Angular Basic Interview Questions and Answers

**Beginner-Level Angular Interview Questions**

1. **What is Angular?**
   * **Answer**: Angular is a TypeScript-based open-source framework developed by Google. It is primarily used to build single-page web applications (SPAs) with a rich and interactive user interface.
2. **What are the key features of Angular?**
   * **Answer**:
     + **Data Binding**: Synchronization between the model and the view.
     + **Dependency Injection (DI)**: Used to inject dependencies into components and services.
     + **Directives**: Special markers that tell the DOM to do something.
     + **Components**: Building blocks of Angular applications.
     + **Routing**: Built-in navigation and routing system for SPAs.
     + **Forms**: Support for template-driven and reactive forms.
3. **What is a Component in Angular?**
   * **Answer**: A component is a building block of an Angular application. It contains a class (which defines data and logic), a template (HTML), and styles (CSS). The component is declared using @Component decorator and is responsible for the view and behavior of a particular part of the UI.
4. **What is Angular CLI, and how do you use it?**
   * **Answer**: Angular CLI (Command Line Interface) is a tool to create, manage, and build Angular applications. It simplifies tasks like generating components, services, and modules. Example commands:

ng new my-app # Create a new project

ng serve # Start the development server

ng generate component my-component # Generate a new component

**What is component and module in Angular**

In Angular, **components** and **modules** are fundamental building blocks that help organize and structure an application. Let's explore each in detail:

**What is a Component?**

A **component** in Angular is a fundamental UI building block that controls a portion of the screen or the user interface. It encapsulates the HTML template, styles, and behavior (logic) related to a specific view. Components are reusable and can interact with each other, enabling modular development.

**Key Features of Components:**

1. **Template**: Defines the HTML layout that the component will render. This can include Angular directives and bindings.
2. **Styles**: The CSS styles that apply to the component's template.
3. **Class**: Contains the logic and data for the component. It is defined in TypeScript and can include properties, methods, and lifecycle hooks.

**Example of a Component:**

typescript

Copy code

// app.component.ts

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

@Component({

selector: 'app-root', // HTML tag to use the component

template: `

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

<button (click)="changeTitle()">Change Title</button>

`,

styles: [`h1 { color: blue; }`]

})

export class AppComponent {

title: string = 'Hello, Angular Components!';

changeTitle() {

this.title = 'Title Changed!';

}

}

In this example:

* The @Component decorator marks the class as a component and provides metadata, including the selector, template, and styles.
* The component class has a property title and a method changeTitle() that modifies the title when the button is clicked.

**What is a Module?**

A **module** in Angular is a cohesive block of code that groups together related components, directives, pipes, and services. Modules help organize an application into cohesive blocks of functionality, making it easier to manage, maintain, and scale.

**Key Features of Modules:**

1. **NgModule Decorator**: The @NgModule decorator defines a module. It contains metadata about the module, including declarations, imports, exports, and providers.
2. **Declarations**: Lists the components, directives, and pipes that belong to the module.
3. **Imports**: Lists other modules that are required by this module.
4. **Exports**: Lists the components, directives, and pipes that can be used in the templates of components in other modules.
5. **Providers**: Specifies the services that the module contributes to the global collection of services.

**Example of a Module:**

typescript

Copy code

// app.module.ts

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

import { BrowserModule } from '@angular/platform-browser';

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

@NgModule({

declarations: [

AppComponent // Declare the AppComponent

],

imports: [

BrowserModule // Import BrowserModule to enable browser functionalities

],

providers: [],

bootstrap: [AppComponent] // Bootstrap the root component

})

export class AppModule {}

In this example:

* The @NgModule decorator defines the AppModule with its metadata.
* declarations includes AppComponent, indicating that this module contains this component.
* imports includes BrowserModule, which is necessary for running the application in a web browser.
* bootstrap specifies the root component to bootstrap when the application starts.

**Summary**

* **Component**:
  + A building block of the UI that contains a template, styles, and logic.
  + Encapsulates a specific part of the user interface and can be reused.
* **Module**:
  + A cohesive block of code that groups related components, directives, pipes, and services.
  + Defines the structure of the application and helps organize it into manageable parts.

Together, components and modules form the core architecture of Angular applications, enabling developers to build scalable and maintainable software.

**what are Directives in Angular explain each type with code example**

In Angular, **directives** are special markers in the DOM that tell Angular to attach specific behavior to elements or even modify their appearance. Directives are a fundamental part of Angular's architecture and allow developers to create reusable and dynamic components.

There are three main types of directives in Angular:

1. **Components Directives**: Technically, components are directives with templates. They are the most common type of directive and encapsulate a piece of functionality along with its associated HTML and styles.
2. **Structural Directives**: These directives change the structure of the DOM by adding or removing elements. They typically have an asterisk (\*) prefix in their usage.
3. **Attribute Directives**: These directives change the appearance or behavior of an existing element without changing its structure. They can manipulate the properties of elements.

Let’s explore each type with examples.

**1. Components**

Components are the most common type of directive in Angular. They encapsulate HTML, CSS, and behavior.

**Example:**

typescript

Copy code

// app.component.ts

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

@Component({

selector: 'app-root',

template: `<h1>{{ title }}</h1>`,

styles: [`h1 { color: blue; }`]

})

export class AppComponent {

title: string = 'Hello, Angular Components!';

}

In this example, AppComponent is a component that displays a title. The selector specifies the HTML tag to use in the template.

**2. Structural Directives**

Structural directives alter the layout of the DOM by adding or removing elements. Common structural directives include \*ngIf, \*ngFor, and \*ngSwitch.

**Example: Using \*ngIf**

typescript

Copy code

// app.component.ts

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

@Component({

selector: 'app-root',

template: `

<h1 \*ngIf="isVisible">Hello, Angular Structural Directives!</h1>

<button (click)="toggleVisibility()">Toggle Visibility</button>

`,

})

export class AppComponent {

isVisible: boolean = true;

toggleVisibility() {

this.isVisible = !this.isVisible;

}

}

In this example, the \*ngIf directive conditionally displays the heading based on the value of isVisible. The button toggles the visibility of the heading.

**Example: Using \*ngFor**

typescript

Copy code

// app.component.ts

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

@Component({

selector: 'app-root',

template: `

<ul>

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

</ul>

`,

})

export class AppComponent {

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

}

In this example, \*ngFor iterates over the items array and generates a list item for each element.

**3. Attribute Directives**

Attribute directives modify the appearance or behavior of an existing element. Common attribute directives include ngClass, ngStyle, and custom directives.

**Example: Using ngClass**

typescript

Copy code

// app.component.ts

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

@Component({

selector: 'app-root',

template: `

<h1 [ngClass]="{'highlight': isHighlighted}">Hello, Angular Attribute Directives!</h1>

<button (click)="toggleHighlight()">Toggle Highlight</button>

`,

styles: [`

.highlight {

color: red;

font-weight: bold;

}

`]

})

export class AppComponent {

isHighlighted: boolean = false;

toggleHighlight() {

this.isHighlighted = !this.isHighlighted;

}

}

In this example, ngClass adds or removes the highlight class based on the isHighlighted property. Clicking the button toggles the class.

**Example: Custom Attribute Directive**

You can also create custom attribute directives. Here’s an example that changes the background color of an element.

typescript

Copy code

// highlight.directive.ts

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

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

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

@HostListener('mouseenter') onMouseEnter() {

this.highlight('yellow');

}

@HostListener('mouseleave') onMouseLeave() {

this.highlight(null);

}

private highlight(color: string) {

this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', color);

}

}

**Usage in Component:**

typescript

Copy code

// app.component.ts

@Component({

selector: 'app-root',

template: `<h1 appHighlight>Hello, Angular Custom Attribute Directive!</h1>`,

})

export class AppComponent {}

In this example, the HighlightDirective changes the background color of the element when the mouse enters and leaves.

**Summary**

* **Components**: Encapsulate HTML, CSS, and behavior; they are the most common directive type.
* **Structural Directives**: Change the structure of the DOM (e.g., \*ngIf, \*ngFor).
* **Attribute Directives**: Change the appearance or behavior of existing elements (e.g., ngClass, ngStyle, custom directives).

Directives enhance the reusability and maintainability of Angular applications by allowing developers to create reusable pieces of functionality that can be easily applied across different components and templates.

**Dependency Injection (DI) in Angular explore with example**

Dependency Injection (DI) is a design pattern used in Angular to improve the modularity and testability of applications. It allows a class to receive its dependencies from an external source rather than creating them internally. This leads to better separation of concerns and makes it easier to manage dependencies.

**How Dependency Injection Works in Angular**

In Angular, DI is managed through the injector, which is a service that is responsible for creating and providing instances of classes, including components and services. You define dependencies in the constructor of the class, and Angular takes care of resolving those dependencies.

**Key Concepts of DI in Angular**

1. **Providers**: A provider is a way to configure the injector. It tells Angular how to create an instance of a dependency.
2. **Injectors**: An injector is a service that knows how to create and manage the instances of the services it provides.
3. **Tokens**: A token is a unique identifier for a dependency, which can be a string or a class reference.

**Example of Dependency Injection in Angular**

Let's create a simple example to demonstrate DI in Angular using a service and a component.

**Step 1: Create a Service**

Create a service that provides a message.

// message.service.ts

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

@Injectable({

providedIn: 'root', // This makes the service available in the root injector

})

export class MessageService {

getMessage(): string {

return 'Hello from the Message Service!';

}

}

**Step 2: Create a Component that Uses the Service**

Now, create a component that injects the MessageService and uses it.

typescript

Copy code

// app.component.ts

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

import { MessageService } from './message.service'; // Import the service

@Component({

selector: 'app-root',

template: `<h1>{{ message }}</h1>`,

})

export class AppComponent {

message: string;

constructor(private messageService: MessageService) {

// Inject the service through the constructor

this.message = this.messageService.getMessage(); // Use the service

}

}

**Step 3: Register the Service in the Module**

Since we used the providedIn: 'root' metadata in the service, it is automatically registered in the root injector, and you don't need to add it explicitly in the providers array of any module. However, if you have a more complex setup, you might need to register it in a specific module.

typescript

Copy code

// app.module.ts

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

import { BrowserModule } from '@angular/platform-browser';

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

import { MessageService } from './message.service'; // Import the service

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule],

providers: [MessageService], // Optional, if not using providedIn

bootstrap: [AppComponent],

})

export class AppModule {}

**Step 4: Using the Application**

In your index.html, you can use the <app-root></app-root> tag to bootstrap the Angular application.

**Summary of the Example**

* **MessageService**: A simple service that returns a message.
* **AppComponent**: A component that injects MessageService to retrieve the message and display it in the template.
* **Dependency Injection**: The service is injected into the component, promoting loose coupling and making the application easier to test.

**Types of data bindings in Angular**

In Angular, data binding is a key concept that allows synchronization between the model (data) and the view (UI). There are four primary types of data binding in Angular:

**1. Interpolation (String Interpolation)**

Interpolation allows you to bind data from your component's class to the view using double curly braces ({{ }}). It is primarily used for displaying string values in the template.

**Example:**

typescript

Copy code

// component.ts

export class AppComponent {

title: string = 'Hello, Angular!';

}

html

Copy code

<!-- component.html -->

<h1>{{ title }}</h1> <!-- Output: Hello, Angular! -->

**2. Property Binding**

Property binding allows you to bind data to the properties of HTML elements or directives. It is done using square brackets ([ ]). This is particularly useful for setting properties of DOM elements.

**Example:**

typescript

Copy code

// component.ts

export class AppComponent {

isDisabled: boolean = true;

}

html

Copy code

<!-- component.html -->

<button [disabled]="isDisabled">Click Me</button> <!-- Button will be disabled -->

**3. Event Binding**

Event binding allows you to listen to events (like clicks, key presses, etc.) and call methods in your component class. It is done using parentheses (( )).

**Example:**

typescript

Copy code

// component.ts

export class AppComponent {

onClick() {

alert('Button clicked!');

}

}

html

Copy code

<!-- component.html -->

<button (click)="onClick()">Click Me</button> <!-- Alert will be shown on click -->

**4. Two-Way Data Binding**

Two-way data binding combines property binding and event binding. It allows for automatic synchronization between the model and the view. It is achieved using the [(ngModel)] directive, which requires importing the FormsModule.

**Example:**

typescript

Copy code

// component.ts

export class AppComponent {

name: string = '';

}

html

Copy code

<!-- component.html -->

<input [(ngModel)]="name" placeholder="Enter your name">

<p>Your name is: {{ name }}</p> <!-- Updates in real-time as you type -->

**Summary**

* **Interpolation**: Used to display data in the view.
* **Property Binding**: Binds data to DOM element properties.
* **Event Binding**: Binds events to methods in the component.
* **Two-Way Data Binding**: Synchronizes data between the model and the view.

These data binding techniques help in building dynamic and responsive Angular applications by enabling seamless communication between the components and the UI.

1. **What is two-way data binding in Angular?**
   * **Answer**: Two-way data binding allows synchronization of data between the view and the component class. It is achieved using the [(ngModel)] directive, which binds data in both directions.
2. **What are Angular Directives?**
   * **Answer**: Directives are instructions in the DOM. There are three types:
     + **Structural Directives**: \*ngIf, \*ngFor (used to change DOM structure).
     + **Attribute Directives**: ngClass, ngStyle (used to change appearance or behavior of an element).
     + **Custom Directives**: Developers can create their own directives.
3. **What is a Service in Angular?**
   * **Answer**: A service is a class that holds business logic and can be injected into different components using Dependency Injection (DI). Services are commonly used for data sharing, managing business rules, and interacting with external APIs.

**Experienced-Level Angular Interview Questions (4+ Years)**

1. **What is the difference between Angular and AngularJS?**
   * **Answer**:
     + **Angular** is a complete rewrite of **AngularJS**. Angular uses TypeScript, while AngularJS uses JavaScript.
     + Angular has a modular structure (based on components), while AngularJS is based on MVC.
     + Angular is much faster due to ahead-of-time (AOT) compilation, while AngularJS is interpreted.
2. **How does Dependency Injection work in Angular?**
   * **Answer**: Angular provides a built-in DI mechanism to inject dependencies (like services) into components or other services. Angular’s DI creates and maintains dependencies in the providers array of the @NgModule or @Component. DI helps decouple classes, making code more testable and scalable.
3. **Explain the difference between ngOnInit() and constructor() in Angular components.**
   * **Answer**:
     + constructor(): It is a standard TypeScript class method that initializes the class. It is called when an instance of the class is created.
     + ngOnInit(): It is a lifecycle hook provided by Angular, and it is called after Angular has initialized the component's data-bound properties. This is the preferred place to write initialization logic related to Angular.
4. **How do you implement routing in Angular?**
   * **Answer**: Angular uses the RouterModule to manage routing. You define routes in the app-routing.module.ts file. Example:

const routes: Routes = [

{ path: '', component: HomeComponent },

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

{ path: '\*\*', component: PageNotFoundComponent }

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

1. **What are Reactive Forms, and how do they differ from Template-driven Forms?**
   * **Answer**:
     + **Reactive Forms**: Built using FormControl, FormGroup, and FormBuilder. They provide more control over form state and validation and allow reactive, dynamic form fields.
     + **Template-driven Forms**: Use directives like ngModel in the template. They are simple but offer less control compared to Reactive Forms.

Example of a reactive form:

typescript

Copy code

this.profileForm = this.fb.group({

name: [''],

age: ['']

});

1. **How do you handle HTTP requests in Angular?**
   * **Answer**: Angular provides the HttpClient module to handle HTTP requests. You need to import HttpClientModule in app.module.ts and use HttpClient to send GET, POST, PUT, DELETE requests.

Example of an HTTP GET request:

typescript

Copy code

constructor(private http: HttpClient) {}

getData() {

this.http.get('https://api.example.com/data')

.subscribe(data => console.log(data));

}

1. **What is Change Detection in Angular, and how does it work?**
   * **Answer**: Change detection in Angular is a mechanism to update the view whenever the model state changes. Angular provides two change detection strategies:
     + **Default**: Angular checks every component for changes during every event cycle.
     + **OnPush**: Angular checks the component only when its input properties change, improving performance for large applications.
2. **How do you improve the performance of Angular applications?**
   * **Answer**:
     + Use **OnPush** change detection strategy.
     + Optimize **Lazy Loading** of modules.
     + Minimize the bundle size by using **Ahead-of-Time (AOT)** compilation.
     + Use **trackBy** with ngFor to optimize rendering.
     + Debounce high-frequency events like keystrokes.
     + Use **service workers** for caching and offline support.
3. **What are Angular Guards?**
   * **Answer**: Angular Guards are used to control access to routes in Angular applications. There are different types:
     + **CanActivate**: Checks if the user can access a route.
     + **CanDeactivate**: Checks if the user can leave the route.
     + **Resolve**: Fetches data before route activation.
     + **CanLoad**: Checks if the user can load a module asynchronously.
4. **What is the purpose of the async pipe in Angular?**
   * **Answer**: The async pipe automatically subscribes to observables and promises in templates and handles their lifecycle, including subscribing and unsubscribing when necessary.

**Code Example: Reactive Form in Angular**

typescript

Copy code

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

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({

selector: 'app-reactive-form',

templateUrl: './reactive-form.component.html',

})

export class ReactiveFormComponent {

profileForm: FormGroup;

constructor(private fb: FormBuilder) {

this.profileForm = this.fb.group({

name: ['', Validators.required],

age: ['', [Validators.required, Validators.min(18)]]

});

}

onSubmit() {

console.log(this.profileForm.value);

}

}

**Conclusion**

For a developer with 4 years of experience, you are expected to have a good grasp of both the **core concepts** and **advanced topics** of Angular. Prepare to discuss your hands-on experience with topics like **services, forms, routing, dependency injection,** and performance optimization techniques.

**Routings in Angular explain with proper example**

In Angular, **routing** is a powerful feature that allows you to navigate between different views or components in your application. The Angular Router enables you to define routes and manage navigation within your application, making it easier to create single-page applications (SPAs).

**Key Concepts in Angular Routing**

1. **Routes**: Define the mapping between URL paths and components.
2. **RouterModule**: A module that provides routing functionality.
3. **RouterOutlet**: A directive that acts as a placeholder for the routed components.
4. **RouterLink**: A directive that allows navigation to defined routes.
5. **Route Parameters**: Enable passing data via the URL.
6. **Route Guards**: Allow you to control access to routes.

**Setting Up Routing in Angular**

Here’s a step-by-step guide to setting up routing in an Angular application:

**Step 1: Create a New Angular Application**

ng new angular-routing-example

cd angular-routing-example

**Step 2: Generate Components**

Generate some components that you will use for routing.

ng generate component home

ng generate component about

ng generate component contact

**Step 3: Define Routes**

Open app-routing.module.ts (or create one if it doesn’t exist) to define your application routes.

// app-routing.module.ts

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

import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';

import { ContactComponent } from './contact/contact.component';

const routes: Routes = [

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

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

{ path: 'contact', component: ContactComponent },

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

In this code:

* We define an array of routes. Each route maps a URL path to a component.
* The RouterModule.forRoot(routes) method sets up the router with the defined routes.

**Step 4: Import AppRoutingModule**

Now, import the AppRoutingModule in your main application module (app.module.ts).

typescript

Copy code

// app.module.ts

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

import { BrowserModule } from '@angular/platform-browser';

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

import { AppRoutingModule } from './app-routing.module';

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';

import { ContactComponent } from './contact/contact.component';

@NgModule({

declarations: [

AppComponent,

HomeComponent,

AboutComponent,

ContactComponent,

],

imports: [

BrowserModule,

AppRoutingModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**Step 5: Add RouterOutlet**

In your main application component (app.component.html), add the RouterOutlet directive, which serves as a placeholder for the routed components.

html

Copy code

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

<nav>

<a routerLink="/">Home</a>

<a routerLink="/about">About</a>

<a routerLink="/contact">Contact</a>

</nav>

<router-outlet></router-outlet>

**Step 6: Add Component Templates**

Next, add some content to each of the components:

**home.component.html**:

html

Copy code

<h2>Welcome to the Home Page</h2>

<p>This is the home page of the Angular routing example.</p>

**about.component.html**:

html

Copy code

<h2>About Us</h2>

<p>This is the about page. Here you can find more information.</p>

**contact.component.html**:

html

Copy code

<h2>Contact Us</h2>

<p>This is the contact page. You can reach us here.</p>

**Step 7: Serve the Application**

Run the application using the following command:

bash

Copy code

ng serve

Now, navigate to http://localhost:4200 in your browser. You should see the home page, and you can navigate to the About and Contact pages by clicking the corresponding links.

**Route Parameters**

You can also define routes that accept parameters. For example, if you want to create a route for user profiles:

**Define the Route**:

typescript

Copy code

{ path: 'user/:id', component: UserProfileComponent }

**Accessing Route Parameters**: In the UserProfileComponent, you can access the route parameter using the ActivatedRoute service.

typescript

Copy code

// user-profile.component.ts

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

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

@Component({

selector: 'app-user-profile',

template: `<h2>User Profile</h2><p>User ID: {{ userId }}</p>`,

})

export class UserProfileComponent implements OnInit {

userId!: string;

constructor(private route: ActivatedRoute) {}

ngOnInit() {

this.userId = this.route.snapshot.paramMap.get('id')!;

}

}

**Route Guards**

Route guards allow you to control access to certain routes based on conditions (e.g., authentication). You can create a guard using Angular CLI:

bash

Copy code

ng generate guard auth

Then implement the guard logic in the generated auth.guard.ts file.

**Summary**

* **Routing** in Angular allows for navigation between different views or components.
* **RouterModule** is used to configure the router and define routes.
* **RouterOutlet** serves as a placeholder for routed components in the template.
* **RouterLink** enables navigation to different routes.
* You can define routes with parameters and implement route guards for access control.

Using Angular’s routing capabilities, you can build dynamic and user-friendly single-page applications that provide a seamless experience.