ANGULAR 2, 4, 5, 6 INTERVIEW QUESTIONS FOR BEGINNERS TO EXPERIENCED

1) What is Angular?

Is a TypeScript-based open-source front-end web application platform bed by the Angular Team at Google and by a community of individuals and corporations to address all of the parts of the developer's workflow while building complex web application. Angular is a complete rewrite from the same team that built AngularJS. Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript. Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or the desktop.

2) What Is The Need Of Angular?

- Angular is not just a typical upgrade but a totally new development. The whole framework is rewritten from the ground. Angular got rid of many things like $scope, controllers, DDO, jqLite, angular.module etc.

- It uses components for almost everything. Imagine that even the whole app is now a component. Also it takes advantage of ES6 / TypeScript syntax. Developing Angular apps in TypeScript has made it even more powerful.

- Apart from that, many things have evolved and re-designed like the template engine and many more.

3) What are the advantages of using Angular over Angular JS?

- Angular was a ground-up rewrite of AngularJS and has many unique features.

- Angular execute run more than two programs at the same time.

- Angular JS is controllers and $scope based but Angular is component based.
- Angular does not have a concept of “scope” or controllers; instead it uses a hierarchy of components as its main architectural concept.
- Angular JS uses only JavaScript but Angular provides the possibility to use different languages like TypeScript, ES5, ES6, Dark etc.
- Angular JS was not built with mobile support in mind, where Angular is mobile oriented.
- Angular based on components, and it provides better performance than Angular JS.
- The use of dependency injection is enhanced in Angular.
- Angular has the flexible routing with lazy loading features.
- Angular is faster and Ahead of Time compilation (AOT) improves rendering speed.

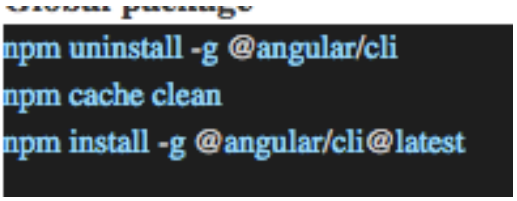
4) What Is Angular CLI? How To Updating Angular CLI?

The Angular CLI is a tool to initialize, develop, scaffold and maintain Angular applications.Using CLI , you can create a UNIT and END-TO-END test of the Angular application.

- To use Angular CLI, we need to install it first and it should be installed globally in your machine.

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

- If you're using Angular CLI lesser version, uninstall angular-cli package and install new versions.



5) What Is Bootstrapping in Angular?

main.ts is the entry point of your application, compiles the application with just-in-time and bootstrap the application.The Bootstrap is the **root** AppComponent that Angular creates and inserts into the "**index.html**" host web page.The bootstrapping process creates the components listed in the bootstrap array and inserts each one into the browser (**DOM**).
The *bootstrapping* process sets up the execution environment, digs
the *root* AppComponent out of the module's bootstrap array, creates an instance of the component and inserts it within the element tag identified by the component 's.selector.

- **Main.ts**

main.ts

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

import { AppModule } from './app/app.module';
import { environment } from
'./environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

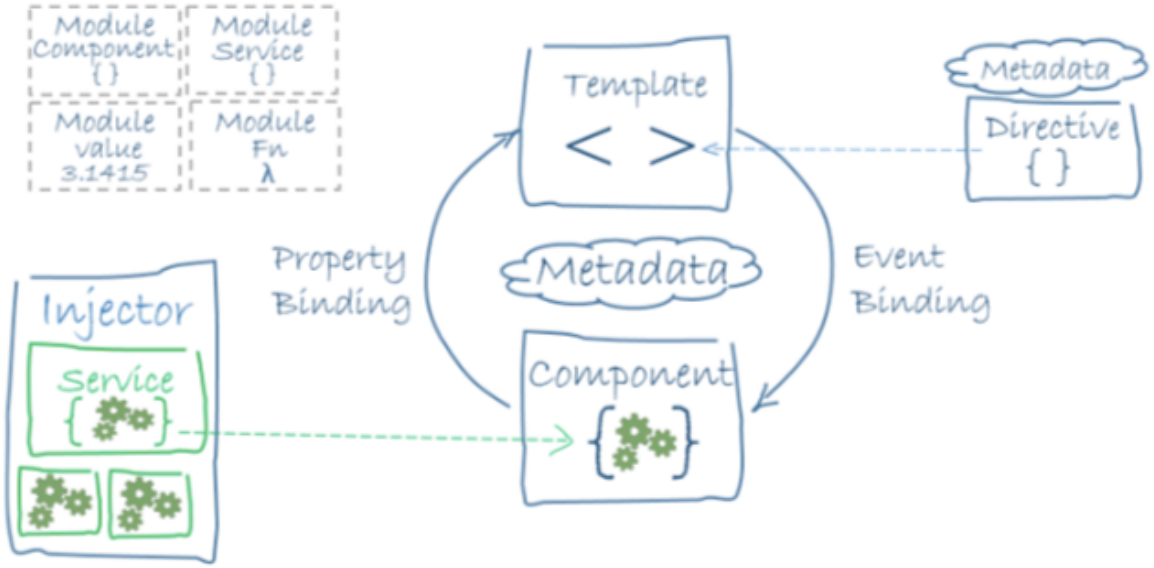
- **App.module.ts**

app.module.ts

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

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

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

Angular doesn't have a bootstrap directive. To launch the app in code, explicitly bootstrap the application's root module (AppModule) in main.ts and the application's root component (AppComponent) in app.module.ts.

6) What Is Architecture Overview of Angular?



Modules

- Angular apps are modular in nature.
- The angular application is nothing but collections of individual modules.
- Angular has its own modularity system called *NgModules*.
- Every Angular app has at least one NgModule class, the *root module*, conventionally named.AppModule
- An NgModule, whether a *root* or *feature*, is a class with a@NgModule decorator.

Components

- A *component* controls a patch of the screen called a *view*.
- You define a component's application logic—what it does to support the view—inside a class. The class interacts with the view through an API of properties and methods.
- Angular creates, updates, and destroys components as the user moves through the application.

- At least one component should be there called Root Component(app.component.ts)

Metadata

- Components have @component decorator , contains selector , template,templateUrl , style , styleUrls , providers.

- selector: CSS selector that tells Angular to create and insert an instance of this component where it finds a <hero-list>tag in *parent* HTML. For example, if an app's HTML contains,<hero-list></hero-list> then Angular inserts an instance of the view HeroListComponent between those tags.

- templateUrl: module-relative address of this component's HTML template, shown above.

- providers: an array of **dependency injection providers** for services that the component requires.

Templates

- You define a component's view with its companion **template**. A template is a form of HTML that tells Angular how to render the component.

- A template looks like regular HTML.

Data binding

- Angular supports **data binding**, a mechanism for coordinating parts of a template with parts of a component. Add binding markup to the template HTML to tell Angular how to connect both sides.

Directives

- Angular templates are *dynamic*. When Angular renders them, it transforms the DOM

according to the instructions given by **directives**.

- A directive is a class with a @Directive decorator.

Services

A class contains the Business Logic.

7) What is the flow of Angular App?

All thing in SRC folder and check index.html file(app root tag) , Main file is :-main.ts is convert into main.bundle.js file (in runtime), app.module.ts (bundle everything in it), In app.module.ts load app.component.ts. Details is given below:

- ng serve -> angular-cli.json -> index.html
- main.ts -> env.ts(Local/Prod) -> bootstrap
- app.module.ts(Angular Module) -> define
- app.component.html(bootstrap Root Component)

8) What's new in Angular 2?

- **Component Based :** It is entirely component based. It is not used to scope and controllers and Angular 2 are fully replaced by components and directives.
- **Directives :** The directive can be declared as @Directive annotation. A component is a directive with a template and the @Component decorator is actually a @Directive decorator extended with template oriented features.
- **Dependency Injection :** Dependency Injection is a powerful pattern for managing code dependencies. There are more opportunities for component and object based to improve the dependency injection.

- **Use of TypeScript :** Type represents the different types of values which are using in the programming languages and it checks the validity of the supplied values before they are manipulated by your programs.

- **Generics :** TypeScript has generics which can be used in the front-end development.

- **Lambdas and Arrow functions :** In the TypeScript, lambdas/ arrow functions are available. The arrow function is additional feature in typescript and it is also known as a lambda function.

- **Forms and Validations :** Angular 2 forms and validations are an important aspect of front-end development.

9) What's new in Angular 4?

Angular 4 contains some additional Enhancement and Improvement. Consider the following enhancements.

- **Smaller & Faster Apps** - Angular 4 applications is smaller & faster in comparison with Angular 2.

- **View Engine Size Reduce** - Some changes under to hood to what **AOT** generated code compilation that means in Angular 4, improved the compilation time. These changes reduce around 60% size in most cases.

- **Animation Package**- Animations now have their own package i.e. @angular/ platform-browser/animations.

- **Improvement** - Some Improvement on ***ngIf** and ***ngFor**.

- **Template** - The template is now **ng-template**. You should use the "ng-template" tag instead of "**template**". Now Angular has its own template tag that is called "ng-template".

- **NgIf with Else** – Now in Angular 4, possible to use an else syntax as,

```
<div *ngIf="user.length > 0; else empty"><h2>Users</h2></div>
<ng-template #empty><h2>No users.</h2></ng-template>
```

- **AS keyword** – A new addition to the template syntax is the "**as** keyword" is use to simplify to the "**let**" syntax.

Use of as keyword,

```
<div *ngFor="let user of users | slice:0:2 as total; index as = i">
   {{i+1}}/{{total.length}}: {{user.name}}
</div>
```

To subscribe only once to a pipe "|" with "**async**" and If a user is an observable, you can now use to write,

```
<div *ngIf="users | async as usersModel">
   <h2>{{ usersModel.name }}</h2> <small>{{ usersModel.age }}</small>
</div>
```

- **Pipes** - Angular 4 introduced a new "**titlecase**" pipe "|" and use to changes the first letter of each word into the uppercase.

The example as,

```
<h2>{{ 'anil singh' | titlecase }}</h2>
<!-- OUPPUT - It will display 'Anil Singh' -->
```

- **Http** - Adding search parameters to an "**HTTP** request" has been simplified as,

```
//Angular 4 -
http.get(`${baseUrl}/api/users`, { params: { sort: 'ascending' } });

//Angular 2-
const params = new URLSearchParams();
params.append('sort', 'ascending');
http.get(`${baseUrl}/api/users`, { search: params });
```

- **Test**- Angular 4, overriding a template in a test has also been simplified as,

```
//Angular 4 -
TestBed.overrideTemplate(UsersComponent, '<h2>{{users.name}}</h2>');

//Angular 2 -
TestBed.overrideComponent(UsersComponent, {
   set: { template: '<h2>{{users.name}}</h2>' }
});
```

- **Service**- A new service has been introduced to easily get or update "**Meta Tags**" i.e.

```
@Component({
   selector: 'users-app',
   template: `<h1>Users</h1>`
})
export class UsersAppComponent {
   constructor(meta: Meta) {
      meta.addTag({ name: 'Blogger', content: 'Anil Singh' });
   }
}
```

- **Forms Validators** - One new validator joins the existing "required", "minLength", "maxLength" and "pattern". An email helps you validate that the input is a valid email.

- **Compare Select Options** - A new "**compareWith**" directive has been added and it used to help you compare options from a select.

```
<select [compareWith]="byUId" [(ngModel)]="selectedUsers">
   <option *ngFor="let user of users" [ngValue]="user.UId">{{user.name}}</option>
</select>
```

- **Router** - A new interface "**paramMap**" and "**queryParamMap**" has been added and it introduced to represent the parameters of a URL.

```
const uid = this.route.snapshot.paramMap.get('Uid');
this.userService.get(uid).subscribe(user => this.name = name);
```

- **CanDeactivate** - This "**CanDeactivate**" interface now has an extra (optional) parameter and it is containing the next state.
- **I18n** - The internationalization is tiny improvement.

```
//Angular 4-
<div [ngPlural]="value">
   <ng-template ngPluralCase="0">there is nothing</ng-template>
   <ng-template ngPluralCase="1">there is one</ng-template>
</div>

//Angular 2-
<div [ngPlural]="value">
   <ng-template ngPluralCase="=0">there is nothing</ng-template>
   <ng-template ngPluralCase="=1">there is one</ng-template>
</div>
```

10) What's new in Angular 5?

The **Angular 5 Contains** bunch of new features, performance improvements and a lot of bug fixes and also some surprises to Angular lovers.

- Make AOT the default.
- Watch mode.
- Type checking in templates.
- More flexible metadata.
- Remove *.ngfactory.ts files.

- Better error messages.
- Smooth upgrades.
- Tree-Shakeable components.
- Hybrid Upgrade Application.
- Performance Improvements.

11) What's new in Angular 6?

Let's start to explore some important changes of Angular 6 is given below.

- **Added ng update -** This CLI commands will update your angular project dependencies to their latest versions. The ng update is normal package manager tools to identify and update other dependencies.

```
ng update
```

- **Angular 6 uses RxJS 6 -** this is the third-party library (RxJS) and introduces two important changes as compared to RxJS 5.

To update to RxJS 6, you simply run -

```
npm install --save rxjs@6
```

Simply run the blow command and update your existing Angular project-

```
npm install --save rxjs-compat
```

Alternatively, you can use the command - *ng update rxjs* to update *RxJS* and install the *rxjs-compat* package automatically.

- **RxJS 6 Related import paths:**

```
import { Observable } from 'rxjs/Observable';
import { Subject } from 'rxjs/Subject';
```

Use a single import -

```
import { Observable, Subject } from 'rxjs';
```

So all from *rxjs/Something* imports become from one *'rxjs'*

- **Operator imports have to change**

Instead of

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/throttle';
```

Now you can use -

```
import { map, throttle } from 'rxjs/operators';
```

And

Instead of

```
import 'rxjs/add/observable/of';
```

Now you can use -

```
import { of } from 'rxjs';
```

- **RxJS 6 Changes -** Changed Operator Usage

Instead of-

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/throttle';

yourObservable.map(data => data * 2)
.throttle(...)
.subscribe(...);
```

You can use the new pipe () method,

```
import { map, throttle } from 'rxjs/operators';

yourObservable
  .pipe(map(data => data * 2), throttle(...))
  .subscribe(...);
```

Angular 6 Renamed Operators - The lists of renamed operators are:

1. do() => tap()
2. catch() => catchError()
3. finally() => finalize()
4. switch() => switchAll()
5. throw() => throwError()
6. fromPromise() => from()

- **NgModelChange -** Now emitted after value and validity is updated on its control. Previously, it was emitted before updated. As the updated value of the control is available, the handler will become more powerful.

Previously -

```
<input [(ngModel)]="name" (ngModelChange)="onChange($event)">
```

And

```
onChange(value) {
 console.log(value);   // would log the updated value, not old value
}
```

Now Use -

```
<input #modelDir="ngModel" [(ngModel)]="name" (ngModelChange)="onChange(modelDir)">
```

And

```
onChange(NgModel: NgModel) {
 console.log(NgModel.value);// would log old value, not updated value
}
```

- **Form Control statusChanges –** Angular 6 emits an event of "PENDING" when we call Abstract Control markAsPending.

- **New optional generic type ElementRef –** This optional generic type will help to get hold of the native element of given custom Element as ElementRef Type.

12) Explain tsconfig.json file?

The tsconfig.json file corresponds to the configuration of the TypeScript compiler (tsc).

```
{
   "compilerOptions": {
      "target": "es5",
      "module": "commonjs",
      "moduleResolution": "node",
      "sourceMap": true,
      "emitDecoratorMetadata": true,
      "experimentalDecorators": true,
      "lib": [ "es2015", "dom" ],
      "noImplicitAny": true,
      "suppressImplicitAnyIndexErrors": true
   }
}
```

- **target** : the language used for the compiled output.
- **module** : the module manager used in the compiled output. system is for SystemJS, commonjs for CommonJS.
- **moduleResolution** : the strategy used to resolve module declaration files (.d.ts files). With the node approach, they are loaded from the node_modules folder like a module (require('module-name'))
- **sourceMap** : generate or not source map files to debug directly your application TypeScript files in the browser,
- **emitDecoratorMetadata** : emit or not design-type metadata for decorated declarations in a source,
- **experimentalDecorators** : enables or not experimental support for ES7 decorators,
- **removeComments** : remove comments or not.
- **noImplicitAny** : allow or not the use of variables/parameters without types (implicit).

13) Explain package.json file?

All npm packages contain a file, usually in the project root, called package.json – this file holds various metadata relevant to the project.

- This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies.
- It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data – all of which can be vital to both npm and to the end users of the package.

- The package.json file is normally located at the root directory of a Node.js project.

14) Explain systemjs.config.json file?

system.config.js is the one which allows to load modules(node modules) compiled using the TypeScript compiler.map refers to the name of modules to JS file that contains the JavaScript code.

15) What is Module in Angular?

In Angular, Module allows to put logical boundaries in our application. It is a fundamental feature of Angular that groups related components, directives, pipe and services together. Every Angular application has at least one module, the root module, conventionally named AppModule. Some important features like lazy loading are done at the Angular Module level.

16) What is Component in Angular?

In Angular application everything is component. Components are a logical piece of code for Angular JS application. Each component is mainly used to build HTML elements and provides logical boundary of functionality for the Angular application. The components encapsulate all the logic, allowing you to reuse them across your application. A Component consists of the following:

- Template : It is used to render the view for the application.
- Class : It is like any class of c and c++
- Metadata : It has the extra data defined for the Angular class. It is defined with a decorator.

Example:

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

@Component ({
  selector: 'Home',
  template: ` <div>
    <h1>{{homeTitle}}</h1>
    <div>To Home Page</div>
  </div> `,
})

export class HomeComponent {
  homeTitle: string = 'Welcome';
}
```

17) Explain the life cycle hooks of the Angular?

The lifecycle events of Angular component/directive are managed by @angular/core. It creates the component and renders it, processes changes when it's data-bound properties change, and then destroys it before removing its template from the DOM. Angular has a set of lifecycle events. These events can be tapped and perform operations when required. The constructor executes prior to all lifecycle events. The constructor is meant for light weight activities and usually used for dependency injection. For example:

- When component is initialized, Angular invokes ngOnInit.
- When a component's input properties change, Angular invokes ngOnChanges.
- When a component is destroyed, Angular invokes ngOnDestroy.

18) What is the possible order of lifecycle hooks in Angular?

The Angular life cycle hooks are nothing but callback function, which angular invokes when a certain event occurs during the component's life cycle. Here is the complete list of life cycle hooks provided by the Angular.

- **ngOnChanges** – When the value of a data bound property changes, then this method is called.
- **ngOnInit** – This is called whenever the initialization of the directive/component after Angular first displays the data-bound properties happens.
- **ngDoCheck** – This is for the detection and to act on changes that Angular can't or won't detect on its own.
- **ngAfterContentInit** – This is called in response after Angular projects external content into the component's view.
- **ngAfterContentChecked** – This is called in response after Angular checks the content projected into the component.
- **ngAfterViewInit** – This is called in response after Angular initializes the component's views and child views.
- **ngAfterViewChecked** – This is called in response after Angular checks the component's views and child views.
- **ngOnDestroy** – This is the cleanup phase just before Angular destroys the directive/component.

Parent Component:

```
import { Component, AfterContentChecked, AfterContentInit, AfterViewChecked, AfterViewInit ,
  DoCheck, OnChanges, OnDestroy, OnInit,  Input, SimpleChanges  } from '@angular/core';


@Component({
  selector: 'app-root',
  template: `
      <h2>Life Cycle Hook</h2>
      <button (click)="toggle()">Hide/Show Child </button>
      <child-component *ngIf="displayChild"  [message]="'Hello'" ></child-component>
      ` ,
  styleUrls: ['./app.component.css']
})
```

```
export class AppComponent implements OnChanges, OnInit, DoCheck,
AfterContentInit, AfterContentChecked,AfterViewInit, AfterViewChecked,OnDestroy {

  displayChild: boolean=false;

  constructor() {
    console.log("AppComponent:Constructor");
  }

  toggle() {
    this.displayChild=!this.displayChild;
  }
```

```
  ngOnChanges() {
    console.log("AppComponent:OnChanges");
  }


  ngOnInit() {
    console.log("AppComponent:OnInit");
  }

  ngDoCheck() {
    console.log("AppComponent:DoCheck");
  }

   ngAfterContentInit() {
    console.log("AppComponent:AfterContentInit");
  }
```

```
  ngAfterContentChecked() {
    console.log("AppComponent:AfterContentChecked");
  }

  ngAfterViewInit() {
    console.log("AppComponent:AfterViewInit");
  }

  ngAfterViewChecked() {
    console.log("AppComponent:AfterViewChecked");
  }

  ngOnDestroy() {
    console.log("AppComponent:OnDestroy");
  }
}
```

- We are listening to all the hooks.
- Child Component can be added/removed using the ngIf directive.
- We are passing data to child component by binding to @Input property of the child component.

Child Component:

```
import { Component, AfterContentChecked, AfterContentInit, AfterViewChecked, AfterViewInit ,
         DoCheck, OnChanges, OnDestroy, OnInit,  Input, SimpleChanges  } from '@angular/core';

@Component({
  selector: 'child-component',
  template: `
      <h2>Child Component</h2>
      ` ,
  styleUrls: ['./app.component.css']
})
```

```
export class ChildComponent implements OnChanges, OnInit, DoCheck,
AfterContentInit, AfterContentChecked,AfterViewInit, AfterViewChecked,OnDestroy {

  @Input() message="";

  constructor() {
    console.log("ChildComponent:Constructor");
  }
```

```
  ngOnChanges() {
    console.log("ChildComponent:OnChanges");
  }


  ngOnInit() {
    console.log("ChildComponent:OnInit");
  }

  ngDoCheck() {
    console.log("ChildComponent:DoCheck");
  }

   ngAfterContentInit() {
    console.log("ChildComponent:AfterContentInit");
  }
```

```
  ngAfterContentChecked() {
    console.log("ChildComponent:AfterContentChecked");
  }

  ngAfterViewInit() {
    console.log("ChildComponent:AfterViewInit");
  }

  ngAfterViewChecked() {
    console.log("ChildComponent:AfterViewChecked");
  }

  ngOnDestroy() {
    console.log("ChildComponent:OnDestroy");
  }

}
```

- We are listening to all the hooks.
- @Input property message is defined.

19) What are the differences between Constructors and ngOnInit?

Constructors:

- The constructor is a default method runs when component is being constructed.
- The constructor is a typescript feature and it is used only for a class instantiations and nothing to do with Angular.
- The constructor called first time before the ngOnInit().

ngOnInit:

- The ngOnInit event is an Angular life-cycle event method that is called after the first ngOnChanges and the ngOnInit method is use to parameters defined with @Input otherwise the constructor is OK.
- The ngOnInit is called after the constructor and ngOnInit is called after the first ngOnChanges.
- The ngOnChanges is called when an input or output binding value changes.

Example:

```
import {Component, OnInit} from '@angular/core';
export class App implements OnInit{
    constructor(){
    }
    ngOnInit(){
    }
}
```

20) When will ngOnInit be called?

The ngOnInit or OnInit hook is called when the component is created for the first time. This hook is called after the constructor and first ngOnChanges hook is fired. This is a perfect place where you want to add any initialization logic for your component. Note that ngOnChanges hook is fired before ngOnInit. Which means all the input properties are available to use when the ngOnInit is hook is called. This hook is fired only once and hook is fired before any of the child directive properties are initialized.

21) How would you make use of ngOnInit and ngOnDestroy?

ngOnInit can be used for following purposes.

- Perform complex initialization in ngOnInit() and not in constructor.
- If we need to fetch data then it should be done in ngOnInit() and not in constructor so that we should not worry while initializing component. A constructor should perform only local variable initialization.

ngOnDestroy can be used for following purposes.

- Stop interval timers.

- Unsubscribe Observables.
- Detach event handlers.
- Free resources that will not be garbage collected automatically.
- Unregister all callbacks.

Example of ngOnInit and ngOnDestroy using directive:

- Here we will provide the use of OnInit and OnDestroy using directive. We will create service for logging that will be LoggerService. In directive when ngOnInit() and ngOnDestroy() will be called then it will set a log using LoggerService.

cp.directive.ts

```
import { Directive, OnInit, OnDestroy, Input } from '@angular/core';
import { LoggerService } from './logger.service';
import { Log } from './log';

@Directive({
        selector: '[cp]'
})
```

```
export class CPDirective implements OnInit, OnDestroy {
  @Input('cp')
  personName: string;
  constructor(private loggerService: LoggerService) {}
  ngOnInit() {
        this.loggerService.createCP2Log(new Log('c', this.personName + ' is created.'));
  }
  ngOnDestroy() {
        this.loggerService.createCP2Log(new Log('r', this.personName + ' is removed.'));
  }
}
```

log.ts

```
export class Log {
        constructor(public logType: string, public message: string) {}
}
```

logger.service.ts

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

@Injectable()
export class LoggerService {
   private allCP2Logs: Log[] = [];
   private cp1Log = new Log('', '');
   createCP2Log(log: Log) {
           this.allCP2Logs.push(log);
   }

   getAllCP2Logs() {
           return this.allCP2Logs;
   }
   setCP1Log(logType: string, message: string) {
           this.cp1Log.logType = logType;
           this.cp1Log.message = message;
   }
   getCP1Log() {
           return this.cp1Log;
   }
}
```

- We will add and delete persons in our demo and spy them using directive. The array of persons will be iterated using ngFor. For each and every row, our directive cp will spy the addition and deletion of persons. We will pass person name to directive. When person will be added, ngOnInit() of directive will be called and when person

will be removed , ngOnDestroy() of directive will be called. We will also display logs of addition and removal of persons.

cp2.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Person } from './person';
import { LoggerService } from './logger.service';
import { Log } from './log';

@Component({
    selector: 'app-cp2',
    templateUrl: './cp2.component.html'
})
export class CP2Component implements OnInit {
  persons: Person[] = [];
  name: string;
  allLogs: Log[] = [];
  constructor(private loggerService: LoggerService) {}
  ngOnInit() {
        this.allLogs = this.loggerService.getAllCP2Logs();
  }
```

```
add() {
      let personId = 0;
      let maxIndex = this.persons.length - 1;
      if (maxIndex === -1) {
            personId = 1;
      } else {
            let personWithMaxIndex = this.persons[maxIndex];
            personId = personWithMaxIndex.id + 1;
      }
      this.persons.push(new Person(personId, this.name));
      this.name = '';
}
```

```
  remove(personId: number) {
        let item = this.persons.find(ob => (ob.id === personId));
        let itemIndex = this.persons.indexOf(item);

        this.persons.splice(itemIndex, 1);
  }
}
```

cp2.component.html

```
<div>
 <p>Add New Person <input [(ngModel)]="name">
 <button type="button" (click)="add()">Add</button> </p>
</div>

<div *ngFor="let p of persons" [cp]="p.name">
    {{p.name}} |
    <font color='blue'><a href="javascript:void" (click)="remove(p.id)"> Remove </a></font>
</div>
```

```
<div>
 <br/><b>--- Logging ---</b>
 <ng-template ngFor let-log [ngForOf]= "allLogs">
    <ng-template [ngIf]= "log.logType === 'c' " >
           <br/><font color= 'green'>{{log.message}}</font>
    </ng-template>
    <ng-template [ngIf]= "log.logType === 'r' " style='color: red'>
           <br/><font color= 'red'>{{log.message}}</font>
    </ng-template>
 </ng-template>
</div>
```

person.ts

```
export class Person {
        constructor(public id: number, public name: string) {}
}
```

app.component.html

```
<h3>OnInit and OnDestroy using Directive</h3>
<app-cp2></app-cp2>
```

22) How would you make use of ngOnChanges and SimpleChanges?

- **ngOnChanges :** The ngOnChanges is a life cycle hook, which angular fires when it detects changes to data bound input property. This method receives a SimpleChanges object, which contains the current and previous property values. The ngOnChanges() method takes an object that maps each changed property name to a SimpleChange object, which holds the current and previous property values. You can iterate over the changed properties and act upon it.

- **SimpleChanges :** SimpleChange class represents a basic change from a previous to new value. It has following class members. Suppose there is a change in input value, then following property can be used to detect changes. SimpleChanges is the interface

that represents the changes object for all input property. SimpleChanges has the key as input property names and values are the instances of SimpleChange class.

PROPERTY NAME	DESCRIPTION
previousValue:any	Previous value of the input property.
currentValue:any	New or current value of the input property.
FirstChange():boolean	Boolean value, which tells us whether it was the first time the change has taken place

Example of ngOnChanges and SimpleChanges:

- Create a class customer.ts under src/app folder.

```
export class Customer {
  code: number;
  name: string;
}
```

- We have 3 user input fields for the message , code, and name. The UpdateCustomer button updates the Customer object.
- The message and Customer is bound to the child component using the @Import annotation.
- The AppComponent class has a message & customer property. The Customer property is updated with new code & name when the updateCustomer button is clicked.

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

@Component({
  selector: 'app-root',
  template: `
        <h1>{{title}}!</h1>
        <p> Message : <input type='text' [(ngModel)]='message'> </p>
        <p> Code : <input type='text' [(ngModel)]='code'></p>
        <p> Name : <input type='text' [(ngModel)]='name'></p>
        <p><button (click)="updateCustomer()">Update </button>
        <child-component [message]=message [customer]=customer></child-component>
        ` ,

        styleUrls: ['./app.component.css']
})
```

```
export class AppComponent {
  title = 'ngOnChanges';
  message = '';
  customer: Customer = new Customer();
  name= '';
  code= 0;

  updateCustomer() {
    this.customer= new Customer();
    this.customer.name = this.name;
    this.customer.code = this.code;
  }

}
```

- First, We imported the Input, OnInit, OnChanges, SimpleChanges, SimpleChange from Angular Core.
- The Template displays the message and the code & name property from the customer object. Both these properties are updated from the parent component.
- The child Component implements the OnChanges & OnInit life cycle hooks.
- The ngOnChanges hook gets all the Changes as an instance of SimpleChanges. This

object contains the instance of SimpleChange for each property.

- We, then loop through each property of the SimpleChanges object and get a reference to the SimpleChange object.
- Next, we will take the current & previous value of each property and add it to change log.
- Now our OnChanges hook is ready to use.

```
import { Component, Input, OnInit, OnChanges, SimpleChanges, SimpleChange,ChangeDetectionStrategy  }
import { Customer } from './customer';

@Component({
    selector: 'child-component',
    template: `<h2>Child  Component</h2>
               <p>Message {{ message }} </p>
               <p>Customer Name {{ customer.name }} </p>
               <p>Customer Code {{ customer.code }} <p>
               <ul><li *ngFor="let log of changelog;"> {{ log }}</li></ul> `
})
```

```
export class ChildComponent implements OnChanges, OnInit {
    @Input() message: string;
    @Input() customer: Customer;
    changelog: string[] = [];

    ngOnInit() {
        console.log('OnInit');
    }

    ngOnChanges(changes: SimpleChanges) {
        console.log('OnChanges');
        console.log(JSON.stringify(changes));
```

```
        // tslint:disable-next-line:forin
        for (const propName in changes) {
             const change = changes[propName];
             const to   = JSON.stringify(change.currentValue);
             const from = JSON.stringify(change.previousValue);
             const changeLog = `${propName}: changed from ${from} to ${to} `;
             this.changelog.push(changeLog);
        }
    }
}
```

- Now, run the code and type the Hello and you will see the following log.

- Open the developer console and you should see the changes object. Note that the first OnChanges fired before the OnInit hook. This ensures that initial values bound to inputs are available when ngOnInit() is called.

```
message: changed from undefined to ""
customer: changed from undefined to {}
message: changed from "" to "H"
message: changed from "H" to "He"
message: changed from "He" to "Hel"
message: changed from "Hel" to "Hell"
message: changed from "Hell" to "Hello"
```

23) What is Pipes?

Pipes transform displayed values within a template. Sometimes, the data is not displays in the well format on the template that time where using pipes. You also can execute a function in the template to get its returned value. Introducing Angular pipes, a way to write display-value transformations that you can declare in your HTML. Example :

- A pipe takes in data as input and transforms it to a desired output. In this page, you'll use pipes to transform a component's birthday property into a human-friendly date:

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

@Component({
  selector: 'app-hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988
}
```

- Inside the interpolation expression, you flow the component's birthday value through the pipe operator (|) to the Date pipe function on the right. All pipes work this way:

```
<p>The hero's birthday is {{ birthday | date }}</p>
```

24) Why use Pipes?

Sometimes, the data is not displays in the correct format on the template that time where using pipes. If you want to display the bank card number on your account detail templates that how to displays this card number? I think you should display the last four digits and rest of all digits will display as encrypted like (****-****-****_and your card numbers) that time you will need to create a custom pipe to achieve this:

25) What Are Inbuilt Pipes in Angular?

Angular comes with a stock of inbuilt pipes. They are all available for use in any template.

- **UpperCase & LowerCase Pipe :** Transforms text to all upper and lower case.

```
@Component({
  selector: 'lowerupper-pipe',
  template: `<div>
    <label>Name: </label><input #name (keyup)="change(name.value)" type="text">
    <p>In lowercase: <pre>'{{value | lowercase}}'</pre>
    <p>In uppercase: <pre>'{{value | uppercase}}'</pre>
  </div>`
})
```

```
export class LowerUpperPipeComponent {
  // TODO(issue/24571): remove '!'.
  value !: string;
  change(value: string) { this.value = value; }
}
```

- **TitleCasePipe :** Transforms text to title case. Capitalizes the first letter of each word, and transforms the rest of the word to lower case. Words are delimited by any whitespace character, such as a space, tab, or line-feed character.

```
@Component({
  selector: 'titlecase-pipe',
  template: `<div>
    <p>{{'some string' | titlecase}}</p> <!-- output is expected to be "Some String" -->
    <p>{{'tHIs is mIXeD CaSe' | titlecase}}</p> <!-- output is expected to be "This Is Mixed Case" -->
    <p>{{'it\\'s non-trivial question' | titlecase}}</p> <!-- output is expected to be "It's Non-trivial
Question" -->
```

```
    <p>{{'one,two,three' | titlecase}}</p> <!-- output is expected to be "One,two,three" -->
    <p>{{'true|false' | titlecase}}</p> <!-- output is expected to be "True|false" -->
    <p>{{'foo-vs-bar' | titlecase}}</p> <!-- output is expected to be "Foo-vs-bar" -->
  </div>`
})
export class TitleCasePipeComponent {
}
```

- **AsyncPipe :** Angular provide a special kind of pipe that are called AsyncPipe and

the AsyncPipe subscribes to an observable or promise and returns the latest value it has emitted. The AsyncPipe allows you to bind your HTML templates directly to values that arrive asynchronously manner that is a great ability for the promises and observables.

- This example binds a Promise to the view. Clicking the Resolve button resolves the promise:

```
@Component({
  selector: 'async-promise-pipe',
  template: `<div>
    <code>promise|async</code>:
    <button (click)="clicked()">{{ arrived ? 'Reset' : 'Resolve' }}</button>
    <span>Wait for it... {{ greeting | async }}</span>
  </div>`
})
```

```
export class AsyncPromisePipeComponent {
  greeting: Promise<string>|null = null;
  arrived: boolean = false;

  private resolve: Function|null = null;

  constructor() { this.reset(); }
```

```
reset() {
  this.arrived = false;
  this.greeting = new Promise<string>((resolve, reject) => { this.resolve = resolve; });
}
```

```
  clicked() {
    if (this.arrived) {
      this.reset();
    } else {
      this.resolve !('hi there!');
      this.arrived = true;
    }
  }
}
```

- It's also possible to use async with Observables. The example below binds the time Observable to the view. The Observable continuously updates the view with the current time:

```
@Component({
  selector: 'async-observable-pipe',
  template: '<div><code>observable|async</code>: Time: {{ time | async }}</div>'
})
export class AsyncObservablePipeComponent {
  time = new Observable<string>((observer: Observer<string>) => {
    setInterval(() => observer.next(new Date().toString()), 1000);
  });
}
```

- **CurrencyPipe :** Transforms a number to a currency string, formatted according to locale rules that determine group sizing and separator, decimal-point character, and other locale-specific configurations.

```
@Component({
  selector: 'currency-pipe',
  template: `<div>
    <!--output '$0.26'-->
    <p>A: {{a | currency}}</p>

    <!--output 'CA$0.26'-->
    <p>A: {{a | currency:'CAD'}}</p>

    <!--output 'CAD0.26'-->
    <p>A: {{a | currency:'CAD':'code'}}</p>
```

```
<!--output 'CA$0,001.35'-->
<p>B: {{b | currency:'CAD':'symbol':'4.2-2'}}</p>

<!--output '$0,001.35'-->
<p>B: {{b | currency:'CAD':'symbol-narrow':'4.2-2'}}</p>

<!--output '0 001,35 CA$'-->
<p>B: {{b | currency:'CAD':'symbol':'4.2-2':'fr'}}</p>
```

```
    <!--output 'CLP1' because CLP has no cents-->
    <p>B: {{b | currency:'CLP'}}</p>
  </div>`
})
export class CurrencyPipeComponent {
  a: number = 0.259;
  b: number = 1.3495;
}
```

- **DatePipe :** Formats a date value according to locale rules.

```
@Component({
 selector: 'date-pipe',
 template: `<div>
   <p>Today is {{today | date}}</p>
   <p>Or if you prefer, {{today | date:'fullDate'}}</p>
   <p>The time is {{today | date:'h:mm a z'}}</p>
 </div>`
})
// Get the current date and time as a date-time value.
export class DatePipeComponent {
  today: number = Date.now();
}
```

- **PercentPipe :** Transforms a number to a percentage string, formatted according to locale rules that determine group sizing and separator, decimal-point character, and other locale-specific configurations.

```
@Component({
  selector: 'percent-pipe',
  template: `<div>
    <!--output '26%'-->
    <p>A: {{a | percent}}</p>

    <!--output '0,134.950%'-->
    <p>B: {{b | percent:'4.3-5'}}</p>
```

```
    <!--output '0 134,950 %'-->
    <p>B: {{b | percent:'4.3-5':'fr'}}</p>
  </div>`
})
export class PercentPipeComponent {
  a: number = 0.259;
  b: number = 1.3495;
}
```

- **JsonPipe :** Converts a value into its JSON-format representation. Useful for debugging.

```
@Component({
  selector: 'json-pipe',
  template: `<div>
    <p>Without JSON pipe:</p>
    <pre>{{object}}</pre>
    <p>With JSON pipe:</p>
    <pre>{{object | json}}</pre>
  </div>`
})
export class JsonPipeComponent {
  object: Object = {foo: 'bar', baz: 'qux', nested: {xyz: 3, numbers: [1, 2, 3, 4, 5]}};
}
```

- **DecimalPipe :** Transforms a number into a string, formatted according to locale rules that determine group sizing and separator, decimal-point character, and other locale-specific configurations.

```
@Component({
  selector: 'number-pipe',
  template: `<div>
    <!--output '2.718'-->
    <p>e (no formatting): {{e | number}}</p>

    <!--output '002.71828'-->
    <p>e (3.1-5): {{e | number:'3.1-5'}}</p>

    <!--output '0,002.71828'-->
    <p>e (4.5-5): {{e | number:'4.5-5'}}</p>
```

```
    <!--output '003.14000'-->
    <p>pi (3.5-5): {{pi | number:'3.5-5'}}</p>

    <!--output '-3' / unlike '-2' by Math.round()-->
    <p>-2.5 (1.0-0): {{-2.5 | number:'1.0-0'}}</p>
  </div>`
})
export class NumberPipeComponent {
  pi: number = 3.14;
  e: number = 2.718281828459045;
}
```

- **SlicePipe :** Creates a new Array or String containing a subset (slice) of the elements.

```
@Component({
  selector: 'slice-string-pipe',
  template: `<div>
    <p>{{str}}[0:4]: '{{str | slice:0:4}}' - output is expected to be 'abcd'</p>
    <p>{{str}}[4:0]: '{{str | slice:4:0}}' - output is expected to be ''</p>
    <p>{{str}}[-4]: '{{str | slice:-4}}' - output is expected to be 'ghij'</p>
    <p>{{str}}[-4:-2]: '{{str | slice:-4:-2}}' - output is expected to be 'gh'</p>
    <p>{{str}}[-100]: '{{str | slice:-100}}' - output is expected to be 'abcdefghij'</p>
    <p>{{str}}[100]: '{{str | slice:100}}' - output is expected to be ''</p>
  </div>`
```

```
})
export class SlicePipeStringComponent {
  str: string = 'abcdefghij';
}
```

26) What is a pure pipe?

A pipe that is only executed when Angular detects a pure change to the input value (e.g. new primitive object or new object reference).

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

@Pipe({
 name: 'currency'
})
export class CurrencyPipe implements PipeTransform {

 transform(value: any, args?: any): any {
  if (!value) {
   return '1.00';
  }

  return value;
 }
```

27) What is an impure pipe?

A pipe that is executed during every component change detection cycle (i.e., often – every keystroke, mouse move).

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

@Pipe({
 name: 'currency',
 pure:false
})
export class CurrencyPipe implements PipeTransform {

 transform(value: any, args?: any): any {
  if (!value) {
   return '1.00';
  }

  return value;
 }
}
```

28) What is parameterizing a pipe?

A pipe can accept any number of optional parameters to fine-tune its output. To add parameters to a pipe, follow the pipe name with a colon (:) and then the parameter value (such as currency:'EUR'). If the pipe accepts multiple parameters, separate the values with colons (such as slice:1:5).

- **Example-1 :** Modify the birthday template to give the date pipe a format parameter. After formatting the hero's April 15th birthday, it renders as 04/15/88:

```
<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>
```

- **Example-2 :** Write a component that *binds* the pipe's format parameter to the

component's format property. Here's the template for the component and add a button to the template and bound its click event to the component's toggleFormat() method. That method toggles the component's format property between a short form ('shortDate') and a longer form ('fullDate'):

```
template: `
  <p>The hero's birthday is {{ birthday | date:format }}</p>
  <button (click)="toggleFormat()">Toggle Format</button>
`
```

```
export class HeroBirthday2Component {
  birthday = new Date(1988, 3, 15); // April 15, 1988
  toggle = true; // start with true == shortDate

  get format()   { return this.toggle ? 'shortDate' : 'fullDate'; }
  toggleFormat() { this.toggle = !this.toggle; }
}
```

29) What is Chaining pipes?

You can chain pipes together in potentially useful combinations. In the following example, to display the birthday in uppercase, the birthday is chained to the DatePipe and on to the UpperCasePipe. The birthday displays as APR 15, 1988.

```
The chained hero's birthday is
{{ birthday | date | uppercase}}
```

30) How to create a custom Pipes?

You can write your own custom pipes. Here's a custom pipe named ExponentialStrengthPipe that can boost a hero's powers:

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
*/
```

```
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

Definition reveals the following key points:

- A pipe is a class decorated with pipe metadata.
- The pipe class implements the PipeTransform interface's transform method that accepts an input value followed by optional parameters and returns the transformed value.
- There will be one additional argument to the transform method for each parameter passed to the pipe. Your pipe has one such parameter: the exponent.
- To tell Angular that this is a pipe, you apply the @Pipe decorator, which you import from the core Angular library.
- The @Pipe decorator allows you to define the pipe name that you'll use within

template expressions. It must be a valid JavaScript identifier. Your pipe's name is exponentialStrength.

Example : It's not much fun updating the template to test the custom pipe. Upgrade the example to a "Power Boost Calculator" that combines your pipe and two-way data binding with ngModel:

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

@Component({
  selector: 'app-power-boost-calculator',
  template: `
    <h2>Power Boost Calculator</h2>
    <div>Normal power: <input [(ngModel)]="power"></div>
    <div>Boost factor: <input [(ngModel)]="factor"></div>
    <p>
      Super Hero Power: {{power | exponentialStrength: factor}}
    </p>
```

```
})
export class PowerBoostCalculatorComponent {
  power = 5;
  factor = 1;
}
```

31) What is an Angular services?

Services are commonly used for storing data and making HTTP calls. The main idea behind a service is to provide an easy way to share the data between the components and with the help of dependency injection (DI) you can control how the service instances are shared. Services use to fetch the data from the RESTful API.

32) What are the features of Angular service?

Following are the features of services in Angular:

- Services are singleton object i.e. only one instance of service exists throughout the application.
- Services are capable of returning the data in the form promises or observables.
- Service class is decorated with @Injectable() decorator.

33) How to create and call Angular service?

- To create a service, we need to make use of the command line. The command for the same is:

```
C:\projectA6\Angular6App>ng g service myservice
CREATE src/app/myservice.service.spec.ts (392 bytes)
CREATE src/app/myservice.service.ts (138 bytes)
```

- Following are the files created **myservice.service.specs.ts** and **myservice.service.ts.** Here, the Injectable module is imported from the @angular/core. It contains the @Injectable method and a class called **MyserviceService**:

myservice.service.ts

```
import { Injectable } from '@angular/core';
@Injectable()
export class MyserviceService {
   constructor() { }
}
```

- We need to include the service created in the main parent **app.module.ts**. We have imported the Service with the class name and the same class is used in the providers:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { RouterModule} from '@angular/router';
import { AppComponent } from './app.component';
import { MyserviceService } from './myservice.service';
import { NewCmpComponent } from './new-cmp/new-cmp.component';
import { ChangeTextDirective } from './change-text.directive';
import { SqrtPipe } from './app.sqrt';
@NgModule({
   declarations: [
      SqrtPipe,
      AppComponent,
      NewCmpComponent,
      ChangeTextDirective
   ],
   imports: [
```

```
      BrowserModule,
      RouterModule.forRoot([
         {
            path: 'new-cmp',
            component: NewCmpComponent
         }
      ])
   ],
   providers: [MyserviceService],
   bootstrap: [AppComponent]
})
export class AppModule { }
```

- In the service class, we will create a function which will display current date. We have created a function **showTodayDate**. Now we will return the new Date () created:

```
import { Injectable } from '@angular/core';
@Injectable()
export class MyserviceService {
   constructor() { }
   showTodayDate() {
      let ndate = new Date();
      return ndate;
   }
}
```

- We inject this service into component using constructor and access a function in the

component class. The **ngOnInit** function gets called by default in any component created. The date is fetched from the service:

app.component.ts

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

```
export class AppComponent {
   title = 'Angular 6 Project!';
   todaydate;
   constructor(private myservice: MyserviceService) {}
   ngOnInit() {
      this.todaydate = this.myservice.showTodayDate();
   }
}
```

- We will display the date in the **.html** file as shown below:

```
{{todaydate}}
<app-new-cmp></app-new-cmp>
// data to be displayed to user from the new component class.
```

34) How are the services injected to your application?

Services injected in application Via Angular's DI (Dependency Injection) mechanism.

- **Inject Services using Constructor :** Once we have specified providers for services then we can inject those services into components and other services using constructor:

```
@Component({
    selector: 'store-app',
    templateUrl: './store.component.html'
})
export class StoreComponent {
    constructor(private itemService: ItemService) { }
}
```

35) How to configure providers for Angular service?

- **@Injectable-level :** The @Injectable() decorator identifies every service class. The providedIn metadata option for a service class configures a specific injector (typically root) to use the decorated class as a provider of the service. When an injectable class provides its own service to the root injector, the service is available anywhere the class is imported:

src/app/heroes/hero.service.ts

```
import { Injectable } from '@angular/core';
import { HeroModule } from './hero.module';
import { HEROES } from './mock-heroes';

@Injectable({
  // we declare that this service should be created
  // by any injector that includes HeroModule.
  providedIn: HeroModule,
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

- **@NgModule-level :** You can configure a provider at the module level using the providedIn metadata option for a non-root NgModule, in order to limit the scope

of the provider to that module. This is the equivalent of specifying the non-root module in the @Injectable() metadata, except that the service provided this way is not tree-shakable. You generally don't need to specify AppModule with providedIn, because the app's root injector is the AppModule injector:

src/app/app.module.ts (providers)

```
providers: [
  { provide: LocationStrategy, useClass: HashLocationStrategy }
]
```

@Component-level : Individual components within an NgModule have their own injectors. You can limit the scope of a provider to a component and its children by configuring the provider at the component level using the @Component metadata:

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

import { HeroService } from './hero.service';
```

```
@Component({
  selector: 'app-heroes',
  providers: [ HeroService ],
  template: `
    <h2>Heroes</h2>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent { }
```

36) What is the use case of services?

One very common use case is providing data to components, often by fetching it from a server. Though there's no real definition of service from Angular point of view – it could do

almost anything (e.g., logging is another common use case, among many).

37) Why is it a bad idea to create a new service in a component like the one below?

- let service = new DataService();
- The object may not be created with its needed dependencies.

38) How to make sure that a single instance of a service will be used in an entire application?

Provide it in the root module.

39) Why do we need provider aliases? And how do you create one?

To substitute an alternative implementation for a provider. Can create like so:

- { provide: LoggerService, useClass: DateLoggerService }

40) What is the way to inject one service into another in angular?

Service can have their own dependencies. HeroService is very simple and doesn't have any dependencies of its own. Suppose, however, that you want it to report its activities through a logging service. You can apply the same *constructor injection* pattern, adding a constructor that takes a Logger parameter.

- **LoggerService:**

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

@Injectable({
  providedIn: 'root'
})
export class Logger {
  logs: string[] = []; // capture logs for testing

  log(message: string) {
    this.logs.push(message);
    console.log(message);
  }
}
```

- **HeroService:**

```
import { Injectable } from '@angular/core';
import { HEROES }     from './mock-heroes';
import { Logger }     from '../logger.service';

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

  constructor(private logger: Logger) {  }

  getHeroes() {
    this.logger.log('Getting heroes ...');
    return HEROES;
  }
}
```

The constructor asks for an injected instance of Logger and stores it in a private field called logger. The getHeroes() method logs a message when asked to fetch heroes. Notice that the Logger service also has the @Injectable() decorator, even though it might not need its own dependencies. In fact, the @Injectable() decorator is required for all services.

41) What Is Angular Singleton Service?

In Angular, two ways to make a singleton service.

- Include the service in the AppModule:

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

@Injectable({
  providedIn: 'root',
})
export class CustomerService {
```

- Declare that the service should be provided in the application root:

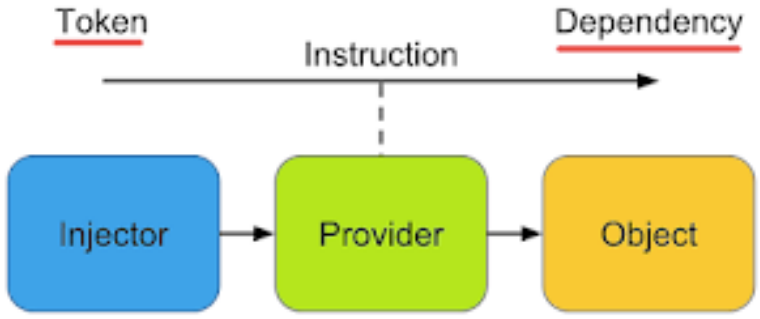
```
import { Injectable } from '@angular/core';

@Injectable()
export class CustomersService {

  constructor() { }
}
```

```
//AppModule class with @NgModule decorator
@NgModule({
 //Static, this is the compiler configuration
 //declarations is used for configure the selectors.
 declarations: [
   AppComponent
 ],
 //Composability and Grouping
 //imports used for composing NgModules together.
 imports: [
   BrowserModule
 ],
 //Runtime or injector configuration
 //providers is used for runtime injector configuration.
 providers: [CustomerService],
 //bootstrapped entry component
 bootstrap: [AppComponent]
})
export class AppModule { }
```

42) What is Dependency Injection (DI) in Angular?

Dependency injection is an important application design pattern and Angular has its own dependency injection framework. It provides the ability to add the functionalities to the components at runtime. Angular Dependency Injection consists of three things:

- Injector : The injector object provides us a mechanism by which the desired dependency is instantiated.
- Provider : A Provider is a medium by which we register our dependency that need to be injected.
- Dependency : The dependency is an object of the class that we want to use.



Angular 2 Dependency Injection

43) What Is Injectors?

A service is just a class in Angular until you register with an Angular dependency injector. The injector is responsible for creating angular service instances and injecting them into classes. You rarely create an injector yourself and Angular creates automatically during the bootstrap process. Angular doesn't know automatically how you want to create instances of your services or injector. You must configure it by specifying providers for every service. Actually, providers tell the injector how to create the service and without a provider not able to create the service. It is useful in the case when services needs to be instantiated and injected before providers.Example :

- First we need to inject Injector into a service using constructor and then using Injector.get method we can instantiate a service. Suppose we want LoggerService to inject into GlobalErrorHandlerService using Injector. First specify the providers for the services.

```
import { NgModule, ErrorHandler } from '@angular/core';

@NgModule({
  providers: [
      LoggerService,
      GlobalErrorHandlerService,
      { provide: ErrorHandler, useClass: GlobalErrorHandlerService }
  ],
  ------
})
export class AppModule { }
```

- Now find the code to
  inject LoggerService into GlobalErrorHandlerService using Injector.

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

@Injectable()
export class GlobalErrorHandlerService implements ErrorHandler {
    constructor(private injector: Injector) { }

    handleError(error: any) {
      const loggerService = this.injector.get(LoggerService);
      ------
    }
}
```

44) What Are @Injectable providers?

@Injectable() decorator identifies the class or service that is applicable for dependency injection by Angular injectors. @Injectable() can also specify providers for the service at which it is decorated. @Injectable() has providedIn properties using which we can specify provider for that service.

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

@Injectable({
    providedIn: 'root'
})
export class BookService {
    ------
}
```

Configuring providedIn: 'root' means that above service instance will be created by root injector by using service constructor. If there are parameters in constructor, that will be provided by the injector. The service configured with providedIn: 'root' will be available for dependency injection for all components and services in the application. @Injectable() decorator can also configure providers using useClass, useExisting, useValue and useFactory properties.

- **useExisting:**

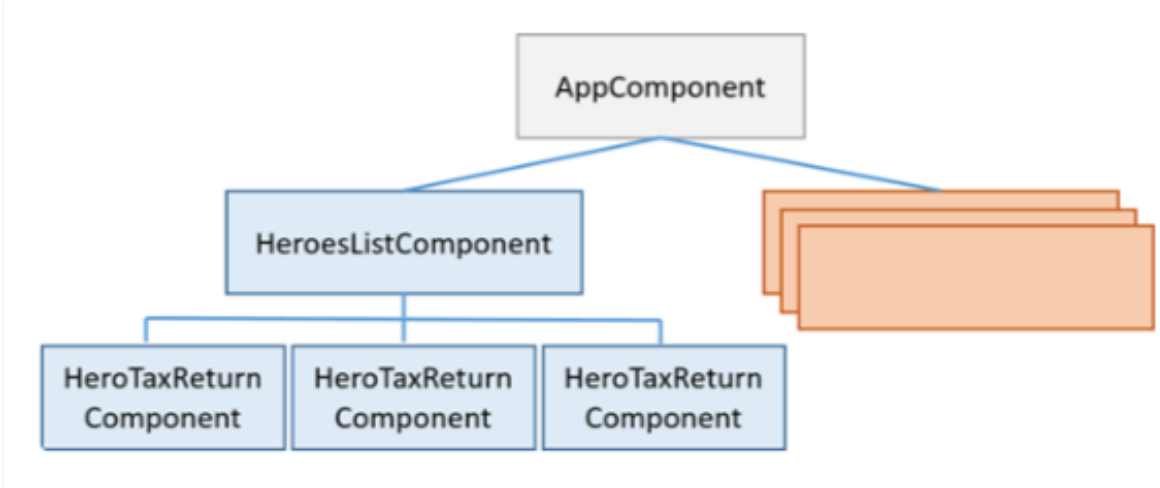
```
import { Injectable } from '@angular/core';
import { Computer } from './computer';
import { LaptopService } from './laptop.service';

@Injectable({
        providedIn: 'root',
        useExisting: LaptopService
})
export class DesktopService implements Computer {
        ------
}
```

- **useFactory:**

```
import { Injectable } from '@angular/core';
import { BookService } from './book.service'

@Injectable({
        providedIn: 'root',
        useFactory: (bookService: BookService) =>
                        new PreferredBookService(bookService),
        deps: [ BookService ]
})
export class PreferredBookService {
        constructor(private bookService: BookService) {}
        ------
}
```

45) What Is Injector tree and bubbling?

- Consider this guide's variation on the Tour of Heroes application. At the top is the AppComponent which has some subcomponents, such as the HeroesListComponent. The HeroesListComponent holds and manages multiple instances of the HeroTaxReturnComponent. The following diagram represents the state of this three-level component tree when there are three instances of HeroTaxReturnComponent open simultaneously.



- When a component requests a dependency, Angular tries to satisfy that dependency

with a provider registered in that component's own injector. If the component's injector lacks the provider, it passes the request up to its parent component's injector. If that injector can't satisfy the request, it passes the request along to the next parent injector up the tree. The requests keep bubbling up until Angular finds an injector that can handle the request or runs out of ancestor injectors. If it runs out of ancestors, Angular throws an error.

- If you have registered a provider for the same DI token at different levels, the first one Angular encounters is the one it uses to provide the dependency. If, for example, a provider is registered locally in the component that needs a service, Angular doesn't look for another provider of the same service.

- If you only register providers with the root injector at the top level (typically the root AppModule), the tree of injectors appears to be flat. All requests bubble up to the root injector, whether you configured it with the bootstrapModule method, or registered all providers with root in their own services.

46) What is @Inject?

The @Inject() is a constructor parameter decorator, which tells angular to Inject the parameter with the dependency provided in the given token. It is manual way of injecting dependency. We can manually inject the LoggerService by using the @Inject decorator applied to the parameter loggerService as shown below. The @Inject takes the Injector token as the parameter. The token is used to locate the dependency in the Providers.

```
export class ProductService{
    constructor(@Inject(LoggerService) private loggerService) {
        this.loggerService.log("Product Service Constructed");
    }
}
```

47) What is Http in Angular with example?

Http Service will help us fetch external data, post to it, etc. We need to import the http

module to make use of the http service. Let us consider an example to understand how to make use of the http service.

- we have imported the HttpModule from @angular/http and the same is also added in the imports array in **app.module.ts.**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HttpModule } from '@angular/http';
import { AppComponent } from './app.component';
@NgModule({
   declarations: [
      AppComponent
   ],
```

```
    imports: [
       BrowserModule,
       BrowserAnimationsModule,
       HttpModule
    ],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

- We need to import http to make use of the service. In the class **app.component.ts**, a constructor is created and the private variable http of type Http. To fetch the data, we need to use the **get API** available with http. Two operations are performed on the fetched url data map and subscribe. The Map method helps to convert the data to json format and the subscribe method we will call the display data method and pass the data fetched as the parameter to it.

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';
import { map} from 'rxjs/operators';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
```

```
export class AppComponent {
   constructor(private http: Http) { }
   httpdata;
   ngOnInit() {
      this.http.get("http://jsonplaceholder.typicode.com/users")
      .pipe(map((response) => response.json()))
      .subscribe((data) => this.displaydata(data));
   }
   displaydata(data) {this.httpdata = data;}
}
```

- In the display data method, we will store the data in a variable httpdata. The data is displayed in the browser using **ngFor** over this httpdata variable, which is done in the **app.component.html** file.

```
<ul *ngFor = "let data of httpdata">
   <li>Name : {{data.name}} Address: {{data.address.city}}</li>
</ul>
```

- Let us now add the search parameter, which will filter based on specific data. We need to fetch the data based on the search param passed. For the **get api**, we will add the search param id = this.searchparam. The searchparam is equal to 2. We need the details of **id = 2** from the json file. Following are the changes done in **app.component.ts**.

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';
import { map} from 'rxjs/operators';
@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
```

```
export class AppComponent {
   constructor(private http: Http) { }
   httpdata;
   name;
   searchparam = 2;
   ngOnInit() {
      this.http.get("http://jsonplaceholder.typicode.com/users?
id="+this.searchparam)
      .pipe(map((response) => response.json()))
      .subscribe((data) => this.displaydata(data));
   }
   displaydata(data) {this.httpdata = data;}
}
```

48) What is HttpClient in Angular with example?

HttpClient is introduced in Angular 6 and it will help us fetch external data, post to it, etc. We need to import the httpClient module to make use of the httpClient service. Let us consider an example to understand how to make use of the httpClient service.

- we have imported the HttpClientModule from @angular/common/http and the same is also added in the imports array in **app.module.ts.**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
@NgModule({
   declarations: [
      AppComponent
   ],
   imports: [
      BrowserModule,
      BrowserAnimationsModule,
      HttpClientModule
   ],
   providers: [],
   bootstrap: [AppComponent]
})
export class AppModule { }
```

- We need to import httpClient to make use of the service. In the class **AppComponent**, a constructor is created and the private variable http of type

HttpClient. To fetch the data, we need to use the **get API** available with http. Using the subscribe method we will call the display data method and pass the data fetched as the parameter to it.

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

```
export class AppComponent {
   constructor(private http: HttpClient) { }
   httpdata;
   ngOnInit() {
      this.http.get("http://jsonplaceholder.typicode.com/users")
      .subscribe((data) => this.displaydata(data));
   }
   displaydata(data) {this.httpdata = data;}
}
```

- In the display data method, we will store the data in a variable httpdata. The data is displayed in the browser using **for** over this httpdata variable, which is done in the **app.component.html** file.

```
<ul *ngFor = "let data of httpdata">
   <li>Name : {{data.name}} Address: {{data.address.city}}</li>
</ul>
```

- Let us now add the search parameter, which will filter based on specific data. We need to fetch the data based on the search param passed. For the **get api**, we will add the search param id = this.searchparam. The searchparam is equal to 2. We need the details of **id = 2** from the json file. Following are the changes done in **app.component.ts**.

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

```
export class AppComponent {
    constructor(private http: HttpClient) { }
    httpdata;
    name;
    searchparam = 2;
    ngOnInit() {
        this.http.get("http://jsonplaceholder.typicode.com/users?
id="+this.searchparam)
        .subscribe((data) => this.displaydata(data));
    }
    displaydata(data) {this.httpdata = data;}
}
```

49) What are Observable?

This is a RxJS API. Observable is a representation of any set of values over any amount of time. All angular Http methods return Observable. Observable provides methods such as map(), catch() etc. map()applies a function to each value emitted by source Observable and returns finally an instance of Observable.catch() is called when an error is occurred. catch() also returns Observable.

50) What are Promise?

This is a JavaScript class. Promise is used for asynchronous computation.
A Promise represents value which may be available now, in the future or never. Promise is a proxy for a value which is not known when the promise is created. Promise has methods such as then(), catch() etc. http.get() returns Observable and to convert it into Promise we need to call RxJS toPromise() on the instance of Observable. The then() method of Promise returns a Promise and that can be chained later. The catch() method of Promise is executed when an error is occurred and it also returns Promise.

51) What is RxJS?

RxJS stands for Reactive Extensions for JavaScript, and is a reactive programming library centered around observables and operators making it easier to write complex asynchronous code.

52) Explain Http get() with promise and observable?

Http.get with Observable:

- Http.get() returns instance of Observable<Response>. To change it into instance of Observable<Book[]>, we need to use RxJS map() operator.

```
getBooksWithObservable(): Observable<Book[]> {
    return this.http.get(this.url)
        .map(this.extractData)
        .catch(this.handleErrorObservable);
}
```

- Find the extractData() method. map() converts Response object of http.get into Book.

```
private extractData(res: Response) {
    let body = res.json();
    return body;
}
```

- To handle error, http.get provides catch() method. Find the handleErrorObservable() method.

```
private handleErrorObservable (error: Response | any) {
    console.error(error.message || error);
    return Observable.throw(error.message || error);
}
```

- To display Observable in our HTML template subscribe to the Observable and fetch values stored by it. Find the code snippet that we will write in component.

```
observableBooks: Observable<Book[]>
books: Book[];
this.observableBooks = this.bookService.getBooksWithObservable();
this.observableBooks.subscribe(
      books => this.books = books,
      error =>  this.errorMessage = <any>error);
```

- Now we can iterate books with ngFor as follows.

```
<ul>
  <li *ngFor="let book of books" >
    Id: {{book.id}}, Name: {{book.name}}
  </li>
</ul>
```

Http.get with Promise:

- Now on this value we call RxJS toPromise() method that will convert it into Promise<Response>. After calling then() method on it, it returns Promise<Book[]>.

```
getBooksWithPromise(): Promise<Book[]> {
     return this.http.get(this.url).toPromise()
            .then(this.extractData)
            .catch(this.handleErrorPromise);
}
```

- The extractData() method used in then(), is the same used in map() method with Observable.

```
private extractData(res: Response) {
    let body = res.json();
    return body;
}
```

- If there is any error then the catch() method will execute. Find the handleErrorPromise() method used in catch().

```
private handleErrorPromise (error: Response | any) {
        console.error(error.message || error);
        return Promise.reject(error.message || error);
}
```

- To display Promise on our HTML template we can call then() method and assign value to our component variable.

```
promiseBooks: Promise<Book[]>
books: Book[];
this.promiseBooks = this.bookService.getBooksWithPromise();
this.promiseBooks.then(
        books => this.books = books,
        error =>  this.errorMessage = <any>error);
```

- Now iterate books in HTML template using ngFor as follows.

```
<ul>
  <li *ngFor="let book of books" >
    Id: {{book.id}}, Name: {{book.name}}
  </li>
</ul>
```

53) Explain Http post() with promise and observable?

Http.post with Observable:

- Find code to use Http.post with Observable.

```
addBookWithObservable(book:Book): Observable<Book> {
    let headers = new Headers({ 'Content-Type': 'application/json' });
    let options = new RequestOptions({ headers: headers });
    return this.http.post(this.url, book, options)
                .map(this.extractData)
                .catch(this.handleErrorObservable);
}
```

- Headers : This is angular class to create header. In our example we are passing request header content type as application/json.

- RequestOptions : It creates a request option object which we need to optionally pass in http.post().

- We are posting book object to http.post() method that will return Observable<Response>. Using RxJS map()method we convert response data into JSON and finally we get Observable<Book> instance.

- Find the definition of extractData and handleErrorObservable methods.

```
extractData(res: Response) {
  let body = res.json();
  return body || {};
}
handleErrorObservable (error: Response | any) {
  console.error(error.message || error);
  return Observable.throw(error.message || error);
}
```

Http.post with Promise:

- Find code to use Http.post with Promise.

```
addBookWithPromise(book:Book): Promise<Book> {
    let headers = new Headers({ 'Content-Type': 'application/json' });
    let options = new RequestOptions({ headers: headers });
    return this.http.post(this.url, book, options).toPromise()
           .then(this.extractData)
           .catch(this.handleErrorPromise);
}
```

- Using RxJS toPromise() method we convert Observable<Response> into Promise<Response>. Inside then()method, response data is converted into JSON. then() method again returns Promise and hence we finally get Promise<Book> instance.

- Find the definition of extractData and handleErrorPromise methods.

```
extractData(res: Response) {
  let body = res.json();
  return body || {};
}
handleErrorPromise (error: Response | any) {
  console.error(error.message || error);
  return Promise.reject(error.message || error);
}
```

54) Explain HttpClient get() and post() with observable?

Http.get with Observable:

The HttpClient.get method allows us to cast the returned response object to a type we require. We make use of that feature and supply the type for the returned value httpClient.get<repos[]>. Is the GetRepos method, which takes the username as the argument and returns an Observable<repos[]> observable of Repos.

```
getRepos(userName:string): Observable<repos[]> {
    return this.httpClient.get<repos[]>(this.baseURL + 'users/' + userName + '/repos')
}
```

The **Subscribe operator** is the glue that connects an observer to an Observable. We need to Subscribe to an observable to see the results of observables.

```
public getRepos() {
    this.loading=true;
    this.errorMessage="";
    this.githubService.getRepos(this.userName)
        .subscribe((response) => {this.repos=response;},
        (error) => {this.errorMessage=error; this.loading=false; },
        () => {this.loading=false;})
}
```

Http.post with Observable:

- We need to provide generic return type to HttpClient.post method. To call HttpClient methods using http instance.

```
createArticle(article: Article): Observable<Article> {
    let httpHeaders = new HttpHeaders()
        .set('Content-Type', 'application/json');
    let options = {
        headers: httpHeaders
    };
    return this.http.post<Article>(this.url, article, options);
}
```

- All HttpClient methods return instance of Observable and they begin when we subscribe to them. So to begin our HttpClient.post method, we will call subscribe() method on it.

```
createArticle(article: Article) {
  this.articleService.createArticle(article).subscribe(
      article => {
        console.log(article);
      }
  );
}
```

55) What's the difference between Http and HttpClient?

- The HttpClient is used to perform HTTP requests and it imported form @angular/common/http.
- The HttpClient is more modern and easy to use the alternative of HTTP.
- HttpClient is an improved replacement for Http. They expect to deprecate Http in Angular 5 and remove it in a later version.

The example with HttpClient -

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

@Injectable()
export class CustomerService {

 //Inject HttpClient into your components or services
 constructor(private http: HttpClient) { }
}
```

And other example with Http –

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

@Injectable()
export class CustomerService {

 //Inject HttpClient into your components or services
 constructor(private http: Http) { }
}
```

56) What's the difference between HttpModule and HttpClientModule?

The HttpClientModule imported form -

```
import { HttpClientModule } from '@angular/common/http';
```

HttpModule imported from –

```
import { HttpModule } from '@angular/http';
```

- NgModule which provides the HttpClient and associated with components services and the interceptors can be added to the chain behind HttpClient by binding them to the multi-provider for HTTP_INTERCEPTORS.
- Http deprecate @angular/http in favour of @angular/common/http.
- They both support HTTP calls but HTTP is the older API and will eventually be deprecated.
- The new HttpClient service is included in the HttpClientModule that used to initiate HTTP request and responses in angular apps. The HttpClientModule is a replacement of HttpModule.

57) What's the difference between Promises and Observable?

Observables:

- Observable is just a function that takes an observer and returns a function. It is nothing more than a specific type of function with a specific purpose.
- It accepts an observer: an object with 'next', 'error' and 'complete' methods on it. And returns a cancellation function.
- Observer allows to subscribe/unsubscribe to its data stream, emit next value to the observer, notify the observer about errors and inform the observer about the stream completion.
- Observer provides a function to handle next value, errors and end of stream (ui events, http responses, data with web sockets).
- Works with multiple values over time.
- It is cancel-able/retry-able and supports operators such as map, filter, reduce etc.

Promise:

- A promise represents a task that will finish in the future.
- Promises is resolved by a value.
- Promises get rejected by exceptions.
- Not cancellable and it returns a single value.

58) What are Angular HttpHeaders?

Headers is the angular class that is used to configure request headers. Find the sample Headers instantiation.

If we want to add multiple headers, we can achieve it by `set()` method as follows.

```
myHeaders.set('Content-Type', 'application/json');
myHeaders.set('Accept', 'text/plain');
```

If we want to add multiple headers by `append()` method, we can achieve it as follows.

```
myHeaders.append('Content-Type', 'application/json');
myHeaders.append('Accept', 'text/plain');
```

59) What are Angular URLSearchParams and RequestOptions?

- URLSearchParams : URLSearchParams creates the query string in the URL. It is a map-like representation of URL search parameters. Find its constructor syntax.

```
constructor(rawParams?: string, queryEncoder?: QueryEncoder)
```

Both arguments in the constructor are optional. Angular queryEncoder parameter is used to pass any custom QueryEncoder to encode key and value of the query string. By default QueryEncoder encodes keys and values of parameter using JavaScript encodeURIComponent() method. Now we can instantiate URLSearchParams as given below.

```
let myParams = new URLSearchParams();
```

- RequestOptionsArgs and RequestOptions : RequestOptionsArgs is an interface that is used to construct a RequestOptions. The fields of RequestOptionsArgs are url, method, search, params, headers, body, withCredentials, responseType. RequestOptions is used to create request option. It is instantiated using RequestOptionsArgs. It contains all the fields of the RequestOptionsArgs interface.

Now if we have instance of `Headers` as follows.

```
let myHeaders = new Headers();
myHeaders.append('Content-Type', 'application/json');
```

And instance of `URLSearchParams` as follows.

```
let myParams = new URLSearchParams();
myParams.append('id', bookId);
```

Then `headers` and `params` can be passed to `RequestOptions` as given below.

```
let options = new RequestOptions({ headers: myHeaders, params: myParams });
```

60) Example of Http.get() with Multiple Headers and Multiple Parameters?

A book service with http.get(), Multiple Headers and Multiple Parameters:

```
@Injectable()
export class BookService {
    url = "api/books";
    constructor(private http:Http) { }
    getAllBooks(): Observable<Book[]> {
        return this.http.get(this.url)
                .map(this.extractData)
                .catch(this.handleError);
    }
```

```
getBookById(bookId: string): Observable<Book[]> {
    let myHeaders = new Headers();
    myHeaders.append('Content-Type', 'application/json');
    let myParams = new URLSearchParams();
    myParams.append('id', bookId);
    let options = new RequestOptions({ headers: myHeaders, params: myParams });
    return this.http.get(this.url, options)
            .map(this.extractData)
            .catch(this.handleError);
}
```

```
getBooksAfterFilter(catg: string, wtr: string): Observable<Book[]> {
    let myHeaders = new Headers();
    myHeaders.set('Content-Type', 'application/json');
    let myParams = new URLSearchParams();
    myParams.set('category', catg);
    myParams.set('writer', wtr);
    let options = new RequestOptions({ headers: myHeaders, params: myParams });
    return this.http.get(this.url, options)
            .map(this.extractData)
            .catch(this.handleError);
}
```

61) How to create a custom ErrorHandler?

The best way to log exceptions is to provide a specific log message for each possible exception. Always ensure that sufficient information is being logged and that nothing important is being excluded. The multiple steps involved in creating custom error logging in Angular:

- Create a constant class for global error messages.
- Create an error log service : ErrorLoggService.
- Create a global error handler for using the error log service for logging errors.

In the first step, we will create a constant class for logging global error messages and its look like this.

```
export class AppConstants {
  public static get baseURL(): string { return 'http://localhost:4200/api'; }
  public static get httpError(): string { return 'There was an HTTP error.'; }
  public static get typeError(): string { return 'There was a Type error.'; }
  public static get generalError(): string { return 'There was a general error.'; }
  public static get somethingHappened(): string { return 'Nobody threw an Error but something happened!'; }
}
```

In the second steps, we will create an error log service (ErrorLoggService) for error logging and it look like this.

```
import { Injectable} from '@angular/core';
import { HttpErrorResponse } from '@angular/common/http';
import{ AppConstants} from '../app/constants'

//#region Handle Errors Service
@Injectable()
export class ErrorLoggService {

 constructor() { }
```

```
 //Log error method
 logError(error: any) {
  //Returns a date converted to a string using Universal Coordinated Time (UTC).
  const date = new Date().toUTCString();

  if (error instanceof HttpErrorResponse) {
   //The response body may contain clues as to what went wrong
   console.error(date, AppConstants.httpError, error.message, 'Status code:',
          (<HttpErrorResponse>error).status);
  }
  else if (error instanceof TypeError) {
   console.error(date, AppConstants.typeError, error.message, error.stack);
  }
  else if (error instanceof Error) {
   console.error(date, AppConstants.generalError, error.message, error.stack);
  }
  else if(error instanceof ErrorEvent){
   //A client-side or network error occurred. Handle it accordingly
   console.error(date, AppConstants.generalError, error.message);
  }
  else {
   console.error(date, AppConstants.somethingHappened, error.message, error.stack);
  }
 }
}
//#endregion
```

In the 3rd steps, we will create a global error handler for using the error log service for

logging errors and its look like this.

its look like this:

```
import { Injectable, ErrorHandler } from '@angular/core';
import {ErrorLoggService} from './error-logg.service';

// Global error handler for logging errors
@Injectable()
export class GlobalErrorHandler extends ErrorHandler {
  constructor(private errorLogService: ErrorLoggService) {
    //Angular provides a hook for centralized exception handling.
    //constructor ErrorHandler(): ErrorHandler
     super();
  }

  handleError(error) : void {
    this.errorLogService.logError(error);
  }
}
```

In the 4th step, we will Import global handler and error handler services in the NgModule and its look like this.

```
import {ErrorLoggService} from './error-logg.service';
import {GlobalErrorHandler} from './global-error.service';

//AppModule class with @NgModule decorator
@NgModule({
 //declarations is used for configure the selectors
 declarations: [
   AppComponent,
 ],
 //Composability and Grouping
 imports: [
   BrowserModule,
   HttpClientModule
 ],
 //Runtime or injector configuration
 //Register global error log service and error handler
 providers: [ErrorLoggService, GlobalErrorHandler],
 //bootstrapped entry component
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Finally, we got a custom error handler for log error in your application errors.

62) How to catch and log specific Angular errors in your app?

The default implementation of ErrorHandler log error messages, status, and stack to the console. To intercept error handling, we write a custom exception handler that replaces this default as appropriate for your app.

```
import { Injectable, ErrorHandler } from '@angular/core';
import {ErrorLoggService} from './error-logg.service';

// Global error handler for logging errors
@Injectable()
export class GlobalErrorHandler extends ErrorHandler {
  constructor(private errorLogService: ErrorLoggService) {
    //Angular provides a hook for centralized exception handling.
    //constructor ErrorHandler(): ErrorHandler
     super();
  }

  handleError(error) : void {
    this.errorLogService.logError(error);
  }
}
```

63) What is Angular Http Interceptors?

- HttpInterceptors is an interface which uses to implement the intercept method. Intercepts HttpRequest and handles them.
- Intercepts is an advanced feature that allows us to intercept each request/response and modify it before sending/receiving.
- Interceptors capture every request and manipulate it before sending and also catch every response and process it before completing the call.
- Firstly, we can implement own interceptor service and this service will “catch” each request and append an Authorization header.

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

@Injectable()
export class MyInterceptor implements HttpInterceptor {
 //Intercepts HttpRequest and handles them.
 intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  const reqHeader = req.clone({headers: req.headers.set('Authorization', 'MyAuthToken')});

  return next.handle(reqHeader);
 }
}
```

- After that you can configure your own interceptor service (MyInterceptor) and HTTP_INTERCEPTORS in the app Module.

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

@NgModule({
 providers: [{
  provide: HTTP_INTERCEPTORS,
  useClass: MyInterceptor,
  multi: true,
 }],
})
export class AppModule {}
```

- Following this logic, Authorization token will be appended to each request. It's also possible to override request's headers by using **set**() method.

64) What is routing in Angular?

Routing in Angular is a mechanism for navigating between pages and displaying an appropriate component/page on browser. The Router helps mapping application URL to application components. Following are some of the main components used to configure routing:

- Routes
- Router Imports
- RouterOutlet
- RouterLink

65) What is router imports?

The Angular Router is an optional service that presents a particular component view for a given URL. It is not part of the Angular core. It is in its own library package, @angular/router. Import what you need from it as you would from any other Angular package.

src/app/app.module.ts (import)

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

66) What is routes and routes properties?

Routes defines and array of roots that map a path to a component. Paths are configured in module to make available it globally. Create array of Routes in which we map a URL with a component.

```
const routes: Routes = [
        { path: 'manage-book', component: ManageBookComponent },
        { path: 'update-book/:id', component: UpdateBookComponent },
        { path: '', redirectTo: '/manage-book ', pathMatch: 'full' },
        { path: '**', component: PageNotFoundComponent }
]
```

Find the properties of Route interface.

- path: Defines path for a component.
- component: Name of the component.
- outlet: Name of the outlet component should be placed into.
- data: Additional data provided to component via ActivatedRoute.
- children: Array of child route definitions.
- loadChildren: Reference to lazy loaded child routes.
- redirectTo: Defines a URL to redirect current matched URL.

Example-1 : configuration with path, component and children property.

```
const countryRoutes: Routes = [
    {
        path: 'country',
        component: CountryComponent,
        children: [
            {
                path: 'add',
                component: AddCountryComponent
            }
        ]
    }
]
```

Example-2 : configuration with path, component, children and redirectTo property.

```
[{
    path: 'country',
    component: CountryComponent,
    children: [
      {
        path: 'add',
        component: AddCountryComponent
      },
      {
        path: 'item/create',
        redirectTo: 'add'
      }
    ]
}]
```

67) What is router module?

RouterModule is a separate module in angular that provides required services and directives to use routing and navigation in angular application. we need to import RouterModule.forRoot(routes) using imports metadata of @NgModule. Here argument routes is our constant that we have defined above as array of Routes to map path with component.

```
imports: [ RouterModule.forRoot(routes) ]
```

68) What is routerLink and routerLinkActive?

- **RouterLink :** The RouterLink is a directive that binds the HTML element to a Route. Clicking on the HTML element, which is bound to a RouterLink, will result in navigation to the Route. The RouterLink may contain parameters to be passed to the route's component.

```
<a routerLink="/manage-book" routerLinkActive="active-link">Manage Book</a>
<a [routerLink]="['/view-detail', book.id]">View Detail</a>
```

- **RouterLinkActive :** RouterLinkActive is a directive for adding or removing classes from an HTML element that is bound to a RouterLink. Using this directive, we can toggle CSS classes for active RouterLinksbased on the current RouterState.

Consider the following example for active a link –

```
<a routerLink="/user/detail" routerLinkActive="active-link">User Detail</a>
```

You can also set more than one class and it look like this.

```
<a routerLink="/user/detail" routerLinkActive="active-class1 active-class2">User detail</a>
<a routerLink="/user/detail" [routerLinkActive]="['active-class1', 'active-class2']">User detail</a>
```

69) What is router-outlet?

The RouterOutlet is a directive from the router library that is used like a component. It acts as a placeholder that marks the spot in the template where the router should display the components for that outlet. We have created menu items using RouterOutlet. They will be shown in every view where we navigate using the route binding with routerLink.

```
<nav [ngClass] = "'menu'">
   <a routerLink="/home" routerLinkActive="active-link">Home</a> |
   <a routerLink="/add-book" routerLinkActive="active-link">Add Book</a> |
   <a routerLink="/manage-book" routerLinkActive="active-link">Manage Book</a>
</nav>
<router-outlet></router-outlet>
```

70) How to configure routing in Angular?

To Configure the Routing in Angular, you need to follow these steps:

- **Set the <base href> :** The HTML <base> element specifies the base URL to use for all relative URLs contained within a document. The Angular Router uses the HTML5 style of Routing (or PathLocationStrategy) as the default option. The router makes use of the browser's history API for navigation and URL interaction. To make HTML5 routing to work, we need to set up the "**base href**" in the DOM. This is done

in app's *index.html* file immediately after the head tag.

```
<!doctype html>
<html>
<head>
  <base href="/">
  <meta charset="utf-8">
  <title>Angular 2 Routing</title>


  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet">
    ...
</head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>
```

- **Define routes for the view :** Next, create an array of route objects. Each route maps path (URL Segment) to the component. Where, **path :** The URL path segment of the route. We will use this value to refer to this route elsewhere in the app. **component :** The component to be loaded.

```
import { Routes } from '@angular/router';

import { HomeComponent} from './home.component'
import { ContactComponent} from './contact.component'
import { ProductComponent} from './product.component'
import { ErrorComponent} from './error.component'
```

```
export const appRoutes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'product', component: ProductComponent },
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: '**', component: ErrorComponent }
];
```

- **Register the Router Service with Routes :** Import the Router Module from @angular/router library in the root module of the application. Then, install the routes using the RouterModule.forRoot method, passing the routes as the argument

in the imports array.

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

import { RouterModule } from '@angular/router';
```

```
import { AppComponent } from './app.component';
import { HomeComponent} from './home.component'
import { ContactComponent} from './contact.component'
import { ProductComponent} from './product.component'
import { ErrorComponent} from './error.component'

import { ProductService } from './product.service';

import { appRoutes } from './app.routes';
```

```
@NgModule({
  declarations: [
    AppComponent,HomeComponent,ContactComponent,ProductComponent,ErrorComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot(appRoutes)                          /*path location strategy */
    /*RouterModule.forRoot(appRoutes, { useHash: true }) */   /*Hashlocationstrategy */
  ],
  providers: [ProductService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

- **Map HTML Element actions to Route :** Next, we need to bind the click event of the link, image or button to a route. This is done using the routerlink directive.
  The routerLink directive accepts an array of route names along with parameters. This array is called as Link Parameters array.

- **Choose where you want to display the view :** Finally, we need to tell the angular where to display the view. This is done using the RouterOutlet directive as

shown. We will add the following directive to the root component.

```
<div class="container">

<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <a class="navbar-brand" [routerLink]="['/']"><strong> {{title}} </strong></a>
    </div>
    <ul class="nav navbar-nav">
        <li><a [routerLink]="['home']">Home</a></li>
        <li><a [routerLink]="['product']">Product</a></li>
        <li><a [routerLink]="['contact']">Contact us</a></li>
    </ul>
  </div>
</nav>

 <router-outlet></router-outlet>

</div>
```

71) What is activated route?

The **ActivatedRoute** is a service, which keeps track of the currently activated route associated with the loaded Component. The ActivatedRoute class has a params property which is an array of the parameter values, indexed by name. We can use the params array to get the parameter value.

- To use ActivatedRoute, we need to import it in our component:

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

- Then inject it into the component using dependency injection:

```
constructor(private _Activatedroute:ActivatedRoute)
```

72) What are the ways in which you can use the ActivatedRoute to get the

parameter value?

There are two ways in which you can use the ActivatedRoute to get the parameter value:

- **Using Snapshot : T**he snapshot property returns initial value of the route. You can then access the params array, to access the value of the id. Use this option, if you only need the initial value.

```
product:Product;
id;

constructor(private _Activatedroute:ActivatedRoute,
            private _router:Router,
            private _productService:ProductService){
}


/* Using snapshot */
ngOnInit() {
    this.id=this._Activatedroute.snapshot.params['id'];
    let products=this._productService.getProducts();
    this.product=products.find(p => p.productID==this.id);
}
```

- **Using Observable :** You can retrieve the value of id by subscribing to the params observable property of the activateRoute. Use this option if you expect the value of the parameter to change over time.

```
sub;

ngOnInit() {

    this.sub=this._Activatedroute.params.subscribe(params => {
    this.id = params['id'];
    let products=this._productService.getProducts();
    this.product=products.find(p => p.productID==this.id);

    });
}

ngOnDestroy() {
    this.sub.unsubscribe();
}
```

73) Explain and how to use path location strategy?

- The PathLocationStrategy is the default strategy in Angular 2 application. To Configure the strategy, we need to add <base href> in the <head> section of root page (index.html) of our application.

```
<base href="/">
```

- The Browser uses this element to construct the relative URLs for static resources (images, CSS, scripts) contained in the document. If you do not have access to <head> Section of the index.html., then you can follow either of the two steps.

- Add the APP_BASE_HREF value.as follows in the providers section of the root module.

```
import {Component, NgModule} from '@angular/core';
import {APP_BASE_HREF} from '@angular/common';

@NgModule({
providers: [{provide: APP_BASE_HREF, useValue: '/my/app'}]
})
class AppModule {}
```

- Use the absolute path for all the static resources like CSS, images, scripts, and HTML files.

74) What is router state?

After the end of each successful navigation lifecycle, the router builds a tree of ActivatedRoute objects that make up the current state of the router. You can access the current RouterState from anywhere in the application using the Router service and the routerState property. Each ActivatedRoute in the RouterState provides methods to traverse up and down the route tree to get information from parent, child and sibling routes.

75) What is router event?

During each navigation, the Router emits navigation events through the Router.events property. These events range from when the navigation starts and ends to many points in between. These events are logged to the console when the enableTracing option is enabled also. The list of some navigation events is displayed below:

Router Event	Description
NavigationStart	An event triggered when navigation starts.
RouteConfigLoadStart	An event triggered before the Router lazy loads a route configuration.
RouteConfigLoadEnd	An event triggered after a route has been lazy loaded.
RoutesRecognized	An event triggered when the Router parses the URL and the routes are recognized.
GuardsCheckStart	An event triggered when the Router begins the Guards phase of routing.

76) What is the difference between ActivatedRoute and RouterState in Angular?

ActivateRoute and RouterState both refer to Routing in Angular.

- ActivatedRoute consists of the information about a route associated with the component loaded in outlet whereas RouterState represents the state.
- ActivatedRouteSnapshort has old data When Route changes, ActivateRouteSnapshort has data from the previous route whereas the RouteState care about the arrangements and application components.
- ActivatedRouteSnapchat to traverse all the activated routes whereas RouterState maintains the states.

77) What is the difference between RouterModule.forRoot() vs RouterModule.forChild()? Why is it important?

forRoot is a convention for configuring app-wide Router service with routes, whereas forChild is for configuring the routes of lazy-loaded modules.

78) How does loadChildren property work?

The Router calls it to dynamically load lazy loaded modules for particular routes.

79) What is Lazy Loading and How to enable Lazy Loading?

Most of the enterprise application contains various modules for specific business cases. Bundling whole application code and loading will be huge performance impact at initial call. Lazy lading enables us to load only the module user is interacting and keep the rest to be loaded at runtime on demand. Lazy loading speeds up the application initial load time by splitting the code into multiple bundles and loading them on demand. Every Angular application must have one main module say AppModule. The code should be splitted into various child modules (NgModule) based on the application business case.

Example:

- We don't require to import or declare lazily loading module in root module.
- Add the route to top-level routing (app.routing.ts) and set loadChildren. loadChildren takes an absolute path from root folder followed by #{ModuleName}. RouterModule.forRoot() takes routes array and configures the router.
- Import module specific routing in the child module.
- In the child module routing, specify a path as empty string ' ', the empty path. RouterModule.forChild again takes routes array for the child module components to load and configure router for child.
- Then, export const routing: ModuleWithProviders = **RouterModule.forChild**(routes);

80) When does a lazy loaded module get loaded?

When its related route is first requested.

81) What is a Route Guard?

Guarding routes means whether we can visit the route or not. For example in login authentication based application, a user has to login first to enter into the application. If

there is no route guard then anyone can access any link but using route guard we restrict the access of links. To achieve route guards, Angular provides following interfaces that are contained in @angular/router package.

82) How would you use a Route Guard?

You would implement CanActivate or CanDeactivate and specify that guard class in the route path you're guarding. Steps are given below:

- Build the Guard as Service.
- Implement the Guard Method in the Service.

```
@Injectable()
export class ProductGuardService implements CanActivate {

  constructor(private _router:Router ) {
  }

  canActivate(route: ActivatedRouteSnapshot,
              state: RouterStateSnapshot): boolean {
      console.log("canActivate");      //return true
     //remove comments to return true
     alert('You are not allowed to view this page. You are redirected to Home Page');
     this._router.navigate(["home"]);
     return false;
} }
```

- Register the Guard Service in the Root Module.

```
providers: [ProductService,ProductGuardService]
```

- Update the Routes to use the guards.

```
{ path: 'product', component: ProductComponent, canActivate : [ProductGuardService]
}
```

83) What are some different types of RouteGuards?

- CanActivate : This guard decides if a route can be activated (or component gets used). This guard is useful in the circumstance where the user is not authorized to navigate to the target component. Or the user might not be logged into the system.

- CanDeactivate : This Guard decides if the user can leave the component (navigate away from the current route). This route is useful in where the user might have some pending changes, which was not saved. The canDeactivate route allows us to ask user confirmation before leaving the component. You might ask the user if it's OK to discard pending changes rather than save them.

- **Resolve :** This guard delays the activation of the route until some tasks are completed. You can use the guard to pre-fetch the data from the backend API, before activating the route

- CanLoad : The CanLoad Guard prevents the loading of the **Lazy Loaded Module**. We generally use this guard when we do not want to unauthorized user to be able to even see the source code of the module. This guard works similar to CanActivate guard with one difference. The CanActivate guard prevents a particular route being accessed. The CanLoad prevents entire lazy loaded module from being downloaded, Hence protecting all the routes within that module.

- CanActivateChild : This guard determines whether a child route can be activated.

84) How would you intercept 404 errors in Angular?

Can provide a final wildcard path like so:

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

**85) This link doesn't work. Why? How do I fix it? **

<a [routerLink]=”[’product.id’]”>{{product.id}}

86) What do route guards return?

A: boolean or a Promise/Observable resolving to boolean value.

87) How is SPA (Single Page Application) technology different from the traditional web technology?

In traditional web technology, the client requests for a web page (HTML/JSP/asp) and the server sends the resource (or HTML page), and the client again requests for another page and the server responds with another resource. The problem here is a lot of time is consumed in the requesting/responding or due to a lot of reloading. Whereas, in the SPA technology, we maintain only one page (index.HTML) even though the URL keeps on changing.

88) Is it possible to have a multiple router-outlet in the same template?

Yes, why not! We can use multiple router-outlets in the same template by configuring our routers and simply adds the router-outlet name.

- **Step-1:**

```
<div class="row">
 <div class="user">
  <router-outlet name="users"></router-outlet>
 </div>
 <div class="detail">
  <router-outlet name="userDetail"></router-outlet>
 </div>
</div>
```

- **Step-2:**

```
//Apps roots
const appRoots = [
 { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
 { path: 'userDetail', component: UserDetailComponent }, //Id is a Roots parameter.
 { path: 'users', component: UserComponent, data:{ title:'User List'} },
 { path: '**', redirectTo: 'PageNotFoundComponent' } //Wild Cards, the router will instantiate the PageNotFound component.
];
```

- **Step-3:**

```
//AppModule class with @NgModule decorator
@NgModule({
 //Composability and Grouping
 //imports used for composing NgModules together
 imports: [
  BrowserModule,
  //enableTracing is used for debugging purposes only
  //Enables the location strategy that uses the URL fragment instead of the history API.
  RouterModule.forRoot(appRoots)
 ],
 //bootstrapped entry component
 bootstrap: [AppComponent]
})
export class AppModule { }
```

89) How to configure animation in Angular?

Find the steps to configure animation in our Angular application.

- Make sure that package.json contains @angular/
  animations in **dependencies** block. If not available then either upgrade Angular CLI or configure @angular/animations in **dependencies** block and run **npm install** .

- Configure BrowserAnimationsModule in application module to support animations.

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@NgModule({
  imports: [
        ---
        BrowserAnimationsModule
  ]
  ---
})
export class AppModule { }
```

- Write animation code.

```
import { animate, state, style, transition, trigger } from '@angular/animations';

export const ON_OFF_ANIMATION =
        trigger('onOffTrigger', [
                state('off', style({
                  backgroundColor: '#E5E7E9',
                  transform: 'scale(1)'
                })),
                state('on',   style({
                  backgroundColor: '#17202A',
                  transform: 'scale(1.1)'
                })),
                transition('off => on', animate('.6s 100ms ease-in')),
                transition('on => off', animate('.7s 100ms ease-out'))
        ]);
```

- To use animation in component, the decorator @Component provides a metadata named as animations.

```
import { Component } from '@angular/core';
import { ON_OFF_ANIMATION } from './animations/on-off.animation';

@Component({
  ---
  animations: [ ON_OFF_ANIMATION ]
})
export class BookComponent {}
```

- To use animation in our HTML template we need to bind trigger name using

property binding. Suppose we have an animation trigger with name onOffTrigger then it will be bound to HTML element using @ as [@onOffTrigger] = "trigger_state_name".

```
<div [@onOffTrigger] = "trigger_state_name ">
  ---
</div>
```

90) How do you define transition between two states?

Using the transition and animate function in an animations block like so:

animations: [transition('inactive => active'), animate('100 ms ease-in')]

91) How do you define a wildcard state?

Using the asterisk - example:

transition('* => active'), animate('100ms ease-in'))

92) What is Protractor?

E2E (end-to-end) testing framework.

93) What is Karma?

Unit test testing library.

94) What are spec files?

Jasmine unit test files.

95) What is TestBed?

Class to create testable component fixtures.

96) What does detectChanges to in Angular jasmine tests?

propagates changes to the DOM by running Angular change detection.

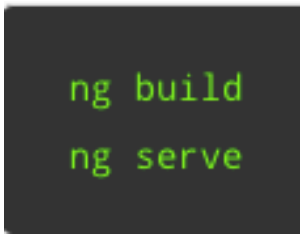
97) Why would you use a spy in a test?

To verify a particular value was returned or a method was called, for example when calling a service.

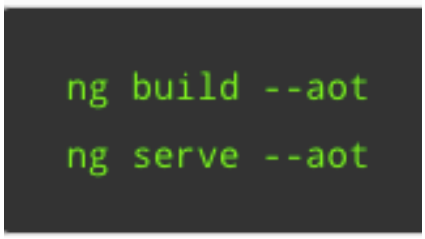
98) What Is the Angular Compiler? Why we need Compilation in Angular?

The **Angular** compiler converts our applications code (**TypeScript**) into **JavaScript** code + HTML before browser downloads and runs that code. The Angular offers two ways to compile our application code:

- **Just-in-Time (JIT) :** JIT compiles our app in the browser at runtime (compiles before running). JIT compilation is the default when you run the ng build (build only) or ng serve (build and serve locally) CLI commands:



- **Ahead-of-Time (AOT) :** Ahead-of-Time (AOT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase *before* the browser downloads and runs that code. Compiling your application during the build process provides a faster rendering in the browser. (compiles while running). For AOT compilation, include the --aot option with the ng build or ng serve command:



99) What Are The Advantages And Disadvantages Of Aot Compilation?

Advantages:

- Faster download: Since the app is already compiled, many of the angular compiler related libraries are not required to be bundled, the app bundle size get reduced. So, the app can be downloaded faster.

- Lesser No. of Http Requests: If the app is not bundled to support lazy loading (or whatever reasons), for each associated html and css, there is a separate request goes to the server. The pre-compiled application in-lines all templates and styles with components, so the number of Http requests to the server would be lesser.

- Faster Rendering: If the app is not AOT compiled, the compilation process happens in the browser once the application is fully loaded. This has a wait time for all necessary component to be downloaded, and then the time taken by the compiler to compile the app. With AOT compilation, this is optimized.

- Detect error at build time: Since compilation happens beforehand, many compile time error can be detected, providing a better degree of stability of application.

Disadvantages:

- Works only with HTML and CSS, other file types need a previous build step.

- No watch mode yet, must be done manually (bin/ngc-watch.js) and compiles all the files.

- Need to maintain AOT version of bootstrap file (might not be required while using

tools like cli).

- Needs cleanup step before compiling.

100) What Is the difference between JIT compiler and AOT compiler?

JIT (Just-in-Time):

- JIT compiles our app in the browser at runtime.
- Compiles before running.
- Each file compiled separately.
- No need to build after changing our app code and it automatically reflects the changes in your browser page.
- Highly secure.
- Very suitable for local development.

AOT (Ahead-of-Time):

- AOT compiles our app code at build time.
- Compiles while running.
- Compiled by the machine itself, via the command line (Faster).
- All code compiled together, inlining HTML/CSS in the scripts.
- Highly secure.
- Very suitable for production builds.

Cheet Sheets

Data direction	Syntax	Type

One-way
from data source
to view target
	`{{expression}}`
	`[target]="expression"`
	`bind-target="expression"`
		Interpolation
		Property
		Attribute
		Class
		Style

One-way from view target to data source	`(target)="statement"` `on-target="statement"`	Event
Two-way	`[(target)]="expression"` `bindon-target="expression"`	Two-way

The **target of a data binding** is something in the DOM. Depending on the binding type, the target can be an (element | component | directive) property, an (element | component | directive) event, or (rarely) an attribute name. The following table summarizes:

Type	Target	Examples

Property	Element property
	Component property
	Directive property
		src/app/app.component.html
		`<img [src]="heroImageUrl">`
		`<app-hero-detail [hero]="currentHero"></app-hero-detail>`
		`<div [ngClass]="{'special': isSpecial}"></div>`

Event	Element event	

Event	Element event
	Component event
	Directive event
		src/app/app.component.html
		`<button (click)="onSave()">Save</button>`
		`<app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>`
		`<div (myClick)="clicked=$event" clickable>click me</div>`

Two-way	Event and property	src/app/app.component.html `<input [(ngModel)]="name">`
Attribute	Attribute (the exception)	src/app/app.component.html `<button [attr.aria-label]="help">help</button>`
Class	`class` property	src/app/app.component.html `<div [class.special]="isSpecial">Special</div>`
Style	`style` property	src/app/app.component.html `<button [style.color]="isSpecial ? 'red' : 'green'">`

Template reference variables (`#var`)

A **template reference variable** is often a reference to a DOM element within a template. It can also be a reference to an Angular component or directive or a web component.

Use the hash symbol (#) to declare a reference variable. The `#phone` declares a `phone` variable on an `<input>` element.

src/app/app.component.html

```
<input #phone placeholder="phone number">
```

You can refer to a template reference variable *anywhere* in the template. The `phone` variable declared on this `<input>` is consumed in a `<button>` on the other side of the template

src/app/app.component.html

```
<input #phone placeholder="phone number">

<!-- lots of other elements -->

<!-- phone refers to the input element; pass its `value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>
```

How a reference variable gets its value

In most cases, Angular sets the reference variable's value to the element on which it was declared. In the previous example, `phone` refers to the *phone number* `<input>` box. The phone button click handler passes the *input* value to the component's `callPhone` method. But a directive can change that behavior and set the value to something else, such as itself. The `NgForm` directive does that.

The following is a *simplified* version of the form example in the Forms guide.

src/app/hero-form.component.html

```
<form (ngSubmit)="onSubmit(heroForm)" #heroForm="ngForm">
  <div class="form-group">
    <label for="name">Name
      <input class="form-control" name="name" required [(ngModel)]="hero.name">
    </label>
  </div>
  <button type="submit" [disabled]="!heroForm.form.valid">Submit</button>
</form>
<div [hidden]="!heroForm.form.valid">
  {{submitMessage}}
</div>
```

A template reference variable, `heroForm`, appears three times in this example, separated by a large amount of HTML. What is the value of `heroForm`?

If Angular hadn't taken it over when you imported the `FormsModule`, it would be the HTMLFormElement. The `heroForm` is actually a reference to an Angular NgForm directive with the ability to track the value and validity of every control in the form.

The native `<form>` element doesn't have a `form` property. But the `NgForm` directive does, which explains how you can disable the submit button if the `heroForm.form.valid` is invalid and pass the entire form control tree to the parent component's `onSubmit` method.

`@Input` decorator binds a property within one component (child component) to receive a value from another component (parent component). This is one way communication from parent to child. The component property should be annotated with `@Input` decorator to act as input property. A component can receive a value from another component using component property binding. Now we will see how to use **@Input**. It can be annotated at any type of property such as number, string, array or user defined class. To use alias for the binding property name we need to assign an alias name as `@Input(alias)`.
Find the use of `@Input` with string data type.

```
@Input()
ctMsg : string;
```

Now find array data type with `@Input` decorator. Here we are aliasing the property name. In the component property binding, alias name `ctArray` will be used.

```
@Input('ctArray')
myctArray : Array<string>
```

Now find `@Input` decorator with a property of user defined class type.

```
@Input('stdLeader')
myStdLeader : Student;
```

`@Output` decorator binds a property of a component to send data from one component (child component) to calling component (parent component). This is one way communication from child to parent component. `@Output` binds a property of the type of angular `EventEmitter` class. This property name becomes custom event name for calling component. `@Output` decorator can also alias the property name as `@Output(alias)` and now this alias name will be used in custom event binding in calling component.
Find the `@Output` decorator using aliasing.

```
@Output('addStudentEvent')
addStdEvent = new EventEmitter<Student>();
```

In the above code snippet `addStudentEvent` will become custom event name. Now find `@Output` decorator without aliasing.

```
@Output()
sendMsgEvent = new EventEmitter<string>();
```

Here `sendMsgEvent` will be custom event name.

@ViewChild() using Component

`@ViewChild()` can be used for component communication. A component will get instance of another component inside it using `@ViewChild()`. In this way parent component will be able to access the properties and methods of child component. The child component selector will be used in parent component HTML template. Now let us

discuss example.

1. Number example

In this example we will increase and decrease the counter using two buttons. The counter will be from a child component. Increase and decrease button will be inside parent component. We will communicate parent and child components using `@ViewChild()` decorator.
First of all we will create a component where we will write methods to increase and decrease counter.
number.component.ts

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

@Component({
  selector: 'app-number',
  template: '<b>{{message}}</b>'
})
export class NumberComponent {
        message:string ='';
        count:number = 0;
        increaseByOne() {
           this.count = this.count + 1;
           this.message = "Counter: " + this.count;
        }
        decreaseByOne() {
           this.count = this.count - 1;
           this.message = "Counter: " + this.count;
        }
}
```

Now we will create the instance of `NumberComponent` in our parent component using `@ViewChild()`.
number-parent.component.ts

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

@Component({
  selector: 'app-number-parent',
  templateUrl: './number-parent.component.html'
})
export class NumberParentComponent {
    @ViewChild(NumberComponent)
    private numberComponent: NumberComponent;
    increase() {
       this.numberComponent.increaseByOne();
    }
    decrease() {
       this.numberComponent.decreaseByOne();
    }
}
```

We will observe that we are able to access the methods of `NumberComponent` in `NumberParentComponent`. We will use selector of `NumberComponent` in HTML template of `NumberParentComponent`.
number-parent.component.html

```
<h3>@ViewChild using Component</h3>
Number Example:
<button type="button" (click)="increase()">Increase</button>
<button type="button" (click)="decrease()">Decrease</button>
<br/><br/>
```

```
<app-number></app-number>
```

As usual both components `NumberComponent` and `NumberParentComponent` need to be configured in `@NgModule` in application module.

`@ViewChildren` gets the list of elements or directives from the view DOM as `QueryList`.

cp1.component.ts

```
import { Component, ViewChild, ViewChildren, AfterViewInit, TemplateRef, ViewContainerRef,
import { MessageDirective } from './message.directive';
import { WriterComponent } from './writer.component';

@Component({
   selector: 'app-cp1',
   templateUrl: './cp1.component.html'
})
export class Cp1Component implements  AfterViewInit {
    //QueryList + @ViewChildren + Directive
    @ViewChildren(MessageDirective)
    private msgList: QueryList<MessageDirective>

    @ViewChild('msgTemp')
    private msgTempRef : TemplateRef<any>

    //QueryList + @ViewChildren + Component
    @ViewChildren('bkWriter')
    allWriters: QueryList<WriterComponent>

    showAllWriter = false;

    //QueryList + @ViewChildren + ElementRef
    @ViewChildren('pname')
    allPersons: QueryList<ElementRef>

    ngAfterViewInit() {
        console.log('--- using QueryList.changes ---');
        this.allWriters.changes.subscribe(list => {
             list.forEach(writer => console.log(writer.writerName + ' - ' + writer.bookName
        });
        console.log('--- using QueryList.forEach ---');
        this.msgList.forEach(messageDirective =>
             messageDirective.viewContainerRef.createEmbeddedView(this.msgTempRef));

        this.allWriters.forEach(writer => console.log(writer.writerName + ' - ' + writer.bo

        console.log('--- using QueryList.length ---');
        console.log(this.allWriters.length);

        console.log('--- using QueryList.find ---');
        let javaWriter = this.allWriters.find(writer => writer.bookName === 'Java Tutorials
```

```
        console.log(javaWriter.writerName);

        console.log('--- using QueryList.map ---');
        let wnames = this.allWriters.map(writer => writer.writerName);
        for (let name of wnames) {
            console.log(name);
        }

        console.log('--- using QueryList.filter ---');
        let writers = this.allWriters.filter(writer => writer.writerName === 'Krishna');
        for (let w of writers) {
            console.log(w.bookName);
        }

        console.log('--- using QueryList.first ---');
        let firstEl = this.allPersons.first;
        console.log(firstEl.nativeElement.innerHTML);

        console.log('--- using QueryList.last ---');
        let lastEl = this.allPersons.last;
        console.log(lastEl.nativeElement.innerHTML);
    }
    onShowAllWriters() {
        this.showAllWriter = (this.showAllWriter === true)? false : true;
    }
}
```

cp1.component.html

```
<h3>QueryList + @ViewChildren + Directive</h3>
<ng-template #msgTemp>
   Welcome to you.<br/>
   Happy learning!
</ng-template>
<div cpMsg> </div>
<div cpMsg> </div>

<h3>QueryList + @ViewChildren + ElementRef</h3>
<div>
  <div #pname>Mohit</div>
  <div #pname>Anup</div>
  <div #pname>Nilesh</div>
</div>

<h3>QueryList + @ViewChildren + Component</h3>
<div>
  <writer name="Krishna" book="Angular Tutorials" #bkWriter></writer> <br/>
  <writer name="Mahesh" book="Java Tutorials" #bkWriter></writer> <br/>
  <writer name="Krishna" book="jQuery Tutorials" #bkWriter></writer> <br/>
  <writer name="Bramha" book="Hibernate Tutorials" #bkWriter *ngIf="showAllWriter"></writer
  <writer name="Vishnu" book="Spring Tutorials" #bkWriter *ngIf="showAllWriter"></writer>
</div>
<button (click)="onShowAllWriters()" >
  <label *ngIf="!showAllWriter">Show More</label>
  <label *ngIf="showAllWriter">Show Less</label>
</button>
```

writer.component.ts

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

@Component({
  selector: 'writer',
  template: `
       {{writerName}} - {{bookName}}
  `
})
export class WriterComponent {
   @Input('name') writerName: string;
   @Input('book') bookName: string;
}
```

message.directive.ts

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

@Directive({
   selector: '[cpMsg]'
})
export class MessageDirective {
   constructor(public viewContainerRef: ViewContainerRef) { }
}
```

Bootstrapping	`import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';`
`platformBrowserDynamic().bootstrapModule(AppModule);`	Bootstraps the app, using the root component from the specified `NgModule`.

NgModules	`import { NgModule } from '@angular/core';`
`@NgModule({ declarations: ..., imports: ..., exports: ..., providers: ..., bootstrap: ...}) class MyModule {}`	Defines a module that contains components, directives, pipes, and providers.
`declarations: [MyRedComponent, MyBlueComponent, MyDatePipe]`	List of components, directives, and pipes that belong to this module.
`imports: [BrowserModule, SomeOtherModule]`	List of modules to import into this module. Everything from the imported modules is available to `declarations` of this module.
`exports: [MyRedComponent, MyDatePipe]`	List of components, directives, and pipes visible to modules that import this module.
`providers: [MyService, { provide: ... }]`	List of dependency injection providers visible both to the contents of this module and to importers of this module.
`entryComponents: [SomeComponent, OtherComponent]`	List of components not referenced in any reachable template, for example dynamically created from code.
`bootstrap: [MyAppComponent]`	List of components to bootstrap when this module is bootstrapped.

Template syntax	
`<input [value]="firstName">`	Binds property `value` to the result of expression `firstName`.
`<div [attr.role]="myAriaRole">`	Binds attribute `role` to the result of expression `myAriaRole`.
`<div [class.extra-sparkle]="isDelightful">`	Binds the presence of the CSS class `extra-sparkle` on the element to the truthiness of the expression `isDelightful`.
`<div [style.width.px]="mySize">`	Binds style property `width` to the result of expression `mySize` in pixels. Units are optional.
`<button (click)="readRainbow($event)">`	Calls method `readRainbow` when a click event is triggered on this button element (or its children) and passes in the event object.
`<div title="Hello {{ponyName}}">`	Binds a property to an interpolated string, for example, "Hello Seabiscuit". Equivalent to: `<div [title]="'Hello ' + ponyName">`
`<p>Hello {{ponyName}}</p>`	Binds text content to an interpolated string, for example, "Hello Seabiscuit".
`<my-cmp [(title)]="name">`	Sets up two-way data binding. Equivalent to: `<my-cmp [title]="name" (titleChange)="name=$event">`

`<video #movieplayer ...>`
`<button (click)="movieplayer.play()">`
`</video>`
	Creates a local variable `movieplayer` that provides access to the `video` element instance in data-binding and event-binding expressions in the current template.

`<p *myUnless="myExpression">...</p>`	The * symbol turns the current element into an embedded template. Equivalent to: `<ng-template [myUnless]="myExpression"><p>...</p></ng-template>`
`<p>Card No.: {{cardNumber | myCardNumberFormatter}}</p>`	Transforms the current value of expression `cardNumber` via the pipe called `myCardNumberFormatter`.
`<p>Employer: {{employer?.companyName}}</p>`	The safe navigation operator (`?`) means that the `employer` field is optional and if `undefined`, the rest of the expression should be ignored.
`<svg:rect x="0" y="0" width="100" height="100"/>`	An SVG snippet template needs an `svg:` prefix on its root element to disambiguate the SVG element from an HTML component.

`<svg>`
`<rect x="0" y="0" width="100" height="100"/>`
`</svg>`
	An `<svg>` root element is detected as an SVG element automatically, without the prefix.

Built-in directives	`import { CommonModule } from '@angular/common';`
`<section *ngIf="showSection">`	Removes or recreates a portion of the DOM tree based on the `showSection` expression.
`<li *ngFor="let item of list">`	Turns the li element and its contents into a template, and uses that to instantiate a view for each item in list.

`<div [ngSwitch]="conditionExpression">`
`<ng-template [ngSwitchCase]="case1Exp">...</ng-template>`
`<ng-template ngSwitchCase="case2LiteralString">...</ng-template>`
`<ng-template ngSwitchDefault>...</ng-template>`
`</div>`
	Conditionally swaps the contents of the div by selecting one of the embedded templates based on the current value of `conditionExpression`.

`<div [ngClass]="{'active': isActive, 'disabled': isDisabled}">`	Binds the presence of CSS classes on the element to the truthiness of the associated map values. The right-hand expression should return {class-name: true/false} map.
`<div [ngStyle]="{'property': 'value'}">`	Allows you to assign styles to an HTML element using CSS. You can use CSS

`<div [ngStyle]="{ property: 'value' }">` `<div [ngStyle]="dynamicStyles()">`	Allows you to assign styles to an HTML element using CSS. You can use CSS directly, as in the first example, or you can call a method from the component.

Forms	`import { FormsModule } from '@angular/forms';`
`<input [(ngModel)]="userName">`	Provides two-way data-binding, parsing, and validation for form controls.

Class decorators	`import { Directive, ... } from '@angular/core';`
`@Component({...})` `class MyComponent() {}`	Declares that a class is a component and provides metadata about the component.
`@Directive({...})` `class MyDirective() {}`	Declares that a class is a directive and provides metadata about the directive.
`@Pipe({...})` `class MyPipe() {}`	Declares that a class is a pipe and provides metadata about the pipe.
`@Injectable()` `class MyService() {}`	Declares that a class has dependencies that should be injected into the constructor when the dependency injector is creating an instance of this class.

Directive configuration	`@Directive({ property1: value1, ... })`
`selector: '.cool-button:not(a)'`	Specifies a CSS selector that identifies this directive within a template. Supported selectors include `element`, `[attribute]`, `.class`, and `:not()`. Does not support parent-child relationship selectors.
`providers: [MyService, { provide: ... }]`	List of dependency injection providers for this directive and its children.

Component configuration	`@Component` extends `@Directive`, so the `@Directive` configuration applies to components as well
`moduleId: module.id`	If set, the `templateUrl` and `styleUrl` are resolved relative to the component.
`viewProviders: [MyService, { provide: ... }]`	List of dependency injection providers scoped to this component's view.
`template: 'Hello {{name}}'` `templateUrl: 'my-component.html'`	Inline template or external template URL of the component's view.
`styles: ['.primary {color: red}']` `styleUrls: ['my-component.css']`	List of inline CSS styles or external stylesheet URLs for styling the component's view.

Class field decorators for directives and components	`import { Input, ... } from '@angular/core';`
`@Input() myProperty;`	Declares an input property that you can update via property binding (example: `<my-cmp [myProperty]="someExpression">`).
`@Output() myEvent = new EventEmitter();`	Declares an output property that fires events that you can subscribe to with an event binding (example: `<my-cmp (myEvent)="doSomething()">`).
`@HostBinding('class.valid') isValid;`	Binds a host element property (here, the CSS class `valid`) to a directive/component property (`isValid`).
`@HostListener('click', ['$event']) onClick(e) {...}`	Subscribes to a host element event (`click`) with a directive/component method (`onClick`), optionally passing an argument (`$event`).
`@ContentChild(myPredicate) myChildComponent;`	Binds the first result of the component content query (`myPredicate`) to a

	property (`myChildComponent`) of the class.
`@ContentChildren(myPredicate) myChildComponents;`	Binds the results of the component content query (`myPredicate`) to a property (`myChildComponents`) of the class.
`@ViewChild(myPredicate) myChildComponent;`	Binds the first result of the component view query (`myPredicate`) to a property (`myChildComponent`) of the class. Not available for directives.
`@ViewChildren(myPredicate) myChildComponents;`	Binds the results of the component view query (`myPredicate`) to a property (`myChildComponents`) of the class. Not available for directives.

Directive and component change detection and lifecycle hooks	(implemented as class methods)
`constructor(myService: MyService, ...) { ... }`	Called before any other lifecycle hook. Use it to inject dependencies, but avoid any serious work here.
`ngOnChanges(changeRecord) { ... }`	Called after every change to input properties and before processing content or child views.
`ngOnInit() { ... }`	Called after the constructor, initializing input properties, and the first call to `ngOnChanges`.
`ngDoCheck() { ... }`	Called every time that the input properties of a component or a directive are checked. Use it to extend change detection by performing a custom check.
`ngAfterContentInit() { ... }`	Called after `ngOnInit` when the component's or directive's content has been initialized.
`ngAfterContentChecked() { ... }`	Called after every check of the component's or directive's content.
`ngAfterViewInit() { ... }`	Called after `ngAfterContentInit` when the component's views and child views / the view that a directive is in has been initialized.
`ngAfterViewChecked() { ... }`	Called after every check of the component's views and child views / the view that a directive is in.
`ngOnDestroy() { ... }`	Called once, before the instance is destroyed.

Dependency injection configuration	
`{ provide: MyService, useClass: MyMockService }`	Sets or overrides the provider for `MyService` to the `MyMockService` class.
`{ provide: MyService, useFactory: myFactory }`	Sets or overrides the provider for `MyService` to the `myFactory` factory function.
`{ provide: MyValue, useValue: 41 }`	Sets or overrides the provider for `MyValue` to the value `41`.

Routing and navigation	`import { Routes, RouterModule, ... } from '@angular/router';`

`const routes: Routes = [`
`{ path: '', component: HomeComponent },`
`{ path: 'path/:routeParam', component: MyComponent },`
`{ path: 'staticPath', component: ... },`
`{ path: '**', component: ... },`
`{ path: 'oldPath', redirectTo: '/staticPath' },`
`{ path: ..., component: ..., data: { message: 'Custom' } }`
`]);`
 	
`const routing = RouterModule.forRoot(routes);`
	Configures routes for the application. Supports static, parameterized, redirect, and wildcard routes. Also supports custom route data and resolve.

`<router-outlet></router-outlet>`	Marks the location to load the component of the active route.

`<router-outlet name="aux"></router-outlet>`	

`<a routerLink="/path">`
`<a [routerLink]="[ '/path', routeParam ]">`
`<a [routerLink]="[ '/path', { matrixParam: 'value' } ]">`
`<a [routerLink]="[ '/path' ]" [queryParams]="{ page: 1 }">`
`<a [routerLink]="[ '/path' ]" fragment="anchor">`
	Creates a link to a different view based on a route instruction consisting of a route path, required and optional parameters, query parameters, and a fragment. To navigate to a root route, use the `/` prefix; for a child route, use the `./`prefix; for a sibling or parent, use the `../` prefix.

`<a [routerLink]="[ '/path' ]" routerLinkActive="active">`	The provided classes are added to the element when the `routerLink` becomes the current active route.

`class CanActivateGuard implements CanActivate {`
`canActivate(`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`): Observable<boolean>|Promise<boolean>|boolean { ... }`
`}`
 	
`{ path: ..., canActivate: [CanActivateGuard] }`
	An interface for defining a class that the router should call first to determine if it should activate this component. Should return a boolean or an Observable/Promise that resolves to a boolean.

`class CanDeactivateGuard implements CanDeactivate<T> {`
`canDeactivate(`
`component: T,`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`): Observable<boolean>|Promise<boolean>|boolean { ... }`
`}`
 	
`{ path: ..., canDeactivate: [CanDeactivateGuard] }`
	An interface for defining a class that the router should call first to determine if it should deactivate this component after a navigation. Should return a boolean or an Observable/Promise that resolves to a boolean.

`class CanActivateChildGuard implements CanActivateChild {`
`canActivateChild(`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`): Observable<boolean>|Promise<boolean>|boolean { ... }`
`}`
 	
`{ path: ..., canActivateChild: [CanActivateGuard],`
`children: ... }`
	An interface for defining a class that the router should call first to determine if it should activate the child route. Should return a boolean or an Observable/Promise that resolves to a boolean.

`class ResolveGuard implements Resolve<T> {`
`resolve(`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`): Observable<any>|Promise<any>|any { ... }`
`}`
 	
`{ path: ..., resolve: [ResolveGuard] }`
	An interface for defining a class that the router should call first to resolve route data before rendering the route. Should return a value or an Observable/Promise that resolves to a value.

`class CanLoadGuard implements CanLoad {`
`canLoad(`
`route: Route`
`): Observable<boolean>|Promise<boolean>|boolean { ... }`
`}`
 	
`{ path: ..., canLoad: [CanLoadGuard], loadChildren: ... }`
	An interface for defining a class that the router should call first to check if the lazy loaded module should be loaded. Should return a boolean or an Observable/Promise that resolves to a boolean.