Angular interview questions

1. **Angular advantages**

* Custom reusable components : We can create our own components as per requirements and which can be reusable across the application
* Productivity and code consistency : Components and services
* Easy testing – Unit testing and we can add debugger in browser with sourcemap as true
* High compatibility : Cross platform compatible e.g chrome or firefox. Can develop small, medium, large applications. JS is in constant state of flux so angular is a framework for which company can rely on as it provides constant support and upgradation

1. **Router**

* [@angular/router](https://v17.angular.io/api/router) API
* Define routes object
* // AppRouting.module.ts  
  const routes = [

{

path : ‘home’,

component: HomeComponent,

title: ‘Home’,

canActivate: [CanActivateAuthGuard],

loadChildren: () =>

import('./menu/menu.module').then((x) => x.MenuModule), // it will load menumodule where we will specify menu routes

pathMatch: full | prefix,  
},

{  
path: ‘learn’,

component: LearnComponent,

children:[

{

path : ‘learnAngular’,

component: LearnAngularComponent

},

path : ‘learnReact’,

component: LearnReactComponent

]},

{

path: 'team/:id',

component: TeamComponent

},

// default route

{

path: '',

redirectTo: '/home’,

pathMatch: 'full'

},

// 404 page not found if any route not matches

{

path: '\*\*',

component: WildcardComponent

},

],

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

// MenuRoutes.modules.ts

export const MenuRoutes: Routes = [

{

path: 'menu',

component: MenuListComponent

},

{

path: 'menu/:id',

component: MenuSingleComponent

}

];

@NgModule({

imports: [RouterModule.forChild(MenuRoutes)]

})

export class MenuModule {}

1. **Components**

* Main building blocks.
* Components provide us the way to build ui using template, logic file ts , styles files.
* Components are reusable
* @Component({

selector : ‘app-component-overview’,

templateUrl:’./component-overview.component.html’,

styleUrls : [‘./component-overview.component.scss’]  
})

1. **Compilation types**

* AOT
  + Best and recommended Compiler
  + Default compilation from Angular 9
  + Code compiles before loading into the browser
  + Source code translate to optimized JS code during the development phase
  + Reduces initial load time as it pre-compiles
  + Detects and reports errors during compilation phase so developer can resolve it and load application
  + Leading to reliable application with fewer runtime issues
  + ng build –aot
  + ng serve –aot
  + To create a production build with AOT optimizations, run the following command: ng build –prod

* JIT
  + Compiles code during run time, right into your browser
  + Detects and reports error during runtime
  + When opens web app in browser JIT starts compilation
  + JIT compiler reads and understands the code
  + Then translate to javascript code understandable by browser
  + Browser uses this translated JS code to build and display the applicatopn
  + More Chances of getting error during running the application in browser
  + You can use this during development but recommended AOT for prod

1. **What are the advantages of TypeScript over JavaScript?**
2. **Directives**

* Directives are used to add some behavior to DOM elements
* We can build custom directives too
* Attribute directive
  + Change look and feel of element
  + ngStyle , ngClass
  + Custom attribute directive

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

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private eleRef: ElementRef) {

eleRef.nativeElement.style.background = 'red';

}

}

* Structural directive
  + Add or remove element
  + \*ngIf
  + Custom structural directive

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({

selector: '[appNot]'

})

export class AppNotDirective {

constructor(

private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) { }

@Input() set appNot(condition: boolean) {

if (!condition) {

this.viewContainer.createEmbeddedView(this.templateRef);

} else {

this.viewContainer.clear(); }

}

}

1. **Services**

* Common tasks can be written is services
* It is used to share the data within application
* Reusable
* Code organization
* Dependency injection – Services are injected into components using DI

1. **Dependency injection**

* For service to be used in component first it needs to be injected. DI takes care of it
* Component doesn’t need to worry about creating the service as it will be take care by DI
* Modular so reusable
* We can use service in component, directive, pipes and other service

1. Providers
   1. Object for managing and creating instances of dependencies that can be injected into component and services
   2. Import { Component, Injectable } from ‘@angular/core’;

@Injectable()

export class MyService{

getData() {  
 return ‘some data’;

}

}

@Component({

Selector: ‘app-my-component’,

Providers: [ MyService ],

Template: ‘templateurl’

})

export class MyComponent {

constructor(private myService: MyService) {}

data = this.myService.getData();

}

1. Injectors
   1. Injectors responsible for managing and creating dependencies
   2. import { Component, Injectable, Injector } from '@angular/core';

@Injectable()

export class MyService {

getData() {

return "Data from MyService";

}

}

@Component({

selector: 'my-component',

template: '{{ data }}'

})

export class MyComponent {

constructor(private injector: Injector){}

data = this.injector.get(MyService).getData();

}

1. Normal way
   1. import { Component, Injectable, Injector } from '@angular/core';

@Injectable(

{

providedIn: 'root'

}

)

export class MyService {

getData() {

return "Data from MyService";

}

}

@Component({

selector: 'my-component',

template: '{{ data }}'

})

export class MyComponent {

constructor( private myService: MyService ){}

data = this.myService.getData();

}

1. **Angular 12 Features**

* **nullish coalescing operator**

old syntax  
{{age !== null && age !== undefined ? age : calculateAge() }}

new

**{{ age ?? calculateAge() }}**

* **TypeScript 4.2**

improvements in operator, type alias, tuple types, and abstract classes

* **Deprecating support for IE11**

As Microsoft stop support for IE11

* **Sass support for inline styles**

We can use SAAS together with inline styles in component’s metadata

e.g

@Component({

Styles: [

`  
@import ‘’style/base’

.heading{

.headigText{

Margin-bottom:40px

}

}

`

]

})

To enable this feature add “inlineStyleLanguage”: “scss” to angular.json

* **Production build by default**

Running ng build now defaults to production which saves teams some extra steps and helps to prevent accidental development builds in production!

* **Router changes**

routerLinkActiveOptions – possible to specify whether we need exact match or not for different part of url to apply css class

* **Http improvements**
* Introduced human-readable names for HTTP status code

e.g if(response.status === HttpStatusCode.Ok){}  
HttpStatusCode.Ok is 200

* httpParams has new appendAll method

const params = new HttpParams().appendAll({  
 ‘pageId’:’invest’,

‘project’:’PE’  
})

Const url = ‘https://test:8080/data’;

Const newUrl = url + ‘?’ + params.toString();

Console.log(newUrl) // https://test:8080/data?pageId=invest&project=PE

1. **Observables**

* To pass the data within components or application
* Asynchronous operations
* Observable will send the data
* Observer will listen to data passed by observable
* Observable part of RxJS library
* Subscriber take three optional parameters  
  this.myObservable.subscribe(next, error, complete);
* Next, error, complete
* Observable will emit data only if there is subscriber (observer)
* Its unicast because each subscribed observer has its own execution
* E.g  
  public observerValue = new Observable((observerVal)=>{  
   observerVal.next(Math.floor(Math.random() \* 99 ) + 1);   
  })

ngOnInit(){  
 observerValue.subscribe((val)=>{  
 console.log(val)  
})

observerValue.subscribe((val)=>{  
 console.log(val)  
})  
}

// Output - it will print random different numbers because each has its own execution  
99  
20

* It has separate observer and observable

1. **Subject**

* Emits data even if there is no subscriber
* Subject are multicast which means execution shared among multiple subscribers
* E.g   
  let mySubject = new Subject<any>();

mySubject.subscribe((val) => {  
 console.log(vale)  
})

mySubject.subscribe((val) => {  
 console.log(vale)  
})

mySubject.next(Math.floor(Math.Random() \* 99 +1))

//Output – it will print same random number because each subscriber shared the same execution  
  
99

* It works as both subscriber and observable

1. **Change detection**

* Sync the data between view and ts
* Change detection happens in below cases
  1. When event like click, submit, mouseover etc called
  2. Asynchronous functions like settimeout, setinterval etc called
  3. Api request is invoked
* Each component has its own change detection
* Even if there is click event in anyone component in application change detection will trigger for all components starting from top to bottom
* Still its doesn’t affect on performance because its unidirectional flow and after single pass change detection tree gets stable
* Default change detection strategy will check all components if event happed in any one component
* To  **optimize the no. of change detection we can use onPush strategy**
* @Component({

changeDetection : ChangeDetectionStartagy.OnPush  
})

If we add this in child component and click event happen in parent com then it will not update the view in child com

* OnPush is checkOnce , implies that change detection is skipped unless a comp is marked as dirty

1. **Pipes**

* Pipes are used for data transformation
* Types

. Pure - by default   
. Impure

* **Impure pipes**   
   a. Pipe will always transform even if data to be changed is not changed from previous
  1. Change detection will always run instead of data change or not

1. **Pure Pipes**
   1. If data has not changed since last time pipe was called, pipe won’t execute again
   2. Use to improve the performance because it will execute only when data is changed from previous run.
   3. Helps to reduce the change detection cycles
   4. We can make custom pipe as pure **with property pure: true**
   5. **Pure pipes check for object or array reference and does not do deep check.** So if any item added in array then it will not rerun the transform function as reference is not change. So it will not display updated array on ui. In this case **impure pipes**  are useful
2. Custom Pipes   
   e.g   
   import { Pipe, PipeTransform } from ‘@angular/core’;  
   @Pipe({  
    name: ‘filterByLength   
   })

export class implements PipeTransform {

transform(values: string[], minLength: number){  
 return values.filter((value) => value.length >= minLength)  
 }

}

.html file  
<ul>  
 <li \*ngFor="let value of values | filterByLength: 5">{{ value }}</li>  
 </ul>

1. Chaining the pipe

{{ myDate | date:’medium’ | upprcase }}

1. In built pipes

date   
uppercase  
async  
percent

1. Async pipe
2. Used to handle asynchronous data stream on html
3. Commonly use to subscribe to observables or promises
4. E.g  
   // component.ts  
   export class MyComponent {  
    myPromise$: Promise<string> = Promise.resolve('Hello World!');  
   }

// component.html  
<h2>Using AsyncPipe with Promise</h2>  
 <p>{{ myPromise$ | async }}</p>  
  
Output  
Using AsyncPipe with Promise  
Hello World!

1. **Data binding**

* One way and two way data binding
* String interpolation {{ propertyName }} – one way – component to view
* [(ngModel)]=”proptyName” – two way - <input type=’text’ [(ngModel)]=’name’ />
* Property binding - <app-child [dataToPass]=”dataPass”></app-child>  
  @Input() dataToPass: string = ‘dataPassed”

Event **binding** - <app-child [dataToPass]=”dataPass” (dataThatPassed)=”receivedData()”></app-child>  
@Output() dataThatPassed;  
  
clickToPassData(){  
this.dataThatPassed.emit(‘name’)  
}

1. Find most occurred character in string  
   function maxChar(str) {

const charMap = {};

let max = 0;

let maxChar = '';

// create character map

for (let char of str) {

if (charMap[char]) {

// increment the character's value if the character existed in the map

charMap[char]++;

} else {

// Otherwise, the value of the character will be increamented by 1

charMap[char] = 1;

}

}

// find the most commonly used character

for (let char in charMap) {

if (charMap[char] > max) {

max = charMap[char];

maxChar = char;

}

}

return maxChar;

}