**What are the differences between AngularJS (angular 1.x) and Angular (Angular 2.x and beyond)?**

* Angular 2 is not the upgrade of Angular 1. It's completly rewritten.
* Angular 2 is using Type Script which is a super set of JavaScript.
* Angular 1 was not built with mobile suppot in mind where as Angular 2 is mobile oriented.
* Angular 1 core concept was $ scope and you will not find any $scope in Angular 2. Angular 2 using zone.js to detect changes.
* Angular 1 controllers are gone. Controllers are replaced with "components in Angular 2".

**What is a component? Why would you use it?**

Component decorator allows you to mark a class as an Angular component and provide additional metadata that determines how the component should be processed, instantiated and used at runtime. ... An Angular application is a tree of Angular components. Angular components are a subset of directives.

**What is the minimum definition of a component?**

A component is a building block of an application that controls a part of your “view”. For example, you may have a component to display a list of active chats in a messaging app (which, in turn, may have child components to display the details of the chat or the actual conversation). Or you may have an input field that uses Angular’s two-way data binding to keep your markup in sync with your JavaScript code. Or, at the most elementary level, you may have a component that substitutes an HTML template with no special functionality just because you wanted to break down something complex into smaller, more manageable parts.

import {Component} from 'angular2/core'; // we are importing component class from core Angular package

@Component({ //Decorator we are using TypeScript Decorator to attach some metadata to AppComponent class. ( Decorator is simply a function that extends your class with Angular code so that it becomes an Angular component. Otherwise, it would just be a plain class with no relation to the Angular framework)

selector: 'my-app', // defined a selector, which is the tag name used in the HTML code so that Angular can find where to insert your component, and a template, which is applied to the inner contents of the selector tag. You may notice that we also used interpolation to bind the component data and display the value of the public variable in the template

template: '<h1>{{ title }}</h1>'

})

export class AppComponent { // Here exporting AppComponent class so that we can import elsewhere.

title = 'Hello World!';

}

**What is a module, and what does it contain?**

In Angular, a module is a mechanism to group components, directives, pipes and services that are related, in such a way that can be combined with other modules to create an application.

Another analogy to understand Angular modules is classes. In a class, we can define public or private methods. The public methods are the API that other parts of our code can use to interact with it while the private methods are implementation details that are hidden. In the same way, a module can export or hide components, directives, pipes and services. The exported elements are meant to be used by other modules, while the ones that are not exported (hidden) are just used inside the module itself and cannot be directly accessed by other modules of our application.

import { NgModule } from '@angular/core';

@NgModule({ // we have turned the class AppModule into an Angular module just by using the NgModule decorator. The NgModule decorator requires at least three properties: imports, declarations and bootstrap.

imports: [ ... ],

declarations: [ ... ],

bootstrap: [ ... ]

})

export class AppModule { }

In the example above, we have turned the class AppModule into an Angular module just by using the NgModule decorator. The NgModule decorator requires at least three properties: imports, declarations and bootstrap.

The property imports expects an array of modules. Here's where we define the pieces of our puzzle (our application). The property declarations expects an array of components, directives and pipes that are part of the module. The bootstrap property is where we define the root component of our module. Even though this property is also an array, 99% of the time we are going to define only one component.

There are very special circumstances where more than one component may be required to bootstrap a module but we are not going to cover those edge cases here.

Here's how a basic module made up of just one component would look like:

app/app.component.ts

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

@Component({

selector: 'app-root',

template: '<h1>My Angular App</h1>'

})

export class AppComponent {}

app/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 { }

The file app.component.ts is just a "hello world" component, nothing interesting there. In the other hand, the file app.module.ts is following the structure that we've seen before for defining a module but in this case, we are defining the modules and components that we are going to be using.

The first thing that we notice is that our module is importing the BrowserModule as an explicit dependency. The BrowserModule is a built-in module that exports basic directives, pipes and services. Unlike previous versions of Angular, we have to explicitly import those dependencies to be able to use directives like \*ngFor or \*ngIf in our templates.

Given that the root (and only) component of our module is the AppComponent we have to list it in the bootstrap array. Because in the declarations property we are supposed to define all the components or pipes that make up our application, we have to define the AppComponent again there too.

Before moving on, there's an important clarification to make. There are two types of modules, root modules and feature modules.

In the same way that in a module we have one root component and many possible secondary components, in an application we only have one root module and zero or many feature modules. To be able to bootstrap our application, Angular needs to know which one is the root module. An easy way to identify a root module is by looking at the imports property of its NgModule decorator. If the module is importing the BrowserModule then it's a root module, if instead is importing the CommonModule then it is a feature module.

As developers, we need to take care of importing the BrowserModule in the root module and instead, import the CommonModule in any other module we create for the same application. Failing to do so might result in problems when working with lazy loaded modules as we are going to see in following sections.

By convention, the root module should always be named AppModule.

Bootstrapping an Application

To bootstrap our module based application, we need to inform Angular which one is our root module to perform the compilation in the browser. This compilation in the browser is also known as "Just in Time" (JIT) compilation.

main.ts

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

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

platformBrowserDynamic().bootstrapModule(AppModule);

It is also possible to perform the compilation as a build step of our workflow. This method is called "Ahead of Time" (AOT) compilation and will require a slightly different bootstrap process that we are going to discuss in another section.

**What is a service, and when will you use it?**

A service is used when a common functionality needs to be provided to various modules. For example, we could have a database functionality that could be reused among various modules. And hence you could create a service that could have the database functionality.

Modules

The following key steps need to be carried out when creating a service.

Step 1 − Create a separate class which has the injectable decorator. The injectable decorator allows the functionality of this class to be injected and used in any Angular JS module.

@Injectable()

export class classname {

}

Step 2 − Next in your appComponent module or the module in which you want to use the service, you need to define it as a provider in the @Component decorator.

@Component ({

providers : [classname]

})

Lets look at an example on how to achieve this. Following are the steps involved.

Step 1 − Create a ts file for the service called app.service.ts.

Demo ts

Step 2 − Place the following code in the file created above.

import {

Injectable

} from '@angular/core';

@Injectable()

export class appService {

getApp(): string {

return "Hello world";

}

}

Following points need to be noted about the above program.

The Injectable decorator is imported from the angular/core module.

We are creating a class called appService that is decorated with the Injectable decorator.

We are creating a simple function called getApp, which returns a simple string called Hello world.Step 3 − In the app.component.ts file, place the following code.

import {

Component

} from '@angular/core';

import {

appService

} from './app.service';

@Component ({

selector: 'demo-app',

template: '<div>{{value}}</div>',

providers: [appService]

})

export class AppComponent {

value: string = "";

constructor(private \_appService: appService) { }

ngOnInit(): void {

this.value = this.\_appService.getApp();

}

}

Following points need to be noted about the above program.

First, we import our appService module in the appComponent module.

Then, we register the service as a provider in this module.

In the constructor, we define a variable called \_appService of the type appService so that it can be called anywhere in the appComponent module.

As an example, in the ngOnInit lifecyclehook, we called the getApp function of the service and assign the output to the value property of the AppComponent class.

**6. What are the lifecycle hooks for components and directives?**

Following is a description of each lifecycle hook.

**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.

Following is an example of implementing one lifecycle hook. In the app.component.ts file, place the following code.

import {

Component

} from '@angular/core';

@Component ({

selector: 'my-app',

template: '<div> {{values}} </div> '

})

export class AppComponent {

values = '';

ngOnInit() {

this.values = "Hello";

}

}

In the above program, we are calling the ngOnInit lifecycle hook to specifically mention that the value of the this.values parameter should be set to “Hello”.

7**. What are pipes? Give me an example?**

Angular 2 also has the facility to create custom pipes. The general way to define a custom pipe is as follows.

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

@Pipe({name: 'Pipename'})

export class Pipeclass implements PipeTransform {

transform(parameters): returntype { }

}

Where,

'Pipename' − This is the name of the pipe.

Pipeclass − This is name of the class assigned to the custom pipe.

Transform − This is the function to work with the pipe.

Parameters − This are the parameters which are passed to the pipe.

Returntype − This is the return type of the pipe.

Let’s create a custom pipe that multiplies 2 numbers. We will then use that pipe in our component class.

Step 1 − First, create a file called multiplier.pipe.ts.

Multiplier

Step 2 − Place the following code in the above created file.

import {

Pipe,

PipeTransform

} from '@angular/core';

@Pipe ({

name: 'Multiplier'

})

export class MultiplierPipe implements PipeTransform {

transform(value: number, multiply: string): number {

let mul = parseFloat(multiply);

return mul \* value

}

}

Following points need to be noted about the above code.

We are first importing the Pipe and PipeTransform modules.

Then, we are creating a Pipe with the name 'Multiplier'.

Creating a class called MultiplierPipe that implements the PipeTransform module.

The transform function will then take in the value and multiple parameter and output the multiplication of both numbers.

Step 3 − In the app.component.ts file, place the following code.

import {

Component

} from '@angular/core';

@Component ({

selector: 'my-app',

template: '<p>Multiplier: {{2 | Multiplier: 10}}</p>'

})

export class AppComponent { }

Note − In our template, we use our new custom pipe.

Step 4 − Ensure the following code is placed in the app.module.ts file.

import {

NgModule

} from '@angular/core';

import {

BrowserModule

} from '@angular/platform-browser';

import {

AppComponent

} from './app.component';

import {

MultiplierPipe

} from './multiplier.pipe'

@NgModule ({

imports: [BrowserModule],

declarations: [AppComponent, MultiplierPipe],

bootstrap: [AppComponent]

})

export class AppModule {}

Following things need to be noted about the above code.

We need to ensure to include our MultiplierPipe module.

We also need to ensure it is included in the declarations section.

**Why would you use angular cli?**

Command Line Interface (CLI) can be used to create our Angular JS application. It also helps in creating a unit and end-to-end tests for the application.

**How would you debug a typescript file?**

Configure and set breakpoints in the TypeScript code.

Compile the TypeScript code into JavaScript.

Run the application in the development mode. ...

Create a debug configuration of the type JavaScript Debug:

**How do components communicate with each other?**

Components can communicate to each other in various ways, including:

* Using @Input()
* Using @Output()
* Using Services
* Parent component calling ViewChild
* Parent interacting with child using a local variable.

**How would you use http to load data from server?**

This is achieved by using APIs like XMLHttpRequest or — more recently — the Fetch API. These technologies allow web pages to directly handle making HTTP requests for specific resources available on a server, and formatting the resulting data as needed, before it is displayed.

**How do you create routes?**

Use the $routeProvider to configure different routes in your application.

The ngRoute module routes your application to different pages without reloading the entire application.

**How can you get the current state of a route?**

By the **Route** Object. A **route** object represents the **state** of the **current** active **route**. It contains parsed information of the **current** URL and the **route** records matched by the URL. The **route** object is immutable.

**How do you create two-way data binding?**

In order to achieve 2 way data binding we use [ ( ngModel ) ] , here ngModel is an angular directive

**How do you load external modules?**

LoadChildren in routing is one of the best way used to load module and components in to angular project. If you are using cli, webpack will build your modules seperatly and they will be loaded dynamically at run time using router.

**How would you display form validation errors?**

in angular you want use the element.$valid to check wheter an model is valid or not - and you use element.$error.{type} to check for a specific validation error.

**Which lifecycle hook would you use to unsubscribe an observable?**

ngOnDestroy

This method will be invoked just before Angular destroys the component.

Use this hook to unsubscribe observables and detach event handlers to avoid memory leaks.

**How are services injected to your application?**

AngularJS services are substitutable objects that are wired together using dependency injection (DI). You can use services to organize and share code across your app.

AngularJS services are:

Lazily instantiated – AngularJS only instantiates a service when an application component depends on it.

Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory.

**How would you create route parameters and access them from a component?**

Rather than define each route's component separately, use RouterOutlet which serves as a component placeholder; Angular dynamically adds the component for the route being activated into the <router-outlet></router-outlet> element.

**Basic Concepts**

**Why would you use Angular instead of another framework, e.g., React?**

Angular is a full framework with all the tooling and best practices designed on top of it. It provides significantly more opinions and functionality out of the box

React on the other hand is just a small view library that you would need while making an app. This might be both good, and a bad thing, and the article should help you to finally make the right choice.

With React, we typically pull many other libraries off the shelf to build a real app. We will likely want libraries for routing, enforcing unidirectional flows, web API calls, testing, dependency management, and so on. The number of decisions is overwhelming.

Angular is easy to scale thanks to its design as well as a powerful CLI.

React claims to be more testable and therefore scalable than vue and I think that is partly true.

Angular has free open source licence. It is supported by Google, making it probably the best choice for a company, as there is also less things that differ between angular practices.

React used to come with a patent clause that for some businesses was quite a big issue, but recently they switched to MIT.

**What is the difference between an observable and a promise?**

An Observable is like a Stream (in many languages) and allows to pass zero or more events where the callback is called for each event. Often Observable is preferred over Promise because it provides the features of Promise and more. With Observable it doesn’t matter if you want to handle 0, 1, or multiple events. You can utilize the same API in each case. Observable also has the advantage over Promise to be cancelable. If the result of an HTTP request to a server or some other expensive async operation isn’t needed anymore, the Subscription of an Observable allows to cancel the subscription, while a Promise will eventually call the success or failed callback even when you don’t need the notification or the result it provides anymore. Observable provides operators like map, forEach, reduce, like an array. There are also powerful operators like retry(), or replay()that are often quite handy.

Observable is more powerful way of handling http asynchronous requests.

A Promise handles a single event when an async operation completes or fails. There are Promise libraries out there that support cancellation, but ES6 Promise doesn’t so far.

**Promises vs Observables**

Promises:  
returns a single value  
not cancellable

Observables:  
works with multiple values over time  
cancellable  
supports map, filter, reduce and similar operators  
proposed feature for ES 2016  
use Reactive Extensions (RxJS)  
an array whose items arrive asynchronously over time.

**What is the difference between a component and a directive?**

Angular2 has introduced two different meta-data annotations, first is @Directive, which is used to add behaviour to an existing element. Second is @Component, which is used to create a component with attached behaviour.

We create Component with the help of @Component meta-data annotation while we create Directive with the help of @Directive meta-data annotation.

@Component is used to create new View(Shadow DOM) with attached behaviour while @Directive is used to add behavior to an existing DOM element.

With the help of @Component we can break our application into smaller components, and can do component based development while with the help of @Directive we can attach different behaviours to an existing DOM element or different existing DOM element.

@Component is used to create reusable components while @Directive is used to create reusable behaviour.

@Component requires a view via @View or template/url while @Directive does not.

@View or template/url is mandatory in the component.

**Why would you use typescript aka benefits of typescript?**

[TypeScript](http://www.typescriptlang.org/) is a superset of JavaScript which primarily provides optional static typing, classes and interfaces. One of the big benefits is to enable IDEs to provide a richer environment for spotting common errors  With TypeScript, we could eliminate a lot of type checks all together.

**Why different life cycle hooks are needed for a component/directive?**

Angular calls lifecycle hook methods on directives and components as it creates, changes, and destroys them.

A component has a lifecycle managed by Angular.

Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur.

A directive has the same set of lifecycle hooks, minus the hooks that are specific to component content and views.

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them. Developers can tap into key moments in that lifecycle by implementing one or more of the lifecycle hook interfaces in the Angular core library.

Each interface has a single hook method whose name is the interface name prefixed with ng. For example, the OnInit interface has a hook method named ngOnInit() that Angular calls shortly after creating the component:

**Why does angular use rxjs?**

RxJS is a library that allows us to easily create and manipulate streams of events and data. This makes developing complex but readable asynchronous code much easier.

In RxJS, we represent asynchronous data streams using observable sequences or also just called observables. Observables are very flexible and can be used using push or pull patterns.

RxJS plays well with Angular but instead of writing your own helper functions to bridge the two you can use rx.angular.js, a dedicated library for RxJS and AngularJS interoperability.

**What is the purpose of using zone.js?**

Zonejs is one of the core library used in Angularjs 2. Zonejs is to maintain contextual execution for single or multi-leveled asynchronous methods. So, it means it helps to keep track of the parent context of currently executing asynchronous method.

**What is the difference between ngOnInit() and the constructor() of a component?**

Constructor: constructor is a default method runs (by deafult) when component is being constructed. When we create an instance of a class that time also constructor (default method) would be called. So in other words, when component is being constructed or/and an instance is created constructor (default method) is called and relevant code written within is called. Basically, and generally in Angular2 it used to inject things like services when component is being constructed for the further use.

OnInit: ngOnInit is component's life cycle hook which runs first after constructor (default method) when component is being initialized.

So, our constructor will be called first and Oninit will be called later after constructor method.

**When will ngOnInit() be called? How would you make use of ngOnInit()?**

The ngOnInit method runs after the constructor method, meaning that all of the injected dependencies will be resolved and all of the class members will be defined. This makes it the perfect place to do any of the initialization work/logic for the component.

**What are the benefits of using formBuilder?**

Angular has a new helper Class called FormBuilder. FormBuilder allows us to explicitly declare forms in our components. This allows us to also explicitly list each form control’s validators.

**Intermediate level**

**How do components communicate with each other?**

A.)As Angular is built using a component-based application structure, it’s possible to pass data between components.

You can pass data to child components using `@Input()`

You can capture data from child components using `@Output()`

As applications grow, parent:child component communication can become difficult.

You can use an external data store such as ngrx to architect larger applications.

**How will you inject custom header in your http call?**

A.)$http headers, a native tool of AngularJS to inquiry remote endpoints via HTTP protocol, may be set to a global level, i.e. they are inherited by all methods (GET, POST, DELETE, PUT, PATCH), or may be inherited by a single method.

Setting an header

The following example shows the setting of an header for a single request, i.e. only when it is executed (e.g. inside a controller):

$http({

method: 'POST',

url: 'http://my-endpoint/login',

headers: {'Authorization': 'xxxyyyzzz'}

});

**How do you identify a structural directive in html?**

A.)Structural directives are easy to recognize. An asterisk (\*) precedes the directive attribute name as in this example.

<div \*ngIf="hero" >{{hero.name}}</div>

**How would you select all the child components' elements?**

A.)If a component, when used, contains child React components or React nodes inside of the component

(e.g., <Parent><Child /></Parent> or <Parent><span>test<span></Parent>)

these React node or component instances can be accessed by using this.props.children.

**How would you cache an observable data?**

A.)One simple approach is to capture the result of the original http call in a cache variable and

not resubscribe to the observable if we have a valid value in the cache variable.

It turns out we can easily add caching to the observable by adding publishReplay(1) and refCount.

getFriends(){

if(!this.\_friends){

this.\_friends = this.\_http.get('./components/rxjs-caching/friends.json')

.map((res:Response) => res.json().friends)

.publishReplay(1)

.refCount();

}

return this.\_friends;

}

publishReplay(1) tells rxjs to cache the most recent value which is perfect for single value http calls.

refCount() is used to keep the observable alive for as long as there are subscribers.

**How would you save data from a form control?**

A.)In order to do that we are going to transform our PersonDetailsComponent into a form and

add some validation on top to ensure that the information that we save is correct.

**How Event Emitters works in Angular?**

A.)The first thing you can learn from the source code is that you can pass a boolean

to EventEmitter that will determine whether to send events in a

synchronous or asynchronous way. ( The default is synchronous ).

export declare class EventEmitter<T> extends Subject<T> {

\_\_isAsync: boolean;

constructor(isAsync?: boolean);

emit(value?: T): void;

subscribe(generatorOrNext?: any, error?: any, complete?: any): any;

}

**How do you mock a service to inject in a unit test?**

A.)Getting an instance of the actual service using an inject block and spying methods of the service.

Implementing a mock service using $provide.

**Intermediate Level**

**Core Concepts Understandability Questions:**

**Tell me about feature module and shared module?**

Ans)FEATURE MODULES : We can now implement our feature modules, that have everyone a specific purphose in the application.

The module deals to import the modules that needs and define with the static method forChild the root paths for the module.

Then declare the components and provide the services that belongs to the module.

**SHARED MODULES:**

In every application we have some stuff that we want to share, libraries such as translations, bootstrap, rxjs, and angular modules like FormsModule and even CommonsModule.

If importing the modules is obvious, we can’t says the same thing about the export; we need to export the modules in order to make them shared; otherwise the others modules couldn’t access to these libraries

**Why angular uses decorator?**

Ans) There are two ways to register decorators

> $provide.decorator, and

> module.decorator

Each provide access to a $delegate, which is the instantiated service/directive/filter, prior to being passed to the service that required it.

$provide.decorator: The $provide service has a number of methods for registering components with the $injector. Many of these functions are also exposed on angular.Module.

An AngularJS service is a singleton object created by a service factory. These service factories are functions which, in turn, are created by a service provider. The service providers are constructor functions. When instantiated they must contain a property called $get, which holds the service factory function.

When you request a service, the $injector is responsible for finding the correct service provider, instantiating it and then calling its $get service factory function to get the instance of the service.

Often services have no configuration options and there is no need to add methods to the service provider. The provider will be no more than a constructor function with a $get property. For these cases the $provide service has additional helper methods to register services without specifying a provider.

provider(name, provider) - registers a service provider with the $injector

constant(name, obj) - registers a value/object that can be accessed by providers and services.

value(name, obj) - registers a value/object that can only be accessed by services, not providers.

factory(name, fn) - registers a service factory function that will be wrapped in a service provider object, whose $get property will contain the given factory function.

service(name, Fn) - registers a constructor function that will be wrapped in a service provider object, whose $get property will instantiate a new object using the given constructor function.

decorator(name, decorFn) - registers a decorator function that will be able to modify or replace the implementation of another service.

Module.decorator :All of the other module-level API methods - service, factory, run, config, etc - can execute in any arbitrary order. However, the new .decorator() method has to run after the target service is defined. This is because the feature has been merely piped into the $provide.decorator() method, which used to be required to run in the configuration phase, after all services have been defined. Now that it can be called at any time, the necessary $get() method doesn't always exist.

**Why do you need type definitions?**

Ans) To avoid cluttering the list of suggestions as you type in your IDE, all interfaces reside in their respective module namespace:

ng for AngularJS' ng module

ng.auto for AUTO

ng.cookies for ngCookies

ng.mock for ngMock

ng.resource for ngResource

ng.sanitize for ngSanitize.

**Which components will be notified when an event is emitted?**

Ans) Every time you access app component, you will see dashboard component emit an event. You can use a Service or a Directive to emit those events as well. We can the concept of a shared service to get around the problem of not being able to bubble output properties in Angular 2.

**Why would you export from ngModule?**

Ans) It enables an Angular module to expose some of its components/directives/pipes to the other modules in the applications. Without it, the components/directives/pipes defined in a module could only be used in that module.

Export declarable classes that components in other NgModules are able to reference in their templates. These are your public classes. If you don't export a class, it stays private, visible only to other component declared in this NgModule.

You can export any declarable class—components, directives, and pipes—whether it's declared in this NgModule or in an imported NgModule.

You can re-export entire imported NgModules, which effectively re-exports all of their exported classes. An NgModule can even export a module that it doesn't import.

**Why is it bad if SharedModule provides a service to a lazy loaded module?**

Ans) Lazy loaded modules have their own root scope. Providers added to lazy loaded modules get an instance in that root scope instead of the root scope of the application. If you add the provider to a module that is not lazy loaded, only a single instance at the application root scope will be created.

This question is addressed in the Why UserService isn't shared section of the NgModules guide, which discusses the importance of keeping providers out of the SharedModule.

Suppose the UserService was listed in the NgModule's providers (which it isn't). Suppose every NgModule imports this SharedModule (which they all do).

When the app starts, Angular eagerly loads the AppModule and the ContactModule.

Both instances of the imported SharedModule would provide the UserService. Angular registers one of them in the root app injector (see above). Then some component injects UserService, Angular finds it in the app root injector, and delivers the app-wide singleton UserService. No problem.

Now consider the HeroModule which is lazy loaded!

When the router lazy loads the HeroModule, it creates a child injector and registers the UserService provider with that child injector. The child injector is not the root injector.

When Angular creates a lazy HeroComponent, it must inject a UserService. This time it finds a UserService provider in the lazy module's child injector and creates a new instance of the UserService. This is an entirely different UserService instance than the app-wide singleton version that Angular injected in one of the eagerly loaded components.

**Can you explain the difference between ActivatedRoute and RouterState?**

Ans) 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. We can access the current RouterState from anywhere in our application using the Router service and the routerState property.

RouterState is the current state of the router including a tree of the currently activated routes in our application along convenience methods for traversing the route tree.

**Essential Terminology Questions:**

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

Ans) forRoot creates a module that contains all the directives, the given routes, and the router service itself.

> forChild creates a module that contains all the directives and the given routes, but does not include the router service. It registers the routers and uses the router service created at the root level.

> This is important because location is a mutable global property. Having more than one object manipulating the location is not a good idea.