## **Angular Introduction**

Angular is a popular open-source web application framework maintained by Google and a community of developers. It's used for building dynamic, single-page web applications (SPAs) and provides a comprehensive solution for front-end development.

Following are some reasons why you might consider using Angular:

- Modularity: Angular offers a modular approach to building applications. It allows you to break your application into smaller, reusable components, making it easier to manage and maintain your codebase.
- Two-way data binding: Angular's two-way data binding feature automatically synchronizes the data between the model and the view. This means that any changes made to the model are immediately reflected in the view, and vice versa, without needing to write explicit code to update the DOM.
- Dependency injection: Angular has a built-in dependency injection system that makes it easy to manage dependencies between different parts of your application. This promotes a modular and testable codebase by allowing you to inject dependencies into your components rather than hardcoding them.
- Directives: Angular provides a rich set of built-in directives that allow you to extend HTML with new attributes and tags. Directives like ngFor, ngIf, and ngSwitch are commonly used to add dynamic behavior to your templates.
- Services: Angular allows you to encapsulate reusable logic into services, which can then be injected into your components. This promotes code reusability and helps to keep your components lean and focused on presentation logic.
- Cross-platform: Angular can be used to build not only web applications but also mobile and desktop applications using frameworks like Ionic and Electron, respectively.
- Strong community support: Angular has a large and active community of developers who contribute to its development, provide support, and create a wide range of third-party libraries and tools to enhance the framework.

Angular is a powerful framework that provides a structured and efficient way to build modern web applications. If you're looking for a comprehensive solution for front-end development with features like data binding, modularity, and dependency injection out of the box, Angular could be a great choice for your project.

## **Angular Versioning**

Angular follows a versioning scheme that consists of three parts: major version, minor version, and patch version. The version numbers follow the format major.minor.patch.

Here's what each part represents:

- Major version: This number changes when there are significant changes or breaking changes in the framework. Major updates often introduce new features, major improvements, or changes that are not backward compatible with previous versions.

- Minor version: This number changes when new features are added in a backward-compatible manner or when significant improvements are made to the framework without breaking existing functionality.
- Patch version: This number changes for bug fixes, patches, or minor updates that do not introduce new features or breaking changes. Patch updates typically address issues found in the current version.

For example, if the current version of Angular is 12.3.4:
- The major version is 12.
- The minor version is 3.
- The patch version is 4.

Developers can keep track of the latest Angular releases and their respective changes through Angular's official documentation and release notes. It's essential to pay attention to version updates to ensure compatibility with the latest features and bug fixes while maintaining backward compatibility with existing codebases.

## Components in Angular
In Angular, components are the building blocks of a web application. They are reusable, self-contained pieces of code that encapsulate the functionality, structure, and styling of a part of the user interface.

Each component consists of three main parts:
- Template: The HTML markup that defines the component's view. This is where you define the structure of the component and bind data to it using Angular's templating syntax.
- Class: The TypeScript code that defines the component's behavior. This class contains properties and methods that manipulate data, handle user input, and interact with other parts of the application.
- Metadata: Annotations that provide additional information about the component. This includes the @Component decorator, which specifies the component's selector, template, styles, and other configuration options.

Here's a basic example of a component in Angular:

**Typescript (.ts)**
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.css']
})
export class MyComponent {
  // Component logic goes here
  message: string = 'Hello, Angular!';
}
```

In this example:

- @Component is a decorator that marks the class as an Angular component and provides metadata about it.
- selector specifies the custom HTML tag that represents this component in the template of other components.
- templateUrl points to the external HTML file that contains the component's template.
- styleUrls points to external CSS files that contain styles specific to this component.
- Inside the class, MyComponent, you can define properties and methods that define the component's behavior. In this case, we have a message property initialized with a string.

Following is the corresponding template (my-component.component.html):

**html**

```
<div>
  <h1>{{ message }}</h1>
</div>
```

In this template, we use interpolation ({{ }}) to bind the value of the message property from the component class to the view.

Once defined, you can use the app-my-component tag in any other component's template to include MyComponent in your application.

Components in Angular promote reusability, maintainability, and separation of concerns, making it easier to develop and maintain complex web applications.

## <u>**Data Binding in Angular**</u>

Data binding in Angular is a powerful feature that establishes a connection between the component's data (the model) and the HTML elements in the component's template (the view). It allows you to synchronize the state of the application's data with the user interface, enabling dynamic and interactive web applications.

### One-Way Data Binding

One-way data binding in Angular refers to the flow of data from the component class to the template (view) or from the template to the component class, but not both simultaneously.

There are two main types of one-way data binding in Angular: interpolation and property binding.

- Interpolation:
    - Interpolation is a one-way data binding method that allows you to display dynamic values from the component class in the HTML template. It is achieved by using double curly braces {{ }}.

- For example:
  **html**
  <p>{{ message }}</p>
  Here, message is a property of the component class, and its value will be dynamically rendered in the paragraph element.

- Property Binding:
  - Property binding allows you to set an HTML element's property to the value of a property in the component class. It's used to dynamically change the properties of HTML elements. Property binding is denoted by square brackets [].
  - For example:
    **html**
    Copy code
    <img [src]="imageUrl">
    Here, imageUrl is a property of the component class, and its value is dynamically bound to the src attribute of the <img> element.

With one-way data binding, changes made to the component class will reflect in the template, but changes made in the template will not affect the component class automatically. This unidirectional flow of data helps to maintain a clear and predictable data flow in your Angular application.

**Two Way Data Binding**
Two-way data binding in Angular allows for synchronization of data between the component class (model) and the HTML template (view) in both directions. This means that changes in the model update the view, and changes in the view update the model automatically.

Angular provides a convenient way to implement two-way data binding using the ngModel directive along with the banana-in-a-box syntax [(ngModel)].

Here's how two-way data binding works:

- ngModel Directive: The ngModel directive is used to establish a two-way data binding connection between an input element in the template and a property in the component class.
- Banana-in-a-Box Syntax: The banana-in-a-box syntax [(ngModel)] is a shorthand for combining property binding ([ngModel]) and event binding ((ngModelChange)). It enables the bidirectional flow of data between the input element and the component class property.

Example of how to use two-way data binding in Angular:
html
<input [(ngModel)]="name">

In this example:

- name is a property of the component class.

- Changes made to the input value in the template will automatically update the name property in the component class.
- Changes made to the name property in the component class will automatically update the input value in the template.

To use ngModel, you need to import the FormsModule in your Angular module. This module provides the necessary directives and services for forms handling, including two-way data binding with ngModel.

**typescript**

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

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

@NgModule({
  imports: [BrowserModule, FormsModule], // Include FormsModule in imports
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

With two-way data binding, you can easily create interactive forms and components in Angular, where changes in the user interface are immediately reflected in the underlying data model, and vice versa.

## Programs

**Angular program to accept login name and password from user. Alert the welcome message if the login name is Admin and Password is admin123, else alert the Wrong Credentials message.**

Following are the steps to achieve the given task

- Create a new component called LoginComponent.
- Define two properties (loginName and password) in the component class to store the user input.
- Create a method called login() to validate the credentials entered by the user.
- Create a separate HTML file (login.component.html) to define the template.
- Reference the HTML file in the component using the templateUrl property.

Here's the code:

**login.component.ts:**

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html'
})
export class LoginComponent {
  loginName: string = '';
  password: string = '';

  login() {
    if (this.loginName === 'Admin' && this.password === 'admin123') {
      alert('Welcome, Admin!');
    } else {
      alert('Wrong Credentials. Please try again.');
    }
  }
}
```

**login.component.html:**

```
<div>
  <label>Login Name:</label>
  <input type="text" [(ngModel)]="loginName">
</div>
<div>
  <label>Password:</label>
  <input type="password" [(ngModel)]="password">
</div>
<button (click)="login()">Login</button>
```

In this code:

- The LoginComponent class defines the component logic.
- The templateUrl property in the @Component decorator specifies the path to the HTML template file.
- The HTML file (login.component.html) contains the login form elements.
- Angular's ngModel directive is used for two-way data binding to bind the input fields to the component properties.
- When the login button is clicked, the login() method is called to validate the credentials, and an appropriate alert message is displayed.

**Consider the student object with attributes PRN, StudName, StudClass, Gender and Score. Write an Angular program to display details of 10 students in table form on a web page.**

Following are the steps to achieve the given task

- Define a Student class to represent the structure of a student.
- Create a component called StudentListComponent to display the list of students.
- In the component class, create an array of Student objects with details for 10 students.

- Use Angular's ngFor directive to loop through the array of students and display their details in a table.

First, let's define the Student class:

**student.model.ts:**

```
export class Student {
  constructor(
    public PRN: string,
    public StudName: string,
    public StudClass: string,
    public Gender: string,
    public Score: number
  ) {}
}
```

Next, let's create the StudentListComponent component:

student-list.component.ts:

```
import { Component } from '@angular/core';
import { Student } from './student.model';

@Component({
  selector: 'app-student-list',
  templateUrl: './student-list.component.html'
})
export class StudentListComponent {
  students: Student[] = [
    new Student('001', 'John Doe', '10th', 'Male', 85),
    new Student('002', 'Jane Doe', '12th', 'Female', 90),
    // Add details for 8 more students
  ];
}
```

Now, let's create the template file for the component to display the student details in a table:

student-list.component.html:

```
<table>
  <thead>
    <tr>
      <th>PRN</th>
      <th>Name</th>
      <th>Class</th>
      <th>Gender</th>
      <th>Score</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let student of students">
      <td>{{ student.PRN }}</td>
      <td>{{ student.StudName }}</td>
      <td>{{ student.StudClass }}</td>
      <td>{{ student.Gender }}</td>
      <td>{{ student.Score }}</td>
    </tr>
  </tbody>
</table>
```

In this template:

- We use an HTML table to display the student details.
- The table header contains column headers for PRN, Name, Class, Gender, and Score.
- We use Angular's ngFor directive to loop through the students array and display each student's details in a row of the table.

## Angular Directives

In Angular, directives are instructions in the DOM (Document Object Model) that tell Angular's HTML compiler (part of the Angular framework) to do something to a DOM element.

Directives in Angular can be categorized into three types:

- Component Directives: Components are the most common type of directive in Angular. They are reusable, self-contained units that encapsulate the presentation and behavior of a part of the UI. Components are typically composed of a TypeScript class (the component class), an HTML template, and optional CSS styles. Examples of component directives include @Component, @Directive, @Pipe, and @Injectable.

- Structural Directives: Structural directives alter the DOM layout by adding, removing, or manipulating elements. They are prefixed with an asterisk (*) and change the structure of the DOM by adding or removing elements based on a condition. Examples of structural directives in Angular include *ngIf, *ngFor, and *ngSwitch.

- Attribute Directives: Attribute directives alter the appearance or behavior of an existing DOM element. They are typically used to change the appearance or behavior of an element based on specific conditions or events. Examples of attribute directives in Angular include ngClass, ngStyle, and ngModel.

Following is a brief overview of how each type of directive works:

- Component Directives: Used to define custom components, which encapsulate behavior and presentation logic.
- Structural Directives: Used to add or remove elements from the DOM based on conditions. They are prefixed with an asterisk (*) in the HTML template.

- Attribute Directives: Used to modify the appearance or behavior of an element. They are applied to HTML elements using HTML attributes.

Directives play a crucial role in Angular development, allowing developers to extend HTML with custom behavior and functionality, and enabling the creation of dynamic and interactive web applications.

**Examples based on Directives**

**Example of ngIf Directive**

Following example demonstrates how you can use the ngIf directive in Angular to conditionally show or hide elements in the HTML template based on a condition:

Suppose you have a component that displays a message based on whether a user is logged in or not.

app.component.ts:

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

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

app.component.html:

```
<div *ngIf="isLoggedIn; else loggedOutTemplate">
  <h1>Welcome, User!</h1>
  <button (click)="isLoggedIn = false">Logout</button>
</div>

<ng-template #loggedOutTemplate>
  <h1>Please log in to continue.</h1>
  <button (click)="isLoggedIn = true">Login</button>
</ng-template>
```

In this example:

- The isLoggedIn property is a boolean variable in the component class, which determines whether the user is logged in (true) or not (false).
- In the HTML template, we use the *ngIf directive to conditionally render different elements based on the value of isLoggedIn.
- If isLoggedIn is true, the "Welcome, User!" message and a logout button will be displayed.

- If isLoggedIn is false, the "Please log in to continue." message and a login button will be displayed.

We use the else keyword along with ngIf to define an alternative template (loggedOutTemplate) that will be rendered when the condition is not met.

**Example of ngStyle Directive**
The ngStyle directive in Angular allows you to conditionally apply inline CSS styles to HTML elements based on expressions in your component class.

Suppose you have a component that displays a message with different styles based on whether it's a success message or an error message.

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

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

app.component.html:
```
<div [ngStyle]="{'color': isSuccess ? 'green' : 'red', 'font-weight': isSuccess ? 'bold' : 'normal'}">
  {{ isSuccess ? 'Success!' : 'Error!' }}
</div>
```

In this example:
- The isSuccess property is a boolean variable in the component class, which determines whether the message is a success message (true) or an error message (false).
- In the HTML template, we use the ngStyle directive to conditionally apply inline CSS styles to the <div> element based on the value of isSuccess.
- If isSuccess is true, the text color will be green and the font weight will be bold.
- If isSuccess is false, the text color will be red and the font weight will be normal.

This example demonstrates how ngStyle can be used to dynamically apply styles to HTML elements in an Angular application based on conditions specified in the component class.

## Services in Angular
In Angular, services are a fundamental part of the architecture that allow you to organize and share code across your application. Services are a way to encapsulate reusable functionality

and data that can be shared across multiple components. They promote a modular and maintainable codebase by allowing you to centralize common functionality in one place.

Following is an overview of services in Angular:

- Purpose: Services are used to encapsulate and share common functionality and data throughout your application. They are typically used for tasks such as data fetching, business logic, authentication, logging, and more.
- Dependency Injection (DI): Angular's dependency injection system is used to provide services to components that depend on them. This means that services are injected into the components that need them, rather than being instantiated directly within the component. This promotes modularity, reusability, and testability.
- Singleton: By default, Angular services are singleton objects. This means that there is only one instance of a service created and shared across the entire application. Any component that injects the service will receive the same instance.
- Creating Services: You can create a service in Angular using the @Injectable() decorator provided by Angular's core library. This decorator marks a class as a service and allows it to be injected into other Angular components.
- Injecting Services: To use a service in a component, you need to inject it into the component's constructor using Angular's dependency injection system. Angular will automatically resolve the dependencies and provide the required service instance.
- Providing Services: You can provide a service at different levels of the Angular application hierarchy: at the root level (available throughout the application), at the component level (available only within the component and its children), or at the module level (available within the module and its components).

Following example demonstrates how you can create and use a simple service in Angular:

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

@Injectable({
  providedIn: 'root' // Provides the service at the root level
})
export class DataService {
  getData(): string {
    return 'Hello from DataService!';
  }
}
```

You can then inject this service into any component that needs it:

```
typescript
import { Component } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-example',
  template: '<p>{{ message }}</p>'
})
export class ExampleComponent {
  message: string;

  constructor(private dataService: DataService) {
    this.message = this.dataService.getData();
  }
}
```

In this example, the DataService is a simple service that provides a method getData(), which returns a string. The ExampleComponent injects this service and uses it to get data to display in its template.