Angular Interview Questions for Freshers

1. Why were client-side frameworks like Angular introduced?

Back in the day, web developers used VanillaJS and jQuery to develop dynamic websites but, as the logic of one's website grew, the code became more and more tedious to maintain. For applications that use complex logic, developers had to put in extra effort to maintain the separation of concerns for the app. Also, jQuery did not provide facilities for data handling across views.

For tackling the above problems, **client-side frameworks like Angular** came into the picture, which made life easier for the developers by handling the separation of concerns and dividing code into smaller bits of information (In the case of Angular, called Components).

Client-side frameworks allow one to develop advanced web applications like Single-Page-Application. Not that we cannot develop SPAs using VanillaJS, but by doing so, the development process becomes slower.

2. How does an Angular application work?

Every Angular app consists of a file named **angular.json**. This file will contain all the configurations of the app. While building the app, the builder looks at this file to find the entry point of the application. Following is an image of the angular.json file:

```
"build": {
      "builder": "@angular-devkit/build-angular:browser",
      "options": {
        "outputPath": "dist/angular-starter",
        "index": "src/index.html",
        "main": "src/main.ts",
        "polyfills": "src/polyfills.ts",
        "tsConfig": "tsconfig.app.json",
        "aot": false,
        "assets": [
          "src/favicon.ico",
          "src/assets"
        ],
        "styles": [
          "./node_modules/@angular/material/prebuilt-themes/deeppurple-amber.css",
          "src/style.css"
        ]
      }
    }
```

Inside the build section, the main property of the options object defines the entry point of the application which in this case is **main.ts**.

The main.ts file creates a browser environment for the application to run, and, along with this, it also calls a function called **bootstrapModule**, which bootstraps the

application. These two steps are performed in the following order inside the main.ts file:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
platformBrowserDynamic().bootstrapModule(AppModule)
```

In the above line of code, **AppModule** is getting bootstrapped.

The AppModule is declared in the app.module.ts file. This module contains declarations of all the components.

Below is an example of app.module.ts file:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  entryComponents: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

As one can see in the above file, **AppComponent** is getting bootstrapped.

This component is defined in **app.component.ts** file. This file interacts with the webpage and serves data to it.

Below is an example of app.component.ts file:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular';
}
```

Each component is declared with three properties:

- **Selector** - used for accessing the component
- **Template/TemplateURL** - contains HTML of the component
- **StylesURL** - contains component-specific stylesheets

After this, Angular calls the **index.html** file. This file consequently calls the root component that is **app-root**. The root component is defined in **app.component.ts**. This is how the index.html file looks:

```
<!doctype html>
    <html lang="en">
    <head>
      <meta charset="utf-8">
      <title>Angular</title>
      <base href="/">
      <meta name="viewport" content="width=device-width, initial-scale=1">
    </head>
    <body>
      <app-root></app-root>
    </body>
    </html>
```

The HTML template of the root component is displayed inside the `<app-root>` tags.

This is how every angular application works.

3. What are some of the advantages of Angular over other frameworks?

- **Features that are provided out of the box** - Angular provides a number of built-in features like routing, state management, rxjs library and http servicesstraight out of the box. This means that one does not need to look for the above-stated features separately. They are all provided with angular.
- **Declarative UI** - Angular uses HTML to render the UI of an application. HTML is a declarative language and is much easier to use than JavaScript.
- **Long-term Google support** - Google announced Long-term support for Angular. This means that Google plans to stick with Angular and further scale up its ecosystem.

You can download a PDF version of Angular Interview Questions.

---

4. What are the advantages of Angular over React?

Angular vs React: **Check out the differences**

Angular	React
Angular supports bidirectional data binding as well as mutable data.	React only supports unidirectional and immutable data binding.
The biggest benefit of Angular is that it enables dependency injection.	React allows us to either accomplish it ourselves or with the aid of a third-party library.

Angular	React
Angular can be used in both mobile and web development.	React can only be used in UI development only.
Angular features a wide wide range of tools, libraries, frameworks, plugins, and so on that make development faster and more fun.	In React we can use third-party libraries for any features.
Angular uses Typescript.	React uses Javascript.

5. List out differences between AngularJS and Angular?

Check out the differences between AngularJS and Angular below. For More Information, **Click here**.

Features	AngularJS	Angular
Architecture	AngularJS uses MVC or Model-View-Controller architecture, where the Model contains the business logic, the Controller processes information and the View shows the information present in the Model.	Angular replaces controllers with Components. Components are nothing but directives with a predefined template.
Language	AngularJS uses JavaScript language, which is a dynamically typed language.	Angular uses TypeScript language, which is a statically typed language and is a superset of JavaScript. By using statically typed language, Angular provides better performance while developing larger applications.
Mobile Support	AngularJS does not provide mobile support.	Angular is supported by all popular mobile browsers.
Structure	While developing larger applications, the process of maintaining code becomes tedious in the case of AngularJS.	In the case of Angular, it is easier to maintain code for larger applications as it provides a better structure.
Expression Syntax	While developing an AngularJS application, a developer needs to remember the correct ng-directive for binding an event or a property.	Whereas in Angular, property binding is done using "[ ]" attribute and event binding is done using "( )" attribute.

6. How are Angular expressions different from JavaScript expressions?

The first and perhaps, the biggest difference is that Angular expressions allow us to write JavaScript in HTML which is not the case when it comes to JavaScript expressions.

Next, Angular expressions are evaluated against a **local** scope object whereas JavaScript expressions are against a **global** window object. Let's understand that better with an example :

Consider the following component named test:

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

    @Component({
      selector: 'app-test',
      template: `
          <h4>{{message}}</h4>
      `,
      styleUrls: ['./test.component.css']
    })
    export class TestComponent implements OnInit {
      message:string = “Hello world”;
      constructor() { }

      ngOnInit() {
      }
    }
```

As one can see that Angular expression is used to display the **message** property of a component. Since we are using Angular expressions, in the present template, we cannot access a property outside of its local scope, which in this case is **TestComponent**.

This proves that Angular expressions are always evaluated based on the **scope** object rather than the global object.

The next difference is how Angular expressions handle **null** and **undefined**.

Consider the following JavaScript example:

```
<!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>JavaScript Test</title>
    </head>
    <body>
        <div id="foo"><div>
    </body>
    <script>
        'use strict';
        let bar = {};
        document.getElementById('foo').innerHTML = bar.x;
    </script>
    </html>
```

If you run the above code, you will see **undefined** displayed on the screen. Although it’s not ideal to leave any property undefined, the user does not need to see this.

Now consider the following Angular example:

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

    @Component({
      selector: 'app-new',
      template: `
          <h4>{{message}}</h4>
      `,
      styleUrls: ['./new.component.css']
    })
    export class NewComponent implements OnInit {
      message:object = {};
      constructor() { }

      ngOnInit() {
      }

    }
```

If you render the above component, you will **not** see undefined being displayed on the screen.

Next, in Angular expressions, one **cannot** use loops, conditionals and exceptions.

The difference which makes Angular expressions quite beneficial is the use of **pipes**. Angular uses pipes(called filters in AngularJS), which can be used to format data before displaying it. Let's see one predefined pipe in action:

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

    @Component({
      selector: 'app-new',
      template: `
          <h4>{{message | lowercase}}</h4>
      `,
      styleUrls: ['./new.component.css']
    })
    export class NewComponent implements OnInit {
      message:string = "HELLO WORLD";
      constructor() { }

      ngOnInit() {
      }

    }
```

In the above code, we have used a predefined pipe called **lowercase**, which transforms all the letters in lowercase. Therefore, if you render the above component, you will see "hello world" being displayed.

In contrast, JavaScript does not have the concept of **pipes.**

7. What are Single Page Applications (SPA)?

Single page applications are web based applications that only need to be loaded once, with new functionality consisting of only minor changes to the user interface. It does not load new HTML pages to display the content of the new page, but rather generates it dynamically. This is made feasible by JavaScript's ability to alter DOM components on the current page. A Single Page Application method is speedier, resulting in a more consistent user experience.

8. What are templates in Angular?

A template is a kind of HTML that instructs Angular about how to display a component. An Angular HTML template, like conventional HTML, produces a view, or user interface, in the browser, but with far more capabilities. Angular API evaluates an HTML template of a component, creates HTML, and renders it.

There are two ways to create a template in an Angular component:

- Inline Template
- Linked Template

Inline Template: The component decorator's template config is used to specify an inline HTML template for a component. The Template will be wrapped inside the single or double quotes.

Example:

```
@Component({
   selector: "app-greet",
   template: `<div>
      <h1> Hello {{name}} how are you ? </h1>
      <h2> Welcome to interviewbit ! </h2>
   </div>`
})
```

Linked Template: A component may include an HTML template in a separate HTML file. As illustrated below, the templateUrl option is used to indicate the path of the HTML template file.

Example:

```
@Component({
   selector: "app-greet",
   templateUrl: "./component.html"
})
```

9. What are directives in Angular?

A directive is a class in Angular that is declared with a **@Directive** decorator.

Every directive has its own behaviour and can be imported into various components of an application.

When to use a directive?

Consider an application, where multiple components need to have similar functionalities. The norm thing to do is by adding this functionality individually to every component but, this task is tedious to perform. In such a situation, one can create a **directive** having the required functionality and then, import the directive to components which require this functionality.

Types of directives:

1. Component directives

These form the main class in directives. **Instead** of @Directive decorator we use **@Component** decorator to declare these directives. These directives have a view, a stylesheet and a selector property.

2. Structural directives

- These directives are generally used to manipulate DOM elements.
- Every structural directive has a ' * ' sign before them.
- We can apply these directives to any DOM element.

Let's see some built-in structural directives in action:

```
<div *ngIf="isReady" class="display_name">
        {{name}}
    </div>


<div class="details" *ngFor="let x of details" >
        <p>{{x.name}}</p>
        <p> {{x.address}}</p>
        <p>{{x.age}}</p>
         </div>
```

In the above example, we can *ngIf and *ngFor directives being used.

*ngIf is used to check a boolean value and if it's truthy,the div element will be displayed.

*ngFor is used to iterate over a list and display each item of the list.

3. Attribute Directives

These directives are used to change the look and behaviour of a DOM element. Let's understand attribute directives by creating one:

How to create a custom directive?

We're going to create an attribute directive:

In the command terminal, navigate to the directory of the angular app and type the following command to generate a directive: ng g directive blueBackground

The following directive will be generated. Manipulate the directive to look like this:

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

@Directive({
 selector: '[appBlueBackground]'
})
export class BlueBackgroundDirective {
 constructor(el:ElementRef) {
   el.nativeElement.style.backgroundColor = "blue";
 }
}
```

Now we can apply the above directive to any DOM element: <p appBlueBackground>Hello World!</p>

10. Explain Components, Modules and Services in Angular

For better understanding, I would like you to create an Angular application by running the following inside the command terminal: ng new angularApp

The above command will create an angular application in the directory.

Next, let's move on to understand Components, Modules, and Services.

- **Components**

In Angular, components are the basic building blocks, which control a part of the UI for any application.

A component is defined using the **@Component** decorator. Every component consists of three parts, the template which loads the view for the component, a stylesheet which defines the look and feel for the component, and a class that contains the business logic for the component.

For creating a component, inside the command terminal, navigate to the directory of the application created, and run the following command: ng generate component testOr ng g c test

One can see the generated component inside src/app/test folder. The component will be defined inside test.component.ts and this is how it looks:

```
import { Component, OnInit } from '@angular/core';
  @Component({
    selector: 'app-test',
    templateUrl: './test.component.html',
    styleUrls: ['./test.component.css']
  })
```

```
export lass TestComponent implements OnInit {
  constructor() {}
  ngOnInit() {
  }
}
```

As we can see in the above image, our component is defined with **@Component** decorator.

- **Modules**

A module is a place where we can group components, directives, services, and pipes. Module decides whether the components, directives, etc can be used by other modules, by exporting or hiding these elements. Every module is defined with a @NgModule decorator.

By default, modules are of two types:

- Root Module
- Feature Module

Every application can have only one root module whereas, it can have one or more feature modules.

A root module imports **BrowserModule**, whereas a feature module imports **CommonModule**.

In the application that we created before, one can see that the root module is defined inside **app.module.ts** and this is how it looks:

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

import { AppComponent } from './app.component';
import { TestComponent } from './test/text.component';

@NgModule({
  declarations: [
    AppComponent,
    TestComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

We can see in the above image that the component we created earlier is already imported in the declarations array.

To create a feature module, run the following command: ng g m test-module

The module is created inside the src/app/test-module/test-module.module.ts file:

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

@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class TestModuleModule { }
```

As one can see, **CommonModule** is imported since this is a feature module.

- **Services**

Services are objects which get instantiated only once during the lifetime of an application. The main objective of a service is to share data, functions with different components of an Angular application.

A service is defined using a **@Injectable** decorator. A function defined inside a service can be invoked from any component or directive.

To create a service, run the following command: ng g s test-service

The service will be created inside src/app/test-service.service.ts:

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

@Injectable({
  providedIn: 'root'
})
export class TestServiceService {

  constructor() { }

}
```

Any method/function defined inside the TestServiceService class can be directly used inside any component by just importing the service.

11. What is the scope?

In Angular, a scope is an object that refers to the application model. It is a context in which expressions can be executed. These scopes are grouped hierarchically, comparable to the DOM structure of the application. A scope aids in the propagation of various events and the monitoring of expressions.

12. What is data binding in Angular?

Data binding is one of the most significant and effective elements for creating communication between the DOM and the component. It makes designing interactive apps easier by reducing the need to worry about data pushing and pulling between the component and the template.

There are Four types of Data binding in Angular:

- Property Binding
- Event Binding
- String Interpolation
- Two way data binding

Property Binding: One method of data binding is called property binding. In property binding, we connect a DOM element's property to a field that is a declared property in our TypeScript component code. In reality, Angular transforms string interpolation into property binding internally.

Event Binding: Using event binding, you may respond to DOM events like button clicks and mouse movements. When a DOM event (such as a click, change, or keyup) occurs, the component's designated method is called.

String Interpolation: In order to export data from TypeScript code to an HTML template(view), String Interpolation is a one way data binding approach. The data from the component is shown to the view using the template expression enclosed in double curly braces. The value of a component property is added by using string interpolation.

13. What is two way data binding in Angular?

Data sharing between a component class and its template is referred to as two-way data binding. If you alter data in one area, it will immediately reflate at the other end. This happens instantly and automatically, ensuring that the HTML template and TypeScript code are always up to date. Property binding and event binding are coupled in two-way data binding.

Example:

app.component.ts

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

@Component({
  selector: "app",
  templateUrl: "./app.component.html",
```

```
})
export class AppComponent {
  data = "This is an example component of two way data binding.";
}
```

app.component.html

```
<input [(ngModel)]="data"  type="text">
  <br> <br>
  <h2> You entered the data:  {{data}}</h2>
```

app.module.ts

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

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

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

14. What are Decorators and their types in Angular?

Decorators are a fundamental concept in TypeScript, and because Angular heavily relies on TypeScript, decorators have become an important element of Angular as well.

Decorators are methods or design patterns that are labeled with a prefixed @ symbol and preceded by a class, method, or property. They enable the modification of a service, directive, or filter before it is utilized. A decorator, in essence, provides configuration metadata that specifies how a component, class, or method should be processed, constructed, and used at runtime. Angular includes a number of decorators which attach various types of metadata to classes, allowing the system to understand what all these classes signify and how they should function.

Types of decorators:

- **Method Decorator:** Method decorators, as the name implies, are used to add functionality to the methods defined within our class.
- **Class Decorator:** Class Decorators are the highest-level decorators that determine the purpose of the classes. They indicate to Angular that a specific class is a component or module. And the decorator enables us to declare this effect without having to write any code within the class.
- **Parameter Decorator:** The arguments of your class constructors are decorated using parameter decorators.

- **Property Decorator:** These are the second most popular types of decorators. They enable us to enhance some of the properties in our classes.

15. What are annotations in Angular ?

These are language features that are hard-coded. Annotations are merely metadata that is set on a class to reflect the metadata library. When a user annotates a class, the compiler adds an annotations property to the class, saves an annotation array in it, and then attempts to instantiate an object with the same name as the annotation, providing the metadata into the constructor. Annotations in AngularJs are not predefined, therefore we can name them ourselves.

16. What are pure Pipes?

These are pipelines that only employ pure functions. As a result, a pure pipe does not employ any internal state, and the output remains constant as long as the parameters provided remain constant. Angular calls the pipe only when the parameters being provided change. A single instance of the pure pipe is utilized in all components.

17. What are impure pipes?

Angular calls an impure pipe for each change detection cycle, independent of the change in the input fields. For each of these pipes, several pipe instances are produced. These pipes' inputs can be altered.

By default, all pipelines are pure. However, as demonstrated below, you can specify impure pipes using the pure property.

Example:

```
@Pipe({
   name: 'impurePipe',
   pure: false/true
})
export class ImpurePipe {}
```

18. What is Pipe transform Interface in Angular?

An interface used by pipes to accomplish a transformation. Angular calls the transform function with the value of a binding as the first argument and any arguments as the second parameter in list form. This interface is used to implement custom pipes.

Example:

```
import { Pipe, PipeTransform } from '@angular/core';
 @Pipe({
  name: 'tranformpipe'
```

```
})
export class TranformpipePipe implements PipeTransform {
  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }
}
```

19. Write a code where you have to share data from the Parent to Child Component?

You have to share the data amongst the components in numerous situations. It may consist of unrelated, parent-child, or child-parent components.

The @Input decorator allows any data to be sent from parent to child.

```
// parent component
import { Component } from '@angular/core';
@Component({
  selector: 'app-parent',
  template: `
<app-child [childMessage]="parentMessage"></app-child>
`,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent{
  parentMessage = "message from parent"
  constructor() { }
}
// child component
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-child',
  template: `Say {{ childMessage }}`,
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  @Input() childMessage: string;
  constructor() { }
}
```

20. Create a TypeScript class with a constructor and a function.

```
class IB {
    name: string;
    constructor(message: string) {
      this.name = message;
    }
    greet() {
      return "Hello, " + this.name + "How are you";
    }
  }
  let msg = new IB("IB");
```

Angular Interview Questions for Experienced

21. Angular by default, uses client-side rendering for its applications.

Can one make an angular application to render on the server-side?

Yes, angular provides a technology called **Angular Universal**, which can be used to render applications on the server-side.

The advantages of using Angular Universal are:

- First time users can instantly see a view of the application. This benefits in providing **better user experience**.
- Many search engines expect pages in plain HTML, thus, Universal can make sure that your content is available on every search engine, which leads to **better SEO**.
- Any server-side rendered application **loads faster** since rendered pages are available to the browser sooner.

22. What is Eager and Lazy loading?

- **Loading:** The eager loading technique is the default module-loading strategy. Eager loading feature modules are loaded before the program begins. This is primarily utilized for small-scale applications.
- **Lazy Loading:** Lazy loading loads the feature modules dynamically as needed. This speeds up the application. It is utilized for larger projects where all of the modules are not required at the start.

23. What is view encapsulation in Angular?

View encapsulation specifies if the component's template and styles can impact the entire program or vice versa.

Angular offers three encapsulation methods:

- **Native:** The component does not inherit styles from the main HTML. Styles defined in this component's @Component decorator are only applicable to this component.
- **Emulated (Default):** The component inherits styles from the main HTML. Styles set in the @Component decorator are only applicable to this component.
- **None:** The component's styles are propagated back to the main HTML and therefore accessible to all components on the page. Be wary of programs that have None and Native components. Styles will be repeated in all components with Native encapsulation if they have No encapsulation.

24. What are RxJs in Angular ?

RxJS is an acronym that stands for Reactive Extensions for JavaScript. It is used to enable the use of observables in our JavaScript project, allowing us to do reactive

programming. RxJS is utilized in many popular frameworks, including Angular since it allows us to compose our asynchronous or callback-based code into a sequence of operations executed on a data stream that releases values from a publisher to a subscriber. Other programming languages, such as Java and Python, offer packages that allow them to develop reactive programs utilizing observables.

Most of the time, rxJs is used in HTTP calls with angular. Because http streams are asynchronous data, we can subscribe to them and apply filters to them.

Example: The following is a simple example of how RxJs can be utilized with HTTP calls.

```
let  stream1 = httpc.get("https://www.example.com/somedata");
 let stream2 = stream1.pipe(filter(x=>x>3));
 stream2.subscribe(res=>this.Success(res),res=>this.Error(res));
```

25. Explain string interpolation and property binding in Angular.

- String interpolation and property binding are parts of **data-binding** in Angular.
- Data-binding is a feature in angular, which provides a way to communicate between the component(Model) and its view(HTML template).
- Data-binding can be done in two ways, **one-way** binding and **two-way** binding.
- In Angular, data from the component can be inserted inside the HTML template. In one-way binding, any changes in the component will directly reflect inside the HTML template but, vice-versa is not possible. Whereas, it is possible in two-way binding.
- String interpolation and property binding allow only one-way data binding.
- String interpolation uses the double curly braces **{{ }}** to display data from the component. Angular automatically runs the expression written inside the curly braces, for example, {{ 2 + 2 }} will be evaluated by Angular and the output 4, will be displayed inside the HTML template. Using property binding, we can bind the DOM properties of an HTML element to a component's property. Property binding uses the square brackets **[ ]** syntax.

26. How are observables different from promises?

The first difference is that an Observable is **lazy** whereas a Promise is **eager**.

Promise	Observable
Emits a single value	Emits multiple values over a period of time
Not Lazy	Lazy. An observable is not called until we subscribe to the observable
Cannot be cancelled	Can be cancelled by using the unsubscribe() method

Promise	Observable
	Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc.

Consider the following Observable:

```
const observable = rxjs.Observable.create(observer => {
      console.log('Text inside an observable');
      observer.next('Hello world!');
      observer.complete();
    });
    console.log('Before subscribing an Observable');
    observable.subscribe((message)=> console.log(message));
```

When you run the above Observable, you can see messages being displayed in the following order:

```
Before subscribing an Observable
Text inside an observable
Hello world!
```

As you can see, observables are lazy. Observable runs only when someone subscribes to them hence, the message “Before subscribing…” is displayed ahead of the message inside the observable.

Now let’s consider a Promise:

```
const promise = new Promise((resolve, reject) => {
      console.log('Text inside promise');
      resolve('Hello world!');
    });
    console.log('Before calling then method on Promise');
    greetingPoster.then(message => console.log(message));
```

Running the above promise, the messages will be displayed in the following order:

```
Text inside promise
Before calling then method on Promise
Hello world!
```

As you can see the message inside Promise is displayed first. This means that a promise runs before the **then** method is called. Therefore, promises are **eager**.

The next difference is that Promises are always **asynchronous**. Even when the promise is immediately resolved. Whereas an Observable, can be both **synchronous** and **asynchronous**.

The above example of an observable is the case to show that an observable is synchronous. Let’s see the case where an observable can be asynchronous:

```
const observable = rxjs.Observable.create(observer => {
      setTimeout(()=>{
          observer.next('Hello world');
          observer.complete();
      },3000)
    });
    console.log('Before calling subscribe on an Observable');
    observable.subscribe((data)=> console.log(data));
    console.log('After calling subscribe on an Observable');
```

The messages will be displayed in the following order:

```
Before calling subscribe on an Observable
After calling subscribe on an Observable
Hello world!
```

You can see in this case, observable runs asynchronously.

The next difference is that Observables can emit **multiple** values whereas Promises can emit only one value.

The biggest feature of using observables is the use of **operators**. We can use multiple operators on an observable whereas, there is no such feature in a promise.

27. Explain the concept of Dependency Injection?

Dependency injection is an application design pattern which is implemented by Angular.

It also forms one of the core concepts of Angular.

So what is dependency injection in simple terms?

Let's break it down, dependencies in angular are nothing but services which have functionality. The functionality of a service, can be needed by various components and directives in an application. Angular provides a smooth mechanism by which we can inject these dependencies into our components and directives.

So basically, we are just making dependencies which are injectable across all components of an application.

Let's understand how DI (Dependency Injection) works:

Consider the following service, which can be generated using: ng g service test

```
import { Injectable } from '@angular/core';
    @Injectable({
      providedIn: 'root'
```

```
})
export class TestService {
  importantValue:number = 42;
  constructor() { }
  returnImportantValue(){
    return this.importantValue;
  }
}
```

As one can notice, we can create injectable dependencies by adding the **@Injectable** decorator to a class.

We inject the above dependency inside the following component:

```
import { TestService } from './../test.service';
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-test',
  templateUrl: './test.component.html',
  styleUrls: ['./test.component.css']
})
export class TestComponent implements OnInit {
  value:number;
  constructor(private testService:TestService) { }

  ngOnInit() {
    this.value = this.testService.returnImportantValue();
  }
}
```

One can see we have imported our TestService at the top of the page. Then, we created an instance inside the constructor of the component and implemented the **returnImportantValue** function of the service.

From the above example, we can observe how angular provides a smooth way to inject dependencies in any component.

28. What are pipes in Angular explain with an example?

Pipes are functions that simplify the process of wiring up your JavaScript expressions and transforming them into their desired output. They can be compared to, say, string functions in other programming languages. Pipes also allow you to combine multiple expressions together, whether they're all values or some values and some declarations.

For example:

```
var css = myTheme.color | "red" ;
```

This line would assign a value to `css` , and it's equivalent to writing out the following code:

Pipes have several built-in functions that allow you to transform data, such as value and extract. We can also create our own custom pipes.

Pipes are data transformers that execute on an Angular Component's output. They take in data and return transformed data. For example, if you have an expression such as number | 1000, the number pipe will take data from the output and transform it into 1000. In Angular, there are many built-in pipes that you can use. You can also create your own custom pipes by implementing the PipeTransform interface in a class.

Pipes receive an input which can be a value expression, a function returning an expression, or even a component property., that outputs a number with a value of 1,000. With a pipe, you can transform this output into a formatted string of "1,000" or "1.000".

Example:

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

@Component({
selector: 'app-root',
template: `{{ title | uppercase}}`,
styleUrls: ['./app.component.css']
})
export class AppComponent {
title = 'this is an example of custom pies in angular';
}
```

Output:

THIS IS AN EXAMPLE OF CUSTOM PIPES IN ANGULAR

29. What exactly is a parameterized pipe?

To fine-tune its output, a pipe can receive any number of optional parameters. The parameterized pipe is constructed by first stating the pipe name followed by a colon (:) and then the parameter value. If the pipe supports several arguments, use colons to separate the values.

Example: Let's look at a birthday with a certain format (dd/MM/yyyy):

```
import { Component } from '@angular/core';
   @Component({
     selector: 'app-example',
     template: `<p>Birthday is {{ birthday | date:'dd/MM/yyyy'}}</p>`
   })
   export class ExampleComponent {
     birthday = new Date(2000, 7, 15);
```

```
  }
```

30. What are class decorators?

Class Decorators are the highest-level decorators that determine the purpose of the classes. They indicate to Angular that a specific class is a component or module. And the decorator enables us to declare this effect without having to write any code within the class.

Example:

```
import { NgModule, Component } from '@angular/core';
@Component({
  selector: 'class-component',
  template: '<div> This is a class component ! </div>',
})
export class ClassComponent {
  constructor() {
    console.log('This is a class component!');
  }
}
@NgModule({
  imports: [],
  declarations: [],
})
export class ClassModule {
  constructor() {
    console.log('This is a class module!');
  }
}
```

It is a component or module in which no code in the class is required to tell Angular. We only need to design it, and Angular will take care of the rest.

31. What are Method decorators?

Method decorators, as the name implies, are used to add functionality to the methods defined within our class.

Example: @HostListener, is a good example of method decorators.

```
import { Component, HostListener } from '@angular/core';
@Component({
  selector: 'method-component',
  template: '<div> This is a test method component ! </div>',
})
export class MethodComponent {
  @HostListener('click', ['$event'])
    onHostClick(event: Event) {
    console.log('clicked now this event is available !');
    }
}
```

The @HostListener decorator is used before the onHostClick () method in the above example code.

32. What are property decorators?

These are the second most popular types of decorators. They enable us to enhance some of the properties in our classes. We can certainly understand why we utilize any certain class property by using a property decorator.

There are many property decorators available for example @Input(), @Output, @ReadOnly(), @Override()

Example:

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'prop-component',
  template: '<div> This is a test component ! </div>'
})
export class PropComponent {
  @Input()
  exampleProperty: string;
}
```

The input binding would be sent via a component property binding:

```
<prop-component
  [propProperty]="propData">
</prop-component>
```

33. What is the Component Decorator in Angular?

TypeScript classes are used to create components. This class genre is then decorated with the "@Component" decorator. The decorator's function is to take a metadata object holding component information and decorate it.

A Decorator is always preceded by @. The Decorator must come before the class definition. We can also make our own decorators.

Example: The example below shows us a Class decorated with the @Component decorator.

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

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    title = 'Example component';
}
```

The metadata object received by the decorator has values such as templateUrl, selector, and others, with the templateUrL property pointing to an HTML file that defines what you see on the application.

34. What are lifecycle hooks in Angular? Explain a few lifecycle hooks.

Every component in Angular has a lifecycle, and different phases it goes through from the time of creation to the time it's destroyed. Angular provides **hooks** to tap into these phases and trigger changes at specific phases in a lifecycle.

- **ngOnChanges( )** This hook/method is called before ngOnInit and whenever one or more input properties of the component change.
  This method/hook receives a SimpleChanges object which contains the previous and current values of the property.
- **ngOnInit( )** This hook gets called once, after the ngOnChanges hook.
  It initializes the component and sets the input properties of the component.
- **ngDoCheck( )** It gets called after ngOnChanges and **ngOnInit** and is used to detect and act on changes that cannot be detected by Angular.
  We can implement our change detection algorithm in this
  hook. **ngAfterContentInit( )** It gets called after the first **ngDoCheck** hook.
  This hook responds after the content gets projected inside the component.
- **ngAfterContentChecked( )** It gets called after **ngAfterContentInit** and every subsequent **ngDoCheck**. It responds after the projected content is checked.
- **ngAfterViewInit( )** It responds after a component's view, or a child component's view is initialized.
- **ngAfterViewChecked( )** It gets called after **ngAfterViewInit**, and it responds after the component's view, or the child component's view is checked.
- **ngOnDestroy( )** It gets called just before Angular destroys the component.
  This hook can be used to clean up the code and detach event handlers.

Let’s understand how to use **ngOnInit** hook, since it’s the most often used hook. If one has to process a lot of data during component creation, it’s better to do it inside **ngOnInit** hook rather than the constructor:

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

    @Component({
      selector: 'app-test',
      templateUrl: './test.component.html',
      styleUrls: ['./test.component.css']
    })
    export class TestComponent implements OnInit {
      constructor() { }

      ngOnInit() {
        this.processData();
      }
```

```
    processData(){
      // Do something..
    }

  }
```

As you can see we have imported OnInit but we have used **ngOnInit** function. This principle should be used with the rest of the hooks as well.

35. What are router links?

RouterLink is an anchor tag directive that gives the router authority over those elements. Because the navigation routes are set.

Example: As seen below, you may pass string values to the router-link directive.

```
<h1>Example of an Angular Router</h1>
<nav>
  <a routerLink="/home" >Home Page of our website</a>
  <a routerLink="/about-us" >About us</a>
</nav>
<router-outlet></router-outlet>
```

36. What exactly is the router state?

RouterState is a route tree. This tree's nodes are aware of the "consumed" URL segments, retrieved arguments, and processed data. You may use the Router service and the routerState property to get the current RouterState from anywhere in the application.

Example:

```
@Component({templateUrl:'example.html'})
class MyComponent {
  constructor(router: Router) {
    const state: RouterState = router.routerState;
    const root: ActivatedRoute = state.root;
    const child = root.firstChild;
    const id: Observable<string> = child.params.map(p => p.id);
    //...
  }
}
```

37. What does Angular Material means?

Angular Material is a user interface component package that enables professionals to create a uniform, appealing, and fully functioning websites, web pages, and web apps. It does this by adhering to contemporary web design concepts such as gentle degradation and browser probability.

38. What is ngOnInit?

ngOnInit is a lifecycle hook and callback function used by Angular to mark the creation of a component. It accepts no arguments and returns a void type.

Example:

```
export class MyComponent implements OnInit {
constructor() { }
   ngOnInit(): void {
      //....
   }
}
```

39. What is transpiling in Angular ?

Transpiling is the process of transforming the source code of one programming language into the source code of another. Typically, in Angular, this implies translating TypeScript to JavaScript. TypeScript (or another language like as Dart) can be used to develop code for your Angular application, which is subsequently transpiled to JavaScript. This occurs naturally and internally.

40. What are HTTP interceptors ?

Using the HttpClient, interceptors allow us to intercept incoming and outgoing HTTP requests. They are capable of handling both HttpRequest and HttpResponse. We can edit or update the value of the request by intercepting the HTTP request, and we can perform some specified actions on a specific status code or message by intercepting the answer.

Example: In the following example we will set the Authorization header Bearer for all the requests:

```
token.interceptor.ts
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class TokenInterceptor implements HttpInterceptor {
   public intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
   const token = localStorage.getItem('token') as string;
      if (token) {
      req = req.clone({
         setHeaders: {
         'Authorization': `Bearer ${token}`
         }
      });
      }
      return next.handle(req);
   }
}
```

We have to register the interceptor as singleton in the module providers

```
app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { AppComponent } from './app.component';
import { TokenInterceptor } from './token.interceptor';

@NgModule({
imports: [
   BrowserModule
],
declarations: [
   AppComponent
],
bootstrap: [AppComponent],
providers: [{
   provide: HTTP_INTERCEPTORS,
   useClass: TokenInterceptor,
   multi: true
}]
})
export class AppModule {}
```

41. What is AOT compilation? What are the advantages of AOT?

Every Angular application consists of components and templates that the browser cannot understand. Therefore, all the Angular applications need to be compiled first before running inside the browser.

Angular provides two types of compilation:

- JIT(Just-in-Time) compilation
- AOT(Ahead-of-Time) compilation

In JIT compilation, the application compiles inside the browser during runtime. Whereas in the AOT compilation, the application compiles during the build time.

The advantages of using AOT compilation are:

- Since the application compiles before running inside the browser, the browser loads the executable code and renders the application immediately, which leads to **faster rendering**.
- In AOT compilation, the compiler sends the external HTML and CSS files along with the application, eliminating separate AJAX requests for those source files, which leads to **fewer ajax requests**.
- Developers can detect and handle errors during the building phase, which helps in **minimizing errors**.

- The AOT compiler adds HTML and templates into the JS files before they run inside the browser. Due to this, there are no extra HTML files to be read, which provide **better security** to the application.

By default, angular builds and serves the application using JIT compiler:

```
ng build
ng serve
```

For using AOT compiler following changes should be made:

```
ng build --aot
ng serve --aot
```

42. What is Change Detection, and how does the Change Detection Mechanism work?

The process of synchronizing a model with a view is known as Change Detection. Even when utilizing the ng Model to implement two-way binding, which is syntactic sugar on top of a unidirectional flow. Change detection is incredibly fast, but as an app's complexity and the number of components increase, change detection will have to do more and more work.

Change Detection Mechanism-moves only ahead and never backward, beginning with the root component and ending with the last component. This is what one-way data flow entails. The tree of components is the architecture of an Angular application. Each component is a child, but the child is not a parent. A $digest loop is no longer required with the one-way flow.

43. What is a bootstrapping module?

Every application contains at least one Angular module, which is referred to as the bootstrapping module. AppModule is the most popular name for it.

Example: The following is the default structure of an AngularCLI-generated AppModule:

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

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
```

```
  HttpClientModule
 ],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

44. Explain MVVM architecture

MVVM architecture consists of three parts:

1. Model
2. View
3. ViewModel

- **Model** contains the structure of an entity. In simple terms it contains data of an object.
- **View** is the visual layer of the application. It displays the data contained inside the Model. In angular terms, this will be the HTML template of a component.
- **ViewModel** is an abstract layer of the application. A viewmodel handles the logic of the application. It manages the data of a model and displays it in the view.

View and ViewModel are connected with data-binding (two-way data-binding in this case). Any change in the view, the viewmodel takes a note and changes the appropriate data inside the model.

Angular Scenario Based Interview Questions

45. How do you choose an element from a component template?

To directly access items in the view, use the @ViewChild directive. Consider an input item with a reference.

```
<input #example>
```

and construct a view child directive that is accessed in the ngAfterViewInit lifecycle hook

```
@ViewChild('example') input;

ngAfterViewInit() {
 console.log(this.input.nativeElement.value);
}
```

46. How does one share data between components in Angular?

Following are the commonly used methods by which one can pass data between components in angular:

- **Parent to child using @Input decorator**

Consider the following parent component:

```
@Component({
  selector: 'app-parent',
  template: `
    <app-child [data]=data></app-child>
  ` ,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent{
  data:string = "Message from parent";
  constructor() { }
}
```

In the above parent component, we are passing “data” property to the following child component:

```
import { Component, Input} from '@angular/core';
@Component({
  selector: 'app-child',
  template:`
    <p>{{data}}</p>
  `,
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  @Input() data:string
  constructor() { }
}
```

In the child component, we are using @Input decorator to capture data coming from a parent component and using it inside the child component’s template.

- **Child to parent using @ViewChild decorator**

Child component:

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-child',
  template:`
    <p>{{data}}</p>
  `,
```

```
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  data:string = "Message from child to parent";
  constructor() { }
}
```

Parent Component

```
import { Component,ViewChild, AfterViewInit} from '@angular/core';
import { ChildComponent } from './../child/child.component';
@Component({
  selector: 'app-parent',
  template: `
    <p>{{dataFromChild}}</p>
  `,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent implements AfterViewInit {
  dataFromChild: string;
  @ViewChild(ChildComponent,{static:false}) child;
  ngAfterViewInit(){
    this.dataFromChild = this.child.data;
  }
  constructor() { }
}
```

In the above example, a property named “data” is passed from the child component to the parent component.

@ViewChild decorator is used to reference the child component as “child” property.

Using the **ngAfterViewInit** hook, we assign the child’s data property to the messageFromChild property and use it in the parent component’s template.

- **Child to parent using @Output and EventEmitter**

In this method, we bind a DOM element inside the child component, to an event (**click** event for example) and using this event we emit data that will captured by the parent component:

Child Component:

```
import {Component, Output, EventEmitter} from '@angular/core';
@Component({
  selector: 'app-child',
  template:`
    <button (click)="emitData()">Click to emit data</button>
  `,
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  data:string = "Message from child to parent";
  @Output() dataEvent = new EventEmitter<string>();
```

```
    constructor() { }
    emitData(){
      this.dataEvent.emit(this.data);
    }
  }
```

As you can see in the child component, we have used **@Output** property to bind an **EventEmitter**. This event emitter emits data when the button in the template is clicked.

In the parent component's template we can capture the emitted data like this:

```
<app-child (dataEvent)="receiveData($event)"></app-child>
```

Then inside the receiveData function we can handle the emitted data:

```
receiveData($event){
    this.dataFromChild = $event;
  }
```

47. How do you deal with errors in observables?

Instead of depending on try/catch, which is useless in an asynchronous context, you may manage problems by setting an error callback on the observer.

Example: You may create an error callback as shown below.

```
myObservable.subscribe({
  next(number) { console.log('Next number: ' + number)},
  error(err) { console.log('An error received ' + err)}
});
```

48. How can I include SASS into an Angular project?

You may use the ng new command while building your project using angular CLI. All of your components are generated using preset sass files.

```
ng new <Your_Project_Name> --style = sass
```

If you want to change the style of your project, use the ng set command.

```
ng set defaults.styleExt scss
```

49. What happens when you use the script tag within a template?

Angular detects the value as unsafe and sanitizes it automatically, removing the script tag but retaining safe material such as the script tag's text content. This reduces the potential of script injection attacks. If you continue to use it, it will be disregarded, and a warning will display in the browser console.

Example: Consider the case of innerHtml property binding, which results in an XSS vulnerability.

```
export class InnerHtmlBindingComponent {
   // For example, a attacker-controlled value from a URL using malicious scripts.
   htmlSnippet = 'Template <script>alert("You are hacked !!!!")</script> <b>Syntax</b>';
}
```

Angular MCQ questions

1.

Which of the following syntaxes is appropriate for AngularJS when applying multiple filters?

- {{ expression - {filter1} - {filter2} - ... }
- {{ expression | filter1 | filter2 | ... }}}
- {{ expression | {filter1} | {filter2} | ... }}
- {{ {filter1} | {filter2} | ...-expression}}

2.

In AngularJS, a module is created using which of the following syntax?

- var myModule= new Module();
- var myModule= angular.module();
- module("app", []);
- None of the above

3.

Which of the following is true about $routeProvider?

- It is a component.
- It is a service.
- It is a module.
- None of the above

4.

Which of the following is an acceptable AngularJS expression?

- {(7+5)}
- {7+5}
- {{7+5}}
- None of the above

5.

What is the purpose of the @Output decorator?

- Share data from a child component to a parent
- Use event binding
- Share data from a parent component to a child
- Use a service

6.

What is Form Builder?

- Class for generating template-driven forms in run-time
- Function for generating template-driven forms in run-time
- Function for generating reactive forms in run-time
- Class for generating reactive forms in run-time

7.

Which architectures does angular use?

- Components Based Architecture
- MVVM
- MVC
- All of the above

8.

Which type of binding uses the banana box [()]

- Interpolation Binding
- Property Binding
- Two way Binding
- Directive Binding

9.

AngularJS is perfect for

- Create Web Services
- Create Single Page Applications
- Creating a Desktop Application
- None of these

10.

Which prefix is used with the AngularJS directives?

- aj-

- ng-
- ag-
- All of the above