**Angular Tutorial**

1. **EVOLUTION OF ANGULAR FRAMEWORK**

Angular is a popular open-source and web application framework for building dynamic and single-page applications (SPAs). It is developed and maintained by Google. Angular provides a comprehensive ecosystem for developing client-side applications, including tools for routing, forms, state management, and more.

1. **AngularJS (2009)**: Angular's story begins with the release of AngularJS in 2009. It was developed by Google and initially led by Misko Hevery and Adam Abrons. AngularJS introduced the concept of two-way data binding, directives, and a modular approach to building web applications.
2. **Angular 2 (2016)**: In 2016, Angular underwent a major rewrite, resulting in Angular 2. This version was a complete departure from AngularJS. It introduced a component-based architecture and used TypeScript as its primary programming language. It aimed to improve performance and maintainability.
3. **Angular (2016 - Present)**: After the release of Angular 2, subsequent versions adopted the name "Angular" without a Angular 4, Angular 5, and so on.
4. **Angular CLI (2016)**: The Angular CLI (Command Line Interface) was introduced to streamline the process of creating, building, and deploying Angular applications. It made it easier for developers to generate components, modules, services, and more.
5. **Angular Ivy (2019)**: Angular Ivy is a significant update to the Angular framework also simplifies the development of dynamic components and enhances tree-shockability.  
   **Tree-shockability:** is a process wherein the unused code, like unused functions are removed during the build phase.  
   Ivy engine is default enable from the version Angular 9.
6. **Angular Versioning (2020)**: Starting from version 9 in 2020, Angular adopted a predictable release schedule. Major updates are scheduled every six months, making it easier for developers to stay up to date with the latest features and improvements.
7. **Angular Ecosystem**: Angular has a robust ecosystem that includes libraries, tools, and best practices. Angular Material provides pre-designed UI components, while libraries like NgRx offer state management solutions. Developers can take advantage of features like server-side rendering, progressive web app capabilities, and more.
8. **Community and Adoption**: Angular has a large and active community of developers and organizations. It is widely used in industry applications, and many companies have adopted Angular for their web development needs.
9. **Future Development**: Angular continues to evolve with a focus on improving performance, developer experience, and integration with modern web technologies.
10. **SETTING UP YOUR DEVELOPMENT ENVIRONMENT**

Setting Up Your Development Environment Node.js and npm Installation Angular CLI Installation Creating Your First Angular App.

1. **Node.js and npm Installation**

Node.js is a JavaScript runtime that allows you to run JavaScript on the server. npm (Node Package Manager) is a package manager for JavaScript.

* **Visit the official Node.js website at** [**https://nodejs.org/**](https://nodejs.org/)
* Download the LTS (Long Term Support) version for your OS (Windows, macOS, Linux).
* Follow the installation instructions for your specific OS.

To verify that Node.js and npm are installed correctly, open a terminal or command prompt and run the following commands:

node -v

npm -v

You should see the versions of Node.js and npm displayed in the terminal.

**Node.js:** Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to execute server-side JavaScript code. It is built on the V8 JavaScript runtime engine, the same engine that powers the Google Chrome browser.

**NPM:** npm stands for "Node Package Manager." It is the default package manager for Node.js, allowing developers to discover, install, and manage packages of code written in JavaScript. npm is a command-line tool that comes bundled with Node.js, and it also provides an online repository for public and private packages, making it easy for developers to share and reuse code.

1. **Angular CLI Installation**

The Angular CLI (Command Line Interface) is a powerful tool for creating, building, and managing Angular applications.

* Open a terminal or command prompt.
* To install the Angular CLI globally on your system, run the following command:

npm install -g @angular/cli

The -g flag indicates a global installation.

* Once the installation is complete, you can verify the Angular CLI version by running:

ng –version

You should see information about the installed Angular CLI version.

1. **Creating Your First Angular App**

* To create a new Angular app, navigate to the directory where you want to create your project using the terminal or command prompt.
* Run the following command to generate a new Angular app:

ng new my-angular-app

Replace my-angular-app with the name you want for your project. The Angular CLI will prompt you to answer several questions, such as whether you want to include Angular routing or which stylesheets you prefer (CSS, SCSS, etc.). You can choose the default options by pressing Enter.

* Once the project is generated, navigate to the project directory:

cd my-angular-app

* To start a development server and see your app in action, run the following command:

ng serve

This will build and serve your Angular app. By default, it will be available at[**http://localhost:4200/**](http://localhost:4200/)**.**

1. **COMPONENTS**

A component is a fundamental building block of an application's user interface. It represents a part of the UI with its own logic and data. Components are reusable and can be composed together to create complex applications.

1. **Creating a** **Component:**

To create a component in Angular, you can use the Angular CLI's **ng generate component** command, which will generate the necessary files and boilerplate code. For example, to create a component called "user-profile," you would run:

ng generate component user-profile

This command creates a folder with the component's name and generates several files, including a TypeScript file for the component's logic, an HTML file for the template, and a CSS file for styling.

1. **Anatomy of a Component:**

Here's a breakdown of the key parts of an Angular component:

**a) Component Class (user-profile.component.ts):** This TypeScript file defines the component class, its properties, and methods. It's where the component's logic resides.

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

@Component({

selector: 'app-user-profile',

templateUrl: './user-profile.component.html',

styleUrls: ['./user-profile.component.css']

})

export class UserProfileComponent {

name = 'John Doe';

}

**b) Component Template (user-profile.component.html):** This file defines the HTML structure of the component's view.

<div>

<h2>User Profile</h2>

<p>Name: {{ name }}</p>

</div>

**c) Component Styles (user-profile.component.css)**: This file contains the component's CSS styles, providing encapsulation to prevent styles from affecting other components.

div {

border: 1px solid #ccc;

padding: 10px;

background-color: #f7f7f7;

}

**d) Component Metadata**: The **@Component** decorator provides metadata about the component. It specifies the component's selector (HTML tag), template file, and style file.

1. **Using a Component:**

To use a component in an Angular application, you can simply include its selector tag in another component's template. For example:

<app-user-profile></app-user-profile>

This renders the UserProfileComponent within the template of another component.

1. **Data Binding in Components:**

Components in Angular support various forms of data binding:

* **Interpolation**: You can display component properties within the template using double curly braces **{{ property }}**.
* **Property Binding**: Bind an HTML element's property to a component property. For example, binding the **src** attribute of an image:

<img [src]="imageUrl">

* **Event Binding:** Listen to events and trigger actions in response. For example, binding a button click event:

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

1. **MODULES**

Angular Modules are a way to organize and structure your Angular application. They group related components, directives, services, and other code into cohesive blocks of functionality. Modules help to keep your application maintainable and allow for easy collaboration among developers.

**Creating an Angular Module**

You can create an Angular Module using the Angular CLI or manually.

**Using Angular CLI:**

To create a new module using the Angular CLI, run:

ng generate module module-name

This command generates a new module file (module-name.module.ts) and registers it in the main app.module.ts file.

**Manual Creation:**

You can create a module manually by following these steps:

1. Create a new TypeScript file, e.g., module-name.module.ts, which should export a class with the @NgModule decorator. Here's an example:

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

@NgModule({

declarations: [/\* List of components, directives, and pipes \*/],

imports: [/\* List of imported modules \*/],

providers: [/\* List of services \*/],

exports: [/\* Exported components, directives, and pipes \*/],

})

export class ModuleNameModule { }

1. You'll need to import the module where it is used, usually in the main app.module.ts file.

**Module Properties**

* **declarations:** This property contains a list of components, directives, and pipes that are part of this module. Components declared here are private to the module by default.
* **imports:** In this property, you specify the other modules that this module depends on. For example, CommonModule is a common import for many modules.
* **providers:** You can define services that are scoped to the module. These services will be available for injection to components within the module.
* **exports:** Use this property to specify which components, directives, and pipes should be made public for other modules to use.

1. **Using Modules**

To use modules in Angular, you typically import them into your main app.module.ts or other feature modules. For example:

import { ModuleNameModule } from './module-name/module-name.module';

@NgModule({

declarations: [/\* your app's components \*/],

imports: [ModuleNameModule, /\* other modules \*/],

bootstrap: [/\* your root component \*/],

})

export class AppModule { }

1. **DATA BINDING**

Data binding is a core concept in Angular, allowing you to establish a connection between the data in your component (the model) and the view (the template). There are four types of data binding are.

1. **Interpolation:**

Interpolation is a one-way data binding technique used to display dynamic data in the template (view). It involves using double curly braces {{ }} to embed expressions in the template, which are replaced by the actual data when rendered.

<!-- In your component -->

<p>{{ greeting }}</p>

<!-- In your component class -->

export class MyComponent {

greeting: string = 'Hello, Angular!';

}

In this example, the value of the greeting property in the component class is displayed in the template.

1. **Property Binding:**

Property binding allows you to set element properties (attributes) with component data. You use square brackets [] to bind a component property to an element property. This is often used to change HTML element attributes based on component data.

<!-- In your component -->

<img [src]="imageUrl">

<!-- In your component class -->

export class MyComponent {

imageUrl: string = 'assets/my-image.jpg';

}

In this example, the src attribute of an image element is bound to the imageUrl property in the component class.

1. **Event Binding**

Event binding is used to listen to client side events (e.g., clicks, keypresses) in the template and trigger actions in response. You use parentheses () to bind a component method to an event.

<!-- In your component -->

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

<!-- In your component class -->

export class MyComponent {

onButtonClick() {

alert('Button clicked!');

}

}

1. **Two-Way Binding**

Two-way binding combines property binding and event binding, allowing you to both display and update data. It's used with forms and form elements like input fields.

To achieve two-way binding, you use the [(ngModel)] directive, which is part of the FormsModule for reactive forms.

<!-- In your component -->

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

<!-- In your component class (with FormsModule imported) -->

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

@NgModule({

imports: [FormsModule],

})

export class MyComponent {

name: string = 'John Doe';

}

1. **DIRECTIVES**

**Definition:** Directives are instructions for DOM elements that extend its functionality like manipulating DOM dynamically or adding behaviors to DOM elements. Angular comes with built-in directives and allows you to create custom directives.

Directives are markers in the DOM that tell Angular to do something with an element or its children. They are attributes that begin with the ng- prefix, such as **ngIf, ngFor, and ngStyle**. Directives can be categorized into three types:

1. **Component Directives:** These are the most common directives and are associated with Angular components. Components encapsulate the logic and view of a part of the user interface. They are reusable and can be composed together to build complex applications.
2. **Structural Directives:** These change the structure of the DOM by adding or removing elements. Examples include \*ngIf, \*ngFor, and \*ngSwitch.
3. **Attribute Directives:** These change the appearance or behavior of an element. Examples include ngStyle, ngClass, and ngModel.
4. **Built-in Angular Directives:**

Angular provides a set of built-in directives to handle common tasks. Here are some of the most commonly used ones:

1. **ngIf:** Conditionally renders an element based on a truthy or falsy condition.

<div \*ngIf="isUserLoggedIn">Welcome, {{ username }}</div>

1. **ngFor:** Iterates over an array or iterable and generates HTML for each item.

<ul>

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

</ul>

1. **ngSwitch:** Conditionally renders elements based on multiple values.

<div [ngSwitch]="status">

<p \*ngSwitchCase="'active'">User is active.</p>

<p \*ngSwitchCase="'inactive'">User is inactive.</p>

<p \*ngSwitchDefault>User status is unknown.</p>

</div>

1. **ngStyle:** Dynamically sets the style of an element based on an object or expression.

<div [ngStyle]="{ 'color': isImportant ? 'red' : 'blue' }">Styled Text</div>

1. **ngClass**: Adds or removes CSS classes from an element based on a condition.

<div [ngClass]="{'active': isActive, 'highlighted': isHighlighted}">Styled Text</div>

1. **ngModel**: Provides two-way data binding for form elements, often used in conjunction with reactive forms.

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

1. **Creating Custom Directives:**

You can create your own custom directives in Angular. These directives allow you to encapsulate complex behavior and make your code more modular and reusable.

Here's a basic structure for creating a custom directive:

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

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private el: ElementRef) {}

@HostListener('mouseenter') onMouseEnter() {

this.highlight('yellow');

}

@HostListener('mouseleave') onMouseLeave() {

this.highlight(null);

}

private highlight(color: string | null) {

this.el.nativeElement.style.backgroundColor = color;

}

}

This example creates a custom directive called appHighlight that changes the background color of an element when hovered over.

1. **Using Custom Directives**

To use a custom directive in your template, declare it in the appropriate module. For example:

@NgModule({

declarations: [HighlightDirective, /\* other components and directives \*/],

})

export class MyModule { }

Then, you can apply the directive to an HTML element:

<p appHighlight>Hover over me to see the custom directive in action!</p>

**Directives Summary**

* Directives in Angular are markers that instruct the framework to manipulate the DOM or add behavior to elements.
* Angular provides built-in directives for common tasks like conditional rendering, iteration, and styling.
* You can create custom directives to encapsulate complex behavior and make your code more modular and reusable.
* Directives are a key aspect of Angular that enable you to build dynamic and interactive web applications. They make your code more expressive, readable, and maintainable.

**Property & Event Binding Overview:**

**1. Binding to Custom Property:  
Property Binding using @Input Decorator:**   
@Input decorator is used to allow a component to receive data from its parent component. It enables the parent component to bind a value to a property of the child component.

**Assigning an Alias to Custom Properties:** @Input('aliasedInput') aliasedProperty: string;

**2. Building Custom Events:**

**Event Binding using @Output decorator** is used to create custom events in a child component that can be listened to by its parent component. This allows the child component to emit events and the parent component to respond to those events.

A custom event in Angular is a user-defined event that allows communication between components. It is implemented using the   
**EventEmitter** class to emit events from a child component and listen for these events in a parent component. The object of EventEmitter from child component can carry data as well when it’s fired. For that we declare EventEmitter with object declaration and emit it with data in the functions.

**Example:**  
Child component:

      <button

        class="btn btn-primary"

        (click)="onAddServer()">Add Server</button>

      <button

        class="btn btn-primary"

        (click)="onAddBlueprint()">Add Server Blueprint</button>

 @Output() serverCreated = new EventEmitter<{serverName: string, serverContent: string}>();

  @Output() blueprintCreated = new EventEmitter<{serverName: string, serverContent: string}>();

onAddServer() {

      this.serverCreated.emit(

        {serverName: this.newServerName, serverContent: this.newServerContent}

      );

  }

  onAddBlueprint() {

    this.blueprintCreated.emit(

      {serverName: this.newServerName, serverContent: this.newServerContent}

    );

  }

Parent component:

<app-cockpit (serverCreated) = "onServerAdded($event)"

               (blueprintCreated) = "onBlueprintAdded($event)">

 onServerAdded(serverData: {serverName: string, serverContent: string}) {

    this.serverElements.push({

      type: 'server',

      name: serverData.serverName,

      content: serverData.serverContent

    });

  }

  onBlueprintAdded(serverData: {serverName: string, serverContent: string}) {

    this.serverElements.push({

      type: 'blueprint',

      name: serverData.serverName,

      content: serverData.serverContent

    });

  }

**View Encapsulation**:  
It’s a concept that defines how the styles defined in a component affect the elements in the component's view and how they interact with styles from other components. Angular provides three types of view encapsulation: **Emulated, Native, and None**.

**1. Emulated View Encapsulation (Default):** This is the default view encapsulation strategy in Angular. It emulates the shadow DOM by adding unique attributes to the HTML elements within the component. Styles are scoped to the component, preventing them from leaking out and affecting other components.

@Component({

selector: 'app-example',

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

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

encapsulation: ViewEncapsulation.Emulated,

})

**2. Native View Encapsulation:** Native view encapsulation uses the browser's native shadow DOM to isolate the styles of a component. This method is closer to the actual shadow DOM specification but may not be fully supported in all browsers.

**3. None View Encapsulation:**

None disables view encapsulation entirely. Styles from the component can affect the global styles and vice versa. While this approach gives the most freedom, it's generally recommended to use encapsulation to prevent unintended style conflicts.

**Local References in Templates:**

Local references in templates allow you to create a reference to a DOM element (any HTML element) or Angular component within the template itself. These references are defined using the hash symbol (**#**) followed by a name. Once defined, you can access and interact with the referenced element or component in the template or in the corresponding component class.

<label>Server Name</label>

<input type="text" class="form-control"

      #serverNameInput>

<button class="btn btn-primary" (click)="onAddServer(serverNameInput)">Add Server</button>

**ViewChild:**

ViewChild Property decorator that configures a view query. The change detector looks for the first element or the directive matching the selector in the view DOM. If the view DOM changes, and a new child matches the selector, the property is updated. **Declaring ViewChild:**

<input type="text" class="form-control"

             #serverContentInput>

@ViewChild('serverContentInput') serverContentInput: ElementRef;

**Using ViewChild:**

this.newServerContent = this.serverContentInput.nativeElement.value;

**ng-content:**

ng-content is a directive used to project content from the parent component into a specified location within the child component's template. It enables the creation of flexible and reusable components by allowing the parent component to pass content (HTML, text, or other Angular components) to the child component.

**Parent Component:**

<app-server-element

        \*ngFor="let serverElement of serverElements" srvElement]="serverElement">

        <p>

          <strong \*ngIf="serverElement.type === 'server'" style="color: red">

          {{ serverElement.content }}</strong>

          <em \*ngIf="element.type === 'blueprint'">{{ serverElement.content }}</em>

        </p>

      </app-server-element>

Child Component (app-server-element):

<div class="panel panel-default">

  <div class="panel-heading">{{ element.name }}</div>

  <div class="panel-body">

   <ng-content></ng-content>

  </div>

</div>

**Understanding Component Life Cycle:**

**1. ngOnChanges:** Its executed right at the beginning when new component is created and called after a bound input property changes.

**2. ngOnInit:** Called when component has been initialized. After the constructor.

**3. ngDoCheck:** Called during every change detection run

**4. ngAfterContentInit:** Called after content (ng-content) has been projected into View.

**5. ngAfterContentChecked:** Called every time the projected content has been checked.

**6. ngAfterViewInit:** Called after component’s view (and child’s view) has been initialized.

**7. ngAfterViewChecked:** Called every time view (and child views) has been checked

**8. ngOnDestroy:** Called once component is about to destroyed.

**Angular Directives Part 2:**

**Attribute directive:** Attribute directives change the appearance or behavior of an element, component, or another directive.

* In attribute directive we can give the value inside { } and Boolean expressions after :

**Syntax:**

<p [class]="{ className : selectedFeature != null}"> Hi </p>

**Creating Basic attribute directive:**

Inside the directive.ts file we use **ElementRef** and **Renderer2** to interact with the DOM.

**File name:** basic-directive/basic-highlight.directive.ts

  intervalId;

  colorSwap: boolean = false;

  constructor(private eleRef: ElementRef) { }

  ngOnInit(): void {

    this.intervalId = setInterval(() => {

    this.eleRef.nativeElement.style.backgroundColor = this.colorSwap ? 'white' : 'lightgreen';

    this.colorSwap = !this.colorSwap;

    }, 500);

  }

Using ElementRef we can access the element, one way is in ngOnInit hook, and then change styles, etc.

**Note:** Accessing elements directly using ElementRef is not always a best practice, there are other ways to do this. Which is **Renderer**. This make simpler to change elements properties like style, etc.

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

  ngOnInit(): void {

    this.intervalId = setInterval(() => {

      this.renderer.setStyle(this.eleRef.nativeElement, 'background-color',

        ( this.colorSwap ? 'white' : '#FFCCCB'));

      this.colorSwap = !this.colorSwap;

    }, 500);

  }

@HostBinding and @HostListener are two decorators in Angular that can be really useful in custom directives.

**Host Listener:**

**Definition:** It’s a decorator that declares DOM event to listen for and provides a handler method to run when that event is occurred. Angular invokes the supplied handler method when host element emits the specified event and updates the bound element with the result.

  @HostListener('mouseenter') mouseOver(eventData: Event) {

    this.renderer.setStyle(this.eleRef.nativeElement, 'background-color',

      'lightgreen');

  }

**Host Binding:**

**Definition:** It’s a decorator that allows you to set the properties of the host element from the directive class. When change is detected in biding angular will update the host element of the directive.

  @HostBinding('style.backgroundColor') backgroundColor: string;

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

  @HostListener('mouseenter') mouseOver(eventData: Event) {

      this.backgroundColor = 'lightgreen';

  }

**Directive properties:** We can change directive properties as well, so that it don’t have to be static. In the host element we can change directive properties by defining first in directive class with @Input decorator like below.

  @Input() defaultColor: string = 'white';

  @Input() hoverColor: string = 'lightgreen';

Note that we need to set the *defaultColor* in ngOnInit method for it to set the property when the host element is loaded for the first time like shown below.

  ngOnInit(): void {

    this.backgroundColor = this.defaultColor;

  }

There are two ways to write the properties of directive as shown below.   
[hoverColor]="'red'" and hoverColor="green"

<a appHoverHighlight [defaultColor]="'gray'" [hoverColor]="'red'"

      class="pad-10">Hover Highlight</a>

<a appHoverHighlight defaultColor="yellow" hoverColor="green"

      class="pad-10">Hover Highlight</a>

**Creating Own Structural Directive:**Just like normal directive with few differences, like we use TemplateRef, ViewContainerRef. We use @Input to take the condition in this particular unless directive and use set to satisfy the condition here. But @Input property name must match with directive name.

@Directive({

  selector: '[appUnless]'

})

export class UnlessDirective {

  @Input() set appUnless(condition: boolean) {

    if(!condition) {

      this.vcRef.createEmbeddedView(this.templateRef);

    } else {

      this.vcRef.clear();

    }

  }

  constructor(private templateRef: TemplateRef<any>,

    private vcRef: ViewContainerRef) {

  }

}

<ul \*appUnless="!showEven">

    <li class="list-group-item" style="width: 100px;"

        \*ngFor="let num of evenNumbers">

        {{ num }}

    </li>

</ul>

**Creating Dropdown Directive:**

@Directive({

    selector: '[appDropdown]'

})

export class DropdownDirective {

    @HostBinding('class.open') isOpen: boolean= false;

    @HostListener('click') toggleOpen() {

        this.isOpen = !this.isOpen;

    }

}

<li class="dropdown" appDropdown> Dropdown code <li>

open is class which we use it to toggle the dropdown here. In the above code drop closes only when you click on it, it will be open if you click anywhere else. To close it from anywhere use the below code.

    @HostListener('document:click', ['$event']) toggleOpen(event: Event) {

        this.isOpen = this.eleRef.nativeElement.contains(event.target) ?

            !this.isOpen : false;

    }

    constructor(private eleRef: ElementRef){ }

1. **SERVICES & DEPENDENCY INJECTION**

**Services Definition:** Services allows us to share data and functionality across different parts of the application. Services are classes that can be injected into components and other services, and they are used to encapsulate logic that is not specific to a particular component.

Services are a good way to organize and share code that is not directly related to the UI of your application. They can be used to abstract away complex logic, such as calling a REST API or performing calculations, and make it available to multiple components.

**Dependency injection:** Dependency injection is a design pattern in software development that involves providing dependencies (i.e., services that a class relies on) from external sources rather than creating them within the class.

**Example of simple service injecting into component:**

@Injectable({

  providedIn: 'root',

})

export class LoggingService {

  logMessage(msg: string) { console.log('Log Message is ' + msg); }

}

**Component file:**

@Component({ ... })

  onAdd() {

    this.ingredientAdded.emit(this.selectedIngredient);

    this.logService.logMessage('New ingredient added')

  }

**Hierarchical Injector:**

Angular dependency injector is hierarchical injector, that means if we provide a service in one component in the application, angular framework creates one instance and passes that instance of service to that component and all the child components. There are three levels to this.

* + **AppModule:** Same instance of service is available Application-wide including all components, directives, services (Provider’s array)

To achieve this, add this in service @Injectable({ providedIn: 'root' }) or in any module file we can add in providers array providers: [TestService], both gives same result.

* + **AppComponent:** Same instance of service is available in all components but no other services
  + **Any other component:** Same instance of service is available for the component and all the child components.

**Injecting Services into Services:** When we add @Injectable( ) to the service A, then we can inject other services to service A. To use other service by creating a private object in constructor just like you do it in component.

**Using Services for Cross-Component Communication:**

1. Create an eventX in the serviceA.  
   eventX = new EventEmitter<string>();
2. In ComponentA injected with serviceA, emit the event eventX  
   this.serviceA.eventX.emit(stringValue)
3. In ComponentB subscribe the event to get the stringValue.

this.serviceA.eventX.subscribe(

(stringValue: string) => alert(‘Status is ’ +stringValue)

);

**Benefits by using new syntax:**

Instead of adding a service class to the providers[] array in AppModule , you can set the following config in @Injectable(). Services can be loaded lazily by Angular (behind the scenes) and redundant code can be removed automatically. This can lead to a better performance and loading speed - though this really only kicks in for bigger services and apps in general.

**Note:** Always use corresponding service, when you are calling some service method. Best practice is to call other service methods from component related service.

1. **ROUTING**

**Definition:** Routing is mechanism in angular, which allows the user to create a single page application with multiple views and allows navigation between them based on the user's interaction or URL changes. Routing system does not reload entirely when the user navigates between different sections of the application.

* We register router configurations in app.module.ts file

**1. Setting up router configurations:**  
1. Create constant array with path and component information, inside app.module.ts file

const routes: Routes = [

  {path: '', component: HomeComponent},

  {path: 'users', component: UsersComponent},

  {path: 'servers', component: ServersComponent}

]

2. Now add the RouterModule in imports array by giving routes constant to forRoot function.

  imports: [

    BrowserModule,

    FormsModule,

    RouterModule.forRoot(routes),

  ]

3. In app.component.ts file, *<router-outlet></router-outlet>* tag will be used to show corresponding component for that particular path. For example when we use “localhost:4200/users”, *router-outlet* will be replaced with users component.

**2. RouterLink Directive:** The routerLink directive is used to create links that navigate to a specific route.

        <li role="presentation" class="active"><a routerLink="/">Home</a></li>

        <li role="presentation"><a routerLink="/servers">Servers</a></li>

        <li role="presentation"><a [routerLink]="['/users']">Users</a></li>

**3. Understanding Navigation Paths:**

* If we use / Infront of the path like “/servers”, it will use absolute path.
* If we remove the / Infront of the path like “servers”, it will use relative path, that means “/servers” will be appended to currently loaded path.
* That means, If we are in “localhost:4200/servers”, then we click “servers” link inside of the servers component, it will try to redirect to “localhost:4200/servers/servers”
* If we use “servers” or “/servers” inside app.component links, which is also not loaded by router module, it will always loads “localhost:4200/servers” because it’s the root component. When we go a level below this we should use / accordingly.
* In case of 3rd point above, We can also use ../ to take the path to go up one level, “../servers”, it will load “localhost:4200/servers”.

[**Lecture Link**](https://www.udemy.com/course/the-complete-guide-to-angular-2/learn/lecture/6656286#overview)

**4. RouterLinkActive:** This directive is used to apply a CSS class to an HTML element based on whether the associated router link is currently active or not. We can add whatever the class we want. It's a helpful way to visually highlight the active link in your application's navigation.

<li role="presentation" routerLinkActive="active"><a routerLink="/">Home</a></li>

* *[routerLinkActiveOptions]* is directive property that provides configuration options for the RouterLinkActive directive.
  + exact (optional): This property is a boolean that, when set to true, indicates that the router link should be considered active only when the URL matches the link exactly. If not specified or set to false, the link is considered active when the URL is a prefix of the link.

<li role="presentation" routerLinkActive="active"

[routerLinkActiveOptions]="{ exact: true }">

          <a routerLink="/">Home</a>

</li>

* + queryParams (optional): This property is a string or string array that allows you to specify the query parameters that should be considered when determining if the router link is active. If specified, the router link will only be considered active if the specified query parameters match.
  + fragment (optional): This property is a string that allows you to specify the fragment (hash) that should be considered when determining if the router link is active. If specified, the router link will only be considered active if the specified fragment matches.

<a routerLink="/" routerLinkActive="active" [routerLinkActiveOptions]="{ exact: true }">Home</a>

<a routerLink="/about" routerLinkActive="active"

[routerLinkActiveOptions]="{ exact: true, queryParams: 'page=1' }">About</a>

<a routerLink="/contact" routerLinkActive="active" [routerLinkActiveOptions]="{ fragment: 'section1' }">Contact</a>

**5. Navigating Pragmatically:**

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

constructor(private router: Router) { }

  onLoadServers() {

    // complex calculation

    this.router.navigate(['/servers']);

  }

**6.** **Using Relative Paths in Programmatic Navigation:**

In navigate method we can pass second argument which takes an object, that is {relativeTo : objectOfActivatedRoute}. We will be using Router and ActivatedRoute.

  constructor(private serversService: ServersService,

    private router: Router, private route: ActivatedRoute) { }

  onReloadServers(){

    this.router.navigate(['/servers'], {relativeTo: this.route});

  }

**7. Parameters in the path:**

We can include parameters in the path using route parameters. It allows us to pass dynamic values, which replaces the parameters in the path.

1. Define route parameters: We us : as prefix in the path to define parameters.

  {path: 'users/:id', component: UserComponent},

1. Fetching route parameters: We use ActivatedRoute to get currently activated router information which also includes the parameters passed. To get particular parameters we use this.route.snapshot.params['id']. To ok to use this for initial loading, but when you change the parameters afterwords it will not change the data, and it’s default behavior. When we click below link after the initial load of users, It will change the URL but it will not change the data.

<a [routerLink]="['/users', user.id, user.name]">Load Anna (10)</a>

constructor(private route: ActivatedRoute) { }

selectedId: string;

  ngOnInit() {

    selectedId = this.route.snapshot.params['id'],

  }

1. Use observable called params and subscribe, where we will get params i.e., parameters.

    this.route.params.subscribe((params) => {

      this.user.id = params['id'];

      this.user.id = params['name'];

    });

1. Angular will unsubscribe the subscription when the component is destroyed, but if we create our own observable we should unsubscribe to it in ngOnDestroy method.

**8. Query parameters & fragments in path:**

1. We can declare the path first like shown below.

  {path: 'servers/:id/edit', component: EditServerComponent}

1. Defining Using routerLink:

<a [routerLink]="['/servers', server.id, 'edit']" [queryParams]="{allowEdit: 1}"

fragment="loading" ngFor="let server of servers">

        {{ server.name }}

</a>

Next to routerLink we use queryParams, fragment. Both are bindable properties of routerLink. queryParams will take key value pair and fragment will take a string. Above code will redirect to localhost:4200/servers/5/edit?allowEdit=1#loading

1. Navigating programmatically with queryParams, fragments: To navigate programmatically from component, we use ActivatedRoute navigate method and pass key value pair for queryParams and string to fragment like shown below. It will redirect to same link like above.

  onLoadServers(id: number) {

    // complex calculation

    this.router.navigate(['/servers', id, 'edit'],

    { queryParams: { allowEdit: 1 }, fragment: 'loading'});

  }

1. Retrieving query parameters and fragments:

Just like we get parameters, we can get query parameters, fragments by using snapshot and subscribe.

    console.log('queryParams ' +this.route.snapshot.queryParams['allowEdit']);

    console.log('fragment ' + this.route.snapshot.fragment);

    this.route.queryParams.subscribe(params => {

      // do something

    });

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

      // do something

    });

**9. Setting up Child Routes or Nested Routes:**

In the below example we can see “servers” are duplicate, instead of using this we can nest them.

  {path: 'servers', component: ServersComponent},

  {path: 'servers/:id', component: ServerComponent},

  {path: 'servers/:id/edit', component: EditServerComponent}

In the below example we have nested the routes and removed “servers” in the path, because it will take from the parent. When we do this *<router-outlet>* must be added inside of the parent component to display its child components which was given in route configurations (ServerComponent, EditServerComponent), in this case ServersComponent. The *<router-outlet>* in app-component is for reserved only for routes on top level.

{ path: 'servers', component: ServersComponent,

    children: [

      { path: ':id', component: ServerComponent },

      { path: ':id/edit', component: EditServerComponent },

    ],

  }

**10. Configuring Handling of Query Parameters:**

*queryParamsHandling* option is used in the Router.navigate method to control how query parameters are handled when navigating to a new route. When we navigate to new route, we can opt in to not to lose the query parameters. The queryParamsHandling option has two possible values:  
 1. 'merge': This option merges the current query parameters with the new ones.

2. 'preserve': This option preserves the existing query parameters, and the new ones are ignored.

this.router.navigate(['edit'],

    { relativeTo: this.route, queryParamsHandling: 'preserve' });

**11. Redirecting & Wildcard (like 404 page):**

**1. Redirecting request:**

We can redirect using redirectTo, there we give whatever the new path we want as shown below.

  { path: 'not-found', component: PageNotFoundComponent},

  { path: 'something', redirectTo: 'not-found'}

**2. Setting up route which handle routes we did not setup:**

By using \*\* in the path we say it’s a wildcard for unknown paths. It must be the last route,

If it is at the top, angular always redirect to this path.

{ path: '\*\*', redirectTo: 'not-found'}

**[Lecture Link: Redirection Path Matching (Extra Info)](https://www.udemy.com/course/the-complete-guide-to-angular-2/learn/lecture/6656336" \l "overview)**

**12. Outsourcing the Route Configuration:**

We can create separate module for routing to make app.module.ts leaner.

1. Create a separate module first, and copy const type of Routes, which holds all the route path configurations.
2. In @NgModule, in imports array, we give our routes configurations to RouterModule, and exports array will have RouterModule to make it available for exporting for app.module.ts.

@NgModule({

    imports: [

        RouterModule.forRoot(appRoutes)

    ],

    exports: [RouterModule]

})

1. In app.module.ts we import the created appRouting.module

  imports: [BrowserModule, FormsModule, AppRoutingModule]

**12. Route Guards:**

**Definition:** Mechanism that allow us to control the navigation behavior by adding logic to routes. It executes a logic before a route is loaded or once we want to leave the route. They help in securing and managing access to certain routes in angular application based on specific conditions.

**canActivate:** This guard determines whether a route can be activated. It is commonly used for access control, ensuring that the user has the necessary permissions to navigate to a specific route.

**CanActivateChild:** Similar to CanActivate, but specifically designed for guarding child routes. It allows you to control whether child routes of a component can be activated.

{ path: 'servers', component: ServersComponent,

      canActivate: [AuthGuard],

      canActivateChild: [AuthGuard],

      children: [

        { path: ':id', component: ServerComponent },

        { path: ':id/edit', component: EditServerComponent },

      ]

    }

export class AuthGuard implements CanActivate {

  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot)

    : Observable<boolean> | Promise<boolean> | boolean {

    return this.authService.isAuthenticated().then((isAuthenticated: boolean) => {

      if (isAuthenticated) {

        return true;

      } else {

        this.router.navigate(['/']);

        return false;

      }

    });

  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot)

  : Observable<boolean> | Promise<boolean> | boolean {

    return this.canActivate(route, state);

  }

}

export class AuthService {

  loggedIn: boolean = false;

  isAuthenticated() {

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

      setTimeout(() => { resolve(this.loggedIn); }, 800);

    });

    return promise;

  }

  login() {

    this.loggedIn = true;

  }

  logout() {

    this.loggedIn = false;

  }

}

**CanDeactivate:**

Here we can control weather we can leave the route or not. For example we can stop the leaving from the page when changes are made but not saved.

1. To do this, First we need to create an interface “CanComponentDeactivate” with a method canDeactivate method will return Observable or Promise which will resolve to boolean or just a boolean.
2. Now create a service class, which actually our guard which implements CanDeactivate from angular router, which takes generic type and in our case we declare it as CanComponentDeactivate interface.
3. Here we need to provide implementation for canDeactivate method which has same return type as our CanComponentDeactivate interface.
4. We get four different arguments to process, and here we are using CanComponentDeactivate instance to call canDeactivate(). Below we can see the code.

**Guard code:**

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

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

import { Observable } from 'rxjs/Observable';

export interface CanComponentDeactivate {

canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;

}

@Injectable({providedIn: 'root'})

export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {

  canDeactivate(component: CanComponentDeactivate, currentRoute: ActivatedRouteSnapshot,

    currentState: RouterStateSnapshot, nextState: RouterStateSnapshot

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

    return component.canDeactivate();

  }

}

1. Now whatever the component we need to add guard for deactivation, we need to implement CanComponentDeactivate interface there, and implement canDeactivate() method. In this we will write logic, if we return true it will allow you to leave the route, if false returned it will not allow you to leave the route.

**Code in component:**

export class EditServerComponent implements OnInit, CanDeactivateGuard{

// All the code

canDeactivate(component: CanComponentDeactivate, currentRoute: ActivatedRouteSnapshot,

    currentState: RouterStateSnapshot, nextState: RouterStateSnapshot)

    : boolean | Observable<boolean> | Promise<boolean> {

    if(!this.allowEdit) {

      return true;

    }

    if((this.serverName !== this.server.name || this.serverStatus !== this.server.status) && !

      this.changesSaved) {

      return confirm('Are you sure you want to discard changes ?');

    } else {

      return true;

    }

  }

}

**Routes code:**

    { path: 'servers',

      component: ServersComponent, canActivateChild: [AuthGuard],

      children: [ { path: ':id/edit', component: EditServerComponent,

          canDeactivate: [CanDeactivateGuard] }

      ]

    }

**12. Resolver:**

Resolver is a service that is used to ensure that the data needed for a component is available before the component is instantiated. It's often used in the context of Angular routing to fetch data from a source (such as an API) before navigating to a route and rendering the associated component.

**Using Static Data:**

We can give the message in the router, and we use “data” to provide data. In that we give key value pair, key is for later use to get data in the component.

**Router code:**

{ path: 'not-found', component: ErrorPageComponent, data :{ message: 'Page not found!!!'}},

**Component code:**

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {

    this.errorMessage = this.route.snapshot.data['message'];

    this.route.snapshot.data.subscribe((data: Data) => {

      this.errorMessage = data['message'];

    });

  }

**Getting Dynamic Data Using Resolver Guard:**

1. We need to create resolver service which implements Resolve<T> (T is the type of data we are expecting in the component). This will make us implement resolve method which gives us two arguments route: ActivatedRouteSnapshot, state: RouterStateSnapshot.
2. In this method we should return type T.

@Injectable({providedIn: 'root'})

export class ServerResolver implements Resolve<Server> {

  constructor(private serversService: ServersService) {}

  resolve( route: ActivatedRouteSnapshot, state: RouterStateSnapshot )

        : Server | Observable<Server> | Promise<Server> {

    return this.serversService.getServer(+route.params['id']);

  }

}

1. Now we should declare created Resolver(ServerResolver) this in router, in resolve which takes JavaScript object.

 { path: ':id', component: ServerComponent, resolve: {server: ServerResolver} },

1. In component, ngInit method we subscribe to the data and fill our desired properties.

  ngOnInit() {

    this.route.data.subscribe((data: Data) => {

      this.server = data['server'];

    });

  }

**13. Hash Mode Routing:**

In Angular's routing configuration, the useHash option is a property that can be set to true or false. It is used to configure how the Angular application's URLs are constructed.

When useHash is set to true, Angular uses the hash fragment of the URL (the part of the URL after the # symbol) to manage its routes. This is known as "hash routing" or "hash mode." In hash routing, changes to the route do not result in a full page reload; instead, only the part of the URL after the # symbol changes, and Angular uses this to determine the route.

@NgModule({

imports: [RouterModule.forRoot(routes, { useHash: true })],

exports: [RouterModule]

})

export class AppRoutingModule { }

**Route configuration (pathMatch: 'full')**:   
The pathMatch property is used to control how the router should match URL segments to the configured route. When pathMatch is set to "full", it means that the router should only consider the URL path if it fully matches the specified path of the route.

const routes: Routes = [

{ path: 'home', component: HomeComponent, pathMatch: 'full' },

{ path: 'about', component: AboutComponent, pathMatch: 'full' },

];

If the URL is /home, the route with path: 'home' will match because the entire URL path matches the specified path.

* If the URL is /home/something or /about/something, the routes won't match. This is because pathMatch: 'full' requires the entire URL path to match, and the additional segment ('something' in this case) makes the paths not fully match.
* If pathMatch were not set or set to the default value ("prefix"), the routes would match as long as the specified path is a prefix of the URL. This means that /home/something would match the 'home' route, and /about/something would match the 'about' route.

**Path Note:**

{ path: 'recipes', component: RecipesComponent , children: [

        { path: '', component: RecipeStartComponent},

        { path: ':id', component: RecipeDetailComponent},

        { path: 'new', component: RecipeEditComponent}

    ]}

* In the above case if we try to access “localhost:4200/new”, It will give error and it actually tries to access :id component which is RecipeDetailComponent instead of 'new' path component RecipeEditComponent.
* Reason is angular will consider ‘new’ as ':id', to fix this we can just declare 'new' path before ':id'

1. **OBSERVABLES**

**Definition:** Observables are a technique for event handling, asynchronous programming, and handling multiple values emitted over time.

* It’s an object created using RxJS third party library.
* Observables uses “Observer Design Pattern”, where one object (the subject or publisher) changes its state, all its dependents (observers or subscribers) are notified and updated automatically.
* This allows for a loosely coupled system where the subject and observers can interact without knowing much about each other.
* Installation: *npm install --save rxjs@6* and *npm install --save rxjs-compat*

**1. interval function:**  
Interval is simple observable available in rxjs, This observable emits sequential numbers at a specified time interval. We must unsubscribe() to it before leaving the component in ngOnDestroy()

  private firstObsSubscription: Subscription;

  ngOnInit() {

    this.firstObsSubscription = interval(1000).subscribe(

      count => { console.log(count) }

    )

  }

  ngOnDestroy(): void {

    this.firstObsSubscription.unsubscribe();

  }

**2. Creating Custom Observable: (Emitting Data)**

In here we can emit 3 types of Observable information. (1. Data, 2. Error, 3. Complete)

    // Create a custom observable

    const customIntervalObservable = Observable.create(observer => {

      let count = 0;

      setInterval(() => {

        observer.next(count);

        count++;

      }, 1000);

    })

    // Subscribe to created observable

    this.firstObsSubscription = customIntervalObservable.subscribe(data => {

      console.log('Count is ' +data);

    })

**Creating Custom Observable: (Emitting Error)**

  ngOnInit() {

    // Create a custom observable which also throws error

    const customIntervalObservable = Observable.create(observer => {

      let count = 0;

      setInterval(() => {

        observer.next(count);

        if(count > 3) {

          observer.error(new Error('Count is more than 3'));

        }

        count++;

      }, 1000);

    })

* In the above code it throws an error when count is more than 3. In the browser console we can see Error thrown in red text. But we can handle this error by subscribing to it, just like data, we get error argument also.
* Throwing error cancels the observable, It will not run anymore.
* If emits error it will not go to complete()

    // Subscribe to created observable and error

    this.firstObsSubscription = customIntervalObservable.subscribe(

data => { console.log('Count is ' +data);},   
 error => { console.log(error); })

**Creating Custom Observable: (Emitting Complete)**

        if(count === 2) { observer.complete(); }

    // Subscribe to created observable and error

    this.firstObsSubscription = customIntervalObservable.subscribe(

data => { console.log('Count is ' +data); },   
 error => { console.log(error);},

() => { console.log('Completed');} );

**3. Operators:**

Operators in observables are functions provided by libraries like RxJS that allow you to manipulate and transform the data emitted by observables. We preform this by using *pipe(operator code)* and we can apply one more operators to transform the data.

1. **map operator:** Transforms each item emitted by an observable by applying a function to it.
2. **filter operator:** Emits only the items that satisfy a specified condition.
3. **merge operator:** Merges multiple observables into a single observable.

**Map example:**

1. this.firstObsSubscription = customIntervalObservable
2. .pipe(map((data: number) => {
3. return 'Round is ' + (data + 1);
4. }))
5. .subscribe((data) => {
6. console.log(data);
7. });

**Filter example:**

    this.firstObsSubscription = customIntervalObservable

      .pipe(filter((data: number) => {

         return !(data % 2 === 0);

       }))

**4. Subjects:**

In RxJS, a **Subject** is a special type of observable that acts as both an observer and an observable itself. This means that you can **subscribe** to a Subject to receive its emitted values, and you can also **push values** into a Subject to notify all its subscribers.

* Subject comes with a variety of operators provided by RxJS, allowing you to transform, filter, and manipulate the data stream easily.
* It is mechanism for managing data streams compared to the simpler event-based approach of Event Emitter.
* We should always unsubscribe in onDestroy method.

**Defining a subject:**

import { Subject } from "rxjs";

userActivateEmitter = new Subject<boolean>();

**Emitting values by subject:**

this.userService.userActivateEmitter.next(true);

**Subscribe to a subject:**

  this.userService.userActivateEmitter.subscribe((data: boolean) =>{

    this.userActivated = data;

  })

1. **Forms**

**Definition:** Forms are mechanism for collecting, validating, and managing user input within a web application. Angular provides a powerful and flexible Forms module that allows developers to create both template-driven forms and reactive forms.

Angular offers two approaches for forms:

1. **Template driven:** Angular infers the Form object from the DOM
2. **Reactive:** Form is created programmatically and synchronized with the DOM

**1. Template driven:**

In this approach, the form is primarily defined in the HTML template, and Angular uses directives and binding syntax to create and manage the form controls. The template-driven approach is more declarative, requiring less code in the component class compared to the reactive approach.

**i) Declaring, Submitting Form & Form Data:**

* **ngModel**: is used to bind the input fields to properties in the form. So, this field will be a part of JavaScript object representation of the form. We should also specify “name” in the field.

*<input ngModel name=”username” type=”text” class=”form-control”>*

* **(ngSubmit)**: directive is used to bind a method to the submit event of a form. When the form is submitted using button with type “submit” in the form, the associated method is executed.
* **#myForm="ngForm":** We declare **ngForm** on form tag, It creates a reference to the form and binds it to the ngForm directive.
* **NgForm object** represents the Angular form and provides methods and properties for interacting with the form's state and controls.

**Template code:**

<form (ngSubmit)="onSubmit(myForm)" #myForm ="ngForm">

<input type="text" id="username" class="form-control"

ngModel name="username">

<input type="email" id="email" class="form-control"

         ngModel name="email">

<button class="btn btn-primary" type="submit">Submit</button>

</form>

**TS code:**

  onSubmit(form: NgForm) {

    console.log(form.value);

  }

* **Access Form with @ViewChild():** Here we don’t use (ngSubmit)="onSubmit(myForm)" ,instead we create a with ViewChild, and we provide the form reference to it. By this method we can access the form data without the submit event as well. @ViewChild(‘form\_reference’)

**Template code:**

<form (ngSubmit)="onSubmit()" #myForm="ngForm">

// Same code as above

</form>

**TS code:**

  @ViewChild('myForm') signupForm;

onSubmit() {

    console.log(this.signupForm.value);

  }

**ii) Form Validations:**

Validators help you define and enforce validation rules for form controls. Validators can be applied to both template-driven and reactive forms, and they ensure that user input meets specific requirements.

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

* In above code “required”, “email” are validators. When we submit form without valid email id, in ngForm object we can see valid() function as false.
* We can use form data inside of the form. In below example disabled attribute will make button enabled once the form do not have any validations.  
  <button [disabled]="!myForm.valid" class="btn” type="submit">Submit</button>
* When email is not valid, angular adds style class to that element called “ng-invalid” like shown below.

A yellow and black text

Description automatically generated

* Once we enter valid email, angular changes style class to “ng-valid”.

A yellow box with black text

Description automatically generated

* We can change the style class in the CSS file to make it look like invalid data entered.

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

* **Adding validation message:** We can add validation for particular input, for that we have to give reference to input *#inputName=”ngModel”*, now using *inputName* values we can display error message with the help of ngIf.

<input type="email" id="email" class="form-control"

      ngModel name="email" required email #email="ngModel">

<span \*ngIf="!email.valid && email.touched"> Enter valid email address </span>

* **Pattern validation:** It’s a built-in validation feature that allows you to enforce a specific regular expression pattern on the value of an input field. This ensures that the user's input adheres to a particular format or structure.

<input type="number" id="amount" class="form-control"

        name="amount" ngModel required pattern="^[1-9]+[0-9]\*$"/>

**iii) Data Setting & Few other features:**

* **Set Default Values with ngModel Property Binding:** We giving default value to ngModel to begin with. We can provide it in template or TS file as well.

<select id="secret" class="form-control" [ngModel]="'pet'" name="secret">

  <option value="pet">Your first Pet?</option>

  <option value="teacher">Your first teacher?</option>

</select>

* **Using ngModel with Two-Way-Binding:** With the help of [(ngModel)], we achieve this and it will update the user inputs for each key strokes. In below example, questionAnwser is defined in TS file.

<textarea name="questionAnwser" class="form-control" rows="3"

[(ngModel)] ="questionAnwser">

  </textarea>

  <p> Your reply is : {{ questionAnwser }} </p>

* **ngModelGroup:** Directive that is used to group multiple form controls together under a common parent. We can get grouped validation status using this.

<div id="user-data" ngModelGroup="userData" #userData="ngModelGroup">

    <div class="form-group">

        <label for="username">Username</label>

        <input type="text class="form-control" ngModel name="username" required>

     </div>

<div class="form-group">

        <label for="email">Mail</label>

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

     </div>

</div>

<span class="help-block" \*ngIf="!userData.valid && userData.touched">

    Enter valid user data!!

</span>

* **Radio buttons:**

 <div class="radio" \*ngFor="let gender of genders">

    <label>

    <input type="radio" name="gender" ngModel [value]="gender" required>

        {{ gender }}

    </label>

</div>

* **Setting and Patching Form Values:**

Using *setValue()* We can set values of inputs pf whole form, one issue is it will override already entered data.

  suggestUserName() {

    const suggestedName = 'Superuser';

    this.signupForm.setValue({

      userData: {

        username: suggestedName,

        email: ''

      },

      secret: 'pet',

      questionAnwser: '',

      gender: 'Female'

    })

}

Using *patchValue()* We can set values of particular input in the form without effecting already entered data.

   this.signupForm.form.patchValue({

      userData: { username: suggestedName }

   });

* **Using Form data:**

  user = {

    username: '', email: '', secretQuestion: '',

    answer: '', gender: ''

  };

  onSubmit() {

    this.submitted = true;

    this.user.username = this.signupForm.value.userData.username;

    this.user.email = this.signupForm.value.userData.email;

    this.user.secretQuestion = this.signupForm.value.secret;

    this.user.answer = this.signupForm.value.questionAnwser;

    this.user.gender = this.signupForm.value.gender;

  }

* **Resetting Form:**

this.signupForm.reset()

Calling reset() method of form will reset the whole form.

**Find Validators shipped with angular here.**

**Template-driven:** <https://angular.io/api?type=directive> [Search by "validator"]

**Reactive:** <https://angular.io/api/forms/Validators>

**2. Reactive:**

Reactive approach uses Reactive Forms module, which provide a programming-centric approach to handling form creation, validation, and submission. Reactive forms are defined using TypeScript and are based on the FormGroup, FormControl, and FormArray classes provided by the *@angular/forms* module.

To use reactive approach, we need to import *ReactiveFormsModule* in AppModule.

1. **Declarations:**

**In TS Class:**

* **FormGroup (entire form):** Represents the entire form and is a container for individual form controls. Created using the FormGroup class. We can use this to group multiple FormControl (inputs) as well.
* **FormControl (individual field):** Represents an individual form control (e.g., an input field) and is responsible for handling its value and validation state.
* **FormArray:** FormArray is a class that represents an array of form controls, making it suitable for managing dynamic forms where the number of form controls is not known at design time.

As of Angular 8+, there's a new way of clearing all items in a FormArray. The clear() method automatically loops through all registered FormControls (or FormGroups) in the FormArray and removes them.

**(<FormArray>this.recipeForm.get('ingredients')).clear();**

**In Template:**

* **[formGroup]:** Represents the entire form in template, declared in <form> tag.
* **formGroupName:** Represents group of multiple inputs, declared at group div of multiple inputs.
* **formControlName:** Represents the entire form in template, declared at input tag.
* **(ngSubmit):** Executes the method provided when form is submitted.

**TS code:**

signuForm: FormGroup;

this.signuForm = new FormGroup({

  userData: new FormGroup({

    username: new FormControl(null, Validators.required),

    email: new FormControl(null, [Validators.required, Validators.email]),

  })

});

**Template code:**

<form [formGroup]="signuForm">  
 <div formGroupName="userData">

<input type="text" id="username" formControlName="username">  
 <input type="text" id="email" formControlName="email">  
 </div>  
</form>

1. **Showing Error Messages:**

**Error Message for particular input:**

<div class="form-group">

<div formGroupName="userData">  
 <input type="text" id="username formControlName="username"/>

  <span class="help-desk" \*ngIf="!signuForm.get('userData.username').valid &&

signuForm.get('userData.username').touched">

Please enter the name! </span>

</div>

</div>

**Error Message for entire form:**

<span \*ngIf="!signuForm.valid && signuForm.touched" class="help-desk">

Please valid data </span>

1. **Array of FormControl (formArrayName):**We can group multiple inputs together using this feature.

**TS code:**

this.signupForm = new FormGroup({

      'userData': new FormGroup({

        'username': new FormControl(null, Validators.required),

        'email': new FormControl(null, [Validators.required, Validators.email]),

      }),

      'hobbies': new FormArray([])

    });

  getControls() {

    return (this.signupForm.get('hobbies') as FormArray).controls;

  }

  onAddHobby() {

    const control = new FormControl(null, Validators.required);

   (<FormArray>this.signupForm.get('hobbies')).push(control);

  }

}

**Template code:**

<div formArrayName="hobbies">

    <h4>Your Hobbies</h4>

    <button class="btn"type="button" (click)="onAddHobby()">Add Hobby</button>

    <div class="form-group">

      <input type="text" class="form-control"

            \*ngFor="let hobbyControl of getControls(); let i = index"

[formControlName]="i">

    </div>

</div>

1. **Creating Custom Validators:**

Validator is a method which checks the supplied formControl object and if not satisfied with condition, it should return JavaScript object like this {'nameIsForbidden': true} or must return null if satisfied with the condition.

**Defining custom validator in FormControl:**

'username': new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),

**Defining custom validator:**

forbiddenUsernames = ['Chris', 'Anna'];

  forbiddenNames(control: FormControl) : {[s: string]: boolean}{

    if(this.forbiddenUsernames.indexOf(control.value) !== -1){

      return {'nameIsForbidden': true};

    }

    return null;

  }

**Defining error messages for this validator:**

We use errors['nameIsForbidden']"> to access the validator is fired or not for particular field.

<span class="help-desk" \*ngIf="!signupForm.get('userData.username').valid &&

signupForm.get('userData.username').touched">

<span \*ngIf="!signupForm.get('userData.username').errors['nameIsForbidden']">

      This username is forbidden!

</span>

  <span \*ngIf="!signupForm.get('userData.username').errors['required']">

      Username is required!

  </span>

</span>

1. **Creating Async Custom Validators:**

* We declare this validator in formControl in 3rd argument. Also, since its async, We will have to return a promise or observable here.
* While it waits checking the validation, our input updates with ng-pending style class. Like in below example for 2 seconds.

**Defining custom validator in FormControl:**

'email': new FormControl(null, [Validators.required, Validators.email], this.forbiddenEmails.bind(this)),

**Defining async custom validator:**

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

    const promise = new Promise<any>((resolve, reject) => {

      setTimeout(() => {

        if(control.value === 'test@test.com'){

          resolve({'emailIsForbidden': true});

        } else{

          resolve(null);

        }

      }, 2000);

    });

    return promise;

  }

1. **Status Changes & Value Changes:**

These are two useful observables, Status Changes will be fired each time status changes & Value changes will be fired when value changed.

    this.signupForm.valueChanges.subscribe(value => {

      console.log(value);

    })

    this.signupForm.statusChanges.subscribe(value => {

      console.log(value);

    })

1. **Setting & Patch Values:**

**Setting:**

    this.signupForm.setValue({

      'userData': {

        'username': 'Max',

        'email': 'max@max.com'

      },

      'gender': 'male',

      'hobbies': []

    });

**Patching:**

    this.signupForm.patchValue({

      'userData': {

        'username': 'Anna',

        'email': 'anna@anna.com'

      }

    });

1. **Reset Form:**

  onSubmit() {

    console.log(this.signupForm);

    this.signupForm.reset();

  }

1. **PIPES**

**Definition:** Pipe are used to transform the data in the template, and we don’t change the data directly but we change the output displayed in the template.

* Pipes can be used within Angular templates using the pipe symbol | followed by the pipe name and optional parameters are followed by : each parameter. Pipes can be chained together to perform multiple transformations.
* Pipes can chained but order of the pipe is important, angular parses pipes left to right.

**Syntax:** Data | <pipe\_name>:<optional\_parameter>:<optional\_parameter >  
**Examples:** {{ user.name | uppercase }}   
 {{ birthday | date:'longDate' }}  
 {{ birthday | date:'longDate' | uppercase }}

**Built-in Pipes:** Angular provides build-in pipes to use.

<https://angular.io/guide/pipes> & <https://angular.io/api?query=pipe>

**Create Pipe:**

@Pipe({ name: 'shorten' })

export class ShortenPipe implements PipeTransform {

  transform(value: any, limit: number) {

    if(value.length > limit) {

      return value.substring(0, limit) + ' ...';

    }

    return value;

  }

}

**Create Filter Pipe:**

@Pipe({ name: 'filter' })

export class FilterPipe implements PipeTransform {

  transform(value: any, filterString: string, propName: string): any {

    if(value.length === 0 || filterString == '') {

      return value;

    }

    const resultArray = [];

    for(const item of value){

      if(item[propane].toUpperCase() === filterString.toUpperCase()){

        resultArray.push(item);

      }

    }

    return resultArray;

  }

}

<input type="text" class="form-control" [(ngModel)]="filteredStatus" placeholder="Filter">

<app-recipe-item \*ngFor="let recipeEl of recipes | filter:filteredStatus:'name';

[recipe] = recipeEl>

</app-recipe-item>

**Pure & Impure Pipes:**

* In the above created filter, when it’s the filter input have some text, If we try to add one more item to list, It will not show immediately but it will show when the input is cleared.
* Its angular default behavior, reason is angular will not rerun the pipe each time the data in the list changed. To make it check each time might cause performance issues.
* Now, to make it check each time data changes, we need to change some syntax like below.

@Pipe({

  name: 'filter',

  pure: false

})

**Async Pipe:** Async Pipe in Angular is a built-in pipe that allows you to work with asynchronous data directly within your templates. It subscribes to an Observable or a Promise and automatically handles the subscription, as well as the handling of emitted values or resolved promises.

**TS code:**

  appStatus = new Promise((resolve, reject) => {

    setTimeout(() => {

      resolve('stable');

    }, 2000);

  });

**Template code:**

<h3>App Status is {{ appStatus | async }}</h3>

1. **HTTP CLIENT**

**Usage:** HTTP requests are used to interact with backend servers and external APIs to fetch, send, and manipulate data. Import it from import { HttpClientModule } from '@angular/common/http';

1. **HTTP Headers:** HTTP headers are additional pieces of information sent along with an HTTP request or response. They provide metadata about the request or response, such as the content type, content length, caching directives, authentication tokens, and more. HTTP headers consist of a name-value pair separated by a colon (:). Examples of common HTTP headers include Content-Type, Accept, Authorization, User-Agent, and Cache-Control.
2. **URL (Uniform Resource Locator):** A URL is a standardized address used to locate resources on the internet. It consists of several components:
   1. **Scheme:** Specifies the protocol used to access the resource (e.g., http://, https://, ftp://).
   2. **Host:** Specifies the domain name or IP address of the server where the resource is located.
   3. **Port:** Optionally specifies the port number on the server to which the client should connect.
   4. **Path:** Specifies the location of the resource on the server's file system or within the application.
   5. **Query Parameters:** Optionally provides additional parameters to the resource, typically used for filtering or customizing the request.
   6. **Fragment Identifier:** Optionally specifies a specific section within the resource

A diagram of a website

Description automatically generated

constructor(private http: HttpClient) { }

getDepartments(): Department[] {

    this.http.get('http://localhost:8080/api/v1/departments').subscribe(

      (respone) => {

        console.log(respone);

      }

    );

    return this.departments;

  }

* The http request will get executed, only when we subscribe to it.
* Best practice is to create a service which calls REST API end point and returns observable. That observable is subscribed in the component.ts
* When angular gets response from http method, it only gives the **body of the response** by default, also it convert to JavaScript object from JSON.
* We can change the response by passing argument like { observe: ‘response’} in the method. This gives whole response from backend server.

**Error Handling:**In the second argument it gives error, if the service throws any error. As shown in the example below.

this.deptService.deleteDepartment(this.selectedDeptUuid).subscribe((data: any) => {

alert(‘Department has been deleted’)

}, (error: any) => {

   console.log(error);

// Do some error handling

})

**Using Subject for Error Handling:**

* Create a subject in the service class and when ever any service throws error in the second argument when subscribed to a http method, throw the error using next().
* Subscribe to the subject in the component class and show the error or do the needful.

**Catch Error (catchError, throwError):**

Catch the error using pipe at http method. Inside pipe method, catch the error using catchError and must throw appropriate error after checking the error using throwError as shown in the example below.

  deleteDepartment(deptUuid: string){

    return this.http.delete('http://localhost:8080/api/v1/departments' +deptUuid)

    .pipe(catchError((error) => {

      if(error.code === 404) {

        return throwError(() => new Error('Department not found!'));

      }

    }));

  }

**Error Panel:**

    <div class="alert alert-danger" \*ngIf="false">

      <h2> An Error Occurred!</h2>

      <p> Department not found!</p>

      <button class="btn btn-danger">Okay</button>

    </div>

**Setting headers to Http request:**

When making HTTP requests, we often need to include additional information in the request headers. HTTP headers provide metadata about the HTTP request or response, such as the content type, authentication tokens, caching directives, and more. In Angular, you can set headers to HTTP requests using the **HttpHeaders** class. We can do this two ways.

1. **Create HttpHeaders Object:** First, create an instance of the HttpHeaders class. HttpHeaders is immutable, so you need to use the set() method to append or set headers.
2. **Set Headers:** Use the set() method to set individual headers in the HttpHeaders object. You can set multiple headers by chaining multiple set() calls.
3. **Pass HttpHeaders to Request:** When making an HTTP request using Angular's HttpClient service, you can pass the HttpHeaders object as an option to the request.

**Example 1:**

  getDepartment(uuid: string): Observable<Department>{

    return <Observable<Department>> this.http.get('http://localhost:8080/api/

/departments/' + uuid,

      { headers: new HttpHeaders({'Accept': 'application/json'}) }

    );

  }

**Example 2:**

  getDepartment(uuid: string): Observable<Department>{

    const headers = new HttpHeaders()

      .set('Auth', 'token\_id').set('Content-Type', 'application/json');

    return <Observable<Department>>

this.http.get('http://localhost:8080/api/departments/' + uuid,

    { headers }

    );

  }

**Adding query parameters to Http request:**  
Very similar to header, check the examples

getDepartment(uuid: string): Observable<Department>{

return <Observable<Department>> this.http.get('http://localhost:8080/api/ departments/' + uuid,

    { headers: new HttpHeaders({'Accept': 'application/json'}),

      params: new HttpParams().set('print', 'pretty') }

    );

  }

  getDepartment(uuid: string): Observable<Department>{

    const headers = new HttpHeaders()

      .set('Authorization', 'your\_token')

      .set('Content-Type', 'application/json');

    const params = new HttpParams()

      .set('view', 'pretty');

    return <Observable<Department>> this.http.get('http://localhost:8080/api/

/departments/' + uuid,

    { headers, params }

    );

  }

**Observing Different Types of Responses:**

We may need to observe different types of responses depending on the scenario. Angular's HttpClient provides methods to handle various response types such as JSON data, full HTTP responses, and more.

* **Syntax:** options: { observe?: 'body' | 'events' | 'response' }
* We usually get body as response by default, we can change it by using these three options we have.
* **'response'** is from backend service including with status code, response header.
* Use the **observe: 'events'** option to observe events during the request progress. Subscribe to the observable returned by the request to receive different types of events such as progress, response headers, and more.

**In post method:**

return <Observable<Department>>

this.http.post('http://localhost:8080/api/v1/departments/', departmentObject

      { headers, params,

        observe: reponse

      })

**Observing Full HTTP Responses:**

Use the request() method to get the full HTTP response, which includes headers, status, and body.

Subscribe to the observable returned by the request() method to receive the full HTTP response.

this.http.request('GET', 'https://api.test.com/data', { observe: 'response' })

  .subscribe((response: HttpResponse<any>) => {

    console.log('Received full response:', response);

    console.log('Response body:', response.body);

    console.log('Response headers:', response.headers);

    console.log('Status code:', response.status);

  });

**observe: 'events' example:**

  this.http.get('https://api.example.com/data', { observe: 'events' })

    .subscribe(event => {

      if (event.type === HttpEventType.DownloadProgress) {

        console.log('Download progress event:', event);

      } else if (event.type === HttpEventType.ResponseHeader) {

        console.log('Response header event:', event);

      } else if (event.type === HttpEventType.Response) {

        console.log('Response event:', event);

        console.log('Response body:', event.body);

      }

    });

  deleteDepartment(deptUuid: string){

    return this.http.delete('http://loc:8080/api/v1/departments/' +deptUuid, {

    observe: 'events' })

    .pipe(

      tap(event => {

        if(event.type === HttpEventType.Response){

          console.log(event.body);

        }

    }));

  }

**Changing Response Type:**

We can change response type like JSON, text, blob, etc. We can specify the response type you expect when making HTTP requests. This helps Angular handle the response appropriately and can simplify your code by avoiding manual type conversions.

  this.http.get('https://api.test.com/text', { responseType: 'text' })

.subscribe(text => {

    console.log('Received text:', text);

  });

**Interceptors:**

Interceptors are services that can be registered globally or locally within an Angular application to intercept HTTP requests and responses. Interceptors implement the ***HttpInterceptor*** interface provided by Angular's @angular/common/http module. They can perform operations like adding headers, logging requests, modifying responses, and handling errors.

**Creating Interceptor service class:**

import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from "@angular/common/http";

import { Observable } from "rxjs";

export class AuthInterceptorService implements HttpInterceptor {

    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

        const newRequest = req.clone(

            { headers: req.headers.append('Auth', 'xyz') }

        )

        return next.handle(newRequest);

    }

}

**Declare in app.module.ts**

  providers: [{

    provide: HTTP\_INTERCEPTORS,

    useClass: AuthInterceptorService,

    multi: true

  }]

**Response Interceptors:**

Response interceptors in Angular intercept and modify HTTP responses globally across your application. They provide a way to preprocess responses, apply error handling, or transform data before it reaches the application components.

export class AuthInterceptorService implements HttpInterceptor {

    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

        const newRequest = req.clone(

            { headers: req.headers.append('Auth', 'xyz') }

        )

        return next.handle(newRequest).pipe(

            tap(event => {

                console.log(event);

                if(event.type === HttpEventType.Response) {

                    console.log('Response received');

                    console.log(event.body);

                }

            })

        );

    }

}

**Multiple Interceptors:**

* Adding multiple interceptors is simple, create any other interceptor just like above and add it in the app.module.ts
* But the order of the inceptors entered in array is important because that’s how they are going to execute.

**Resources**

<https://angular.io/guide/understanding-communicating-with-http>

1. **AUTHENTICATION & ROUTE PROTECTION**

!user ? false : true same as !!user

Use CatchError when subscribing to service. Maintain a common method and pass in the pipe().

Pipe(take(1))

ExhaustMap

BehaviorSubject

Localstorange store token

Sign in

Sign out

Auto sign in

Auto sign out