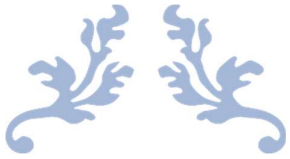

ANGULAR INTERVIEW QUESTIONS

1. What is Angular?

Ans: - Angular is a typescript-based framework, which is used to create a client-side web application.

2. What is a Framework?

Ans: - A framework is like a platform for developing software applications. A framework can have predefined classes and functions that can be re-used to add several functionalities, which otherwise we would have to write from scratch by our own.

3. What are the advantages of Angular?

Ans: -

1. **Custom Component** – Angular enable users to build their own components that can pack functionality along with render logic into reusable pieces.
2. **Data Binding** – Angular enable users to effortlessly move data from JavaScript code to the view, and react to user events without having to write code manually.
3. **Dependency Injection** – Angular enable users to write modular services and inject them wherever they are needed.
4. **Comprehensive** – Angular is a full-fledged framework and provides out-of-the-box solutions for server communication, routing within your application and more.
5. **Browser compatibility** – Angular is cross-platform and compatible with multiple browsers.

4. What is the difference between AngularJS and Angular?

Ans: -

AngularJS	Angular
AngularJS is based on MVC architecture.	Angular is based on Service/Controller architecture.
Uses JavaScript to build application.	Uses Typescript to build application.
Based on controller concept.	Based on Component-Based UI Approach.
No support for mobile platforms.	Fully Supports mobile platforms.
Difficult to build SEO friendly applications.	Easier to build SEO friendly applications.

5. What is npm?

Ans: - npm stands for “node package manager”. It is a primary package manager for JavaScript runtime environment node.js. It utilized for managing packages and dependencies for various JavaScript framework and libraries.

6. What is CLI tool?

Ans: - Angular CLI (Command Line Interface) is a tool that scaffold and build Angular apps using NodeJS style (CommonJS) modules.

Some basic list of commands –

- **To create new project** – ng new project-name
- **To create class** – ng new class class-name
- **To create component** – ng generate component component-name
- **To create directive** – ng generate directive directive-name
- **To create enum** – ng generate enum enum-name
- **To create module** – ng generate module module-name
- **To create pipe** – ng generate pipe pipe-name
- **To create service** – ng generate service service-name
- **To create lazy loaded module** – ng generate module module-name - -route module-name - -module insert-module-name.module
- **To create component in specific module** – ng generate component component-name - -module module-name
- **To Run Project** – ng serve or ng server –open (ng s -o)

7. What is a typescript? What are the advantages over JavaScript?

Ans: -Typescript is a strongly typed superset of JavaScript created by Microsoft that adds optional types, classes, async/await and many other features, and compiles to plain JavaScript. Angular is written entirely in Typescript as a primary language. Here are some advantages of TypeScript over JavaScript:

1. **Static Typing:** TypeScript allows developers to specify types for variables, function parameters, return types, and more. This enables catching type-related errors during development rather than at runtime, leading to more robust code.
2. **Enhanced Tooling Support:** TypeScript provides better tooling support, including features like code navigation, auto-completion, and refactoring tools, thanks to the availability of type information.
3. **Improved Code Quality:** With static typing, developers can write more self-documenting code, making it easier for other developers to understand and maintain the codebase. Additionally, TypeScript can help catch common errors early in the development process, leading to higher code quality.
4. **Modern JavaScript Features:** TypeScript supports features from the latest versions of JavaScript, allowing developers to use modern language features while maintaining compatibility with older browsers through transpilation.
5. **Ecosystem and Community:** TypeScript has a growing ecosystem and community support. Many popular libraries and frameworks have TypeScript type definitions available, making it easier to integrate TypeScript into existing projects.
6. **Easy Adoption:** Since TypeScript is a superset of JavaScript, existing JavaScript code can be gradually migrated to TypeScript. Developers can start by adding type annotations to existing code and gradually refactor it to take advantage of TypeScript's features.

8. Where to store static files in Angular projects?

Ans: - In Angular projects, static files such as images, fonts, stylesheets, and other assets are typically stored in the src/assets directory.

9. What is the role of Angular.json file in Angular?

Ans: - It is used to define the configuration and settings for an Angular workspace, including project-specific settings, build options, and dependencies.

10.What is the difference between JIT and AOT in Angular?

Ans: - Angular offers two ways to compile your Angular application,

1. Just–In–Time (JIT)
2. Ahead-Of-Time (AOT)

JIT (Just in Time)	AOT (Ahead of Time)
Compile your app in the browser at runtime.	Compile your app at build time.
Slow rendering as the browser compiles the code runtime.	Fast rendering as browser gets the pre-compiled code.
Template errors will only be detected during runtime.	Template errors are detected earlier during application building process.
Injection attacks can occur since templates are compiled in the browser.	Injection attacks are less likely to happen since templates are compiled before runtime.

11.What are the components in Angular?

Ans: - Components are the most basic UI building block of an Angular app, which form a tree of Angular components.

Let's See example of Angular component

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

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

In the above code, a class is decorated with @Component decorator. The @Component decorator in Angular is used to define and configure a component. It allows you to specify metadata for the component, including its selector, template or templateUrl, and styles or styleUrls.

Here's a breakdown of the properties of the @Component decorator:

- **selector**: Specifies the CSS selector that identifies the component in a template. When Angular encounters this selector in a template, it replaces it with the component's HTML template. In your example, the selector is 'app-root', which means you can use <app-root></app-root> in your HTML to render this component.
- **templateUrl / template**: Defines the HTML template for the component. You can either provide the template directly using the template property, or specify the path to an external HTML file using the templateUrl property. The component's HTML content will be rendered inside the element identified by the selector.
- **styleUrls / styles:** Specifies the CSS stylesheets for the component. You can either provide styles directly using the styles property, or specify an array of paths to external CSS files using the styleUrls property. The styles defined here will be applied only to the component and its children, encapsulating the styles to prevent global conflicts.

12.What is modules in Angular? What is app.module.ts file?

Ans: - Modules are logical bounderies in your application and the application is divided into separate module to separate the functionality of your application.

Let's take an example of app.module.ts root module declared with @NgModule decorator as below,

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

The NgModule decorator has five important (among all) options:

1) **declarations:** It is used to define components in the respective module.
2) **imports:** It is used to import other dependent modules.
3) **providers:** It is used to configure a set of injectable objects that are available in the injector of this module.
4) **bootstrap:** It tells Angular which component to bootstrap in the application.
5) **entryComponents:** It is a set of components dynamically loaded into the view.

13.How an Angular app get loaded and started? What are index.html, app-root, selector and main.ts file?

Ans: - When you launch an Angular application, the following steps outline how the app gets loaded and started:

Browser Loads index.html: The Angular application starts by loading the index.html file in the browser. This HTML file serves as the entry point for the application.

1) **index.html:** This is the main HTML file of your Angular application. It typically contains the basic structure of an HTML document, including <head> and <body> sections. In an Angular application, index.html acts as the host for the Angular application and includes a placeholder element where the Angular application will be rendered.
2) **app-root:** In the index.html file, there is typically a placeholder element with the selector app-root. This element serves as the mounting point for the Angular application. When Angular bootstraps, it searches for this selector in the index.html file and replaces it with the root component of the Angular application.
3) **selector:** The selector is a property defined in Angular components using the @Component decorator. It is a CSS selector that identifies the component in a template. When Angular encounters this selector in a template, it replaces it with the component's HTML template. In the context of app-root, it is the selector used to identify the root component of the Angular application in the index.html file.
4) **main.ts file:** This is the main entry file for an Angular application. It is typically located in the src directory. In the main.ts file, the Angular application is bootstrapped using the platformBrowserDynamic().bootstrapModule() function, passing in the root module of the application (AppModule by default). This function initializes the Angular application and starts the application execution process.

14.What is a bootstrap module and bootstrap component?

Ans: - In Angular, the terms "bootstrap module" and "bootstrap component" are key concepts related to bootstrapping an Angular application.

- **Bootstrap Module:** The bootstrap module is the entry point of an Angular application. It is responsible for starting the application and loading the root component. In Angular, the AppModule is typically used as the bootstrap module. The bootstrap module is defined using the @NgModule decorator, where you specify metadata such as declarations, imports, providers, and bootstrap components. The @NgModule decorator includes a bootstrap property, which specifies the root component(s) that should be bootstrapped when the application starts. Angular bootstraps the bootstrap module to initiate the application.
- **Bootstrap Component:** The bootstrap component is the root component of an Angular application. It is the component that is loaded and rendered when the application starts. The bootstrap component is specified in the bootstrap property of the bootstrap module's @NgModule decorator. Angular searches for the selector of the bootstrap component in the index.html file and replaces it with the component's template during the bootstrapping process. The bootstrap component typically

contains the main structure and layout of the application and serves as the starting point for the UI hierarchy. Other components are typically nested within the bootstrap component to build the application's UI.

15.What is data binding in Angular?

Ans: - Data binding in Angular allow us to communicate between a component class and its corresponding view template and vice versa.

16.What is string interpolation in Angular?

Ans: - Interpolation is a special syntax that Angular converts into property binding. It is a convenient alternative to property binding. It is represented by double curly braces.

Let's take an example,

```
<h1>{{show()}}</h1>
```

In the example above, Angular evaluates the show function and fill the return text in h1 element.

17.What is property binding in Angular?

Ans: - Property binding let us bind a property of a DOM object, for example the hidden property of some data value. This can let us show or hide a DOM element or manipulate the DOM in some other way.

Let's take an example,

```
<input type="text" placeholder = "Enter your name" [value]="data">
```

In the above example, value attribute of input element is bind with data property of respected component class.

18.What is event binding in Angular?

Ans: - Event binding in Angular is a mechanism that allows you to listen for and respond to user interactions such as mouse clicks, keyboard inputs, and other DOM events within your Angular templates. It enables you to execute custom code or trigger component methods in response to user actions.

For example,

```
<button [disabled]="allowUser" (click)="change()">Disable Me!</button>
```

In the above example, the change function is getting called when user click on the button. That change function change the property 'allowUser' value and button get disabled as disabled property of button is bind with it.

19.What is two-way binding in Angular?

Ans: - Two-way binding binds data from component class to view template and view template to component class. It is a combination of property binding and event binding.

```
<input type="range" min="0" max="255" [(ngModel)]="red_Value">
```

In the above example, two-way binding is done with the help of ngModel class of Forms module.

20.What are directives? What are the types of directives?

Ans: - Directive is an instruction to the DOM. We use directive to manipulate the DOM, change its behaviour and Add / Remove the DOM elements.

There are mainly three types of directives.

1) **Component directive: -** Component directive is the angular component. It is a directive with template.
2) **Attribute directive: -** It is used to change the behaviour of a DOM element. E.g. built in attribute directive – ngStyle, ngClass.
3) **Structural directive: -** It is used to add or remove DOM element on the webpage. E.g. *ngIf , *ngFor, *ngSwitch.

21.What is *ngIf structural directive?

Ans: - Sometimes an app needs to display a view or a portion of a view only under specific circumstances. The Angular *ngIf directive inserts or removes an element based on a truthy/falsy condition. Let's take an example,

```
<h3 *ngIf="condition"> condition is true ..... </h3>
```

In the above example, this <h3> element will display only if condition is true.

22.What is *ngFor structural directive?

Ans: - We use Angular *ngFor directive template to display each item in the list. For example, here we can iterate over a list of cities:

```
<h1>List of Cities</h1>
<ul *ngFor="let i of city">
  <li>{{i}}</li>
</ul>
```

23.What is ng*switch structural directive?

Ans: - The ngSwitch structural directive in Angular is used to conditionally display one element from a set of elements based on the value of an expression. It is similar to the switch statement in JavaScript and allows you to choose between multiple options and render different content based on those options.

```
<select  [(ngModel)]="choice">
  <option *ngFor="let i of items" [value]="i.val">{{i.val}}
{{i.name}}</option>
</select>

<div [ngSwitch]="choice">
  <h1 *ngSwitchCase="1">One is Selected</h1>
  <h1 *ngSwitchCase="2">Two is Selected</h1>
```

```
  <h1 *ngSwitchDefault>Invalid input</h1>
</div>
```

24.What is [ngStyle] attribute directive?

Ans: - The [ngStyle] attribute directive in Angular allows you to dynamically set inline styles for HTML elements based on expressions in your component class. It provides a way to apply CSS styles conditionally or based on data in your application.

```
<div [ngStyle]="{ 'font-size.px': fontSize, 'color': fontColor, 'background-color': backgroundColor }">
  Dynamic Styled Text
</div>
```

25.What is [ngClass] attribute directive?

Ans: - The [ngClass] attribute directive in Angular allows you to dynamically apply CSS classes to HTML elements based on expressions in your component class. It provides a way to conditionally apply or remove CSS classes based on certain conditions or data in your application.

```
<div [ngClass]="classList">
  My Name is Anthony Gonzalves.
</div>
```

26.What is the difference between component, attribute and structural directive?

Ans: -

Directive Type	Component Directive	Attribute Directive	Structural Directive
Purpose	Create reusable UI components	Modify behaviour or appearance of elements	Modify structure of DOM.
Syntax	Defined using `@Component` decorator	Applied as attribute of HTML elements	Applied as attribute of HTML elements with `*` prefix
Behaviour	Encapsulates both UI and logic	Adds, removes, or modifies element attributes	Adds or removes elements from DOM based on conditions
Examples	ButtonComponent, HeaderComponent	ngClass, ngStyle, ngModel	ngIf, ngFor, ngSwitch

27.What is decorator?

Ans: - Decorator is used to provide metadata to the class. So, that it can configure the expected behaviour of the class.

28.What are the types of decorators?

Ans: - There are mainly four types of decorators.

1) **Class decorators: -** e.g. @Component and @NgModule

```
@Component({
  selector: 'app-class-binding',
  templateUrl: './class-binding.component.html',
  styleUrls: ['./class-binding.component.css']
})
export class ClassBindingComponent {

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

2) **Property decorators: -** Used for property inside the class. E.g. @Input and @Output

```
  @Input() mobNum!: number;
  @Output() myName = new EventEmitter<string>();
  emitName(): void {
  this.myName.emit(this.userName);
  }
```

3) **Method decorators: -** Used for method inside classes. E.g. @HostListener

```
@Directive({
  selector: '[Highlight]'
})
export class HighlightDirective {

  constructor(private element: ElementRef, private render: Renderer2) { }

  //the @HostListener decorator is used to listens the DOM events of the
HostElement
  // and react to that events by executing the event handler method.
  @HostListener('mouseover') onMouseEnter() {
    this.render.addClass(this.element.nativeElement,'highlightContainer'); //
adding class to the DOM element
  }
```

4) **Parameter decorators: -** Used for parameter inside class constructor. E.g. @Inject, @Optional

```
import { Component, Inject } from '@angular/core';
import { MyService } from './my-service';

@Component({
    selector: 'my-component',
    template: '<div>Parameter decorator</div>'
})
export class MyComponent {
    constructor(@Inject(MyService) myService) {
        console.log(myService); // MyService
    }
}
```

29.What are pipes? What are types of pipes and parameterized pipes?

Ans: - Pipes in Angular are used to transform or format the data before displaying it in the view. Angular pipe takes data as an input and it formats or transform that data before displaying it in the template. A Pipe can accept any number of parameters to fine tune its output. The Parameterized pipe can be created by declaring the pipe name with a colon (:) and then the parameter value. If pipe takes multiple parameters, separate the value with colons.

Let's take some examples,

1. **Built In pipes**
   1.1. **Upper Case Pipe: -** `<h1>{{username | uppercase }} in Uppercase using pipe</h1>`
   1.2. **Lower Case Pipe: -** `<h1>{{username}} in Lowercase using pipe</h1>`
   1.3. **Decimal Pipe: -** `<h1>{{ 123.456 | number:'4.1-2'}}</h1>` (Output: 0,123.46)
   1.4. **Percent Pipe: -** `<h1>{{0.54 | percent}}</h1>` (Output:54%)
   1.5. **Currency Pipe: -** `<h1>{{a | currency:'INR'}}</h1>`
   1.6. **Date Pipe: -** `<h1>default: {{today | date}}</h1>`
   1.7. **Slice Pipe: -** `<h2>{{months | slice:6 :7 }}</h2>`

   Here, months is an array => `months: string[] = ["Jan", "Feb", "Mar", "Apr","May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];`

   So, output will be => Jul

   1.8. **JSON Pipe: -** <h1>{{ object | json }}</h1>
   1.9. **Async Pipe: -** `<p>Time: {{time | async |date:'fullTime'}}</p>`
2. **Custom Pipes**

   ```
   <h1>{{10 | multiplier:20}}</h1>

   import { Pipe, PipeTransform } from '@angular/core';

   @Pipe({
   ```

```
  name: 'multiplier'
})
export class MultiplierPipe implements PipeTransform {
  transform(value:number,first:number): number {
    return value * first;
  }
}
```

In the above example, the custom pipe MultiplierPipe has been created. It takes one argument, which represents the multiplier, and returns the result of multiplying it with the input value provided in the template.

30.What is chaining in pipes?

Ans: - In Angular, chaining in pipes refers to the practice of applying multiple pipes consecutively within a template expression. This allows you to perform a sequence of transformations on a single piece of data before displaying it to the user.

For example,

```
<h3> {{"Welcome" |reverse | uppercase}} </h3>
```

In the above example, we have used two pipes, the first pipe is a custom pipe which transform the string “Welcome” to “emocleW” and then Built in upper case pipe transform it into “EMOCLEW“.

31.Explain services with example?

Ans: - A service is used when a common functionality needs to be provided to various modules. Service allow for greater separation of concerns for your application and better modularity by allowing you to extract common functionality out of components.

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

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

  constructor(private http: HttpClient) { }
  fetchUsers(): any {
    return this.http.get("https://jsonplaceholder.typicode.com/posts");
  }
}
```

In the provided example, a service named MyServiceService is created to fetch user data from a fake API using the Angular HttpClient module. To declare a class as a service in Angular, we use the @Injectable decorator. This decorator allows the service to be injected into other components or services. The providedIn: 'root' option in the @Injectable decorator ensures

that the service is registered at the root level injector, making it available throughout the application.

32.How to create service in Angular?

Ans: - In angular we can create service in two ways,

1) Using angular CLI
2) Manually

Using Angular CLI

- Open terminal and write " ng generate service serviceName"
- This command will create a service file and set it defaults provider to root.

Manually

- Create a new TypeScript file in your Angular project's src/app directory.
- Inside the file, you can define your service class with the desired functionality.
- Import any necessary Angular dependencies, such as Injectable from @angular/core.
- Decorate the class with @Injectable() to make it an Angular service.
- Optionally, you can specify the providedIn property in the @Injectable() decorator to set the service's provider scope. For example, setting it to 'root' makes the service available throughout the application.
- Define methods and properties within the service class to implement its functionality.
- Save the file with an appropriate name, typically ending with .service.ts, such as my-service.service.ts.

33.How to use dependency injection with service in Angular?

Ans: - Dependency Injection (DI), is an important application design pattern in which a class ask for dependencies from external resources rather than creating them itself. To use external dependencies in service class,

1) Create a service class.
2) Import the service class in your component, directive or service where you want to use.
3) Create an instance of that service.
4) Use the service with that instance.

34.What is hierarchical dependency injection?

Ans: - Hierarchical Dependency Injection (DI) in Angular refers to the way Angular's dependency injection system manages the creation and sharing of instances of services across different levels of the application's component tree.

Angular has two injector hierarchies:

1) Module Injector Hierarchy

2) Element Injector Hierarchy

Module Injector Hierarchy: - The Module Injector Hierarchy refers to the organization of injectors based on Angular modules. Each Angular module has its own injector, known as the module injector. When an Angular application is bootstrapped, Angular creates a root module injector for the root module (AppModule by default). Each module injector encapsulates the services provided by its respective module. Services provided at the module level are available throughout the module and its child modules. If a service is provided in multiple modules, Angular creates separate instances of the service for each module injector.

Element Injector Hierarchy: - The Element Injector Hierarchy refers to the organization of injectors based on the component tree. Each Angular component has its own injector, known as the element injector. When a component is created, Angular creates an element injector specifically for that component. The element injector encapsulates the services provided by the component's module injector and its ancestors' module injectors. Services provided at the module level or the component level are available within the component and its child components. If a service is provided in multiple component injectors, Angular creates separate instances of the service for each component injector.

35.What is provider in Angular?

Ans: - In Angular, a provider is a configuration that tells Angular's dependency injection system how to obtain or create instances of a service or value. Providers are used to register services, values, or other dependencies within Angular's dependency injection system, making them available for injection into components, directives, pipes, etc.

36.What is the role of @Injectable decorator in services?

Ans: - The @Injectable() decorator in Angular is used to mark a class as a candidate for Angular's dependency injection system. When a class is decorated with @Injectable(), Angular's DI system knows that it may need to inject instances of this class into other components, services, directives, or pipes.

37.What are parent child component?

Ans: - In Angular, parent and child components refer to the relationship between components within the component tree hierarchy.

Parent Component: A parent component is a component that contains other components within its template. It is typically higher in the component tree hierarchy. Parent components can pass data to their child components using input properties. Child components are typically included within the template of the parent component using selector tags.

Child Component: A child component is a component that is contained within another component's template. It is typically lower in the component tree hierarchy. Child components receive data from their parent components through input properties. Child components can also emit events to communicate with their parent components using output properties.

38.What is @Input decorator? How to transfer data from parent component to child component?

Ans: - In Angular, the @Input decorator is used to pass data from a parent component to a child component. It allows the parent component to bind data to properties of the child component.

To pass data from parent component to child component, define a property in child component class and decorate it with @Input decorator. Then Bind the property of the child component to a value in the parent component's template.

For example,

Child Component –

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  @Input() inputData: any;
}
```

Parent Component –

```
<app-child [inputData]="parentData"></app-child>
```

39.What is @Output decorator and event emitters?

Ans: - In Angular, the `@Output` decorator is used along with `EventEmitter` to allow a child component to communicate with its parent component by emitting custom events.

To pass data from child component to parent component, define the output event in child component and then in parent component bind this event to the function and handle the data.

Child Component -

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  @Output() outputEvent = new EventEmitter<any>();
```

```
  sendDataToParent() {
    this.outputEvent.emit(dataToSend);
  }
}
```

Parent Component –

```
<app-child (outputEvent)="handleChildEvent($event)"></app-child>

export class ParentComponent{
  handleChildEvent(data: any) {
    // Handle the emitted data from the child component
  }
}
```

40.What are lifecycle hooks in Angular?

Ans: - The Angular lifecycle hooks are the methods that angular invokes on a directive or a component as it creates, changes and destroy them.

41.What is constructor in Angular?

Ans: - Constructor is a special type of method that is defined within a class and is automatically invoked when an instance of that class is created. It is primarily used for initialization purposes, such as setting up the initial state of the class, initializing properties, and performing other setup tasks.

42.What is ngOnChanges lifecycle hook in Angular?

Ans: - The ngOnChanges is a lifecycle hook in Angular which get invoke when a new component is created or its input bound properties are updated.

43.What is ngOnInit life cycle hook in Angular?

Ans: - Angular raises ngOnInit hook after it creates the component and update its input properties. This hook is raised after ngOnChanges.

The ngOnInit hook is fired only once. i.e during the first change detection cycle. By the time ngOnInit gets called, none of the child components or projected contents or view are available at this point. Hence any property decorated with @ViewChild, @ViewChildren, @ContentChild or @ContentChildren will not available to use.

44.What is difference between constructor and ngOnInit?

Ans: - The Constructor is a default method of the class that is executed when the class is instantiated and ensures proper initialisation of fields in the class and its subclasses. Angular, or better Dependency Injector (DI), analyses the constructor parameters and when it creates a new instance by calling new MyClass() it tries to find providers that match the types of the constructor parameters, resolves them and passes them to the constructor.

ngOnInit is a life cycle hook called by Angular to indicate that Angular is done creating the component.

Mostly we use ngOnInit for all the initialization/declaration and avoid stuff to work in the constructor. The constructor should only be used to initialize class members but shouldn't do actual "work".

So you should use constructor() to setup Dependency Injection and not much else. ngOnInit() is better place to "start" - it's where/when components' bindings are resolved.

45.What is routing? How to setup routing?

Ans: - Routing allows us to navigate from one part of our application to another part. In Angular, using routing we can move from view of one component to view of another component.

To implement routing in Angular, we use a built-in @angular/ router module. Here's a step-by-step guide on how to set up routing in an Angular application:

1) Define the Routes array in app-routing.module.ts file
2) Add the path and component in the Routes array.
3) In the @NgModule decorator of the AppRoutingModule class
    a. In the imports array call the RouterModule.forRoot method and pass the Routes array
    b. In the export array pass the RouterModule

```
import {NgModule} from '@angular/core';
import {Routes, RouterModule} from '@angular/router';
import {HomeComponent} from './home.component';
import {AboutComponent} from './about.component';

const routes: Routes = [
  {path: '', component: HomeComponent },
  {path: 'about', component: AboutComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

4) Add Router Outlet in the Component where you want to render the routed components.
```
<router-outlet></router-outlet>
```
5) set up navigation link
```
<a routerLink="/">Home</a>
<a routerLink="/about">About</a>
```

46.What is router outlet?

Ans: - Router outlet is a directive from the router library and it acts as a placeholder that marks the spot in the template where the router should display the components for that outlet. Router outlet is used like a component,

```
<router-outlet></router-outlet>
```

47.What are router links?

Ans: - The routerLink is a directive on the anchor tags give the router control over those elements. Since the navigation paths are fixed, you can assign string values to router-link directive as below,

```
<nav>
<a routerLink="/todosList" >List of todos</a>
<a routerLink="/completed" >Completed todos</a>
</nav>
<router-outlet></router-outlet>
```

48.What are asynchronous operation?

Ans: - Asynchronous operations are tasks or actions in a program that don't necessarily happen in sequential order or in sync with the rest of the program's execution. Instead, they occur independently of the main program flow, allowing the program to continue executing other tasks while waiting for the asynchronous operation to complete. Common example of asynchronous operations are:

1) Fetching data from server
2) File I/O operations
3) Timers and delays

49.What is the difference between promise and observable?

Ans: -

Feature	Observable	Promise
Execution Behavior	Declarative: Computation does not start until subscription, so they can run whenever you need the result	Executes immediately on creation
Value Delivery	Provides multiple values over time	Provides only one
Error Handling	Subscribe method is used for error handling that facilitates centralized and predictable error handling	Push errors to the child promises
Handling Complex Applications	Provides chaining and subscription to handle complex applications	Uses only .then() clause

50.What is the role of HttpClient in Angular?

Ans: - Most of the Front-end applications communicate with backend services over HTTP protocol using either XMLHttpRequest interface or the fetch() API. Angular provides a

simplified client HTTP API known as HttpClient which is based on top of XMLHttpRequest interface. This client is available from @angular/common/http package.

51.What are the steps for fetching the data with HttpClient and Observables?

Ans: - Below are the steps for fetching data with HttpClient and Observables,

1) Import the HttpClient into root module.

```
import { HttpClientModule } from '@angular/common/http';
@NgModule({
imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
],
......
})
export class AppModule {}
```

2) Inject the HttpClient into the Application

   Let's create a userProfileService(userprofile.service.ts) as an example. It also defines get method of HttpClient:

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

const userProfileUrl: string = 'assets/data/profile.json';

@Injectable()
export class UserProfileService {
constructor(private http: HttpClient) { }

getUserProfile() {
    return this.http.get(this.userProfileUrl);
}
}
```

3) Create a component for subscribing service

   Let's create a component called UserProfileComponent(userprofile.component.ts), which injects UserProfileService and invokes the service method:

```
fetchUserProfile() {
  this.userProfileService.getUserProfile()
      .subscribe((data: User) => this.user = {
          id: data['userId'],
          name: data['firstName'],
          city:  data['city']
      });
```

```
}
```

52.How to do http error handling in Angular?

Ans: - If the request fails on the server or fails to reach the server due to network issues, then HttpClient will return an error object instead of a successful response. In this case, you need to handle in the component by passing error object as a second callback to subscribe() method.

Let's see how it can be handled in the component with an example,

```
fetchUser() {
  this.userService.getProfile()
      .subscribe(
        (data: User) => this.userProfile = { ...data }, // success path
        error => this.error = error // error path
      );
  }
```

```
It is always a good idea to give the user some meaningful feedback instead of
displaying the raw error object returned from HttpClient.
```

53.What is typescript?

Ans: - TypeScript is a strongly typed superset of JavaScript created by Microsoft that adds optional types, classes, async/await and many other features, and compiles to plain JavaScript. Angular is written entirely in TypeScript as a primary language.

54.What is difference between typescript and JavaScript?

Ans: - Typescript is a statically typed where as a JavaScript is a dynamically typed.

Typescript Required to be compile into JavaScript before execution, while JavaScript executes directly.

TypeScript has extended JavaScript with additional features, while JavaScript has core language features only.

55.How to install typescript and check version?

Ans: - To install typescript run this command in terminal,

```
npm install -g typescript
```

To check the version of typescript,

```
tsc -- version
```

56.What is difference between let and var?

Ans: - Var keyword is available from the beginning of the JavaScript, while let keyword is available from ES6 version of JavaScript. Var has function scope, while let has block scope. Variables will be hoisted and undefined if it is declared with Var keyword and Variables with let keyword will be hoisted but not initialize. So, it will throw an error if we try to access the variable with let keyword before initializing it. In case of var it will print undefined.

57.What is type annotation?

Ans: - Type annotation in TypeScript is a way to explicitly specify the type of a variable, parameter, property, or function return type. It allows developers to define the expected data type of a value, providing clarity and enabling static type checking by the TypeScript compiler.

58.What are built in or primitive and user defined or non-primitive types in typescript?

Ans: -

1) Built-In or Primitive data types: -
    a. String
    b. Number
    c. Boolean
    d. Bigint
    e. Symbol
    f. Null
    g. Undefined
2) User-defined or non-primitive data types: -
    a. Arrays
    b. Objects
    c. Enums

59.What is any type in typescript?

Ans: - any is a type that disables type checking and effectively allows all types to be used.

60.What is enum type in typescript?

Ans: - An enum (enumeration) is a user-defined data type that represents a collection of named constants. These constants are typically used to represent a set of related values that cannot change during program execution.

```
enum HttpStatus {
  Ok = "200",
  NotFound = "404",
  InternalServerError = "500"
}
let responseStatus: HttpStatus = HttpStatus.Ok;
console.log(responseStatus); // Output: "200"

enum Weekday {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday
```

```
}
let today: Weekday = Weekday.Tuesday;
console.log(today); // Output: 1 (assuming Tuesday is the second value)
```

61.What is the difference between void and never type in typescript?

Ans: - Both void and never are types in TypeScript used for functions that don't return a value in the traditional sense, but they represent different scenarios:

void:

- Represents a function that does not explicitly return a value.
- The function might have side effects (like modifying variables or performing actions) but doesn't intend to return a usable data point.
- Implicitly, a function with a void return type might still return undefined.

```
function printMessage(message: string): void {
  console.log(message);
}

printMessage("Hello, world!");  // This function doesn't return anything to be assigned to a variable.
```

never:

- Represents a function that never reaches a normal return statement.
- This typically happens when the function throws an error, enters an infinite loop, or encounters some condition that prevents it from continuing normally.
- A function with a never return type cannot return undefined either.

```
function throwError(message: string): never {
  throw new Error(message);
}

// This will cause an error because the function never returns after the throw statement.
let result = throwError("This function always throws an error");
```

62.What is unknown type in typescript?

Ans: - In TypeScript, the unknown type represents a type-safe counterpart to the any type. While any allows for any type of value to be assigned to a variable, unknown is more restrictive. When a value is of type unknown, TypeScript requires you to perform type checks or assertions before you can use it in a type-safe manner.

```
let x: unknown = "Hello world!";

console.log(x); // Hello world!
// console.log(x.length);  // not allowed as x is unknown type

console.log((x as string).toLowerCase()); // hello world!
```

63.What is type assertion in typescript?

Ans: - Type assertion in TypeScript is a way to tell the TypeScript compiler that you know more about the type of a value than it does. It allows you to explicitly specify the type of a variable, even if TypeScript's type inference might infer a different or broader type.

```
let myValue: any = "Hello, TypeScript!";
let strLength: number = (myValue as string).length;
```

64.What is arrow function in typescript?

Ans: - Arrow functions in TypeScript (and JavaScript) are a concise way to define anonymous functions using a compact syntax, introduced in ECMAScript 6 (ES6). They provide a more concise syntax compared to traditional function expressions, especially for short, one-liner functions.

```
// Traditional function expression
let add = function(x: number, y: number): number {
  return x + y;
};

// Arrow function
let addArrow = (x: number, y: number): number => {
  return x + y;
};
```

65.What is Object oriented programming in typescript?

Ans: - Object-oriented programming (OOP) in TypeScript is a programming paradigm that revolves around the concept of objects and classes. It allows developers to model real-world entities as objects and define their behaviour and interactions through classes and their relationships.

66.What are class and object in typescript?

Ans: - In TypeScript, classes and objects are fundamental concepts for object-oriented programming. Here's a breakdown of each:

Class: A blueprint or template that defines the properties (data) and methods (functions) that objects of that class will share. Acts as a reusable construct to create objects with similar characteristics. You can define access modifiers (public, private, protected) to control access to properties and methods within the class and from outside objects.

Object: An instance of a class. It's a concrete entity that holds data (specific values for the properties defined in the class) and can execute the methods defined in the class. You create objects using the new keyword followed by the class name.

67.What is constructor?

Ans: - A constructor is a special function within a class that is invoked automatically whenever you create a new object (instance) of that class. Constructor is used to initialize the properties

(data) of a new object when it's created. It does not return any value. It can have optional parameters to accept values that will be used to initialize the object's properties.

68.What are access modifiers in typescript?

Ans: - In typescript class we can modify visibility of properties with the help of access modifiers. There are mainly three visibility modifiers in typescript.

public - (default) allows access to the class member from anywhere

private - only allows access to the class member from within the class

protected - allows access to the class member from itself and any classes that inherit it.

69.What is encapsulation in typescript?

Ans: - Encapsulation in TypeScript is a fundamental principle of object-oriented programming (OOP) that involves bundling the data (properties) and methods (functions) that operate on the data within a single unit called a class. Encapsulation allows the internal state of an object to be hidden from the outside world, and it provides a mechanism for controlling access to the object's data and behaviour.

```
class Car {
  private make: string;
  private model: string;

  constructor(make: string, model: string) {
      this.make = make;
      this.model = model;
  }

  getMake(): string {
      return this.make;
  }

  getModel(): string {
      return this.model;
  }
}

let myCar = new Car("Toyota", "Camry");

console.log(myCar.getMake());   // Output: Toyota
console.log(myCar.getModel());  // Output: Camry

// Accessing private members directly causes a compilation error
// console.log(myCar.make);   // Compilation Error: Property 'make' is private
and only accessible within class 'Car'.
// console.log(myCar.model);  // Compilation Error: Property 'model' is
private and only accessible within class 'Car'.
```

70.What is inheritance in typescript?

Ans: - Inheritance in TypeScript is a mechanism by which a class (subclass or derived class) can inherit properties and methods from another class (superclass or base class). It allows developers to create a hierarchical relationship between classes, promoting code reuse and establishing an "is-a" relationship between related types.

Inheritance in TypeScript is achieved using the extends keyword. The subclass inherits all members (properties and methods) of the superclass and can also define additional members or override existing ones.

```
class Animal {
  name: string;

  constructor(name: string) {
      this.name = name;
  }

  makeSound(): string {
      return "Some sound";
  }
}

class Dog extends Animal {
  // The Dog class inherits properties and methods from the Animal class

  makeSound(): string {
      return "Woof!";
  }
}

let dog = new Dog("Buddy");
console.log(dog.name);          // Output: Buddy (inherited from Animal)
console.log(dog.makeSound());   // Output: Woof! (overridden method in Dog)
```

71.What is polymorphism in typescript?

Ans: - Polymorphism in TypeScript is a concept that allows objects of different classes to be treated as objects of a common superclass or interface. It enables code to work with objects of various types in a uniform manner, providing flexibility and extensibility in object-oriented programming.

There are two main types of polymorphism:

1) **Compile-Time Polymorphism (Method Overloading):** Compile-time polymorphism occurs when a method is overloaded with multiple definitions, each having a different signature (number or types of parameters). The appropriate method to invoke is determined at compile time based on the number and types of arguments provided.

   ```
   class Calculator {
   ```

```
  add(x: number, y: number): number {
      return x + y;
  }

  add(x: string, y: string): string {
      return x + y;
  }
}

let calc = new Calculator();
console.log(calc.add(2, 3));       // Output: 5 (calls the first add
method)
console.log(calc.add("Hello", "!")); // Output: Hello! (calls the
second add method)
```

2) **Run-Time Polymorphism (Method Overriding):** Run-time polymorphism occurs when a method in a subclass overrides a method with the same signature in its superclass. The appropriate method to invoke is determined at runtime based on the actual type of the object.

```
class Animal {
  speak(): string {
      return "Animal makes a sound";
  }
}

class Dog extends Animal {
  speak(): string {
      return "Woof!";
  }
}

class Cat extends Animal {
  speak(): string {
      return "Meow!";
  }
}

let dog: Animal = new Dog();
let cat: Animal = new Cat();

console.log(dog.speak()); // Output: Woof!
console.log(cat.speak()); // Output: Meow!
```

72.What is interface in typescript?

Ans: - An interface is a syntactical contract that defines the structure of an object. It defines the properties and methods that a class must implement if it implements that interface. Interfaces provide a way to define the shape of objects, allowing for better type checking and code documentation.

```
interface Person {
  firstName: string;
  lastName: string;
  age: number;
}

let person: Person = {
  firstName: "John",
  lastName: "Doe",
  age: 30
};
```

Interfaces in TypeScript can also include optional properties, readonly properties, methods, and index signatures:

Optional Properties:

```
interface SquareConfig {
  color?: string;
  width?: number;
}
```

Readonly Properties:

```
interface Point {
  readonly x: number;
  readonly y: number;
}
```

Methods:

```
interface Printer {
  print(): void;
}
```

Index Signatures:

```
interface StringArray {
  [index: number]: string;
}
```

73.What is the difference between extended and implements in typescript?

Ans: - In TypeScript, both extends and implements are used to establish relationships between classes and interfaces, but they serve different purposes:

extends: extends is used to create a subclass (or derived class) that inherits properties and methods from a superclass (or base class). It establishes an "is-a" relationship between classes, indicating that the subclass is a more specialized version of the superclass. Subclasses inherit all members (properties and methods) of the superclass and can also define additional members or override existing ones.

```
class Animal {
  move() {
      console.log("Moving...");
  }
```

```
}

class Dog extends Animal {
  bark() {
      console.log("Woof!");
  }
}

let dog = new Dog();
dog.move();  // Output: Moving...
dog.bark();  // Output: Woof!
```

implements: `implements` is used to ensure that a class meets the structure defined by an interface. It enforces a contract between the class and the interface, requiring the class to provide implementations for all members defined by the interface. It establishes a "has-a" or "supports" relationship between a class and an interface, indicating that the class supports the behaviour specified by the interface. Classes can implement multiple interfaces, allowing them to provide implementations for the members of each interface.

```
class Doc implements Printable {
  private content: string;

  constructor(content: string) {
      this.content = content;
  }

  print() {
      console.log("Printing:");
      console.log(this.content);
  }
}

(new Doc("This is the content of the document.")).print();
```

74.What is multiple inheritance? is multiple inheritance is possible in typescript?

Ans: - Multiple inheritance refers to a programming language feature where a class can inherit properties and methods from more than one superclass. In other words, a subclass can have multiple parent classes.

Multiple inheritance can lead to issues such as the diamond problem, where ambiguity arises if two parent classes have methods or properties with the same name. To address this, some languages, like Java, only allow single inheritance but support interfaces, which are similar to abstract classes in some aspects but cannot contain implementations of methods.

In TypeScript, multiple inheritance is not directly supported. TypeScript follows the principle of single inheritance for classes, meaning a class can inherit from at most one superclass. This design choice helps avoid complications associated with multiple inheritance, such as the diamond problem.

75.What are Angular forms?

Ans: - Angular forms are a crucial part of building dynamic and interactive web applications with Angular. They allow users to input data, submit it to the server, and interact with the application's data in various ways.

76.What are the types of Angular forms?

Ans: - Angular provides two types of forms:

Template-driven forms: Template driven forms are model-driven forms where you write the logic, validations, controls etc, in the template part of the code using directives.

Reactive forms: Reactive forms is a model-driven approach for creating forms in a reactive style (form inputs changes over time). These are built around observable streams, where form inputs and values are provided as streams of input values.

77.What is the difference between template driven form and reactive form?

Ans: -

Feature	Reactive	Template-Driven
Form model setup	Created(FormControl instance) in component explicitly	Created by directives
Data updates	Synchronous	Asynchronous
Form custom validation	Defined as Functions	Defined as Directives
Testing	No interaction with change detection cycle	Need knowledge of the change detection process
Mutability	Immutable(by always returning new value for FormControl instance)	Mutable(Property always modified to new value)
Scalability	More scalable using low-level APIs	Less scalable using due to abstraction on APIs

78.How to setup template driven form?

Ans: - To setup template driven form in angular,

1) import ‘FormsModule’ in @NgModule decorator of a module class in which you want to create a template driven form.
2) Create form template using directives like ngForm, ngSubmit, ngModel etc.
3) Add Validation.
4) Handle form submission.

79.How to apply required field validation in template driven form?

Ans: - To apply the "required" field validation in a template-driven form in Angular, you can simply use the "required" attribute on the form control elements in your HTML template.

80.What is form group and form control in Angular?

Ans: - In Angular, FormGroup and FormControl are classes provided by the @angular/forms module that are used to work with forms in reactive forms approach.

FormControl: FormControl represents an individual form control, such as an input field, textarea, select dropdown, etc. It tracks the value and validation status of the form control. You can create instances of FormControl for each form control in your form. You can apply validators to FormControl instances to enforce validation rules on the form control's value.

FormGroup: FormGroup represents a group of form controls and tracks the values and validation statuses of all the form controls within the group. It's used to group related form controls together, such as a group of input fields representing user details. You can create instances of FormGroup to define the structure of your form and manage its values and validation. You can nest FormGroup instances within each other to create complex forms with nested structures.

81.How to setup reactive forms?

Ans: - To setup reactive forms in angular,

1) Import ReactiveFormsModule in the AppModule or the module where you intend to use reactive forms.
2) Create Form Controls and Form Group in component class.
3) Bind form controls to HTML elements in the template using Angular's reactive forms directives such as formControlName, formGroupName, and formGroup directive.
4) Define a method in the component class to handle form submission.

82.How to do validations in reactive forms?

Ans: - To perform validations we can use Validators provided by Angular's `@angular/forms` module. Validators can be applied to individual form controls to enforce validation rules on their values.

83.What is authentication and authorization in Angular?

Ans: - In Angular applications, authentication and authorization are essential aspects of building secure and reliable web applications.

Authentication: Authentication is the process of verifying the identity of a user, typically through credentials such as username/password, tokens, or other authentication mechanisms like OAuth/OpenID Connect. Once a user is authenticated, they are granted access to the application's resources and functionalities.

Authorization: Authorization is the process of determining whether a user has permission to access a particular resource or perform a specific action within the application. It involves evaluating the user's identity and associated roles or permissions against the access control rules defined by the application. In Angular, authorization is typically implemented using route guards and role-based access control (RBAC).

84.What is JWT token authentication in Angular?

Ans: - JWT (JSON Web Token) authentication is a popular method used in Angular (and other web applications) for securing APIs and authenticating users. It involves the use of JSON Web Tokens to securely transmit authentication information between the client (Angular application) and the server.

85.How to mock or fake an API from JWT authentication?

Ans: - To mock or fake an API authentication using JWT authentication in Angular for testing or development purposes, you can create a mock authentication service that simulates the behaviour of a real authentication service. This mock authentication service can generate JWT tokens and handle authentication requests without actually communicating with a backend server.

86.How to implement the authentication with JWT in Angular?

Ans: - Implementing authentication with JWT in Angular involves several steps, including setting up the authentication service, handling login/logout, protecting routes, and storing tokens securely.

Here's a basic guide on how to implement JWT authentication in Angular:

1) **Set Up Authentication Service:** Create an authentication service in Angular to handle authentication-related logic, such as login, logout, token storage, and user authentication status.
2) **Login Component:** Create a login component where users can enter their credentials (e.g., username and password) and initiate the authentication process.
3) **Authentication API Interaction:** Implement methods in the authentication service to interact with the backend authentication API. This includes sending login requests to the server, receiving and storing JWT tokens, and handling authentication errors.
4) **Token Storage:** Store the JWT token securely upon successful authentication. Common storage mechanisms include browser localStorage or sessionStorage.
5) **Protect Routes:** Use Angular route guards to protect routes that require authentication. Implement an authentication guard to check if the user is authenticated (i.e., has a valid JWT token) before allowing access to protected routes.
6) **Logout Functionality:** Implement a logout function in the authentication service to clear the stored JWT token and log the user out.
7) **Interceptors:** Use Angular HTTP interceptors to automatically attach the JWT token to outgoing HTTP requests. This ensures that authenticated users can access protected API endpoints without manually adding the token to each request.
8) **Error Handling:** Implement error handling in the authentication service to handle authentication errors gracefully. This includes handling invalid credentials, server errors, token expiration, etc.

87.What is auth guard?

Ans: - An auth guard, short for authentication guard, is a feature in Angular that allows you to control access to certain routes in your application based on whether the user is authenticated

or not. Auth guards are used to protect routes and prevent unauthorized access to sensitive parts of your application.

88.What is http interceptor?

Ans: - The angular interceptor help us to modify http request by intercepting it before the request is send to the server. Interceptor can also modify the incoming response from the server.

The interceptor globally catches every outgoing and incoming request at a single place. We can use interceptor to add custom headers to the outgoing request, log the incoming response.

89.How to retrieve automatically if there is an error in response from Api?

Ans: - To automatically handle errors in responses from an API in Angular, you can use Angular's built-in HttpInterceptor. HttpInterceptor allows you to intercept HTTP requests and responses, enabling centralized error handling and logging in your application.

90.What are the paths of JWT tokens?

Ans: - JSON Web Tokens (JWT) typically contain several standard claims, which are pieces of information asserted about a subject (usually the user) and any additional custom claims defined by the application. These claims are encoded into the JWT and can be accessed by the recipient to verify the authenticity of the token and extract relevant information about the user.

Here are the standard claims commonly found in JWT tokens:

1) **Issuer (iss):** Indicates the issuer of the token. This is typically the URL of the authentication server or the identity provider that issued the token.
2) **Subject (sub):** Identifies the subject of the token, which is usually the unique identifier of the user or entity that the token represents.
3) **Audience (aud):** Specifies the intended audience for the token. This can be the URL of the resource server or API that the token is intended for.
4) **Expiration Time (exp**): Specifies the expiration time of the token, after which it should no longer be considered valid. This is usually represented as a Unix timestamp.
5) **Not Before (nbf):** Specifies the time before which the token should not be accepted for processing. This is useful for implementing token replay prevention.
6) **Issued At (iat):** Specifies the time at which the token was issued (created). This can be used to determine the age of the token.
7) **JWT ID (jti):** Provides a unique identifier for the token. This can be used to prevent replay attacks by tracking used tokens.

91.What is postman?

Ans: - Postman is a popular API client tool used for testing, debugging, and documenting APIs. It allows users to send HTTP requests to APIs, inspect the responses, and automate testing workflows. Postman supports various request types (e.g., GET, POST, PUT, DELETE), authentication methods, and environments.

92.Which part of request has the token stored when sending to Api?

Ans: - When sending a token to an API for authentication, the token is typically included in the request headers, specifically in the Authorization header. The token is usually prefixed with the token type (e.g., Bearer) followed by a space, and then the token itself. For example,

```
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

93.What are the various ways to communicate between the components?

Ans: - In Angular, there are several ways to communicate between components:

- Using Input and Output properties for parent-child communication.
- Using services for sharing data between unrelated components.
- Using @ViewChild and @ContentChild decorators for accessing child components and elements respectively.
- Using RxJS BehaviorSubject or Subject for cross-component communication.

94.What is content projection? What is ng content?

Ans: - Content projection is a technique in Angular for passing content from a parent component to a child component via transclusion. <ng-content> is a built-in Angular directive used to define insertion points within the template of a component where the content provided by the parent component will be projected.

95.What is template reference variable in Angular?

Ans: - template reference variable in Angular is a variable declared in the template that provides a reference to a specific element or component. It is prefixed with the hash symbol (#) and can be used to access the element or component in the template or in event handlers.

96.What is the role of view child in Angular?

Ans: - ViewChild is a decorator in Angular used to access child components, directives, or elements from the parent component's template. It allows the parent component to interact with its child components and access their properties and methods programmatically.

97.How to access the child components from parent component with view child?

Ans: - To access child components from parent component with view child decorator

```
import { Component, ViewChild } from '@angular/core';
import { ChildComponent } from './child.component';

@Component({
  selector: 'app-parent',
  template: '<app-child></app-child>'
})
```

```
export class ParentComponent {
  @ViewChild(ChildComponent) childComponent: ChildComponent;

  ngAfterViewInit() {
    // Access child component properties or methods here
    console.log(this.childComponent.someMethod());
  }
}
```

98. What is the difference between view child and view children? What is query list?

Ans: - ViewChild is used to access a single child component or element, while ViewChildren is used to access multiple child components or elements. QueryList is a collection-like object provided by Angular that holds the results of a query performed on the view.

99. What is content child?

Ans: - ContentChild is a decorator used to access projected content or directives within the content of a component. It allows a component to query and access content projected into it from its parent component's template.

100. What is the difference between content child and content children? Compare <ng-content>, view child, view children, content child and content children?

Ans: - **<ng-content>** is used to project content from a parent component to a child component.

ViewChild is used to access child components or elements within the parent component's template.

ViewChildren is used to access multiple instances of child components or elements within the parent component's template.

ContentChild is used to access projected content or directives within the content of a component.

ContentChildren is used to access multiple instances of projected content or directives within the content of a component.