Basics

What is a Framework?

Frameworks are foundational structures that provide developers a head start in creating applications. They eliminate the need to start coding from scratch, allowing developers to focus on solving specific problems and adding unique features. One such web framework designed specifically for web applications is Angular. It provides a set of predefined guidelines and structures, streamlining the development process. This not only saves time but also improves the efficiency of developing web applications.

What is Client-Server Communication?

Client-server communication is the fundamental mechanism that enables web applications to interact with servers. The process involves:

1. **Sending Requests:** The client, typically a web browser, initiates an HTTP request to the server.
2. **Server Processing:** The server processes the HTTP request, performing operations like retrieving data from a database or executing specific tasks.
3. **Sending Responses:** The server sends an HTTP response back to the client, providing the requested data or conveying the success or failure of the operation.

What is JavaScript?

JavaScript is a versatile programming language with several key attributes:

- **Dynamically Typed:** Developers do not need to declare explicit data types, offering flexibility but requiring caution to prevent runtime errors.
- **Object-Oriented:** Supports objects, classes, and inheritance, enhancing code organisation and reusability.
- **Event-Driven:** Responds to events triggered by user actions or other components, enabling interactive and user-friendly web applications.
- **Cross-Browser Compatible:** Supported by all major web browsers like Chrome, Firefox, Safari, and Edge, ensuring consistent user experiences across platforms.

What is TypeScript?

TypeScript is a superset of JavaScript that introduces static typing to enhance development:

- **Static Typing:** Developers can define variable types, function parameters, and return types, allowing the TypeScript compiler to catch type-related errors during development.

- **Error Prevention:** By identifying errors early, TypeScript enhances code quality and stability, especially in large-scale projects.
- **Integration with Frameworks:** TypeScript is often integrated with modern frameworks like Angular, supporting a more robust and maintainable codebase.

Angular

Definition:

- Not a Programming Language: Angular is a comprehensive framework for building dynamic single-page applications (SPAs).
- **Framework:** Provides a structured environment with predefined rules, enabling developers to write efficient and maintainable code.

Framework Characteristics:

- **Structured Approach:** Enforces guidelines and patterns for scalable applications.
- **Component-Based:** Breaks down complex applications into reusable components.
- **Code Reusability:** Promotes creating reusable components to save development time.

Web Development:

- **Building Websites:** Simplifies creating dynamic and interactive websites.

Single-Page Applications (SPAs):

- Comparison with Multi-Page Applications: Unlike multi-page applications, SPAs load a single HTML page and dynamically update content without requiring full page reloads.
- **Advantages:**
  - Smoother user experience with faster navigation.
  - Reduced server load by transmitting only data rather than entire HTML pages.

Practical Applications of Angular:

- Customer Relationship Management (CRM) systems
- Travel and booking platforms
- E-commerce platforms
- Healthcare applications
- Real-time chat applications

History and Versions of Angular

AngularJS (Version 1.x):

- Release Date: 2010
- **Key Features:**
  - Two-way data binding for automatic synchronization.
  - Directives for creating reusable components.

Angular 2+:

- Release Date: 2016 onwards
- **Key Changes:**
  - Complete rewrite with a component-based architecture.
  - Transitioned to TypeScript for enhanced maintainability.
  - Introduced reactive programming with RxJS.

Current Version (Angular 17.0.0):

- Release Date: November 8, 2023
- **Key Features:**
  - Improved performance and security.
  - Enhanced support for modern web standards.

Why Use Angular Instead of AngularJS?

- **Performance:** Angular introduces a more efficient change detection mechanism compared to AngularJS.
- **Mobile Responsiveness:** Angular is optimized for mobile-friendly web applications.
- **TypeScript Integration:** Provides static typing benefits, enhancing maintainability and reducing errors.

What is a Single-Page Application (SPA)?

- **Definition:** A type of web application that dynamically rewrites the current page instead of loading entire new pages from the server.
- **Operational Mechanism:**
  - **Initial Loading:** Loads essential resources (HTML, CSS, JavaScript) at the start.
  - **Subsequent Interactions:** Updates the page dynamically using asynchronous requests without full page reloads.
- **Advantages:**
  - Faster user experience and smooth navigation.
  - Reduced server load.

Technologies Used:

SPAs are commonly built using frameworks like Angular, React, or Vue.js.

Use Cases:

- Rich web applications such as social media platforms, interactive dashboards, and real-time collaboration tools.

Stay Consistent

2.Setup and Installation

1. IDE: Visual Studio Code

1. Download VS Code from its official website: Visual Studio Code Download.
2. Follow the steps shown in the installation guide or video tutorial.

2. Node.js Installation

1. Download Node.js from its official website: Node.js Download (Download the latest LTS version).
2. Verify the installation by opening the Command Prompt and running the command:
3. node --version

3. Installing Angular CLI

1. Open a terminal or command prompt and run the following command to install Angular CLI globally:
2. npm install -g @angular/cli
3. Verify the installation by running the command:
4. ng --version

4. Adjusting PowerShell Execution Policy (Windows Only)

By default, Windows restricts the execution of PowerShell scripts. To enable the execution of PowerShell scripts, follow these steps:

1. Open the Command Prompt and run the command:
2. set-ExecutionPolicy RemoteSigned -Scope CurrentUser
3. Verify the execution policy by running:
4. Get-ExecutionPolicy
5. To view the list of policies, run:
6. Get-ExecutionPolicy -List

5. Creating a Workspace and Running the Application

1. Create a Workspace:
2. ng new my-app
3. Navigate to the Project Folder:
4. cd my-app
5. Run the Application:
6. ng serve

Stay Consistent

Angular Architecture

Modules

- Angular applications are organized into modules, which act as containers for different parts of the application, helping in organizing and separating functionality.
- Each module can have its own components, services, and other features, making the application modular and manageable.

Components

- Components are the basic building blocks of Angular applications.
- Each component consists of:
  - A TypeScript class that defines the component's behavior.
  - An HTML template that defines its view.

Templates

- Templates are written in HTML and define the view of a component.
- They include Angular-specific syntax to bind data and listen to events, enabling dynamic updates.

Directives

- Directives are markers on a DOM element that instruct Angular to attach a specific behavior to it.
- Built-in Directives:
  - *ngIf: For conditional rendering.
  - *ngFor: For iterating over lists.

Services

- Services are used to organize and share code across components.
- They are singleton objects that can be injected into components using Angular's Dependency Injection (DI) mechanism.
- Common use cases include data fetching, business logic, and state management.

Router

- Angular's Router Module enables navigation between components and views in a single-page application (SPA).
- Features include:
  - Route parameters.
  - Lazy loading.
  - Guards for access control.

Forms

- Angular provides a powerful Forms Module for handling user input and form validation.
- Two approaches:
  - Template-driven forms: Suitable for simple use cases.
  - Reactive forms: Ideal for complex forms and dynamic validation.

Angular CLI (Command Line Interface)

- The Angular CLI is a command-line tool that simplifies development tasks.
- Key Features:
  - Project Scaffolding: Quickly create the basic structure of an Angular application, including modules, components, services, and other necessary files.
  - Code Generation: Automates the creation of components, services, modules, and directives, reducing boilerplate code.
  - Build and Deployment: Provides commands to create production-ready bundles of the application for deployment to servers or cloud platforms.

Angular Imports

- The import keyword in TypeScript is used to bring functionalities from other files or modules.
- Syntax: import { Symbol } from 'module';
- Examples:
  - Importing Angular core modules:
  - import { Component, NgModule } from '@angular/core';
  - Importing custom modules and services:
  - import { MyService } from './my-service';
  - import { SharedModule } from '../shared/shared.module';
- Imports are essential for creating modular, organized, and reusable code in Angular.

Angular Decorators and Selectors

- Decorators are special declarations in TypeScript used to configure and enhance classes or their members in Angular.
- Common Angular Decorators:
  - @Component: Defines a component and its metadata.
  - @Directive: Defines a directive and its metadata.
  - @Injectable: Defines a service and its metadata for dependency injection.
  - @NgModule: Defines a module and its metadata.

Example: Component Decorator

```
@Component({

 selector: 'app-root',

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

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

})

export class AppComponent {}

- Selector: Specifies the HTML tag used to represent the component, e.g., <app-root> </app-root>.
- Selectors should be unique within the application to avoid conflicts.

Angular Classes

- Classes act as blueprints for creating objects, encapsulating properties and behaviors.
- Example:
- export class Person {
- age: number = 25;
-
- celebrateBirthday(): void {
- this.age++;
- }
- }

Angular Modules

- Modules are containers that group related parts of the application together.
- Benefits:
  - Organization: Helps keep code organized by grouping related features.
  - Reusability: Modules can be reused across different parts of the application or in other projects.
  - Maintainability: Smaller modules are easier to maintain compared to a large monolithic codebase.
  - Dependency Management: Simplifies the management of dependencies between parts of the application.

Angular Components

- Components encapsulate logic and views related to specific parts of the application.
- They are reusable, making it easier to manage and maintain complex UIs.
- Example:
- @Component({
- selector: 'app-header',
- templateUrl: './header.component.html',
- styleUrls: ['./header.component.css']
- })
- export class HeaderComponent {}

Stay Consistent

Standalone Components

Standalone components provide a simplified way to build Angular applications. Standalone components, directives, and pipes aim to reduce the need for NgModules.

- Components, directives, and pipes can now be marked as standalone: true.
- Angular classes marked as standalone do not need to be declared in an NgModule (the Angular compiler will report an error if you try).
- Standalone components specify their dependencies directly instead of getting them through NgModules.
- For example, if Component 1 is a standalone component, it can directly import another standalone component, Component 2.

Variables: A variable is a named storage location (a memory location) that holds a value. Variables are used to store and manage data within a program. Each variable has a unique identifier (its name) that allows developers to reference and manipulate the stored value.

Data Types:

Primitive Data Types:

- Primitive data types are simple and directly hold a value.
- The values of primitive types are stored directly in memory locations.
- Operations on primitive types are generally faster because they involve working directly with the values stored.
- **Number:** Represents numeric values, including integers and floating-point numbers.
  - Example: let count: number = 42;
- **Boolean:** Represents true or false values.
  - Example: let isActive: boolean = true;
- **String:** Represents textual data enclosed in single or double quotes.
  - Example: let name: string = "ABC";
- **Null:** A special value that represents the intentional absence of any object value.
  - Example: let myVar: null = null;
- **Undefined:** A value that represents a variable that has been declared but not assigned a value.
  - Example: let myVar: undefined = undefined;

Non-Primitive (Reference) Data Types:

- Non-primitive data types store references to memory locations where the data is stored.
- They are more complex data structures compared to primitives.
- **Array:** Represents an ordered list of values of the same type or a combination of types.
  - Example: let numbers: number[] = [1, 2, 3];
- **Object:** Represents a collection of key-value pairs where values can be of any data type.

    - Example: let person: { name: string, age: number } = { name: "ABC", age: 25 };
- **Any:** Represents a variable that can hold values of any data type.
    - Example:
    - let dynamicValue: any = 5;
    - dynamicValue = "Hello";
    - dynamicValue = true;
- **Void:** Represents the absence of a type. Often used as the return type of functions that do not return a value.
    - Example:
    - function logMessage(): void {
    - console.log("Hello, world!");
    - }

Global and Local Variables:

- **Global Variables:** Declared outside of any function or block and have global scope.
    - Accessible throughout the entire program.
    - Lifetime extends throughout the entire program.
- **Local Variables:** Declared within a function or block and have local scope.
    - Accessible only within the function or block where they are declared.
    - Lifetime is limited to the execution context of the function or block.

Example:

```
function example() {

  if (true) {

    var a = 20;

    console.log(a); // Output: 20

  }

  console.log(a); // Output: 20

}
```

Var, Let, and Const:

- **Var:**
    - Function-scoped, not block-scoped.
    - Values can be reassigned.
    - Example:
    - var a = 10;
    - if (true) {
    - var a = 20;
    - console.log(a); // Output: 20
    - }

    - console.log(a); // Output: 20
- **Let:**
    - Block-scoped.
    - Values can be reassigned.
    - Example:
    - let b = 10;
    - if (true) {
    - let b = 20;
    - console.log(b); // Output: 20
    - }
    - console.log(b); // Output: 10
- **Const:**
    - Block-scoped.
    - Values cannot be reassigned.
    - Example:
    - const c = 10;
    - c = 20; // Error: Assignment to constant variable.

Stay Consistent

Operators

Arithmetic Operators

A mathematical function that takes two operands and calculates anything with them is called an arithmetic operator. It performs the following calculations:

1. Addition (+): Adds two values.
2. let sum = 5 + 3; // sum is 8
3. Subtraction (-): Subtracts the right operand from the left operand.
4. let difference = 10 - 4; // difference is 6
5. Multiplication (*): Multiplies two values.
6. let product = 2 * 3; // product is 6
7. Division (/): Divides the left operand by the right operand.
8. let quotient = 8 / 2; // quotient is 4
9. Modulus (%): Returns the remainder of the division of the left operand by the right operand.
10. let remainder = 9 % 4; // remainder is 1
11. Increment (++), Decrement (--): Increases or decreases the value of a variable by 1.
12. let x = 5;
13. x++; // x is now 6
14. x--; // x is back to 5

Comparison Operators

Comparison operators are used for evaluations and comparisons of strings or integers. Comparison expressions return either true or false.

1. Less Than (<): Checks if the left operand is less than the right operand.
2. let isLess = 3 < 5; // isLess is true
3. Greater Than (>): Checks if the left operand is greater than the right operand.
4. let isGreater = 7 > 4; // isGreater is true
5. Less Than or Equal To (<=): Checks if the left operand is less than or equal to the right operand.
6. let isLessOrEqual = 5 <= 5; // isLessOrEqual is true
7. Greater Than or Equal To (>=): Checks if the left operand is greater than or equal to the right operand.
8. let isGreaterOrEqual = 8 >= 8; // isGreaterOrEqual is true
9. Not Equal (!==): Checks if the left operand is not equal to the right operand.
10. let notEqual = 2 !== 4; // notEqual is true
11. Equal (==): Checks if the left operand is equal to the right operand (type coercion may occur).
12. let isEqual = '2' == 2; // isEqual is true

Logical Operators

1. Logical AND (&&): Returns true if both operands are true.
2. let andResult = (5 > 3) && (2 < 4); // andResult is true
3. Logical OR (||): Returns true if at least one operand is true.
4. let orResult = (5 < 3) || (2 > 1); // orResult is true
5. Logical NOT (!): Returns the opposite boolean value of the operand.
6. let notResult = !(4 > 2); // notResult is false

Assignment Operators

A variable can be assigned a value using assignment operators. The value on the right side needs to be of the same data type as the variable on the left side.

1. Assignment (=): Assigns the value of the right operand to the left operand.
2. let x = 20; // x is assigned the value 20
3. Addition Assignment (+=): Adds the right operand to the left operand and assigns the result to the left operand.
4. let y = 5;
5. y += 3; // y is now 8
6. Subtraction Assignment (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.
7. let z = 7;
8. z -= 2; // z is now 5

Ternary Operator

The ternary operator requires three operands. It offers a way to shorten a simple if-else block.

1. Ternary (condition ? true : false):If the condition is true, it returns the true expression; otherwise, it returns the false expression.
2. let age = 20;
3. let status = (age >= 18) ? 'Adult' : 'Minor';
4. // status is 'Adult' because age is 20

Bitwise Operators

1. Bitwise AND (&): Performs a bitwise AND operation.
2. let result = 5 & 3; // result is 1 (binary: 101 & 011)
3. Bitwise OR (|): Performs a bitwise OR operation.
4. let result = 5 | 3; // result is 7 (binary: 101 | 011)
5. Bitwise XOR (^): Performs a bitwise XOR (exclusive OR) operation.
6. let result = 5 ^ 3; // result is 6 (binary: 101 ^ 011)
7. Bitwise NOT (~): Flips the bits of its operand.
8. let result = ~5; // result is -6 (bitwise NOT of 5)
9. Left Shift (<<): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
10. let result = 5 << 1; // result is 10 (binary: 101 << 1)
11. Right Shift (>>): Shifts the bits of the left operand to the right by the number of positions specified by the right operand.
12. let result = 5 >> 1; // result is 2 (binary: 101 >> 1)

Loops

If-Else Statement

The if-else statement is used for conditional execution.

Syntax:

```
if (condition) {

   // Code to be executed if the condition is true

} else {

   // Code to be executed if the condition is false

}
```

Example:

```
let age: number = 20;

if (age >= 18) {
```

```
    console.log('You are an adult.');
} else {
    console.log('You are a minor.');
}
```

For Loop

The for loop is used for iterating over a block of code multiple times.

Syntax:

```
for (initialization; condition; iteration) {
    // Code to be executed in each iteration
}
```

Example:

```
for (let i = 0; i < 5; i++) {
    console.log('Iteration:', i);
}
```

While and Do-While Loops

While Loop:

```
let i = 0;
while (i < 10) {
    console.log('Iteration:', i);
    i++;
}
```

Do-While Loop:

```
let i = 0;
do {
    console.log('Iteration:', i);
```

```
    i++;

} while (i < 2);
```

Practice Questions

Operators:

1. **Arithmetic Operations:**
   - Declare two numbers and check their sum, difference, product, and quotient.
2. **Comparison Operations:**
   - Compare two numbers and check whether the first number is greater than, equal to, or less than the second number.
3. **Logical Operations:**
   - Declare two numbers and check if a number is less than 20 or greater than 40.
4. **Assignment Operations:**
   - Create a variable that uses the += operator to update a variable by adding 5 to its current value.
5. **Ternary Operator:**
   - Declare two numbers and check if a number is positive or negative using the ternary operator.
6. **Bitwise Operators:**
   - Declare two numbers and perform bitwise AND, OR, and XOR operations on two numbers; check the results.

If-Else:

1. Take a number and check whether it's positive, negative, or zero.
2. Declare three variables and check which one is the largest.
3. Declare a variable and check if it is an even or odd number.

For Loops:

1. Print numbers from 1 to 10.
2. Print the even or odd numbers from 1 to 20.
3. Calculate the sum of 5 digits using a for loop.

While Loops:

1. Using a while loop, check the count from 1 to 5.
2. Get the sum of the first 10 digits.

Stay Consistent

1. **Lazy Loading**

Lazy Loading Routing: Lazy loading is a technique used in web development to optimize the loading time of a web page by deferring the loading of certain resources until they are actually needed. In the context of routing in a web application, lazy loading refers to loading specific modules or components only when the user navigates to a particular route, rather than loading all modules and components at the initial page load.

Lazy loading routing is commonly used in Single Page Applications (SPAs) where the entire application is loaded initially, but certain parts of the application are loaded on-demand as the user interacts with the application.

Without Lazy Loading: In a traditional setup, both the Home and About components would be loaded when the application starts, increasing the initial load time:

```
// Initialize the application

const app = new App({

  routes: [

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

    { path: '/about-us', component: AboutUsComponent },

  ],

});
```

With Lazy Loading: Lazy loading involves importing the component only when it's needed. In a lazy-loaded setup, the About Us component would be loaded only when the user navigates to the '/about-us' route.

When lazy loading a component, you typically use dynamic imports. Dynamic imports return a Promise, and the component is loaded when the Promise is resolved:

```
{

  path: 'about-us',

  loadChildren: () => import('./features/about-us/about-us.module').then(m =>
m.AboutUsModule)

}
```

- **path: 'about-us': Specifies the route path. In this case, it's 'about-us'. This means that when the user navigates to the 'about-us' route, the specified module (lazy-loaded) will be loaded and associated with this route.**
- **loadChildren: () => import('./features/about-us/about-us.module').then(m => m.AboutUsModule): Enables lazy loading. Instead of directly importing the module at**

the time the application loads, it uses the loadChildrenproperty to specify a function that will be called when the module is needed.

- **import('./features/about-us/about-us.module') is a dynamic import statement that returns a Promise. The module is not loaded immediately; it will be loaded asynchronously when the Promise is resolved.**
- **.then(m => m.AboutUsModule) ensures that the module is loaded and the specific module class (AboutUsModule) is retrieved.**

In summary, when a user navigates to the 'about-us' route, the associated module (AboutUsModule) will be loaded asynchronously, reducing the initial bundle size and improving the application's loading performance. Lazy loading is especially useful in large applications where loading all modules at once would lead to slower initial load times.

Advantages of Using Lazy Loading Routing:

1. **Reduced Initial Bundle Size: In a traditional Angular application, all modules are loaded at the initial startup, which can result in a larger bundle size.**
2. **Faster Initial Page Load: By loading only the required modules on demand, the initial page load time is significantly improved. Users see the main part of the application quickly, and additional features are loaded as they navigate to specific routes.**
3. **Improved User Experience: Users experience faster load times when accessing your application, leading to a better overall user experience.**
4. **Code Splitting: Lazy loading enables code splitting, where different parts of your application are divided into separate chunks. These chunks are loaded dynamically as needed, allowing for better optimization and utilization of browser caching.**
5. **Route-Level Loading: The loadChildren property is used at the route level, specifying which module should be loaded when a particular route is accessed.**

CLASS

A class is a fundamental building block that encapsulates data and behavior into a single unit. It serves as a blueprint or a template for creating objects. Encapsulation is the bundling of data (attributes or properties) and methods that operate on the data into a single unit (i.e., a class). It helps in hiding the internal details of the class and exposing only what is necessary.

```
class Animal {

 constructor(name) {

  this.name = name;

 }



 makeSound() {
```

```
  // Abstract method, to be implemented by subclasses
 }
}
```

In this example, Animal is an abstract class. It defines a property name and an abstract method makeSound. Concrete subclasses (e.g., Dog, Cat) will implement the makeSound method with specific behavior.

OBJECTS

An object is a self-contained unit that consists of both data (often referred to as attributes or properties) and the methods (functions) that operate on that data. Objects are instances of classes and are created based on the blueprint provided by the class.

Example: The Student class encapsulates the properties (name, age, grade) and behavior (displayInfo) into a single unit.

Reusability: If you want to represent another student, you can create a new instance of the Student class with different values. The same class can be reused for different students.

Without Object:

```
let studentName = "Alice";
let studentAge = 20;
let studentGrade = "A";

displayStudentInfo(name, age, grade) {
 console.log(`Student: ${name}, Age: ${age}, Grade: ${grade}`);
}

displayStudentInfo(studentName, studentAge, studentGrade);
```

With Object:

```
class Student {
 constructor(name, age, grade) {
  this.name = name;
```

```
    this.age = age;
    this.grade = grade;
  }

  displayInfo() {
    console.log(`Student: ${this.name}, Age: ${this.age}, Grade: ${this.grade}`);
  }
}

const alice = new Student("Alice", 20, "A");
alice.displayInfo();

const john = new Student("John", 24, "B");
john.displayInfo();
```

Now, you can easily manage a list of students by creating multiple instances of the Student class and calling the displayInfo method for each student.

PROPERTIES

Properties are variables declared within a class, defining the data that an object created from the class will hold. Each object created from the class will have its own set of these properties. These attributes or variables represent the object's state and store information about the object.

```
// Define a simple class
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
```

```
  greet() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  }
}

// Create an object (instance) of the Person class
const person1 = new Person('John', 25);

// Accessing properties and calling methods
console.log(person1.name); // Output: John
person1.greet(); // Output: Hello, my name is John and I am 25 years old.
```

METHODS

- A method is a block of code associated with an object or a class, which performs a specific action or provides a service.
- They represent the behavior or actions that objects of the class can perform.
- Methods operate on the data (properties) of an object and can interact with other objects.
- Methods can be reused across different parts of the program or even in different programs, enhancing code maintainability and reducing redundancy.

Without Parameters and Arguments:

```
getInfo() {
  console.log("Information Logged!!");
}
```

Parameters and Arguments:

- Parameters are variables declared in the method's signature.
- Arguments are values passed to the method when it is called.

Parameters receive the values of the corresponding arguments during method invocation:

```
addNumbers(a, b) {
```

```
  return a + b;
}


const result = addNumbers(3, 4); // Here, 3 and 4 are arguments passed to the addNumbers method.

console.log(result); // Output: 7
```

Return Statement: A method may return a value using the return statement. The returned value can be assigned to a variable or used in expressions:

```
multiply(a, b) {
  return a * b;
}


const product = multiply(5, 6);

console.log(product); // Output: 30
```

NAMING CONVENTIONS

1. **Method Naming:Use camelCase for method names.**
   - Example: getUserInfo()
2. **Event Handlers:Prefix event handler methods with "on".**
   - Example: onClick()
3. **Variable Naming:Use camelCase for variable names.**
   - Example: let userName = "John";
4. **Constants:Use uppercase letters and underscores for constant variables.**
   - Example: const MAX_COUNT = 10;
5. **Boolean Variables:Prefix boolean variables with "is" or "has".**
   - Example: let isActive = true;
6. **Parameter Naming:Use camelCase for parameter names.**
   - Example:
   - calculateSum(number1, number2) {
   - // method implementation
   - }

Stay Consistent

Data Binding

Data binding is a concept in software development that establishes a connection between the application's user interface (UI) and the underlying data. It simplifies the process of updating the UI when the data changes and vice versa. There are two main types of data binding: one-way data binding and two-way data binding.

One-way Data Binding:

- In one-way data binding, the data flows in a single direction, from the data source to the UI or vice versa.
- Changes in the data source are reflected in the UI, but changes in the UI do not affect the data source.
- This is commonly used when you want to display data in the UI but don't necessarily need to update the data based on user interactions.

<h2>{{message}}</h2>

Here is the formatted content as per your request:

Data Binding

Data Binding
Data binding is a concept in software development that establishes a connection between the application's user interface (UI) and the underlying data. It simplifies the process of updating the UI when the data changes and vice versa. There are two main types of data binding: one-way data binding and two-way data binding.

One-way Data Binding:

- In one-way data binding, the data flows in a single direction, from the data source to the UI or vice versa.
- Changes in the data source are reflected in the UI, but changes in the UI do not affect the data source.
- This is commonly used when you want to display data in the UI but don't necessarily need to update the data based on user interactions.

html

Copy code

<h2>{{message}}</h2>

Two-way Data Binding:

- In two-way data binding, changes in the UI automatically update the data source, and changes in the data source automatically update the UI.

- This is useful in scenarios where you want to keep the UI and data in sync bidirectionally, such as form input fields.

```
<form #userForm="ngForm">

 <label for="name">Name:</label>

 <input type="text" id="name" name="name" ngModel required>

 <label for="email">Email:</label>

 <input type="email" id="email" name="email" ngModel required>

 <button (click)="onSubmit(userForm)">Submit</button>

</form>
```

Two-way Binding using ngModel:

- The ngModel directive is used for two-way data binding.
- The [(ngModel)] syntax is applied to the <input> fields for "name" and "email."
- This establishes a connection between the UI and the corresponding properties in the form model.

```
<div class="form-container">

 <h2>Reactive Form</h2>

 <form [formGroup]="userForm">

  <div>

   <label for="name">Name:</label>

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

   <label for="email">Email:</label>

   <input type="email" id="email" formControlName="email">

   <button (click)="onSubmit()">Submit</button>

  </div>

 </form>

</div>
```

Two-way Binding using formControlName:

- The <input> fields are bound to specific form controls using formControlName.

- This establishes a two-way binding between the input fields and the corresponding properties in the userForm FormGroup.

Templates and Directives

Templates

A **template** in Angular is a declarative way of defining the structure of the UI (User Interface) for a component. It is an HTML file that includes Angular-specific syntax and expressions.
Purpose:
Templates allow you to define the layout and structure of your application's UI. They include placeholders for data binding, which allows you to display dynamic content and respond to user interactions.

<h1>{{ title }}</h1>

{{ description }}

<button (click)="handleClick()">Click me</button>

Directives

Directives in Angular are markers on a DOM (Document Object Model) element that tell Angular to attach a specific behavior to that element or transform it in a certain way.
Purpose:
They are a way to extend, transform, and manipulate the DOM and its behavior within Angular applications. Directives can be classified into two main types: structural directives and attribute directives.

Structural Directives

Structural directives are responsible for manipulating the structure of the DOM by adding, removing, or replacing elements. They are prefixed with an asterisk (*) in the template syntax.

Angular provides built-in directives, such as ngIf, ngFor, and ngSwitch, which are used for conditional rendering, looping through arrays, and switching between multiple views, respectively. Additionally, you can also create custom directives to encapsulate and reuse behavior across components.

Host Elements

A **host element** refers to the element to which a directive is applied. When you use a directive in an Angular template, you apply it to a specific element, and that element is considered the host element for this directive.

Commonly Used Structural Directives in Angular

ngIf
Usage: *ngIf="condition"
Description: Conditionally adds or removes the host element and its children from the DOM based on the specified condition.

```
<div *ngIf="isUserLoggedIn">

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

</div>

isUserLoggedIn: boolean = true;
```

ngFor
Usage: *ngFor="let item of items"
Description: Iterates over a collection (e.g., array) and repeats the host element for each item in the collection. It is used for rendering lists of items dynamically.

```
<ul>

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

</ul>

items: string[] = ['Item 1', 'Item 2', 'Item 3'];
```

ngSwitch
Usage: <div [ngSwitch]="expression"> ...
Description: Conditionally renders content based on the value of an expression. It's similar to a switch statement in programming languages.

```
<div [ngSwitch]="userRole">

  <div *ngSwitchCase="'admin'">

    <p>Welcome, Admin!</p>

  </div>

  <div *ngSwitchCase="'user'">

    <p>Welcome, User!</p>

  </div>

  <div *ngSwitchDefault>
```

```
  <p>Welcome, Guest!</p>
 </div>
</div>
userRole: string = 'guest';
```

Event Handling

Event Handling in Angular involves responding to user interactions, such as clicks, key presses, or form submissions.

```
<button (click)="onSubmit()">Submit</button>
onSubmit() {
 console.log("Form Submitted", this.userForm.value);
}
```

Explanation:

- (click) is an Angular event binding syntax. It associates the onSubmit() method from the component class with the click event of the "Submit" button.

Stay Consistent

Component Lifecycle Hooks & Pipes

Lifecycle Hooks

These are lifecycle hooks in Angular, which are methods that Angular calls at specific points in the lifecycle of a component or directive. Let's go through each one with a brief explanation.

Sequence of lifecycle hooks when a component is created:

- **Constructor**
  The constructor of the component class is called first.

1. **ngOnChanges**
   If the component has input/output bindings, ngOnChanges is called next. This hook is called before ngOnInit and provides information about the changes in the component's input/output property when the binding value changes.
2. **ngOnInit**
   After ngOnChanges, the ngOnInit lifecycle hook is called. This is called once after the component is initialized. It is ideal for initializing component properties.
3. **ngDoCheck**
   This hook is called after ngOnInit. It provides an opportunity for the developer to

implement custom change detection logic. It is called during every change detection cycle.

4. **ngAfterContentInit**
   Called after the component's content has been initialized. This hook is often used when you need to perform initialization logic related to content projection.
5. **ngAfterContentChecked**
   It is called after every check of the component's content. It provides an opportunity to perform logic after the content has been checked during the change detection cycle.
6. **ngAfterViewInit**
   Called once after the component's view and child views have been initialized. It is useful for performing operations that require the view to be ready.
7. **ngAfterViewChecked**
   It is called after every check of the views of the component. It allows you to perform logic after the views have been checked during the change detection cycle.
8. **ngOnDestroy**
   This hook is called just before the directive (or component) is destroyed. It provides an opportunity to perform cleanup logic, such as unsubscribing from observables or releasing resources.
   The ngOnDestroy lifecycle hook won't be automatically called for a component unless it is actually destroyed. Components in Angular are typically destroyed when their associated views are removed from the DOM or when their parent component is destroyed, Here are a few scenarios where the ngOnDestroy hook is typically called:

- **Routing Navigation:** If the component is associated with a route, it will be destroyed when the user navigates away from that route.
- ***ngIf or ngIf Directives:** If the component is conditionally rendered using *ngIf or ngIf directive, it will be destroyed when the condition becomes false.

Pipes

In Angular, pipes are a way to transform data in the template before displaying it. They are similar to filters in other frameworks and can be used for various tasks, such as formatting dates, converting text to uppercase, or filtering lists. Angular provides several built-in pipes, and you can also create custom pipes for your specific needs. Here are a few examples of built-in Angular pipes:

1. **Uppercase and Lowercase Pipes**
   The uppercase and lowercase pipes transform the input string to uppercase or lowercase, respectively.
2. **Date Pipe**
   The date pipe formats a date based on the provided format. In this example, the 'short' format is used.
3. **Percent Pipe**
   The percent pipe multiplies the input by 100 and appends a percentage sign.
4. **Json Pipe**
   The json pipe converts an object to its JSON string representation. It's useful for debugging or displaying structured data.

5. **Keyvalue Pipe**
   The keyvalue pipe iterates over the key-value pairs of an object, allowing you to display them in a template.
6. **Slice Pipe**
   The slice pipe extracts a portion of a string or array. In this example, it extracts characters from index 0 to 6.
7. **Title Case Pipe**
   The titlecase pipe converts the input string to title case (capitalizing the first letter of each word).

Stay Consistent

Event Binding

Definition:
Event binding is a mechanism in Angular that allows you to capture and respond to events triggered by user interactions, such as clicks, keystrokes, mouse movements, etc. It provides a way to execute custom code when certain events occur in the application.

Syntax:
In Angular, event binding is accomplished by using a set of parentheses () in the template to bind a method or expression to a specific event.

Example:
HTML:

```
<button type="submit" (click)="submit()">Submit</button>
```

TypeScript:

```
submit() {

  console.log("SUBMIITEDDDDDDD");

}
```

In this example, the (click) event binding is used to bind the submit() method to the click event of the button. When the button is clicked, the submit() method is executed, logging "SUBMITTED" to the console.

Event Filtering

Definition:
Event filtering involves selectively handling or filtering events based on certain conditions within the event handler. It allows you to evaluate conditions before allowing the default behavior associated with the event to take place. For example, you might want to perform an

action only if certain keys are pressed, a specific element is targeted, or some other criteria are met.

Event filtering typically involves adding conditions inside the function that handles the event to determine whether to proceed with the default action or not based on certain criteria.

Example:
HTML:

```
<button type="submit" (click)="handleSubmit($event)">Submit</button>
```

TypeScript:

```
handleSubmit(event: any) {

  console.log("EVENTTTTTTTT", event);

  console.log("EVENTTTTTTTT", event.altKey);



  if (event.altKey) {

    console.log("ALTTT key is pressed");

  } else {

    console.log("Normal");

  }

}
```

Explanation:

- The ($event) syntax in the template is used to pass the event object to the corresponding method in the component.
- The event object contains information about the event, such as which key was pressed, the type of event, etc.
- Inside the event handler method (handleSubmit), you can access properties of the event object to perform conditional actions based on certain criteria.
- In this case, it checks whether the Alt key is pressed and logs different messages accordingly. In this example, the (click) event is bound to the handleSubmit($event) method. The $event is a special variable that captures the event object. Inside the handleSubmit method, it checks if the Alt key is pressed (event.altKey). If the Alt key is pressed, it logs "ALT key is pressed," otherwise, it logs "Normal."

Stay Consistent

OnChange

Definition:
The (change) event is triggered when the value of an input element, such as a dropdown (select) or an input field (input), changes. It typically occurs after the user has selected a different option in a dropdown or entered text into an input field and then moved the focus away from that element.

Example:
HTML:

```
<label for="gender" class="asterisk">Gender</label>

<select id="gender" formControlName="gender" placeholder="Please select Gender"
(change)="onGenderChange($event)">

  <option value="" disabled selected>Select Gender</option>

  <option value="male">Male</option>

  <option value="female">Female</option>

  <option value="other">Other</option>

</select>
```

TypeScript:

```
selectedGender: any;

onGenderChange(event: any) {

  console.log("onGenderChange", event);

  console.log("onGenderChange target", event.target);

  console.log("onGenderChange value", event.target.value);

  this.selectedGender = event.target.value;

}
```

Explanation:
The (change) event is attached to the select element, and it triggers the onGenderChange method when the user selects a different option.
Now, selectedGender is a property in the component that is used to store the selected gender.

The onGenderChange method is called when the (change) event occurs. Let's see how this works here:

- event is the event object itself.
- event.target refers to the DOM element that triggered the event, in this case, the select element.
- event.target.value retrieves the selected value from the select element, which corresponds to the chosen gender.
  Finally, the selected gender value is assigned to the selectedGender property.

Subscribe Value Changes

Definition:

valueChanges:

- In Angular's Reactive Forms, each form control has a property called valueChanges.
- valueChanges is an observable that emits an event every time the value of the associated form control changes.
- It provides a way to observe and react to changes in real-time.

subscribe:

- The subscribe method is part of the Observable pattern in JavaScript and is used to listen to events emitted by an observable.
- In the context of Angular's Reactive Forms, when you call subscribe on the valueChanges observable of a form control, you're essentially saying, "I want to be notified whenever the value of this form control changes."

Example:
HTML:

html

Copy code

```
<label for="gender" class="asterisk">Gender</label>

<select id="gender" formControlName="gender" placeholder="Please select Gender">

  <option value="" disabled selected>Select Gender</option>

  <option value="male">Male</option>

  <option value="female">Female</option>

  <option value="other">Other</option>

</select>
```

TypeScript:

typescript

Copy code

```
ngOnInit() {

  this.medicalForm.get('gender').valueChanges.subscribe((value: string) => {

    this.selectedGender = value;

    console.log("selectedGender", this.selectedGender);

  });

}
```

Explanation:
this.medicalForm.get('gender'):

- medicalForm is an instance of the Angular FormGroup class, representing a form in the component.
- .get('gender') is used to retrieve the form control named 'gender' from the medicalForm.

valueChanges.subscribe(callback):

- valueChanges returns an observable that you can subscribe to.
- When you subscribe to valueChanges, you provide a callback function (callback) that will be executed whenever the value of the form control changes.

.subscribe((value: string) => {...}):

- .subscribe is used to subscribe to the changes emitted by the valueChanges observable.
- The callback function (value: string) => {...} is executed whenever the value changes.
- value is the new value of the 'gender' form control, and it is expected to be a string. Inside the callback function, this.selectedGender = value; assigns the new value to the selectedGender property in the component.

The ngOnInit method sets up a subscription to changes in the 'gender' form control using the valueChanges observable. Whenever the value of the 'gender' control changes, the callback function is executed, updating the selectedGender property and logging the new value to the console.

Stay Consistent

TimeOuts

Definition:
Timeouts in Angular refer to a mechanism where a function or a piece of code is scheduled to run after a specified amount of time has passed. It allows you to delay the execution of a specific task.

Example:
TypeScript:

typescript

Copy code

```
constructor(){

  this.startTimeout();

}

startTimeout(){

  setTimeout(() =>{

    console.log("Timeeoutttt completeeddd!!!!");

  }, 5000);

}
```

Explanation:
In the example, the startTimeout method is called in the constructor, and it uses setTimeout to log a message after a delay of 5000 milliseconds (5 seconds). This is useful for scenarios where you want to execute code after a certain delay.

INTERVALS

Definition:
Intervals involve repeatedly executing a function or a block of code at specified time intervals. It is similar to timeouts but differs in that it continues executing the provided function at regular intervals until explicitly stopped.

Example:
TypeScript:

typescript

Copy code

```
constructor(){
  this.startInterval();
}
startInterval(){
  setInterval(() => {
    console.log("Set intervall completed");
  }, 2000);
}
```

Explanation:

- The startInterval method uses setInterval to repeatedly execute the provided function (logging a message) every 2000 milliseconds (2 seconds).
- The function provided to setInterval will be invoked at regular intervals until the application is closed or the interval is explicitly cleared.

To stop the interval and exit the loop, you need to use the clearInterval function, passing in the interval ID returned by setInterval. Here's how you can modify your code to include a way to stop the interval:

Example:
TypeScript:

typescript

Copy code

```
intervalId: any; // Store the interval ID
constructor() {
  this.startInterval();
}
startInterval() {
  this.intervalId = setInterval(() => {
    console.log("Set interval completed");
```

```
  }, 2000);

}

stopInterval() {

  clearInterval(this.intervalId); // Clear the interval using the stored ID

  console.log("Interval stopped");

}
```

Explanation:
In this example, I've added a variable intervalId to store the ID returned by setInterval. The stopInterval method can then be called to clear the interval and stop the execution of the provided function. Now, whenever you want to exit the loop, you can call the stopInterval method. For example, you might call it based on some condition or user action.

CHANGE DETECTOR REF

Definition:
ChangeDetectorRef is an Angular service that allows you to manually trigger change detection.
When you update data outside Angular's normal lifecycle (e.g., through third-party libraries or asynchronous operations), Angular might not be aware of the changes. Calling detectChanges ensures that Angular checks for changes and updates the view accordingly.

Example:
HTML:

html

Copy code

```
<div>{{message}}</div>

<button (click)="updateMessage()">Update</button>
```

TypeScript:

typescript

Copy code

```
updateMessage(){

  this.message = "Updated message";
```

```
  this.cdr.detectChanges();
}
```

Explanation:

- The HTML template displays a message using the Angular interpolation syntax ({{message}}) and provides a button triggering the updateMessage method.
- The updateMessage method updates the value of message and then calls cdr.detectChanges().

It's important to note that while ChangeDetectorRef provides a way to manually trigger change detection, it's generally recommended to rely on Angular's automatic change detection mechanism as much as possible, as it is optimized for performance.

Stay Consistent

Interfaces & Enums

Definition:

An interface is a way to define a contract for the structure of an object. It declares a set of properties and their types, but it doesn't provide an implementation. It's used to specify what properties an object must have without actually defining how those properties will be implemented.
By enforcing this structure, you make it clear what properties are expected for a valid Product object, providing type safety and making the code more maintainable.
If you attempt to assign an object to the array of products that does not adhere to the structure defined by the Product interface, TypeScript will raise a compile-time error. TypeScript interfaces are used for static type checking, and they help catch potential issues during the development phase.

Example:

TypeScript:

typescript

Copy code

```
// Outside the class:

interface Product {
```

```
  id: number;
  name: string;
  price: number;
  quantity: number;
}

// Inside the class:
product: Product[] = [
  {
    id: 1,
    name: 'ABC',
    price: 5,
    quantity: 10,
  },
]
```

Explanation:
The Product interface defines a contract for objects that should have properties like id, name, price, and quantity, each with a specific data type.
Inside your Component class, an array of products is declared using the Product interface. This ensures that each product in the array adheres to the structure defined by the interface. If an object doesn't match the structure defined in the interface, TypeScript will raise a compilation error.

ENUMS

Definition:
Enums, short for enumerations, are a feature in many programming languages, including TypeScript. In TypeScript, enums allow you to define a set of named constant values. These values can then be used in your code to represent a finite set of distinct options or states. Each of these members is assigned an automatic numeric value starting from 0. Enums are often used when you have a fixed set of values that a variable can take. For example, you might use an enum to represent the possible directions in a game, the days of the week, or the status of an operation.

Example:

TypeScript:

typescript

Copy code

```
// Outside the class:
enum Direction {
  Up,
  Down,
  Left,
  Right
}


// Inside the class:
enums() {
  let direction: Direction;
  direction = Direction.Left;
  console.log("direction", direction);
}
```

Explanation:
The Direction enum is defined with four values: Up, Down, Left, and Right. By default, each value is assigned a numeric index starting from 0.
Suppose, if you declare enum Direction with Up = 5, it sets Up to 5, and subsequent enum members are assigned incremental values. Down will be 6, Left will be 7, and Right will be 8.
Inside your Component class, a variable direction of type Direction is declared and assigned the value Direction.Left. Enums help make code more readable by providing meaningful names for numeric values.

By defining interfaces and enums separately, you make them reusable across multiple classes.

Stay Consistent

Angular Animations

For using this, firstly enter the command in your command prompt:
npm i @angular/animations

Definition:
Angular animations are a feature in the Angular framework that allows you to create dynamic and visually appealing effects in your web applications. These animations are built on top of the Web Animations API and provide a powerful way to manipulate the presentation of HTML elements during different states or transitions.

Here are the key concepts in Angular animations:

Animation Trigger:
An animation is defined by an animation trigger, which is created using the trigger function. The trigger has a name and contains one or more states, transitions, and associated styles.
Example:

typescript

Copy code

```
trigger('myAnimation', [

  // states, transitions, and styles

])
```

States:
States represent different configurations or appearances of an element. States are defined using the state function and include a set of styles that should be applied when the element is in that state.
Example:

typescript

Copy code

```
state('start', style({ opacity: 1, transform: 'scale(1)' }))
```

Transitions:
Transitions define how an element moves from one state to another. They are created using the transition function, specifying the start and end states, as well as the animation that should be applied during the transition.
Example:

typescript

Copy code

transition('start => end', animate('500ms ease-in'))

Styles:
Styles represent the CSS properties and values that should be applied to an element in a particular state. Styles are defined using the style function.
Example:

typescript

Copy code

style({ backgroundColor: 'red', color: 'white' })

Types of Angular Animations

Single Animation:
This property is used to define animations for the component. It includes an array with a single animation trigger named myAnimation. The trigger specifies two states (start and end) and transitions between them with associated styles and timing.
currentState: This is a variable in the component that keeps track of the current state of the animation. It is initially set to 'start'.

Example:
HTML:

html

Copy code

<div [@myAnimation]="currentState" (click)="toggleState()">

  Click me to toggle Animation

TypeScript:

typescript

Copy code

animations: [

  trigger('myAnimation', [

    state('start', style({

```
    transform: 'scale(1)',
    backgroundColor: 'red'
  })),
  state('end', style({
    transform: 'scale(1.5)',
    backgroundColor: 'blue'
  })),
  transition('start => end', animate('500ms ease-in')),
  transition('end => start', animate('500ms ease-out'))
 ])
]

currentState: string = 'start';


toggleState() {
 this.currentState = this.currentState === 'start' ? 'end' : 'start';
}
```

Explanation:
[@myAnimation]="currentState": This is an Angular animation binding. It binds the myAnimation animation trigger to the current state (currentState variable) of the component. As the state changes, the associated animation will be triggered.

Group Animation:
This property is used to define animations for the component. It includes an array with a single animation trigger named combinedAnimations. The trigger specifies two states ('start' and 'end') and transitions between them with a group of two animations for each transition. currentState: This is a variable in the component that keeps track of the current state of the animation. It is initially set to 'start'.

Example:
HTML:

html

Copy code

```
<div [@combinedAnimations]="currentState"></div>

<button (click)="toggleAnimation()">Toggle Button</button>
```

TypeScript:

typescript

Copy code

```
animations: [
  trigger('combinedAnimations', [
    state('start', style({
      opacity: 1,
      transform: 'translateX(0)'
    })),
    state('end', style({
      opacity: 0,
      transform: 'translateX(100px)'
    })),
    transition('start => end', group([
      animate('500ms ease-out', style({ transform: 'scale(1.5)', backgroundColor: 'blue' })), // First animation
      animate('300ms 200ms ease-in', style({ opacity: 0 })), // Second animation with delay
    ])),
    transition('end => start', group([
      animate('500ms ease-out', style({ transform: 'translateX(0)' })), // Reverse the first animation
      animate('300ms ease-in', style({ opacity: 1 })), // Reverse the second animation
    ])),
```

```
  ])
]
currentState: string = 'start';


toggleAnimation() {
  this.currentState = this.currentState === 'start' ? 'end' : 'start';
}
```

Explanation:
[@combinedAnimations]="currentState": This is an Angular animation binding. It binds the combinedAnimations animation trigger to the current state (currentState variable) of the component. As the state changes, the associated combined animation will be triggered.

Stay Consistent

Error Handling

Error handling is crucial to manage and respond to unexpected issues that may occur during the execution of a program. By handling errors, you can ensure that your application doesn't crash or stop unexpectedly when it encounters unexpected situations. Instead, it can degrade gracefully, providing a better user experience. One of the common ways to handle errors is by using the try, catch, and finally blocks.

try block:
In this block, you place the code that might throw an exception. The try block is the region of code where you expect an error to occur.

catch block:
If an exception is thrown in the try block, the control flow jumps to the corresponding catch block. The catch block contains the code that will handle the exception. You can define a variable (often named error or err) that will hold information about the thrown error.

finally block (optional):
This block is executed regardless of whether an exception occurred or not. It is commonly used for cleanup code that must be executed no matter what.

Example:

typescript

Copy code

```
constructor() {
  this.errorhandling();
}

sampleError() {
  let x: number = null!;
  console.log("Sample", x);
  throw new Error("");
}

errorhandling() {
  try {
    let result = this.sampleError();
    console.log("Result", result);
  } catch (error: any) {
    console.error("Type error", error.message);
  } finally {
    console.log("Finally");
  }
}
```

Explanation:

- **sampleError():**
  It is a method that is throwing an error intentionally. It declares a variable x with a type annotation of number, but assigns it the value null!. The ! here is the non-null assertion operator, essentially telling TypeScript to treat null as a valid value for x. Then it logs the value of x and throws an empty Error.
- **errorhandling():**
  It is a method that contains a try-catch-finally block. Here's how it works:

- **Try Block:** It attempts to execute the code inside. It calls this.sampleError() and assigns its result to the result variable.
- **Catch Block:** If an exception is thrown inside the try block, it catches the exception. The catch block logs an error message to the console, specifically mentioning it as a "Type error." The error variable in the catch block is of type any, meaning it can be of any type.
- **Finally Block:** This block is always executed, whether an exception is thrown or not. It logs "Finally" to the console.

In summary, when an instance of this class is created, the constructor calls errorhandling(). errorhandling() in turn calls sampleError() which intentionally throws an error. The catch block in errorhandling() catches the error, logs a message, and the finally block is executed regardless of whether an error occurred or not.

Metadata

In Angular, metadata plays a crucial role in defining and configuring various aspects of components, directives, modules, and services. Metadata is essentially additional information that describes how a particular class should be processed or behave in the Angular application.

Angular uses decorators to attach metadata to classes. A decorator is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter. Decorators are denoted by the @ symbol followed by the decorator's name.

Example:

typescript

Copy code

```
@Component({
  // Metadata
  selector: 'app-error-handling',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './error-handling.component.html',
  styleUrls: ['./error-handling.component.scss']
})
```

Stay Consistent

Service/DI/Constructor

Services

- In Angular, services are a fundamental building block of the framework. They are used to encapsulate and provide a specific functionality or set of functionalities that can be shared across multiple components, directives, or other services within an Angular application.
- Services have a lifecycle managed by Angular. When provided at the root level (providedIn: 'root'), a service is a singleton, meaning there is only one instance of the service in the entire application.
- Angular services are typically created as TypeScript classes. You use the @Injectable decorator to mark a class as a service. It's not always required, but it's good practice to use it.
- Services play a crucial role in promoting modularity, code organization, and reusability. Services are used to encapsulate and manage functionality that doesn't belong to a specific component, such as data fetching, business logic, or communication with a backend server.

Generate services using the ng generate service service-name command.

Dependency Injection

- Dependency injection in Angular allows you to declare the dependencies of a class (like services) in its constructor, and Angular's dependency injection system will provide instances of those dependencies when the class is instantiated.
- Angular's dependency injection system is used to inject services into the components or other services that need them. When a component or another service requests a dependency (like a service) in its constructor, Angular provides the instance of that dependency.
- This makes the code more modular, testable, and maintainable, as dependencies can be easily swapped or mocked during testing. The @Injectable decorator is an essential part of this mechanism, allowing Angular to understand how to create and manage instances of services.

Example:

User.service.ts:

typescript

Copy code

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



@Injectable({
```

```
  providedIn: 'root'
})
export class UserService {
  users: any = ['user1', 'user2', 'user3'];

  constructor() {}

  getData() {
    return 'Data from Service';
  }

  getUsers() {
    return this.users;
  }
}
```

HTML:

```
html
Copy code
<p>{{data}}</p>
<h2>Users</h2>
<ul>
  <li *ngFor="let user of users">
    {{user}}
  </li>
</ul>
```

TS:

typescript

Copy code

```
data: any;

users: any;


constructor(public userService: UserService) {

 this.data = this.userService.getData();

 console.log("Result", this.data);

 this.users = this.userService.getUsers();

 console.log("Userss---", this.users);

}
```

Explanation:

- **Class Properties:**
  - data: It is a property to store the result of calling the getData() method from the UserService.
  - users: It is a property to store the result of calling the getUsers() method from the UserService.
- **Constructor:**
  - The constructor takes an instance of UserService as a parameter (public userService: UserService), indicating that an instance of the UserService will be injected into this component.
  - Inside the constructor, it calls getData() and getUsers() methods from the injected UserService.
  - The result of getData() is assigned to the data property, and the result of getUsers() is assigned to the users property.
  - Finally, it logs the results to the console.

Stay Consistent

Asynchronous Operations, Async/Await, Promise

Asynchronous Operation

- In programming, operations can be broadly categorized as synchronous or asynchronous. Synchronous operations block the execution of code until the operation is complete,

whereas asynchronous operations allow the program to continue executing while waiting for the operation to finish.
- Asynchronous operations are common in scenarios like fetching data from a server, reading files, or handling user input. In traditional synchronous code, these operations could lead to blocking the entire program, resulting in poor user experience and performance.

Async / Await

- **async Function:**
  - An async function is a function that always returns a promise.
  - It can contain the await keyword, signaling that the function will work with promises and asynchronous code.
- **await Keyword:**
  - The await keyword is used inside an async function to pause the execution of the function until the promise is resolved.
  - It can be applied to any promise-like object, including promises returned by asynchronous functions or methods.
- **Error Handling with try/catch:**
  - try/catch blocks can be used to handle errors in an async function, providing a structured way to handle both resolved values and errors.

Promise

A Promise is an object representing the eventual completion or failure of an asynchronous operation. It is a container for a value, which may be available now, or in the future, or never.

The key characteristics of a Promise are:

- **Pending:** The initial state of a Promise. The operation represented by the Promise is ongoing, and the final result is not yet known.
- **Fulfilled (Resolved):** The Promise successfully completed, and a result value is available.
- **Rejected:** The Promise encountered an error or was unable to fulfill the operation, and an error reason is available.

Creating a Promise

A Promise is created using the Promise constructor, which takes a single argument, a function called the executor. The executor function is passed two functions: resolve and reject. These functions are used to signal the successful completion or failure of the asynchronous operation.

Example:

User.service.ts:

typescript

Copy code

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


@Injectable({
  providedIn: 'root'
})
export class UserService {
  fetchData(): Promise<string> {
    return new Promise((resolve, reject) => {
      const success = true;
      setTimeout(() => {
        if (success) {
          resolve('Data from resolve---')
        } else {
          reject('Error while fetching---rejected')
        }
      }, 1000)
    })
  }
}
```

HTML:

html

Copy code

```
<h2>{{this.fetchData}}</h2>
```

TS:

typescript

Copy code

```
async ngOnInit() {

  this.fetchData = await this.userService.fetchData();

  console.log("Fetched data-----", this.fetchData);

}
```

Explanation:

- **fetchData:**
  This is a method that returns a Promise<string>. This function is designed to simulate an asynchronous operation, such as an HTTP request, using setTimeout to introduce a delay of 1000 milliseconds (1 second).
  Inside the Promise constructor, there are two parameters: resolve and reject. resolve is a function used to fulfill the promise with a successful result, and reject is a function used to reject the promise with an error.
  The success variable is used to simulate whether the asynchronous operation was successful. If success is true, the promise is resolved with the string 'Data from Service'. Otherwise, it is rejected with the string 'Error fetching data'.
- **async ngOnInit():**
  This method is marked as async, indicating that it contains asynchronous operations.
  - async is used to define a function as asynchronous, allowing the use of await within it.
  - await is used to pause the execution of the function until the awaited promise is settled (resolved or rejected).

Stay Consistent

Http Requests, RXJS, Observable, Subscribe & Observables

HTTP REQUESTS

In Angular, when you make HTTP requests, such as fetching data from a server, the response is often represented as an observable. An observable is an object that emits a sequence of values over time, and it can be used to handle asynchronous operations like HTTP requests. Observables are part of the Reactive Extensions for JavaScript (RxJS) library.

RXJS

RxJS, which stands for Reactive Extensions for JavaScript, is a library that provides a set of tools for reactive programming in JavaScript. It is based on the principles of the ReactiveX library and extends the concept of observables, allowing you to work with asynchronous data streams and events. Here are some key concepts and features of RxJS:

- Observables
- Operators
- Subscription
- Error Handling, etc.

OBSERVABLE

In Angular, observables are often used to handle asynchronous operations like HTTP requests or user interactions. Observable is the result of the HTTP request.

- When you make an HTTP request, it returns an observable representing the stream of data that will come from the server.
- You subscribe to this observable to listen for data changes or handle errors.

Imagine a Stream

Observable as a Stream: Think of an observable as a stream of events or data over time. Just like a river that flows continuously, an observable emits values or events continuously.

Data Over Time

Values Over Time: Imagine you're tracking the temperature every hour. The temperature at each hour is a value in the stream of temperatures over time.

- At 9 AM: 20°C
- At 10 AM: 22°C
- At 11 AM: 25°C

This series of temperature values over time forms a stream, and an observable is like this stream of data.

SUBSCRIBE AND OBSERVERS

Subscribe

Listening to Changes: Now, imagine you want to know the temperature changes. You subscribe to this stream of temperature data. Whenever the temperature changes, you get notified.

- You subscribe at 9 AM.
- You receive 20°C.
- You subscribe at 10 AM.
- You receive 22°C.

You are "subscribed" to the observable stream, and you get updates whenever there's a new value.

Observer:

The observer is created through the subscribe method. The three functions passed to subscribe represent the observer:

- The subscribe method is used to subscribe to the observable. It takes an object as an argument, following the observer pattern.
- The next property is a function that will be called when the observable emits a new value. In this case, it logs the received data to the console.
- The error property is a function that will be called if there is an error during the observable's execution. It logs the error to the console.
- The complete property is a function that will be called when the observable completes. It logs a message indicating that the request has completed. This function is optional, and you don't have to include it.

Example:

User.service.ts:

typescript

Copy code

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


@Injectable({
  providedIn: 'root'
})
export class UserService {
  observeData() {
    let request = of('Data from my side'); // Return Observable--stream of data
    request.subscribe({
      // observer
      next: (data) => {
        console.log("Subscribed data---", data);
      },
      error: (error) => {
```

```
      console.log("Subscription Error--", error);
    },
    complete: () => {
      // optional
      console.log("Request complete");
    },
  });
 }
}
```

HTML:

html

Copy code

```
<h2>{{this.observeData}}</h2>
```

TS:

typescript

Copy code

```
async ngOnInit() {
 this.observeData = await this.userService.observeData();
 console.log("Observed data-----", this.observeData);
}
```

Explanation

Observable Creation:

- of is an RxJS creation operator used to create an observable that emits a specific value or set of values.
- In this case, of('Data from my side') creates an observable that emits the string 'Data from my side'.

Subscription:

- The subscribe method is used to subscribe to the observable and start listening for emitted values.
- It takes an object (the observer) with three properties: next, error, and complete.

Observer Pattern:

- The observer pattern is employed to handle different aspects of the observable's lifecycle.
- next is a function that gets called when a new value is emitted.
- error is a function that gets called when an error occurs during the observable's execution.
- complete is a function that gets called when the observable completes. It is optional and not always needed.

Result: When you call observeData(), it creates an observable using of('Data from my side'). The observable emits the specified data, and the observer's next function logs the data to the console. If there were any errors during the process, they would be logged in the error function. The complete function (optional) logs a message when the observable completes.

Stay Consistent

API

CRUD OPERATIONS

CRUD operations refer to the basic operations that can be performed on data:

 Create (C): Adding new data to the system.

 Read (R): Retrieving existing data from the system.

 Update (U): Modifying existing data in the system.

 Delete (D): Removing data from the system.

HttpClientModule

The HttpClientModule is a module provided by Angular's HTTP client library (@angular/common/http) that offers a powerful way to interact with HTTP services in Angular applications. Here's why we use it:

I. Simplified HTTP Requests

II. Observables for Asynchronous Operations

III. Integration with Angular Features

IV. Security Features

and many more...

For using this HttpClientModule:

–Add

provideHttpClient()

In the providers in your app.config.ts file

And then import HttpCLientModule in your component

The URL that has been used below is used to make HTTP requests to fetch, modify, or delete data from

the JSON file Create a json file in the assets folder to store the Data:

Data.json

{

"items": [

{ "id": 1, "name": "Item 1" },

{ "id": 2, "name": "Item 2" },

{ "id": 3, "name": "Item 3" }

]

}

The Data.json file contains an array of items with each item having an id and a name.

User.service.ts

This service contains methods to perform CRUD operations on the data stored in the JSON file.

Get All Details

User.service.ts

//Url

private getUrl = 'assets/data.json';

constructor(private http: HttpClient) {}

// Get all items

getItems(): Observable<any>{

return this.http.get(this.getUrl);

}

User.service.ts

//Url

private getUrl = 'assets/data.json';

constructor(private http: HttpClient) {}

// Get all items

getItems(): Observable<any>{

return this.http.get(this.getUrl);

}

Explanation:

 getItems() method in the User.service.ts fetches all items from the JSON file using an HTTP GET

request.

 It subscribes to the observable returned by HttpClient.get() method and logs the items to the

console.

Get By Id

User.service.ts

//Url

private getUrl = 'assets/data.json';

```
constructor(private http: HttpClient) {}

// Get a specific item by ID
getItemById(id:number){
return this.http.get(`${this.getUrl}?id=${id}`);
}
```

App.ts

```
items: any[] = [];
constructor(public userService: UserService) {
}
ngOnInit() {
this.getItemById(1);
}
getItemById(id: number) {
this.userService.getItemById(id).subscribe((response: any) => {
console.log("Item by Id", id);
const item = response.items.find((item: any) => item.id === id);
console.log("SIngle Item", item);
});
}
```

Explanation

 getItemById(id: number) method in the User.service.ts fetches a specific item by its ID from the JSON file using an HTTP GET request.

 It interpolates the id parameter into the URL to retrieve the specific item.

 It subscribes to the observable returned by HttpClient.get() method and logs the retrieved item to

the console.

Add Items

User.service.ts

```
//Url
private getUrl = 'assets/data.json';
constructor(private http: HttpClient) {}

// Create a new item
addItem(item:any): Observable<any>{
return this.http.post(this.getUrl,item)
}
```

App.ts

```
items: any[] = [];
constructor(public userService: UserService) {
}
ngOnInit() {
this.addItem({ id: 4, name: 'New Bansi' });
}
// Create a new item
addItem(newItem: any) {
this.userService.addItem(newItem).subscribe((data) => {
console.log("added item", newItem);
})
}
```

Explanation

 addItem(item: any) method in the User.service.ts adds a new item to the JSON file using an HTTP POST request.

 It sends the new item data in the request body.

 It subscribes to the observable returned by HttpClient.post() method and logs the added item to the console.

Update Item

User.service.ts

```
//Url

private getUrl = 'assets/data.json';

constructor(private http: HttpClient) {}


// Update an existing item by ID

updateItem(id:number,updatedItem:any): Observable<any>{

return this.http.put(`${this.getUrl}?id=${id}`,updatedItem)

}
```

App.ts

```
items: any[] = [];

constructor(public userService: UserService) {

}

ngOnInit() {

this.updateItem(3, { id: 3, name: 'Updated Bansi' });

}

// Update an existing item by ID

updatedItem(id: number, updatedItem: any) {

this.userService.updateItem(id, updatedItem).subscribe((data) => {
```

```
console.log("Updated Item", id);

console.log("Updated sucessfully");

})

}
```

Explanation

 updateItem(id: number, updatedItem: any) method in the User.service.ts updates an existing item

in the JSON file by its ID using an HTTP PUT request.

 It interpolates the id parameter into the URL to identify the item to be updated.

 It sends the updated item data in the request body.

 It subscribes to the observable returned by HttpClient.put() method and logs the updated item to

the console.

Delete Item

User.service.ts

```
//Url

private getUrl = 'assets/data.json';

constructor(private http: HttpClient) {}


// Delete an item by ID

deleteItem(id:number): Observable<any>{

return this.http.delete(`${this.getUrl}?id=${id}`);

}
```

App.ts

```
items: any[] = [];
```

```
constructor(public userService: UserService) {

}

ngOnInit() {

this.deleteItem(2);

}

// Delete an item by ID

deleteItem(id:number){

this.userService.deleteItem(id).subscribe(()=>

{

console.log("Item dleeted succesfuully");

})

}
```

Explaination:

 deleteItem(id: number) method in the User.service.ts deletes an item from the JSON file by its ID

using an HTTP DELETE request.

 It interpolates the id parameter into the URL to identify the item to be deleted.

 It subscribes to the observable returned by HttpClient.delete() method and logs a success message

to the console.

Summary

In summary, the code structure demonstrates how to perform CRUD operations (Create, Read, Update,

Delete) on a JSON file using Angular's HttpClient module.

Create (C):

 addItem(item: any): This method adds a new item to the JSON data file.

Read (R):

 getItems(): This method retrieves all items from the JSON data file.

 getItemById(id: number): This method retrieves a specific item by its ID from the JSON data file.

Update (U):

 updateItem(id: number, updatedItem: any): This method updates an existing item in the JSON data file by its ID.

Delete (D):

 deleteItem(id: number): This method deletes an existing item from the JSON data file by its ID.

These methods collectively enable the application to perform CRUD operations on the data stored in the

JSON file, allowing for efficient management and manipulation of the dataset.

Each method then subscribes to the service method's observable and handles the response accordingly. It

leverages observables and subscription to handle asynchronous HTTP requests and responses effectively.

//////////////////////

```
 <!-- Modal (Popup) -->
<div class="modal fade show custom-modal">
 <div class="modal-dialog modal-dialog fixed-size-modal">
 <div class="modal-content">
 <!-- Header section -->
 <div class="modal-header custom-header">
```

```
<div class="d-flex justify-content-between w-100">
<h4 class="modal-title">Filters</h4>
<div class="position-relative">
<div class="d-flex flex-wrap button-group">
<button type="button" class="btn btn-outline-danger custom-default-filter me-3"
[ngClass]="{'active': isDefault, 'clicked': isDefault}" (click)="isDefaultt()">
Default
</button>

<button type="button" class="btn btn-outline-danger custom-filter-of me-3"
[ngClass]="{'active': isFilterOff, 'clicked': isFilterOff}" (click)="isFilterOfff()">
Filter Off
</button>

<button type="button" class="btn btn-outline-danger custom-filter-list"
[ngClass]="{'active': isFilterList, 'clicked': isFilterList}" (click)="isFilterListt()">
Filter List
</button>

</div>

<span class="close" (click)="closeModal()">&times;</span>
</div>
</div>
</div>

<!-- Edit Condition Section -->
```

```
<div class="modal-body">

<h5 class="modal-title">Edit Condition</h5>


<div class="mb-2">

<label for="currentFilter" class="d-block">Name</label>

</div>


<div class="row align-items-center mb-3">

<div class="col-auto">

<input type="text" id="currentFilter" value="Current Filter" class="form-control custom-input">

</div>


<!-- Center the Share checkbox -->

<div class="col-auto d-flex align-items-center justify-content-center">

<div class="form-check custom-check">

<input class="form-check-input" type="checkbox" id="defaultCheck1">

<label class="form-check-label" for="defaultCheck1">Share</label>

</div>

</div>

</div>


<!-- Button Group -->

<mat-tab-group (selectedTabChange)="onTabChange($event)">

<mat-tab label="Conditions">

<nav class="navbar navbar-expand navbar-dark bg-dark my-2">

<div class="d-flex justify-content-center w-100">
```

```
<ul class="navbar-nav custom-navbar">
<li class="nav-item"><span class="nav-link">Column</span></li>
<li class="nav-item"><span class="nav-link">Condition</span></li>
<li class="nav-item"><span class="nav-link">Value</span></li>
<li class="nav-item"><span class="nav-link">Action</span></li>
</ul>
</div>
</nav>
<div class="example-small-box mat-elevation-z4">
<div *ngFor="let filter of filters; let i = index" class="d-flex align-items-center"
style="gap: 50px;">
<div class="btn-group">
<button type="button" class="btn btn-light dropdown-toggle"
data-bs-toggle="dropdown" aria-expanded="false">
{{ filter.selectedDateOption }}
</button>
<ul class="dropdown-menu">
<li><a class="dropdown-item" href="#"
(click)="onDateSelect(filter, 'Current Date'); $event.preventDefault()">Current
Date</a></li>
<li><a class="dropdown-item" href="#"
(click)="onDateSelect(filter, 'Last Week'); $event.preventDefault()">Last
Week</a></li>
<li><a class="dropdown-item" href="#"
(click)="onDateSelect(filter, 'Last Month'); $event.preventDefault()">Last
```

```
Month</a></li>
<li><a class="dropdown-item" href="#"
(click)="onDateSelect(filter, 'Last Year'); $event.preventDefault()">Last
Year</a></li>
</ul>
</div>
<div class="btn-group">
<button type="button" class="btn btn-light dropdown-toggle"
data-bs-toggle="dropdown" aria-expanded="false">
{{ filter.selectedComparisonOption }}
</button>
<ul class="dropdown-menu">
<li><a class="dropdown-item" href="#"
(click)="onComparisonSelect(filter, 'Less than'); $event.preventDefault()">Less
than</a></li>
<li><a class="dropdown-item" href="#"
(click)="onComparisonSelect(filter, 'Greater Than'); $event.preventDefault()">Greater
Than</a></li>
<li><a class="dropdown-item" href="#"
(click)="onComparisonSelect(filter, 'Less Than or Equal To'); $event.preventDefault()">Less
Than or Equal To</a></li>
<li><a class="dropdown-item" href="#"
(click)="onComparisonSelect(filter, 'Greater than or equal to');
$event.preventDefault()">Greater
than or equal to</a></li>
</ul>
```

```
</div>
<div class="centered-label">
<mat-form-field>
<mat-label *ngIf="!selectedDate">Choose a date</mat-label>
<input matInput [matDatepicker]="picker" (dateChange)="onDateChange($event)">
<mat-datepicker-toggle matSuffix [for]="picker"></mat-datepicker-toggle>
<mat-datepicker #picker></mat-datepicker>
</mat-form-field>
</div>
<span class="delete-icon" (click)="removeFilter(i)">
<i class="fas fa-trash"></i>
</span>
</div>
<hr>
<div class="d-grid gap-2 d-md-block mb-5">
<button class="btn btn-primary" type="button" (click)="addNewFilter()">+ Add New
Condition</button>
</div>
<hr>
</div>
</mat-tab>
<mat-tab label="Columns">
<div class="example-large-box mat-elevation-z4">
<div class="checkbox-grid">
<mat-checkbox *ngFor="let key of columnNames.allKeys"
```

```
[checked]="columnNames.selectedKeys.includes(key)"
(change)="onColumnSelectionChange($event, key)" color="warn">
{{ key }}
</mat-checkbox>
</div>
</div>
</mat-tab>

</mat-tab-group>
<!-- Divider (hr) after each row -->
<hr class="w-100">


<hr>
<div class="d-flex justify-content-between mb-2 align-items-center">
<div class="d-flex">
<button type="button" class="btn btn-outline-danger custom-save-filter me-3 custom-margin"
[ngClass]="{'active': isSaveF}" (click)="isSave()">
Save Filter
</button>

<button type="button" class="btn custom-delete-filter custom-margin"
[ngClass]="{'active': isActive}" (click)="isDelete()">
Delete Filter
</button>
</div>
```

```
<div>
<button type="button" mat-raised-button class="btn btn-danger custom-btn"
(click)="applyFilters()">Apply</button>
</div>
</div>
</div>
</div>
</div>
```

////////////////

```
<!-- Modal (Popup) -->
<div class="modal fade show custom-modal">
<div class="modal-dialog modal-dialog fixed-size-modal">
<div class="modal-content">
<!-- Header section -->
<div class="modal-header custom-header">
<div class="d-flex justify-content-between w-100">
<h4 class="modal-title">Filters</h4>
<div class="position-relative">
<div class="d-flex flex-wrap button-group">
<button type="button" class="btn btn-outline-danger custom-default-filter me-3"
[ngClass]="{'active': isDefault, 'clicked': isDefault}" (click)="isDefaultt()">
Default
</button>

<button type="button" class="btn btn-outline-danger custom-filter-of me-3"
```

```
[ngClass]="{'active': isFilterOff, 'clicked': isFilterOff}" (click)="isFilterOfff()">

Filter Off

</button>

<button type="button" class="btn btn-outline-danger custom-filter-list"

[ngClass]="{'active': isFilterList, 'clicked': isFilterList}" (click)="isFilterListt()">

Filter List

</button>


</div>


<span class="close" (click)="closeModal()">&times;</span>

</div>

</div>

</div>


<!-- Edit Condition Section -->

<div class="modal-body">

<h5 class="modal-title">Edit Condition</h5>


<div class="mb-2">

<label for="currentFilter" class="d-block">Name</label>

</div>


<div class="row align-items-center mb-3">

<div class="col-auto">

<input type="text" id="currentFilter" value="Current Filter" class="form-control custom-input">
```

```
</div>

<!-- Center the Share checkbox -->
<div class="col-auto d-flex align-items-center justify-content-center">
<div class="form-check custom-check">
<input class="form-check-input" type="checkbox" id="defaultCheck1">
<label class="form-check-label" for="defaultCheck1">Share</label>
</div>
</div>
</div>

<!-- Button Group -->
<mat-tab-group (selectedTabChange)="onTabChange($event)">
<mat-tab label="Conditions">
<nav class="navbar navbar-expand navbar-dark bg-dark my-2">
<div class="d-flex justify-content-center w-100">
<ul class="navbar-nav custom-navbar">
<li class="nav-item"><span class="nav-link">Column</span></li>
<li class="nav-item"><span class="nav-link">Condition</span></li>
<li class="nav-item"><span class="nav-link">Value</span></li>
<li class="nav-item"><span class="nav-link">Action</span></li>
</ul>
</div>
</nav>
<div class="example-small-box mat-elevation-z4">
<div *ngFor="let filter of filters; let i = index" class="d-flex align-items-center"
```

```
style="gap: 50px;">
<div class="btn-group">
<button type="button" class="btn btn-light dropdown-toggle"
data-bs-toggle="dropdown" aria-expanded="false">
{{ filter.selectedDateOption }}
</button>
<ul class="dropdown-menu">
<li><a class="dropdown-item" href="#"
(click)="onDateSelect(filter, 'Current Date'); $event.preventDefault()">Current
Date</a></li>
<li><a class="dropdown-item" href="#"
(click)="onDateSelect(filter, 'Last Week'); $event.preventDefault()">Last
Week</a></li>
<li><a class="dropdown-item" href="#"
(click)="onDateSelect(filter, 'Last Month'); $event.preventDefault()">Last
Month</a></li>
<li><a class="dropdown-item" href="#"
(click)="onDateSelect(filter, 'Last Year'); $event.preventDefault()">Last
Year</a></li>
</ul>
</div>
<div class="btn-group">
<button type="button" class="btn btn-light dropdown-toggle"
data-bs-toggle="dropdown" aria-expanded="false">
{{ filter.selectedComparisonOption }}
```

```
</button>
<ul class="dropdown-menu">
<li><a class="dropdown-item" href="#"
(click)="onComparisonSelect(filter, 'Less than'); $event.preventDefault()">Less
than</a></li>
<li><a class="dropdown-item" href="#"
(click)="onComparisonSelect(filter, 'Greater Than'); $event.preventDefault()">Greater
Than</a></li>
<li><a class="dropdown-item" href="#"
(click)="onComparisonSelect(filter, 'Less Than or Equal To'); $event.preventDefault()">Less
Than or Equal To</a></li>
<li><a class="dropdown-item" href="#"
(click)="onComparisonSelect(filter, 'Greater than or equal to');
$event.preventDefault()">Greater
than or equal to</a></li>
</ul>
</div>
<div class="centered-label">
<mat-form-field>
<mat-label *ngIf="!selectedDate">Choose a date</mat-label>
<input matInput [matDatepicker]="picker" (dateChange)="onDateChange($event)">
<mat-datepicker-toggle matSuffix [for]="picker"></mat-datepicker-toggle>
<mat-datepicker #picker></mat-datepicker>
</mat-form-field>
</div>
<span class="delete-icon" (click)="removeFilter(i)">
```

```
<i class="fas fa-trash"></i>
</span>
</div>
<hr>
<div class="d-grid gap-2 d-md-block mb-5">
<button class="btn btn-primary" type="button" (click)="addNewFilter()">+ Add New
Condition</button>
</div>
<hr>
</div>
</mat-tab>
<mat-tab label="Columns">
<div class="example-large-box mat-elevation-z4">
<div class="checkbox-grid">
<mat-checkbox *ngFor="let key of columnNames.allKeys"
[checked]="columnNames.selectedKeys.includes(key)"
(change)="onColumnSelectionChange($event, key)" color="warn">
{{ key }}
</mat-checkbox>
</div>
</div>
</mat-tab>

</mat-tab-group>
<!-- Divider (hr) after each row -->
<hr class="w-100">
```

```
<hr>
<div class="d-flex justify-content-between mb-2 align-items-center">
<div class="d-flex">
<button type="button" class="btn btn-outline-danger custom-save-filter me-3 custom-margin"
[ngClass]="{'active': isSaveF}" (click)="isSave()">
Save Filter
</button>

<button type="button" class="btn custom-delete-filter custom-margin"
[ngClass]="{'active': isActive}" (click)="isDelete()">
Delete Filter
</button>
</div>

<div>
<button type="button" mat-raised-button class="btn btn-danger custom-btn"
(click)="applyFilters()">Apply</button>
</div>
</div>
</div>
</div>
</div>
```

////////

```
import { CommonModule } from '@angular/common';
```

```
import { AfterViewInit, Component, EventEmitter, Inject, Output } from '@angular/core';
import { MatDialogRef, MAT_DIALOG_DATA } from '@angular/material/dialog';
import { ReactiveFormsModule } from '@angular/forms';
import { ControlCenterService } from '../../services/control-center.service';
import { HttpClientModule } from '@angular/common/http';
import { MatTabsModule } from '@angular/material/tabs';
import { MatCheckboxModule } from '@angular/material/checkbox';
import { MatDatepickerModule } from '@angular/material/datepicker';
import { MatInputModule } from '@angular/material/input';
import { filter } from 'rxjs';

interface Filter {
 selectedDateOption: string;
 selectedComparisonOption: string;
}

@Component({
 selector: 'app-filter-dialogbox',
 standalone: true,
 imports: [CommonModule, ReactiveFormsModule, HttpClientModule, MatTabsModule, MatCheckboxModule, MatInputModule, MatDatepickerModule],
 providers: [ControlCenterService],
 templateUrl: './filter-dialogbox.component.html',
 styleUrls: ['./filter-dialogbox.component.scss']
})
export class FilterDialogboxComponent implements AfterViewInit {
```

```
selectedButton: string | null = 'conditions'; // Track the selected button for navigation

selectedTab: string = 'Conditions';

filters: Filter[] = [{ selectedDateOption: 'Current Date', selectedComparisonOption: 'Less than' }]; // Initial filter

columnNames: { allKeys: string[], selectedKeys: string[] } = { allKeys: [], selectedKeys: [] };

onTabChange(event: any) {
this.selectedTab = event.tab.textLabel; // Update selected tab based on user selection
}

displayedColumns: string[] = [];
availableColumns: any;

constructor(
private dialogRef: MatDialogRef<FilterDialogboxComponent>,
@Inject(MAT_DIALOG_DATA) public data: any,
private controlCenterService: ControlCenterService
) { }

// selectedDate: any;
selectedDate: Date | null = null;
onDateChange(event: any) {
this.selectedDate = event.value;
}

@Output() filtersApplied: EventEmitter<string[]> = new EventEmitter<string[]>();
```

```
ngOnInit() {
this.controlCenterService.getRecords().subscribe(records => {
this.columnNames = this.controlCenterService.getKeysData();
console.log(this.columnNames);
});
}

isDefault: boolean = true;
isFilterOff: boolean = false;
isFilterList: boolean = false;
isSaveF: boolean = false;
isActive = false;

isApply = false;


isDefaultt() {
this.isDefault = true;
this.isFilterOff = false;
this.isFilterList = false;
console.log('Default Button clicked, active state:', this.isDefault);
}

isFilterOfff() {
this.isDefault = false;
this.isFilterOff = true;
this.isFilterList = false;
```

```
}

isFilterListt() {

this.isDefault = false;

this.isFilterOff = false;

this.isFilterList = true;

}

isSave() {

this.isSaveF = !this.isSaveF

console.log('Save Button clicked, active state:', this.isSaveF);

}

isDelete() {

this.isActive = !this.isActive;

console.log('delete button clicked, active state:', this.isActive);

}

isApplyy() {

this.isApply = !this.isApply;

console.log('Apply Button clicked, active state:', this.isApply);

}

ngAfterViewInit(): void {

// Initialize Bootstrap tooltips

const tooltipTriggerList = [].slice.call(document.querySelectorAll('[data-bs-
toggle="tooltip"]'));

tooltipTriggerList.forEach((tooltipTriggerEl) => {
```

```
new (window as any).bootstrap.Tooltip(tooltipTriggerEl);
});

}

// Handle selection from the date dropdown
onDateSelect(filter: Filter, option: string) {
filter.selectedDateOption = option;
}

// Handle selection from the comparison dropdown
onComparisonSelect(filter: Filter, option: string) {
filter.selectedComparisonOption = option;
}

// Add a new filter row
addNewFilter() {
this.filters.push({ selectedDateOption: 'Current Date', selectedComparisonOption: 'Less than' });
}

// Remove a filter row
removeFilter(index: number) {
this.filters.splice(index, 1);
}

// Handle button selection for navigation
```

```
selectButton(button: string) {
this.selectedButton = button;
}

// Close the modal
closeModal() {
this.dialogRef.close();
}

// Define setButtonStyle method to dynamically apply styles
setButtonStyle(button: string) {
return this.selectedButton === button
? { 'font-weight': 'bold', 'color': 'red' }
: { 'font-weight': 'normal', 'color': 'black' };
}

// Handle checkbox change event for column selection
// Handle checkbox change event for column selection
onColumnSelectionChange(event: any, key: string) {
if (event.checked) {
if (!this.columnNames.selectedKeys.includes(key)) {
this.columnNames.selectedKeys.push(key);
}
} else {
const index = this.columnNames.selectedKeys.indexOf(key);
if (index >= 0) {
this.columnNames.selectedKeys.splice(index, 1);
```

```
}

}

}

// Apply selected filters and emit event

applyFilters() {

console.log(this.columnNames.selectedKeys);

this.filtersApplied.emit(this.columnNames.selectedKeys);

this.dialogRef.close(this.columnNames);

}

}
```

Stay Consistent

Patch Values & Set Values in Angular:

In Angular, "patchValue" and "setValue" are methods used to update form controls within reactive forms. Reactive forms are a way to manage form data using observable streams and immutability. Below are examples for both methods with detailed explanations.

PATCH VALUES

The patchValue method allows you to update only a subset of form controls within a FormGroup or a FormControl. You provide an object containing the values you want to update. It is useful when you want to update specific parts of the form without affecting the entire form. If the provided object contains only a subset of the form controls, it will update only those controls, leaving the others unchanged. It’s more flexible when you don’t need to provide values for all form controls.

Example:

typescript

Copy code

```
ngOnInit() {
```

```
this.initializeForm();

// Assume this is the data fetched from API
const formData = {
  name: 'John Doe',
  age: 30,
  gender: 'male',
  address: '123 Main St',
  diagnosis: 'Headache',
  contact: '1234567890',
  email: 'john@example.com',
  personalHistory: 'None',
  familyHistory: 'None',
  painScore: '5'
};

// Use patchValue to update only specific fields
this.medicalForm.patchValue({
  name: formData.name,
  age: formData.age,
  gender: formData.gender,
  address: formData.address,
  diagnosis: formData.diagnosis,
  contact: formData.contact,
  email: formData.email
```

```
  });
}
```

SET VALUES

The setValue method is used to update the entire form with new values. You provide an object containing values for all form controls within the FormGroup. It requires a complete set of values for all controls in the form. If any control is not provided a value, it will be reset to its default state. It’s useful when you have a complete set of values and want to update the entire form. It enforces that you provide values for all controls, ensuring the form is in a valid state after the update.

Example:

typescript

Copy code

```
ngOnInit() {
  this.initializeForm();
  // Assume this is the data fetched from API
  const formData = {
    name: 'John Doe',
    age: 30,
    gender: 'male',
    address: '123 Main St',
    diagnosis: 'Headache',
    contact: '1234567890',
    email: 'john@example.com',
    personalHistory: 'None',
    familyHistory: 'None',
    painScore: '5'
  };
```

```
  // Use setValue to update the entire form

  this.medicalForm.setValue(formData);

}
```

Summary

The key difference is that when updating a single form control:

- patchValue() allows selective updates.
- setValue() requires a complete set of values for all controls within the form.

Use patchValue() when you want to update specific parts of the form, and use setValue() when you have a complete set of values for the entire form.

Real World Use Cases:

- **User Profile Forms**: Users may update specific details such as their email address or phone number without altering other profile information.
- **Order Forms**: When editing an order, only certain fields like shipping address or payment method may need updating, while other details remain intact.
- **Multi-Step Forms**: In multi-step forms, each step may update a subset of the form data until the final submission, where all values are set before processing the form.