**Angular Tutorial – Part 01**

1. **Evolution of Angular Framework**

Angular is a popular open-source 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.

1. **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.

1. **Angular (2016 - Present)**:

After the release of Angular 2, subsequent versions adopted the name "Angular" without a version number. Angular continued to evolve with regular updates. Notable versions include Angular 4, Angular 5, and so on.

1. **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.

1. **Angular Ivy (2019)**:

Angular Ivy is a significant update to the Angular framework. It is a new rendering engine that aims to improve the performance and bundle size of Angular applications. It also simplifies the development of dynamic components and enhances tree-shakability.

1. **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.

1. **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.

1. **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.

1. **Future Development**:

Angular continues to evolve with a focus on improving performance, developer experience, and integration with modern web technologies.

1. **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 operating system (Windows, macOS, or 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.

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 several types of data binding in Angular:

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 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**

Directives are a powerful and fundamental feature in Angular. They allow you to extend the HTML vocabulary, providing instructions to manipulate the Document Object Model (DOM) and add behavior to your templates. Angular comes with built-in directives and allows you to create custom directives. Here's everything you need to know about directives in Angular:

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 two types:

1. **Structural Directives:** These change the structure of the DOM by adding or removing elements. Examples include \*ngIf, \*ngFor, and \*ngSwitch.
2. **Attribute Directives:** These change the appearance or behavior of an element. Examples include ngStyle, ngClass, and ngModel.
3. **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, you must 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:  
Decorator:** @Input:   
@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:**

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.

**@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.

**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 is a decorator that allows a component to access a child component, directive, or a DOM element within its template. It is part of the Angular Core library (@angular/core). ViewChild is commonly used when a parent component needs to interact with or retrieve information from its child components or elements.

**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.