Angular Interview Questions

1) What is Angular and require software to setup?

-Angular is the complete rewrite of the Angular1.0.
-It supports two-way data binding
-It uses TypeScript to build the application.
-open-source web application development framework created by Google
-It makes use of directives and components
- Angular offers mobile support
-It uses @RouteConfig{(…)} for Routing
-It supports hierarchical dependency injection, along with a unidirectional tree-based change direction
-Its simplified structure makes it easy for professionals to develop and maintain large applications easily
- NodeJS software is required to for Angular project
- npm – Node Package Manager is used to install dependencies of the project

2) What is SPA(Single Page Application)?

-In angular we have only one html file called index.html.
- In the SPA, the whole page is not reloaded every time, only every time the view will be changed.
-So when you load the application for the first time, not all the pages from the server will be rendered... It's only index.html that loads when you load the application. Since only a single page is loaded it is called SPA.

3) What is the Angular CLI and cmd to install?

- The Angular CLI is a command-line interface tool that you use to initialize, develop, scaffold, and maintain Angular applications directly from a command shell.

```
npm install -g @angular/cli
```

4) How to Create a new Angular project using Angular-cli cmd?

ng new project-name

```
installing ng2
  create .editorconfig
  create README.md
  create src/app/app.component.css
  create src/app/app.component.html
  create src/app/app.component.spec.ts
  create src/app/app.component.ts
  create src/app/app.module.ts
  create src/assets/.gitkeep
  create src/environments/environment.prod.ts
  create src/environments/environment.ts
  create src/favicon.ico
  create src/index.html
  create src/main.ts
  create src/polyfills.ts
  create src/styles.css
  create src/test.ts
  create src/tsconfig.json
  create .angular-cli.json
  create e2e/app.e2e-spec.ts
  create e2e/app.po.ts
  create e2e/tsconfig.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tslint.json
Successfully initialized git.
Installing packages for tooling via npm.
```

5) What is the package.json file?

- Contains dependencies of the project and contains third party dependencies.
npm install cmd is used to install all the dependencies of projects

package.json

```
{
          "name": "angular-srcs",
          "version": "12.2.0-next.1",
          "private": true,
          "description": "Angular - a web framework for modern web apps",
          "homepage": "https://github.com/angular/angular",
          "bugs": "https://github.com/angular/angular/issues",
          "engines": {
            "node": "^12.20.0 || >=14.0.0",
            "yarn": ">=1.22.4 <2",
            "npm": "Please use yarn instead of NPM to install dependencies"
          },
```

```
  "repository": {
    "type": "git",
    "url": "https://github.com/angular/angular.git"
  },
  "scripts": {
    "ng-dev": "node dev-infra/ng-dev",
    "ng-dev:dev": "ts-node --transpile-only -- dev-infra/cli.ts"
  },
  "// 1": "dependencies are used locally and by bazel",
  "dependencies": {
    "@angular-devkit/build-angular": "12.0.4",
    "@angular-devkit/build-optimizer": "0.1200.4",
    "@angular-devkit/core": "12.0.4",
    "requirejs": "^2.3.6",
    "rollup": "2.16.1",
    "rollup-plugin-commonjs": "^10.1.0",
    "rollup-plugin-node-resolve": "^5.2.0",
    "rollup-plugin-sourcemaps": "^0.6.3",
    "yargs": "^17.0.0"
  },
  "// 2": "devDependencies are not used under Bazel. Many can be removed after
test.sh is deleted.",
  "devDependencies": {
}
```

6) Angular Folder structure?

```
├── README.md              // an useful README
├── .angular-cli.json      // angular-cli configuration file
├── e2e/                   // end to end tests
├── karma.conf.js          // unit test configuration
├── node_modules/          // installed dependencies
├── package.json           // npm configuration
├── protractor.conf.js     // e2e test configuration
├── src/                   // application source
└── tslint.json            // linter config file
```

Angular-cli.json

- angular.json is the file which has various properties and configuration of your Angular project.
- This is the file which is first referred by the builder to look for all the paths and configurations and to check which is the main 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"
      ]
    }
      "es5BrowserSupport":true
  }
```

Src→

```
|-- app/
|   |-- app.component.css
|   |-- app.component.html
|   |-- app.component.spec.ts
|   |-- app.component.ts
|   `-- app.module.ts
|-- assets/
|-- environments/
|   |-- environment.prod.ts
|   `-- environment.ts
|-- favicon.ico
|-- index.html
|-- main.ts
|-- polyfills.ts
|-- styles.css
|-- test.ts
`-- tsconfig.json
```

Index.html

-Here, the index.html file is called. It is found in the src folder of the app.
-Compiler dynamically adds all the javascript files at the end of this file.
-Since all the components are now known, the html file calls the root component that is app-root.
- The root component is defined in app.components.ts which targets app.component.html.

```
<!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>
```

main.ts

-This file acts as the entry point of the application.

-This entry point is defined in the internals of webpack that is used by Angular to support the modular functionality.

-The path/name of the main file can be changed but it should also be changed in the angular.json file. Main.ts helps in creating the browser environment for the application to run.

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

platformBrowserDynamic().bootstrapModule(AppModule)
```

app.module.ts

-This is the module, created with the @NgModule decorator, which has declarations of all the components we are creating within the app module so that angular is aware of them.

- Here, we also have an imports array where we can import other modules and use them in our app. Below is an example of an app.module.ts file with a test component declared and two modules imported.

```
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 { }
```

Angular has a powerful concept of *modules*. When you boot an Angular app, you're not booting a component directly, but instead you create an `NgModule` which points to the component you want to load.

Take a look at this code:

code/first-app/angular-hello-world/src/app/app.module.ts

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

app.component.ts

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

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

7) Run Angular Project in cmd?

```
$ cd angular-hello-world
$ ng serve
** NG Live Development Server is running on http://localhost:4200. **
// ...
// a bunch of other messages
// ...
Compiled successfully.
```

ng serve
Open URl in browser **http://localhost:4200**

8) What are Angular building blocks?

The following building blocks play a crucial role in Angular:

Components: A component can control numerous views wherein each of the views is a particular part on the screen. All Angular applications have a minimum of one component called the root component. This component is bootstrapped in the root module, the main module. All the components include the logic of the application that is defined in a class, while the main role of the class is to interact with the view using an API of functions and properties.

Data binding: Data binding is the process in which the various sections of a template interact with the component. The binding markup needs to be added to the HTML template so that Angular can understand how it can connect with the component and template.

Dependency injection: It uses DI so that it can offer the necessary dependencies, mainly services, to the new components. The constructor parameters of a component inform Angular regarding the numerous services needed by the component, and DI provides a solution that gives the necessary dependencies to the new class instances.

Directives: Angular templates are of dynamic nature, and directives help Angular understand how it can transform the DOM while manifesting the template.

Metadata: Classes have metadata attached to them with the help of decorators so that Angular will have an idea of processing the class.

Modules: Module or NgModule is a block of code organized using the necessary capabilities set, having one specific workflow. All Angular applications have at least one module, the root module, and most of the applications have numerous modules.

Routing: Angular router helps interpret the URL of a browser to get a client-generated experience and view. This router is bound to page links so that Angular can go to the application view as soon as the user clicks on it.

Services: Service is a vast category that ranges from functions and values to features that play a significant role in Angular application.

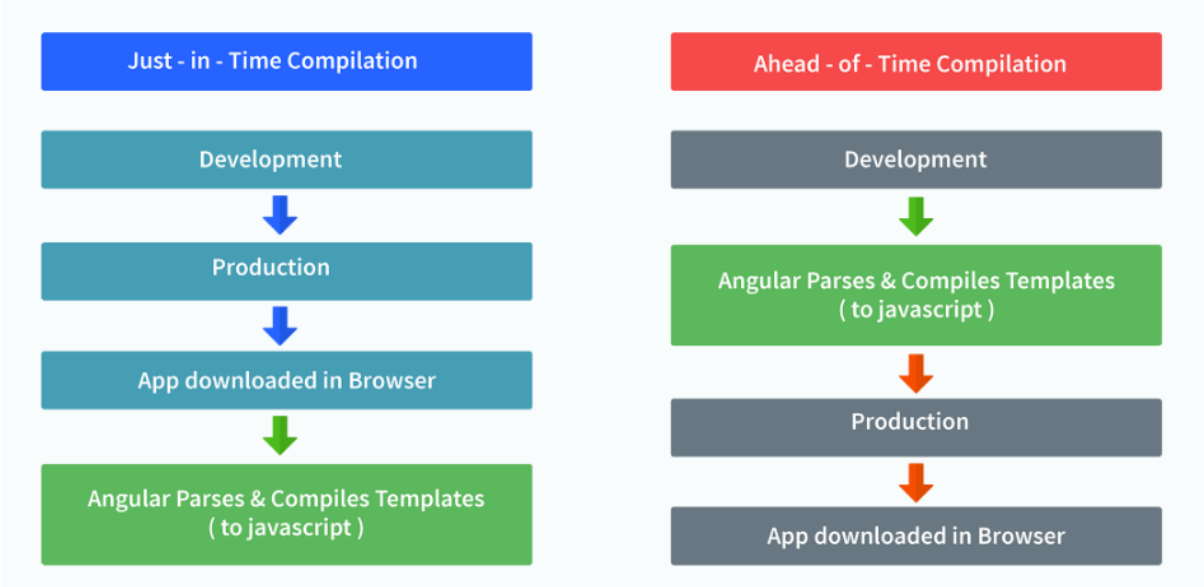
Template: The view of each component is linked with a template, and an Angular template is a type of HTML tag that allows Angular to get an idea of how it needs to render the component.

9) What is AOT compilation? What are the advantages of AOT?

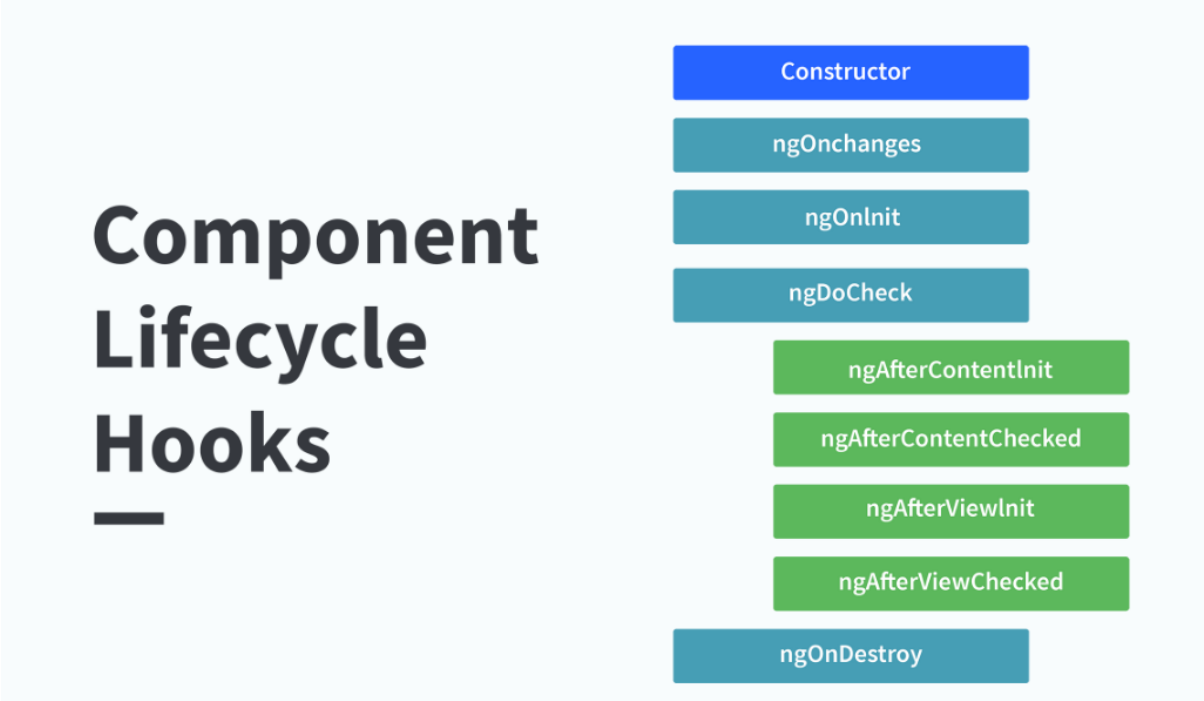
Angular application consists of components and templates which the browser cannot understand. Therefore, all the Angular applications need to be compiled first before running inside the browser.

Angular provides two types of compilation:

- JIT(Just-in-Time) compilation (ng serve)
- AOT(Ahead-of-Time) compilation (ng serve –aot)



10) What are lifecycle hooks in Angular?



ngOnChanges( )
-This hook/method is called before **ngOnInit** and whenever one or more input properties of the component changes.
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 oftenly used hook. If one has to process 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..
 }

}
```

11) What is binding in Angular?

Data binding deals with how to bind your data from components to HTML DOM elements (Templates).

We can easily interact with applications without worrying about how to insert your data.

We can make connections in two different ways:

1. one way data binding
2. two-way data binding.

One way data binding

1) String interpolation
   Ex: <h1>{{variable-name}}</h1>
2) Event Binding
   Ex: <button (click)=MethodName() ></button>
3) Property binding
   Ex: <input [value]=”ComponentData”>
4) Attribute binding
   Ex: <input [attr.ATTR]=”ComponentData”>
5) Class binding
   Ex: <input [class.blue]=”ComponentData”>

Two way data binding

-Two-way data binding is a two-way interaction, data flows in both ways (from component to views and views to component).

-Simple example is ngModel. If you do any changes in your property (or model) then, it reflects in your view and vice versa. It is the combination of property and event binding.

-ngModel is a standalone directive.

- ngModel directive binds form control to property and property to form control.

Ex: <input [(ngModel)]=”ComponentData”>
<h1>{{ComponentData}}>

12) Promise or observable in angular?

- Promise emits a single value while Observable emits multiple values.
- So, while handling an HTTP request, Promise can manage a single response for the same request, but what if there are multiple responses to the same request, then we have to use Observable.

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
	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:

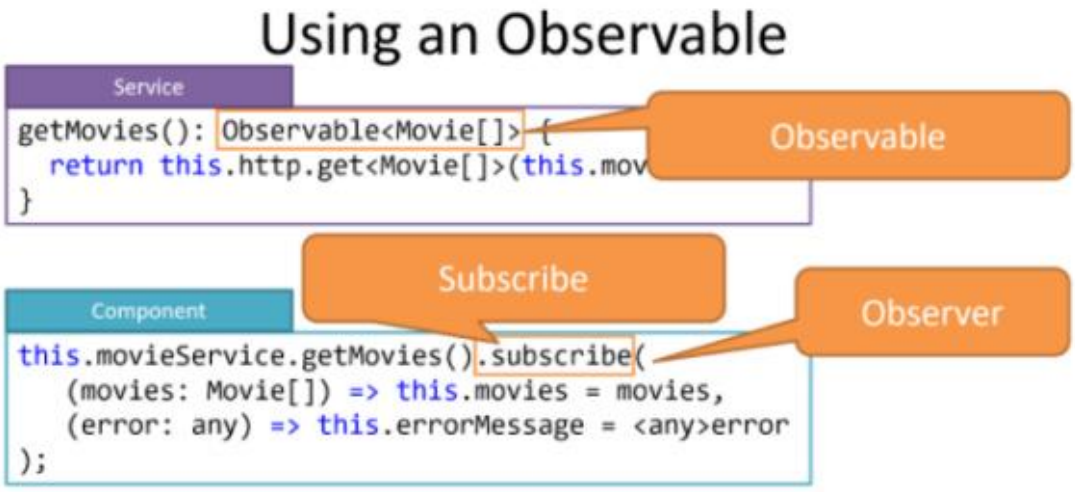
```
Before subscribing an Observable
Text inside an observable
Hello world!
```

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!
```



13) What is an Interceptor in angular?

-Interceptors are a unique type of Angular Service that we can implement. Interceptors allow us to intercept incoming or outgoing HTTP requests using the HttpClient .

-By intercepting the HTTP request, we can modify or change the value of the request.

```
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from "@angular/common/http";
import { Injectable } from "@angular/core";
import { Observable } from "rxjs";

@Injectable()
export class HttpHeadersInterceptor implements HttpInterceptor {
  constructor() {}

  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    req = req.clone({
      setHeaders: {
        'x-rapidapi-key': 'esGbwrm390mshS2BCl0RALxQRtZTp1W7sFMjsnyJlJzDXVkW0H',
        'x-rapidapi-host': 'rawg-video-games-database.p.rapidapi.com',
      },
      setParams: {
        key: 'e40e743af2c94b0c916a8aa618fb4473',
      }
    });
    return next.handle(req);
  }
}
```

14) What is 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.

15) 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.

16) What are the types of Directive?

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.

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `
    <h1>Angular 2 App</h1>
    <p>Component is directive with template</p>
  `
})
export class AppComponent {
}
```

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.

```
<div *ngFor="let item of items">{{ item.description }}</div>

<span *ngIf="isVisible"></span>
```

Attribute Directives

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

```
<div [class.active]="isActive"></div>

<span [style.color]="'red'"></span>

<p [attr.data-note]="'This is value for data-note attribute'">A lot of text here</p>
```

17) 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>
```

18) What is Angular Routing?

In a single-page app, you change what the user sees by showing or hiding portions of the display that correspond to particular components, rather than going out to the server to get a new page.

To handle the navigation from one to the next, you use the Angular router The router enables navigation by interpreting a browser URL as an instruction to change the view.

```
<router-outlet></router-outlet>
<!-- Routed components go here -->
```

code/routes/routing/src/app/app.module.ts

```
const routes: Routes = [
  // basic routes
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'contactus', redirectTo: 'contact' },

  // authentication demo
  { path: 'login', component: LoginComponent },
  {
    path: 'protected',
    component: ProtectedComponent,
    canActivate: [ LoggedInGuard ]
  },

  // nested
  {
    path: 'products',
    component: ProductsComponent,
    children: childRoutes
  }
];
```

19) What is While card routes?

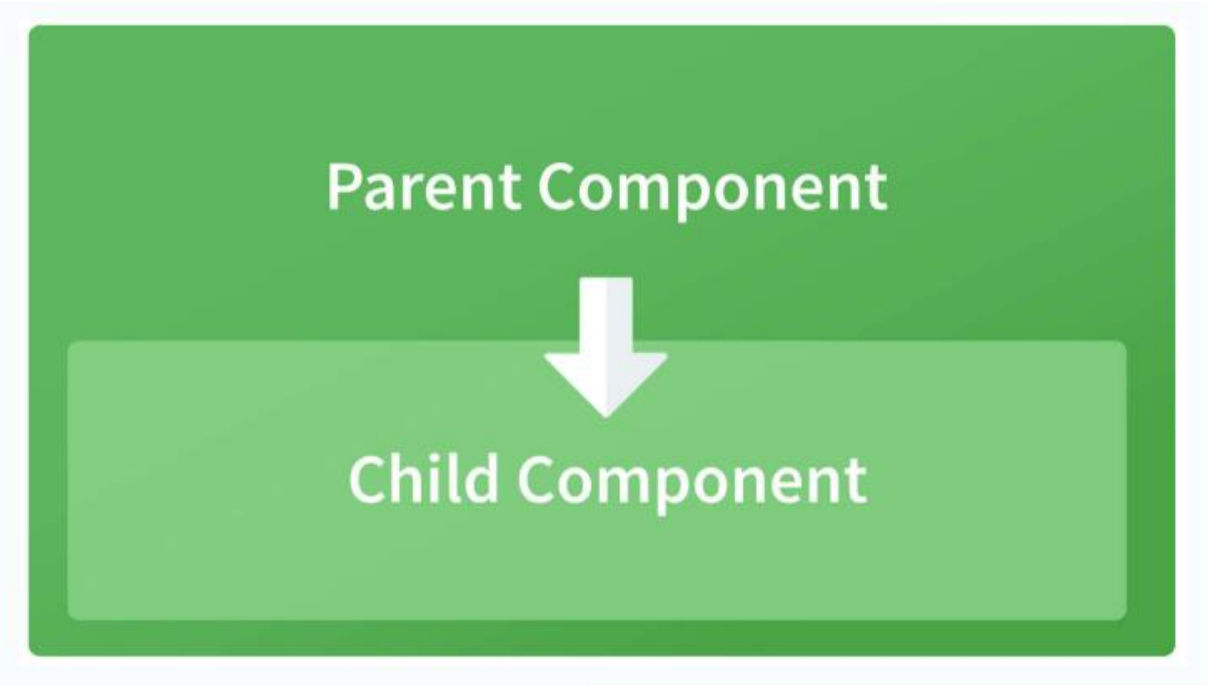
A well-functioning application should gracefully handle when users attempt to navigate to a part of your application that does not exist. To add this functionality to your application, you set up a wildcard route. The Angular router selects this route any time the requested URL doesn't match any router paths.

{ path: '**', component:ComponentName}

20) 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



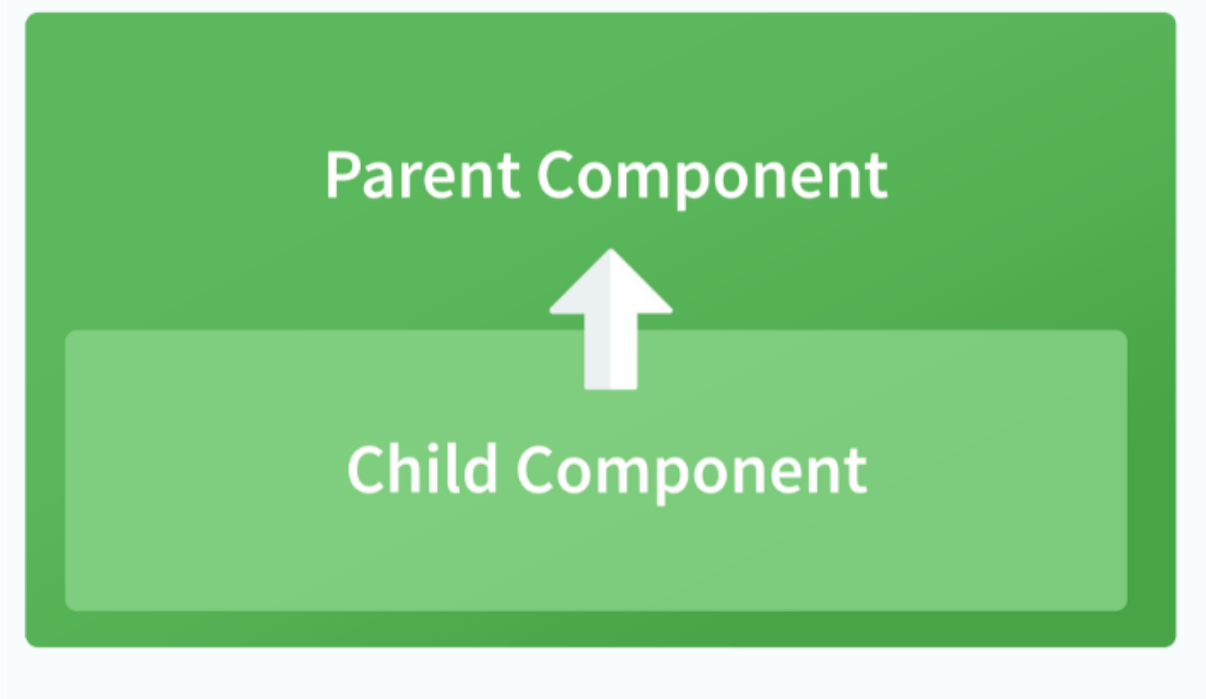
Consider the following parent component:

```
@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;
}
```

21) 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.

```
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();
  }
}
```

22) What is AuthGuard?

- AuthGuard is a class which implements the interface CanActivate , to decide whether the user has access/permission to view a specific page / route / path in the application or not.
-This will be useful when we need authentication/authorization based control over the application.

23) What is pipe in angular?

-Pipes are a useful feature in Angular.
-They are a simple way to transform values in an Angular template. There are some built in pipes, but you can also build your own pipes. -A pipe takes in a value or values and then returns a value.
- Use pipes to transform strings, currency amounts, dates, and other data for display.
-Pipes are simple functions you can use in template expressions to accept an input value and return a transformed value.
-Pipes are useful because you can use them throughout your application, while only declaring each pipe once.

```
{{ exp | myPipe }}
```

24) Types of Angular Forms ?

There are two types of form approaches in Angular.

1. ReactiveForms
2. Template-driven Forms

Reactive Forms
Creating a form using FormControl, FormGroup, and FormArray are said to be reactive forms. They use the ng module as ReactiveFormsModule.

FormControl:
FormControl is the class that is used to get and set values and validation of the form control such as <input> and <select> tag.

FormGroup:
FormGroup has the role to track the value and validity state of a group of FormControl.

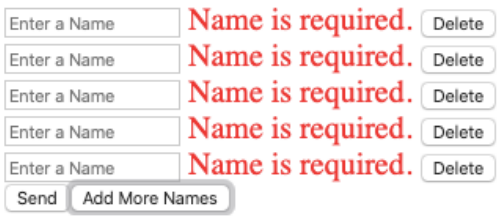
FormArray:
FormArray tracks the value and validity state of the array of FormControl, FormGroup, or FormArray instances.

```
// app.component.ts import { Component } from '@angular/core';
import { FormControl, FormGroup, FormArray, Validators } from '@angular/forms';
@Component({ selector: 'app-root', templateUrl: './app.component.html', styleUrls: ['./app.component.css'] })
```

```
export class AppComponent {

angForm = new FormGroup({ names: new FormArray([ new FormControl('', Validators.required), new
FormControl('', Validators.required), ]) });

get names(): FormArray {
 return this.angForm.get('names') as FormArray;
}

addNameField() {
   this.names.push(new FormControl('', Validators.required));
}

deleteNameField(index: number) {
this.names.removeAt(index);
}
onFormSubmit(): void {
 for(let i = 0; i < this.names.length; i++) { console.log(this.names.at(i).value);
  }
  }
}
```

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

<form [formGroup] = "angForm" (ngSubmit)="onFormSubmit()">
  <div formArrayName="names">
    <div *ngFor="let name of names.controls; index as idx">
      <input [formControlName]="idx" placeholder="Enter a Name">
      <label *ngIf="name.invalid" [ngClass] = "'error'"> Name is required. </label>
      <button (click)="deleteNameField(idx)">Delete</button>
    </div>
  </div>
  <div>
    <button type="submit">Send</button>
    <button type="button" (click)="addNameField()">Add More Names</button>
  </div>
</form>
```

Save the file, and you will see the following.



Template-Driven Form

Template Drive Forms are just Model Driven Form but *driven* by directives in the template versus code in the component.

In template-driven, we use directives to create the model. In model-driven, we generate a model on the component and then use directives to map elements in the template to our form model.

25) Let vs Var vs Constat?

var

Variables in TypeScript can be declared using var keyword, same as in JavaScript. The scoping rules remains the same as in JavaScript.

let

To solve problems with var declarations, ES6 introduced two new types of variable declarations in JavaScript, using the keywords let and const. TypeScript, being a superset of JavaScript, also supports these new types of variable declarations.

Example: let Variables Scope

```
let num1:number = 1;

function letDeclaration() {
    let num2:number = 2;

    if (num2 > num1) {
        let num3: number = 3;
        num3++;
    }

    while(num1 < num2) {
        let num4: number = 4;
        num1++;
    }

    console.log(num1); //OK
    console.log(num2); //OK
    console.log(num3); //Compiler Error:
    console.log(num4); //Compiler Error:
}

letDeclaration();
```

Example: var Variables Scope

```
var num1:number = 1;

function varDeclaration() {
    var num2:number = 2;

    if (num2 > num1) {
        var num3: number = 3;
        num3++;
    }

    while(num1 < num2) {
        var num4: number = 4;
        num1++;
    }

    console.log(num1); //2
    console.log(num2); //2
    console.log(num3); //4
    console.log(num4); //4
}

varDeclaration();
```

Const

Variables can be declared using const similar to var or let declarations. The const makes a variable a constant where its value cannot be changed. Const variables have the same scoping rules as let variables.

```
const num:number = 100;
num = 200; //Compiler Error: Cannot assign to 'num' because it is a constant or read-only property
```

26) Share Data between Components and its types?

There are five ways to share data between components:

Parent to child component

```
<app-child                          [childToMaster]=product.productName
(childToParent)=”childToParent($event)”></app-child> //Parent


@Input('childToMaster') masterName: string; //Child
```

Child to parent component

```
@Output() childToParent = new EventEmitter<String>();//Child
sendToParent(name){
  this.childToParent.emit(name);
}//child

<app-child                          [childToMaster]=product.productName
(childToParent)=”childToParent($event)”></app-child>//parent
childToParent(name){
      this.product.productName=name;
}//Parent
```

Sharing data between sibling components

Sharing data between siblings can be done by using points 1 and 2. First share data between the ***child to parent*** using output decorator and EventEmitter. Once received data in ***parent component share it to another child*** component using Input decorator. So siblings can talk each other via parent components**.**

Sharing data using ViewChild property

Using ViewChild decorator: ViewChild allows child component to be ***injected in parent component. So this make ViewChild more powerful.*** It allows parents to controls the child's methods and properties. But a parent can get access to the properties after view init event. That means we have to implement ngAfterViewInit life cycle hook in order to get the properties from parent components.

```
@ViewChild(AppChildComponent) child;constructor() { }

ngAfterViewInit()    {  this.product.productName=child.masterName; //<= This will set data
```

Here **masterName** is defined in the child component. ViewChild gives a reference to the child component in parent components. And it sets the value of child masterName in parent productName

Sharing data between not related components

When there is no relation between the component we can not pass the data using the above four methods. This happens when your components are in different modules. There are other scenarios when you have list of products and click on a particular product and then redirect to product details components. In these kinds of scenarios, we have to use data service to share data between components.

For creating data service. We have to define BehaviorSubject. BehaviorSubject holds the current value and the last value.

I always prefer to use BehaviorSubject because of following reasons:

- It automatically ***updates the lates value wherever subscribed***.
- Always give last value when called via ***getValue()*** method.
- No need to call next, just create a set and get method in order to get value.

In the data service, I have created messageSource as BehaviorSubject. This accepts editDataDetails as any. We can create editDataDetails as the type of Product interface which is a much better practice. Create changeMessage method which will set the current value of observables.

```
export class SharedDataService {

constructor(){}//Using any
public editDataDetails: any = [];
public subject = new Subject<any>();

private messageSource = new BehaviorSubject(this.editDataDetails);
currentMessage = this.messageSource.asObservable();

changeMessage(message: string) {
        this.messageSource.next(message)
  }
}
```

Now we have to share this service for the component where we want to get set the values. Call changeMessage method with value as parameter to set the value. In component where we want to receive the value subscribe it. Once you subscribe you will always get the latest value without any code changes.

```
//Set value in component 1
this.sharedDataService.changeMessage("message here");

//Get value in component 2
selectedMessage:any;ngOnInit() {
 this.sharedDataService.currentMessage.subscribe(message => (this.selectedMessage= message));

 //<= Always get current value!
}
```

Creating Get and Set Property and BehaviourSubject:

In the above example, we can optimize more using Product interface and create a getter and setter method by following method.

```
import { Injectable } from '@angular/core';import { Subject, BehaviorSubject, Observable,ReplaySubject } from 'rxjs';
```