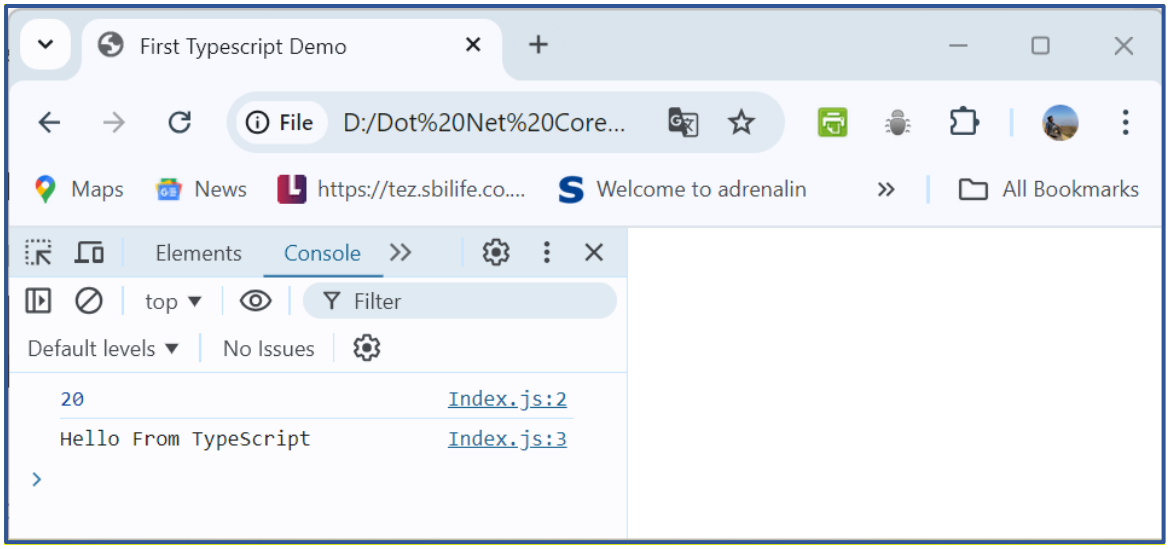
- In Visual Studio Code check the Auto Save option under file menu.
- Browser only understand HTML, CSS, JavaScript, it wont understand TypeScript.
- Install **Live Server** extension in VSCode editor.
- JavaScript Code must be added at the end of body not at the end of head and CSS at end of head section.

First.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script src="Index.js"></script>
</body>
</html>
```



Output:



The watch mode in TypeScript allows you to automatically recompile your TypeScript code whenever changes are made to your .ts files. This is useful during development, as it eliminates the need to manually run the tsc command every time you make a change.

How to Use the watch Command

You can enable watch mode by using the --watch (or -w) flag with the tsc command.

1. Basic Usage

To watch a single TypeScript file for changes and automatically recompile it, run:

tsc --watch <filename>.ts

For example, if you have a file named app.ts, you would run:

tsc --watch app.ts

This will watch the file for any changes and recompile it automatically when changes are detected.

2. Watch an Entire Project

If you have multiple TypeScript files in your project and you want to watch all of them, it's more efficient to use the tsconfig.json configuration file. You can enable watch mode for the entire project by running:

tsc --watch

This will watch all the files specified in your tsconfig.json and automatically recompile them whenever changes are made.

06-09-2024 Friday

Type inference:

Type inference in TypeScript is a feature that allows the TypeScript compiler to automatically determine the type of a variable, function return value, or expression based on the value assigned or the context in which it is used. This means you don't always have to explicitly specify the type, making the code more concise while still maintaining type safety.

How Type Inference Works

TypeScript infers types based on the value assigned to a variable or the return value of a function. This inferred type is then used throughout the code, providing type safety without requiring explicit type annotations.

Example: Variable Type Inference

```
let message = "Hello, TypeScript!"; // TypeScript infers 'message' as string
let count = 42;                     // TypeScript infers 'count' as number
let isActive = true;                // TypeScript infers 'isActive' as boolean
```

In this example:

- message is inferred to be of type string because it is initialized with a string.
- count is inferred to be of type number because it is initialized with a number.
- isActive is inferred to be of type boolean because it is initialized with a boolean.

Example: Function Return Type Inference

TypeScript can also infer the return type of a function based on the values returned within the function.

```
function multiply(a: number, b: number) {
```

```
    return a + b; // TypeScript infers the return type as number
}
let result = multiply(5, 10); // TypeScript infers 'result' as number
```

Here, TypeScript infers that the return type of the multiply function is number because the result of adding two numbers is a number.

Example: Inference with Arrays

TypeScript infers the type of array elements based on the initial values provided.

```
let numbers = [1, 2, 3, 4, 5]; // TypeScript infers numbers as number[]
let fruits = ["apple", "banana", "cherry"]; // TypeScript infers fruits as string[]
```

In this example:

- numbers is inferred to be of type number[] (an array of numbers).
- fruits is inferred to be of type string[] (an array of strings).

Example: Contextual Typing

TypeScript can also infer types based on the context, such as event handling in the DOM.

```
window.addEventListener("click", (event) => {
    console.log(event.clientX); // TypeScript infers 'event' as MouseEvent
});
```

In this case, TypeScript infers that event is of type MouseEvent because it's being used in the context of a click event listener.

Benefits of Type Inference

- **Conciseness**: Reduces the need to explicitly specify types, leading to cleaner and more readable code.
- **Type Safety**: Even without explicit types, TypeScript still checks types at compile-time, catching potential errors.
- **Better Tooling**: With inferred types, editors and IDEs can provide better autocompletion, refactoring, and error-checking features.

When to Use Explicit Types

While type inference is powerful, there are situations where explicitly declaring types might be better:

- **Clarity**: Explicit types can make your code more understandable, especially in complex situations.
- **Public APIs**: When defining public interfaces or APIs, explicit types can help other developers understand how to use your code correctly.
- **Complex Types**: In cases where the inferred type is complex or where the default inference might be too general (e.g., any), explicit typing is useful.

Summary

Type inference in TypeScript allows the compiler to automatically determine types based on the assigned values or context, providing a balance between type safety and code simplicity. While explicit types are sometimes necessary, type inference helps reduce the need for boilerplate type annotations in many cases.

Type Annotation:

Type annotation in TypeScript is the practice of explicitly specifying the type of a variable, function parameter, function return type, or expression. This provides more clarity, improves code readability, and ensures type safety by explicitly defining what types are expected.

How to Use Type Annotations

Type annotations are added by placing a colon : followed by the type after the variable name, function parameter, or function return type.

Example: Variable Type Annotation

```
let name: string = "Alice";
let age: number = 25;
let isStudent: boolean = true;
```

In this example:

- name is explicitly annotated as string.
- age is explicitly annotated as number.
- isStudent is explicitly annotated as boolean.

Example: Function Parameter and Return Type Annotations

You can also use type annotations for function parameters and return types:

```
function greet(name: string): string {
    return "Hello, " + name;
}
let greeting: string = greet("Alice");
console.log(greeting); // Outputs: Hello, Alice
```

In this example:

- The name parameter is annotated as string.
- The function greet is annotated to return a string.

Example: Object Type Annotation

You can annotate the types of properties in an object:

```
let user: { name: string; age: number; isAdmin: boolean } = {
    name: "Alice",
    age: 25,
    isAdmin: true
};
```

In this example:

- The user object is annotated with an object type that specifies name as string, age as number, and isAdmin as boolean.

Example: Array Type Annotation

Type annotations can also be used to specify the type of elements in an array:

```
let numbers: number[] = [1, 2, 3, 4, 5];
let fruits: string[] = ["apple", "banana", "cherry"];
```

In this example:

- numbers is annotated as an array of numbers (number[]).
- fruits is annotated as an array of strings (string[]).

Example: Function Type Annotation

You can annotate both the parameter types and return type for a function type:

```
let add: (a: number, b: number) => number = function(a: number, b: number):
number {
    return a + b;
};
console.log(add(5, 10)); // Outputs: 15
```

In this example:

- add is annotated as a function type that takes two number parameters and returns a number.

Example: Union Type Annotation

TypeScript allows you to specify that a variable can be of multiple types using union types:

```
let value: string | number;
value = "Hello"; // OK
value = 42;      // OK
// value = true; // Error: Type 'boolean' is not assignable to type 'string | number'.
```

In this example:

- value is annotated to be either a string or a number. It can hold values of either type, but nothing else.

Example: Optional Parameter Annotation

You can annotate parameters as optional by using a question mark (?):

```
function greet(name: string, age?: number): string {
    if (age) {
        return `Hello, ${name}. You are ${age} years old.`;
    } else {
        return `Hello, ${name}.`;
    }
}
console.log(greet("Alice")); // Outputs: Hello, Alice.
console.log(greet("Bob", 30)); // Outputs: Hello, Bob. You are 30 years old.
```

In this example:

- The age parameter is optional (age?: number), so it may or may not be provided.

Summary

Type annotations in TypeScript allow you to explicitly specify the types of variables, function parameters, return values, objects, arrays, and more. This enhances type safety, improves code clarity, and makes it easier for others (and tools like IDEs) to understand and work with your code.

Variable declaration and its meaning:

In TypeScript, variable declaration can be done using three main keywords: let, const, and var. Each has its own scope, mutability, and behavior. Here's an overview of how they work:

1. let Declaration:

- **Scope**: Block-scoped (local to the block {} where it's declared).
- **Reassignment**: Can be reassigned.
- **Hoisting**: Hoisted but not initialized, meaning the variable exists in memory but cannot be used before it's declared.

```
let x: number = 5;
x = 10; // Reassignment is allowed
```

2. const Declaration:

- **Scope**: Block-scoped (local to the block {}).
- **Reassignment**: Cannot be reassigned (immutable reference). However, if it's an object or array, the contents can be mutated.
- **Hoisting**: Hoisted but not initialized, similar to let.

```
const y: string = "Hello";
// y = "World"; // Error: cannot reassign a const variable
const obj = { name: "Alice" };
obj.name = "Bob"; // Object contents can be mutated
```

3. var Declaration:

- **Scope**: Function-scoped (if declared within a function) or global-scoped (if declared outside any function). It ignores block-level scope.
- **Reassignment**: Can be reassigned.
- **Hoisting**: Hoisted and initialized, meaning the variable can be used before it's declared, but will return undefined if accessed before the declaration.

```
var z: boolean = true;
z = false; // Reassignment is allowed
function testVar() {
  var inside = "inside";
  console.log(inside); // "inside" is accessible here
}
```

Q. Differences between let, const, and var:

- **Scope**: let and const are block-scoped, while var is function-scoped.
- **Hoisting**: var is hoisted and initialized with undefined, while let and const are hoisted but not initialized (they exist in a "temporal dead zone" until their declaration is reached).
- **Mutability**: const creates immutable bindings (though object properties can be mutated), while let and var allow reassignment.

Type Annotations in TypeScript:

TypeScript allows you to add explicit type annotations when declaring variables. For example:

```
let count: number = 10;      // Variable 'count' is of type 'number'
const name: string = "John"; // Constant 'name' is of type 'string'
```

These annotations help TypeScript's type-checking system to ensure that variables are used correctly throughout the code.

Var	**let**	**const**
The scope of a *var* variable is functional or global scope.	The scope of a *let* variable is block scope.	The scope of a *const* variable is block scope.
It can be updated and re-declared in the same scope.	It can be updated but cannot be re-declared in the same scope.	It can neither be updated or re-declared in any scope.
It can be declared without initialization.	It can be declared without initialization.	It cannot be declared without initialization.

It can be accessed without initialization as its default value is “undefined”.	It cannot be accessed without initialization otherwise it will give ‘referenceError’.	It cannot be accessed without initialization, as it cannot be declared without initialization.
These variables are hoisted.	These variables are hoisted but stay in the temporal dead zone untill the initialization.	These variables are hoisted but stays in the temporal dead zone until the initialization.

When to Use let and const

var can be tricky because its scope is either global or within a function, which can lead to bugs. To avoid these issues:

- Use **let** when you know a variable’s value might change later in your code.
- Use **const** for variables that should never change once you set them.

Using let and const makes your code easier to understand and helps prevent errors caused by unexpected variable changes.

Arrays in TypeScript:

In TypeScript, arrays can be categorized based on the types of elements they hold and how they are defined. Here's a breakdown of the different types of arrays in TypeScript:

1. Homogeneous Array

A homogeneous array is an array where all elements are of the same type.

Example:

```
let numbers: number[] = [1, 2, 3, 4, 5];
let strings: string[] = ["apple", "banana", "cherry"];
```

In this case, both numbers and strings arrays contain elements of a single type (number and string, respectively).

Alternate Syntax:

```
let numbers: Array<number> = [1, 2, 3, 4, 5];
```

Here, the Array<number> syntax is another way of declaring an array of numbers.

2. Heterogeneous Array

A heterogeneous array allows different types of elements within the same array. This can be achieved using **union types** or **tuples**.

Using Union Types:

```
let mixedArray: (number | string | boolean)[] = [1, "hello", true, 2, "world"];
```

This array can hold number, string, and boolean types.

Using Tuples:

```
let tupleArray: [number, string, boolean] = [1, "Alice", true];
```

Tuples enforce the exact type and order of elements.

3. Array of Arrays (Multidimensional Array)

In TypeScript, you can create an array of arrays, which is often referred to as a multidimensional array. This type is commonly used for matrices or grid-like data.

Example (2D array):

```
let matrix: number[][] = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
```

This represents a 2D array (array of arrays), where each inner array is a row in the matrix.

4. Array of Objects

You can also have arrays where each element is an object. This is common when dealing with data like JSON.

Example:

```
interface Person {
  name: string;
  age: number;
}
let people: Person[] = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 }
];
```

In this example, people is an array of Person objects, where each object has properties name and age.

5. Readonly Array

A **readonly** array is an immutable array whose elements cannot be modified once assigned. You can declare this using readonly keyword.

Example:

```
let readonlyNumbers: readonly number[] = [1, 2, 3, 4];
// readonlyNumbers[0] = 10; // Error: Cannot assign to '0' because it is a read-only property.
```

In this case, you cannot modify the elements of the readonlyNumbers array.

6. Generic Arrays

TypeScript arrays can also be created using **generics**, which makes the array flexible and reusable across different data types.

Example:

```
function getArray<T>(items: T[]): T[] {
  return new Array().concat(items);
}

let numberArray = getArray<number>([1, 2, 3, 4]);
let stringArray = getArray<string>(["apple", "banana", "cherry"]);
```

In this example, the getArray function is a generic function that can work with any type of array, depending on what type T is provided.

7. Array with Optional Elements (Sparse Array)

A sparse array is an array in which some of the elements are intentionally left undefined.

Example:

```
let sparseArray: number[] = [1, , 3, 4];
console.log(sparseArray); // Output: [1, empty, 3, 4]
```

In this array, the second element is intentionally left empty.

Summary of Types:

- **Homogeneous Array**: All elements are of the same type.
- **Heterogeneous Array**: Contains elements of different types (using union types or tuples).
- **Array of Arrays**: Multidimensional arrays or nested arrays.
- **Array of Objects**: Arrays containing objects, often used with interfaces.
- **Readonly Array**: Immutable arrays that cannot be modified.
- **Generic Arrays**: Flexible arrays defined with generics to work with any type.
- **Sparse Arrays**: Arrays with missing or undefined elements.

Each of these array types has different use cases, depending on the structure and requirements of the data you are working with.

Continued Angular:

Note: Angular use command line interface to generate / delete/update any of its components, directives etc..,

This technique of using command prompt for everything is called scaffolding.

IQ :

What is Scaffolding in Angular?

Scaffolding refers to the automatic generation of code or project structure using the Angular CLI (Command Line Interface). It allows developers to quickly create components, services, modules, and other Angular artifacts with predefined templates and structures. This helps maintain consistency and speeds up development by reducing the manual effort of writing boilerplate code.

Key Scaffolding Commands in Angular:

How to Scaffold in Angular

To scaffold an Angular project or component, you use the ng generate command or its shortcut ng g. Here are some commonly used commands and examples:

1. Scaffold an Angular Project

To create a new Angular project, you run the following command: bash

ng new my-angular-app

This command scaffolds the entire project structure with a basic setup, including the following:

- src/ folder containing the main application code
- app/ folder with an initial component
- Configuration files such as angular.json, package.json, and tsconfig.json

2. Scaffold Components

Components are fundamental building blocks in Angular. You can scaffold new components like this: bash

ng generate component my-component

or using the shorthand: bash

ng g c my-component

This command generates:

- A TypeScript file (my-component.component.ts)
- An HTML template (my-component.component.html)
- A CSS or SCSS stylesheet (my-component.component.css)
- A test file (my-component.component.spec.ts)

Example: Running ng g c header will generate the following: bash

src/app/header/header.component.ts
src/app/header/header.component.html
src/app/header/header.component.css
src/app/header/header.component.spec.ts

3. Scaffold Services

Angular services handle business logic and data management. To scaffold a service, use: bash

ng generate service my-service

or the shorthand: bash

ng g s my-service

This creates a TypeScript file for the service with basic boilerplate code: bash
src/app/my-service.service.ts

4. Scaffold Modules

Modules organize related components, services, and other Angular constructs. To scaffold a module: bash

ng generate module my-module

or the shorthand: bash

ng g m my-module

This command generates a new module file: bash
src/app/my-module/my-module.module.ts

5. Scaffold Other Angular Artifacts

Angular CLI can scaffold other constructs such as **directives**, **pipes**, **guards**, and **classes**.

- **Directives**: bash
  ng g directive my-directive
- **Pipes**: bash
  ng g pipe my-pipe
- **Guards**: bash
  ng g guard my-guard

Example Workflow:

Let's say you want to create a new Angular project for an online store. You can scaffold it as follows:

1. **Create a new Angular project**: bash
   ng new online-store
2. **Generate a product component**: bash
   cd online-store
   ng g c product
3. **Generate a service to fetch product data**: bash
   ng g s product-service
4. **Generate a module to group related components**: bash
   ng g m products

This scaffolding process saves time by creating all the required files with boilerplate code and directory structure for you, allowing you to focus on writing the business logic.

Benefits of Scaffolding in Angular:

1. **Speeds up Development**: Reduces manual work and automates the creation of files and structure.
2. **Consistency**: Ensures that all generated components, services, and other parts follow Angular best practices and conventions.
3. **Reduces Errors**: By using predefined templates, scaffolding minimizes the risk of errors in boilerplate code.
4. **Focus on Business Logic**: Developers can focus on core functionality instead of writing repetitive code.

Scaffolding in Angular enhances productivity by quickly generating the necessary components and services for your project.

Angular 17

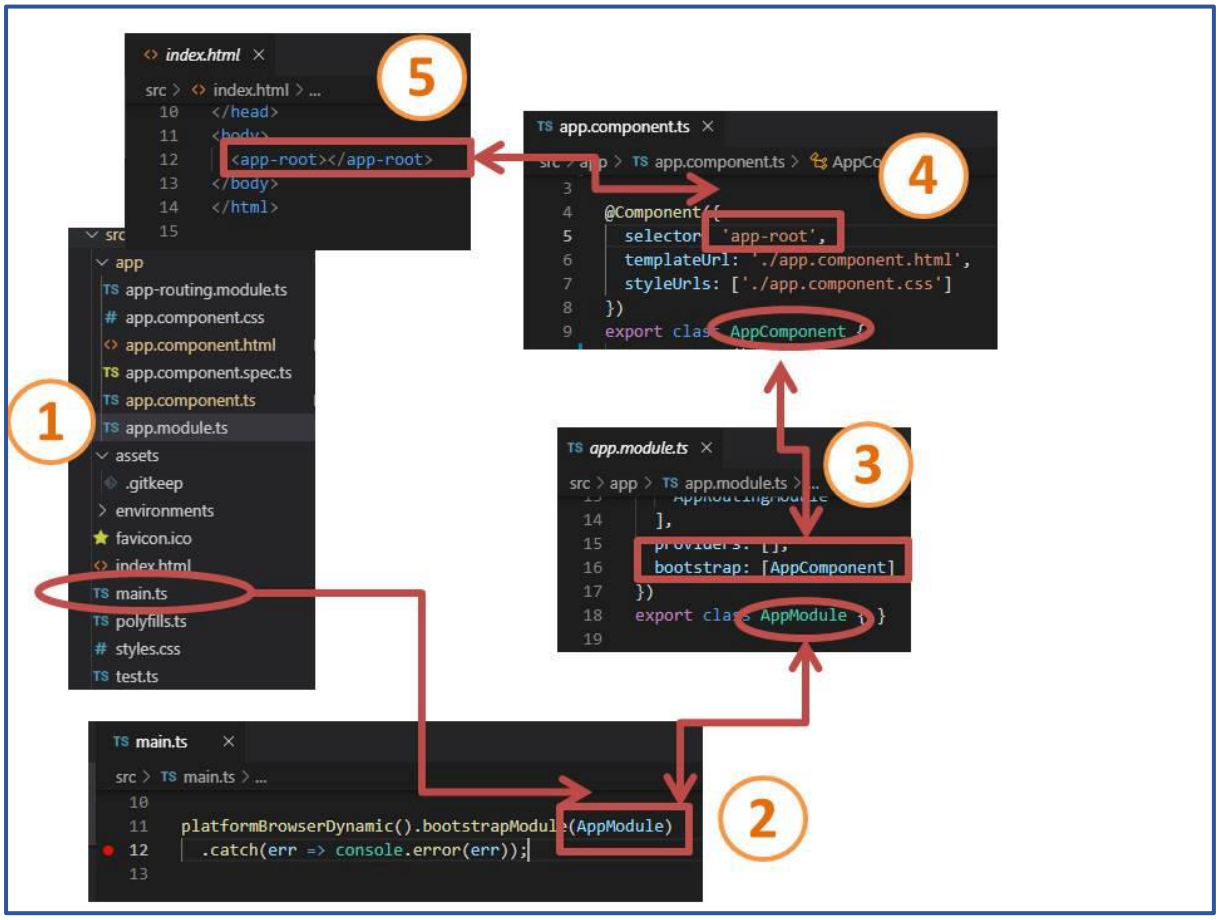
https://www.youtube.com/watch?v=uJIbc2YE58E ARC Tutorial

Angular Folder Structure and Files

1. Parent folder will be the main project folder
2. .angular : Ignore this folder (internally used for caching, memory management etc)
3. .vscode : Ignore this folder (internally used for caching, memory management etc)
4. Node_modules :
    a. Packages will be installed in this folder whenever you install/add new packages using npm install or ng add.
    b. You don't have to go through these folder or files.
    c. Unless and until you don't make any changes to core libraries or modules.
5. .editorconfig: Make your custom editor changes in this file.
6. .gitignore: we can add fodders/files that we want to ignore while committing to git.
7. angulare.json :
    a. this is the file configured styles, js for deploying in a pipeline.
    b. It is having all the configuration details of angular application like what is the version, where is the schema located, what is the prefix that you want to add to angular project, root:"" meaning it's a parent directory, schematics decides what kind of style we want to design that could be css, scss etc, we can add different configuration specific to the different projects separately.
    c. Under the architect we can see the different settings while build , serve , production, development with default defaultconfiguration.
8. package.json :
    a. In this we can het various entries of new packages installed along with there versions.
    b. When we run npm install inside project : the module listed will be installed
    c. here we add our scripts that can be used to run application with single command and multiple settings. Like "arc-build":"ng serve && json-server –watch db.json"
9. package-lock.json :
    a. same details of package.json + dev dependencies broken down in details.
    b. Don't touch this manually
10. Tsconfig.app.json :
    a. Tells you the typescript configuration for your project
    b. Don't touch this manually – for dev purpose
11. Tsconfig.spec.json: typescript test specific configuration
12. ReadMe.md : starting file : documentation of your project
13. SRC :
    a. Source code of project
    b. App :
        i. This is actual code of project/ application
        ii. Every component in Angular has 4 files
            1. .html : Template / HTML code
            2. .scss/.css/.less : Styles
            3. .spec.ts : used for unit testing
            4. .ts : Component class / logical piece of component.
        iii. App.component.ts : It has selector attribute with app-root value and which will ring the bell app-root in index.html i.e. the code present inside the app will be injected dynamically as a SPA inside the index.html.

iv. App.component.spec.ts : There is no end to end Unit framework for unit testing it is shifted to Jasmine
    1. Jasmine is for writing unit tests
    2. Karma is to Test Runner

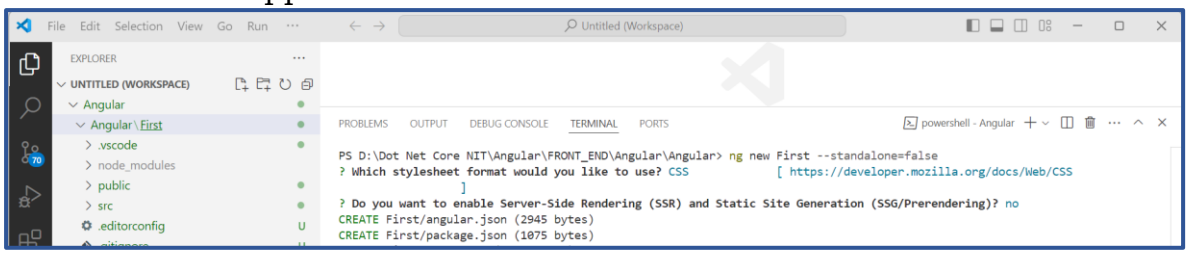
v.



c. Assets : use this folder to serve the assets which are public and may contains images, videos, js files that are kept as public.
d. Favicon : favicon for application specific.
e. Index.html : Angular is a SPA(Single Page Application), there is only one html file index.html. when we develop/build the app : the index.html contains only **<app-root>** First component to be initialized.
f. main.ts : Entry point to project. This is the first file to be called which decides which component to be rendered next ex: 1. Main.ts – picks appComponent and go to index.html and into <app-root></app-root>
g. styles.scss : Global styling for your project. The extension varies from project to project like .css/.less/.scss etc. depends on your initial setup for specific project.

Creating First Application in Angular:

1) Goto specific folder where you want to create Angular application and write
   ng new <app_Name> --standalone=false
   Note:
   ng : Next Generation
   new : used for new element
   app_Name : Application Name
   --standalone=false : use --standalone=false when you are working with the traditional Angular structure that relies on modules and when you want the component to be declared inside an NgModule, as was the standard in Angular versions prior to v14.
   --standalone=false means that the generated component or directive will **not be standalone**. It will be part of an Angular module, and you will need to declare it within an NgModule.
   Ex:
   ng new First --standalone=false
   First : Name of application



2) To run application use
   ng serve --o
   serve : To run application
   --o will open the application in browser with default port number
3) Delete default code available in app.component.html file, and write your own without writing boiler plate code as the boiler plate code is already in index.html and only app.component.html code will be injected.

```
<div class="card">
  <img
src="https://pbs.twimg.com/profile_images/787106179482869760/CwwG2e2M_400x400.jpg" alt="Mahesh PRofile photo">
  <div><strong>Name : </strong> Mahesh Baradkar</div>
  <div><strong>Email : </strong>agilemahesh33&#64;gmail.com</div>
  <div class="socialLinks">
    <a href="https://www.facebook.com">facebook</a>
    <a href="https://www.LinkedIn.com">LinkedIn</a>
    <a href="https://www.GitHub.com">GitHub</a>
  </div>
</div>
```

4) Add css styles into app.component.css.

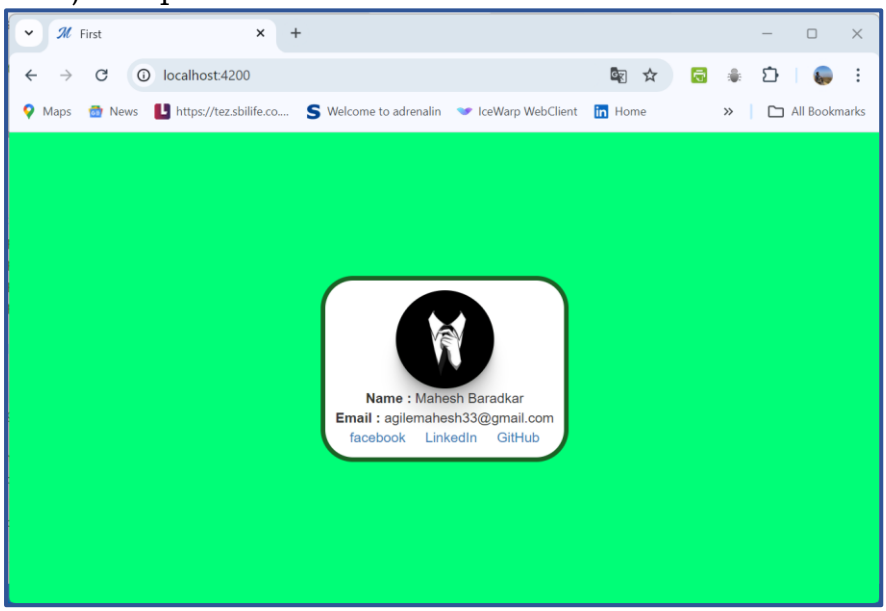
```
.card{
    padding: 10px;
    display: flex;
```

```
    flex-direction: column;
    justify-content: center;
    align-items: center;
    height: max-content;
    background-color: white;
    border: 5px solid #1c6125;
    border-radius: 33px;
}
img{
    height: 100px;
    width: 100px;
    border-radius: 50px;
    box-shadow: rgba(29, 236, 10, 0.17) 0px -23px 25px 0px inset, rgba(0, 0, 0, 0.15) 0px -36px 30px 0px inset, rgba(0, 0, 0, 0.1) 0px -79px 40px 0px inset, rgba(0, 0, 0, 0.06) 0px 2px 1px, rgba(0, 0, 0, 0.09) 0px 4px 2px, rgba(0, 0, 0, 0.09) 0px 8px 4px, rgba(0, 0, 0, 0.09) 0px 16px 8px, rgba(0, 0, 0, 0.09) 0px 32px 16px;
}
.socialLinks{
    display: flex;
    justify-content: space-between;
    gap: 20px;
}
```

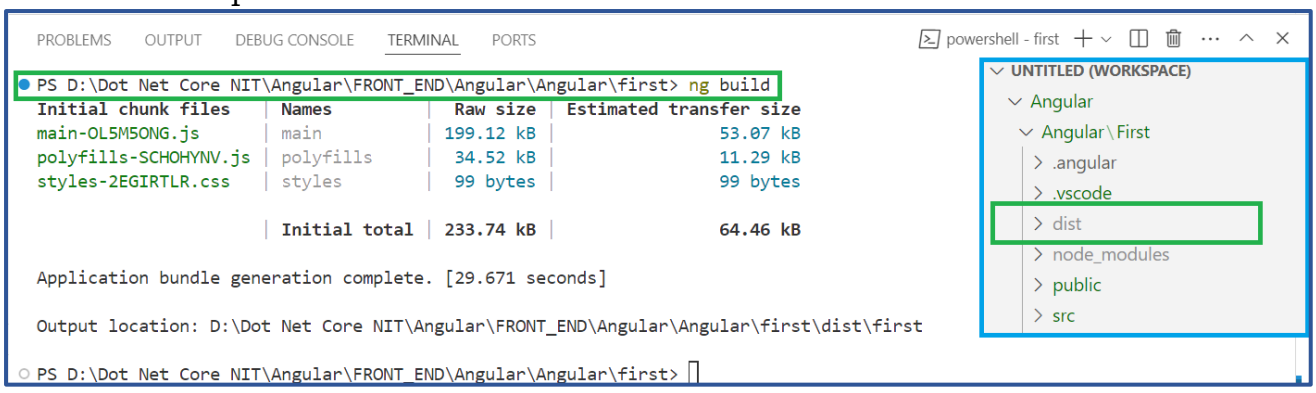
5) Add common Global style Styles.css

```
/* You can add global styles to this file, and also import other style files */
body{
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    background-color:rgb(0, 254, 119)
}
```

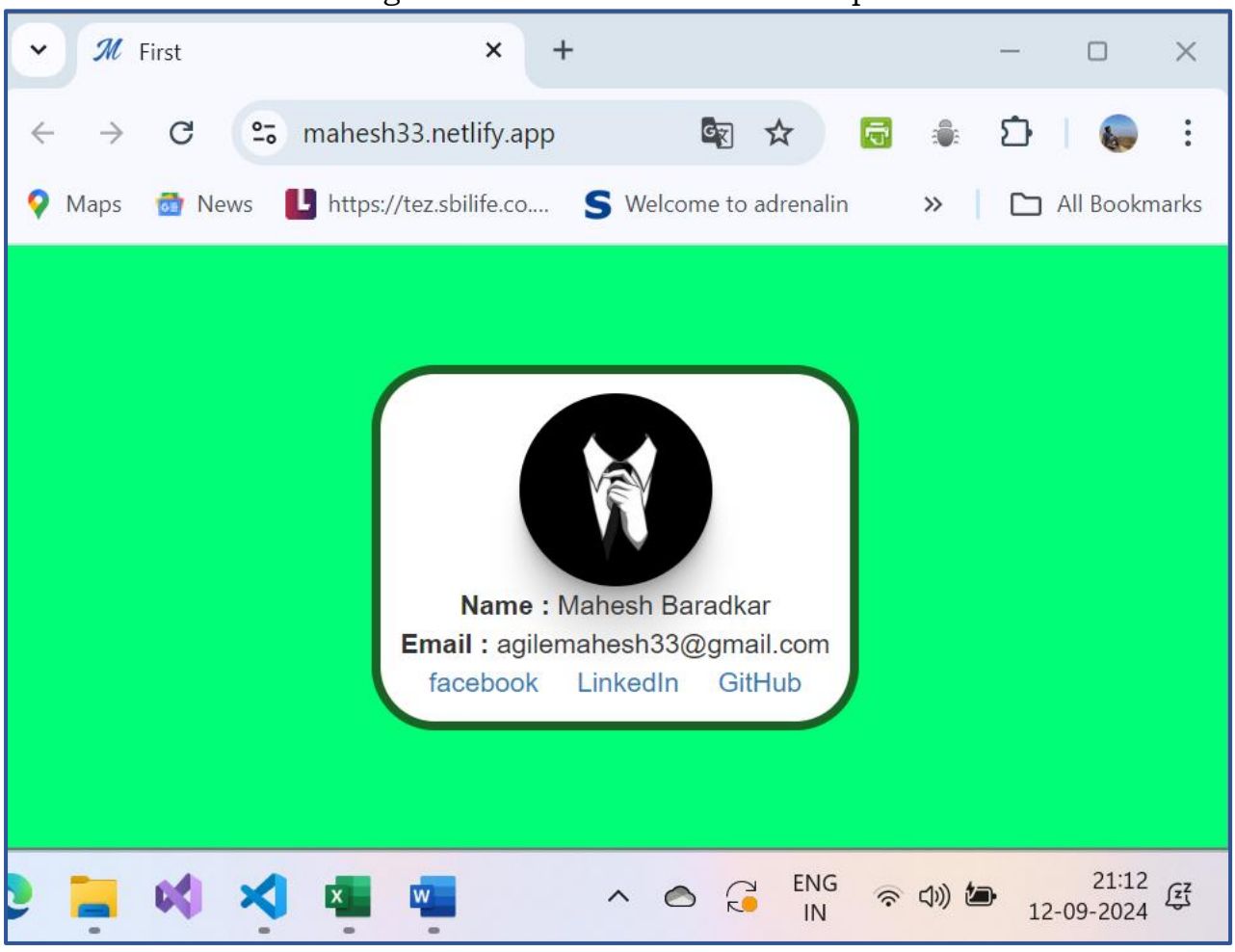
6) Output of the above will look like



7) To push the above code to live server follow below steps:
   a. Stop current execution by pressing ctrl+c in VS Code.
   b. To build the project execute "ng build" command so that the project will be ready to move on live server.
   c. "ng build" command also creates dist folder which can then be directly copied to the live server.



   d. Create or login to https://app.netlify.com/ for temporary deployment.
   e. Upload dist folder that is created in step c. which will generate new url for the browser.
   f. We can change URL but that should be unique.



8)

Data Binding:

https://www.scholarhat.com/tutorial/angular/angular-interview-questions-and-answers

In every component, there will be one HTML file also called as Template/View file and one ts file also called component/model file.

Data Binding refers to the synchronization between the model (component) and the view (HTML template) i.e. transferring data between template and component files.

There are two types of data binding in Angular:

A. **One Way Data Binding:** Using one way data binding, we can transfer data from either template to component or component to template, there are three possible ways:

1. **String Interpolation (One-Way Data Binding)**

   String Interpolation allows you to display data from the component (TS) in the view (HTML). We use {{}} to bind the data from the component to the view.

   Example:

```
// app.component.ts
export class AppComponent {
  title = 'Angular Data Binding Example';
}
<!-- app.component.html -->
<h1>{{ title }}</h1>
```

Explanation:

The value of title in the component will be displayed inside the <h1> tag.

Note: String Interpolation always takes value as a string in terms of properties and other than number i.e. numbers are treated as numbers only, Booleans is treated as string. So instead of sending string always we can send property values by property binding.

In details other examples :

Interpolation in Angular is a technique that allows you to display dynamic data from the component in the HTML template. It is a type of one-way data binding where the data flows from the component (the TypeScript class) to the view (the HTML template).

Syntax:

Interpolation is done using double curly braces: {{expression}}.

The expression inside the curly braces is typically a property or method from the component class.

Key Points:

- Interpolation can only bind data **from** the component to the view (one-way).
- It evaluates expressions like property access, method calls, or simple calculations.
- You can use any valid JavaScript expression in the interpolation except for statements like if, for, or while.

Basic Example:

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  title = 'Welcome to Angular!';
  currentDate = new Date();
}
<!-- app.component.html -->
<h1>{{ title }}</h1>
<p>Today's date is: {{ currentDate }}</p>
```

Explanation:
- {{ title }} will display "Welcome to Angular!" because the title property is defined in the component.
- {{ currentDate }} will display the current date.

Using Methods in Interpolation:
You can also call methods in interpolation, though it's not recommended for complex logic because the method is called every time Angular renders the view.
Example:
```
// app.component.ts
export class AppComponent {
  title = 'Angular App';

  getWelcomeMessage() {
    return `Hello, welcome to ${this.title}`;
  }
}
<!-- app.component.html -->
<p>{{ getWelcomeMessage() }}</p>
```
Explanation:
- {{ getWelcomeMessage() }} will display "Hello, welcome to Angular App" by calling the method from the component.

Expressions in Interpolation:
You can also perform simple arithmetic or string concatenation within the interpolation.
Example:
```
// app.component.ts
export class AppComponent {
  num1 = 10;
  num2 = 20;
}
<!-- app.component.html -->
<p>The sum is: {{ num1 + num2 }}</p>
<p>{{ 'The total is: ' + (num1 + num2) }}</p>
```
Explanation:
- {{ num1 + num2 }} will display 30.
- {{ 'The total is: ' + (num1 + num2) }} will display The total is: 30.

Conditional Display Using Interpolation:
You can use interpolation with ternary operators to conditionally display content.
Example:
```
// app.component.ts
export class AppComponent {
  isLoggedIn = true;
}
<!-- app.component.html -->
<p>{{ isLoggedIn ? 'Welcome back!' : 'Please log in.' }}</p>
```
Explanation:
- If isLoggedIn is true, it will display "Welcome back!".
- If isLoggedIn is false, it will display "Please log in."

Limitations of Interpolation:
- **Cannot use event handling**: Interpolation is purely for output. You cannot use it to handle events (for that, you'd need event binding).
- **Cannot set attributes**: Interpolation cannot be used to bind attributes such as disabled, src, etc. For that, you need **property binding**.

Example with Styling:
Although interpolation is limited to displaying data, you can combine it with inline styles for dynamic styling (although **property binding** is more suited for this task).

```
// app.component.ts
export class AppComponent {
  color = 'red';
}
<!-- app.component.html -->
<p style="color: {{ color }}">This text will be red.</p>
```

Explanation:
- The text will be displayed in red because the color variable is set to red.

Summary:
- Interpolation is a simple, one-way data binding technique that injects component data into the template.
- You can use it to display variables, call methods, and perform basic calculations directly in the template.
- Interpolation cannot handle complex logic or bind attributes or events.

2. **Property Binding (One-Way Data Binding)**
   Property binding binds an element's property to a value in the component. We can bind property value from component file to template file using property binding. It wraps the data between square braces [].
   Example:

```
// app.component.ts
export class AppComponent {
  imageUrl = 'https://example.com/image.png';
}
<!-- app.component.html -->
<img [src]="imageUrl" alt="Image">
```

   Explanation:
- The [src] property of the <img> tag is bound to the imageUrl property in the component. It will dynamically update when the imageUrl changes.

Note: Property Binding can be used for dynamic classes or dynamic styling.

In Details other Examples :

Property Binding in Angular allows you to bind values from the component to the HTML element properties. This is also a form of one-way data binding where data flows from the component to the view.
In property binding, instead of using interpolation {{ }}, you bind the property of an HTML element or directive using square brackets [ ].
Syntax:
[elementProperty]="componentProperty"
- elementProperty: The DOM property or directive you want to bind to (e.g., src, disabled, href, etc.).
- componentProperty: The component's property whose value is passed to the element property.

Key Points:
- Property binding updates the DOM property of an element based on the value in the component.
- It's useful for setting properties like src, disabled, value, etc., which cannot be done via interpolation.

- The property is dynamically updated whenever the component data changes.

Basic Example:
Let's bind the src property of an <img> tag to display an image dynamically.

```
// app.component.ts
export class AppComponent {
  imageUrl = 'https://angular.io/assets/images/logos/angular/angular.png';
}
<!-- app.component.html -->
<img [src]="imageUrl" alt="Angular Logo">
```

Explanation:

- The src property of the <img> tag is dynamically set to the value of the imageUrl variable from the component.

Property Binding with Boolean Properties:
Property binding is commonly used with boolean properties like disabled, checked, hidden, etc.
Example:

```
// app.component.ts
export class AppComponent {
  isButtonDisabled = true;
}
<!-- app.component.html -->
<button [disabled]="isButtonDisabled">Click Me</button>
```

Explanation:

- The disabled property of the button is bound to isButtonDisabled. When isButtonDisabled is true, the button is disabled, and when false, it becomes clickable.

Dynamic Class and Style Binding:
You can also use property binding to dynamically apply CSS classes and inline styles.
Binding to class:

```
// app.component.ts
export class AppComponent {
  isActive = true;
}
<!-- app.component.html -->
<p [class.active]="isActive">This paragraph is active.</p>
```

Explanation:

- The active class is applied to the <p> element only when isActive is true.

Binding to style:

```
// app.component.ts
export class AppComponent {
  backgroundColor = 'lightblue';
}
<!-- app.component.html -->
<p [style.background-color]="backgroundColor">This paragraph has dynamic background color.</p>
```

Explanation:

- The background-color style of the paragraph is dynamically set based on the backgroundColor property from the component.

Setting Attribute Values:
Although interpolation can't bind element attributes, property binding can bind both DOM properties and attributes.
Example (href for a link):

```
// app.component.ts
export class AppComponent {
  websiteUrl = 'https://angular.io';
}
<!-- app.component.html -->
<a [href]="websiteUrl">Go to Angular website</a>
```

Explanation:

- The href attribute is bound to websiteUrl. The link will point to https://angular.io.

Conditional Property Binding:

You can use property binding with ternary operators or logical expressions to make the element properties dynamic.

Example:

```
// app.component.ts
export class AppComponent {
  isSpecial = true;
  size = 16;
}
<!-- app.component.html -->
<p [class.special]="isSpecial">This paragraph has a special class applied.</p>
<p [style.font-size.px]="size">This text has dynamic font size.</p>
```

Explanation:

- The class special is applied only if isSpecial is true.
- The font size is dynamically bound to the size property (16px).

Property Binding with Components:

You can also bind properties between parent and child components. When you pass data from a parent component to a child component, you use property binding.

Parent Component (Passing data to child):

```
// parent.component.ts
export class ParentComponent {
  parentData = 'Data from Parent';
}
```

Child Component (Receiving data):

```
// child.component.ts
import { Input } from '@angular/core';

export class ChildComponent {
  @Input() childInput: string;
}
```

Parent HTML (Binding data to child component's input):

```
<!-- parent.component.html -->
<app-child [childInput]="parentData"></app-child>
```

Explanation:

- The parent component passes the value of parentData to the child component's childInput property via property binding.

Avoiding Common Pitfalls:

1. **Using Square Brackets:**
   Always use square brackets for property binding (e.g., [src] instead of src). Without the brackets, Angular will treat it as a string literal.

```
html
<!-- Correct -->
<img [src]="imageUrl">

<!-- Incorrect -->
```

<img src="imageUrl"> <!-- This would be interpreted as a string "imageUrl" -->

2. **Binding to DOM Properties vs. Attributes:**
   Property binding works with DOM properties, not HTML attributes. For example, disabled is a DOM property, but aria-label is an attribute.
3. **Avoid Calling Functions in Property Binding:**
   While it's possible to call methods in property binding (just like in interpolation), it can lead to performance issues as the method is called every time Angular re-renders the view. It's better to use a component property to store the result and bind to that property.

Summary:

- Property binding is a powerful technique for binding DOM properties or attributes to component data.
- It allows you to dynamically set values like src, href, disabled, class, style, and more.
- Always use square brackets for property binding and avoid using methods for complex calculations inside the bindings.

3. **Event Binding (One-Way Data Binding)**
   Event binding allows you to listen to user events like clicks, keypresses, etc., and call component methods in response. We can bind events data from template file to component file using event binding.
   It wraps the data in to rounded braces().
   Example:

```
// app.component.ts
export class AppComponent {
  message = '';
  handleClick() {
    this.message = 'Button clicked!';
  }
}
<!-- app.component.html -->
<button (click)="handleClick()">Click Me</button>
<p>{{ message }}</p>
```

Explanation:

- When the button is clicked, the handleClick() method is triggered, which updates the message. The updated message is displayed using interpolation.

In Details other Examples :

Event Binding in Angular is used to listen to and respond to user events like clicks, key presses, mouse movements, etc. It allows the component to handle events from the DOM elements (such as buttons, input fields, etc.) by executing methods in the component class.
In event binding, you bind an event of a DOM element to a method in the component using parentheses ().
Syntax:
(elementEvent)="componentMethod($event)"

- elementEvent: The DOM event you want to listen for (e.g., click, input, change, etc.).
- componentMethod: The method in the component that will be executed when the event occurs.

- $event: A special variable that captures event-specific data (optional).

Basic Example:
In this example, we'll handle a button click event and update a property in the component.
Component:

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  message = '';

  onButtonClick() {
    this.message = 'Button clicked!';
  }
}
```

Template:

```
<!-- app.component.html -->
<button (click)="onButtonClick()">Click Me</button>
<p>{{ message }}</p>
```

Explanation:

- The click event of the button is bound to the onButtonClick() method. When the button is clicked, the method updates the message property, which is displayed in the paragraph using interpolation.

Using $event in Event Binding:
You can capture the event object using $event. This is useful when you need to access event-related information, such as the input value, key pressed, or mouse coordinates.
Example with $event (Getting Input Value):

```
// app.component.ts
export class AppComponent {
  userInput = '';

  onInputChange(event: any) {
    this.userInput = event.target.value;
  }
}
```

```
<!-- app.component.html -->
<input (input)="onInputChange($event)" placeholder="Type something">
<p>You typed: {{ userInput }}</p>
```

Explanation:

- The input event is triggered when the user types into the input field. The onInputChange($event) method captures the event object, and the input's value (event.target.value) is assigned to the userInput property.

Handling Keyboard Events:

Event binding can also capture keyboard events such as key presses.

Example with keyup Event:

```
// app.component.ts
export class AppComponent {
  lastKeyPressed = '';

  onKeyUp(event: KeyboardEvent) {
    this.lastKeyPressed = event.key;
  }
}
<!-- app.component.html -->
<input (keyup)="onKeyUp($event)" placeholder="Press any key">
<p>Last key pressed: {{ lastKeyPressed }}</p>
```

Explanation:

- The keyup event is triggered when the user releases a key. The method onKeyUp($event) captures the KeyboardEvent, and the key pressed (event.key) is displayed in the paragraph.

Handling Mouse Events:

You can also handle mouse events such as click, dblclick, mouseenter, mouseleave, and more.

Example with mouseenter and mouseleave:

```
// app.component.ts
export class AppComponent {
  hoverMessage = '';

  onMouseEnter() {
    this.hoverMessage = 'Mouse entered!';
  }

  onMouseLeave() {
    this.hoverMessage = 'Mouse left!';
  }
}
<!-- app.component.html -->
<div (mouseenter)="onMouseEnter()" (mouseleave)="onMouseLeave()">
  Hover over me!
</div>
<p>{{ hoverMessage }}</p>
```

Explanation:

- When the user hovers over the div, the mouseenter event is triggered, and when the mouse leaves the div, the mouseleave event is triggered. The corresponding methods update the hoverMessage.

Event Binding with DOM Elements:

Event binding can be used with different HTML elements such as buttons, inputs, forms, and more.

Example: Submitting a Form:

You can handle form submissions by binding to the submit event.

```
// app.component.ts
export class AppComponent {
  submittedMessage = '';

  onSubmit() {
    this.submittedMessage = 'Form submitted!';
  }
}
<!-- app.component.html -->
<form (submit)="onSubmit()">
  <input type="text" placeholder="Your name" required>
  <button type="submit">Submit</button>
</form>
<p>{{ submittedMessage }}</p>
```

Explanation:

- The submit event is triggered when the form is submitted. The onSubmit() method is executed, and the message is displayed.

Passing Additional Data with Event Binding:

You can pass additional arguments to the method when handling events.

Example:

```
// app.component.ts
export class AppComponent {
  logMessage(message: string) {
    console.log(message);
  }
}
<!-- app.component.html -->
<button (click)="logMessage('Button was clicked!')">Log Message</button>
```

Explanation:

- The logMessage() method accepts a string argument. When the button is clicked, it logs the message "Button was clicked!" to the console.

Stop Event Propagation:

Sometimes, you may want to stop the propagation of events (e.g., stopping a click event from propagating to parent elements). You can use the $event.stopPropagation() method.

Example:

```
// app.component.ts
export class AppComponent {
  onDivClick() {
    console.log('Div clicked!');
  }

  onButtonClick(event: MouseEvent) {
    event.stopPropagation(); // Stop the click from propagating to the div
    console.log('Button clicked!');
  }
}
<!-- app.component.html -->
<div (click)="onDivClick()">
  <button (click)="onButtonClick($event)">Click Me</button>
</div>
```

Explanation:

- When the button is clicked, the click event is handled by the onButtonClick() method, but the stopPropagation() prevents the event from bubbling up to the parent div.

Using Template Reference Variables for Event Binding:

In addition to $event, you can use **template reference variables** to interact with DOM elements in the template.

Example:

```
<!-- app.component.html -->
<input #userInput type="text" placeholder="Enter name">
<button (click)="onSave(userInput.value)">Save</button>
// app.component.ts
export class AppComponent {
  onSave(inputValue: string) {
    console.log('Saved value:', inputValue);
  }
}
```

Explanation:

- The #userInput is a template reference variable that holds a reference to the input field. When the button is clicked, the input's value is passed to the onSave() method.

Summary:

- **Event Binding** is used to listen for user actions (like clicks, key presses, mouse events, etc.) and respond by executing component methods.
- The ($event) object allows access to event-related data (like the value of an input field, the key pressed, etc.).
- Event binding is crucial for user interaction and handling complex form inputs, validations, and other events.
- You can stop event propagation using event.stopPropagation() and use template reference variables to get direct access to DOM elements.

B. Two Way Data Binding

Two-way data binding in Angular allows for the synchronization of data between the component's property and the view (template). It enables updates in both directions—when the user modifies the view (e.g., by typing into an input field), the component property is automatically updated, and when the component property changes, the view reflects that change.

Angular provides the [(ngModel)] directive for achieving two-way data binding. This directive is part of the **FormsModule**, which must be imported into your Angular application to use it.

Syntax:

html

Copy code

```
<input [(ngModel)]="componentProperty">
```

- The [(ngModel)] binds the input element to the componentProperty. When the input value changes, the componentProperty is updated, and vice versa.

Steps to Use ngModel:

1. Import the FormsModule in your app module.
    1. Goto App.module.ts file and add directive **“import {FormsModule} from '@angular/forms';”**
    2. Add module name into section as highlighted

       imports: [
       BrowserModule,
       AppRoutingModule,
       FormsModule],
2. Use the [(ngModel)] directive in your template to bind the view and component.

Basic Example:

Step 1: Import FormsModule in the App Module:

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';  // <-- Import FormsModule
import { AppComponent } from './app.component';

@NgModule({
declarations: [AppComponent],
imports: [BrowserModule, FormsModule],  // <-- Add FormsModule to imports
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 2: Component (Define a property):

```
// app.component.ts
import { Component } from '@angular/core';
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  username = '';  // Property to bind }
```

Step 3: Template (Two-Way Binding with ngModel):

```
<!-- app.component.html -->
<input [(ngModel)]="username" placeholder="Enter your name">
<p>Your name is: {{ username }}</p>
```

Explanation:

- The [(ngModel)] directive binds the input field to the username property. When the user types in the input field, the username property is updated, and the new value is displayed in the paragraph using interpolation ({{ username }}).

Example with Input Fields:

You can use two-way binding with multiple input fields to synchronize values between the component and the template.

```
// app.component.ts
export class AppComponent {
  firstName = '';
  lastName = '';
}
<!-- app.component.html -->
<div>
  <label>First Name:</label>
  <input [(ngModel)]="firstName" placeholder="First Name">
  <label>Last Name:</label>
  <input [(ngModel)]="lastName" placeholder="Last Name">
  <p>Full Name: {{ firstName }} {{ lastName }}</p>
</div>
```

Explanation:

- The input fields are bound to firstName and lastName. When the user enters text in either field, both the component properties and the display (Full Name) are updated in real-time.

Binding to Select Dropdowns:

You can use [(ngModel)] with a <select> element for dropdown menus as well.

```
// app.component.ts
export class AppComponent {
  selectedCountry = '';
  countries = ['USA', 'Canada', 'UK', 'Australia'];
}
<!-- app.component.html -->
<label for="country">Select your country:</label>
<select [(ngModel)]="selectedCountry" id="country">
  <option *ngFor="let country of countries" [value]="country">{{ country }}</option>
</select>
<p>You selected: {{ selectedCountry }}</p>
```

Explanation:

- The <select> element is bound to selectedCountry. When a user selects a country from the dropdown, the component's selectedCountry property is updated, and the selected value is displayed.

Two-Way Binding with Custom Components:

Two-way data binding can also be used with custom components using @Input() and @Output() decorators.

Step 1: Child Component (Custom Component):

```
// child.component.ts
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <input [value]="value" (input)="onInputChange($event)">
  `
})
export class ChildComponent {
  @Input() value: string = '';
  @Output() valueChange = new EventEmitter<string>();

  onInputChange(event: any) {
    this.valueChange.emit(event.target.value);
  }
}
```

Explanation:

- The ChildComponent accepts an input value through the @Input() decorator and emits changes to its parent through the @Output() decorator using EventEmitter.

Step 2: Parent Component:

```
// parent.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-parent',
  template: `
    <app-child [(value)]="parentValue"></app-child>
    <p>Parent Value: {{ parentValue }}</p>
  `
})
export class ParentComponent {
  parentValue = 'Initial Value';
}
```

Explanation:

- The parent component binds its parentValue to the value of the child component using two-way data binding. Any changes made in the child component's input field are reflected in the parent component, and vice versa.

Common Use Cases:

1. **Form Inputs:** Two-way binding is typically used in forms where user input needs to be reflected in the component, and vice versa.
2. **Dynamic Forms:** When creating dynamic forms where the model changes based on user input.
3. **Real-Time Feedback:** Providing real-time feedback to users, such as showing a preview of their input.

Avoid Common Mistakes:

- **Importing FormsModule:** Always ensure that FormsModule is imported into your module when using ngModel.
- **One-Way vs. Two-Way Binding:** Use [(ngModel)] when you need synchronization between the component and the view. For simple one-way binding, use [value] and (input) separately.

Summary:

- **Two-way data binding** allows for real-time synchronization between the component and the view.
- It is implemented using [(ngModel)], which both listens to user events and updates the model, and reflects model changes in the view.
- It is especially useful in forms, dropdowns, and custom components where data flow in both directions is required.

ng-template :

In Angular, the **ng-template directive** defines a template that is not rendered directly but can be used later in the view, or conditionally rendered. It allows for more dynamic and flexible layouts, often used in conjunction with structural directives like *ngIf, *ngFor, or ngTemplateOutlet.

Basic Example:

In this example, the content inside the ng-template will not be displayed until it is explicitly referenced in the template.

html

Copy code

```
<ng-template #myTemplate>
  <p>This is rendered using ng-template!</p>
</ng-template>

<button (click)="showTemplate = !showTemplate">Toggle Template</button>

<div *ngIf="showTemplate">
  <ng-container *ngTemplateOutlet="myTemplate"></ng-container>
</div>
```

Component Code:

ts

Copy code

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  showTemplate = false;
}
```

Explanation:

- The ng-template with the reference #myTemplate defines a block of HTML that can be used elsewhere.
- The button toggles the visibility of the template content.
- The *ngTemplateOutlet directive is used to render the content of the ng-template conditionally, depending on the value of showTemplate.

Example with *ngIf:

ng-template can be used with *ngIf to define what should be rendered when the condition is false.

```
<div *ngIf="isLoggedIn; else loggedOutTemplate">
  <p>Welcome, you are logged in!</p>
</div>
<ng-template #loggedOutTemplate>
  <p>Please log in to continue.</p>
</ng-template>
```

Component Code:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  isLoggedIn = false;
}
```

Explanation:

- If isLoggedIn is true, the content inside the will be displayed.
- If isLoggedIn is false, the content inside the ng-template (referenced as #loggedOutTemplate) will be displayed instead.

Example with *ngFor:

You can also use ng-template with *ngFor to customize how a list is rendered.

html

Copy code

```
<ul>
  <ng-template ngFor let-item let-i="index" [ngForOf]="items">
    <li>{{ i + 1 }}. {{ item }}</li>
  </ng-template>
</ul>
```

Component Code:

ts

Copy code

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  items = ['Apple', 'Banana', 'Orange'];
}
```

Explanation:

- ngFor is used inside the ng-template to render a list of items with their index.
- The let-item variable is used to access the current item, and let-i="index" is used to access the index of the current item in the list.

Conclusion:

ng-template in Angular provides a way to define reusable templates that can be conditionally displayed or dynamically inserted into the DOM. It's useful when you need to create flexible views, manage conditional content, or dynamically insert sections of HTML.

Two Way Data Binding Implemented Example with output:

```
<!-- app.component.html -->
<div id="input">
  <input type="text" [(ngModel)]="mn"  placeholder="Enter Movie Name">
  <input type="text" [(ngModel)]="mu" placeholder="Enter Image URL">
  <input type="button" (click)="ShowMovie()" value="Click me">
</div>
<div id="output" [class]="isShow?'show':'hide'">
  <img src="{{txtMovieURL}}" alt="{{txtMovieName}}">
  <h2>{{txtMovieName}}</h2>
</div>

//app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  txtMovieName = '';
  txtMovieURL = '';
  mn = '';
  mu = '';
  isShow = false;
  ShowMovie()
  {
    this.isShow=true;
    this.txtMovieName = this.mn;
    this.txtMovieURL = this.mu;
  }
}

/* app.component.html */
.hide{
    display: none;
}
.show{
    display: block;
}
```

Sample project from above concepts :

```
//app.module.ts
import {FormsModule} from '@angular/forms';
  imports: [
    BrowserModule,
    AppRoutingModule,
```

```
    FormsModule
  ],

// app.component.ts
// ngIf Example
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  isVisible = false;
  myArray = ["Apple","Banana","Oranges","Chikku"];
  WeekDays:string[] =
['Sunday','Monday','Tuesday','Wednesday','Thursday','Friday','Saturday'];
  WDno:number=0;
  isSelected=false;
  ddSelectedItem='';
  ddSI='';
  Show1(){
        this.isVisible = false;
  }
  Show2(){
        this.isVisible = true;
  }
  SelectFun(){
    this.isSelected=true;
    //this.ddSelectedItem = this.ddSI;

    // Get the select element
  const selectElement = document.getElementById('fruits') as HTMLSelectElement;

  // Get the selected value
  const selectedValue = selectElement.value;

  // Check if a valid option is selected (i.e., not the default)
  if (selectedValue) {
    document.getElementById('SelectedContent')!.textContent = `You selected:
${selectedValue}`;
  } else {
    document.getElementById('SelectedContent')!.textContent = 'No fruit selected,
Please select';
  }
  }
  getDay()
```

```
  {
    this.WDno =this.WDno;
  }

  fruits: string[] = ['Apple', 'Banana', 'Mango', 'Orange','Papaya'];
  selectedFruit?: string;
  selectFruit(index: number) {
    this.selectedFruit = this.fruits[index];
  }
}
<!-- app.component.html -->
<!-- ngIf Example -->
<!-- This directive adds or removes an element based on the condition. -->
<h2>Example of ngFor</h2>
<div class="Model" *ngIf="isVisible">
  <div><h2>Hello This is ngIf</h2><button (click)="Show1()">❌</button></div>
  <h2>Lorem ipsum dolor sit amet consectetur adipisicing elit. Deleniti provident,
itaque accusantium illo expedita veniam quia adipisci distinctio dolorum nisi! Eum
facilis quia blanditiis quaerat soluta harum quis! Reiciendis, odit.</h2>
</div>
<div>
  <button (click)="Show2()"> </button>
</div>
<hr>
<!-- ngFor : This directive repeats a block of HTML for each item in a list. -->
<div>
  <h2>Example of ngFor</h2>
  <ul>
    <li *ngFor="let item of myArray">{{item}}</li>
  </ul>
</div>
<hr>
<div>
  <label for="WeekDays">Enter Week Number : </label>
  <input type="text" id="WeekDays" name="WeekDays" (change)="getDay()"
[(ngModel)]="WDno" placeholder="Enter digit between 0 to 6">
  <ul>
    <li>
    <h2>{{WDno}} : {{WeekDays[WDno]}}</h2>
    </li>
</ul>
</div>
<hr>
<!-- ngSwitch : This directive conditionally displays elements based on an
expression. -->
 <div>
  <h2>Example of ngSwitch</h2>
```

```
<label for="fruits">Choose a fruit : </label>
<select id="fruits" name="fruits" [(ngModel)]="ddSI">
  <option value="">--Select a fruit--</option>
  <option value="apple">Apple</option>
  <option value="banana">Banana</option>
  <option value="cherry">Cherry</option>
  <option value="grape">Grape</option>
</select> 
<button (click)="SelectFun()">Submit</button>
</div>
<div id="SelectedContent" *ngIf="isSelected">
  <h2>{{ddSelectedItem}}</h2>
</div><hr>
<div>
  <h3>Dynamic buttons Created depend on size of array<br>Select a Fruit:</h3>
  <button *ngFor="let fruit of fruits; let i = index" (click)="selectFruit(i)">
    {{ fruit }}
  </button>
  <p *ngIf="selectedFruit !== undefined">
    You selected: {{ selectedFruit }}
  </p>
</div>
/* app.component.css */
/* ngIf Example */
.Model{
    height: max-content;
    width: 500px;
    padding : 13px;
    z-index: 1000;
    position: absolute;
    top: 30%;
    left: 30%;
    background: lightblue;
    animation: moveDown 2s ease;
}
.Model div{
    display: flex;
    justify-content: space-between;
}
/* Define the keyframes for the animation */
@keyframes moveDown {
    0% {
      top: 0;
    }
    100% {
      top: 197px;
    }    }
```

Property Directives:

Property Directives allow you to modify the behavior or appearance of elements dynamically by binding to their properties. Angular already provides built-in property binding, but you can also create custom **attribute directives** to modify an element's appearance or behavior.

Let's break it down:

1. Built-in Property Directives (like ngClass, ngStyle)

You can use Angular's built-in directives to modify element properties such as classes and styles.

Example: Using ngClass Directive

ngClass is a built-in directive that allows you to dynamically add or remove CSS classes to/from an element.

Component (TypeScript):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  isActive: boolean = true;
}
```

Template (HTML):

```
<div [ngClass]="{'active': isActive, 'inactive': !isActive}">
  Toggle class with ngClass!
</div>
<button (click)="isActive = !isActive">Toggle Active</button>
```

Explanation:

- The ngClass directive binds the active class when isActive is true and binds the inactive class when isActive is false.
- Clicking the button toggles the value of isActive, which updates the class dynamically.

2. Custom Property Directive

You can create a custom directive that will modify an element property dynamically. For example, let's create a custom directive that changes the background color of an element when a specific condition is met.

Steps to Create a Custom Directive:

1. **Generate a Directive** using Angular CLI:

```
ng generate directive highlight
```

2. **Update the Custom Directive (TypeScript):**

```
import { Directive, ElementRef, Input, OnChanges, Renderer2 } from
'@angular/core';
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective implements OnChanges {
  @Input() appHighlight: string = ''; // Property binding for the background color
```

```
  constructor(private el: ElementRef, private renderer: Renderer2) {}

  ngOnChanges() {
    // Apply the background color to the element
    this.renderer.setStyle(this.el.nativeElement, 'backgroundColor',
this.appHighlight);
  }
}
```

3. **Use the Directive in the Template (HTML):**

```
<div [appHighlight]="'yellow'">
  This div has a yellow background!
</div>

<div [appHighlight]="'lightblue'">
  This div has a light blue background!
</div>
```

Explanation:

- The directive uses property binding with [appHighlight] to receive a color string (e.g., 'yellow' or 'lightblue').
- Inside the directive, the ngOnChanges() method detects changes to the bound property and applies the new background color to the element using Angular's Renderer2.

Summary:

- **Built-in Property Directives:** Use directives like ngClass, ngStyle to bind and modify the appearance of elements.
- **Custom Property Directives:** You can create your own directives that dynamically modify the behavior or style of elements based on input properties.

In the custom example, [appHighlight] is a property directive that dynamically changes the background color of an element.

****

What Is a Constructor?

An Angular constructor is a function that is used to initialize an Angular application. The constructor is run when the application is first created, and it is responsible for setting up the Angular environment. The constructor can be used to inject dependencies, set the default values for properties, and run any other initialization code that is needed.

For example:

```
class CoffeeSwallower {
  name: string;

  constructor(name = "SpewCoffeeGalore") {
    this.name = name;
  }
}
```

JavaScript

Here, as you can see, it's automatically called when we created a new instance of our class

Types of the Constructor

Default Constructor

A default constructor is a constructor that is automatically generated by the compiler if one is not explicitly provided by the programmer. A default constructor typically initializes member variables to their default values (e.g., 0 for ints, null for objects) and does not perform any other actions.

A default constructor is necessary for a class if any of the following are true:

- The class has any non-static member variables that require initialization
- The class has any base classes (i.e., is derived from another class)
- The class has any virtual member functions

If a class does not have a default constructor and one of the above conditions is true, the compiler will generate an error.

Parameterized Constructor

Parameterized constructors accept one or more parameters. Parameters are simply variables that are passed into the constructor when it is invoked. The parameterized constructor can be used to set the initial values of the instance variables of an object. For example, a parameterized constructor for a class called Employee might accept an employee ID and a name. The constructor would then use these parameters to set the corresponding instance variables.

Copy Constructors

There are two types of copy constructors: shallow and deep.

The **shallow** copy constructor simply copies the values of the data members of the object being initialized to the data members of the new object.

The **deep** copy constructor also copies any pointers that the object being initialized has, and also copies the objects that those pointers point to.

Conversion Constructors

There are two types of conversion constructors: implicit and explicit.

An **implicit** conversion constructor is invoked without the user specifying that they want to convert an object.

In an **explicit** conversion, the constructor is invoked only when the user specifically requests a conversion.

In general, it is best to avoid conversion constructors because they can lead to unexpected results. For example, if an implicit conversion constructor is invoked when the user is expecting an explicit conversion, the results may not be what the user expects.

Move Constructors

There are two types of move constructors: **lvalue** move constructors and **rvalue** move constructors. Lvalue move constructors take an lvalue reference as their parameter, while rvalue move constructors take an rvalue reference.

Lvalue move constructors typically copy the contents of the object they are called on, while rvalue moves constructors typically move the contents of the object they are called on.

Uses of the Angular Constructor

An Angular constructor is a powerful tool that can be used to create, modify and manipulate AngularJS applications. Here are some of the use full uses of the Angular constructor:

- Bootstrap the application by instantiating the root scope and compiling the application's templates.
- Angular constructor is used to initialize the Angular environment and to create the root Angular object.
- Create custom directives to manage dependencies for testing and debugging Angular applications.
- Also create custom services, filters, injectors and more. Each of these uses has its specific purpose and can be very helpful in certain situations.
- An Angular constructor can be used to inject dependencies into Angular controllers and to access the Angular global scope.

A Practical Guide To Angular: Services and Dependency Injection

Learn how to use services and dependency injection to improve your Angular development by making it modular, extensible and loosely coupled.

How to Initialize Angular Constructors

There are several ways to initialize constructors in Angular.

Method 1: Using ng-init Directive

One method is to use the ng-init directive. This directive can be used to initialize values in scope.

For example, the following code will initialize the name and city variables in scope:

```
<div ng-init="name='John'; city='New York'">
  {{name}} lives in {{city}}.
</div>
```

JavaScript

Method 2: Using Constructor Function

Another method to initialize constructors is to use the constructor function. This function is automatically called when an Angular component is created. The constructor can be used to initialize values in the component's scope.

For example, the following code will initialize the name and city variables in the component's scope:

```
constructor(scope) {
   scope.name = 'John';
   scope.city = 'New York';
}
```

JavaScript

Method 3: Initialization in ng-controller Directive

Finally, constructors can be initialized in the ng-controller directive. This directive can be used to create a new scope for the controller. The new scope will contain any values initialized in the constructor.

For example, the following code will initialize the name and city variables in the new scope:

```
<div ng-controller="MyController">
```

```
  {{name}} lives in {{city}}.
</div>

MyController.$inject = ['$scope'];

function MyController($scope) {
   $scope.name = 'John';
   $scope.city = 'New York';
}
```

JavaScript

What's the Difference Between NgOnInit and Constructors?

A lot of times developers get confused between ngoninit and constructors. There are two ways to start up a component in Angular: using the constructor or ngOnInit. Here are some of the key differences between the two:

- The constructor is called before the component is initialized, while ngoninit is a lifecycle hook that is called after the component is initialized.
- The constructor is used for setting global configuration variables that need to be set before the app starts running. NgOnInit is used for anything that needs to be done after the component has been initialized, such as fetching data from an API.
- The constructor is only used for initializing the component, while ngoninit allows you to access the component's properties and make any necessary changes.

Difference between constructor and ngOnInit in Angular

Constructor	**ngOnInit**
Executes before ngOnInit.	Executes after the constructor.
Used for dependency injection and initializing instance variables.	Used for initialization tasks that require the component to be fully initialized.
Cannot access the component's DOM elements.	Can access the component's DOM elements.
Executed every time a component is created.	Executed only once after the component has been initialized.
Can be used in both classes and directives.	Only available in classes that implement the OnInit interface.
Can be used to configure the component's metadata, such as its selector and inputs.	Cannot be used to configure the component's metadata.
Cannot use @ViewChild or @ContentChild decorators to query child components or content projection.	Cannot use @ViewChild or @ContentChild queries; use ngAfterViewInit for @ViewChild and ngAfterContentInit for @ContentChild.

****

Component Directives:

a **Component** is a special type of directive with its own template, styles, and logic. It is the fundamental building block of an Angular application and is used to create views, handle user interactions, and define application logic. Each Angular application is a tree of components that work together to create the user interface.

Key Features of Component Directives:

- **Template**: Defines the HTML structure of the view.
- **Styles**: CSS or styles that apply to the component.
- **Selector**: Specifies how the component is referenced in HTML.
- **Logic**: The component class handles data, logic, and interactions.

Defining a Component

A component is defined using the @Component decorator, which includes metadata like the selector, template, and styles.

Example of a Simple Angular Component

1. Component (TypeScript)

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

@Component({
  selector: 'app-hello-world',  // Selector to use this component in HTML
  template: `
    <div>
      <h1>{{ title }}</h1>
      <p>Welcome to Angular Component!</p>
    </div>
  `,  // Inline template defining the view
  styles: [
    `div { border: 1px solid black; padding: 10px; width: 300px; }`,
    `h1 { color: green; }`
  ]  // Inline styles
})
export class HelloWorldComponent {
  title: string = 'Hello World';  // Component logic
}
```

- **@Component Decorator**: Declares that this class is a component and provides the metadata.
  - selector: Specifies how this component is referenced in an HTML file. You can use <app-hello-world></app-hello-world> to include this component.
  - template: Defines the view (HTML) of the component.
  - styles: Defines the styles that apply specifically to this component.
- **Class Logic**: The HelloWorldComponent class contains the business logic, such as properties (title) and methods.

2. Using the Component in Another Template (HTML)

You can use the component's selector to include it in other templates, such as the root component's HTML:

```
<!-- In app.component.html -->
<app-hello-world></app-hello-world>
```

This will render the HelloWorldComponent wherever the <app-hello-world> tag is used.

3. Adding the Component to a Module

To use the component in an Angular application, you need to declare it in an Angular module (e.g., app.module.ts):

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { HelloWorldComponent } from './hello-world.component';  // Import the component

@NgModule({
  declarations: [
    AppComponent,
    HelloWorldComponent  // Declare the component
  ],
  imports: [ BrowserModule ],
  providers: [],
  bootstrap: [AppComponent]  // Root component to bootstrap
})
export class AppModule { }
```

How This Component Works:

- The HelloWorldComponent defines a simple view with an h1 element and a p element.
- The title property is bound to the template using Angular's interpolation {{ title }}.
- The HelloWorldComponent is then used in the application by adding <app-hello-world></app-hello-world> in any template.

Key Points:

1. **Selector**: Specifies the HTML tag that represents the component in the DOM (app-hello-world).
2. **Template**: Defines the structure of the component's view (either inline or in an external file).
3. **Styles**: Component-specific styles can be defined inline or in an external file.
4. **Logic**: The class contains the data and logic for the component (title property in this case).

Components allow Angular to create a modular, reusable, and maintainable code structure, where each UI block can be encapsulated into its own component with its own view, styles, and behavior.

Attribute Directives:

Attribute directives in Angular are used to modify the behavior or appearance of elements without changing the structure of the DOM. They can change the styling, properties, or attributes of elements in the template dynamically based on the component's data or logic.

There are two primary ways to use attribute directives:

1. **Using built-in attribute directives** like ngClass, ngStyle, ngModel.
2. **Creating custom attribute directives** to apply specific functionality or behavior.

1. Built-in Attribute Directives

Example: ngClass and ngStyle

In this example, we'll use ngClass to apply CSS classes dynamically and ngStyle to apply inline styles conditionally.

Component (TypeScript):

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  isActive: boolean = true;
  fontSize: number = 18;

  toggleActive() {
    this.isActive = !this.isActive;
  }

  increaseFontSize() {
    this.fontSize += 2;
  }
}
```

Template (HTML):

```
<div [ngClass]="{ 'active': isActive, 'inactive': !isActive }">
  This div has a dynamic class applied.
</div>

<div [ngStyle]="{ 'font-size': fontSize + 'px', 'color': isActive ? 'green' : 'red' }">
  This text has a dynamic font size and color.
</div>

<button (click)="toggleActive()">Toggle Active</button>
<button (click)="increaseFontSize()">Increase Font Size</button>
```

CSS:

```
.active {
  background-color: lightgreen;
```

```
}
.inactive {
  background-color: lightcoral;
}
```

Explanation:

- **ngClass** dynamically adds the class active or inactive depending on the isActive boolean property.
- **ngStyle** applies styles like font-size and color dynamically based on the component properties (fontSize and isActive).

2. Custom Attribute Directive

You can also create your own custom attribute directive to encapsulate reusable behavior and apply it to any DOM element.

Example: Custom Directive to Highlight an Element on Hover

Let's create a custom directive that changes the background color of an element when the user hovers over it.

Step 1: Create the Directive

Use Angular CLI to generate a directive: bash

```
ng generate directive highlight
```

Step 2: Implement the Directive

Directive (TypeScript):

```
import { Directive, ElementRef, Renderer2, HostListener, Input } from
'@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @Input('appHighlight') highlightColor: string = 'yellow';  // Default color

  constructor(private el: ElementRef, private renderer: Renderer2) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.setBackgroundColor(this.highlightColor);  // Change background on hover
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.setBackgroundColor(null);  // Reset background on mouse leave
  }

  private setBackgroundColor(color: string | null) {
    this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', color);
  }
}
```

Step 3: Use the Directive in a Template

Template (HTML):

```
<p [appHighlight]="'lightblue'">Hover over this text to see a blue highlight.</p>
<p [appHighlight]="'lightgreen'">Hover over this text to see a green highlight.</p>
```

Explanation:

- **appHighlight**: The directive listens for mouseenter and mouseleave events to change the background color dynamically.
- **@Input()**: Allows the highlight color to be passed as an input to the directive.
- **Renderer2**: Ensures safe manipulation of the DOM (setting styles in this case).

Step 4: Declare the Directive in the Module

You need to add the directive to the declarations array in your module (typically app.module.ts):

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { HighlightDirective } from './highlight.directive';  // Import the custom directive

@NgModule({
  declarations: [
    AppComponent,
    HighlightDirective  // Declare the directive here
  ],
  imports: [ BrowserModule ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Summary:

- **Built-in attribute directives** like ngClass, ngStyle, and ngModel are commonly used to dynamically apply classes, styles, or bind data to form elements.
- **Custom attribute directives** allow you to create reusable behavior that can be applied to any element in your application, encapsulating logic that you may want to reuse across different components.

Attribute directives enhance the interactivity of your Angular application by manipulating the behavior and appearance of DOM elements based on component data.

To read more : https://www.geeksforgeeks.org/attribute-directives-in-angular/

Difference between Component, Attribute and Structural Directives?

Component	**Attribute Directive**	**Structural Directives**
Component directive is used to specify the template/html for the Dom Layout	Attribute directive is used to change/modify the behaviour of the html element in the Dom Layout	Structural directive used to add or remove the html Element in the Dom Layout,
Built in @component	**Built in** NgStyle, NgClass	**Built in** *NgIf,*NgFor,*NgSwitch

What are directives in Angular?

Directives are special markers on a DOM element (such as an attribute, element name, class, or comment) that tell Angular to do something with that DOM element. There are three kinds of directives in Angular:

1. **Component Directives**: These are directives with templates. Every Angular component is essentially a directive with a template.
2. **Structural Directives**: These directives change the DOM layout by adding, removing, or manipulating elements.
3. **Attribute Directives**: These directives change the appearance or behavior of an element, component, or another directive.

1. Component Directives

A component is the most commonly used directive in Angular. It combines the HTML markup and logic in one unit.

Example:

TS

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

@Component({
  selector: 'app-hello',
  template: `<h1>Hello, {{name}}!</h1>`,
  styles: ['h1 { color: green; }']
})
export class HelloComponent {
  name: string = 'Angular';
}
```

2. Structural Directives

Structural directives alter the structure of the DOM by adding or removing elements. Common structural directives in Angular are *ngIf, *ngFor, and *ngSwitch.

Example: *ngIf

This directive adds or removes an element based on the condition.

html

```
<div *ngIf="isVisible">This text is visible</div>
```

In the component:

TS

```
export class AppComponent {
  isVisible = true;
}
```

Example: *ngFor

This directive repeats a block of HTML for each item in a list.

html

```
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>
```

In the component:

ts

```
export class AppComponent {
```

```
  items = ['Apple', 'Banana', 'Orange'];
}
```

Example: *ngSwitch

This directive conditionally displays elements based on an expression.

html

```
<div [ngSwitch]="color">
  <p *ngSwitchCase="'red'">Red color selected</p>
  <p *ngSwitchCase="'blue'">Blue color selected</p>
  <p *ngSwitchDefault>Pick a color</p>
</div>
```

In the component:

TS

```
export class AppComponent {
  color = 'red';
}
```

3. Attribute Directives

Attribute directives change the appearance or behavior of an element. Some examples are ngClass, ngStyle, and custom attribute directives.

Example: ngClass

This directive dynamically adds or removes classes on an element.

html

```
<div [ngClass]="{'red-text': isRed, 'bold-text': isBold}">
  This text changes style based on conditions.
</div>
```

In the component:

TS

```
export class AppComponent {
  isRed = true;
  isBold = true;
}
```

Example: ngStyle

This directive dynamically sets inline styles.

html

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

In the component:

TS

```
export class AppComponent {
  textColor = 'blue';
  fontSize = 18;
}
```

Example: Custom Attribute Directive

You can create your own attribute directives. For example, a directive to change the background color of an element when the mouse hovers over it.

1. **Create the directive:**

```
import { Directive, ElementRef, HostListener, Renderer2 } from '@angular/core';
```

```
@Directive({
  selector: '[appHoverHighlight]'
})
export class HoverHighlightDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', 'yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', 'white');
  }
}
```

2. Use it in the template:

```
<p appHoverHighlight>Hover over this text to see the effect.</p>
```

What is Attribute Directive?

Attribute directives are a powerful tool that allows you to manipulate the behavior and appearance of HTML elements.

Directives are the fundamental concepts in angular that help us to add or modify the behavior or appearance of HTML elements. They help us to modify DOM behavior, user functionality, and customizing components.

Benefits of Attribute Directive:

- **Dynamic Styling:** Attribute directives can be used to dynamically apply styles to HTML elements based on certain conditions.
- **DOM Manipulation:** They enable you to interact with and manipulate the DOM based on user actions or application state changes.
- **Reusability:** Attribute directives promote code reuse by encapsulating behavior that can be applied to multiple elements across the application.
- **Enhanced Functionality:** They allow you to extend HTML with custom functionality, improving the overall user experience.

Types of Attribute Directives:

1. Built-in Attribute directives:

These attributes are used within html tags to modify the appearance of elements, components or directives.

We have 3 main built in attribute directives: ngClass, ngStyle and ngModel

1. ngClass

This attribute is used to conditionally give the classes to the elements based on the value binded to ngClass directive.

Syntax:

```
<element [ngClass]="expression"></element>
```

2. ngStyle

This attribute is used to dynamically apply the styles to elements based on the value binded to ngStyle directive. It helps us to modify the appearance of elements on conditional basis. We can also use ngStyles to override in

Syntax:

<element [ngStyle]="expression"></element>

3. ngModel

This attribute is generally used to bind form control data to a class variable . This allows a two way binding and this is most commonly used directive with forms. It also comes with few basic validations like required,minLength , maxLength which can be directly used in our input tags.

To use this directive we need to import Forms Module in our module file.

Syntax:

<input name="name" [(ngModel)]="name"/>

2. Custom Attribute directives

We can also create our own directives based on our own requirements. This helps us creating reusable components and validating data etc. We can also create our own directives based on our own requirements. This helps us creating reusable components and validating data etc. Custom directives can be created using the `@Directive` decorator and can implement various methods to interact with the host element and perform actions.

Steps to create Custom Directives:

Step 1: Create a Directive

ng generate directive <directive-name>

The above command helps us to create new directive.

Step 2: Implement necessary imports

Open the generated directive file and import necessary modules like ElementRef , HostListener etc.

ElementRef : Provides access to the respective DOM element to change the styles or properties .

HostListener : Decorator used to listen for events on the host element such as mouse controls , clicks etc.

Input (optional) : Allows you to pass data from the component template to the directive.

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

Step 3: Define the Selector

In the @Directive decorator , we need to provide the **`selector`** property to specify how the directive will be used in the template. If we use the ng generate directive command, it gives selector property by default, we can also change this selector name for our usage.

```
@Directive({
  selector: '[appHighlight]'
}
```

Step 4: Implement the logic

Here, we can write our custom functionality in the directive file. We can also implement life cycle hook methods in the directive file if required. Here we can also pass inputs to the directive using **@Input** decorator.

Step 5: Using our directive in template

In our component's template, we can use the selector given in the directive as an attribute to the required element on which we want to perform our logic.

<element appNewDirective>.....content </element>

Here appNewDirective is the selector of our directive. In this way we can use the directive in our component templates with the custom functionality.
As mentioned above, we can also pass inputs to directive.
<element appNewDirective [input1]="value">.....content </element>
In this way we can pass inputs to our custom directive, here input1 is the input we declared as @Input() in our decorative, and `value` is the data we are passing to that particular input. Based on our requirement we can pass multiple inputs .

Example of Attribute Directives:

Now let us take an example to understand both built-in attribute directives and also custom directives.

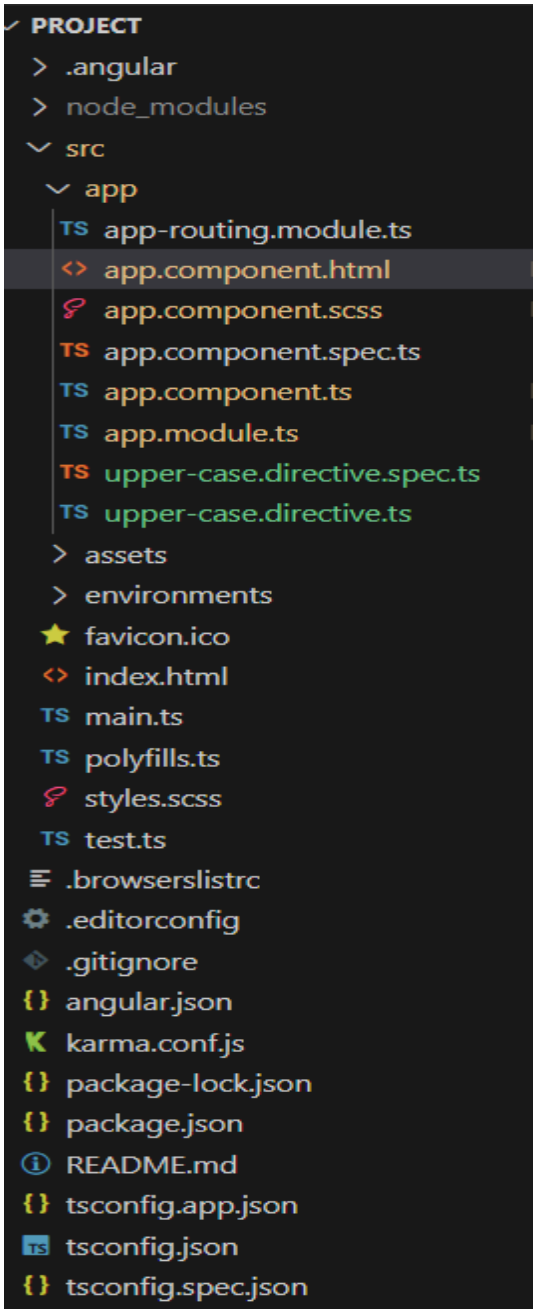
Step 1: Create a new angular project

ng new project

Step 2. To generate a new directive, we can use the following command.

ng generate directive upperCase

Folder Structure :



Dependencies:

```
  "dependencies": {
    "@angular/animations": "~13.0.0-next.0",
    "@angular/common": "~13.0.0-next.0",
    "@angular/compiler": "~13.0.0-next.0",
    "@angular/core": "~13.0.0-next.0",
    "@angular/forms": "~13.0.0-next.0",
    "@angular/platform-browser": "~13.0.0-next.0",
    "@angular/platform-browser-dynamic": "~13.0.0-next.0",
    "@angular/router": "~13.0.0-next.0",
    "rxjs": "~7.4.0",
    "tslib": "^2.3.0",
    "zone.js": "~0.11.4"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "~13.0.0",
    "@angular/cli": "~13.0.0",
    "@angular/compiler-cli": "~13.0.0-next.0",
    "@types/jasmine": "~3.10.0",
    "@types/node": "^12.11.1",
    "jasmine-core": "~3.10.0",
    "karma": "~6.3.0",
    "karma-chrome-launcher": "~3.1.0",
    "karma-coverage": "~2.0.3",
    "karma-jasmine": "~4.0.0",
    "karma-jasmine-html-reporter": "~1.7.0",
    "typescript": "~4.4.3"
  }
```

3. Implement the logic

HTMLJavaScriptJavaScript

```
<!-- app.component.html -->

<form>
  <div>
    <label>Name: </label>
    <input
      appUpperCase
      type="text"
      name="name"
      minlength="4"
      [(ngModel)]="name"
      #nameInput="ngModel"
      required
      [ngClass]="{ 'is-invalid': nameInput.touched && nameInput.invalid }"
    />
    <div *ngIf="nameInput.touched && nameInput.invalid">
      Minimum length of name is 4 characters
```

```
    </div>
  </div>
  <div>
    <label>Age: </label>
    <input
      type="number"
      name="age"
      [(ngModel)]="age"
      #ageInput="ngModel"
      required
      [ngStyle]="{
        'border-color': ageInput.invalid && ageInput.touched ? 'red' : ''
      }"
    />
    <div *ngIf="ageInput.touched && ageInput.invalid">Age is required</div>
  </div>
</form>
```

Output:

Summary:

- **Component Directives**: Combine HTML and logic (@Component).
- **Structural Directives**: Modify the DOM layout (*ngIf, *ngFor, *ngSwitch).
- **Attribute Directives**: Change the appearance or behavior of elements (ngClass, ngStyle, custom directives).

You can also create your own structural or attribute directives to control behavior specific to your application.

Example from video:

HTML

```
<h1>Products</h1>
<p>The square of {{b}} is {{b|square}}</p>
<p>The power of {{b}} with 3 is {{b|power:3}}</p>
{{obj | json}} <!-- name: "raj", age:50 -->
 <p class="disc">Congratulations, use code "PRASAD" to get
```

```
  {{a | percent:'1.0-2'}} discount excluding tax {{tax|number:'2.2-2'|percent:'2.2-
2'}}</p>

<div class="products"
[ngStyle]="{'fontSize':isHeading?'48px':'24px','background-color':bgColor}">
  <div class="product" *ngFor="let p of products"
  [ngClass]="{'dark':isDark}">
      <img src={{p.image}} alt={{p.title}} height="200"/>
      <p>{{dt | date:"dd/MM/YY HH:mm:SS"}}</p>
      <h2>{{p.title | lowercase}}</h2>
      <p>Price: {{p.price | currency:"INR"}}</p>
      <button>Add to cart</button>
  </div>
</div>
```

TS:

```
b=5
  obj = {
    name: "raj",
    age:50
  }
  a = 0.5;
  tax =0.2;
bgColor = "white";
  isHeading = false
  isDark = false;
  dt: any;
```

Pipes:

I. Angular.dev

Pipes are a special operator in Angular template expressions that allows you to transform data declaratively in your template. Pipes let you declare a transformation function once and then use that transformation across multiple templates. Angular pipes use the vertical bar character (|), inspired by the Unix pipe.

Note: Angular's pipe syntax deviates from standard JavaScript, which uses the vertical bar character for the bitwise OR operator. Angular template expressions do not support bitwise operators.

Here is an example using some built-in pipes that Angular provides:

```
import { Component } from '@angular/core';
import { CurrencyPipe, DatePipe, TitleCasePipe } from '@angular/common';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CurrencyPipe, DatePipe, TitleCasePipe],
  template: `
    <main>
      <!-- Transform the company name to title-case and
      transform the purchasedOn date to a locale-formatted string -->
<h1>Purchases from {{ company | titlecase }} on {{ purchasedOn | date }}</h1>
       <!-- Transform the amount to a currency-formatted string -->
     <p>Total: {{ amount | currency }}</p>
    </main>
  `,
})
export class ShoppingCartComponent {
  amount = 123.45;
  company = 'acme corporation';
  purchasedOn = '2024-07-08';
}
```

When Angular renders the component, it will ensure that the appropriate date format and currency is based on the locale of the user. If the user is in the United States, it would render:

```
<main>
  <h1>Purchases from Acme Corporation on Jul 8, 2024</h1>
  <p>Total: $123.45</p>
</main>
```

Built-in Pipes

Angular includes a set of built-in pipes in the @angular/common package:

Name	Description
AsyncPipe	Read the value from a Promise or an RxJS Observable.
CurrencyPipe	Transforms a number to a currency string, formatted according to locale rules. currency: currency is used to transform data to specific currency by taking arguments. default is dollar 250 | currency ===> $250 250 | currency:"INR" ===> ₹250
DatePipe	Formats a Date value according to locale rules. Date: date is used to transform the given date into specific format variable | date:"YYYY-MM-dd HH/mm/SS"
DecimalPipe	Transforms a number into a string with a decimal point, formatted according to locale rules. decimal : it is used to transform given integers to decimals. value | decimal : ' min digits before decimal . min digits after decimal - max digits after decimal '
I18nPluralPipe	Maps a value to a string that pluralizes the value according to locale rules.
I18nSelectPipe	Maps a key to a custom selector that returns a desired value.
JsonPipe	Transforms an object to a string representation via JSON.stringify, intended for debugging. json: it converts given values of object into json obj | json
KeyValuePipe	Transforms Object or Map into an array of key value pairs.
LowerCasePipe	Transforms text to all lower case.
PercentPipe	Transforms a number to a percentage string, formatted according to locale rules. percentage: percentage pipe is used to transform the given data to specific percentage. it multiplies value with 100 and return with % in given format value | percent : ' min digits before decimal. min digits after decimal - max digits after decimal '
SlicePipe	Creates a new Array or String containing a subset (slice) of the elements.
TitleCasePipe	Transforms text to title case.
UpperCasePipe	Transforms text to all upper case.

Using pipes

Angular's pipe operator uses the vertical bar character (|), within a template expression. The pipe operator is a binary operator– the left-hand operand is the value passed to the transformation function, and the right side operand is the name of the pipe and any additional arguments (described below).

Total: {{ amount | currency }}

In this example, the value of amount is passed into the CurrencyPipe where the pipe name is currency. It then renders the default currency for the user's locale.

Combining multiple pipes in the same expression

You can apply multiple transformations to a value by using multiple pipe operators. Angular runs the pipes from left to right.
The following example demonstrates a combination of pipes to display a localized date in all uppercase:

```
<p>The event will occur on {{ scheduledOn | date | uppercase }}.</p>
```

Passing parameters to pipes

Some pipes accept parameters to configure the transformation. To specify a parameter, append the pipe name with a colon (:) followed by the parameter value.
For example, the DatePipe is able to take parameters to format the date in a specific way.

```
<p>The event will occur at {{ scheduledOn | date:'hh:mm' }}.</p>
```

Some pipes may accept multiple parameters. You can specify additional parameter values separated by the colon character (:).
For example, we can also pass a second optional parameter to control the timezone.

```
<p>The event will occur at {{ scheduledOn | date:'hh:mm':'UTC' }}.</p>
```

How pipes work

Conceptually, pipes are functions that accept an input value and return a transformed value.

```
import { Component } from '@angular/core';
import { CurrencyPipe} from '@angular/common';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CurrencyPipe],
  template: `
    <main>
      <p>Total: {{ amount | currency }}</p>
    </main>
  `,
})
export class AppComponent {
  amount = 123.45;
}
```

In this example:

1. CurrencyPipe is imported from @angular/common
2. CurrencyPipe is added to the imports array
3. The amount data is passed to the currency pipe

Pipe operator precedence

The pipe operator has lower precedence than other binary operators, including +, -, *, /, %, &&, ||, and ??.

<!-- firstName and lastName are concatenated before the result is passed to the uppercase pipe -->
{{ (firstName + lastName | uppercase }}

The pipe operator has higher precedence than the conditional (ternary) operator.

{{ (isAdmin ? 'Access granted' : 'Access denied') | uppercase }}

If the same expression were written without parentheses:

{{ isAdmin ? 'Access granted' : 'Access denied' | uppercase }}

It will be parsed instead as:

{{ isAdmin ? 'Access granted' : ('Access denied' | uppercase) }}

Always use parentheses in your expressions when operator precedence may be ambiguous.

Change detection with pipes

By default, all pipes are considered pure, which means that it only executes when a primitive input value (such as a String, Number, Boolean, or Symbol) or a changed object reference (such as Array, Object, Function, or Date). Pure pipes offer a performance advantage because Angular can avoid calling the transformation function if the passed value has not changed.
As a result, this means that mutations to object properties or array items are not detected unless the entire object or array reference is replaced with a different instance. If you want this level of change detection, refer to detecting changes within arrays or objects.

Creating custom pipes

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

- A name, specified in the pipe decorator
- A method named transform that performs the value transformation.

To create custom pipe use following scaffolding :
ng generate pipe square

```
C:\Users\pc\Desktop\NareshIt\730Angular\Angular\Projects\fakeStore>ng genera
te pipe square
CREATE src/app/square.pipe.spec.ts (195 bytes)
CREATE src/app/square.pipe.ts (229 bytes)
UPDATE src/app/app.module.ts (473 bytes)
```

Above command will generate 2 files wisely square.pipe.spec.ts for testing and square.pipe.ts and update one file i.e app.module.ts shown above.

square.pipe.ts (No Parameters)

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

@Pipe({
  name: 'square'
})
export class SquarePipe implements PipeTransform {

  transform(value: number, ...args: unknown[]): number {
    return value*value;
  }
}
```

```
C:\Users\pc\Desktop\NareshIt\730Angular\Angular\Projects\fakeStore>ng generate
pipe power
CREATE src/app/power.pipe.spec.ts (191 bytes)
CREATE src/app/power.pipe.ts (227 bytes)
UPDATE src/app/app.module.ts (532 bytes)
```

power.pipe.ts : (Accepts Parameters)

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

@Pipe({
  name: 'power'
})
export class PowerPipe implements PipeTransform {

  transform(value: number, powerValue:number): number {
    return Math.pow(value,powerValue);
  }
}
```

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

Here is an example of a custom pipe that transforms strings to kebab case:

```
// kebab-case.pipe.ts
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'kebabCase',
  standalone: true,
})
export class KebabCasePipe implements PipeTransform {
  transform(value: string): string {
    return value.toLowerCase().replace(/ /g, '-');   }  }
```

Using the @Pipe decorator

When creating a custom pipe, import Pipe from the @angular/core package and use it as a decorator for the TypeScript class.

```
import { Pipe } from '@angular/core';
@Pipe({
  name: 'myCustomTransformation',
  standalone: true
})
export class MyCustomTransformationPipe {}
```

The @Pipe decorator requires two configuration options:

- name: The pipe name that will be used in a template
- standalone: true - Ensures the pipe can be used in standalone applications

Naming convention for custom pipes

The naming convention for custom pipes consists of two conventions:

- name - camelCase is recommended. Do not use hyphens.
- class name - PascalCase version of the name with Pipe appended to the end

Implement the PipeTransform interface

In addition to the @Pipe decorator, custom pipes should always implement the PipeTransform interface from @angular/core.

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'myCustomTransformation',
  standalone: true
})
export class MyCustomTransformationPipe implements PipeTransform {}
```

Implementing this interface ensures that your pipe class has the correct structure.

Transforming the value of a pipe

Every transformation is invoked by the transform method with the first parameter being the value being passed in and the return value being the transformed value.

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'myCustomTransformation',
  standalone: true
})
export class MyCustomTransformationPipe implements PipeTransform {
  transform(value: string): string {
    return `My custom transformation of ${value}.`
  } }
```

Adding parameters to a custom pipe

You can add parameters to your transformation by adding additional parameters to the transform method:

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'myCustomTransformation',
  standalone: true
})
export class MyCustomTransformationPipe implements PipeTransform {
  transform(value: string, format: string): string {
    let msg = `My custom transformation of ${value}.`
    if (format === 'uppercase') {
      return msg.toUpperCase()
    else {
      return msg
    }
  }
}
```

Detecting change within arrays or objects

When you want a pipe to detect changes within arrays or objects, it must be marked as an impure function by passing the pure flag with a value of false. Avoid creating impure pipes unless absolutely necessary, as they can incur a significant performance penalty if used without care.

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'featuredItemsImpure',
  pure: false,
  standalone: true
})
export class FeaturedItemsImpurePipe implements PipeTransform {
  transform(value: string, format: string): string {
    let msg = `My custom transformation of ${value}.`
    if (format === 'uppercase') {
      return msg.toUpperCase()
    else {
      return msg
    }
  }
}
```

Angular developers often adopt the convention of including Impure in the pipe name and class name to indicate the potential performance pitfall to other developers.

II. other

Table of Contents:

1. Introduction to Pipes in Angular

Pipes provide a simple and efficient way to transform data before displaying it in the view.

Pipes are used to format, filter, and sort data and they can be used with both template-driven and reactive forms, as well as with other Angular components and services.

Pipes are mainly used to change the data display format.

- By using the Pipe operator (|), we can apply the Pipe's features to any of the property in our Angular project.
- In addition to that, we can also chain pipe and pass parameters to the Pipe.

2. Pure and Impure Pipes

- ✓ In Angular, pipes can be either pure or impure.
- ✓ Pure pipes are designed to be stateless, meaning that they don't have any internal state that could affect their output.
- ✓ Instead, they take input data and return transformed output data. Pure pipes are also memorized, which means that if the input data hasn't changed since the last time the pipe was called, the pipe won't be executed again.
- ✓ The benefit of using pure pipes is that they can help improve the performance of your Angular application, since they only execute when necessary. Additionally, pure pipes can help prevent unnecessary change detection cycles, which can improve overall application performance.
- ✓ To create a pure pipe in Angular, you need to add the pure: true option to the @Pipe decorator, like this:

```
@Pipe({
  name: 'myPurePipe',
  pure: true
})
```

- ✓ Impure pipes can be useful in some cases, such as when you need to perform a heavy calculation or retrieve data from an external API.
- ✓ However, it's generally recommended to use pure pipes whenever possible to improve performance and prevent unnecessary change detection cycles.

3. Built-in Pipes in Angular

Angular comes with a set of built-in pipes that you can use in your templates.

Here are some of the most commonly used built-in pipes in Angular:

- Currency Pipe
- Date Pipe
- Json Pipe

- LowerCase Pipe
- UpperCase Pipe
- PercentPipe
- SlicePipe
- TitleCasePipe
- AsyncPipe

1. Currency Pipe

CurrencyPipe is a built-in pipe in Angular that is used to format a number as a currency value.

It provides a way to display currency values in a user-friendly format, taking into account the currency symbol, decimal separator, and grouping separators based on the current locale.

Here's an example of how to use the CurrencyPipe in Angular:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  price: number = 12345.6789;
}
```

app.component.html:

```
<div>
  <h2>Using CurrencyPipe</h2>
  <p>Price: {{ price | currency }}</p>
  <p>Price: {{ price | currency:'EUR':'symbol-narrow':'4.2-2' }}</p>
</div>
```

In the above example, we have a price variable that holds a number value of 12345.6789. We then use the currency pipe to format the price variable as a currency value in the template.

The first usage of the pipe is with the default settings. It will format the price variable with the default currency of the current locale.

The second usage of the pipe includes some additional parameters. It formats the price variable with the EUR currency symbol, a narrow symbol, and a format string of 4.2-2. The format string indicates that the number should have a minimum of 4 digits before the decimal separator, a maximum of 2 digits after the decimal separator, and should use the locale's decimal and grouping separators.

When the above code is run, it will display the following output:

```
Using CurrencyPipe
Price: $12,345.68
Price: €12,345.68
```

The first output line shows the default formatted currency value for the current locale. The second output line shows a custom-formatted currency value with the Euro currency symbol, a narrow symbol, and a specific format string.

2. Date Pipe

DatePipe is a built-in pipe in Angular that is used to format a date object.

It provides a way to display date values in a user-friendly format based on the current locale.

Here's an example of how to use the DatePipe in Angular:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  currentDate: Date = new Date();
}
```

app.component.html:

```
<div>
  <h2>Using DatePipe</h2>
  <p>Current date: {{ currentDate | date }}</p>
  <p>Current date: {{ currentDate | date:'fullDate' }}</p>
  <p>Current date: {{ currentDate | date:'short' }}</p>
</div>
```

In the above example, we have a currentDate variable that holds the current date object. We then use the date pipe to format the currentDate variable as a date value in the template.

The first usage of the pipe is with the default settings. It will format the currentDate variable with the default date format of the current locale.

The second usage of the pipe includes a format string of fullDate. This will format the currentDate variable as a full date string, such as "Tuesday, March 8, 2022".

The third usage of the pipe includes a format string of short. This will format the currentDate variable as a short date string, such as "3/8/22".

When the above code is run, it will display the following output:

Using DatePipe
Current date: Mar 3, 2023
Current date: Friday, March 3, 2023
Current date: 3/3/23, 12:17 AM

The first output line shows the default formatted date value for the current locale. The second output line shows the currentDate variable formatted as a full date string. The third output line shows the currentDate variable formatted as a short date string.

3. Json Pipe

JsonPipe is a built-in pipe in Angular that is used to transform an object into a JSON string.

It provides a way to display object values in a formatted JSON string.

Here's an example of how to use the JsonPipe in Angular:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  myObject: any = {
    name: 'John',
    age: 30,
    email: 'john@example.com'
  };
}
```

app.component.html:

```
<div>
  <h2>Using JsonPipe</h2>
  <pre>{{ myObject | json }}</pre>
</div>
```

In the above example, we have a myObject variable that holds an object with a name, age, and email property. We then use the json pipe to transform the myObject variable into a JSON string.

The pre tag is used to preserve white spaces and formatting in the output.

When the above code is run, it will display the following output:

```
Using JsonPipe
{
  "name": "John",
  "age": 30,
  "email": "john@example.com"
}
```

The output shows the myObject variable transformed into a JSON string with formatted white spaces. This can be useful for debugging and displaying object values in a readable format.

4. LowerCase Pipe

LowerCasePipe is a built-in pipe in Angular that is used to transform a string into a lowercased string.

Here's an example of how to use the LowerCasePipe in Angular:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  myString: string = 'This is a STRING in Mixed CASE';
}
```

app.component.html:

```
<div>
  <h2>Using LowerCasePipe</h2>
  <p>Original String: {{ myString }}</p>
  <p>Lowercased String: {{ myString | lowercase }}</p>
</div>
```

In the above example, we have a myString variable that holds a string value in mixed case. We then use the lowercase pipe to transform the myString variable into a lowercased string.

When the above code is run, it will display the following output:

```
Using LowerCasePipe
Original String: This is a STRING in Mixed CASE
Lowercased String: this is a string in mixed case
```

The first output line shows the original string value. The second output line shows the transformed string value after using the lowercase pipe. Note that all characters in the string have been transformed to lowercase.

5. UpperCase Pipe

UpperCasePipe is a built-in pipe in Angular that is used to transform a string into an uppercased string.

Here's an example of how to use the UpperCasePipe in Angular:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  myString: string = 'This is a STRING in Mixed CASE';
}
```

app.component.html:

```
<div>
  <h2>Using UpperCasePipe</h2>
  <p>Original String: {{ myString }}</p>
  <p>Uppercased String: {{ myString | uppercase }}</p>
</div>
```

In the above example, we have a myString variable that holds a string value in mixed case. We then use the uppercase pipe to transform the myString variable into an uppercased string.

When the above code is run, it will display the following output:

```
Using UpperCasePipe
Original String: This is a STRING in Mixed CASE
Uppercased String: THIS IS A STRING IN MIXED CASE
```

The first output line shows the original string value. The second output line shows the transformed string value after using the uppercase pipe. Note that all characters in the string have been transformed to uppercase.

5. Percent Pipe

PercentPipe is a built-in pipe in Angular that is used to transform a number into a percent value.

Here's an example of how to use the PercentPipe in Angular:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  myNumber: number = 0.23;
}
```

app.component.html:

```
<div>
  <h2>Using PercentPipe</h2>
  <p>Original Number: {{ myNumber }}</p>
  <p>Percentage Value: {{ myNumber | percent }}</p>
</div>
```

In the above example, we have a myNumber variable that holds a number value. We then use the percent pipe to transform the myNumber variable into a percentage value.

When the above code is run, it will display the following output:

```
Using PercentPipe
Original Number: 0.23
Percentage Value: 23%
```

The first output line shows the original number value. The second output line shows the transformed percentage value after using the percent pipe. Note that the decimal value is multiplied by 100 and a percentage sign is added to the end of the value.

6. Slice Pipe

SlicePipe is a built-in pipe in Angular that is used to create a new array or string that contains a portion of an existing array or string.

Here's an example of how to use the SlicePipe in Angular:

app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  myArray: any[] = ['apple', 'banana', 'orange', 'grape', 'mango'];
  myString: string = 'This is a long string.';
}
```

app.component.html:

```
<div>
  <h2>Using SlicePipe on Array</h2>
  <p>Original Array: {{ myArray }}</p>
  <p>Sliced Array: {{ myArray | slice:1:3 }}</p>
</div>
<div>
  <h2>Using SlicePipe on String</h2>
  <p>Original String: {{ myString }}</p>
  <p>Sliced String: {{ myString | slice:0:7 }}</p>
</div>
```

In the above example, we have a myArray variable that holds an array of fruits and a mySecodString variable that holds a string value. We then use the slice pipe to create a new array or string that contains a portion of the original array or string.
When the above code is run, it will display the following output:
Using SlicePipe on Array

Original Array: apple,banana,orange,grape,mango
Sliced Array: banana,orange

Using SlicePipe on String

Original String: This is a long string.
Sliced String: This is
The first output section shows the original array value and the sliced array value using the slice pipe. Note that the slice starts from index 1 and ends at index 3, which means that the sliced array will contain the elements at index 1 and index 2 (not including index 3). The second output section shows the original string value and the sliced string value using the slice pipe. Note that the slice starts from index 0 and ends at index 7, which means that the sliced string will contain the characters at index 0 to index 6 (not including index 7).

7. TitleCase Pipe

TitleCasePipe is a built-in pipe in Angular that is used to transform a string into a title case, which means the first letter of each word is capitalized.

Here's an example of how to use the TitleCasePipe in Angular:
app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: './app.component.html'
})
export class AppComponent {
  myString: string = 'this is a sentence in lowercase.';
}
```

In the above example, we have a myString variable that holds a string value. We then use the titlecase pipe to transform the myString variable into a title case.

```
<h2>Using TitleCasePipe</h2>
<p>Original String: {{ myString }}</p>
<p>Transformed String: {{ myString | titlecase }}</p>
```

When the above code is run, it will display the following output:

```
Using TitleCasePipe
Original String: this is a sentence in lowercase.
Transformed String: This Is A Sentence In Lowercase.
```

The first output line shows the original string value. The second output line shows the transformed title case value after using the titlecase pipe.

Note that the first letter of each word is capitalized.

8. Async Pipe

AsyncPipe is a built-in pipe in Angular that is used to handle asynchronous data streams. It is commonly used to subscribe to observables or promises and display the emitted values in the view.

Here's an example of how to use the AsyncPipe in Angular:

app.component.ts:

```
import { Component } from '@angular/core';
import { Observable, of } from 'rxjs';
@Component({
  selector: 'app-root',
  template: './app.component.html'
})
export class AppComponent {
  myObservable$: Observable<number> = of(42);
  myPromise$: Promise<string> = Promise.resolve('Hello World!');
}
```

In the above example, we have a myObservable$ variable that holds an observable that emits the number 42. We also have a myPromise$ variable that holds a promise that resolves to the string Hello World!. We then use the async pipe to subscribe to these observables and promises and display the emitted values in the view.

When the above code is run, it will display the following output:

```
    <h2>Using AsyncPipe with Observable</h2>
    <p>{{ myObservable$ | async }}</p>
    <h2>Using AsyncPipe with Promise</h2>
    <p>{{ myPromise$ | async }}</p>
```

```
Using AsyncPipe with Observable
42
Using AsyncPipe with Promise
Hello World!
```

The first output section shows the value emitted by the observable using the async pipe. The second output section shows the value resolved by the promise using the async pipe. Note that the async pipe automatically subscribes to the observables and promises and unsubscribes when the component is destroyed to prevent memory leaks.

4. Creating Custom Pipes

Custom pipes are used to transform the data in an Angular application.

You can create custom pipes in Angular by defining a new class and implementing the PipeTransform interface. The PipeTransform interface contains a single method called transform that takes an input value and returns the transformed value.

custom.pipe.ts:

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({name: 'filterByLength'})
export class CustomPipe implements PipeTransform {
  transform(values: string[], minLength: number): string[] {
    return values.filter(value => value.length >= minLength);
  }
}
```

In the above example, we define a CustomPipe class that implements the PipeTransform interface. The transform() method takes two arguments - an array of strings and a minimum length. It then filters out any strings in the array that are greater than or equal to the specified length.

We then decorate the class with the @Pipe decorator and provide a name property to give the pipe a name. The name is what we'll use to reference the pipe in our templates.

app.module.ts

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

import { CustomPipe } from "./custom.pipe";

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

@NgModule({
  declarations: [AppComponent, CustomPipe],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: './app.component.html'
})
export class AppComponent {
  values: string[] = ['apple', 'banana', 'carrot', 'date'];
}
```

```
    <h2>Using Custom Pipe</h2>
    <ul>
      <li *ngFor="let value of values | filterByLength: 5">{{ value }}</li>
    </ul>
```

In the above example, we have a values array that contains some strings. We then use the filterByLength pipe to filter out any strings that are shorter than 5 characters. We use the *ngFor directive to loop through the filtered values and display them in an unordered list.

When the above code is run, it will display the following output:

Using Custom Pipe
apple
banana
carrot

The output shows the two strings from the values array that are longer than or equal to 5 characters.

5. Chaining The Pipes

Chaining pipes in Angular involves applying multiple pipes in sequence to transform data in a template. You can chain pipes by using the pipe operator (|) multiple times, with each pipe representing a separate transformation.

For example, suppose you have a date string that you want to format and then convert to uppercase. You could chain the date and uppercase pipes like this:

{{ **myDate | date:'medium' | uppercase** }}

In this example, the myDate value is first passed through the date pipe, which formats the date using the 'medium' format. The resulting value is then passed through the uppercase pipe, which converts the value to uppercase.

It's important to note that chaining too many pipes can impact performance, particularly when working with large datasets. If you find yourself chaining multiple pipes in a template, consider moving the transformation logic to a custom pipe or transforming the data in the component before passing it to the template. This can help improve performance and make the code easier to read and maintain.

6. Summary Of Key Points

- Pipes are used to transform data in Angular templates before it is displayed to the user.
- Angular comes with several built-in pipes, such as CurrencyPipe, DatePipe, DecimalPipe, UpperCasePipe, LowerCasePipe, TitleCasePipe, and AsyncPipe.
- Pipes can be used in template expressions with the | character.
- Pipes can also take one or more arguments, which are passed after the | character.
- You can create your own custom pipes by defining a class that implements the PipeTransform interface and adding it to the declarations array of your module.
- When defining a custom pipe, you should provide a name for the pipe, which can be used in the template, and a transformation function that takes the input data and any arguments and returns the transformed data.
- To use a custom pipe in the template, you should add the pipe name after the | character and any arguments after a colon :.
- When working with pipes, it's important to ensure that you're using the correct pipe name and arguments and that you're passing the correct input data to the pipe. If a pipe isn't working as expected, you may need to check these things to find the issue.

7. Best Practices to use Pipes

Here are some best practices to keep in mind when using pipes in Angular:

1. **Use built-in pipes whenever possible** — Angular provides a number of built-in pipes for common transformations, such as DatePipe, CurrencyPipe, and DecimalPipe. These pipes have been optimized for performance and should be used whenever possible to avoid unnecessary overhead.
2. **Avoid chaining too many pipes** — Chaining multiple pipes can cause performance issues, particularly when working with large datasets. Instead, consider using a custom pipe or transforming the data in the component before passing it to the template.
3. **Be mindful of performance** — Pipes can be expensive operations, particularly when working with large datasets. To avoid performance issues, try to limit the number of times a pipe is called, and avoid performing heavy calculations within a pipe.
4. **Keep pipes simple** — Pipes should be simple and focused on a single transformation. Avoid creating pipes that do too much or have complex logic, as this can make them difficult to maintain and debug.
5. **Use pure pipes when possible** — Pure pipes are only called when their input changes, which can improve performance. When creating a custom pipe, consider making it pure if possible.
6. **Be aware of data types** — Pipes work differently depending on the data type being transformed. For example, DatePipe expects a Date object as input, while CurrencyPipe expects a number. Make sure you're passing the correct data type to the pipe to avoid unexpected behavior.
7. **Test your pipes** — When creating a custom pipe, make sure to test it thoroughly to ensure it's working as expected. This can include unit testing the pipe's transformation logic, as well as testing it in the context of a component.

By following these best practices, you can ensure that your pipes are performant, maintainable, and effective at transforming data in your Angular applications.

8. Final Thoughts and Recommendations

Overall, pipes are a powerful feature in Angular that allow you to transform data in templates without requiring additional code in the component.

By using built-in pipes or creating custom pipes, you can format dates, numbers, and strings, filter data, and perform other useful transformations.

When using pipes in your Angular applications, it's important to keep performance in mind.

Avoid chaining too many pipes and be mindful of heavy calculations within pipes, as these can impact performance.

Additionally, consider using pure pipes and testing your custom pipes thoroughly to ensure they're working as expected.

Read more : https://www.c-sharpcorner.com/article/angular-pipes-with-examples/

Component Communication:
If there are more than one component exists then the one of component can send data to another component this is called component communication.
Nested Component: If inside one component there are one or more components present then this is called nested component.
In this case if parent component wants to send data to child component and vice versa this is called nested component communication.

Steps to create nested component:
Step 1: Create application as below
ng new ProjectNestedComponent --standalone=false
Step 2: Create new component using below scaffolding inside **ProjectNestedComponent**
ng generate component <Component_Name>
ng generate component Movies

```
D:\Dot Net Core NIT\Angular\FRONT_END\Angular\Angular>cd ProjectCompoCommunication

D:\Dot Net Core NIT\Angular\FRONT_END\Angular\Angular\ProjectCompoCommunication>ng generate component Movies
CREATE src/app/movies/movies.component.html (22 bytes)
CREATE src/app/movies/movies.component.spec.ts (615 bytes)
CREATE src/app/movies/movies.component.ts (246 bytes)
CREATE src/app/movies/movies.component.css (0 bytes)

D:\Dot Net Core NIT\Angular\FRONT_END\Angular\Angular\ProjectCompoCommunication>ng generate component Movie
CREATE src/app/movie/movie.component.html (21 bytes)
CREATE src/app/movie/movie.component.spec.ts (608 bytes)
CREATE src/app/movie/movie.component.ts (242 bytes)
CREATE src/app/movie/movie.component.css (0 bytes)

D:\Dot Net Core NIT\Angular\FRONT_END\Angular\Angular\ProjectCompoCommunication>
```

app.component.ts

```
.MainProject{
    height: auto;
    background-color: aquamarine;
    border: 3px solid darkgreen;
}
```

app.component.html

```
<div class="MainProject">
    <app-movies></app-movies>
</div>
```

movies.component.html

```
<p>Component inside main App!</p>
<div class="Category">Marathi Movies</div>
<div class="Movies">
    <app-movie title="Navara Maza Navasacha" [imgURL]="nmn2URL"></app-movie>
    <app-movie title="Satya Shodhak" [imgURL]="SSURL"></app-movie>
    <app-movie title="Ek Gadi Baki Anadi" [imgURL]="ekGadiURL"></app-movie>
```

```
    <app-movie title="Zapatlela" [imgURL]="ZapatlelaURL"></app-movie>
    <app-movie title="Pachhadalela" [imgURL]="PachhadlelaURL"></app-
movie>
</div>
<div class="Category">Hindi Movies</div>
<div class="Movies">
    <app-movie title="Navara Maza Navasacha" [imgURL]="nmn2URL"></app-
movie>
    <app-movie title="Satya Shodhak" [imgURL]="SSURL"></app-movie>
    <app-movie title="Ek Gadi Baki Anadi" [imgURL]="ekGadiURL"></app-
movie>
    <app-movie title="Zapatlela" [imgURL]="ZapatlelaURL"></app-movie>
    <app-movie title="Pachhadalela" [imgURL]="PachhadlelaURL"></app-
movie>
</div>
<div class="Category">Telagu Movies</div>
<div class="Movies">
    <app-movie title="Navara Maza Navasacha" [imgURL]="nmn2URL"></app-
movie>
    <app-movie title="Satya Shodhak" [imgURL]="SSURL"></app-movie>
    <app-movie title="Ek Gadi Baki Anadi" [imgURL]="ekGadiURL"></app-
movie>
    <app-movie title="Zapatlela" [imgURL]="ZapatlelaURL"></app-movie>
    <app-movie title="Pachhadalela" [imgURL]="PachhadlelaURL"></app-
movie>
</div>
```

movies.component.ts

```
import { Component } from '@angular/core';
import { MovieComponent } from '../movie/movie.component';

@Component({
  selector: 'app-movies',
  standalone: true,
  imports: [MovieComponent],
  templateUrl: './movies.component.html',
  styleUrl: './movies.component.css'
})
export class MoviesComponent {
nmn2URL='https://upload.wikimedia.org/wikipedia/en/1/1b/Navra_Maza_Navsa
cha_2.jpg';
SSURL='https://m.media-
amazon.com/images/M/MV5BNjU4YzU3ZWItODViNS00ZTY0LWExMDMtYTViMjY
3ZjllODc3XkEyXkFqcGc@._V1_.jpg';
ekGadiURL='https://assets-in.bmscdn.com/discovery-
catalog/events/et00301093-qhmgyctbvc-landscape.jpg';
```

ZapatlelaURL='https://m.media-amazon.com/images/M/MV5BODhmZGE4ZDYtYjlhMS00YTk3LWEyYmYtNjI3YWMzMmM2MmMxXkEyXkFqcGc@._V1_.jpg';
PachhadlelaURL='https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEg1sO3QXhTJfFbcLFE-UHgs_1yTeqBNp4MeQf1HJq1pbs0oWVDJAKDXPG1HRUGFYKR49WtdVw4TBeZnoAt-8rdb67w9KPagm6gSv3Px3AtPFAkAFX_nZSRMNrOLWCB54Wsc5lcpsqIYhOk/s400/Pachadlela-.jpg';
}

movies.component.css

```
.Movies{
    display: flex;
    justify-content: space-between;
}
.Category{
    background-color: aqua;
    padding: 10px;
}
```

movie.component.html

```
<div class="LeafElement">
<img src="{{imgURL}}" alt="{{title}}" height="200px" width="200px">
<h3>{{title}}</h3>
</div>
```

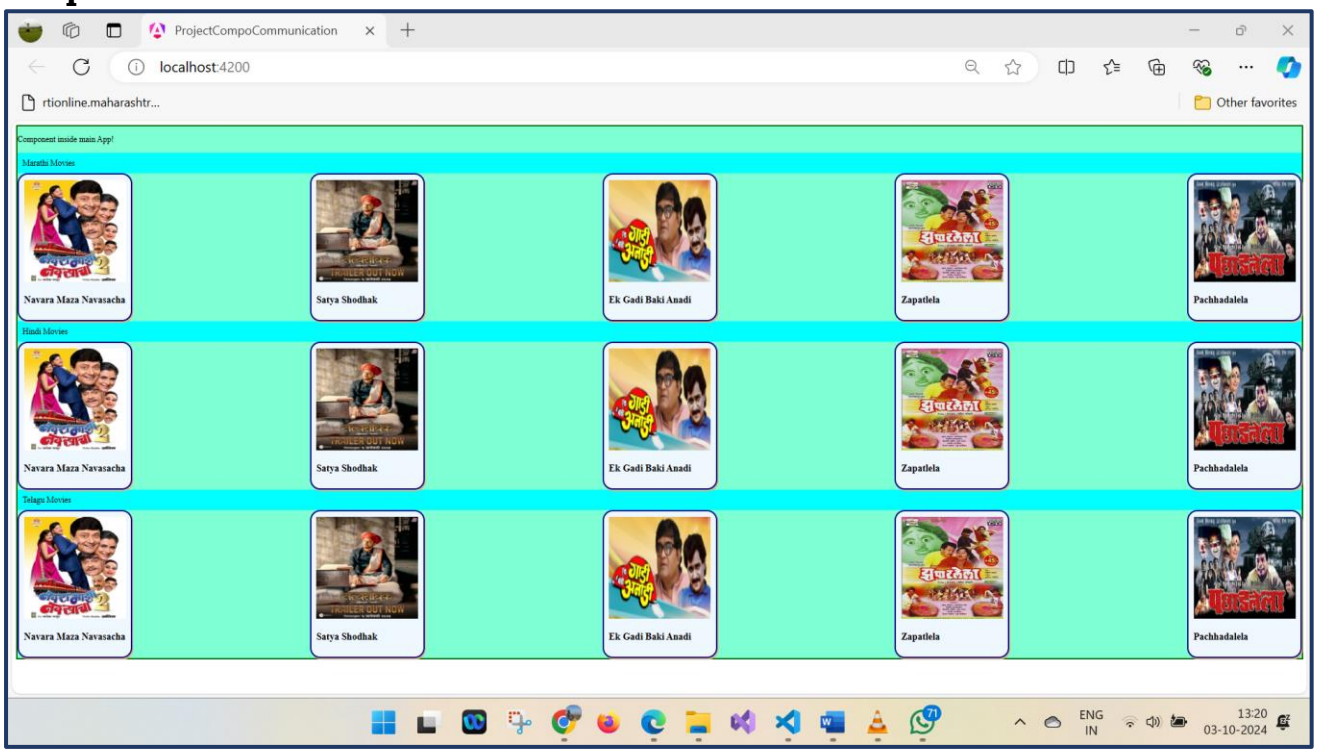
movie.component.ts

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

@Component({
  selector: 'app-movie',
  standalone: true,
  imports: [],
  templateUrl: './movie.component.html',
  styleUrl: './movie.component.css'
})
export class MovieComponent {
@Input() imgURL!:string;
@Input() title!:string;
}
```

movie.component.css

```
.LeafElement{
    padding: 10px;
    background-color: aliceblue;
    border: 3px solid darkblue;
    border-radius: 20px;
    box-shadow: 2px 3px burlywood;
}
```

Output:



In above example Parent sends data to child and child displays data.

Child to parent Communication:
Step 1: Create application named ProjectChild2Parent
ng new ProjectChild2Parent --standalone=false
Step 1: Inside ProjectChild2Parent, create component Books and Book
ng generate component <Component-Name>
ng generate component Books
ng generate component Book

To send data from Child to parent there are 3 different ways as follows:

h. @Output
i. @ViewChild
j. Template reference variable

@Output with EventEmitter :

Add a button with method in child component as below
child.component.ts

```
import { Component, EventEmitter, Output } from '@angular/core';
@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrl: './child.component.css'
})
export class ChildComponent {
@Output() myData = new EventEmitter<string>();

  btnClick(){
    this.myData.emit('This data is from Child');
  }
}
```

child.component.html

```
<p>child works!</p>
<button (click)="btnClick()">Child button</button>
```

parent.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html',
  styleUrl: './parent.component.css'
})
export class ParentComponent {
childtext:any;
  public childData($event:any): void{
    this.childtext=$event;
  }
}
```

parent.component.html

```
<p>parent works!</p>
<app-child (myData)="childData($event)"></app-child>
<h3>{{childtext}}</h3>
```

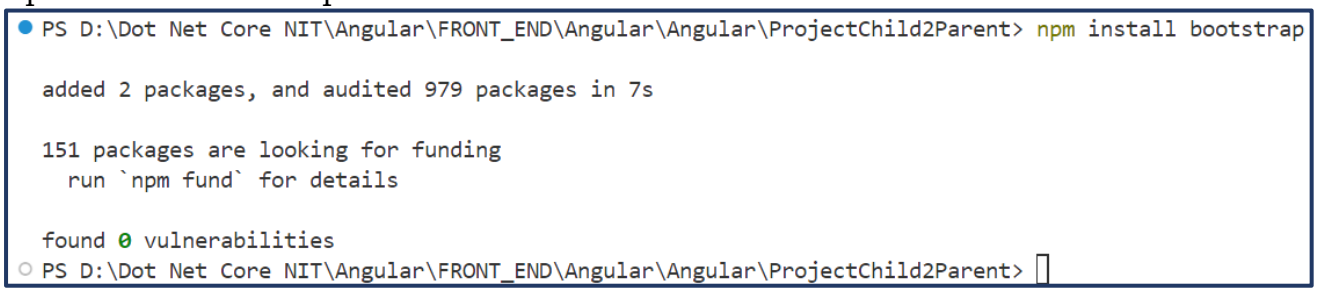
Pending as a practice:

@ViewChild

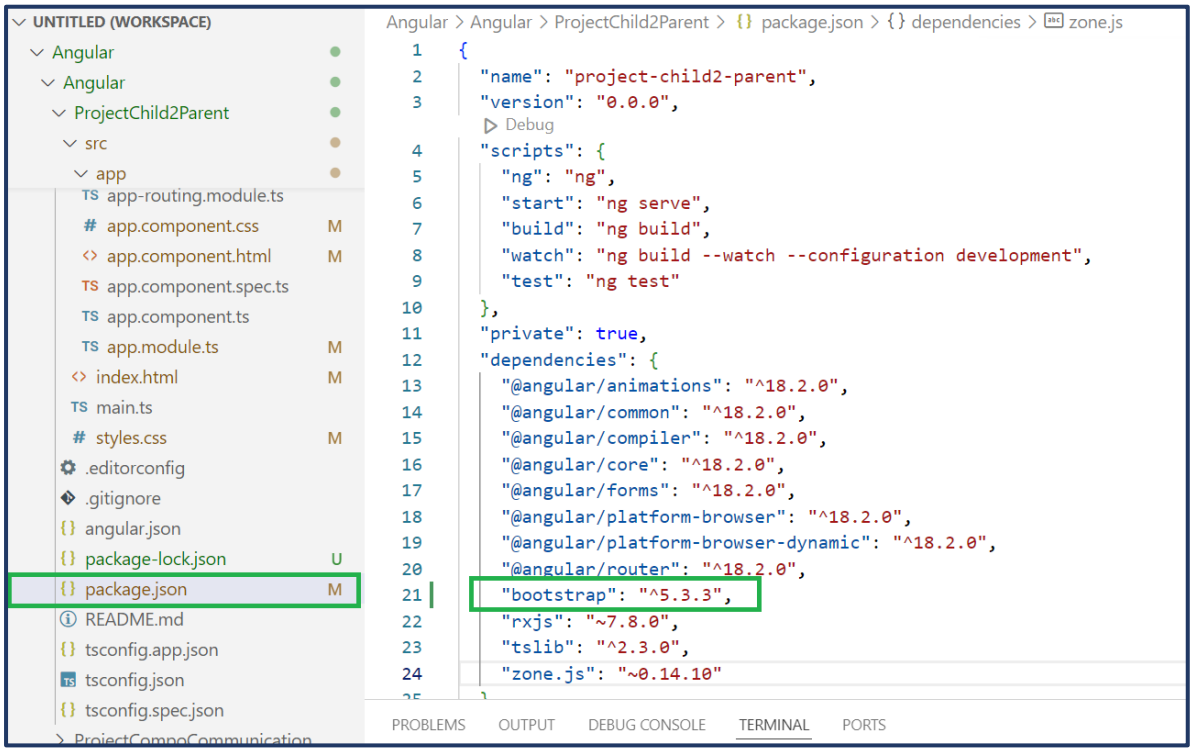
Template reference variable

Install and use Bootstrap in Angular:

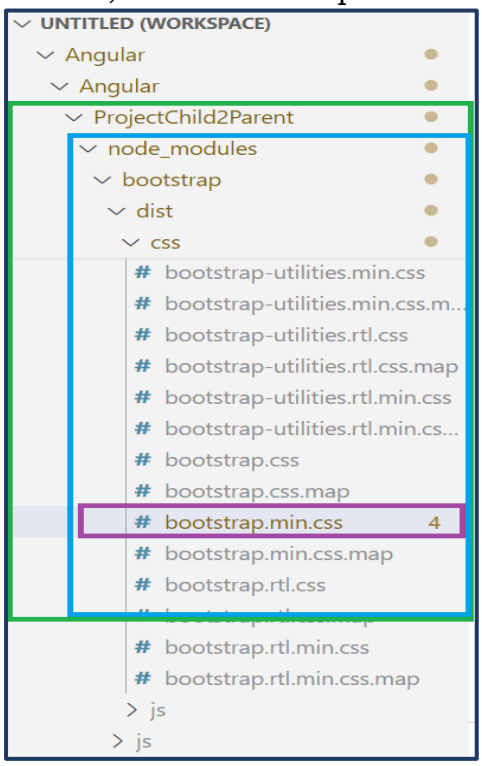
npm install bootstrap



To verify installation check package.json



To use, it must be imported into main style from the following path



Import it into style.css as

@import url('../node_modules/bootstrap/dist/css/bootstrap.min.css');

Now we will be able to use bootstrap in our complete application.

<input type="text" class="form-control form-control-lg" placeholder="Enter Book Name">

<button type="submit" class="btn btn-primary btn-lg">Send To Child</button>

---

https://www.youtube.com/watch?v=HYD9IrmAU5w&list=PL6OUUXajIr_iLmyOuCzIf6Vzz3EW95MpG&index=3

Services in Angular:

ARC

https://www.youtube.com/watch?v=U71G375Aw6E&list=PLp50dWW_m40XTcxIaXVqO60LaIlyHWxDS&index=74

services are classes that provide specific functionality and can be shared across components. They are used to encapsulate business logic, handle data communication, and manage state, making the application more modular and maintainable.

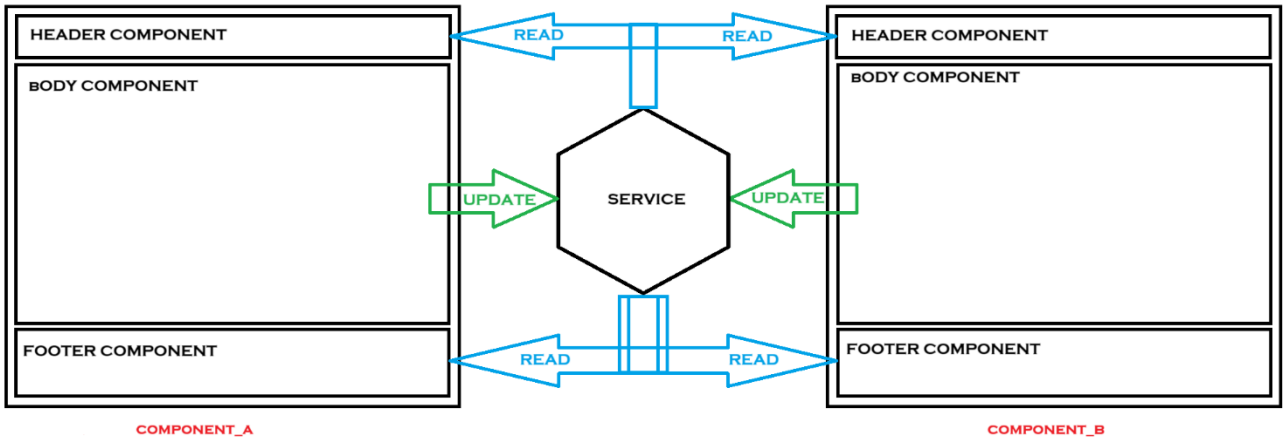
Services can depend on other services or can get deliverables from other services.

A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

Angular distinguishes components from services to increase modularity and reusability.

[Ideally, a component's job is to enable only the user experience. A component should present properties and methods for data binding to mediate between the view and the application logic. The view is what the template renders and the application logic is what includes the notion of a *model*.]

A component should use services for tasks that don't involve the view or application logic. Services are good for tasks such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an *injectable service class*, you make those tasks available to any component. You can also make your application more adaptable by injecting different providers of the same kind of service, as appropriate in different circumstances.



Here are some of the main advantages:

1. Code Reusability

- Angular services allow you to write logic that can be reused across different components of the application. Instead of duplicating code in multiple components, you can create a service that provides a specific function, like fetching data or handling business logic, and inject it wherever it's needed.

2. Separation of Concerns

- By using services, you separate the logic from the view (component). Components handle the UI and user interaction, while services manage tasks like data handling, APIs, or any other operations. This makes your code more modular and easier to maintain.

3. Dependency Injection (DI)

- Angular's built-in dependency injection system simplifies the process of providing services to components and other services. Angular takes care of instantiating the services and managing their lifecycle, which reduces the boilerplate code and makes your app more testable.

4. Singleton Nature

- Services in Angular, by default, are singletons when provided in the root module or a specific feature module. This means that a single instance of the service is created and shared across the app, ensuring consistency and reducing memory usage.

5. Encapsulation of Logic

- Services encapsulate the business logic of an application, allowing components to focus only on the presentation layer. This improves the maintainability and readability of your codebase.

6. Facilitates Unit Testing

- Since services handle business logic separately, testing becomes easier. You can mock services and test components independently from the actual implementation, leading to more effective and modular unit tests.

7. Inter-Component Communication

- Angular services enable sharing of data between components without using @Input and @Output. A service can act as a data store or event broadcaster, allowing different components to subscribe to changes and stay in sync.

8. Centralized Logic

- With Angular services, you can centralize logic for specific concerns such as authentication, logging, error handling, or configuration, making it easier to manage and modify as your application scales.

9. Improved Performance

- By moving logic out of components and into services, Angular allows for better separation of concerns, resulting in leaner components that perform better and are less prone to bugs.

10. Lazy Loading Support

- Services can be provided in feature modules, allowing for lazy loading. This means that the service is only loaded when needed, improving the startup performance of large applications.

In summary, Angular services enhance the scalability, testability, reusability, and performance of an application while fostering clean architecture through the separation of concerns.

There are several ways to implement and provide Angular services, depending on how and where you want to use them within your application. Here are the key methods:

1. Providing a Service in the Root (@Injectable({ providedIn: 'root' }))

- **Description**: This is the most common and recommended way to implement a service in Angular. When you set providedIn: 'root' in the @Injectable() decorator, Angular registers the service at the root injector level, making it a singleton that is available throughout the entire application.
- **Advantages**:

- No need to manually add the service to providers array in any module.
- The service is lazily loaded only when injected, so it doesn't increase the app's initial load time.

- **Example**:

```
@Injectable({
  providedIn: 'root'
})
export class DataService {
  constructor(private http: HttpClient) {}
}
```

2. Providing a Service in a Specific Module (providers array in @NgModule)

- **Description**: You can register a service in the providers array of a specific Angular module (e.g., a feature module). The service will then be scoped to that module and its children.
- **Advantages**:
  - Limits the service to a specific feature, helping to reduce memory consumption and improve performance.
- **Example**:

```
@NgModule({
  declarations: [...],
  imports: [...],
  providers: [DataService] // Service scoped to this module
})
export class FeatureModule {}
```

3. Providing a Service in a Specific Component (providers array in @Component)

- **Description**: A service can be provided at the component level by adding it to the providers array within a component's metadata. The service is then unique to that component and its child components.
- **Advantages**:
  - Each component that declares the service gets its own instance, which is useful if each instance of the component needs a fresh instance of the service.
- **Example**:

```
@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  providers: [DataService] // Scoped to this component and its children
})
export class MyComponent {
  constructor(private dataService: DataService) {}
}
```

4. Providing a Service in a Child Injector (Injector.create())

- **Description**: Angular allows you to create custom injectors manually using Injector.create(). This method provides more control over when and how the service is instantiated. It's used for advanced use cases.
- **Advantages**:

- Greater flexibility in how services are instantiated and injected, particularly in complex or dynamic use cases.

- **Example**:

```
const injector = Injector.create({
  providers: [{ provide: DataService, useClass: DataService }]
});
const dataService = injector.get(DataService);
```

5. Providing a Service with forRoot() Method (Singleton for a Module)

- **Description**: The forRoot() pattern is used when you want to provide a singleton service to a specific module (usually a core or shared module). It ensures that the service is instantiated only once, even if the module is imported multiple times.
- **Advantages**:
  - Prevents accidental multiple instances of services when modules are imported into different parts of the app.
- **Example**:

```
@NgModule({
  providers: [SharedService]
})
export class SharedModule {
  static forRoot(): ModuleWithProviders<SharedModule> {
    return {
      ngModule: SharedModule,
      providers: [SharedService] // Singleton service
    };
  }
}
```

6. Providing a Service with forChild() Method (Non-Singleton for a Module)

- **Description**: Similar to forRoot(), the forChild() method is used for feature modules where the service should not be shared globally but only among components within that feature.
- **Advantages**:
  - Provides feature-scoped services that don't affect the global state.
- **Example**:

```
@NgModule({
  providers: [FeatureService]
})
export class FeatureModule {
  static forChild(): ModuleWithProviders<FeatureModule> {
    return {
      ngModule: FeatureModule,
      providers: [FeatureService] // Non-singleton service
    };
  }
}
```

7. Using Factory Providers for Dynamic Service Creation

- **Description**: In some cases, you may need to create a service dynamically or based on certain runtime conditions. Angular's factory providers allow you to use a factory function to decide how a service should be instantiated.
- **Advantages**:
  - Allows flexibility to create services based on runtime parameters or application state.
- **Example**:

```
@Injectable({
  providedIn: 'root'
})
export class ConfigurableService {
  constructor(private config: Config) {}
}
const configProvider = {
  provide: ConfigurableService,
  useFactory: (config: Config) => new ConfigurableService(config),
  deps: [Config]
};
@NgModule({
  providers: [configProvider]
})
export class AppModule {}
```

Summary of Usage

- **Global service (singleton)**: Use providedIn: 'root' or register in the AppModule.
- **Feature-scoped service**: Register the service in a feature module or use the forRoot() method.
- **Component-scoped service**: Use the providers array inside a component's decorator.
- **Factory-based service**: Use factory providers for dynamic service instantiation.

By choosing the right method to implement your Angular services, you can ensure optimal performance, maintainability, and flexibility in your application.

Creating a service in Angular is a fundamental part of structuring an Angular application. Angular services allow you to encapsulate reusable logic that can be injected into different components or other services.

Step-by-Step Guide to Create a Service in Angular

1. Create a Service

- You can create a service using the Angular CLI or manually.
- The recommended way is to use the Angular CLI to generate a service as it automatically adds the necessary boilerplate code.

```
ng generate service my-service
```

This command creates two files:

- my-service.service.ts (the service logic)
- my-service.service.spec.ts (for unit testing)

2. Example Service Code

Let's say we are building a simple service that fetches data from an external API.

my-service.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root' // The service will be provided globally in the root module
})
export class MyService {

  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';

  constructor(private http: HttpClient) {}

  // Method to fetch posts from API
  getPosts(): Observable<any> {
    return this.http.get(this.apiUrl);
  }
}
```

- **@Injectable({ providedIn: 'root' })**: This decorator tells Angular that the service should be available globally (singleton).
- **HttpClient**: We inject HttpClient to make HTTP requests. Ensure you have imported HttpClientModule in your AppModule.

3. Provide the Service Globally (Optional)

If you're not using providedIn: 'root', you can provide the service in the root module manually. But with providedIn: 'root', this step is not necessary.

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
import { MyService } from './my-service.service'; // Import the service
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule // Import HttpClientModule to make HTTP calls
  ],
  providers: [MyService], // You can also register here, but it's unnecessary with providedIn: 'root'
  bootstrap: [AppComponent]
})
export class AppModule {}
```

4. Use the Service in a Component

Now that the service is created, inject it into a component where it's needed and use it.

app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { MyService } from './my-service.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  posts: any[] = [];
  constructor(private myService: MyService) {}
  ngOnInit() {
    // Fetch data from the service when the component is initialized
    this.myService.getPosts().subscribe((data) => {
      this.posts = data;
    });   }  }
```

5. Display Data in the Template

Finally, display the fetched data in the component template.

app.component.html

```
<h1>Posts</h1>
<ul>
  <li *ngFor="let post of posts">
    {{ post.title }}
  </li>
</ul>
```

6. Run the Application

After setting up the service and component, run your Angular application:

ng serve

Now the application will fetch data from the API and display it in the browser.

Complete Example Code

Service (my-service.service.ts)

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
export class MyService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';
  constructor(private http: HttpClient) {}
  getPosts(): Observable<any> {
    return this.http.get(this.apiUrl);
  }
}
```

Component (app.component.ts)

```
import { Component, OnInit } from '@angular/core';
import { MyService } from './my-service.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  posts: any[] = [];
  constructor(private myService: MyService) {}
  ngOnInit() {
    this.myService.getPosts().subscribe((data) => {
      this.posts = data;
    });
  }
}
```

Module (app.module.ts)

typescript
Copy code

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

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [MyService],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Summary:

- Use ng generate service to create a service.
- Use providedIn: 'root' for a singleton, globally available service.
- Inject the service into components using the constructor and make use of its methods.
- For HTTP services, ensure that HttpClientModule is imported into your AppModule.

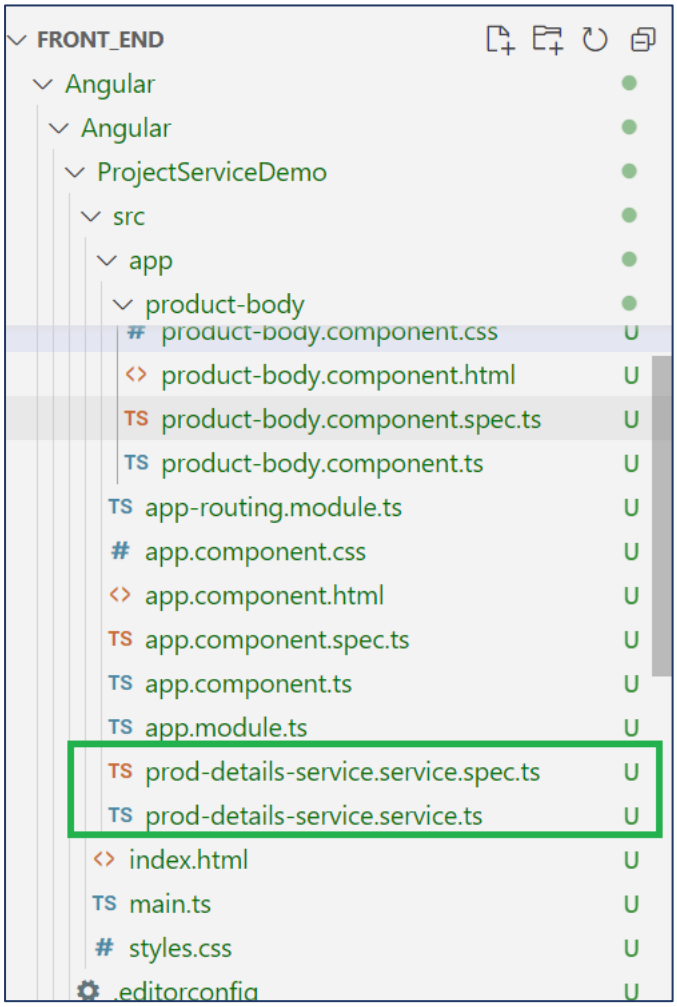
Create simple Service Demo Application:
ng new **ProjectServiceDemo** --standalone=false

ng generate component header
ng generate component footer
ng generate component productbody

ng generate service ProdDetailsService

Two Files will get generated from the above Service Command:

src/app/prod.service.spec.ts **: for testing purpose**
src/app/prod.service.ts **: for service logic purpose**



src/app/prod.service.ts contains following code:

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class ProdDetailsServiceService {
  constructor() { }
}
```

The @Injectable() decorator is used to mark a class as available for dependency injection. This decorator tells Angular's dependency injection system that the class can be injected into other components or services. The class with @Injectable() can then be provided to Angular's injector, allowing it to be used across different parts of the application.

When to Use @Injectable()

1. **Service Classes**: Most commonly, it is used for services that you want to inject into components, other services, or directives.
2. **Dependency Injection**: If your class depends on other services or needs dependencies injected, you'll typically use @Injectable() to mark it for injection.

Basic Example:

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

@Injectable({
  providedIn: 'root',  // This registers the service at the root level, making it a singleton
})
export class MyService {
  constructor() {
    console.log('MyService instantiated');
  }

  getMessage() {
    return 'Hello from MyService!';
  }
}
```

Explanation of @Injectable():

- **providedIn**: A feature added in Angular 6, where you can specify where the service should be provided (typically at the root level). It ensures that the service is a singleton and available globally if provided at the root.
    - **providedIn: 'root':** Makes the service available globally.
    - **providedIn: 'any':** Creates a new instance in lazy-loaded modules, allowing multiple instances if loaded in different modules.
    - **providedIn: 'platform':** Provides the service across different Angular applications within the same platform.

Key Points:

- **Injectable Classes**: You should use @Injectable() on classes that need to be injected into components or other services. This makes the class injectable and tells Angular to manage its creation and lifecycle.
- **Optional Dependencies**: The decorator is necessary when you want to inject dependencies into a class constructor, even if the class itself doesn't need to be injected elsewhere.

Example of Service Injection in a Component:

```
import { Component } from '@angular/core';
import { MyService } from './my-service.service';

@Component({
  selector: 'app-root',
  template: `<h1>{{ message }}</h1>`,
})
export class AppComponent {
  message: string;
  constructor(private myService: MyService) {
    this.message = this.myService.getMessage();
  }
}
```

In this example, MyService is injected into the AppComponent and used to get a message.

Use Cases:

- **Shared Services**: You can use @Injectable() to define services that can be shared across different parts of the application, such as for handling data, business logic, or state management.
- **Modular Services**: By controlling where services are provided (at root or module level), you can scope services appropriately based on the needs of your application.

Import the common service to the product component into : prod-body.component.ts file

```
import { ProdDetailsServiceService } from '../prod-details-service.service';
```

Dependency Injection:

Dependency Injection (DI) is a fundamental concept in Angular that allows you to provide dependencies to classes (like components, services, and directives) without them having to create those dependencies themselves.

We can achieve it **by creating an instance of service in the component constructor parameters.**

In other words :

Dependency Injection, or DI, is a design pattern and mechanism for creating and delivering some parts of an application to other parts of an application that require them.

Example :

private pds: ProdDetailsServiceService **is injected into** prod-body.component.ts file

```
constructor(private pds: ProdDetailsServiceService){

}
```

Example :

header.component.ts

```
import { Component } from '@angular/core';
import { ProdDetailsServiceService } from '../prod-details-service.service';
@Component({
  selector: 'app-header',
```

```
  templateUrl: './header.component.html',
  styleUrl: './header.component.css'
})
export class HeaderComponent {
  c=0;
constructor(private pds: ProdDetailsServiceService){
}
  getData(){
    this.c=this.pds.getC();
  }
}
```

header.component.html

```
    <P>My Site</P>
    <p>Count: {{c}}</p>
    <button (click)="getData()">Get Data</button>
```

product-body.component.ts

```
import { Component } from '@angular/core';
import { ProdDetailsServiceService } from '../prod-details-service.service';
@Component({
  selector: 'app-product-body',
  templateUrl: './product-body.component.html',
  styleUrl: './product-body.component.css'
})
export class ProductBodyComponent {
  constructor(private pds: ProdDetailsServiceService){
  }
  Increase(){
      this.pds.Increament();
  }
}
```

product-body.component.html

```
<div class="Product">
    <img src="https://images-cdn.ubuy.co.in/654b8e16333d0d5c3130b3e8-lige-mens-watches-waterproof-stainless.jpg" alt="Wrish Watch">
    <h1>Wrist Watch</h1>
    <p>Price:₹23333</p>
    <button (click)="Increase()">Add to Cart</button>
</div>
```

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'ProjectServiceDemo';
}
```

Create header: https://www.youtube.com/watch?v=gg0-WMVQbdQ

Routing in Angular:

Routing is a mechanism used by angular framework to manage the "Paths" and "Routes" of the angular applications.

Routing strategy helps us in navigating between different views / components in angular application.

Angular Framework comes with "Router" module which has everything we need to design, develop and implement routes and navigation links.

IQ: is router singleton? What is it? How do you inject?

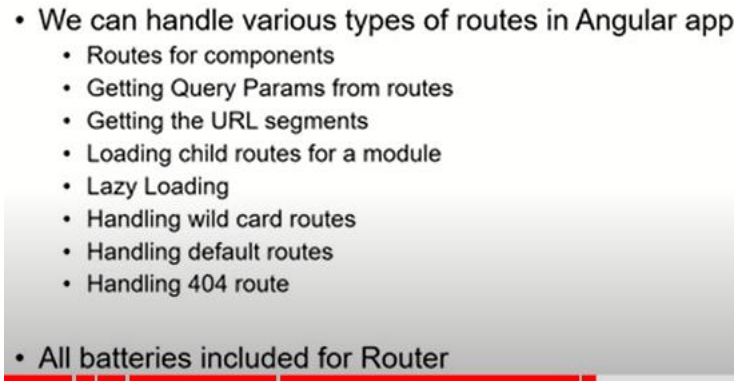
Route is a singleton – which mean **ONLY ONE** instance of route in our angular application.

Angular route is an official router module which is written and maintained by core angular team.

The router module is found in the package "@angular/router".

We need to setup router array – every time a request is made, the router will search in the list of array and find the most relevant match.

Router has states which helps us to get important information about the current state and data related to routes.



Example:

Home -> http://myapplication.com/ -> Default Route

Profile -> http://myapplication.com/profile -> Component Routing

Search -> http://myapplication.com/search?user=abc -> Query Params

Tasks -> http://myapplication.com/tasks/10/category/pending -> URL Segments/patterns

Users -> http://myapplication.com/users -> Module

view-user -> http://myapplication.com/users/view/10 -> Child Routes

edit-user -> http://myapplication.com/users/edit/10 -> Child Routes

add-user -> http://myapplication.com/users/add -> Child Routes

manage-user -> http://myapplication.com/users/manage -> Child Routes

PageNotFound -> http://myapplication.com/pageNotFound -> 404 error -> No matching routes.

Add <router-outlet></router-outlet> in app-component.html file to work with routing.

1. Router outlet is a built-in directive
2. Every Angular app should have "at least" 1 router outlet
   -> primary router outlet is default.
3. By default - the router outlet is defined in **app.component.html** file
4. Router outlet will match the matching routes for the components
   -> Takes its output
      -> inside the page
5. Multiple router outlets in application
   -> We can have more than 1 router outlet

Component Routes:

- **Routes for components**
  - Each component can have its own Routes
  - Various examples of **component routes are:**
    - /products
    - /products/view
    - /products/add
    - /users

Default files under new fresh applications
ng new RoutingDemo --standalone=false
that are used for routing are app-routing.module.ts and app.module.ts

```
//app-routing.module.ts
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
const routes: Routes = [];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

This module is configured in app.module.ts

```
//app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent
  ],
```

```
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

To understand the concept of routing create following **components** in above project.

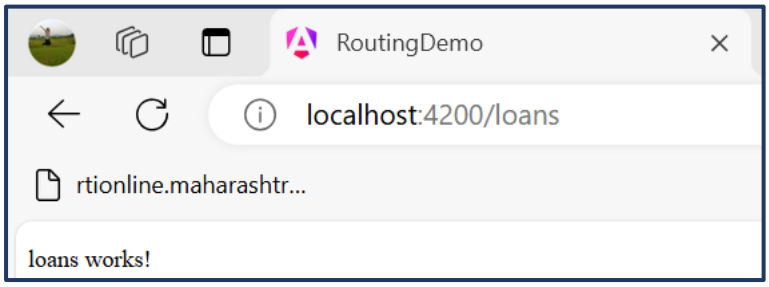
ng generate component loans
ng generate component loantypes
ng generate component add-loans

add routing path and component into routes array and import the respective component path(if not auto imported) as below

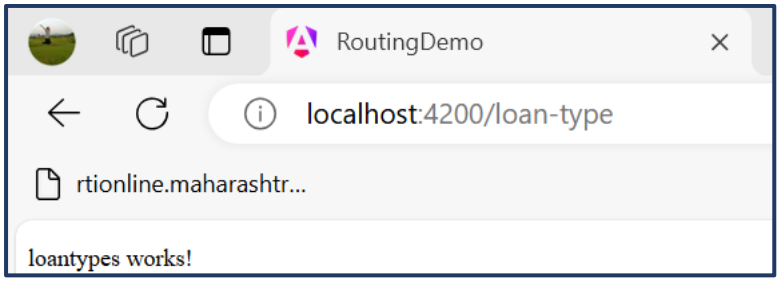
app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { LoansComponent } from './loans/loans.component';
import { LoantypesComponent } from './loantypes/loantypes.component';
import { AddLoansComponent } from './add-loans/add-loans.component';

const routes: Routes = [
  {
    path:'loans',
    component:LoansComponent
  },
  {
    path:'loan-type',
    component:LoantypesComponent
  },
  {
    path:'loans/add-loans',
    component : AddLoansComponent
  }
];

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

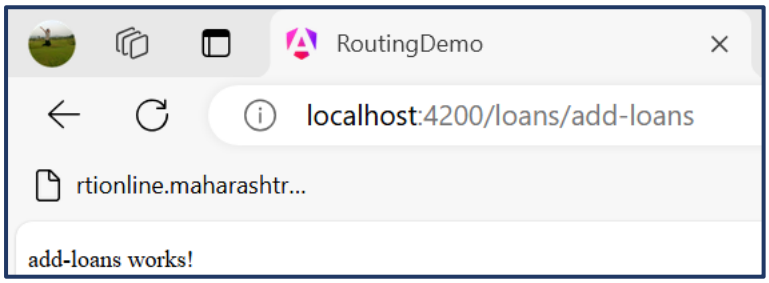
loans:



Loan-tyle:



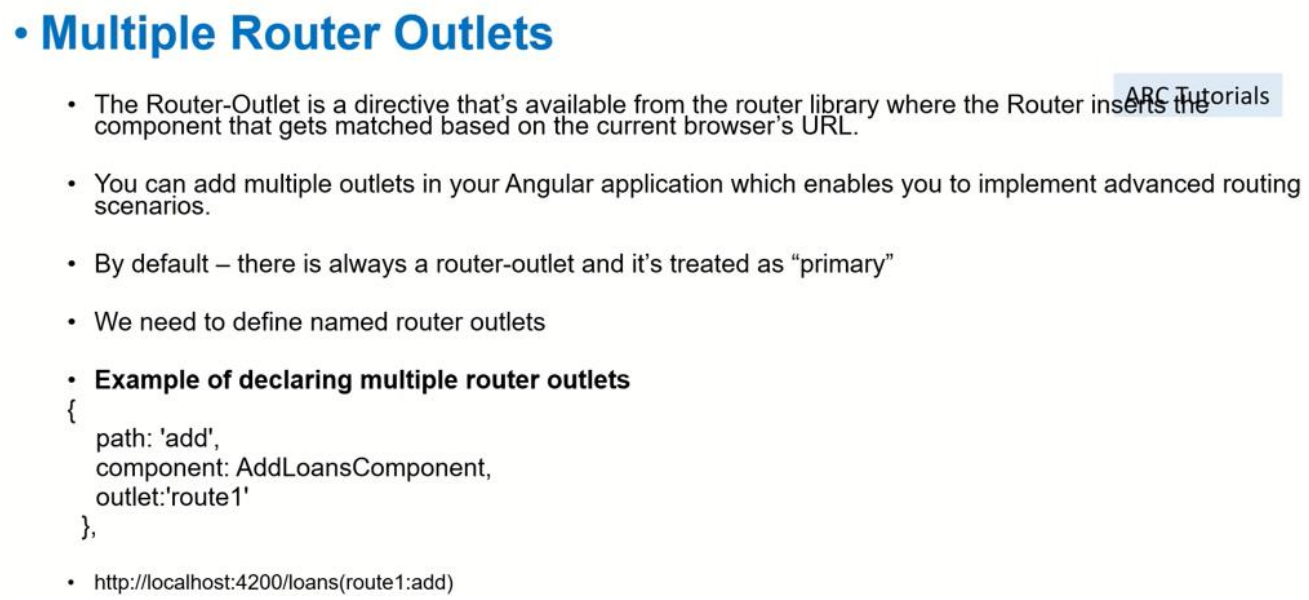
Loans/add-loans



Common mistakes:

1. Developers will add "/" in path at the start which should not be added.
2. Component is a class and its name should not be in quotes.
3. Bad formed array of routes
4. Sometimes your editor is not importing component correctly
5. Red color underline means there is some error
   -> Best practice is to leave router-outlet empty

Multiple Router Outlets:



1. We can have multiple router Outlets defined under app.component.html like

```
<!-- Primary Router Outlet -->
<router-outlet></router-outlet>
<!-- Named Router Outlet -->
<router-outlet name="addLoan"></router-outlet>
<!-- Named Router Outlet -->
<router-outlet name="editLoan"></router-outlet>
```

Add/edit routes array like below in app-routing.module.ts
If name is provided with outlet then it will route to respective, if no outlet parameter then it will load default router outlet.

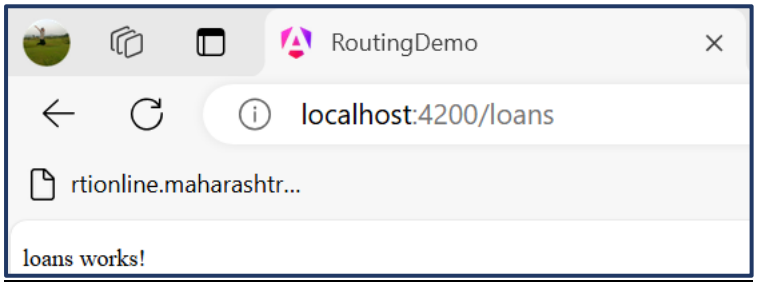
```
const routes: Routes = [
  {
    path:'loans',
    component:LoansComponent
  },
  {
    path:'loan-type',
    component:LoantypesComponent,
    outlet:'editLoan'
  },
  {
    path:'loans/add-loans',
    component : AddLoansComponent,
    outlet: 'addLoan'
  }
];
```

2. by default there is always/"at least" 1 router outlet in app.component.html file
3. When we don't provide any name for router-outlet it becomes primary
4. There should be only 1 primary

5. We can define multiple router outlets by giving name to them
6. That's why we call them "**named**" router outlets
7. We can give any name we want - give meaningful names, avoid using -,_ or other symbols.
8. It will NOT show if you directly access it in the URL, always access with the primary URL.
9. Syntax

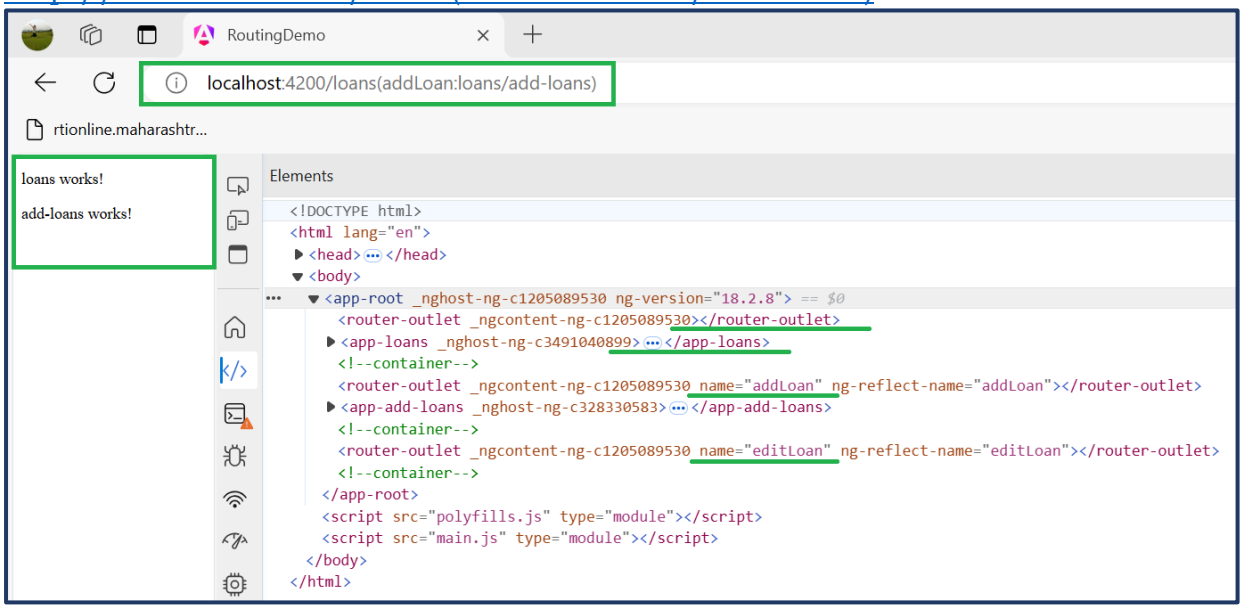
http://localhost:4200/<primary-route>(<routerOutletName>:<secondaryPath>)

Default loaded:



addLoan route selected:

http://localhost:4200/loans(addLoan:loans/add-loans)



10. Why are using this? It will create lot of problems of sharing data between tow outlets.
    - Avoid this use case in applications?
    - You can inject components
11. URL is not user friendly
    - bookmarkable URL
12. I have not personally seen this used a lot

Routing Strategy

• Routing – Routing Strategy

ARC Tutorials

- Before we start implementing our routes in our application, its important to understand and plan what will be our routing strategy
  - **import { LocationStrategy } from '@angular/common';**
- We need to add this in Providers of our Module
  - **{provide: LocationStrategy, useClass: HashLocationStrategy}**
- Angular provides 2 types of routing strategy we can use:
  - PathLocationStrategy
  - HashLocationStrategy
- **By default** – Angular makes use of the **PathLocationStrategy**
- With **HashLocationStrategy** - we will see the # in the URL

Location strategy refers to the way Angular manages and manipulates the browser's URL and navigation history. It determines how URLs are structured and how route changes reflect in the browser's address bar.

1. Routing behaviour of the applications URLs
2. Angular provides 2 types of routing strategies

- PathLocationStrategy

- Default routing strategy for Angular apps
- HTML 5 push state URL

- Examples

- http://myapp.com/dashboard
- http://myapp.com/user/10
- http://myapp.com/user/10/photos
- http://myapp.com/search?query=abc&state=ka&city=bengaluru

- HashLocationStrategy

To use this add/import

```
import {HashLocationStrategy, LocationStrategy} from '@angular/common';
```

into app.module.ts

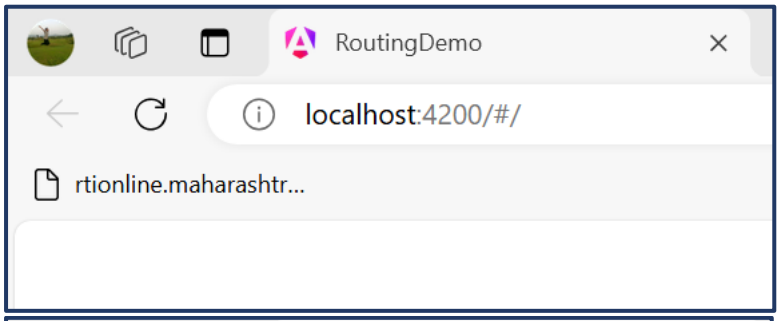
update providers array

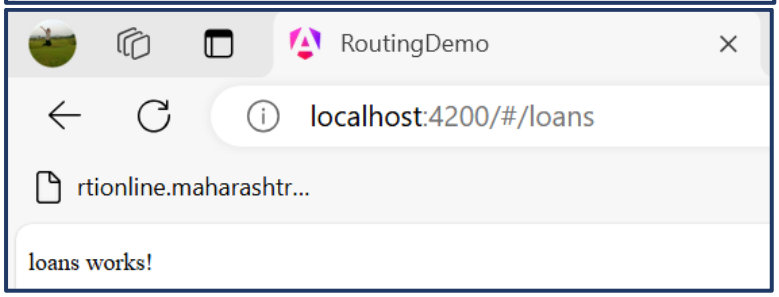
```
providers: [
    {provide:LocationStrategy, useClass:HashLocationStrategy}
  ]
```

- URL segments/patterns
- URLs will have hash in the URLs

- Examples

- http://myapp.com/#/dashboard
- http://myapp.com/#/user/10
- http://myapp.com/#/user/10/photos
- http://myapp.com/#/search?query=abc&state=ka&city=bengaluru





3. Hands-on examples for PathLocationStrategy
   - Default behaviour of Angular apps

4. Hands-on examples for HashLocationStrategy

- We need to import HashLocationStrategy from @angular/core

- Add it to Providers array

   - Angular will start loading our URLs using #

5. Why do we need 2 different types of routing?
   Angular is a SPA(single page app)
      - index.html
   Cloud vendors
      AWS
      GCP
      Azure
      Hosting Provider (Bluehost, Siteground, DigitalOcean)
   - /#/loans/add -> Route
   index.html/#/loans/add

6. Which one you should use when?
   Really there is no difference affect your application
   PathLocationStrategy
      -> Clean URLs
      -> Simple
      -> Bookmarbale
      -> Easy to Remember

Base Href:

•Routing – Base Href

ARC Tutorials

- Every Angular application has MANDATORY base href
- Angular application is a SPA (Single Page Architecture) which means there will be only one HTML file
- The default base href is set to "/" the root folder
- The Base HREF is present in index.html file for all Angular applications
- Wrong configuration leads to pointing to wrong folder root path
- Setting the base href using the command line –base-href=
- Syntax: <base href="/" >

1. Base HREF is mandatory for all Angular apps
2. Base HRef is present in your index.html file
3. The project is pointing to the "root" directory/folder of your server which is running at 4200 port

index.html

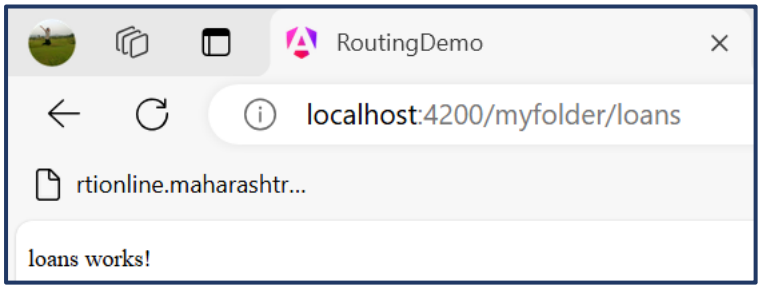
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>RoutingDemo</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

4. When you are deploying your Angular apps to server into root folder or other folder at that time base href value gets changed.
   -> http://myapp.com/app1/
      http://myapp.com

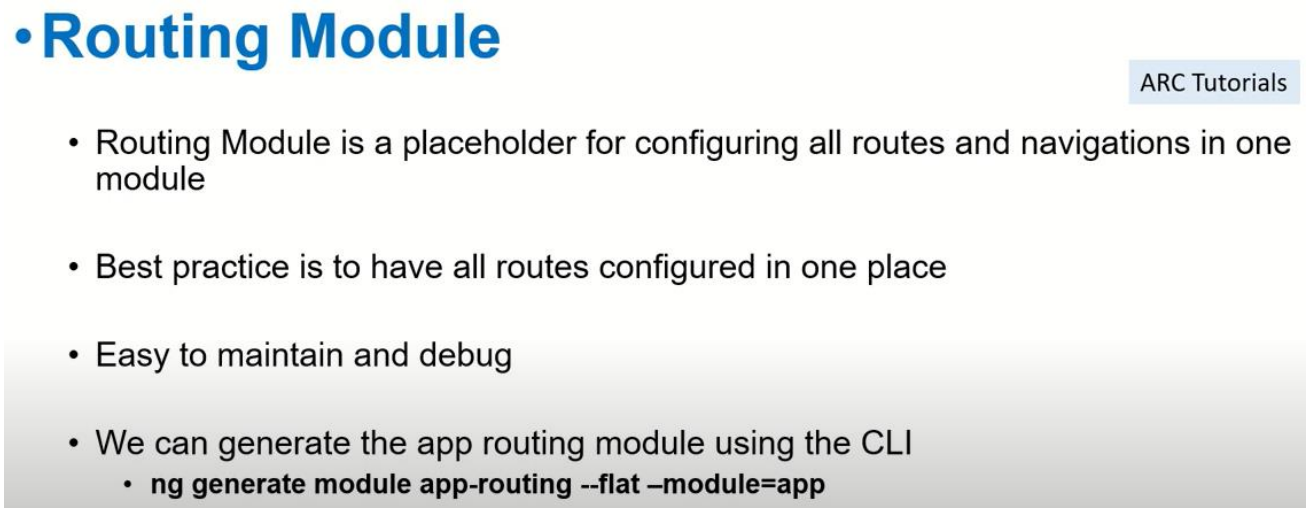
5. It decides where you want to deploy your app
   - thats why its extremely important

6. <base href="/">
There are different ways to configure it.
index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>RoutingDemo</title>
  <base href="/myfolder">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```



Router Module / Routing Module / app-routing Module:



1. Its a single module and placeholder where all our routes are configured for that particular module
2. Each module can have its own routes
3. During the angular app installation
- We get an option - Do you want to have routing in your application?
- It will automatically create the app-routing module file for us
4. ng g module app-routing --flat --module=app

Component Routes – Configuring Routes:

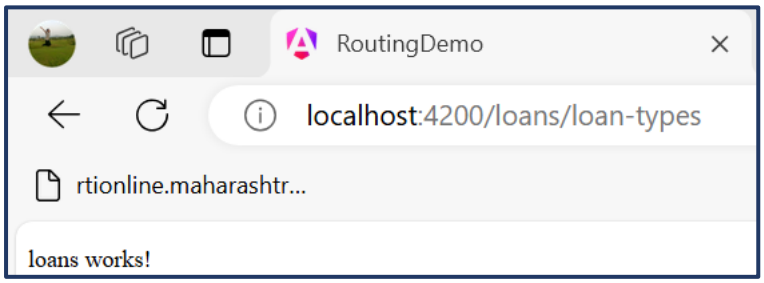
- We can configure routes to redirect route for various paths
  - Path
  - Component
  - redirectTo
  - Children
  - Outlet
  - pathMatch
- Let's learn how to configure routes in the routing module

1. There are various options that we can configure in Component Routes.
2. Some of the ones that we have seem in previous/earlier tutorials ar earlier are
   2.1 Path: Already discussed above
   2.2 component: Already discussed above
   2.3 Router Outlet: Already Discussed above
   2.4 children

We can have sub routes in component that can be children of other component and to represent the children of component we user children attribute in routes array as below

```
const routes: Routes = [
 {
  path:'loans',
  children:[
   {
    path:'addLoan', component:LoansComponent
   },
   {
    path:'editLoan', component:LoansComponent
   },
   {
    path:'loan-types', component:LoansComponent
   }
 ]
 },
 {
  path:'customers',
  children:[
   {
    path:'addCustomer', component:LoansComponent
   },
   {
    path:'editCustomer', component:LoansComponent,
    children:[
     {
      path:'movetomain',component:LoansComponent
     }
```

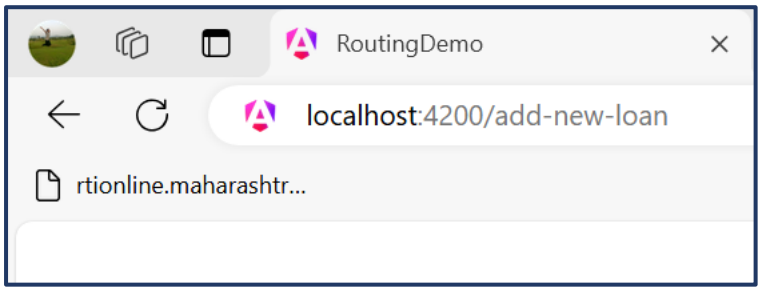
```
      ]
    },
    {
      path:'deleteCustomer', component:LoansComponent
    }
  ]
 }
];
```

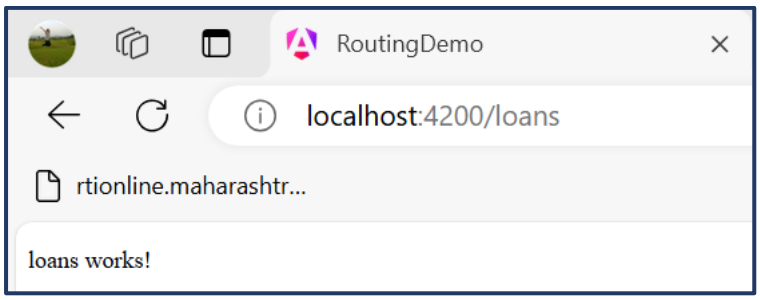


2.5 redirectTo: This is used to redirect to specific component or URL whenever we found specific component in the URL as follows:

Example:

```
const routes: Routes = [
  {
    path:'add-new-loan',
    redirectTo:'loans'
  },
  {
    path:'loans', component:LoansComponent
  }
];
```





2.6 pathMatch: pathMatch is an important property used in route definitions to specify how the router should match the URL path. It helps control which route gets activated based on the path in the browser.

pathMatch Options

There are two main options for pathMatch:

1. **full** – The full URL path must match exactly.
2. **prefix** – The URL must start with the specified path (default behavior).

pathMatch: 'full'

- The URL must **exactly match** the specified path in the route.
- This is commonly used with **redirects** or when you need strict matching.

Example:

```
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' }, // strict match
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
];
```

Behavior:

- When the path is exactly '' (empty), the router will **redirect** to /home.
- If you don't use pathMatch: 'full', Angular may match partial paths incorrectly (e.g., it could match a route like /home even when it's supposed to match just the empty path).

pathMatch: 'prefix' (Default Behavior)

- The route will match if the URL **starts with** the given path.
- This is useful for parent routes with child routes or shared path prefixes.

Example:

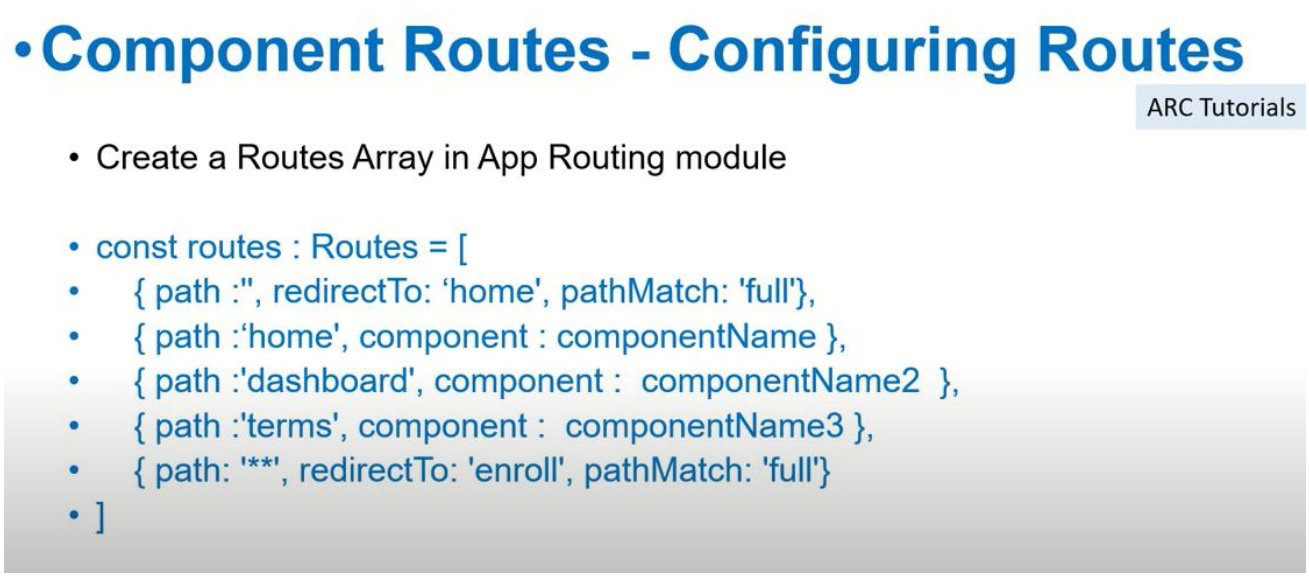
```
const routes: Routes = [
  { path: 'products', component: ProductsComponent, children: [
      { path: 'list', component: ProductListComponent },
      { path: 'details/:id', component: ProductDetailsComponent }
  ]},
];
```

Behavior:

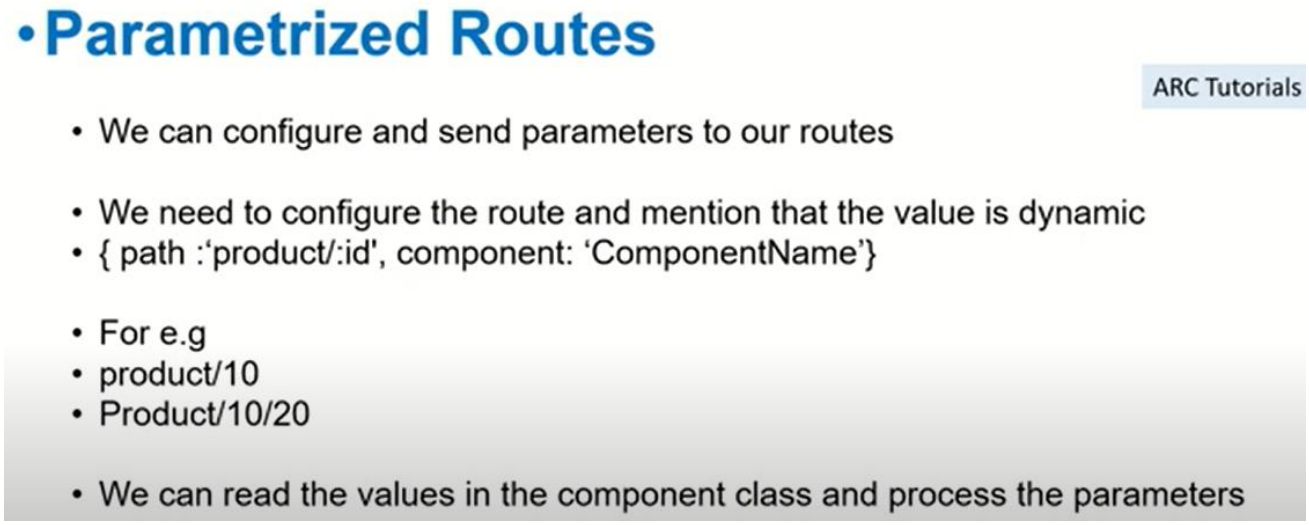
- The route products will match any URL starting with /products, such as /products/list or /products/details/1.
- Since prefix is the default behavior, you don't need to specify pathMatch explicitly unless you want to change it to full.

Key Differences Between full and prefix

pathMatch: 'full'	**pathMatch: 'prefix'**
Matches only if the entire URL matches exactly.	Matches if the URL starts with the specified path.
Useful for redirects or strict routing logic.	Useful for parent-child route structures.
Example: path: '' matches only '' (empty).	Example: path: 'products' matches /products/list.



Parameterized Routes / Dynamic Routes:



1. We can send dynamic data or parameters.
2. URLs will look something like this
   http://localhost.com/user/10 -> get the user with Id as 10
   http://localhost.com/search/ka/bangalore -> state and city
   http://localhost.com/user/10/photos/34 -> user id = 10 and photo id = 34
3. While writing dynamic URLs/Params - make sure you write :(colon) for dynamic data
4. Import the ActivatedRoute class

products.component.ts

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

@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrl: './products.component.css'
})
export class ProductsComponent{
```

```
  productId = 0;
  photoId = 0;
  constructor(private activatedRoute: ActivatedRoute){
  this.activatedRoute.params.subscribe((params)=>{
    console.log(params);
    const internValue = params;
    this.productId = internValue['ProductID'];
    this.photoId = internValue['PhotoId'];
  })
  }
}
```

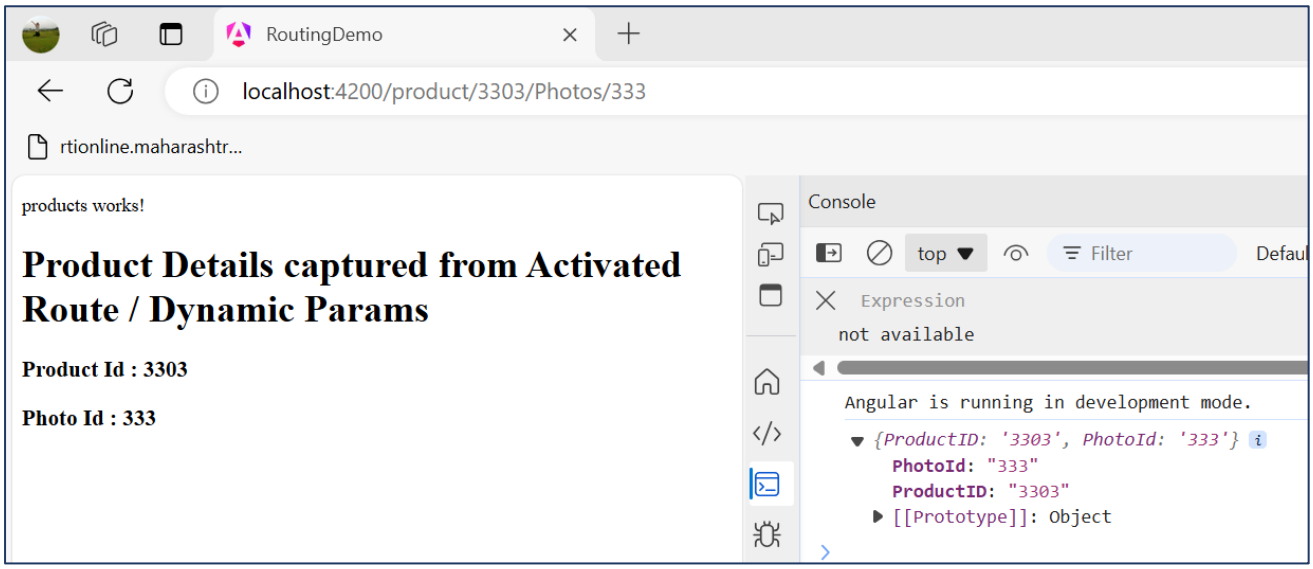
products.component.html

```
<p>products works!</p>
<h1>Product Details captured from Activated Route / Dynamic Params</h1>
<h3>Product Id : {{productId}}</h3>
<h3>Photo Id : {{photoId}} </h3>
```

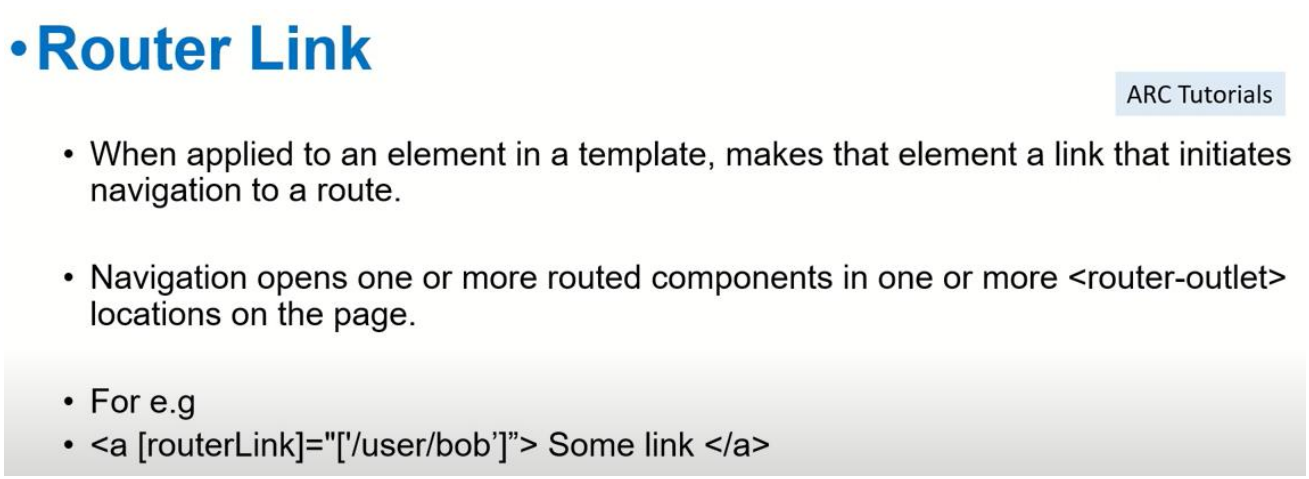
app-routing.module.ts

```
import { ProductsComponent } from './products/products.component';
const routes: Routes = [
  {path:'product/:id', component: ProductsComponent},
  {path:'product/:ProductID/Photos/:PhotoId', component:ProductsComponent}
];
```

5. Create an object in constructor parameter -> Dependency injection
6. We can create any number of dynamic params in our URLs



Router Links:



It is an attribute directive to apply to an element to make it a link.

1. We can have any number of router links in the template
2. Router Links can be static or can be dynamic in nature
3. Common Mistakes
   - Not putting strings in single quote
   - Not passing dynamic data correctly
4. Static Router Link : **<a [RouterLink]="'/user'"> **
5. Dynamic Router Link
6. We DO NOT have to put "/" in variables in router Link
7. Router Link Query Params : we will cover along with Query params in routes

Example :

Create component as below
ng generate component clients

Import and Add component into app-routing.module.ts

import { ClientsComponent } from './clients/clients.component';
{ path:'clients', component:ClientsComponent }

Write below code into clients component

//clients.component.css

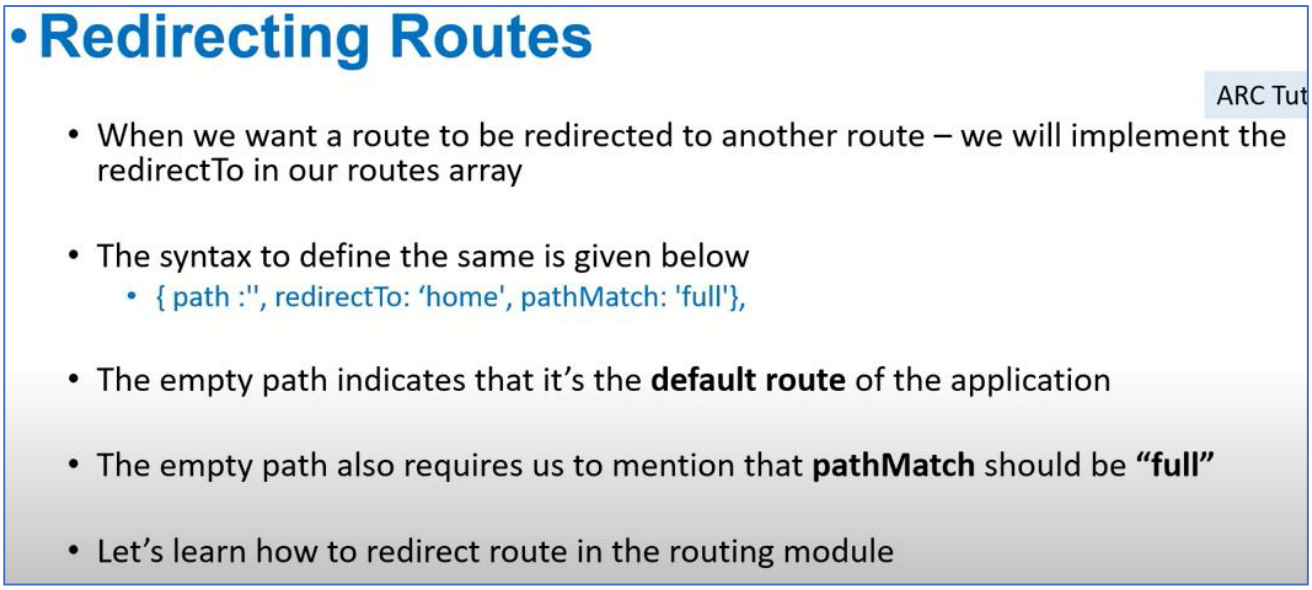
```
table, th, td {
    border: 3px solid green;
    border-collapse: collapse;
  }
```

//clients.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-clients',
  templateUrl: './clients.component.html',
  styleUrl: './clients.component.css'
})
```

```
export class ClientsComponent {
  clientList = [
    {clientId : 11, firstName:'Mahesh',lastName:'Raut'},
    {clientId : 22, firstName:'Ganesh',lastName:'Jadhav'},
    {clientId : 33, firstName:'Suresh',lastName:'Borchate'},
    {clientId : 44, firstName:'Bhavesh',lastName:'Kulkarni'},
    {clientId : 55, firstName:'Ritesh',lastName:'Borhade'},
    {clientId : 66, firstName:'Haresh',lastName:'Tikone'},
    {clientId : 77, firstName:'Vignesh',lastName:'Panchal'}
  ]
  constructor(){}
}
//clients.component.html
<p>clients works!</p>
<a [routerLink]="'/loans'">Loans Static Link</a>
<table>
    <th>Client ID</th>
    <th>First Name</th>
    <th>Last Name</th>
    <th>Static Links</th>
    <th>Dynamic Links 1</th>
    <th>Dynamic Links 2</th>
    <th>Dynamic Links 3</th>
    <tr *ngFor="let client of clientList">
        <td>{{client.clientId}}</td>
        <td>{{client.firstName}}</td>
        <td>{{client.lastName}}</td>
        <!-- localhost:4200/edit -->
        <td><a [routerLink]="'/edit'">Edit</a> | <a
[routerLink]="'/delete'">Delete</a></td>
        <!-- localhost:4200/edit/77
        localhost:4200/delete/33 -->
        <td><a [routerLink]="['/edit',client.clientId]">Dynamic Edit</a> | <a
[routerLink]="['/delete',client.clientId]">Dynamic Delete</a></td>
        <!-- localhost:4200/client/edit/77
        localhost:4200/client/delete/33 -->
        <td><a [routerLink]="['edit',client.clientId]">Dynamic Edit</a> | <a
[routerLink]="['delete',client.clientId]">Dynamic Delete</a></td>
        <!-- passing multiple parameters in single URL  -->
        <td><a
[routerLink]="['edit',client.clientId,'firstName',client.firstName]">Dynamic
Edit</a> | <a
[routerLink]="['delete',client.clientId,'firstName',client.firstName]">Dynamic
Delete</a></td>
    </tr>
</table>
```

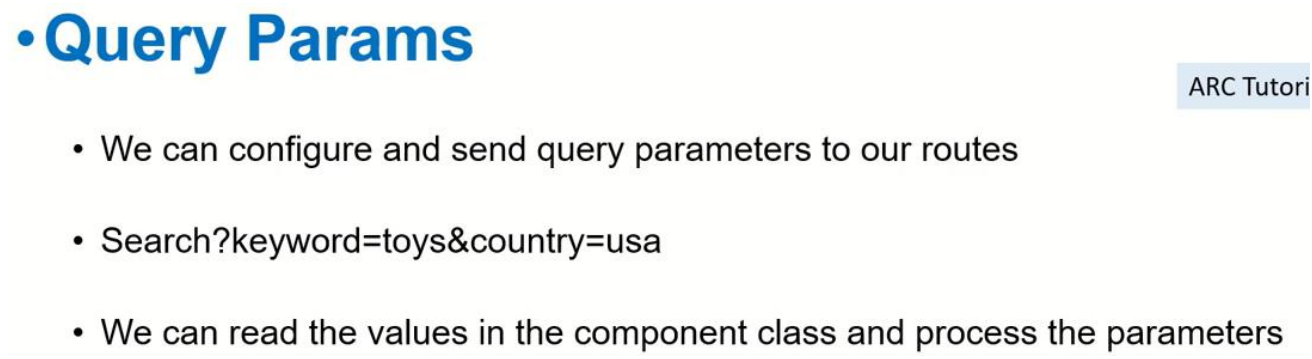
Redirecting Routes:



Whenever user enters default URL then we can transfer that URL to specific component by default.
1. By default the root level route is ''
2. redirectTo and specify which route it has to go

```
{
    path: '',
    redirectTo: 'home',
    pathMatch: 'full'
}
```

Query Params:



1. We can send data from Form to backend using query parameters.
2. We can have data from click ->
Basically -> URL -> http://localhost.com/search?key=10&state=ka&city=bangalore
3. Query Params -> visible in the URL
4. Mostly used for querying, searching or filtering data etc
facebook.com/search?page=10&pagesize=20

Example: To implement search functionality in component, Create component **"search"**
ng generate component search

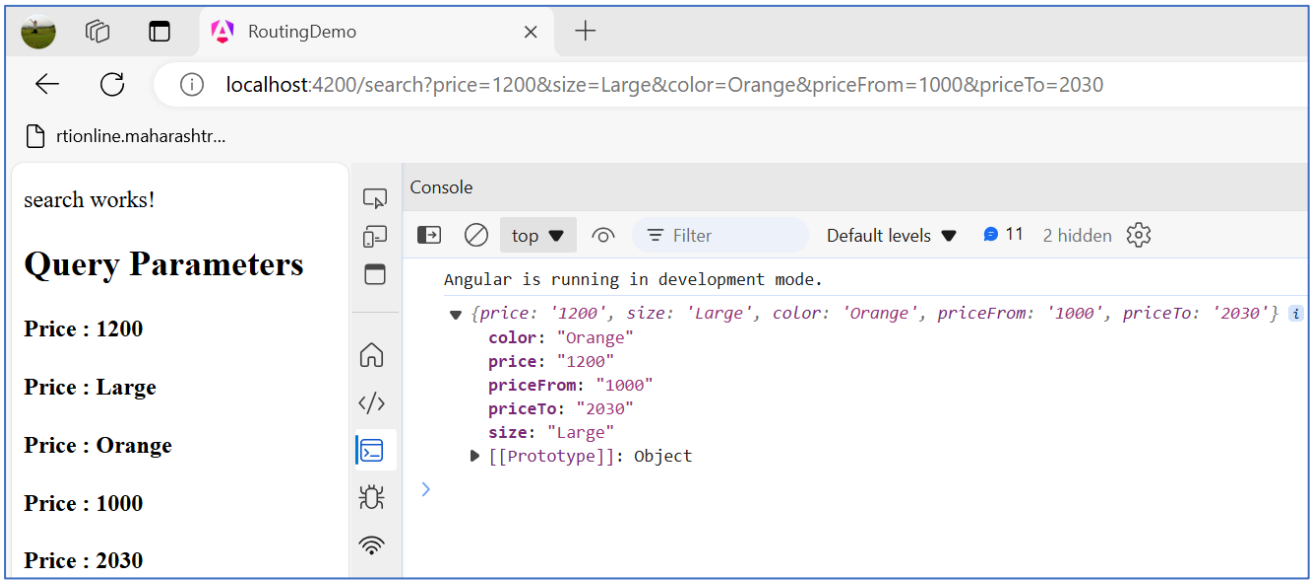
//app-routing.module.ts

```
import { SearchComponent } from './search/search.component';
const routes: Routes = [
  {path:'product/:id', component: ProductsComponent},
  {path:'product/:ProductID/Photos/:PhotoId', component:ProductsComponent},
  {
    path:'clients', component:ClientsComponent
  },
  {
    path:'search',component:SearchComponent
  }
];
```

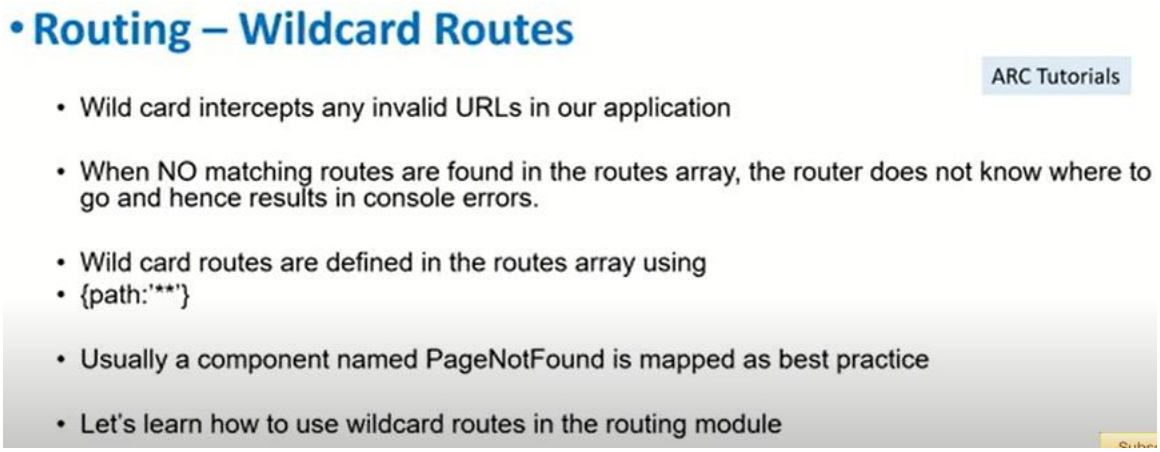
To capture the search functionality from URL parameters

```
//search.component.ts
import { Component } from '@angular/core';
import {ActivatedRoute} from '@angular/router';
@Component({
  selector: 'app-search',
  templateUrl: './search.component.html',
  styleUrl: './search.component.css'
})
export class SearchComponent {
  price=0;
  size='';
  color='';
  priceFrom=0;
  priceTo=0;
  constructor(private activatedRoute:ActivatedRoute){
    this.activatedRoute.queryParams.subscribe(params=>{
    console.log(params);
    this.price=params['price'];
    this.size=params['size'];
    this.color=params['color'];
    this.priceFrom=params['priceFrom'];
    this.priceTo=params['priceTo'];}
  );
  }
}
```

```
//search.component.html
<p>search works!</p>
<h2>Query Parameters</h2>
<h4>Price : {{price}}</h4>
<h4>Price : {{size}}</h4>
<h4>Price : {{color}}</h4>
<h4>Price : {{priceFrom}}</h4>
<h4>Price : {{priceTo}}</h4>
```



Wildcard Routes:



1. Any unmatched route will be intercepted by Wild card route and then the application will be routed to new page/component.
2. This has to be the last route in your configuration
3. We define by saying the path to match "**"
4. Any Path below the wildcard will be treated as wildcard

Create new component as below
ng generate component pageNotFound

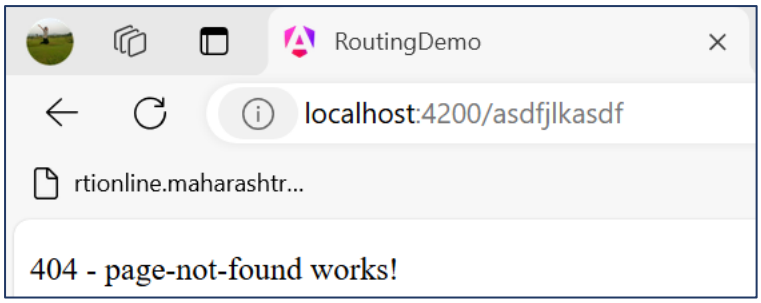
app-routing.module.ts

```
import { ProductsComponent } from './products/products.component';
import { ClientsComponent } from './clients/clients.component';
import { SearchComponent } from './search/search.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

const routes: Routes = [
  {path:'product/:id', component: ProductsComponent},
  {path:'product/:ProductID/Photos/:PhotoId', component:ProductsComponent},
  {
```

```
    path:'clients', component:ClientsComponent
  },
  {
    path:'search',component:SearchComponent
  },
  {
    path:'**',
    component:PageNotFoundComponent
  }
];
```

page-not-found.component.html

```
<p>404 - page-not-found works!</p>
```



Lazy Loading:

1. Any angular application is made up of multiple Modules

Example :

Loan Management System consists of following modules

- Loans
- Customers
- Payments
- Invoices
- Reports
- Authentication
- Authorization
- Downloads
- Admin

2. Angular by default will load all modules at start
   - As user Logins load all modules.
3. Loading all modules initially weather required or not:
   - Makes your application slow performance wise.
   - Also, it is a bad idea to expose modules which user is NOT going to use and which user should not see/use.
4. Lazy Loading comes into Picture:
   -> Initially we will load only modules which are mandatory
   -> Rest we will serve as "requested"
   -> We will create routes for each module

Example: if user request Payments page then only load payment module

/payments

1. It will increase performance app.
2. We can verify if the user has access to this module.

By default, ngModules are eagerly loaded, which means that as soon as the app loads, so do all the NgModules, whether or not they are immediately necessary.

For large scale apps with lot of routes, consider lazy loading is a design pattern that loads NgModules as needed.

Lazy loading helps to keep initial bundles sizes smaller, which in turns helps to decrease load time.

For angular, LoadChildren expects a function that uses the dynamic import syntax to import your lazy-loaded module only when its needed.

5. lazy Loading will help in keeping your builds smaller
   -> ng build / compile application to deploy
      -> files
         -> the size of those files will be smaller
         -> it will load fast
         -> it will respond better

6. If you are coming from previous version of Angular 8
   -> the syntax has changed
      -> please use expected function / syntax.

With lazy loaded modules in angular, it is easy to have features loaded only when the user navigates to their routes for the first time.

This can be a huge help for your applications performance and reducing the initial bundle size and its pretty straightforward to setup.
When the application grows in size, we should always modularize the application into individual.

Load the modules on demand(we can verify them in the console).
There are 2 steps to create lazy loading feature module
- Create Feature Module
- Configure loadChildren in appRouting.

Feature module is a module specific to certain functionality.
To load a feature module lazily(only on demand) we need to load its children using the loadChildren property in route configuration.

7. The modules generated using the Angular CLI - for lazy Loading
   -> There will be NO entry in AppModule
   -> Hence, it will not be loaded initially

Syntax to create lazy loading
8. ng g module <module_name> --route <module_route> --module app.module
E.g ng g module payments --route payments --module app.module

9. The above command will generate the following
   - A routing file for the module
   - A module file
   - A component
      - html
      - css/scss
      - spec
      - class
   - UPDATE the app routing module with an entry indicating payment module.

10. /payments is requested will load module on demand and its children - if needed
      /payments/success: payments is plane module and success is a child.

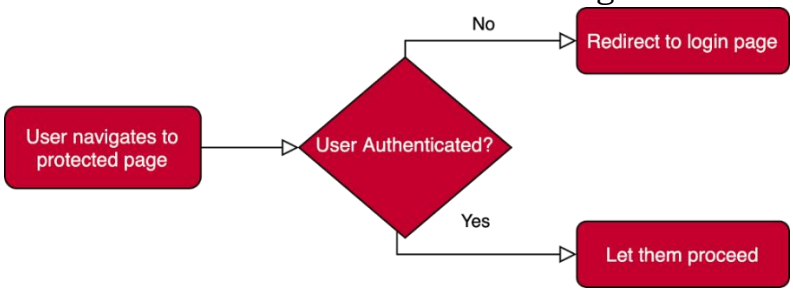
This means when application is loaded default, it will load all the pages except payment and success, but if we hit payment, it will load only payment and success pages only.

Route Guards:

- Use route guards to prevent users from navigating to parts of an application without authorization.
- Route Guards are used to secure the route paths
- In most cases, the routes and screens are protected behind a good authentication system.
- The route guard resolves to true or false based on custom logic and functionality. Example if the user is from different category, quota, regions, states or country, or if the user is logged in or not etc
- We can generate any number of guards based on our application requirements.

1. Route Guards helps us secure our routes and screens
2. E.g User tries to access Admin page this has to be filtered from Route Guard

User -> Route Guard -> Admin
-> Custom Logic -> True -> Admin
-> Then can access the route
-> False
-> custom logic on failure condition -> home



3. Generate Route Guard
   ng g guard <guard_name>

4. Route Guards have **"interfaces"**

Angular route guards are interfaces that allow us to control the access of a route. We can write a condition inside the class implementation. This enables extra protection for routes, whether to show route components or not. There are various interfaces exist to guard routes in different ways. On a single route, multiple route guards can be applied.

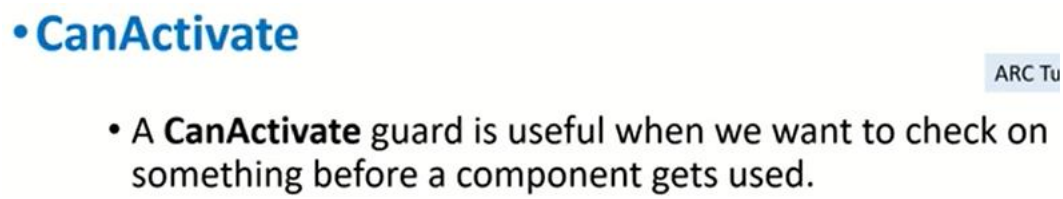
Note: Don't forget to add the created route guard service to the providers array in either specific NgModule or AppModule.

```
@NgMdoule({
  imports: [...],
  declarations: [...],
  providers: [
    ...,
    PermissionGuard,
  ],
})   export class AppModule { }
```

- canActivate	Can a user access a route
- canActivateChild	Can user access child routes of a parent route
- canDecativate	Check if user can exit the route
- canLoad	Can a lazy loaded module be loaded
- resolve	Route data retrieval before route activating
-canMatch	**CanMatch** is a route guard used to conditionally load **lazy-loaded routes** or **module-level routes** based on some logic. It is particularly useful when you want to **prevent module-level routing** based on conditions such as authentication or roles, improving performance by avoiding unnecessary module loading.

CanActivate:

```
PS D:\Dot Net Core NIT\Angular\FRONT_END\Angular\Angular\RoutingDemo> ng generate guard authguard
? Which type of guard would you like to create? CanActivate
CREATE src/app/authguard.guard.spec.ts (498 bytes)
CREATE src/app/authguard.guard.ts (138 bytes)
PS D:\Dot Net Core NIT\Angular\FRONT_END\Angular\Angular\RoutingDemo>
```



This interface can be implemented to guard a route and decide whether to activate the route or not. In case multiple guards are applied on the route, if all guards return true, then the route gets activated.
If it returns false, then the current ongoing navigation is cancelled. In some cases, we need to redirect from the canActivate method. That time you can return a URLTree from the canActivate implemented method.
Create route guard as below
ng generate guard authguard
To create guard in separate directory use command with extra parameter as below
ng generate guard <DirectoryName>/<GuardName>
ng generate guard Guards/Admin
ng generate component dashboard
by default guard returns true

```
import { CanActivateFn } from '@angular/router';
export const authguardGuard: CanActivateFn = (route, state) => {
  return true;
};
```

app-routing.module.ts

```
import { ProductsComponent } from './products/products.component';
import { ClientsComponent } from './clients/clients.component';
import { SearchComponent } from './search/search.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';
import { authguardGuard } from './authguard.guard';
```

```
const routes: Routes = [
  {path:'product/:id', component: ProductsComponent},
  {path:'product/:ProductID/Photos/:PhotoId', component:ProductsComponent},
  {
    path:'clients', component:ClientsComponent,
    canActivate:[authguardGuard]   // if it returns true then clients
                                   //component will be accessed else not
  },
  {
    path:'search',component:SearchComponent
  },
  {
    path:'**',
    component:PageNotFoundComponent
  }
];
```

Prior to angular 16 guards are created with class template and Angular 16 onwards they are not using class template. They are following functional way of implementation.

ng generate guard child --test-tests --functional=false.

Functional with false then it will be creating with class template and if without functional parameter it will be default function.

5. I will cover all of these in detail in coming episodes
   - quick examples
   - use cases

6. We can implement more than 1 guards in our application since it's an array.

Can Activate guard | Guards in angular | Angular 16
TechSharessk
https://www.youtube.com/watch?v=tFsCynatnlo

//login.component.html

```
<p>login works!</p>
<div>
   <input type="text" placeholder="Enter Name">
</div>
<div>
   <input type="password" placeholder="Enter password">
</div>
<div><button (click)="login()">Submit</button></div>
```

//login.component.ts

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
@Component({
  selector: 'app-login',
```

```
  templateUrl: './login.component.html',
  styleUrl: './login.component.css'
})
export class LoginComponent {
  constructor(private route:Router){
  }
  login(){
    localStorage.setItem('token',Math.random().toString());
    this.route.navigate(['/dashboard']);
  }
}
```

//dashboard.component.html

```
<p>dashboard works!</p>
<div><button (click)="Logout()">Logout</button></div>
```

//dashboard.component.ts

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrl: './dashboard.component.css'
})
export class DashboardComponent {
  constructor(private router:Router){

  }
  Logout(){
    const confirmation = confirm('Do you want to Logout?');
    if(confirmation){
      this.router.navigate(['login']);
      localStorage.clear();
  }
  }
}
```

//auth.guard.ts

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';

export const authGuard: CanActivateFn = (route, state) => {
  const token = localStorage.getItem('token');
  const router = inject(Router);
  if(token)
    {
      return true;
```

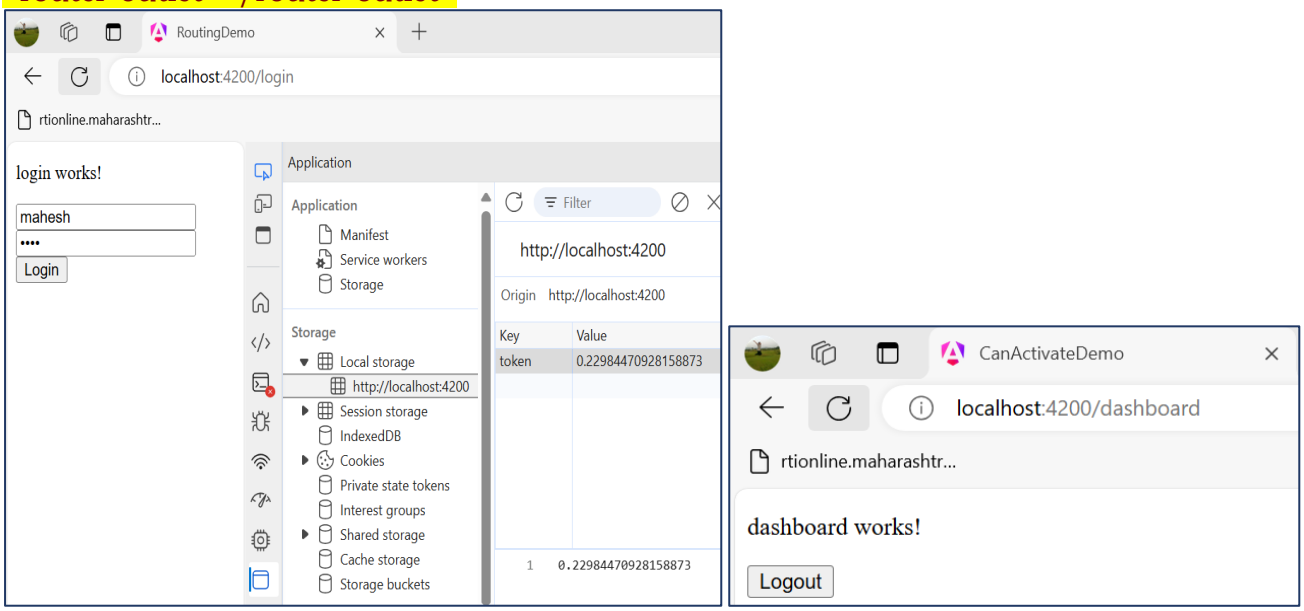
```
    }
    else{
      return false;
    }
};
```

//app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { LoginComponent } from './login/login.component';
import { DashboardComponent } from './dashboard/dashboard.component';
import { authGuard } from './guards/auth.guard';

const routes: Routes = [
  {
    path:'', component:LoginComponent
  },
  {path:'login', component:LoginComponent},
  {
    path:'dashboard',component:DashboardComponent,
    canActivate:[authGuard]
  }
];

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

//app.component.html

```
<!-- Primary Router Outlet -->
<router-outlet></router-outlet>
```



CanActivateChild:

- The **CanActivateChild** guard works similarly to the CanActivate guard, but the difference is its run before each child route is activated

The **CanActivateChild** route guard in Angular ensures that a user must meet certain criteria before accessing **child routes**. If the guard returns true, the child route is activated. If it returns false, the user is redirected.
Here's a **simple example** using Angular 14.

Step 1: Create a Child Route Guard

Generate the guard using the Angular CLI:
ng generate guard auth-child
This will create a new guard file, typically named auth-child.guard.ts.

Step 2: Implement the CanActivateChild Logic

Modify the generated guard to check for a simple token or permission. Here's an example:
auth-child.guard.ts

```
import { Injectable } from '@angular/core';
import { CanActivateChild, Router } from '@angular/router';

@Injectable({
  providedIn: 'root',
})
export class AuthChildGuard implements CanActivateChild {
  constructor(private router: Router) {}

  canActivateChild(): boolean {
    const token = localStorage.getItem('authToken'); // Check for token

    if (token) {
      return true; // Allow access to child routes
    } else {
      this.router.navigate(['/login']); // Redirect to login if not authenticated
      return false;
    }
  }
}
```

Step 3: Define Routes with CanActivateChild

Modify the app-routing.module.ts to add routes with child routes guarded by CanActivateChild.

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AuthChildGuard } from './auth-child.guard';
import { ParentComponent } from './parent/parent.component';
import { ChildComponent } from './parent/child/child.component';
import { LoginComponent } from './login/login.component';

const routes: Routes = [
  { path: 'login', component: LoginComponent },
  {
    path: 'parent',
    component: ParentComponent,
    canActivateChild: [AuthChildGuard], // Protect child routes
    children: [
```

```
      { path: 'child', component: ChildComponent }, // Child route
    ],
  },
  { path: '**', redirectTo: 'login' }, // Redirect unknown paths to login
];

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

Step 4: Create Components

Create the necessary components:

```
ng generate component parent
ng generate component parent/child
ng generate component login
```

Step 5: Add Navigation Links

Modify the **app.component.html** to add links to the parent and login routes:

app.component.html

```
<nav>
  <a routerLink="/parent/child">Go to Child Route</a> |
  <a routerLink="/login">Login</a>
</nav>

<router-outlet></router-outlet>
```

Step 6: Store Token on Login

Simulate storing a token in localStorage on a successful login:

login.component.ts

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css'],
})
export class LoginComponent {
  constructor(private router: Router) {}

  login() {
    localStorage.setItem('authToken', 'valid-token'); // Store token
    this.router.navigate(['/parent/child']); // Redirect to child route
  }
}
```

login.component.html

```
<h2>Login</h2>
<button (click)="login()">Login</button>
```

Step 7: Run the Application

Start the Angular server:

```
ng serve
```

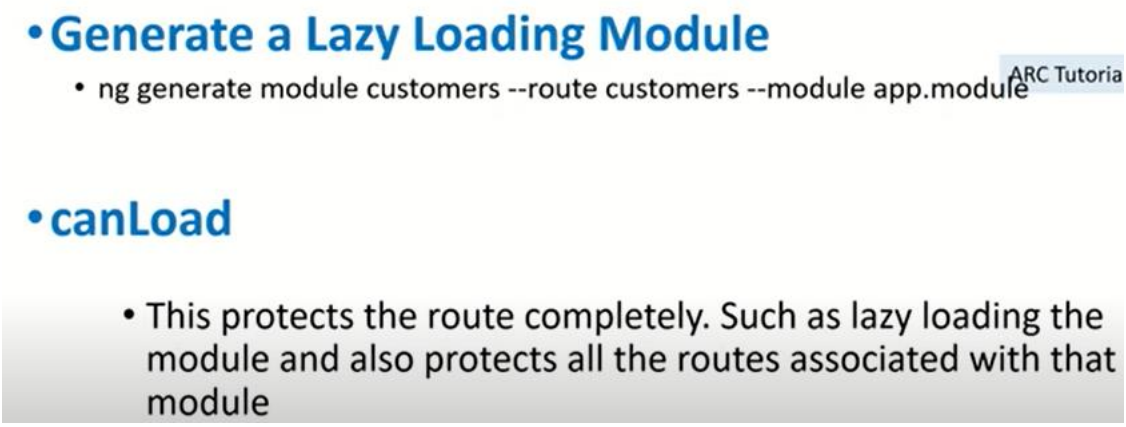
Explanation

1. **CanActivateChild Guard:**
   - The AuthChildGuard checks if the user has a token in localStorage.
   - If the token exists, access to the **child route** is granted.

- If not, the user is redirected to the **login page**.

2. **Child Route Protection:**
   - All child routes under **/parent** are protected by the CanActivateChild guard.
3. **Login Simulation:**
   - On clicking the **login button**, a token is stored, and the user is redirected to the **child route**.

CanLoad:



CanLoad is a route guard that prevents lazy-loaded modules from being loaded unless specific conditions are met. This improves performance and security by blocking module loading until the user meets the required criteria (like authentication).

Step-by-Step Guide to Using CanLoad Guard
Step 1: Create a CanLoad Guard
Generate a new guard using the Angular CLI:
ng generate guard auth-load
This will create a guard file (auth-load.guard.ts).
Step 2: Implement the CanLoad Guard Logic
Modify the generated guard file to check for an authentication token.
auth-load.guard.ts

```
import { Injectable } from '@angular/core';
import { CanLoad, Route, Router, UrlSegment } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class AuthLoadGuard implements CanLoad {
  constructor(private router: Router) {}

  canLoad(
    route: Route,
    segments: UrlSegment[]
  ): boolean | Observable<boolean> | Promise<boolean> {
    const token = localStorage.getItem('authToken'); // Check for a token

    if (token) {
      return true; // Allow loading of the module
    } else {
```

```
      this.router.navigate(['/login']); // Redirect to login if not authenticated
      return false;
    }
  }
}
```

Step 3: Define Lazy-Loaded Routes with CanLoad Guard

In your **app-routing.module.ts**, apply the CanLoad guard to protect a lazy-loaded module.

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AuthLoadGuard } from './auth-load.guard';
import { LoginComponent } from './login/login.component';

const routes: Routes = [
  { path: 'login', component: LoginComponent },
  {
    path: 'dashboard',
    loadChildren: () =>
      import('./dashboard/dashboard.module').then((m) => m.DashboardModule),
    canLoad: [AuthLoadGuard], // Protect the lazy-loaded module with CanLoad
  },
  { path: '**', redirectTo: 'login' }, // Redirect unknown paths to login
];

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

Step 4: Create the Lazy-Loaded Dashboard Module

Generate the **DashboardModule** with the CLI:

```
ng generate module dashboard --route dashboard --module app-routing.module
```

This will create the **DashboardModule** and set up the lazy-loaded route automatically.

Step 5: Add Login and Dashboard Components

1. **Login Component**: Store a token on login.

login.component.ts

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css'],
})
export class LoginComponent {
  constructor(private router: Router) {}

  login() {
    localStorage.setItem('authToken', 'valid-token'); // Store token
    this.router.navigate(['/dashboard']); // Redirect to dashboard
  }
}
```

login.component.html

```
<h2>Login</h2>
<button (click)="login()">Login</button>
```

2. **Dashboard Component**: Display a message.

dashboard.component.html

```
<h2>Dashboard</h2>
<p>Welcome to the dashboard!</p>
```

Step 6: Run the Application

Start the Angular server:

```
ng serve
```

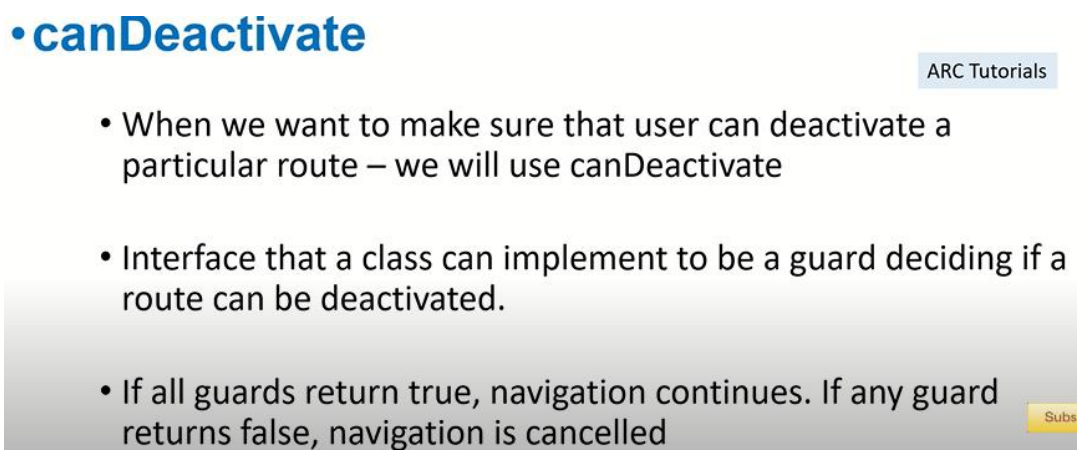
How It Works

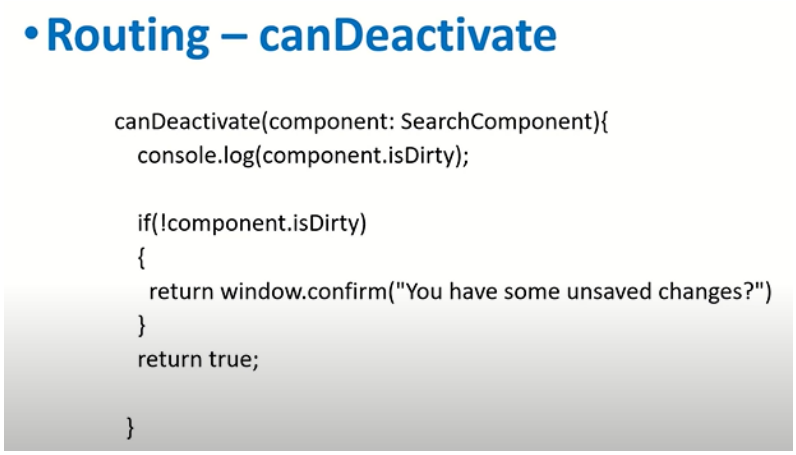
1. **Lazy-Loaded Module Protection**:
   - The **DashboardModule** will only load if the **CanLoad guard** returns true.
   - If the user is not authenticated, they are **redirected to the login page**.
2. **Login Simulation**:
   - When the user logs in, a token is stored in **localStorage**.
   - After login, the user is redirected to the **dashboard** route.
3. **Prevent Unnecessary Loading**:
   - If the guard returns false, the module is not loaded, saving resources.

Why Use CanLoad?

- **Performance**: Prevents unnecessary loading of large modules.
- **Security**: Ensures the user meets certain conditions before loading a module.
- **Optimization**: Useful when only authenticated users should access certain parts of the app.

CanDeactivate:





The **CanDeactivate** guard in Angular is used to **prevent the user from navigating away** from a route. It is useful when you want to **warn users about unsaved changes** or confirm navigation when leaving a form.
Here's a simple example of using **CanDeactivate** guard in Angular 14.

Step-by-Step Example of CanDeactivate Guard
Step 1: Create a CanDeactivate Guard
Generate the guard using Angular CLI:
bash
Copy code
ng generate guard can-deactivate
Modify the guard to implement the **CanDeactivate** logic.
can-deactivate.guard.ts
typescript
Copy code

```
import { Injectable } from '@angular/core';
import { CanDeactivate } from '@angular/router';
import { Observable } from 'rxjs';

// Interface to define the expected behavior of guarded components
export interface CanComponentDeactivate {
  canDeactivate: () => boolean | Observable<boolean>;
}

@Injectable({
  providedIn: 'root',
})
export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {
  canDeactivate(
    component: CanComponentDeactivate
  ): boolean | Observable<boolean> {
    // Call the canDeactivate method from the component
    return component.canDeactivate
      ? component.canDeactivate()
      : true; // Default to true if not implemented
  }
}
```

This guard checks if the component implements the **canDeactivate()** method, which determines whether the user can leave the page.

Step 2: Modify a Component with canDeactivate() Method
In this example, we'll apply the **CanDeactivate** guard to a form component to **warn users about unsaved changes**.
form.component.ts
typescript
Copy code

```
import { Component } from '@angular/core';
import { CanComponentDeactivate } from '../can-deactivate.guard';

@Component({
  selector: 'app-form',
  templateUrl: './form.component.html',
  styleUrls: ['./form.component.css'],
})
export class FormComponent implements CanComponentDeactivate {
```

```
  isFormDirty: boolean = false; // Track if the form is modified

  // This method will be called by the CanDeactivate guard
  canDeactivate(): boolean {
    if (this.isFormDirty) {
      return confirm('You have unsaved changes. Do you really want to leave?');
    }
    return true;
  }

  // Simulate form input modification
  markAsDirty() {
    this.isFormDirty = true;
  }
}
```

Here, the canDeactivate() method asks the user for confirmation if the form is dirty (modified).

Step 3: Define the Route with CanDeactivate Guard

Modify the **app-routing.module.ts** to use the CanDeactivate guard on the route.

app-routing.module.ts

typescript

Copy code

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { FormComponent } from './form/form.component';
import { CanDeactivateGuard } from './can-deactivate.guard';

const routes: Routes = [
  {
    path: 'form',
    component: FormComponent,
    canDeactivate: [CanDeactivateGuard], // Apply the CanDeactivate guard
  },
  { path: '**', redirectTo: 'form' }, // Redirect unknown paths to form
];

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

Step 4: Create the Form Component Template

Create a form in the **form.component.html** template:

form.component.html

html

Copy code

```
<h2>Form Component</h2>
<input type="text" (input)="markAsDirty()" placeholder="Enter something..." />
<p>If you try to leave, you will be prompted if the form has unsaved changes.</p>
```

This form triggers markAsDirty() when the user types something, marking the form as modified.

Step 5: Add Navigation Links

Modify **app.component.html** to add a navigation link to the form:
app.component.html
html
Copy code
```
<nav>
  <a routerLink="/form">Go to Form</a>
</nav>

<router-outlet></router-outlet>
```

Step 6: Run the Application
Start the Angular server:
bash
Copy code
```
ng serve
```

How It Works

1. When the user **navigates to /form**, they can fill in the form.
2. If the user tries to **navigate away** without saving (after modifying the input), the **CanDeactivate** guard will call the canDeactivate() method.
3. The guard will **prompt the user** with a confirmation dialog:
    - If the user confirms, they are allowed to navigate away.
    - If the user cancels, they stay on the form page.

Why Use CanDeactivate?

- **Prevent Data Loss:** Warn users about unsaved changes in forms.
- **User Confirmation:** Ask for confirmation before navigating away.
- **Improve UX:** Ensure users don't accidentally lose their progress.

This simple example demonstrates how to implement and use a **CanDeactivate guard** in Angular 14 to enhance navigation control.

Bonus Point: Create theme toggle between dark and light

Step 1. Create sample application as below with name **ThemeToggleDemo**:
ng new ThemeToggleDemo --standalone=false

Step 2. Install angular material :
ng add @angular/material =➔ this will give theming options, if issue comes with this try using install and then use add command. If material UI is already installed then it will ask for theming option.
Enable typography
Enable animation and activate
ng install @angular/material

Step 3. Run the application: ng serve –o
Step 4. Go to https://material.angular.io/ ➔ Components

Import below things as per the requirements
//Import
import {MatToolbarModule} from '@angular/material/toolbar';
import {MatSlideToggleModule} from '@angular/material/slide-toggle';
import {MatCardModule} from '@angular/material/card';
import {MatIconModule} from '@angular/material/icon';
and import into @ngModule

Step 5. Add some angular material components and customize the dark mode theme.
Make some changes in the style.scss
Added below code and make enable dark them whenever dark theme enabled.

```
$ThemeToggleDemo-Dark-theme: mat.define-Dark-theme((
  color: (
    theme-type: light,
    primary: mat.$azure-palette,
    tertiary: mat.$blue-palette,
  ),
  density: (
    scale: 0,
  )
));

.dark-them{
  @include mat.all-component-themes($ThemeToggleDemo-Dark-theme);
}

//App.component.html
<div [ngClass]="isDarkTheme? 'dark-theme' : 'light-theme'">
<p>
<mat-toolbar>
  <button mat-icon-button class="example-icon" aria-label="Example icon-button with
menu icon">
    <mat-icon>menu</mat-icon>
  </button>
```

```
  <span>My App</span>
  <span class="example-spacer"></span>
  <button mat-icon-button class="example-icon favorite-icon" aria-label="Example icon-button with heart icon">
    <mat-icon>favorite</mat-icon>
  </button>
  <button mat-icon-button class="example-icon" aria-label="Example icon-button with share icon">
    <mat-icon>share</mat-icon>
  </button>
</mat-toolbar>
</p>
<mat-card>
  <mat-card-content>
    <p><mat-slide-toggle>Slide me!</mat-slide-toggle></p>
<p><mat-slide-toggle labelPosition="before">...and slide me too!</mat-slide-toggle></p>
  </mat-card-content>
</mat-card>
<router-outlet></router-outlet>
</div>

//app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss'
})
export class AppComponent {
  title = 'ThemeToggleDemo';
  //Usually user settings/preferences API-Which will
  //fetch the saved settings.
  //User has saved DarkThem is ture else it is false
  isDarkTheme=true;
}
```

Angular 17
https://www.youtube.com/watch?v=hejR2GfFXiA&t=1160s
Theme : https://themewagon.github.io
API: https://freeapi.miniprojectideas.com/index.html

Angular 16
https://www.youtube.com/watch?v=qQOkMB44nyg

https://github.com/lionashu/AngRestaurantApp.git

Working with Forms in Angular:

Complete guide related to forms in angular
https://www.tektutorialshub.com/angular/angular-forms-fundamentals/

Forms in Angular are a way for users to input data into a web application. They are a key part of many applications, allowing users to log in, update their profile, and enter sensitive information.

Forms are an integral part of a web application. Practically every application comes with forms to be filled in by the users. Angular forms are used to log in, update a profile, enter sensitive information, and perform many other data-entry tasks. In this article, you will learn about how to create a form and validate the information filled.

Angular offers two approaches to handling user input through forms:

- Reactive: This approach uses an immutable data structure, meaning that when the form changes, it returns a new state instead of updating the existing data model.
- Template-driven: This approach uses the mutable ngModel approach.

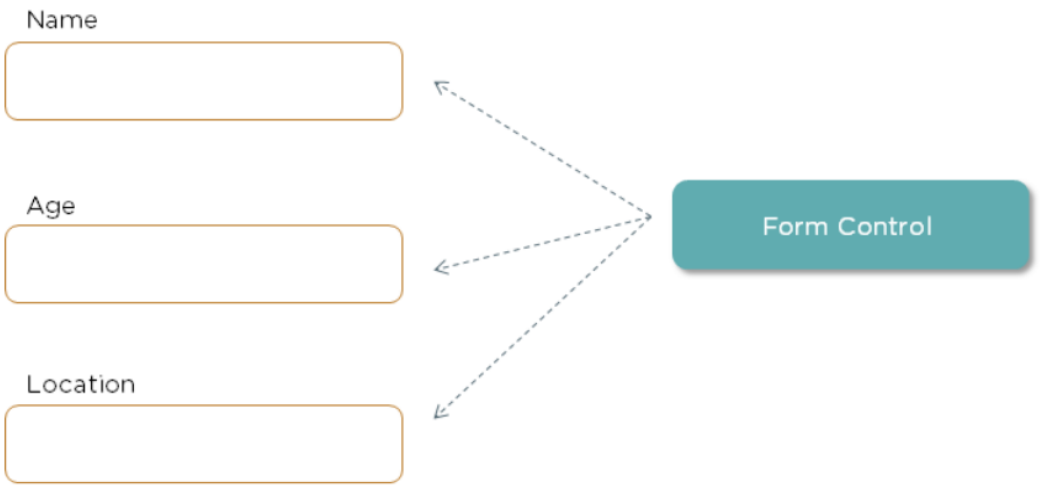
Template-Driven Approach

- In this method, the conventional form tag is used to create forms. Angular automatically interprets and creates a form object representation for the tag.
- Controls can be added to the form using the NGModel tag. Multiple controls can be grouped using the NGControlGroup module.
- A form value can be generated using the “form.value” object. Form data is exported as JSON values when the submit method is called.
- Basic HTML validations can be used to validate the form fields. In the case of custom validations, directives can be used.
- Arguably, this method is the simplest way to create an Angular App.

Reactive Form Approach

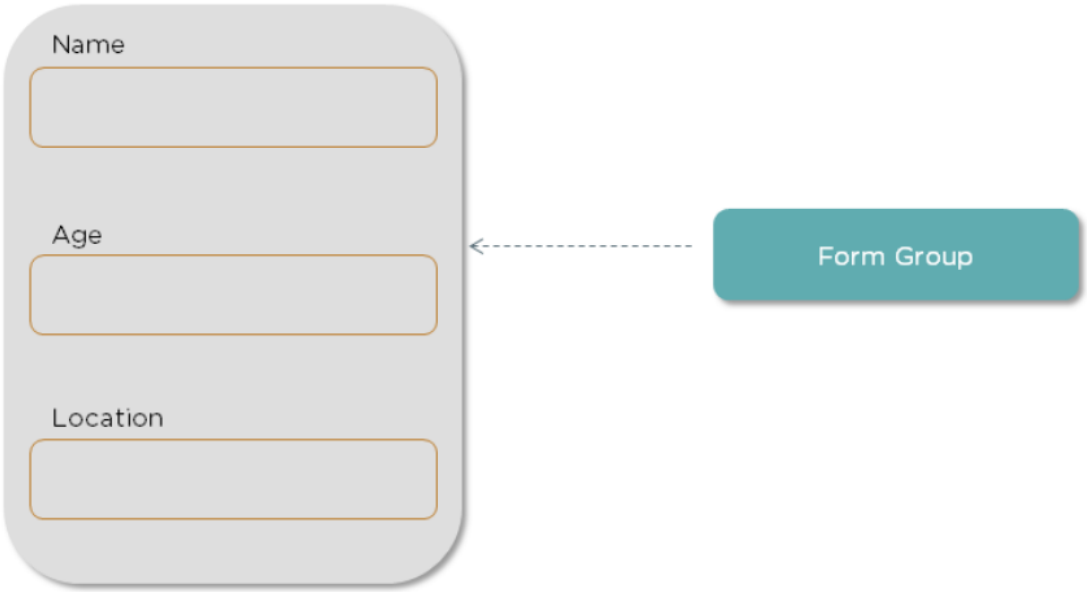
- This approach is the programming paradigm oriented around data flows and propagation of change.
- With Reactive forms, the component directly manages the data flows between the form controls and the data models.
- Reactive forms are code-driven, unlike the template-driven approach.
- Reactive forms break from the traditional declarative approach.
- Reactive forms eliminate the anti-pattern of updating the data model via two-way data binding.
- Typically, Reactive form control creation is synchronous and can be unit tested with synchronous programming techniques.

Form Control



Form Control is a class that enables validation. For each input field, an instance of this class is created. These instances help check the values of the field and see if they are touched, untouched, dirty, pristine, valid, invalid, and so on.

Form Group



FormGroup class represents a group of controls. A form can have multiple control groups. The Form Group class returns true if all the controls are valid and also provides validation errors, if any.

Example:

In this demo, we'll be creating a User Registration form consisting of four fields, viz, Firstname, Lastname, Email ID, and Password.

But before creating a form, create a component using the Angular CLI and provide the name of your choice. In this case, we've created a component called "formComponent." In the template-driven approach, the form's module needs to be imported into the app.module.ts file. Go ahead and import it.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
```

```
import { AppComponent } from './app.component';
import { NewComponentComponent } from './components/new-component/new-component.component';
import { FormComponentComponent } from './form-component/form-component.component';
@NgModule({
  declarations: [
    AppComponent,
    NewComponentComponent,
    FormComponentComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

We've highlighted the code for better visibility.
Once created, follow the step-by-step instructions to create your form.

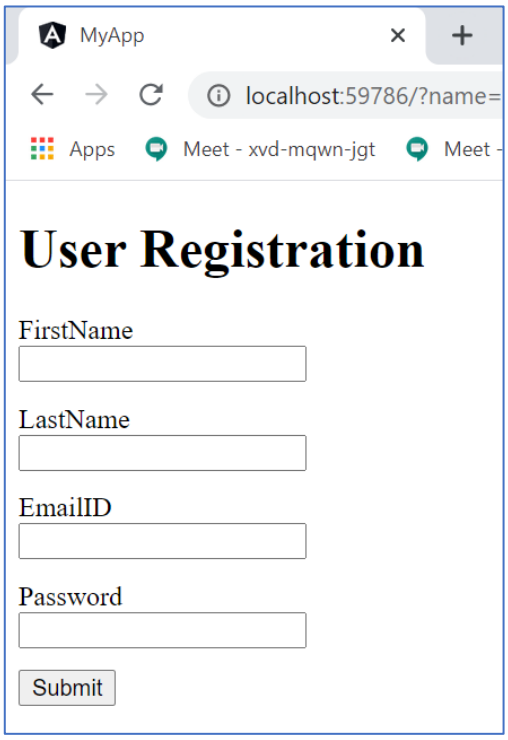
Step1: Form Creation

Create a form tag within a div tag to create the four fields.

```
<div class="container">
    <h1>User Registration</h1>
    <form>
        <div class="form-group">
            <label for="firstname">FirstName</label><br>
            <input type="text" name="firstname" class = "form-control" ngModel>
        </div>
        <pre></pre>
        <div class="form-group">
            <label for="lastname">LastName</label><br>
            <input type="text" name="lastname" class = "form-control"    ngModel>
        </div>
        <pre></pre>
        <div class="form-group">
            <label for="email">EmailID</label><br>
            <input type="text" name="email" class = "form-control" ngModel>
        </div>
        <pre></pre>
        <div class="form-group">
            <label for="password">Password</label><br>
            <input type="password" name="password" class = "form-control" ngModel>
        </div>
```

```
        <pre></pre>
        <button class="btn btn-primary" type="submit">Submit</button>
    </form>
</div>
```

Once served, the browser looks like this:



Install and use Bootstrap in Angular:

npm install bootstrap

To verify installation check package.json

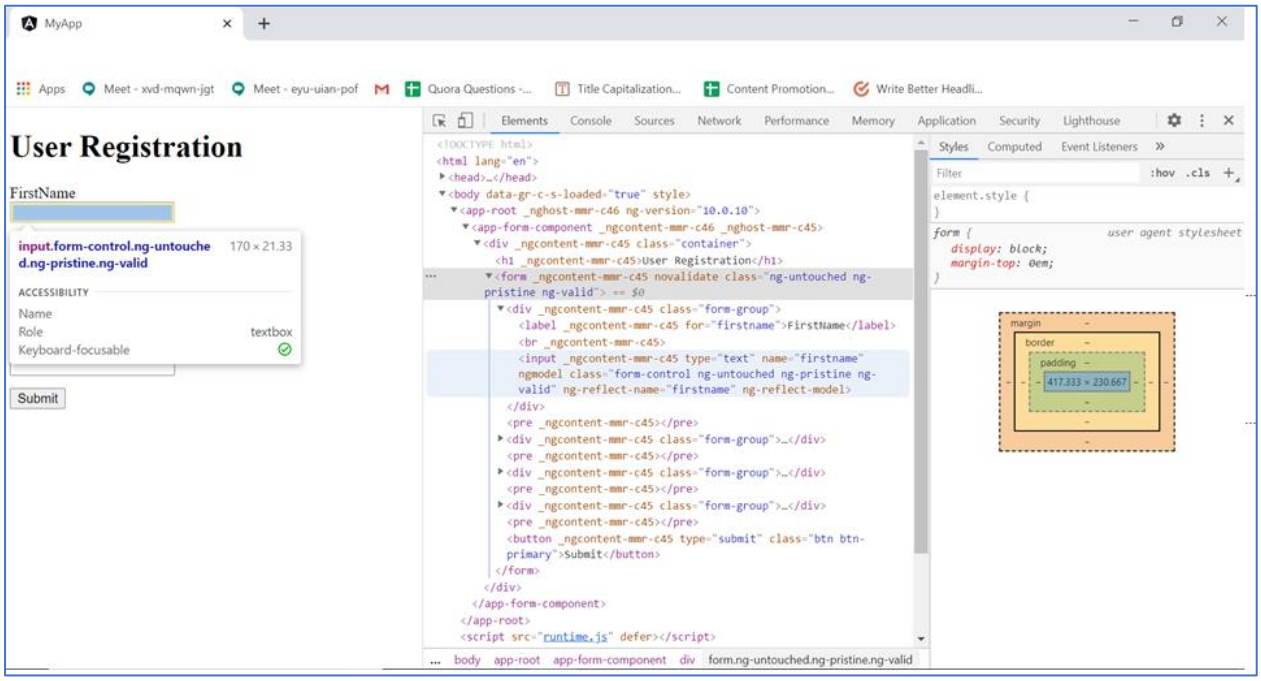
To use, it must be imported into main style from the following path

Import it into style.css as

@import url('../node_modules/bootstrap/dist/css/bootstrap.min.css');

Step2: Adding Angular Form Controls

When you inspect the form, classes like ng-untouched, ng-pristine, and ng-valid are added. This indicates that Angular has recognized the form tag and has added the classes to the form as well as the fields. By adding the ngModel directive in the input tag, form controls are added to every input field.



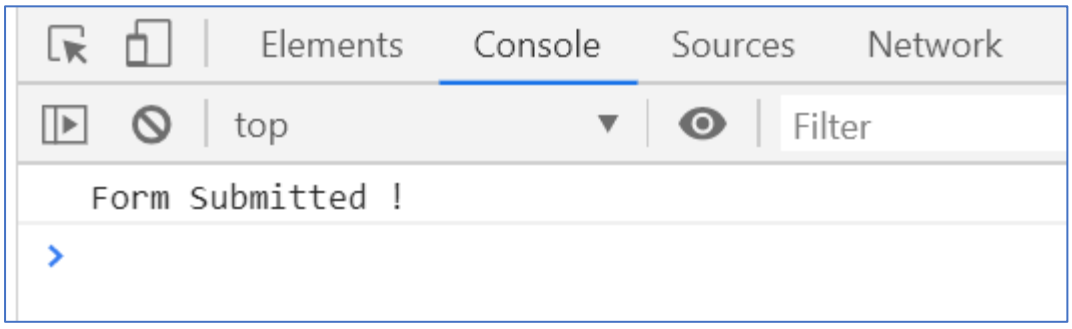
The form has an output property attached to it called **ngSubmit**. This property can be bound with a method called **"Submit()."** Once submitted, this method is called.

```
<form (ngSubmit)="submit()">
```

The submit method can be defined in the component,
i.e., form-component.component.ts file.

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-form-component',
  templateUrl: './form-component.component.html',
  styleUrls: ['./form-component.component.css']
})
export class FormComponentComponent {
  submit(){
    console.log("Form Submitted !")
  }
}
```

The output is seen on the console:



Step3: Getting the JavaScript Object Representation

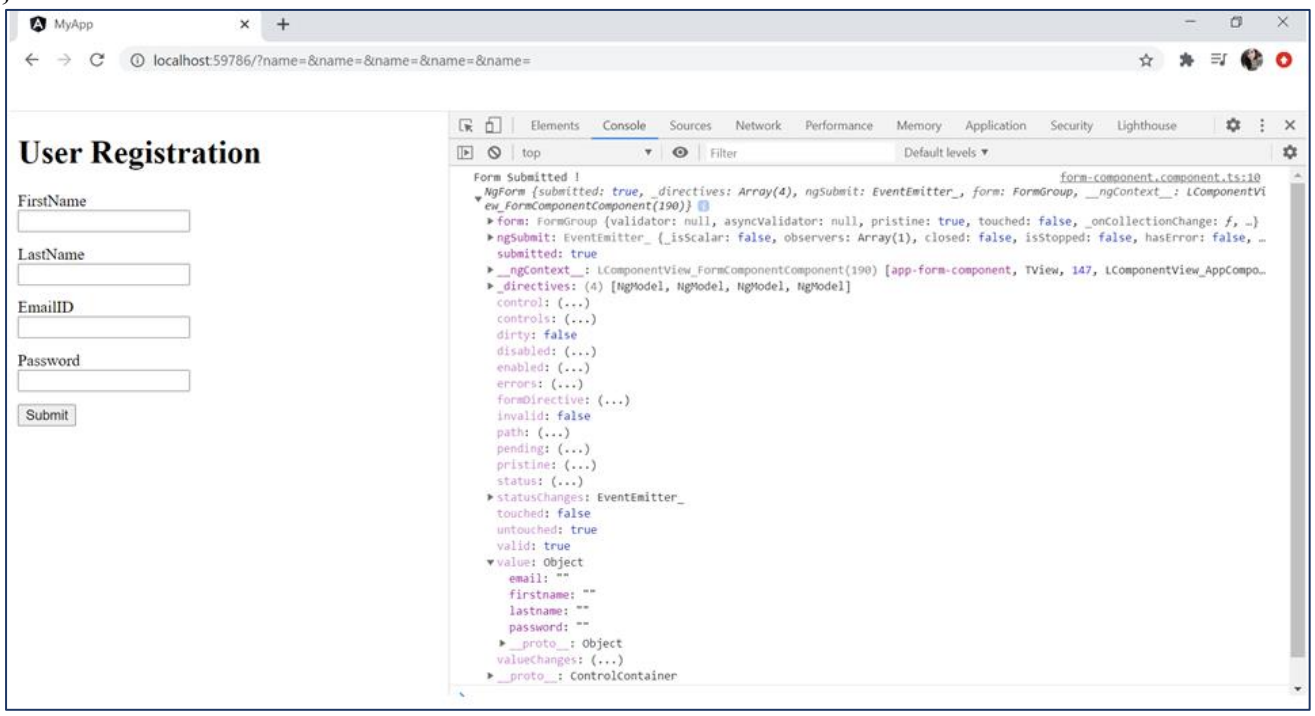
To generate the JavaScript Representation, another directive called **NgForm** is assigned to a template variable.

```
<form #login = "ngForm" (ngSubmit)="submit(login)">
```

Consequently, update the **submit()** method in the .ts file as well.

```
export class FormComponentComponent {
```

```
  submit(login){
    console.log("Form Submitted !",login);
  }
}
```



Once submitted, the message is printed and along with it, the Ngform object is obtained indicating the JavaScript Representation of the form. Getting this representation makes form validation easy.

If you observe closely, the value object includes the form controls for the input fields.

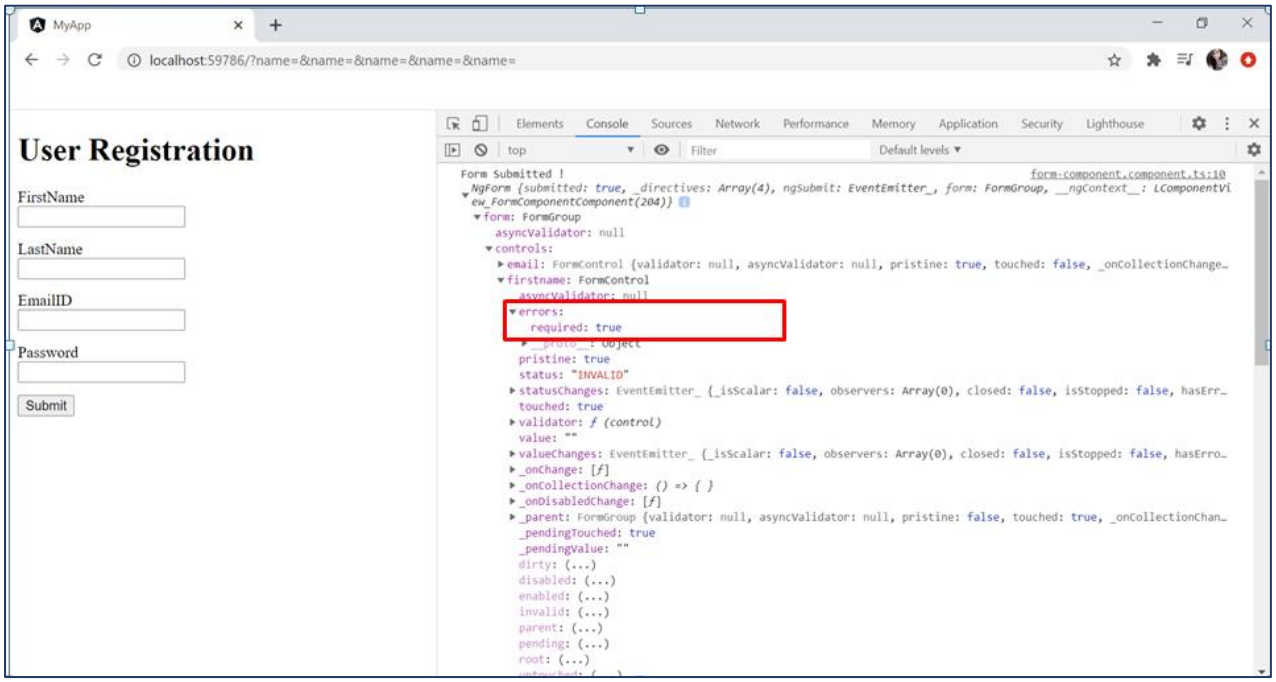
Step 4: Angular Form Validation

One way to ensure that all the fields are filled correctly is by disabling the submit button in case the fields aren't filled. Apart from this, you can also specify certain properties in your input tag for the corresponding input field.

Let's say, that the fields can't be empty and the form only accepts names with a minimum length of 2 and a maximum length of 7.

<input required minlength="2" maxlength="7" type="text" name="firstname" class = "form-control" ngModel>

To check this, we've submitted the form without filling the FirstName field and evidently, an error can be seen.



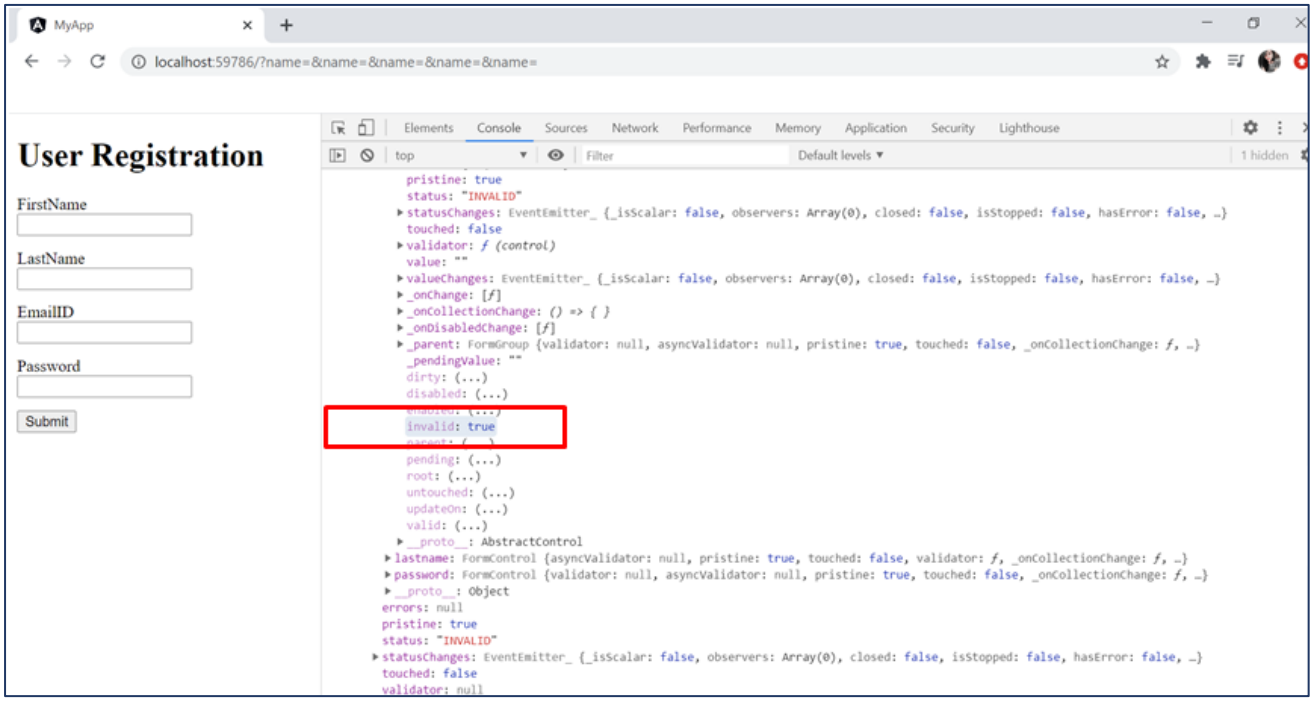
So you can leverage the form control objects to ensure field validation and display a message when an error occurs.

To do that, you need to access the form control objects and for that, create a template variable for the field and assign it to the form control object.

<input required minlength="2" maxlength="7" type="text" name="firstname" class = "form-control" #name="ngModel" ngModel>

Here, the variable name receives the form control object.

When submitted without filling the FirstName field, the invalid property is set to true.

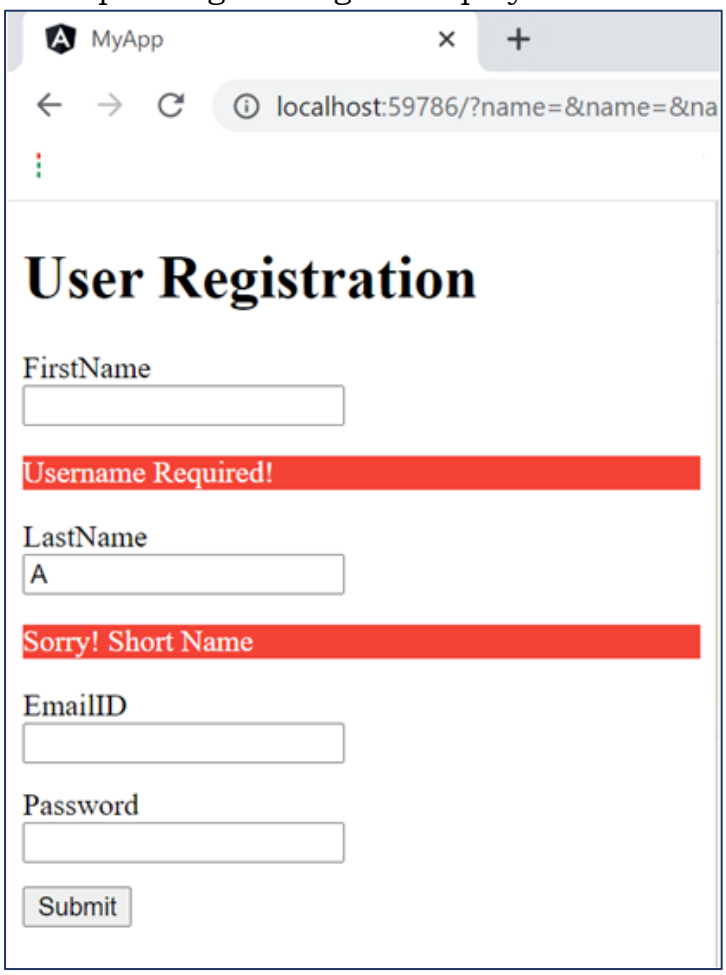


This helps us use this property to alert the user with the help of the simple 'if' logic.

```
<div class="form-group">
        <label for="firstname">FirstName</label><br>
```

```
        <input required minlength="2" maxlength="7" type="text" name="firstname"
class = "form-control" #name="ngModel" ngModel>
        <div *ngIf="name.touched && name.invalid" class="alert alert-danger">
            <p *ngIf="name.errors?.['required']">Enter Username</p>
            <p *ngIf="name.errors?.['minlength']">Sorry! Short Username</p>
        </div>
    </div>
```

The ngIf directive is used to check for the properties. Depending on the error, the corresponding message is displayed.

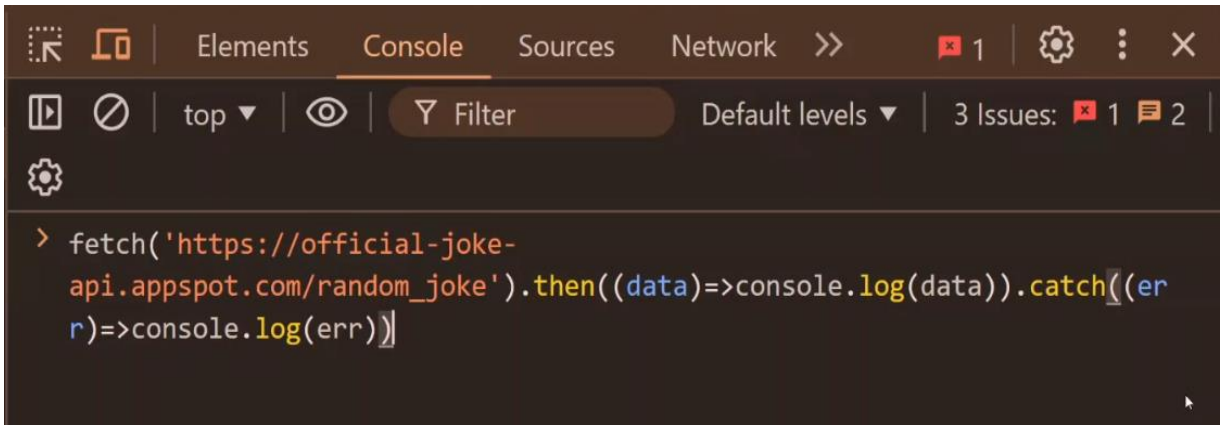


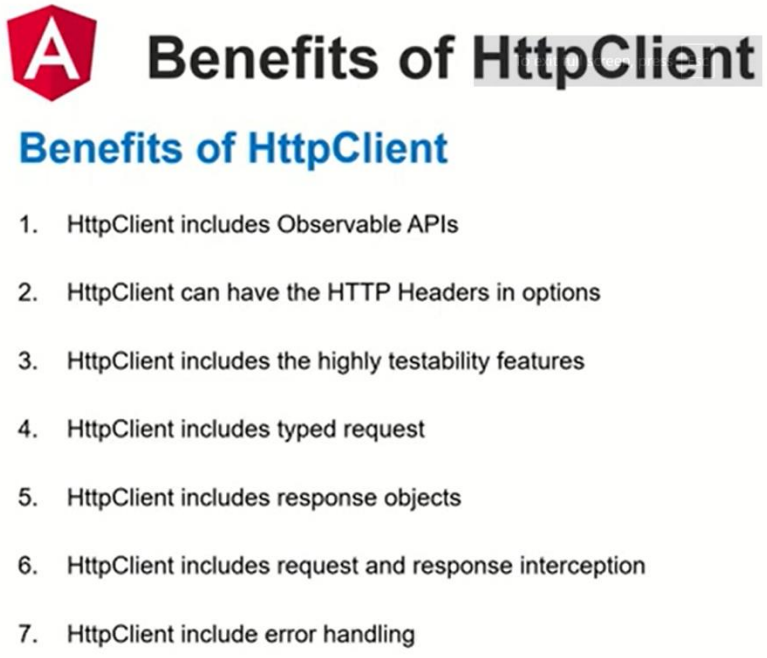
HttpClient in Angular:
The HttpClient in Angular is a service that enables your Angular application to communicate with backend services API's/REST API's over HTTP. It is part of the **@angular/common/http** package and allows making HTTP requests, such as GET, POST, PUT, DELETE, and others, to interact with REST APIs or any web service.

Standard Http Methods:

Name	**Description**
get()	Request resource from API
post()	Send date to API
put()	Update an existing resource
delete()	Delete the resource using API
head()	The HTTP HEAD method simply returns metadata about a resource on the server. This HTTP request method returns all of the headers associated with a resource at a given URL, but does not actually return the resource. The HTTP HEAD method is commonly used to check the following conditions: • The size of a resource on the server. • If a resource exists on the server or not. • The last-modified date of a resource. • Validity of a cached resource on the server. The following example shows sample data returned from a HEAD request: HTTP/1.1 200 OK Date: Fri, 19 Aug 2023 12:00:00 GMT Content-Type: text/html Content-Length: 1234 Last-Modified: Thu, 18 Aug 2023 15:30:00 GMT
jsonp()	JSONP allows you to request data from different domains by embedding a <script> tag in the page. **What is JSONP?** JSONP is a way to request JSON data from a server in a way that bypasses the restrictions of cross-origin HTTP requests (CORS). It works by making a request to a remote server, which returns a JavaScript function call containing the data as an argument. The browser can load data from a different origin because <script> tags are not restricted by the same-origin policy. **-Only works for GET requests:** JSONP can only be used with GET requests. You cannot send POST, PUT, or DELETE requests using JSONP. **-Security concerns:** JSONP is generally less secure than using CORS because it allows executing scripts from other domains, which may introduce cross-site scripting (XSS) vulnerabilities if not handled carefully.
options()	The server does not have to support every HTTP method for every resource it manages. Some resources support the PUT and POST operations. Other resources only support GET operations.

	The HTTP OPTIONS method returns a listing of which HTTP methods are supported and allowed. The following is a sample response to an HTTP OPTIONS method call to a server: OPTIONS /example/resource HTTP/1.1 Host: www.example.com HTTP/1.1 200 OK Allow: GET, POST, DELETE, HEAD, OPTIONS Access-Control-Allow-Origin: * Access-Control-Allow-Methods: GET, POST, DELETE, OPTIONS Access-Control-Allow-Headers: Authorization, Content-Type
patch()	Sometimes object representations get very large. The requirement for a PUT operation to always send a complete resource representation to the server is wasteful if only a small change is needed to a large resource. The PATH HTTP method, added to the Hypertext Transfer Protocol independently as part of RFC 5789, allows for updates of existing resources. It is significantly more efficient, for example, to send a small payload rather than a complete resource representation to the server.
request()	





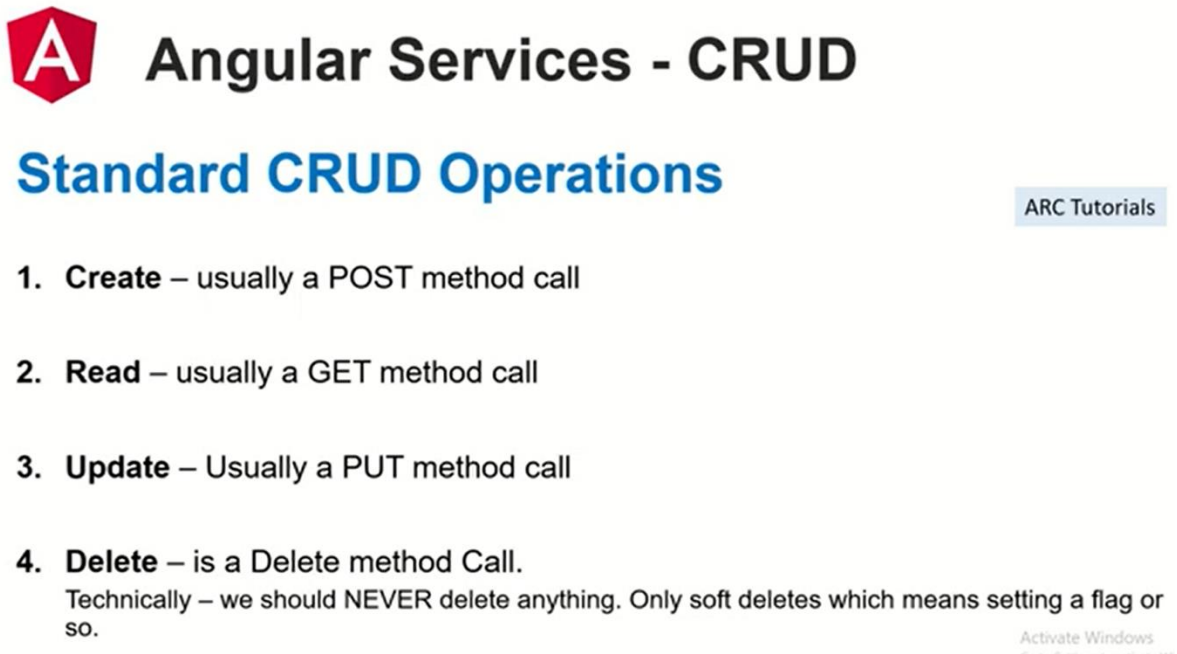
To perform various operations, we need

1. Import the httpClient Module in app-Module.
2. We need to inject httpClient in components/services and create its instance in constructor
3. Mostly we perform CRUD operatioins

httpClinet allows us for sending/configuring the headers that are sent.

httpClinet helps us in creating Interceptors for API requests.

httpClinet can be used for common error handling.



HttpClient Get Method : we can perform below operations using Get methods

- Simple HTTP Get method call
- Get method call with headers
- Get method call with Params
- Get method call with Type casting
- Subscribe and catch Error Handling

Recap Steps :

Import httpclientmodul into app-model
Import it into @import section of app-model
Import httpClient into service
Inject the httpClient into contructor creating instance of httpClient

Steps :

1. **Created new app : HttpClientDemo**
   ng new HttpClientDemo --standalone=false
2. **Create new Service named as User**
   ng generate service Users
3. **Create new Component Named as Customers**
   ng generate component Customers

Import in app.models.ts

```
import { HttpClientModule } from '@angular/common/http';
imports: [
    BrowserModule,
    HttpClientModule,
    AppRoutingModule
  ]
```

4. **Inside Users Service :** create Endpoints (backend team) API's, we used fake api in below example. Used Fake Api from https://jsonplaceholder.typicode.com/

Case 1: Static Array and API get request

```
//users.service.ts
import { Injectable } from '@angular/core';
//import HttpClient from @angular/common/http
import{HttpClient} from '@angular/common/http';
@Injectable({
  providedIn: 'root'
})
export class UsersService {
  Users:any;
//create instance of HttpClient
  constructor(private http:HttpClient) {
  }
  getStaticUsers(){
this.Users=[
      {userID:10,userName:'Youtube'},
      {userID:20,userName:'Twitter'},
```

```
      {userID:30,userName:'Facebook'},
      {userID:40,userName:'Instagram'},
      {userID:50,userName:'LinkedIn'}
    ];
    //console.log(this.Users);
    return this.Users;
  }
  getAPIUsers(){
    //we need to pass the url into get()
    return this.http.get("https://jsonplaceholder.typicode.com/users");
  }
}
```

customers.component.ts

```
import { Component } from '@angular/core';
//Service Imported to the component
import { UsersService } from '../users.service';

@Component({
  selector: 'app-customers',
  templateUrl: './customers.component.html',
  styleUrl: './customers.component.css'
})
export class CustomersComponent {
staticUsers:any;
APIUsers:any;
constructor(private userService : UsersService){
}
ngOnInit(){
  //calling API method need to subscribe
  this.userService.getAPIUsers().subscribe(data =>{
    this.APIUsers=data;
  });
  this.staticUsers=this.userService.getStaticUsers();
}
}
```

customers.component.css

```
table {
    width: max-content;
    border-collapse: collapse; /* Ensures that the borders don't double up */
  }

  th, td {
    border: 3px solid rgb(8, 243, 35); /* Adds border to table cells */
    padding: 8px;            /* Adds some padding inside the cells */
    text-align: left;        /* Aligns text to the left */
  }
```

```
 thead {
   background-color: #72a905; /* Adds a light background to the header */
 }
```

customers.component.html

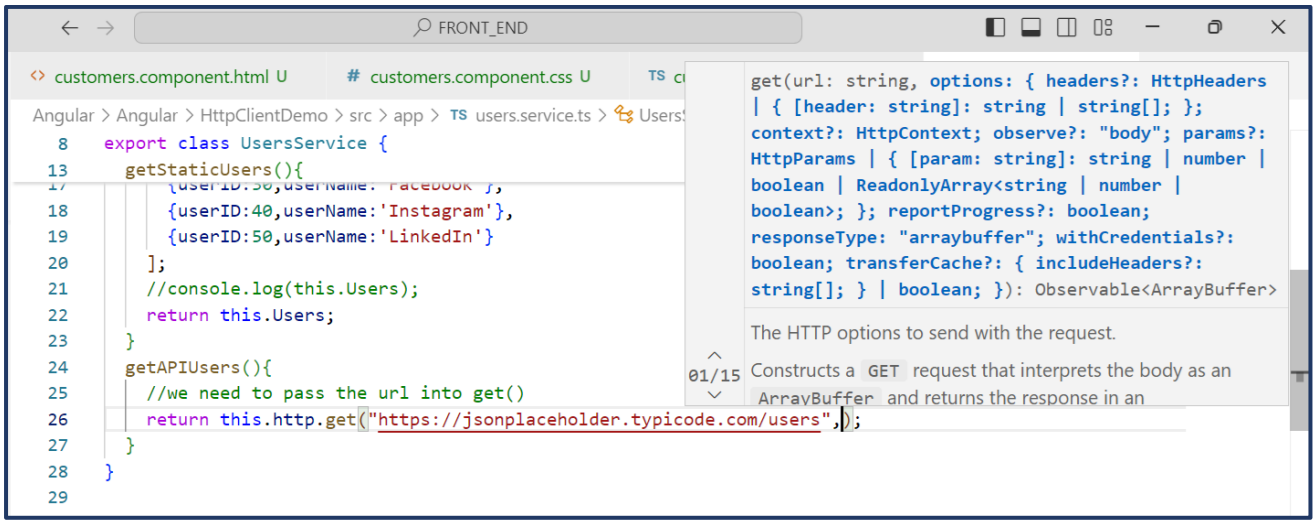
```
<h2>Data from Static Constant service Variable</h2>
<div *ngFor="let c of staticUsers">
   <p>{{c.userID}} {{c.userName}}</p>
</div>
<hr>
<h2>Data from API included in service</h2>
<table>
      <thead>
         <tr>
         <th>Name</th>
         <th>Username</th>
         <th>email</th>
         </tr>
      </thead>
<tbody *ngFor="let c of APIUsers">
   <tr>
      <td>{{c.name}}</td><td>{{c.username}}</td><td>{{c.email}}</td>
   </tr>
</tbody>
</table>
```

app.component.html

```
<app-customers></app-customers>
```

Case 2: Appending header information

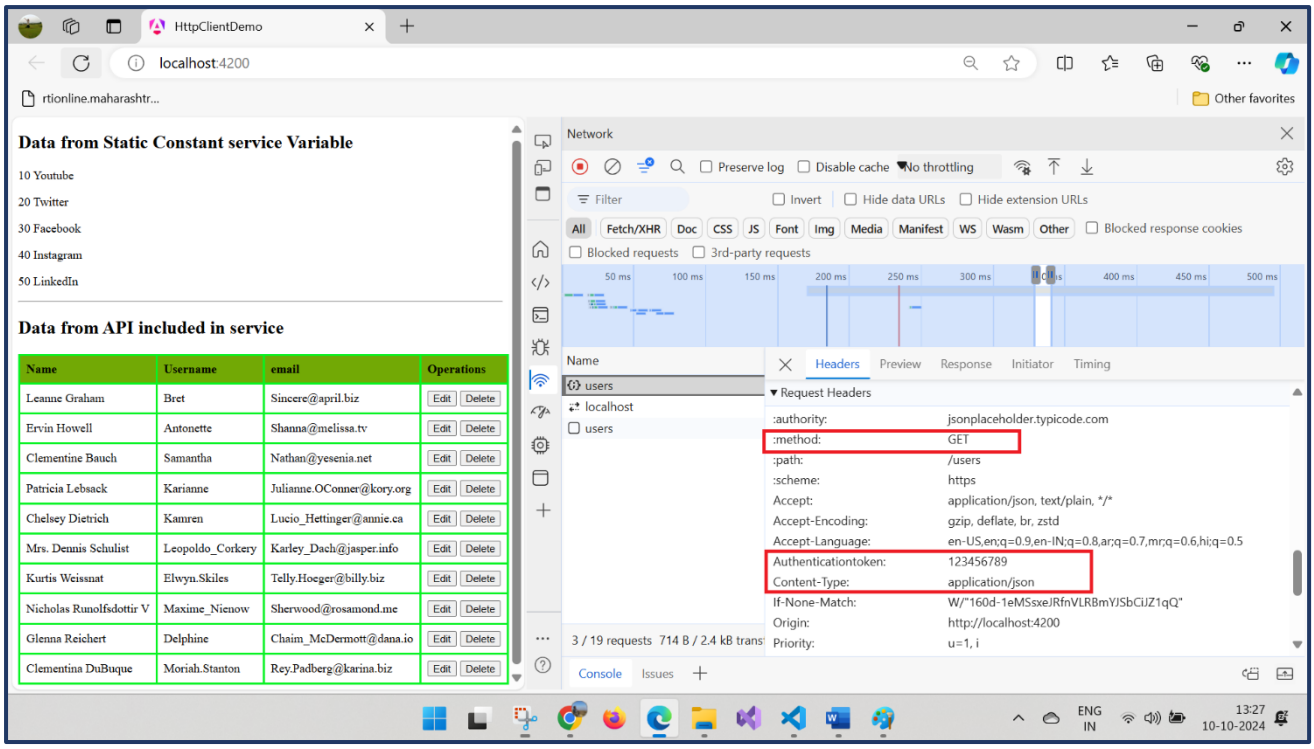
We can check various parameters can be passed to the get()



Update users.service.ts with following code:

```
getAPIUsers(){
   //we need to pass the url into get() URL paramter is mandatory other are optional
   const _headers = new HttpHeaders({
     'content-type':'application/json',
     'authenticationToken':'123456789'
   });
   return this.http.get("https://jsonplaceholder.typicode.com/users",{headers :_headers});
 }
```

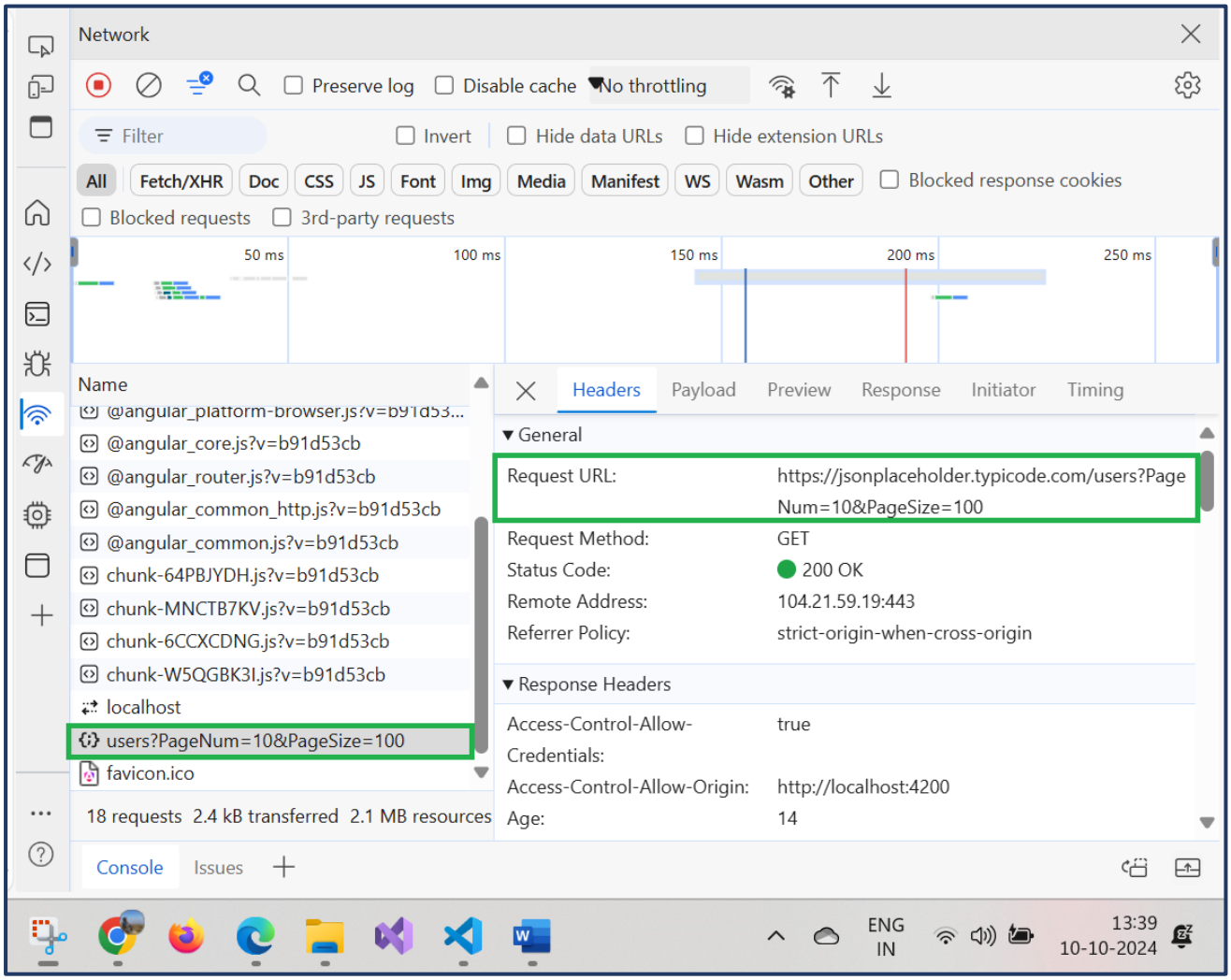
As we can see in the below screenshot method is GET and content-type and Authenticatintoken values in browser.



Case 3 : Setting parameters to get methods

Modify getAPIUsers() in following way

```
 getAPIUsers(){
   //we need to pass the url into get() URL paramter is mandatory other are optional
   //set Http headers using following way
   const _headers = new HttpHeaders({
     'content-type':'application/json',
     'authenticationToken':'123456789'
   });
   //set Http Parameters using following way
   const _params = new HttpParams()
   .set('PageNum','10')
   .set('PageSize','100');
   return this.http.get("https://jsonplaceholder.typicode.com/users",{headers :_headers,
params:_params});
  } }
```



Note : By default all the http returns observables of type “Object” if we don’t typecast it.

Case 4: how to typecast returned objects from http

Create Interface of required type and typecast object into required way as mentioned below.

customers.component.ts

```
ngOnInit(){
  //calling API method need to subscribe
  this.userService.getAPIUsers().subscribe(data =>{
    this.APIUsers=data;
    //we can now get individual data that we typecasted it
    console.log(data.email + data.name + data.phonenumber);
  });
  this.staticUsers=this.userService.getStaticUsers();
```

users.service.ts

```
//users.service.ts
import { Injectable } from '@angular/core';
//import HttpClient from @angular/common/http
import{HttpClient, HttpHeaders, HttpParams} from '@angular/common/http';
import { Observable } from 'rxjs';

//Created interface for making type
interface User{
  name:string,
  email: string,
  phonenumber: number
}
@Injectable({
  providedIn: 'root'
})
export class UsersService {
  Users:any;
//create instance of HttpClient
  constructor(private http:HttpClient) {
  }
  getStaticUsers(){
this.Users=[
      {userID:10,userName:'Youtube'},
      {userID:20,userName:'Twitter'},
      {userID:30,userName:'Facebook'},
      {userID:40,userName:'Instagram'},
      {userID:50,userName:'LinkedIn'}
    ];
    //console.log(this.Users);
    return this.Users;
  }
  getAPIUsers():Observable<User>{
    //we need to pass the url into get() URL paramter is mandatory other are
optional
    //set Http headers using following way
    const _headers = new HttpHeaders({
      'content-type':'application/json',
      'authenticationToken':'123456789'
    });
    //set Http Parameters using following way
    const _params = new HttpParams()
    .set('PageNum','10')
    .set('PageSize','100');
    return this.http.get<User>("https://jsonplaceholder.typicode.com/users",{headers
:_headers, params:_params});
  }
}
```

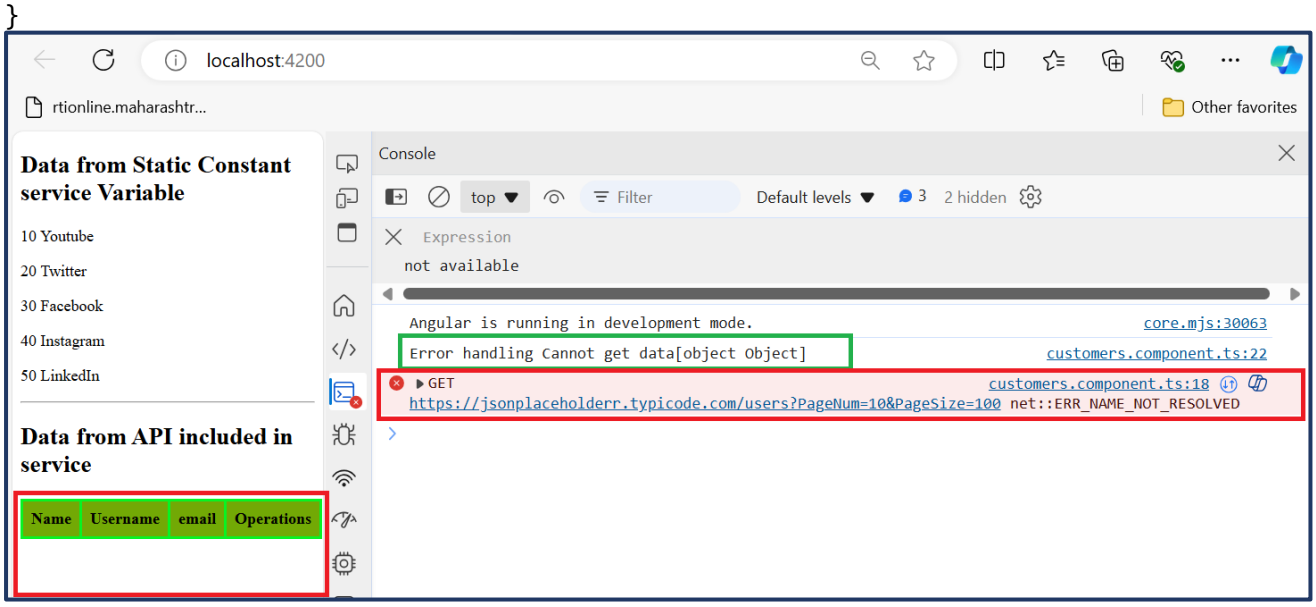
Case 5: Error handling when we subscribe it

We can show error message to the user also regarding error but below shown in console log.

Whenever we subscribe the data we also catch the errors.

customers.component.ts

```
ngOnInit(){
  //calling API method need to subscribe
  this.userService.getAPIUsers().subscribe(data =>{
    //we can now get individual data that we typecasted it
    //custom error handling
    console.log(data.email + data.name + data.phonenumber);
    this.APIUsers=data;
  },(err)=>{console.log("Error handling Cannot get data" + err)});
  this.staticUsers=this.userService.getStaticUsers();
}
```



Summary:

HttpClient GET Methods

ARC Tutorials

1. Making API calls to retrieve data is GET method call
2. To make a call all we need is a endpoint or an API URL
   1. get('api-endpoint')
3. We can also pass various parameters as options to the GET call
   1. get('url', options : { });
4. We will use options to pass parameters like **headers**, **params**, **responseType**, **withCredentials** etc
   1. Get('url', options: { headers: {}, params : {}}
5. The response type will be an **observable**

HTTPClient Get Method:

- Used for making API calls to retrieve data
- To make a call we need is a endpoint or an API URL:
    1. get('api-endpoint')
- We can also pass various parameters as options to get call
    1. get('URL',options:{});
- we will use options to pass parameters like headers, params, responseType, withCredentials etc.
    1. get('URL',options:{headers:{}, params:{}}
- The response type will be an observable.

Now we are able to do

- Simple HTTP Get method call
- Get method call with headers
- Get method call with Params
- Get method call with Type casting
- Subscribe and catch Error Handling

What are Observables in Angular?

Observables in Angular are a core feature of reactive programming that allow for the handling of asynchronous data, event handling, and multiple values over time:

- Asynchronous data: Observables can handle asynchronous data sources like user input, network requests, and timers.
- Event handling: Observables can be used to listen for and respond to user-input events.
- Multiple values: Observables can emit multiple values over time, unlike arrays which are static and contain a fixed set of values.
- Real-time data updates: When combined with HTTP, Observables can be used to manage real-time data updates.
- Error handling: Observables can be used to handle errors.
- Canceling requests: Observables can be used to cancel HTTP requests.

Observables are part of the Reactive Extensions for JavaScript (RxJS) library. Angular apps often use the RxJS library for Observables.

Here are some examples of how Observables are used in Angular:

- HTTP module: Uses Observables to handle AJAX requests and responses.
- Router and Forms modules: Use Observables to listen for and respond to user-input events.

HTTP and Observables in Angular with examples

HTTP and Observables in Angular are essential topics for building dynamic web applications. **HTTP** (Hypertext Transfer Protocol) allows Angular apps to communicate with back-end servers by sending and receiving data. This is crucial when you need to fetch data from an API or submit user inputs to a server. Angular's HttpClient service makes working with **HTTP requests** simpler and more efficient, handling various types of requests like GET, POST, PUT, and DELETE. Without it, integrating server data with the front-end would be complex and time-consuming.

On the other hand, **Observables** in Angular offer a powerful way to manage asynchronous data. They allow you to handle streams of data over time, which is ideal for dealing with HTTP responses. When combined with **HTTP**, Observables make it easy to manage real-time data updates, handle errors, and even cancel HTTP requests. By subscribing to an Observable, you can process data once it arrives, providing a smooth, reactive experience in your Angular application. Throughout this guide, I'll walk you through examples to show how **HTTP and Observables** work together in Angular applications.

HTTP in Angular

HTTP in Angular is a crucial component for making web requests to communicate with backend services, such as RESTful APIs. Angular provides the HttpClient module, which is a robust tool for making HTTP requests. This module offers methods like get , post , put , delete , etc., allowing developers to interact with remote servers by sending and receiving data. The HttpClient module simplifies the process of making HTTP calls and handling responses, including error handling, request cancellation, and the ability to transform response data.

For example, if you want to fetch data from an API, you can use the HttpClient service like this:

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
```

```
@Injectable({
  providedIn: 'root',
})
export class DataService {
  private apiUrl = 'https://api.example.com/data';

  constructor(private http: HttpClient) {}

  getData(): Observable<any> {
    return this.http.get<any>(this.apiUrl);
  }
}
```

In this example, the DataService class uses HttpClient to make a GET request to an external API. The getData() method returns an Observable that will emit the data received from the API when the request completes. This allows Angular applications to handle data asynchronously, making it easy to work with dynamic and remote data sources.

Key Features of Angular's HTTP Client

Here are five key features of Angular's **HttpClient :**

1. **Simplified API for HTTP Requests**: The HttpClient provides a clean and straightforward API for making HTTP requests like GET , POST , PUT , DELETE , and more. It abstracts the complexities of building HTTP requests and handling responses, making it easier for developers to interact with backend services.
2. **Automatic JSON Parsing**: By default, HttpClient automatically parses JSON responses into JavaScript objects. This eliminates the need for manual parsing and allows developers to work directly with the data in a more intuitive way.
3. **Interceptors**: Angular's HttpClient allows developers to define interceptors, which are functions that can intercept and manipulate HTTP requests or responses globally. This feature is useful for adding custom headers, logging requests, handling errors, or implementing authentication mechanisms.
4. **Observable-based API**: The HttpClient uses RxJS Observables to handle asynchronous operations. This makes it easy to perform operations like retrying failed requests, canceling ongoing requests, and composing multiple asynchronous tasks together using RxJS operators.
5. **Type Safety and Response Types**: With HttpClient , you can specify the expected response type for HTTP requests, allowing TypeScript to enforce type safety. This reduces runtime errors and enhances code reliability by ensuring that the returned data conforms to the expected structure.

What are Observables?

Observables in Angular are a powerful mechanism for handling asynchronous operations. They are part of the Reactive Extensions for JavaScript (RxJS) library, which Angular uses extensively. An Observable is a data stream that can emit multiple values over time, and subscribers can listen to these emitted values. This pattern is ideal for scenarios like HTTP requests, event handling, or dealing with real-time data, where you need to respond to data as it arrives or changes.

For example, when making an HTTP request using Angular's HttpClient , the request returns an Observable. You can subscribe to this Observable to get the data once the HTTP request completes. This allows you to handle asynchronous operations elegantly, including error handling, retry mechanisms, and even chaining multiple asynchronous tasks together.

Here's a basic example of using an Observable with **HttpClient** :

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
```

```
@Injectable({
  providedIn: 'root',
})
export class DataService {
  private apiUrl = 'https://api.example.com/data';

  constructor(private http: HttpClient) {}

  getData(): Observable<any> {
    return this.http.get<any>(this.apiUrl);
  }
}
```

In this example, **getData()** returns an Observable that emits the data from the HTTP GET request. The component that consumes this service can subscribe to the Observable and handle the data like this:

```
this.dataService.getData().subscribe(
  (data) => {
    console.log('Data received:', data);
  },
  (error) => {
    console.error('Error occurred:', error);
  }
);
```

Here, the component subscribes to the **getData()** Observable. When the HTTP request completes, the data is logged to the console, and if an error occurs, it is handled in the error callback. This pattern enables Angular applications to efficiently manage asynchronous data streams and create reactive, responsive applications.

Making HTTP Requests with Angular

Making HTTP requests in Angular is streamlined through the use of the HttpClient service, which is part of the @angular/common/http module. This service provides a straightforward way to perform HTTP operations such as GET, POST, PUT, DELETE, and more. By leveraging HttpClient , Angular applications can easily communicate with backend services to fetch or send data, making it essential for interacting with RESTful APIs.

Here's an example of how to make a GET request using HttpClient :

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class DataService {
  private apiUrl = 'https://api.example.com/data';

  constructor(private http: HttpClient) {}

  getData(): Observable<any> {
    return this.http.get<any>(this.apiUrl);
  }

  postData(data: any): Observable<any> {
    return this.http.post<any>(this.apiUrl, data);
  }
}
```

In this example, **DataService** has two methods: **getData()** and **postData()** . The **getData()** method makes a GET request to retrieve data from an API, returning an Observable that emits the response. The **postData()** method sends data to the API using a POST request. Both methods return Observables, allowing components to subscribe to them and handle the responses asynchronously. This approach ensures that Angular applications can easily interact with external data sources while maintaining a clean, reactive design pattern.

Why Use Observables for HTTP Requests?

Asynchronous Data Handling: Observables allow you to handle asynchronous data streams effectively. When making HTTP requests, the response may not be immediate. Observables let you subscribe to the data stream and react whenever the response is received, ensuring smooth and responsive application behavior.

Chaining and Composition: Observables support powerful operators like **map** , **filter** , **mergeMap** , and **switchMap** , enabling complex data transformations, chaining of multiple asynchronous operations, and better management of sequences of tasks. This makes handling complex scenarios, like dependent HTTP requests, much easier and more maintainable.

Built-in Error Handling: Observables provide a standardized way to handle errors in asynchronous operations. When making an HTTP request, you can easily manage errors using the **catchError** or **retry** operators, allowing you to gracefully handle failures and implement retry logic if needed.

Support for Multiple Emissions: Unlike Promises, which can only handle a single value or error, Observables can emit multiple values over time. This makes them suitable for handling scenarios where the data may arrive in chunks or be updated frequently, like WebSocket connections or live data streams.

Cancellation of Requests: Observables provide an easy way to cancel ongoing HTTP requests. By unsubscribing from the Observable, you can terminate the request, which is particularly useful in scenarios where the user navigates away from a page before the request completes or when dealing with live data updates that are no longer needed.

Frequently Asked Questions (FAQs)

1. What are HTTP services in Angular, and how do they work?

HTTP services in **Angular** are used to make **HTTP requests** from a client-side application to a server. They are typically used to fetch data from an API or send data to a server, using **RESTful** web services. In Angular, we use the **HttpClient** module, which simplifies the process of making HTTP requests and handling responses. Angular's **HttpClient** service automatically handles the complexities of making network requests, like handling headers, sending parameters, and dealing with different request types.

When I use **HttpClient**, I can make GET, POST, PUT, DELETE requests, and more. For example, to fetch data from an API, I can use the **http.get()** method, which returns an **Observable**. I'll then subscribe to this Observable to receive the response from the server. This makes it easy to handle asynchronous operations, such as waiting for data to arrive from an API. Angular also provides easy ways to deal with errors, transform responses, and retry failed requests.

2. What is an Observable in Angular, and why is it used with HTTP?

In **Angular,** an **Observable** is a way to handle asynchronous operations. It's part of **RxJS (Reactive Extensions for JavaScript)**, which is a powerful library for handling streams of data. When I make an HTTP request, the response isn't immediate. That's where Observables come in handy, as they allow me to "subscribe" to the response and act upon it once it arrives. With Observables, I can also handle multiple values over time, making it a flexible tool for **asynchronous** programming.

The reason **Angular** uses Observables for HTTP is that it allows for greater flexibility. With an Observable, I can manage multiple values, cancel requests, retry failed requests, and chain operations, all with simple, clean syntax. I can also transform the data received using operators like **map() , filter() ,** and **tap()** , which are part of **RxJS**.

3. How do you create an HTTP request in Angular using HttpClient?

To create an **HTTP request** in **Angular**, I first need to import the **HttpClientModule** in my **app.module.ts** file and include it in the imports array. Once that's done, I can inject the **HttpClient** service into my components or services where I need to make HTTP requests. The **HttpClient** service provides methods like **get() , post() , put() , delete()** to interact with a RESTful API.

Here's an example of how I make a **GET** request:

```
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) {}

getData() {
  this.http.get('https://api.example.com/data').subscribe(response => {
    console.log(response);
  });
}
```

In the example above, I use **http.get()** to fetch data from a given URL. The **subscribe()** method is necessary to actually send the request and receive the response. Without it, the request won't be executed.

4. What are the different types of HTTP requests supported by Angular's HttpClient?

Angular's **HttpClient** supports several types of HTTP requests, including **GET**, **POST**, **PUT**, **PATCH**, **DELETE**, and **OPTIONS**. Each type of request serves a different purpose depending on what I need to do with the data. For example, **GET** requests are used to fetch data from a server, while **POST** requests are used to send new data to the server. **PUT** and **PATCH** requests are used to update existing data, and **DELETE** is used to remove data from the server.

Here's a brief overview of what each request type does:

- **GET**: Retrieve data from the server.
- **POST**: Send new data to the server.
- **PUT**: Update existing data completely.
- **PATCH**: Update part of existing data.
- **DELETE**: Remove data from the server.

Using these methods, I can perform **CRUD operations** (Create, Read, Update, Delete) on my server, making it easy to build fully functional web applications. Here's an example of making a **POST** request:

```
postData() {
  const data = { name: 'Angular', version: '11' };
  this.http.post('https://api.example.com/new', data).subscribe(response => {
    console.log(response);
  });
}
```

This code sends new data to the server using the **POST** method and logs the response when the data is successfully sent.

5. How do you handle HTTP responses in Angular using Observables?

Handling **HTTP responses** in **Angular** with Observables is quite straightforward. When I make an HTTP request using **HttpClient** , the response comes back as an Observable. To access this data, I use the **subscribe()** method. This method allows me to specify what to do when the data arrives, or when an error occurs. Here's an example of handling a **GET** request:

```
getData() {
  this.http.get('https://api.example.com/data').subscribe(
    (response) => {
      console.log('Data received:', response);
    },
    (error) => {
      console.error('Error occurred:', error);
    }
  );
}
```

In this example, when the request succeeds, the response is logged, and if an error occurs, it's handled in the second callback. This approach ensures that I always know when the data arrives and can handle any issues that arise during the request.

RxJS operators also make handling responses easier. For example, I can use **map()** to transform the data before handling it in the **subscribe()** block. I can also use **catchError()** to catch and process errors globally, providing a much more scalable way of handling errors across multiple HTTP requests.

6. How do you subscribe to an Observable in Angular to get data from an HTTP request?

To get data from an **Observable** in Angular, I need to subscribe to it using the **subscribe()** method. This method allows me to "listen" to the values emitted by the Observable. When making an **HTTP request**, Angular's **HttpClient** returns an Observable, which doesn't execute until I subscribe to it. Once subscribed, the request is sent, and I can handle the data that comes back.

Here's an example:

```
getData() {
  this.http.get('https://api.example.com/data').subscribe((data) => {
    console.log('Data:', data);
  });
}
```

In this example, I use **http.get()** to retrieve data from an API. The response is logged inside the **subscribe()** method. Without this subscription, the request wouldn't actually be sent.

It's also important to handle errors by adding a second argument to **subscribe()** that takes an error callback.

Subscribing to an Observable is also useful for **canceling HTTP requests**. I can unsubscribe from the Observable if I want to cancel the request before it completes. This is particularly useful when dealing with user actions that might trigger multiple requests at the same time.

7. What are the differences between Promises and Observables in Angular?

The main difference between **Promises** and **Observables** in Angular lies in how they handle asynchronous data. **Promises** deal with a single value, while **Observables** can handle multiple values over time. If I only care about a single response from a service, a **Promise** might be enough, but for more complex scenarios where I need multiple values, I would use **Observables**.

A **Promise** resolves or rejects once, and it's not cancelable. Once the request is initiated, I cannot stop it. Here's an example of a Promise-based HTTP request:

```
fetchData() {
  fetch('https://api.example.com/data')
    .then((response) => response.json())
    .then((data) => console.log(data))
    .catch((error) => console.error(error));
}
```

An **Observable**, on the other hand, offers more control. It allows me to emit multiple values over time, retry requests, or even cancel them. With Observables, I have access to powerful **RxJS** operators like **map() , filter() ,** and **combineLatest()** to manipulate data streams more effectively.

8. How can you handle errors in HTTP requests using Observables?

Handling errors in **HTTP requests** using **Observables** is a crucial part of building robust applications. When making an HTTP request with Angular's **HttpClient** , errors can occur for various reasons, such as network failures or invalid responses from the server. The good thing about **Observables** is that they provide built-in mechanisms to handle these errors gracefully.

I can catch and manage errors by using **RxJS operators** like **catchError()** . This allows me to intercept the error and decide how to handle it. Here's an example of handling errors in a **GET** request:

```
getData() {
  this.http.get('https://api.example.com/data').pipe(
    catchError((error) => {
      console.error('Error occurred:', error);
      return throwError(error);
    })
  ).subscribe((data) => {
    console.log('Data:', data);
  });
}
```

In this example, if an error occurs during the HTTP request, it is logged using **catchError()** , and the error is rethrown to allow further handling. I can also return fallback data instead of throwing the error, ensuring the app keeps functioning even when requests fail.

9. How do you cancel an HTTP request in Angular with an Observable?

Canceling an **HTTP request** in Angular is possible when using **Observables**. One of the powerful features of Observables is their ability to be canceled by unsubscribing. This is particularly useful in scenarios where I have multiple HTTP requests happening simultaneously, and I want to cancel one or more of them to save resources or avoid unwanted updates.

To cancel a request, I need to create a **Subscription** to the Observable and then call **unsubscribe()** when I no longer need the data. Here's an example:

```
let subscription = this.http.get('https://api.example.com/data').subscribe(
```

```
  (data) => console.log(data)
);

// Later in the code, cancel the request
subscription.unsubscribe();
```

In this example, the HTTP request is sent, but if I decide that I no longer need the data before the response arrives, I can call **unsubscribe()** to cancel it. This is useful when the user navigates away from a page, and I want to avoid handling unnecessary responses.

10. What are common use cases for combining multiple Observables with HTTP in Angular?

There are several use cases for **combining multiple Observables** when dealing with **HTTP requests** in Angular. Often, I might need to make multiple requests to different APIs and wait for all the responses before performing some action. This can be done using **RxJS** operators like **forkJoin()** , **c ombineLatest() ,** or **mergeMap() .**

For example, if I need to fetch user data and their associated posts, I can use **forkJoin()** to wait for both HTTP requests to complete:

```
forkJoin({
  user: this.http.get('https://api.example.com/user'),
  posts: this.http.get('https://api.example.com/posts')
}).subscribe((results) => {
  console.log('User:', results.user);
  console.log('Posts:', results.posts);
});
```

In this example, both requests are sent at the same time, and I handle their responses only when both have finished. This is a common pattern when I need to load multiple pieces of data simultaneously and display them together on the UI.

Conclusion

HTTP and Observables are fundamental to Angular, enabling efficient, streamlined communication with servers and sophisticated handling of asynchronous data. By leveraging Angular's HTTP client and the power of RxJS Observables, developers can easily implement complex data retrieval and handling scenarios. This integration not only simplifies server-side communication but also enhances the overall responsiveness and performance of Angular applications. Understanding these concepts is essential for anyone looking to develop dynamic, data-driven web applications with Angular.

Httpclient Post Method:

1. Import and httpClientModule in app module

https://www.youtube.com/watch?v=TFFrbW9_iZE

HTTP POST – Use Cases

1. Simple HTTP POST method call
2. POST Method call with Headers
3. POST Method call with Params
4. POST Method call with Type Casting
5. Subscribe and Catch error

HttpClient POST Methods

1. Import and Add the HttpClientModule in our App Module
2. Import { HttpClientModule } from '@angular/core/http'
3. Import HttpClient in our service or component wherever we are making the HTTP request
   1. It's good practice to have all HTTP calls in services
   2. Becomes resuable and easy to maintain code
   3. Easy to share between various components
4. Import { HttpClient } from '@angular/core/http'
5. Inject the HttpClient in the constructor method of the class
6. Implement the POST method call
7. Step 5: Import the Services into the required calling component class
8. Step 6: Call the method to make the HTTP request

https://jasonwatmore.com/post/2019/11/21/angular-http-post-request-examples

Simple POST request with a JSON body and response type <any>

This sends an HTTP POST request to the Reqres api which is a fake online REST api that includes a /api/posts route that responds to POST requests with the contents of the post body and an id property. The id from the response is assigned to the local postId property in the subscribe callback function.

The response type is set to <any> so it can handle any properties returned in the response.

```
ngOnInit() {
    this.http.post<any>('https://reqres.in/api/posts', { title: 'Angular POST Request Example' }).subscribe(data => {
        this.postId = data.id;
    })
}
```

Example Angular component at https://stackblitz.com/edit/angular-post-request-examples?file=src/app/components/post-request.component.ts

POST request with strongly typed response

This sends the same request as the above but sets the response type to a custom Article interface that defines the expected response properties.

```
ngOnInit() {
    this.http.post<Article>('https://reqres.in/api/posts', { title: 'Angular POST Request Example' }).subscribe(data => {
        this.postId = data.id;
    })
}

...

interface Article {
    id: number;
    title: string;
}
```

Example Angular component at https://stackblitz.com/edit/angular-post-request-examples?file=src/app/components/post-request-typed.component.ts

POST request with headers set

This sends the same request again with a couple of headers set, the HTTP Authorization header and a custom header My-Custom-Header.

The below headers are created as a plain javascript object, they can also be created with the HttpHeaders class, e.g. const headers = new HttpHeaders({ 'Authorization': 'Bearer my-token', 'My-Custom-Header': 'foobar' })

To set or update headers on an existing HttpHeaders object call the set() method, e.g. const headers = new HttpHeaders().set('Authorization', 'Bearer my-token')

```
ngOnInit() {
    const headers = { 'Authorization': 'Bearer my-token', 'My-Custom-Header': 'foobar' };
    const body = { title: 'Angular POST Request Example' };
    this.http.post<any>('https://reqres.in/api/posts', body, { headers }).subscribe(data => {
        this.postId = data.id;
    }); }
```

Example Angular component at https://stackblitz.com/edit/angular-post-request-examples?file=src/app/components/post-request-headers.component.ts

POST request with error handling

The below examples show two different ways of handling an error from an HTTP POST request in Angular. They both send the same request to an invalid url on the api then assign the error to the errorMessage component property and log it to the console. The difference is how the error is caught, the first uses the subscribe error callback, the second uses the catchError operator.

Using the error callback to the RxJS subscribe method

The subscribe() method is passed an *observer* object with two callback functions, the next() function is called if the request is successful, the error() function is called if the request fails.

There is also a complete() function that we're not using here, it gets called after an Observable successfully emits its final value, i.e. after the next() function for an HTTP request. In this case it wouldn't be called because the Observable errors out.

```
ngOnInit() {
    this.http.post<any>('https://reqres.in/invalid-url', { title: 'Angular POST Request Example' }).subscribe({
        next: data => {
            this.postId = data.id;
        },
        error: error => {
            this.errorMessage = error.message;
            console.error('There was an error!', error);
        }
    })
}
```

Example Angular component at https://stackblitz.com/edit/angular-post-request-examples?file=src/app/components/post-request-error-handling-callback.component.ts

Using the RxJS catchError operator

The Observable response from the POST request is *piped* to the catchError operator which handles the error and returns a new Observable that doesn't emit any values.

The function passed to subscribe() is called if the request is successful, it's the equivalent of passing an *observer* object with only a next() function. It isn't called here because no values are emitted after the error is handled, the Observable simply completes.

RxJS Operators

Operators in RxJS accept an input Observable and return an output Observable, multiple can be piped together in the Observable chain and they are executed before values are emitted to subscribers. With the catchError operator you have the added options of returning a fallback value to subscribers when there is an error, or rethrowing a custom error using the RxJS throwError() function. For more info on RxJS operators see https://rxjs.dev/guide/operators.

```
ngOnInit() {
    this.http.post<any>('https://reqres.in/invalid-url', { title: 'Angular POST Request Example' })
        .pipe(catchError((error: any, caught: Observable<any>): Observable<any> => {
            this.errorMessage = error.message;
            console.error('There was an error!', error);
```

```
            // after handling error, return a new observable
            // that doesn't emit any values and completes
            return of();
        }))
        .subscribe(data => {
            this.postId = data.id;
        });
}
```

Example Angular component at https://stackblitz.com/edit/angular-post-request-examples?file=src/app/components/post-request-error-handling-catch-error.component.ts

Prerequisites for making HTTP requests from Angular

Before making HTTP requests from your Angular app you need to do a couple of things.

1. Add the HttpClientModule to the imports array of your AppModule like below on lines 3 and 10.

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

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

@NgModule({
    imports: [
        BrowserModule,
        HttpClientModule
    ],
    declarations: [
        AppComponent
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

2. Import the HttpClient into your component and add it to the constructor() params like below on lines 2 and 8.

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({ selector: 'app', templateUrl: 'app.component.html' })
export class AppComponent implements OnInit {
    postId;

    constructor(private http: HttpClient) { }

    ngOnInit() {
        // Simple POST request with a JSON body and response type <any>
        this.http.post<any>('https://reqres.in/api/posts', { title: 'Angular POST Request Example' }).subscribe(data => {
            this.postId = data.id;
        })
    }    }
```

Update History:

- 8 Nov 2022 - Updated examples to **Angular 14.2.9**. The previous version (**Angular 8.0.1**) is still available at https://stackblitz.com/edit/angular-http-post-examples.
- 7 Nov 2022 - Added error handling example with RxJS catchError operator
- 22 Apr 2021 - Replaced JSONPlaceholder API with Reqres API because JSONPlaceholder stopped allowing CORS requests
- 28 May 2020 - Posted video to YouTube of HTTP POST request examples with Angular at https://youtu.be/w2HFuzxxkRs
- 16 Jan 2020 - Added example of how to set HTTP headers on the request
- 21 Nov 2019 - Created Angular HTTP POST request examples

Angular - HTTP PUT Request Examples

https://jasonwatmore.com/post/2020/10/07/angular-http-put-request-examples

Simple PUT request with a JSON body and response type <any>

This sends an HTTP PUT request to the JSONPlaceholder api which is a fake online REST api that includes a /posts/1 route that responds to PUT requests with the contents of the put request body and the post id property. The post id from the response is assigned to the local postId property in the subscribe callback function. The response type is set to <any> so it handle any properties returned in the response.

```
ngOnInit() {
    const body = { title: 'Angular PUT Request Example' };
    this.http.put<any>('https://jsonplaceholder.typicode.com/posts/1', body)
        .subscribe(data => this.postId = data.id);
}
```

Example Angular component at https://stackblitz.com/edit/angular-http-put-examples?file=app/components/put-request.component.ts

PUT request with strongly typed response

This sends the same request as the above but sets the response type to a custom Article interface that defines the expected response properties.

```
ngOnInit() {
    const body = { title: 'Angular PUT Request Example' };
    this.http.put<Article>('https://jsonplaceholder.typicode.com/posts/1', body)
        .subscribe(data => this.postId = data.id);
}

...

interface Article {
    id: number;
    title: string;
}
```

Example Angular component at https://stackblitz.com/edit/angular-http-put-examples?file=app/components/put-request-typed.component.ts

PUT request with error handling

This sends a request to an invalid url on the api then assigns the error to the errorMessage component property and logs the error to the console.

The object passed to the request subscribe() method contains two callback functions, the next() function is called if the request is successful and the error() function is called if the request fails.

```
ngOnInit() {
    const body = { title: 'Angular PUT Request Example' };
    this.http.put<any>('https://jsonplaceholder.typicode.com/invalid-url', body)
        .subscribe({
            next: data => {
```

```
            this.postId = data.id;
        },
        error: error => {
            this.errorMessage = error.message;
            console.error('There was an error!', error);
        }
    });
}
```

Example Angular component at https://stackblitz.com/edit/angular-http-put-examples?file=app/components/put-request-error-handling.component.ts

PUT request with headers set

This sends the same request again with a couple of headers set, the HTTP Authorization header and a custom header My-Custom-Header.

The below headers are created as a plain javascript object, they can also be created with the HttpHeaders class, e.g. const headers = new HttpHeaders({ 'Authorization': 'Bearer my-token', 'My-Custom-Header': 'foobar' })

To set or update headers on an existing HttpHeaders object call the set() method, e.g. const headers = new HttpHeaders().set('Authorization', 'Bearer my-token')

```
ngOnInit() {
    const headers = { 'Authorization': 'Bearer my-token', 'My-Custom-Header': 'foobar' };
    const body = { title: 'Angular PUT Request Example' };
    this.http.put<any>('https://jsonplaceholder.typicode.com/posts/1', body, { headers })
        .subscribe(data => this.postId = data.id);
}
```

Example Angular component at https://stackblitz.com/edit/angular-http-put-examples?file=app/components/put-request-headers.component.ts

Prerequisites for making HTTP requests from Angular

Before making HTTP requests from your Angular app you need to do a couple of things.

1. Add the HttpClientModule to the imports array of your AppModule like below on lines 3 and 10.

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

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

@NgModule({
    imports: [
        BrowserModule,
        HttpClientModule
    ],
    declarations: [
```

```
        AppComponent
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

2. Import the HttpClient into your component and add it to the constructor() params like below on lines 2 and 8.

```
import { Component, OnInit } from "@angular/core";
import { HttpClient } from '@angular/common/http';

@Component({ selector: 'put-request', templateUrl: 'put-request.component.html' })
export class PutRequestComponent implements OnInit {
    postId;

    constructor(private http: HttpClient) { }

    ngOnInit() {
        const body = { title: 'Angular PUT Request Example' };
        this.http.put<any>('https://jsonplaceholder.typicode.com/posts/1', body)
            .subscribe(data => this.postId = data.id);
    }
}
```

Angular - HTTP DELETE Request Examples

https://jasonwatmore.com/post/2020/10/10/angular-http-delete-request-examples

Simple DELETE request

This sends an HTTP DELETE request to the JSONPlaceholder api which is a fake online REST api that includes a /posts/1 route that responds to DELETE requests with a HTTP 200 OK response. When the response is received the Angular component displays the status message 'Delete successful'.

```
ngOnInit() {
    this.http.delete('https://jsonplaceholder.typicode.com/posts/1')
        .subscribe(() => this.status = 'Delete successful');
}
```

Example Angular component at https://stackblitz.com/edit/angular-http-delete-examples?file=app/components/delete-request.component.ts

DELETE request with error handling

This sends a request to an invalid url on the api then assigns the error to the errorMessage component property and logs the error to the console.

The object passed to the request subscribe() method contains two callback functions, the next() function is called if the request is successful and the error() function is called if the request fails.

```
ngOnInit() {
    this.http.delete('https://jsonplaceholder.typicode.com/invalid-url')
        .subscribe({
            next: data => {
                this.status = 'Delete successful';
            },
            error: error => {
                this.errorMessage = error.message;
                console.error('There was an error!', error);
            }
        });
}
```

Example Angular component at https://stackblitz.com/edit/angular-http-delete-examples?file=app/components/delete-request-error-handling.component.ts

DELETE request with headers set

This sends the same request again with a couple of headers set, the HTTP Authorization header and a custom header My-Custom-Header.

The below headers are created as a plain javascript object, they can also be created with the HttpHeaders class, e.g. const headers = new HttpHeaders({ 'Authorization': 'Bearer my-token', 'My-Custom-Header': 'foobar' })

To set or update headers on an existing HttpHeaders object call the set() method, e.g. const headers = new HttpHeaders().set('Authorization', 'Bearer my-token')

```
ngOnInit() {
    const headers = { 'Authorization': 'Bearer my-token', 'My-Custom-Header': 'foobar'
};
```

```
    this.http.delete('https://jsonplaceholder.typicode.com/posts/1', { headers })
        .subscribe(() => this.status = 'Delete successful');
}
```

Example Angular component at https://stackblitz.com/edit/angular-http-delete-examples?file=app/components/delete-request-headers.component.ts

Prerequisites for making HTTP requests from Angular

Before making HTTP requests from your Angular app you need to do a couple of things.

1. Add the HttpClientModule to the imports array of your AppModule like below on lines 3 and 10.

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

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

@NgModule({
    imports: [
        BrowserModule,
        HttpClientModule
    ],
    declarations: [
        AppComponent
    ],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

2. Import the HttpClient into your component and add it to the constructor() params like below on lines 2 and 8.

```
import { Component, OnInit } from "@angular/core";
import { HttpClient } from '@angular/common/http';

@Component({ selector: 'delete-request', templateUrl: 'delete-
request.component.html' })
export class DeleteRequestComponent implements OnInit {
    status;

    constructor(private http: HttpClient) { }

    ngOnInit() {
        this.http.delete('https://jsonplaceholder.typicode.com/posts/1')
            .subscribe(() => this.status = 'Delete successful');
    }
}
```

Top Angular Interview Question & Answers
Prepare Your Fundamentals Interview Questions
1. What is Angular and its key features?
Angular is a TypeScript-based front-end web application framework. It follows the MVC (Model-View-Controller) architecture. It is used to build front-end and single-page applications that run on JavaScript. It targets both the browser and the server. Angular is a full-fledged framework, i.e., it takes care of many aspects of front–end web applications such as HTTP requests, routing, layout, forms, reactivity, validation, etc.
Key features of Angular are:

- Component-based architecture – applications are written as a set of independent components.
- Parts can be created, tested, and integrated using Angular CLI.
- Great support for complex animations without writing much code.
- Because of the component router, Angular supports automatic code-splitting. Hence only the code required to render a particular view is loaded.
- Cross-platform application development.
- Template syntax for creating UI views.

2. Explain the difference between Angular and AngularJS.

Features	**Angular**	**AngularJS**
Architecture	It makes use of directives and components	Supports Model-View-Controller design
Language	TypeScript	JavaScript
Mobile Support	Angular offers mobile support.	Unlike Angular, AngularJS does not offer mobile support
Routing	@Route configuration is used to define routing information	@routeProvider is used to provide routing information
Dependency Injection	It supports hierarchical dependency injection, along with a unidirectional tree-based change direction	It does not support dependency injection
Structure	Its simplified structure makes it easy for professionals to develop and maintain large applications easily	It is comparatively less manageable
Expression Syntax	Angular uses () to bind an event while [] to bind a property	It requires professionals to use the correct ng directive to bind a property or an event

3. What are the different types of data binding in Angular?
There are two types of data binding in Angular:

1. **One-Way Data Binding:** Here, the data flows in a single direction, either from the component to the view (interpolation or property binding) or from the view to the component (event binding).
2. **Two-Way Data Binding:** Here, the immediate changes to the view & component will be reflected automatically, i.e. when the changes are made to the component or model then the view will render the changes simultaneously.

Angular Components Interview Questions Preparation
4. What are the different ways to pass data between components?
In Angular, there are several ways to pass data between components:

1. **Input Properties (Parent to Child)**

With this, you can pass data from a parent component to a child component. In the child component, you declare an input property using the @Input() decorator and the parent component binds to this property using property binding syntax ([property]="value").

2. **Output Properties with Event Emitters (Child to Parent)**

Output properties combined with event emitters allow child components to send data to parent components. The child component emits events using an EventEmitter and the @Output() decorator. The parent component listens for these events using event binding syntax eventEmitterName.subscribe().

3. **Services (Unrelated Components)**

Services act as singletons within an Angular application and can be used to share data between unrelated components. Components can inject the same service instance and use it to share data across the application.

4. **ViewChild and ContentChild (Parent to Child)**

ViewChild and ContentChild decorators allow a parent component to access its child components directly. These decorators can be used to reference child component instances and access their properties and methods.

5. **NgRx (State Management)**

NgRx is a state management library for Angular applications based on the Redux pattern. It allows components to share data by managing the application state centrally. Components can dispatch actions to update the state and subscribe to changes in the state to react accordingly.

6. **Route Parameters (Routing)**

Route parameters can be used to pass data between components in different routes. Components can retrieve route parameters from the ActivatedRoute service and use them to fetch data or configure component behavior.

5. How do you handle events in Angular components?

In Angular components, we can handle events using event binding, event handlers, and event emitters. Let's look at them:

1. **Event Binding:** This allows the view to communicate changes back to the component when an event occurs, such as a button click or input change. Event binding is denoted by parentheses, like (event)="handler()".

Example

```
<button (click)="onClick()">Click Me</button>
```

2. **Event Handlers:** In the component class, define the event handler method that will be called when the event occurs. This method can take parameters to capture event data passed by the event object.

Example

```
// Component class (TypeScript file)
export class MyComponent {
  onClick() {
    console.log('Button clicked!');
  }
}
```

3. **Event Emitters:** They allow child components to communicate with parent components by emitting custom events.

Example

Child Component

```
// Child component class (TypeScript file)
import { EventEmitter, Output } from '@angular/core';

export class ChildComponent {
  @Output() myEvent = new EventEmitter();
```

```
  onClick() {
    this.myEvent.emit('Event data');
  }
}
```

```
<button (click)="onClick()">Click Me</button>
```

Parent Component

```
<app-child (myEvent)="onChildEvent($event)"></app-child>
```

```
// Parent component class (TypeScript file)
export class ParentComponent {
  onChildEvent(data: string) {
    console.log('Event data:', data);
  }
}
```

Angular Directives & Pipes Interview Questions For Freshers

6. What are directives and their types in Angular?

Angular Directives are attributes that allow the user to write new HTML syntax specific to their applications. They execute whenever the Angular compiler finds them in the DOM. Angular supports three types of directives:

1. **Component Directives:** These are the directives that come with a template and are the most common type of directives.
2. **Attribute Directives:** These are the directives that can change the appearance of a component, page, or even other directives.

The following command is used to create an attribute directive:

```
ng g directive YellowBackground
```

3. **Structural Directives:** These directives are responsible for changing the DOM layout either by adding or removing the DOM elements. Every structural directive has a '*' sign before it.

7. How do you create custom directives?

1. **Create Directive Class**
   - Create a TypeScript class for your custom directive. This class should be decorated with the @Directive decorator.
   - Use the selector property of the decorator to specify the CSS selector that identifies where the directive should be applied.

Example

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

@Directive({
  selector: '[appCustomDirective]'
})
export class CustomDirective {
  constructor(private el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

2. **Inject ElementRef**

- Inject the ElementRef service into the constructor of your directive class. This service provides access to the host element to which the directive is applied.
- Use ElementRef.nativeElement to access the DOM element and apply custom functionality.

```
constructor(private el: ElementRef) {
  el.nativeElement.style.backgroundColor = 'yellow';
}
```

3. **Register Directive**
   - To use the custom directive, you need to declare it in the declarations array of the Angular module where it will be used.
   - If the directive is used in multiple modules, you can create a shared module and import it into the modules where the directive is needed.

```
import { NgModule } from '@angular/core';
import { CustomDirective } from './custom.directive';

@NgModule({
  declarations: [
    CustomDirective
  ],
  exports: [
    CustomDirective
  ]
})
export class SharedModule { }
```

4. **Use Directive in HTML**
   - To apply the custom directive to an HTML element, use the selector specified in the @Directive decorator.

```
<div appCustomDirective>
  This div will have a yellow background.
</div>
```

Angular Services & Dependency Injection Questions

8. What are services and their benefits in Angular?

Objects classified as services are those that are only instantiated once in the course of an application. A service's primary goal is to exchange functions and data with various Angular application components. To define a service, use the @Injectable decorator. Any component or directive can call a function specified inside a service.

Here are some key benefits of Angular services:

- **Code Reusability:** By separating your business logic into services, you can reuse the same code in different parts of your application, promoting code modularity and reducing code duplication.
- **Dependency Injection:** It allows you to define dependencies for your components and have them injected automatically by the framework.
- **Single Responsibility Principle (SRP):** Services help enforce the Single Responsibility Principle (SRP) by providing a dedicated place to put your business logic and data manipulation code.
- **State Management:** You can use services to store and manipulate data, allowing components to access and update it as needed.

- **Scalability:** By structuring your application's functionality into services, you can easily add new features or modify existing ones without impacting other parts of your codebase.

9. Describe best practices for using services in Angular.

Below are some of the best practices for using services in Angular:

- **Don't include business logic in your services:** Services should be used for providing functionality, not for implementing business logic. Business logic should be implemented in components or other classes that are specific to your application domain.
- **Make sure your services are testable:** Services should be designed in a way that makes them easy to test. This means avoiding using global state or other external dependencies, and instead relying on dependency injection to provide any necessary functionality.
- **Use interfaces to define your services:** It can help to make your code more modular and maintainable. It also makes it easier to use other libraries and tools that rely on interfaces.
- **Keep your services small:** Services should be designed to provide specific functionality, rather than attempting to handle everything at once. If a service starts to become too large, consider breaking it up into smaller, more specific services.
- **Avoid using services to handle view logic:**Services should be used to provide functionality that can be shared across multiple components. They should not be used to handle view-related logic, which should be implemented in the components themselves.

Angular Routing & Navigation Questions

10. What are the different types of routing in Angular?

There are four main types of routing in Angular:

1. **Component Routing:** Component routing is used to navigate between different pages or components within an Angular application. The URL for each page or component is defined in a routing configuration, and when the user navigates to that URL, the corresponding component is displayed.

```
const appRoutes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' }
];

@NgModule({
  imports: [ RouterModule.forRoot(appRoutes) ],
  exports: [ RouterModule ]
})
export class AppRoutingModule { }
```

2. **Child Routing:** Child routing is used to navigate between child components within a parent component. The parent component serves as a container for the child components, and the URL for each child component is defined as a child route of the parent component.

```
const routes: Routes = [
  {
    path: 'home',
    component: HomeComponent,
```

```
    children: [
      { path: 'about', component: AboutComponent },
      { path: 'contact', component: ContactComponent }
    ]
  }
];

@NgModule({
  imports: [ RouterModule.forChild(routes) ],
  exports: [ RouterModule ]
})
export class HomeRoutingModule { }
```

3. **Lazy Loading:** Lazy loading is a technique in which a module is loaded only when it's needed, rather than loading all modules at the start of the application. This can help reduce the initial load time of your application, especially if it has many modules.

```
const routes: Routes = [
  {
    path: 'home',
    loadChildren: () => import('./home/home.module').then(m => m.HomeModule)
  },
  {
    path: 'about',
    loadChildren: () => import('./about/about.module').then(m => m.AboutModule)
  },
  {
    path: 'contact',
    loadChildren: () => import('./contact/contact.module').then(m => m.ContactModule)
  }
];

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

4. **Dynamic Routing:** Dynamic routing is a technique in which the routing configuration is generated dynamically, based on the data that's available at runtime.

```
const routes: Routes = [
  {
    path: ':id',
    component: DynamicComponent,
    resolve: {
      data: DynamicDataResolver
    }
  }
];

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

```
  exports: [ RouterModule ]
})
export class AppRoutingModule {}
```

11. Explain how to configure routing in an Angular application.

1. **Install Angular Router**
   - Install Angular Router in your project by running the following command in your terminal

```
npm install @angular/router
```

2. **Define Routes**
   - In your Angular application, define the routes for different views/pages in the app-routing.module.ts file.
   - Define an array of route objects, where each object specifies the path, component to render, and any additional configuration.

Example

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent }
];

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

3. **Configure Router Outlet**
   - In your root component's template file (usually app.component.html), add an element. This is where Angular will render the component associated with the current route.

Example

```
<router-outlet></router-outlet>
```

4. **Configure Navigation**
   - Use the routerLink directive in your HTML templates to navigate between different routes. The routerLink directive generates a link based on the specified route path.

Example

```
<a routerlink="/">Home</a>
<a routerlink="/about">About</a>
<a routerlink="/contact">Contact</a>
```

5. **Handle Route Parameters**
   - You can define routes with parameters to pass data between components or to specify dynamic paths.

Example

```
const routes: Routes = [
  { path: 'products/:id', component: ProductDetailComponent }
```

```
];
```

6. **Import AppRoutingModule**
    - Finally, import the AppRoutingModule in your root module (usually app.module.ts) to enable routing in your Angular application.

Example

```
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 { }
```

Angular Interview Questions for Freshers

12. How does an Angular application work?

1. Every Angular app consists of a file named **angular.json**. This file will contain all the configurations of the app. While building the app, the builder looks at this file to find the entry point of the application.

```
"build": {
      "builder": "@angular-devkit/build-angular:browser",
      "options": {
        "outputPath": "dist/angular-starter",
        "index": "src/index.html",
        "main": "src/main.ts",
        "polyfills": "src/polyfills.ts",
        "tsConfig": "tsconfig.app.json",
        "aot": false,
        "assets": [
          "src/favicon.ico",
          "src/assets"
        ],
        "styles": [
          "./node_modules/@angular/material/prebuilt-themes/deeppurple-amber.css",
          "src/style.css"
        ]
      }
    }
```

2. Inside the build section, the main property of the options object defines the entry point of the application which in this case is main.ts.
3. The main.ts file creates a browser environment for the application to run, and, along with this, it also calls a function called bootstrapModule, which bootstraps the

application. These two steps are performed in the following order inside the main.ts file:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
platformBrowserDynamic().bootstrapModule(AppModule)
```

4. The AppModule is declared in the app.module.ts file. This module contains declarations of all the components.

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  entryComponents: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Here, the AppComponent is getting bootstrapped.

5. This component is defined in the **app.component.ts** file. This file interacts with the webpage and serves data to it.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular';
}
```

6. Each component is declared with three properties:
    1. **Selector:**used for accessing the component
    2. **Template/TemplateURL:** contains HTML of the component
    3. **StylesURL:** contains component-specific stylesheets
7. After this, Angular calls the **index.html** file. This file consequently calls the root component that is app-root. The root component is defined in **app.component.ts**.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Angular</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
```

```
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

The HTML template of the root component is displayed inside thetags.

13. What are the benefits of using Angular?

The top benefits of utilizing the Angular framework are listed below:

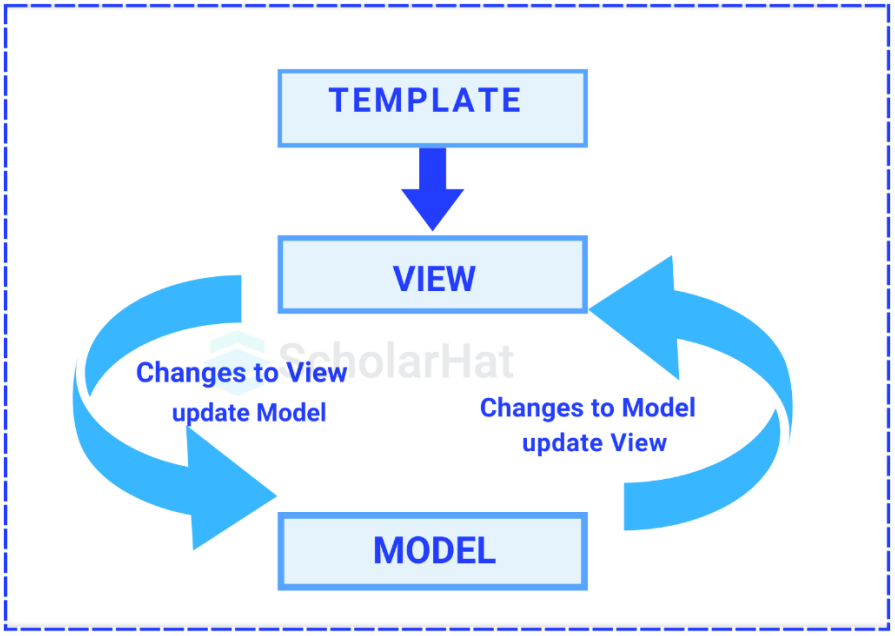
- Angular facilitates bidirectional data coupling.
- Its architecture adheres to the MVC pattern.
- Both Angular and static templates are supported.
- It makes adding a custom directive easier.
- RESTfull services are also supported.
- Angular provides functionality for validations.
- Client and server communication is enabled by Angular.
- Support for dependency injection is offered.
- It has strong features like animation and event handlers, among others.

14. What is the purpose of TypeScript in Angular development?

- **Static Typing:** There is a feature of static typing in TypeScript, through which you can define and enforce the types of variables, function parameters, and return values. This helps in error detection at compile-time.
- **Enhanced Tooling:** TypeScript provides a rich set of tooling features like code navigation, autocompletion, and refactoring support. These features improve developer productivity and make it easier to work with larger codebases.
- **Better Code Organization:** TypeScript supports object-oriented programming features like classes, interfaces, and modules. These features help organize code into reusable and maintainable components, making the development process more structured and efficient.
- **Improved Readability:** With the help of TypeScript, developers can write self-documented code by providing type annotations. This improves code readability and makes it easier for other developers to understand and collaborate on the project.
- **Compatibility with Existing JavaScript Code:** Being a superset of JavaScript, TypeScript can seamlessly integrate with existing JavaScript projects. This allows developers to gradually introduce TypeScript into their codebase without needing a complete rewrite.

15. Explain the concept of data binding in Angular and its different types.

Data binding is a mechanism that allows synchronization of data between the model and the view, making it easier to manage and update user interfaces efficiently.



There are four types of Data binding in Angular:

1. **Property Binding:** This is achieved by using square brackets to bind a property of an HTML element to a component property. For instance, [property]="data" binds the value of the component's "data" property to the HTML element's property.

Syntax

```
[class]="variable_name"
```

2. **Event Binding:** This allows the view to communicate changes back to the component when an event occurs, such as a button click or input change. Event binding is denoted by parentheses, like (event)="handler()".

Syntax

```
<button class="btn btn-block"
   (click)=showevent($event)>
   Event
</button>
showevent(event) {
   alert("Welcome to ScholarHat");
}
```

3. **String Interpolation:**This involves displaying component data in the view by enclosing the property or expression in double curly braces, like {{ data }}. Angular automatically updates the view whenever the underlying data changes.

Syntax

```
class="{{variable_name}}"
```

4. **Two-way Data Binding:** Here, the immediate changes to the view & component will be reflected automatically, i.e. when the changes are made to the component or model then the view will render the changes simultaneously. Similarly, when the data is altered or modified in the view then the model or component will be updated accordingly.

16. Explain the concept of single-page applications (SPAs).

A single-page application is a website that loads a single document and overwrites the existing one with new data from a web server instead of reloading pages individually from the beginning. Due to this ability, the page content updates in real-time based on user actions with quick transitions and without refreshing.

The ability to provide new content seamlessly based on user actions, such as button clicks makes single-page applications stand out from their counterparts. Instead of refreshing an entire page, the application updates or alters components based on the user's actions and needs, making it quick to respond and easy to interact with in real-time.

17. What is new in Angular 17?

- New, declarative control flow
- Deferred loading blocks
- View Transitions API support
- Support for passing in @Component.styles as a string
- Angular's animation code is lazy-loadable
- TypeScript 5.2 support
- The core Signal API is now stable (PR)
- Signal-based components are not ready yet, they won't be a part of Angular 17
- Node.js v16 support has been removed and the minimum support version is v18.13.0 (PR)
- We expect that Esbuild will be the default builder and the default dev server will use Vite

18. What are decorators in Angular?

Decorators are design patterns or functions that define how Angular features work. They are employed to alter a class, service, or filter beforehand. Angular supports four types of decorators, they are:

1. Class decorators, such as **@Component** and **@NgModule**
2. Property decorators for properties inside classes, such as **@Input** and **@Output**
3. Method decorators for methods inside classes, such as **@HostListener**
4. Parameter decorators for parameters inside class constructors, such as **@Inject**

Read in Details : https://www.javatpoint.com/angular-decorators

19. What are Angular Templates?

Angular templates are written in HTML and feature Angular-specific elements and properties. These templates are merged with information from the model and controller, which is then rendered to present the user with a dynamic view. In an Angular component, there are two ways to construct a template:

1. **Linked Template:** A component may include an HTML template in a separate HTML file. As illustrated below, the templateUrl option is used to indicate the path of the HTML template file.

Example

```
@Component({
   selector: "app-greet",
   templateUrl: "./component.html"
})
```

2. **Inline Template:** The component decorator's template config is used to specify an inline HTML template for a component. The Template will be wrapped inside the single or double quotes.

Example

```
@Component({
   selector: "app-greet",
   template: `
      Hello {{name}} how are you ?
```

Welcome to interviewbit !

`

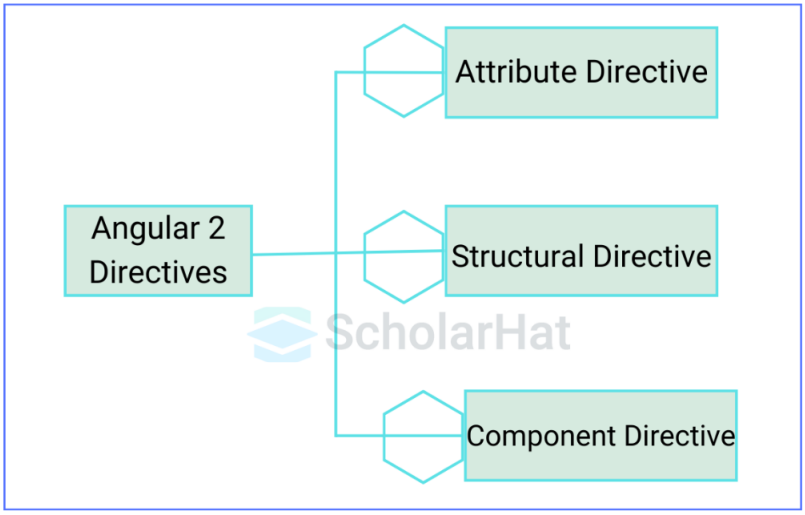
})

20. What are Angular Annotations?

Angular Annotations are hard-coded language features. Annotations are merely metadata that is set on a class to reflect the metadata library. When a user annotates a class, the compiler adds an annotations property to the class, saves an annotation array in it, and then attempts to instantiate an object with the same name as the annotation, providing the metadata into the constructor. Annotations in AngularJs are not predefined, therefore we can name them ourselves.

21. What are Angular Directives?

A directive is a class in Angular that is declared with a @Directive decorator. Every directive has its own behavior and can be imported into various components of an application. When Angular begins compiling the TypeScript, **CSS**, and **HTML** files into a single JavaScript file, it scans through the entire code and looks for a directive that has been registered.



There are three types of directives in Angular:

1. **Component Directives**
2. **Structural Directives**
3. **Attribute Directives**

22. What are Angular Components?

Components are the core building pieces in Angular that manage a portion of the UI for any application. The @Component decorator is used to define a component. Every component comprises three parts: a template that loads the component's view, a stylesheet that specifies the component's look and feel, and a class that includes the component's business logic.

Angular Interview Questions for Experienced Professionals

After getting conceptual clarity on the fundamental topics, let's step a level further towards the questions for a little trained and experienced frontend or Angular developer.

23. Explain the differences between AOT (Ahead-of-Time) and JIT (Just-in-Time) compilation and their pros and cons.

AOT (Ahead-of-Time)	**JIT (Just-in-Time)**
Compiles code before the Angular application is loaded in the browser	Compiles Code during runtime when the Angular app is launched in the client's browser.

Generates a production-ready output with optimizations, ready for deployment without additional build steps.	Requires an additional build for production, potentially adding extra time to the deployment process
AOT produces smaller bundle sizes, which means faster downloads for users	Produces larger bundle sizes due to in-browser compilation, potentially impacting loading speed
AOT catches and reports template errors during the compilation phase, ensuring more reliable applications with fewer runtime issues.	Identifies errors during runtime, which may lead to issues being discovered after the application is already in use
Does not allow dynamic updates in production, requiring a rebuild for any changes	Allows dynamic updates during development, making it easier to see immediate results
Better compatibility with older browsers, ensuring wider accessibility	Slightly less compatible with older browsers compared to AOT

Advantages of AOT Compilation

- Faster rendering
- Fewer asynchronous requests
- Smaller Angular framework download size
- Quick detection of template errors
- Better security

Advantages of JIT Compilation

- Faster Development Cycle
- Dynamic Compilation
- Optimized Bundle Sizes
- Runtime Error Reporting
- Dynamic Template Compilation

Disadvantages of AOT Compilation

- Increased Build Time
- Complexity of Configuration
- Increased Bundle Size
- Debugging Challenges

Disadvantages of JIT Compilation

- Browser Compatibility
- Debugging Complexity
- Potential Performance Overhead

24. Describe lazy loading and its benefits for optimizing application performance.

Lazy loading in angular refers to the technique of loading webpage elements only when they are required. Instead of loading all media at once, which would use a lot of bandwidth and bog down page views, those elements are loaded when their location on the page is about to scroll into view.

Implementing Lazy Loading in Angular

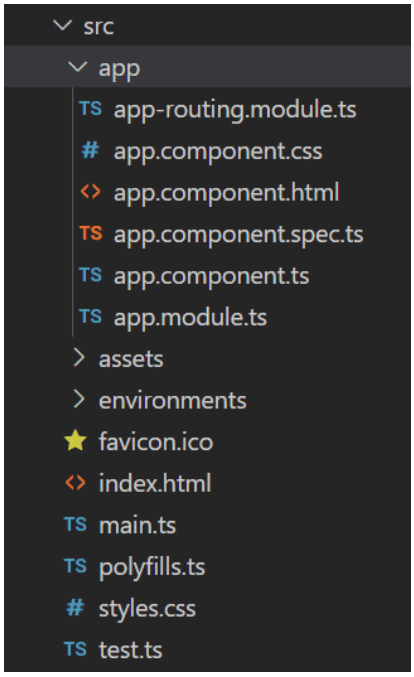
1. **Set Up Your Project:** Install the CLI using npm by running the command

```
npm install -g @angular/cli
```

Create a project, for example, Lazy Loading Demo

```
ng new lazy-loading-demo --routing
```

Now, you'll be working exclusively in the src/app folder, which contains the code for your app. This folder contains your main routing file, app-routing.module.ts.



2. **Create Feature Modules**

Create separate modules for each feature of your application. Each feature module should contain its components, services, and other related files.

Example

products.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterModule } from '@angular/router';
import { ProductsListComponent } from './products-list/products-list.component';
import { ProductDetailComponent } from './product-detail/product-detail.component';

@NgModule({
  declarations: [
    ProductsListComponent,
    ProductDetailComponent
  ],
  imports: [
    CommonModule,
    RouterModule.forChild([
      { path: '', component: ProductsListComponent },
      { path: ':id', component: ProductDetailComponent }
    ])
  ]
})
export class ProductsModule { }
```

3. **Configure Routes for Lazy Loading:** Define routes for each feature module in your app-routing.module.ts file.
4.
5. const routes: Routes = [
6. { path: 'dashboard', component: DashboardComponent },
7. { path: 'products', loadChildren: () => import('./products/products.module').then(m => m.ProductsModule) },
8. // Other routes...

9.];
10. **Load Feature Modules Lazily:** Use the loadChildren property in the route configuration to specify the path to the feature module file and load it dynamically using the import() function.

Example

loadChildren: () => import('./products/products.module').then(m => m.ProductsModule)

11. **Update AppModule:** Remove references to feature modules from the imports array of the AppModule since they are now loaded lazily.
12. **Test Lazy Loading:** Test your application to ensure that feature modules are loaded only when their routes are accessed.

25. What are Pure Pipes?

These are pipelines that only employ pure functions. As a result, a pure pipe does not use any internal state, and the output remains constant as long as the parameters provided remain constant. Angular calls the pipe only when the parameters being provided change. Throughout all components, a single instance of the pure pipe is used.

26. What do you understand by impure pipes?

Angular calls an impure pipe for each change detection cycle, independent of the change in the input fields. For each of these pipes, several pipe instances are produced. These pipes' inputs can be altered.

By default, all pipes are pure. However, you can specify impure pipes using the pure property as specified below:

```
@Pipe({
   name: 'impurePipe',
   pure: false/true
})
export class ImpurePipe {}
```

27. What is Bootstrap? How is it embedded into Angular?

Bootstrap is a popular open-source CSS framework used for building responsive and mobile-first websites and web applications. It provides a set of pre-styled components, such as buttons, forms, navigation bars, and grid layouts, as well as CSS utilities for styling and layout management.

The bootstrap library can be integrated into your program in two different methods:

1. **Angular Bootstrap through CDN:** You can include Bootstrap CSS and JavaScript files directly from a content delivery network (CDN) in your Angular application's index.html file.

Example

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Angular App</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
  <app-root></app-root>
  <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>
```

```
  <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.1/dist/umd/popper.min.js"></sc
ript>
  <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</body>
</html>
```

2. **Using npm Package:** You can install Bootstrap as an npm package and import its CSS files directly into your Angular components or stylesheets.
    - Install Bootstrap using npm
    -
    - npm install bootstrap
    - Import Bootstrap CSS in your styles.scss or styles.css file
    -
    - @import '~bootstrap/dist/css/bootstrap.min.css';

Angular Interview Questions for Experienced Professionals (2 to 3 years)

If you have spent more than two years in the frontend or Angular development, you are now in the position to understand the little bit advanced level of questions mentioned below:

28. What do Angular filters do? List a few of them.

Filters are used to format an expression and present it to the user. They can be used in view templates, controllers, or services.

Filter name	**Description**
Uppercase	Convert string to uppercase
Lowercase	Convert string to lowercase
Date	Convert the date to the specified format
Currency	Convert the number to currency format
Number	Format the number into a string
Orderby	Orders an array by specific expression
limitTo	Limits array into the specified number of elements; string to specified number of characters
JSON	Format object to JSON string
Filter	Select a subset of items from the array

29. In Angular, what is the scope?

The scope in Angular binds the HTML, i.e., the view, and the JavaScript, i.e., the controller. It as expected is an object with the available methods and properties. The scope is available for both the view and the controller. When you make a controller in Angular, you pass the $scope object as an argument.

How to Use the Scope in Angular?

```
<!DOCTYPE html>
<html>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="myApp" ng-controller="myCtrl">

<h1>{{websitename}}</h1>

</div>
```

```
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
   $scope.websitename = "ScholarHat";
});
</script>
<p>The property "websitename" was made in the controller, and can be referred to in the view by using the {{ }} brackets.</p>
</body>
</html>
```

30. What are lifecycle hooks in Angular? Explain a few lifecycle hooks.

Every component in Angular has a lifecycle. Angular creates and renders these components and also destroys them before removing them from the DOM. This is achieved with the help of lifecycle hooks. Throughout the entire process, Angular Lifecycle hooks are used to monitor the phases and initiate modifications at particular points.

The following are the eight lifecycle hooks in Angular

Lifecycle hooks	**Functions**
ngOnChanges( )	Called when the input properties of the component are changed.
ngOnInit( )	Called after the ngOnChanges hook, to initialize the component and set the input properties
ngDoCheck( )	Called to detect and act on changes
ngAfterContentInit( )	Called after the first ngDoCheck hook, to respond after the content is projected inside the component
ngAfterContentChecked( )	Called after ngAfterContentInit (and every subsequent ngDoCheck) to respond after the projected content is checked
ngAfterViewInit( )	Called after a component's view, or after initializing a child component's view
ngAfterViewChecked( )	Called after ngAfterViewInit, to respond when the component's view or child component's view is checked
ngOnDestroy( )	Called immediately before destroying the component, to clean up the code and detach the event handlers

31. What is Eager and Lazy Loading?

1. **Eager Loading:** It is the default module-loading strategy. Eager-loading feature modules are loaded before the program begins. This is primarily utilized for small-scale applications.
2. **Lazy Loading:** Lazy loading loads the feature modules dynamically as needed. This speeds up the application. It is utilized for larger projects where all of the modules are not required at the start.

32. How can I utilize an Angular template with ngFor?

1. **Component Class:** First of all you need to have a component class with a property that holds the collection of items you want to iterate over. import { Component } from '@angular/core';

Example

```
@Component({
  selector: 'app-my-component',
```

```
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.css']
})
export class MyComponent {
  items: string[] = ['Item 1', 'Item 2', 'Item 3'];
}
```

2. **Template:** In your template file (my-component.component.html), use the ngFor directive to iterate over the items array and render elements dynamically for each item
3.
4. <div *ngFor="let item of items">
5. {{ item }}
6.

Here, *ngFor="let item of items" iterates over the items array and assigns each item to the local variable item. The content inside the
tag is repeated for each item in the array, and {{ item }} displays the value of each item.

7. **Result:** When you render the MyComponent component, Angular will dynamically generate HTML elements for each item in the items array, resulting in
8.
9. Item 1
10. Item 2
11. Item 3

The number of elements rendered will be equal to the number of items in the items array, and the content of each will display the corresponding item.

33. What do Angular's Template-driven and Reactive forms mean?

Angular 17 continues to support both Template-Driven and Reactive forms. Choosing between Template-Driven and Reactive forms largely depends on the specific requirements of your project and your personal or team's familiarity with Angular.

1. **Template-driven Forms:** They are the Angular way of leveraging HTML and its form elements to manage form data. Here, most of the form logic is handled by directives in the template itself. The FormsModule is essential here, enabling two-way data binding using ngModel to link domain model values to form input fields.

Example

```
/// Import FormsModule to enable template-driven forms
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    FormsModule
    // other imports...
  ],
  // other module properties...
})
export class AppModule {}

<!-- your-component.html -->
<form #userForm="ngForm">
  <input type="text" name="username" [(ngModel)]="user.username" required>
  <input type="email" name="email" [(ngModel)]="user.email" required>
  <button type="submit" [disabled]="!userForm.valid">Submit</button>
</form>
```

Here, ngModel binds the input fields to the user.username and user.email properties.

2. **Reactive Forms:** Here, the ReactiveFormsModule is used, and form controls are explicitly created in the component class. This approach leverages the FormControl, FormGroup, and FormArray classes to manage form data.

Example

```
// app.module.ts
// Import ReactiveFormsModule for reactive forms
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    ReactiveFormsModule
    // other imports...
  ],
  // other module properties...
})
export class AppModule { }

// your-component.ts
import { FormGroup, FormControl, Validators } from '@angular/forms';

export class YourComponent {
  userForm = new FormGroup({
    username: new FormControl('', Validators.required),
    email: new FormControl('', [Validators.required, Validators.email])
  });

  onSubmit() {
    console.log(this.userForm.value);
  }
}

<!-- your-component.html -->
<form [formGroup]="userForm" (ngSubmit)="onSubmit()">
  <input type="text" formControlName="username">
  <input type="email" formControlName="email">
  <button type="submit" [disabled]="!userForm.valid">Submit</button>
</form>
```

34. What kind of DOM is implemented by Angular?

Angular implements a dynamic and extensible Document Object Model (DOM) that is based on the standard DOM provided by the browser. This dynamic DOM is known as the Angular-specific DOM. This updates the entire tree structure of HTML tags until it reaches the data to be updated.

Below are some properties of this Angular-specific DOM:

- **Virtual DOM:** Angular uses a virtual representation of the DOM.
- **Template Syntax:** Angular templates are written using a declarative and expressive syntax that defines the structure and behavior of the user interface.
- **Data Binding:** Angular implements two-way data binding.
- **Directives and Components:** Angular extends HTML with custom directives and components that encapsulate behavior and presentation logic.

- **Change Detection:** Angular performs change detection to detect and propagate changes to the view.
- **Cross-Browser Compatibility:** Angular's DOM abstraction layer ensures cross-browser compatibility and consistency by providing a unified API.

35. Discuss your understanding of server-side rendering (SSR) and when you would consider using Angular Universal.

Server-side rendering (SSR) is a process that involves rendering pages on the server, resulting in initial HTML content that contains the initial page state. Once the HTML content is delivered to a browser, Angular initializes the application and utilizes the data contained within the HTML.

Working of SSR

1. An HTTP request is made to the server.
2. The server receives the request and processes all (or most of) the necessary code immediately.
3. The result is a fully formed and easily consumed HTML page that can be sent to the client's browser via the server's response.

Angular Universal

Angular Universal is a server-side rendering solution. It allows rendering the angular applications on the server before sending them to the client. The content of the application is available to first-time users instantly.

Here, are the scenarios for using Angular Universal:

- **Improved SEO:** By using Angular Universal to pre-render your application on the server side, you can ensure that search engines can crawl and index your content more effectively, improving your site's search engine optimization (SEO) and visibility.
- **Better Performance:** Rendering pages on the server side can lead to faster initial page loads and improved perceived performance for users, especially on devices with slower network connections or limited processing power.
- **Optimized Social Sharing:** Server-side rendering with Angular Universal ensures that the shared links include meaningful content and metadata, improving the appearance and usability of shared links on social media.
- **Accessibility and Progressive Enhancement:** Server-rendered pages provide a solid foundation for progressive enhancement, allowing you to enhance the user experience with client-side interactivity while ensuring a baseline level of functionality for all users.
- **Optimized Time to Interactive (TTI):** Server-side rendering can help reduce the time to interactive (TTI) for your application by pre-rendering the initial view on the server side and sending it to the client.
- **Improved Performance on Mobile Devices:** Mobile devices, especially those with limited processing power and network bandwidth, can benefit from server-side rendering to reduce the time and resources required to render pages.

* **What is an Observable in Angular, and why is it used with HTTP?**

In **Angular,** an **Observable** is a way to handle asynchronous operations which is a part of **RxJS (Reactive Extensions for JavaScript)**, which is a powerful library for handling streams of data.

When we make an HTTP request, the response isn't immediate. That's where Observables come in handy, as they allow us to "subscribe" to the response and act upon it once it arrives. With Observables, we can also handle multiple values over time, making it a flexible tool for **asynchronous** programming.

The reason **Angular** uses Observables for HTTP which makes angular more flexibility. With an Observable, We can manage multiple values, cancel requests, retry failed requests, and

chain operations, all with simple, clean syntax. We can also transform the data received using operators like **map() , filter() ,** and **tap()** , which are part of **RxJS**.

36. Explain RxJS Observables and their advantages over Promises for asynchronous operations.

RxJS is a framework for reactive programming that makes use of Observables, making it easy to write asynchronous code. This project is a kind of reactive extension to JavaScript with better performance, modularity, and debuggable call stacks while staying mostly backward-compatible, with some changes that reduce the API surface. RxJS is the official library used by Angular to handle reactivity, converting pull operations for call-backs into Observables.

Advantages of RxJS over promises for asynchronous operations:

- **Functional Reactive Programming (FRP):** RxJS follows the Functional Reactive Programming paradigm, which allows developers to work with streams of data over time.
- **Powerful Operators:** RxJS provides a wide range of operators that allow developers to manipulate, transform, and combine streams of data. These operators enable powerful data processing and manipulation workflows, making it easier to handle complex asynchronous scenarios.
- **Handling Complex Scenarios:** With RxJS, developers can handle complex asynchronous scenarios such as debounce, throttle, retry, and timeout with ease.
- **Lazy Evaluation:** RxJS uses lazy evaluation, i.e. operators are only executed when the observable is subscribed to.
- **Cancellation and Resource Management:** RxJS provides mechanisms for cancellation and resource management, which allows developers to clean up resources and cancel ongoing operations when they're no longer needed.
- **Integration with Angular:** RxJS is deeply integrated with Angular and is used extensively throughout the framework, especially in features like reactive forms, HTTP requests, and event handling.
- **Error Handling:** RxJS provides robust mechanisms, including operators for catching and handling errors within observables. This makes it easier to handle errors in asynchronous operations and recover gracefully from failures.

37. Describe your experience with state management libraries like NgRx or NgXS, highlighting their strengths and weaknesses in different contexts.

1. **NgRx**

NgRx is a powerful state management library for Angular, inspired by Redux. It follows a unidirectional data flow pattern and provides a centralized store to manage the application state.

Example

```
// Actions
export enum CounterActionTypes {
  Increment = '[Counter] Increment',
  Decrement = '[Counter] Decrement',
}

export class Increment implements Action {
  readonly type = CounterActionTypes.Increment;
}

export class Decrement implements Action {
  readonly type = CounterActionTypes.Decrement;
```

```
}

// Reducer
export function counterReducer(state: number = 0, action: CounterActions): number {
  switch (action.type) {
    case CounterActionTypes.Increment:
      return state + 1;
    case CounterActionTypes.Decrement:
      return state - 1;
    default:
      return state;
  }
}

// Store Setup
@NgModule({
  imports: [
    StoreModule.forRoot({ counter: counterReducer }),
  ],
})
export class AppModule {}

// Component
@Component({
  selector: 'app-counter',
  template: `
    <div>
      <button (click)="increment()">Increment</button>
      <span>{{ counter$ | async }}</span>
      <button (click)="decrement()">Decrement</button>
    </div>
  `,
})
export class CounterComponent {
  counter$: Observable<number>;

  constructor(private store: Store<{ counter: number }>) {
    this.counter$ = this.store.select('counter');
  }

  increment() {
    this.store.dispatch(new Increment());
  }

  decrement() {
    this.store.dispatch(new Decrement());
  }
}
```

2. **NgXs**

NgXs is a lightweight and developer-friendly state management library for Angular applications. It offers a straightforward setup and intuitive syntax.

Example

```
// State
@State<number>({
  name: 'counter',
  defaults: 0,
})
export class CounterState {}

// Actions
export class Increment {
  static readonly type = '[Counter] Increment';
}

export class Decrement {
  static readonly type = '[Counter] Decrement';
}

// Component
@Component({
  selector: 'app-counter',
  template: `
    <div>
      <button (click)="increment()">Increment</button>
      <span>{{ counter$ | async }}</span>
      <button (click)="decrement()">Decrement</button>
    </div>
  `,
})
export class CounterComponent {
  counter$: Observable<<number>;

  constructor(private store: Store) {
    this.counter$ = this.store.select(state => state.counter);
  }

  increment() {
    this.store.dispatch(new Increment());
  }

  decrement() {
    this.store.dispatch(new Decrement());
  }
}
```

Strengths and Weaknesses of Ngrx in different contexts:

Strengths:

- **Predictable State Management:** NgRx follows the Redux pattern, providing a predictable state management approach.
- **Single Source of Truth:** NgRx stores application state in a single immutable store, making it easier to maintain and debug complex applications.
- **Middleware Support:** NgRx supports middleware, allowing developers to intercept actions and perform asynchronous operations such as API calls or logging.

- **Integration with Angular:** NgRx is specifically designed for Angular applications, providing seamless integration with Angular's reactive programming model and dependency injection system.
- **Time-travel Debugging:** NgRx DevTools enables time-travel debugging, allowing developers to replay actions and inspect state changes at different points in time.

Weaknesses:

- **Boilerplate Code:** Implementing NgRx can lead to a significant amount of boilerplate code, especially for simple applications or features.
- **Learning Curve:** NgRx has a steep learning curve, especially for developers new to reactive programming concepts and the Redux pattern.
- **Complexity:** As applications grow in size and complexity, managing NgRx stores and actions can become challenging, leading to potential performance issues and codebase maintenance overhead.

Strengths and Weaknesses of NgXS in different contexts:

Strengths:

- **Simplicity:** NgXS aims to simplify state management in Angular applications by providing a lightweight and intuitive API for managing state.
- **Minimal Boilerplate:** NgXS reduces the amount of boilerplate code required compared to NgRx, making it easier to get started with state management.
- **Angular Integration:** Similar to NgRx, NgXS integrates well with Angular and leverages Angular's reactive programming model.
- **DevTools Support:** NgXS supports DevTools extensions, enabling developers to inspect state changes and debug applications more effectively.

Weaknesses:

- **Limited Middleware Support:** As compared to NgRx, NgXS has limited support for middleware, which may limit its capabilities for handling complex asynchronous operations.
- **Community and Ecosystem:** NgXS has a smaller community and ecosystem compared to NgRx, which may result in fewer third-party extensions, tools, and resources available for developers.
- **Scalability:** While NgXS is suitable for managing state in smaller to medium-sized applications, it may face scalability challenges in larger and more complex applications compared to NgRx.

38. Describe the Angular modules, services, and components.

1. **Angular modules**

An Angular module is a deployment sub-set of your whole Angular application. It's useful for splitting up an application into smaller parts and lazy load each separately, and to create libraries of components that can be easily imported into other applications. Modules are defined using the @NgModule decorator and typically contain declarations, imports, providers, and export arrays.

Example

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

@NgModule({
  declarations: [AppComponent, MyComboboxComponent,
                 CollapsibleDirective, CustomCurrencyPipe],
  imports: [BrowserModule],
  providers: [UserService, LessonsService]
```

```
})
export class ExampleModule {
}
```

In the above code,

- the @NgModule annotation is what defines the module
- The components, directives, and pipes that are part of the module are listed in the declarations array
- We can import other modules by listing them in the imports array
- We can list the services that are part of the module in the providers' array

2. **Angular Components**

Components are the core building pieces in Angular that manage a portion of the UI for any application. The @Component decorator is used to define a component. Every component comprises three parts:

1. a template that loads the component's view
2. a stylesheet that specifies the component's look and feel
3. a class that includes the component's business logic

3. **Angular Services**

Objects classified as services are those that are only instantiated once in the course of an application. A service's primary goal is to exchange functions and data with various Angular application components. To define a service, use the @Injectable decorator. Any component or directive can call a function specified inside a service.

39. What distinguishes JavaScript expressions from Angular expressions?

- **Context:** In Angular, the expressions are evaluated against a scope object, while the Javascript expressions are evaluated against the global window.
- **Forgiving:** In Angular expression evaluation is forgiving to null and undefined, while in Javascript undefined properties generate TypeError or ReferenceError.
- **No Control Flow Statements:** Loops, conditionals, or exceptions cannot be used in an angular expression
- **Security Restrictions:** Angular expressions have security restrictions to prevent code injection and execution of potentially harmful JavaScript code. This makes Angular expressions safer to use in templates.
- **Scope of Evaluation:** Angular expressions are evaluated within the context of Angular's templating engine, whereas JavaScript expressions are evaluated within the broader JavaScript runtime environment.
- **Syntax Differences:** While Angular expressions use similar syntax to JavaScript, there are some differences, such as the use of filters, template variables, and special Angular directives like ngIf and ngFor.

40. Describe Angular's dependency injection concept.

Dependency Injection is a design pattern that promotes the separation of concerns by decoupling components and their dependencies. In Angular, dependencies are typically services, but they also can be values, such as strings or functions. DI is used to inject instances of services, components, and other objects into classes that depend on them, promoting modularity, reusability, and testability within the application.

Implementing Angular Dependency Injection involves the following steps to set up and use services within your components.

1. **Create a Service**

First, create a service that will provide functionality or data to other components. You can use Angular CLI to generate a service:

```
ng generate service my-service
```

This will create a my-service.service.ts file. Open the file and define your service:

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

```
@Injectable({
  providedIn: 'root', // Provides the service at the root level
})
export class MyService {
  // Implement your service logic here
}
```

2. **Inject the Service into a Component**

Now, you can inject the service into a component that needs to use its functionality. Open the component file (e.g., app.component.ts) and inject the service through the constructor:

```
import { Component } from '@angular/core';
import { MyService } from './my-service.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  constructor(private myService: MyService) {
    // Use myService in the component
  }
}
```

3. **Register the Service in a Module**

Angular needs to know about the service and how to create an instance of it. Register the service in the providers array of an Angular module. If you want the service to be available throughout the entire application, use the root module (app.module.ts):

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { MyService } from './my-service.service';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [MyService], // Register the service here
  bootstrap: [AppComponent],
})
export class AppModule {}
```

4. **Use the Service in the Component**

Now that the service is injected into the component, you can use its methods and properties within the component:

```
import { Component } from '@angular/core';
import { MyService } from './my-service.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  constructor(private myService: MyService) {
```

```
    // Use myService in the component
    const data = this.myService.getData();
    console.log(data);
  }
}
```

In this example, assume that MyService has a method called getData().

41. Describe MVVM architecture concerning Angular.

MVVM is a variation of the traditional MVC (Model-View-Controller) software design pattern. Model-View-ViewModel (MVVM) architecture allows developers to divide their work into two categories: the development of the graphical user interface (the View) and the development of the business logic or back-end logic (the Model). This architecture eliminates the view's reliance on any one model platform. There are three components to the Angular MVVM architecture:

1. **Model:** It represents the business logic and data of a particular application. It consists of an entity structure. The model has the business logic, including model classes, remote and local data sources, and the repository.
2. **View:** the view embodies the visual layer of the application. Its primary role involves presenting the data sourced from the component and managing user interactions. Constructed through HTML templates, the view dynamically renders and adjusts its content according to the component's data and the application's logic.
3. **ViewModel:** It is an abstract layer of the application. A viewmodel handles the logic of the application. It manages the data of a model and displays it in the view. View and ViewModel are connected with two-way data binding. Hence, the ViewModel takes note of all the changes in the view and changes the appropriate data inside the model.

42. What is Change Detection, and how does the Change Detection Mechanism work?

Change Detection is the process of synchronizing a model with a view. It determines when and how to update the user interface based on changes in the application's data model.

For this, Angular uses a tree of change detectors to track changes in component properties and update the DOM accordingly. When a change occurs, Angular performs change detection, which involves checking each component's properties for changes and updating the DOM if necessary. The change detection mechanism is responsible for keeping the UI in sync with the application's data.

The mechanism moves only ahead and never backward, beginning with the root component and ending with the last component. Each component is a child, but the child is not a parent.

43. What are observables in Angular?

An observable is a declarative way to perform asynchronous tasks. One can imagine it as streams of data flowing from a publisher to a subscriber. An observable is a unique object just like a promise that is used to manage async requests. However, observables are considered to be a better alternative to promises as the former comes with a lot of operators that allow developers to better deal with asynchronous requests, especially if there is more than one request at a time.

Observables are not part of the JavaScript language so the developers have to rely on a popular Observable library called RxJS. The observables are created using the new keyword. They are only executed when subscribed to them using the subscribe() method. They emit multiple values over a while. They help perform operations like forEach, filter, and retry, among others. They deliver errors to the subscribers. When the unsubscribe() method is called, the listener stops receiving further values.

Example

```
import { Observable } from 'rxjs';
const observable = new Observable(observer => {
  setTimeout(() => {
    observer.next('This is a message from Observable!');
  }, 1000);
});
```

44. What does Angular Material mean?

Angular Material is a UI component library for Angular applications. It provides a set of pre-built and customizable UI components in the form of buttons, forms, navigation menus, and dialog boxes, that follow the Material Design guidelines. Angular Material simplifies the process of building consistent and visually appealing user interfaces in Angular. It offers a range of features and styles that can be easily integrated into Angular projects.

45. How can one create a service in Angular?

To create a service in Angular, go through the below steps:

1. **Generate a Service**
   - Run the following command in your Angular CLI.

   ng generate service my-service

   This command will create a new service file (my-service.service.ts) and a corresponding test file (my-service.service.spec.ts) in your Angular project.
2. **Define Service Logic**
   - Open the newly created service file (my-service.service.ts) in your code editor.
   - Define the logic and functionality of your service within the TypeScript class.

Example

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class MyService {
  constructor() { }
  // Define methods and properties for your service
  greet(): string {
    return 'Hello, Angular!';
  }
}
```

3. **Inject the Service**
   - After the service definition, you can inject it into Angular components, directives, or other services.
   - To inject the service, add it as a constructor parameter in the component where you want to use it.

Example

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

@Injectable({
  providedIn: 'root'
})
export class MyService {
  constructor() { }
  // Define methods and properties for your service
  greet(): string {
    return 'Hello, Angular!';
```

```
  }
}
```

4. **Provide the Service**
    - You can provide the service at a specific module level or component level instead of its default availability throughout the application by specifying the module or component name in the providedIn property of the @Injectable decorator.

Example

```
@Injectable({
  providedIn: 'root' // or 'any specific module or component'
})
```

Advanced Angular Interview Questions for Senior Developers (6 to 10 Years)

46. Discuss your experience with continuous integration and continuous delivery (CI/CD) pipelines for Angular projects and best practices.

I'll share my experiences in the form of step by step process you need to follow for continuous integration and continuous delivery (CI/CD) pipelines for Angular projects.

1. **Setup Automation:** Automate the build, test, and deployment processes for Angular projects using CI/CD tools like GitHub Actions.
2. **Version Control Integration:** Integrate CI/CD pipelines with version control systems like Git to trigger builds automatically whenever changes are pushed to the repository.
3. **Build Process:** Configure the CI pipeline to build the Angular application from the source code. Use tools like Angular CLI for building and packaging the application artifacts.
4. **Testing:** Incorporate unit tests, integration tests, and end-to-end tests into the CI pipeline to ensure code quality and reliability.
5. **Static Code Analysis:** Include static code analysis tools like ESLint, TSLint, or SonarQube in the CI pipeline to identify code quality issues, coding standards violations, and potential bugs.
6. **Artifact Management:** Publish build artifacts and dependencies to artifact repositories like Nexus or Artifactory for versioning and dependency management.
7. **Deployment Strategies:** Implement different deployment strategies such as blue-green deployments, canary releases, or rolling deployments to minimize downtime and mitigate risks during deployment.
8. **Environment Configuration:** Manage environment-specific configurations using environment variables or configuration files.
9. **Monitoring and Logging:** Integrate monitoring and logging solutions into CI/CD pipelines to track build and deployment status, monitor application health, and troubleshoot issues.
10. **Security Scans:** Include security scanning tools like OWASP Dependency-Check or Snyk in the CI pipeline to identify and remediate security vulnerabilities in third-party dependencies.

Best Practices

- **Pipeline as Code:** Define CI/CD pipelines using infrastructure as code (IaC) principles to version control pipeline configurations and ensure reproducibility.
- **Incremental Builds:** Optimize build times by implementing incremental builds and caching dependencies to avoid rebuilding unchanged code.
- **Feedback Loop:** Establish a feedback loop by integrating automated notifications and alerts to notify developers of build failures, test errors, or deployment issues.
- **Immutable Infrastructure:** Treat infrastructure components and deployment artifacts as immutable to ensure consistency and repeatability across environments.

- **Continuous Improvement:** Continuously monitor and optimize CI/CD pipelines by analyzing build metrics, identifying bottlenecks, and implementing performance improvements.
- **Documentation:** Document CI/CD pipeline configurations, deployment processes, and best practices.

47. What are Angular router links?

Router links in Angular are used for navigation within an application. They are defined using the routerLink directive and allow us to navigate to different routes or components. Router links can be used in HTML templates and are generally placed on anchor <a> tags or other clickable elements. By specifying the destination route or component, router links allow users to navigate between different parts of an Angular application.

Example

```
<nav>
  <a routerLink="/home" >Home Page of our website</a>
  <a routerLink="/about-us" >About us</a>
</nav>
<router-outlet></router-outlet>
```

48. How do you create directives using CLI?

For creating directives in Angular using CLI, follow the below step-by-step procedure:

1. Open Terminal or Command Prompt
2. Navigate to your Angular project directory: Use the cd command to navigate to your Angular project directory where you want to create the directive.
3. Run the Angular CLI command: Use the ng generate directive command followed by the name of your directive to generate the directive files.
   ng generate directive directive-name
   or
   ng g d directive-name
4. Verify the directive creation: After running the command, the Angular CLI will generate the necessary files for your directive, including the directive class file, and the directive test file, and it will update the appropriate module file to import the directive.

49. What exactly is a parameterized pipe?

In Angular, a parameterized pipe is a pipe that takes one or more parameters, which are also referred to as arguments. Pipes are used in Angular templates to change data; parameterized pipes let you adjust the transformation according to certain needs. A pipe's behaviour can be changed and various data transformations can be applied by handling its parameters.

50. What is multicasting in Angular?

Multicasting or Multi-casting is the practice of broadcasting to a list of multiple subscribers in a single execution. 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.

Example

```
var source = Rx.Observable.from([7, 8, 9]);
var subject = new Rx.Subject();
var multicasted = source.multicast(subject);
// These are, under the hood, `subject.subscribe({...})`:
multicasted.subscribe({
  next: (v) => console.log('observerA: ' + v)
});
multicasted.subscribe({
```

```
  next: (v) => console.log('observerB: ' + v)
});
```

51. What will happen if you do not supply a handler for an observer?

If you don't supply a handler for a notification type, the observer just ignores notifications of that type. Angular components or services subscribing to the observable without a handler won't be affected by the lack of handling logic. The subscription will still be established, but no action will be taken when the observable emits values or completes.

If you subscribe to an observable in Angular without providing a handler for the observer and you don't unsubscribe from the observable, it can potentially lead to memory leaks. This is because the subscription will keep a reference to the observable, preventing it from being garbage-collected.

52. Share your knowledge of upcoming Angular features and how you would utilize them in your projects.

- **Ivy Renderer Improvements:** Ivy is Angular's next-generation renderer, which brings significant performance improvements, smaller bundle sizes, and better debugging capabilities. As Ivy continues to evolve, leveraging its features can lead to faster rendering times, improved application performance, and easier debugging in Angular projects.
- **Strict Mode:** Angular's strict mode aims to provide stricter type checking and improved developer experience. It enforces more rigorous typing rules, eliminates certain runtime errors, and encourages better coding practices. Adopting a strict mode can enhance code quality, reduce bugs, and make Angular applications more maintainable and scalable.
- **Component Test Harnesses:** Angular Component Test Harnesses provide a set of utilities for testing Angular components in isolation. These harnesses offer standardized APIs for interacting with Angular components in unit tests, simplifying the testing process and improving test reliability. Utilizing component test harnesses can streamline the testing workflow and enhance the overall test coverage of Angular applications.
- **Improved CLI Features:** The Angular CLI (Command Line Interface) continues to receive updates and new features aimed at improving developer productivity and project maintainability. Features such as enhanced code generation, better build optimizations, and improved project scaffolding can help developers streamline their workflow and build more robust Angular applications.
- **Official State Management Solutions:** These solutions could provide standardized patterns and best practices for managing complex application states in Angular projects.
- **Integration with Web Components:** As the adoption of Web Components grows, Angular is likely to continue improving its support for integrating with Web Components. This includes features such as seamless interoperability between Angular components and Web Components, improved encapsulation, and better performance optimizations.

53. Explain your approach to implementing and managing state in large Angular applications. Discuss the advantages and disadvantages of different state management libraries.

Approaches to implement and manage state in large Angular applications:

1. **Component State Management:** Here, each Angular component manages its state using component properties and two-way data binding.
2. **Service-Based State Management:** Angular services can be used to manage application-wide state by storing and providing access to shared data and stateful logic.

3. **RxJS Observables and Subjects:** Observables and subjects can be used to create streams of data that represent the application state and propagate changes throughout the application. Reactive programming enables declarative and composable state management.
4. **State Management Libraries:** They offer centralized and predictable state management solutions based on well-established patterns like Redux. These libraries provide patterns and utilities for managing complex application states, including features like actions, reducers, selectors, and effects.

Advantages and Disadvantages of State Management Libraries

State Management Libraries	Advantages	Disadvantages
NgRx	Robust architecture, extensive tooling, scalability, support for complex scenarios, and a large community	Steeper learning curve, higher boilerplate code, and potentially increased complexity for smaller projects.
NgXs	Lightweight, developer-friendly, shallow learning curve, seamless integration with Angular, and suitability for smaller to medium-sized projects.	Fewer advanced features compared to NgRx and a smaller community compared to NgRx
Akita	Simplicity, flexibility, built-in entity management, ease of use, and suitability for small to medium-sized projects.	Relatively smaller community compared to NgRx, and fewer advanced features compared to NgRx

54. What is the state() function in Angular?

Angular's state() function is used to define different states to call at the end of each transition. The state() function takes two arguments:

1. a unique name like open or closed
2. style() function

Example

```
state('open', style({
  height: '100px',
  opacity: 0.8,
  backgroundColor: 'yellow'
})),
```

55. What are macros in Angular?

In Angular, the AOT compiler supports macros in the form of functions or static methods that return an expression in a single return expression.

Example

```
export function wrapInArray(value: T): T[] {
  return [value];
}
```

- You can use it inside metadata as an expression

```
@NgModule({
  declarations: wrapInArray(TypicalComponent),
})
export class TypicalModule {}
```

- The compiler treats the macro expression as it is written directly

```
@NgModule({
  declarations: [TypicalComponent],
})
export class TypicalModule {}
```

Scenario Based Angular Interview Questions

56. How do you deal with errors in observables?

Below are some of the best practices to deal with errors in observables:

1. **Use the catchError Operator:** An Observable stream's failures may be detected and handled with the catchError operator.

Example

```
import { catchError } from 'rxjs/operators';
this.httpClient.get('/api/data')
  .pipe(
    catchError((error: any) => {
      // Handle the error here
      console.error('An error occurred:', error);
      // Optionally, re-throw the error or return a default value
      return throwError('Something went wrong');
    })
  )
  .subscribe(
    (response) => {
      // Handle the successful response
    },
    (error) => {
      // This block will only execute if catchError is used
      console.error('Error handler:', error);
    }
  );
```

2. **Centralize Error Handling:** Make a universal error-handling service that can be injected into other services and components.

Example

```
import { Injectable } from '@angular/core';
import { throwError } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
export class ErrorHandlerService {
  handle(error: any): void {
    // Log the error, send it to a remote service, or perform other actions
    console.error('An error occurred:', error);
    // Optionally, re-throw the error or return a default value
    throwError('Something went wrong');
  }
}
```

3. **Provide Meaningful Error Messages:** Avoid exposing sensitive information and use descriptive error messages that guide developers and users in understanding the issue.
4. **Logging Errors:** Angular provides a logging mechanism, and you can use libraries like ngx-logger for more advanced logging features.

Example

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

@Injectable({
```

```
  providedIn: 'root'
})
export class LoggerService {
  logError(message: string, error: any) {
    console.error(message, error);
  }
}
```

57. Your Angular application is experiencing slow loading times. You need to identify the bottleneck and optimize performance. How would you approach this?

The following is a systematic approach for addressing slow loading times and optimizing the performance of the Angular application:

1. **Performance Profiling:**
   - Use browser developer tools (e.g., Chrome DevTools) to profile the application's loading time, network requests, rendering performance, and memory usage.
   - Look for long-running tasks, excessive network requests, large asset sizes, and inefficient JavaScript execution.
2. **Network Optimization:**
   - Minimize the number of HTTP requests by combining and compressing CSS and JavaScript files.
   - Enable server-side compression (e.g., gzip) to reduce the size of transferred data.
   - Leverage HTTP/2 for multiplexing and parallelism of requests to improve loading times.
3. **Bundle Optimization:**
   - Use Angular CLI's production build mode (ng build --prod) to enable optimizations like code minification, tree-shaking, and dead code elimination.
   - Analyze bundle sizes using tools like Webpack Bundle Analyzer to identify large dependencies and optimize imports.
   - Consider code splitting to create smaller bundles and load only necessary code chunks on demand.
4. **Rendering Performance:**
   - Optimize Angular templates by minimizing DOM manipulations, avoiding excessive ngFor loops, and reducing the number of bindings.
   - Use trackBy function with ngFor to improve rendering performance by providing a unique identifier for each item in the iterable.
   - Implement OnPush change detection strategy for components to reduce change detection cycles and improve rendering performance.
5. **Caching and Prefetching:**
   - Implement caching strategies using HTTP caching headers (e.g., Cache-Control) to cache static assets and API responses.
   - Use service workers to enable client-side caching and offline capabilities for static assets and API requests.
   - Prefetch critical resources using thetag to reduce perceived loading times for subsequent navigations.
6. **Third-Party Libraries and Plugins:**
   - Evaluate the performance impact of third-party libraries and plugins used in the application.
   - Consider replacing or optimizing heavy dependencies with lighter alternatives or custom solutions where applicable.
7. **Monitoring and Continuous Improvement:**

- Implement performance monitoring and analytics tools (e.g., Google Analytics, New Relic) to track key performance metrics and identify performance regressions over time.
- Set up automated performance tests and benchmarks to detect performance regressions during development and deployment.

58. You're building a complex data-driven application with multiple components needing access to a shared state. How would you choose and implement an effective state management strategy?

My approach to implementing a solid state management strategy for an application with multiple components needing access to a shared state will be:

1. **Analyze Requirements:**
   - Understand the complexity and scale of the application.
   - Identify the types of data and states that need to be managed.
   - Determine how state changes propagate across components and modules.
2. **Evaluate State Management Options and Choose a Suitable One:**
   - Service-based State: Angular services can be used to manage shared state across components by storing data and providing methods to access and update state.
   - RxJS Observables and Subjects: Leverage RxJS for reactive programming and use observables and subjects to create streams of data representing the application state.
   - State Management Libraries: Consider third-party state management libraries like NgRx, Akita, or Ngxs for managing complex application state using patterns like Redux.
3. **Implement the Chosen Approach:**
   - Design stateful services to encapsulate shared state and provide methods for reading and updating state.
   - Use observables and subjects to propagate state changes and trigger updates across components.
   - Leverage Angular's dependency injection mechanism to inject stateful services into components and modules.
   - Implement patterns like actions, reducers, selectors, and effects if using a state management library like NgRx.
   - Follow best practices for organizing state logic, separating concerns, and optimizing performance.
4. **Test and Iterate:**
   - Write comprehensive unit and integration tests to validate the correctness and reliability of state management implementations.
   - Monitor application performance and behavior using browser developer tools and performance profiling tools.
   - Gather feedback from users and stakeholders to identify pain points and areas for improvement.
   - Continuously iterate and refactor state management logic based on evolving requirements and performance metrics.

59. You need to integrate a complex third-party library with your Angular application. How would you ensure seamless integration and maintainability?

1. **Research and Evaluation:**
   - Thoroughly research the third-party library's documentation, features, compatibility with Angular, and community support.
   - Evaluate the library's suitability for your project based on its capabilities, performance, licensing, and support.
2. **Dependency Management:**

- Use a package manager like npm or yarn to install the third-party library and manage its dependencies.
- Ensure that the library's version is compatible with your Angular project's version and other dependencies.

3. **Angular Component Wrapper:**
   - Whenever possible, create Angular component wrappers around the third-party library's components to encapsulate functionality and ensure Angular compatibility.
   - Implement Angular lifecycle hooks, input and output properties, and event handling to seamlessly integrate third-party components into your Angular application.
4. **Modularization and Lazy Loading:**
   - Consider modularizing the integration by creating feature modules dedicated to the third-party library's functionality.
   - Implement lazy loading for modules containing the third-party library's components to improve initial loading times and reduce bundle sizes.
5. **Error Handling and Debugging:**
   - Implement robust error-handling mechanisms to gracefully handle errors and edge cases arising from the integration.
   - Use browser developer tools and logging frameworks to debug integration issues and troubleshoot runtime errors effectively.
6. **Documentation:**
   - Document the integration process, including setup instructions, configuration options, usage examples, and troubleshooting tips.
7. **Version Control and Updates:**
   - Regularly update the third-party library to newer versions to leverage bug fixes, performance improvements, and new features.
   - Use version control systems like Git to track changes and updates to the integration codebase and revert changes if necessary.
8. **Testing and Quality Assurance:**
   - Implement comprehensive unit tests, integration tests, and end-to-end tests to validate the functionality and behaviour of the integrated components.

60. What happens when we use the script tag within a template?

Angular recognizes the value as unsafe and automatically sanitizes it, which removes the script tag but keeps safe content such as the text content of the script tag. This way it eliminates the risk of script injection attacks. If you still use it then it will be ignored and a warning appears in the browser console.

Issues while developing:

1. Port 4200 is already in use. Use '--port' to specify a different port.

Solution 1 : to get process Id associated with port no 4200
Use admin cmd prompt : netstat -a -n -o
Again use cmd prompt : taskkill -f /pid 18932

Solution 2 : restart application / VS Code / System

2. What is authentication and Authorization? how do you implement it with APIs .net or python and integrate it with angular?
3. How to implement JWT token authentication in angular? https://freeapi.miniprojectideas.com/index.html SWAGer
4. How to add/Integrate bootstrap in to Angular project?
5. How to convert ratings value(i.e. 4.5) into stars and vice versa
6. Async pipe ? not discussed
7. How to show database records into angular gridview and apply searching sorting update delete on grid itself?
8. How to add calendar control to textbox using angular?
9. How to send multiple values from parent to child and vice versa, as well as how to send object?
10. How to handle sessions in angular?