What is the difference between AngularJS and Angular?

AngularJS	Angular
It is based on MVC architecture	This is based on Service/Controller
Controllers	Component Based
This uses use JavaScript to build the application	Introduced the typescript which is a superset of JavaScript
Not a mobile friendly framework	Developed considering mobile platform. Mobile First approach
Difficulty in SEO friendly application development	Ease to create SEO friendly Single Page Applications by rendering the HTML at the server side.
No DI	**Dependency Injection (DI) -** Angular supports a hierarchical Dependency Injection with unidirectional tree-based change detection.
it support for one-way data binding.	In optional, it support for two-way data binding

Advantages of Angular 2in

- Angular 2 uses Hierarchical Dependency Injection system which is the major performance booster of it.
  Angular 2 implements unidirectional-tree based change detection which, again, increases the performance.
  Angular 2 uses camelCase syntax for built-in directives.
- One of the biggest advantages of Angular is Dependency Injection. In Angular 2, DI is there but now there is a different way to inject dependencies. As everything is a class in Angular, so DI is achieved via a constructor.
  Best for dynamic apps with a single page
- It offers fast development process
- Offers advanced testing features
- In most of the cases, demands less coding
- Perfect for server side rendering
- Easy to test
- Improved data binding
- Components based

Angular 4 Features List

- Smaller and faster
- It is possible to view engine and how AOT generated code looks like

- Improved nglf and ngFor
- Flat ESM/FESM modules for tree shaking and reducing size of generated bundles
- Backward compatible as PATCH increased when some troubleshooting is done even without changing the API
- Dynamic components with NgComponentOutlet to build dynamic components in a declarative way
- Forms get assigned "novalidate" automatically
- Availability of source maps for templates

Angular 6

- Angular 6 was released on May 4th, 2018. The highlights of Angular 6 include the **Angular Command Line Interface** (CLI), **The Component Development KIT** (CDK) and the **Angular Material package** update.
- Angular 6 uses the **RXJS library**, so hurray for reactive programming for web!
- **The Angular Material Design Library**
- **Angular Elementss**
  Remember the Elements package? Angular 6 fully supports it now. What it did was allow us to use Angular components outside of Angular like in JQuery or VueJS apps.
  It also supports Responsive Web Design layouts so you don't have to use other libraries like Flex Layout or even learn using the CSS Grid. It covers them all.
- **Command Line Interface (CLI)**
  The Angular **command-line interface** is now equipped with new commands such as ng-update , which updates dependencies and code, and ng-add , which helps quickly add application features and also supports turning applications into progressive web apps.
- **An Improved Service Worker, PWA** – progressive web app
- **Web Pack Updated**
- Web pack module bundler has been updated to version4. By using the scope hosting technique, modules created will now be smaller.
- **Tree Shakable Services**
  You can apply Tree shaking on services as well. How great is that!
- **Multiple Validators For Your Forms**

Node Framework

- 1. AdonisJs [GitHub Stars: 5,053]
- 2. Express.js [GitHub Stars: 41,036]
- 3. Meteor.js [GitHub Stars: 40,490]
- 4. Nest.js [GitHub Stars: 10,128]
- 5. Sails.js [GitHub Stars: 19,887]
- 6. Koa.js [GitHub Stars: 23,902]
- 7. LoopBack.js [GitHub Stars: 11,985]
- 8. Hapi.js [GitHub Stars: 10,371]
- 9. Derby.js [4,350]
- 10. Total.js [Github stars: 3,853]

JIT VS AOT

JIT	AOT
• Compiled in the browser. • Each file compiled separately. • No need to build after changing your code and before reloading the browser page. • Suitable for local development.	• Compiled by the machine itself, when build happens,via the command line (Faster). • All code compiled together, inlining HTML/CSS in the scripts. • No need to deploy the compiler (Half of Angular size).
• When application is bootstrapped in the browser, the JIT compiler performs a lot of work to analyze the components in the application at runtime and generate code in memory. When the page is refreshed, all the work that has been done is thrown away, and the JIT compiler does the work all over again.	Compilation phases • Phase 1 is code analysis. • Phase 2 is code generation. • Phase 3 is template type checking.

AOT Advantages

- More secure, original source not disclosed.
- Suitable for production builds.

Now, this **compiled** js code is compiled again by browser again so that the html can be rendered. But, *the catch here is that the features of angular has already been taken care by AOT compiler and hence the browser don't have to worry much* about component creation, change detection, Dependency Injection.
So, we have :

Faster rendering

With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first.

Fewer asynchronous requests

The compiler inlines external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.

Smaller Angular framework download size

There's no need to download the Angular compiler if the app is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.

Detect template errors earlier

The AOT compiler detects and reports template binding errors during the build step before users can see them.

Better security

AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

What are the key components of Angular?

Angular has the below key components,

i. **Component:** These are the basic building blocks of angular application to control HTML views.

ii. **Modules:** Modules are logical boundaries in your application and the application is divided into separate modules to separate the functionality of your application. Lets take an example of **app.module.ts** root module declared with **@NgModule** decorator as below,

```
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 NgModule decorator has three options
The **imports** option is used to import other dependent modules. The BrowserModule is required by default for any web based angular application.
The **declarations** option is used to define components in the respective module
The **bootstrap** option tells Angular which Component to bootstrap in the application

iii. **Templates:** This represent the views of an Angular application.

iv. **Services:** It is used to create components which can be shared across the entire application.

v. **Metadata:** This can be used to add more data to an Angular class.

What are directives?

Directives add behaviour to an existing DOM element or an existing component instance.

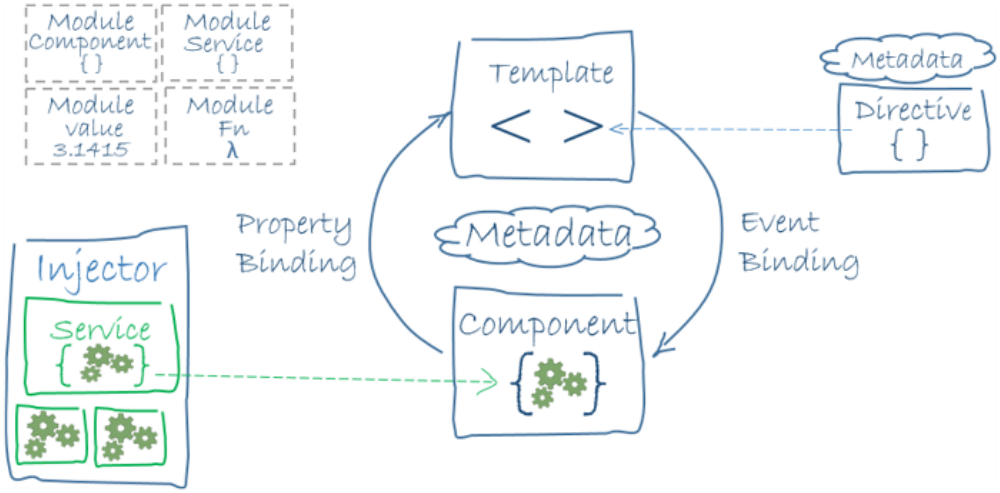
```
import { Directive, ElementRef, Input } from '@angular/core';

@Directive({ selector: '[myHighlight]' })
export class HighlightDirective {
   constructor(el: ElementRef) {
      el.nativeElement.style.backgroundColor = 'yellow';
   }
}
```

Now this directive extends HTML element behavior with a yellow background as below

```
<p myHighlight>Highlight me!</
```

Write a pictorial diagram of Angular architecture?



What is a service?

A service is used when a common functionality needs to be provided to various modules. Services allow for greater separation of concerns for your application and better modularity by allowing you to extract common functionality out of components. Let's create a repoService which can be used across components,

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

@Injectable({ // The Injectable decorator is required for dependency injection to work
  // providedIn option registers the service with a specific NgModule
  providedIn: 'root',  // This declares the service with the root app (AppModule)
})
export class RepoService{
  constructor(private http: Http){
  }

  fetchAll(){
    return this.http.get('https://api.github.com/repositories');
  }
}
```

Dependency injection (DI)

DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can *inject* a service into a component, giving the component access to that service class.
To define a class as a service in Angular, use the @Injectable() decorator to provide the metadata that allows Angular to inject it into a component as a *dependency*.

Dependency Injection

Angular's dependency injection system is hierarchical. A hierarchical dependency injection system allows us to define different boundaries or scopes for our dependencies to run in and follows the component tree structure. By default, services registered to Angular are application wide but we can also create services that are isolated to a subset of components. Our first example will show a basic service that we typically see in an Angular application.

Application-Wide Singleton Services

Typically when using Angular services, we think of services as being an application-wide singleton. Singleton services by default in Angular mean that Angular creates one instance of our service and shares that instance to all the components in our application. Let's take a look at an example of how this works.

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

@Injectable({
  providedIn: 'root'
})
export class MessageService {
  messages = [
    '10 rockets built',
    'new configurations available'
  ];

  addMessage(message: string) { ... }
}
```

By default, when we create a service with the Angular CLI, we get something similar to the code example above. On our service class, we have the @Injectable decorator letting Angular know that other components can inject and use this service. In the decorator, the providedIn property value is root. By setting the providedIn property to root Angular registers the service to the root injector. When a service registers to the root injector, it allows the service to be used application wide.

By registering services application wide, we can easily share the services and any logic contained within them. This can also be useful for sharing state or data across our entire application within multiple components. Singleton services work great for a large majority of tasks in Angular applications. Sometimes though we may want to adjust this default behavior. With Angular, we have a few options.

Component Level Services

Using component level services, we can share state and logic between isolated branches of components. Now every time we create a new RocketOrderComponent it and the RocketTotalComponent share the same instance of RocketOrderService.

Note that there is a tradeoff with this pattern of sharing data between components instead of using Inputs and Outputs. It was easier to share data between the components but they are now tightly coupled to the data source (RocketOrderService) meaning they are more difficult to reuse elsewhere in our application.

NgModule Feature Services

By leveraging lazily loaded feature modules, we can create services that are isolated and retained within only that given feature. Module level providers are beneficial if we want to make sure a service is available only within a specific feature, or we want that state to persist in only that feature module.

We covered the three main ways to register services in Angular, root application, component level, and lazy loaded feature level modules. By taking advantage of these techniques, we can safely isolate the responsibilities and state of large Angular applications.

Injector Tree

The Angular creates the Injector, when the application root module (named as AppModule) is bootstrapped. This injector is called as *root injector* and acts as a parent to all other injectors.
The *root injector* also gets its own copy of Providers. It gets it from the Providers is metadata of @NgModule of AppModule.
The AppModule loads the AppComponent, which is the root component of our application.
The AppComponent gets its own injector with a copy of Providers defined in Providers metadata of the AppComponent.

The Root Component acts as a parent to every component we create. Each of those components can contain child components creating a tree of components. The Injector is created for each of those component creating a tree of injector, which closely resembles the component tree. This is called a hierarchical pattern. The injectors also get their own copy of providers from the @component metadata.
The injector is destroyed when the associated component is destroyed.

Dependency Resolution

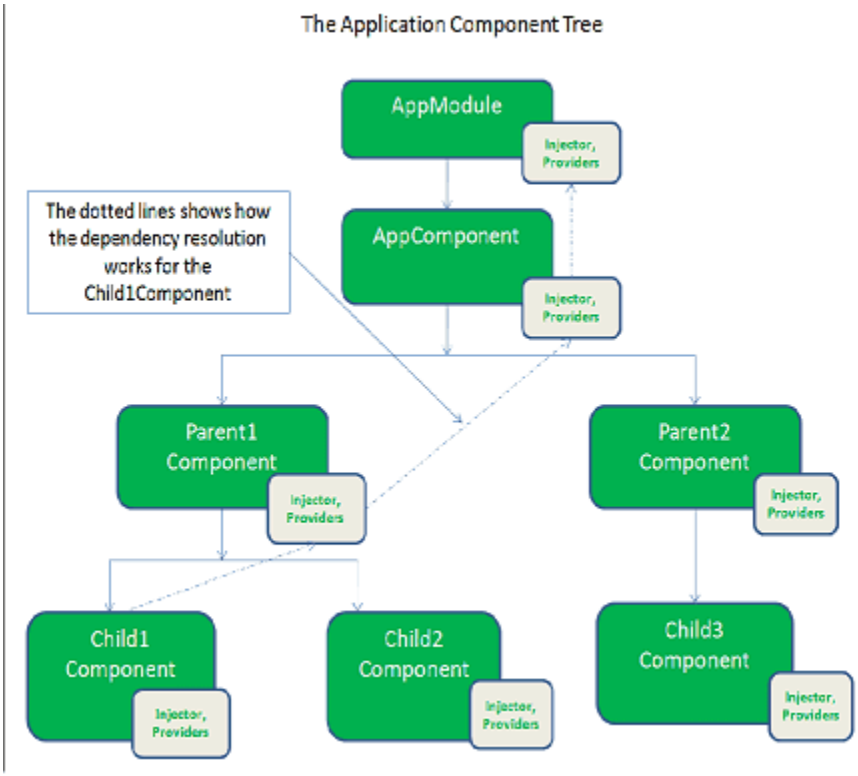
The components ask for the dependencies to be injected in the constructor using the token. The injector attached to the component looks for a provider in the Providers collection using the token. If the provider is found, it is then instantiated and injected into the component

What if the provider is not found?

The injector passes the request to the injector of the parent component. If the provider is found, the request returns the instance of the Provider. If not found then the request continues until the request reaches the topmost injector in the injector chain.

If it fails to find the service then the request returns the error *"EXCEPTION: Error in Component class – inline template caused by No provider for Service!"*

hierarchical dependency injection



We covered the three main ways to register services in Angular, root application, component level, and lazy loaded feature level modules. By taking advantage of these techniques, we can safely isolate the responsibilities and state of large Angular applications.

Service Instance

The Services are singletons *within the scope of an injector*. That is, there is at most one instance of a service in a given injector.

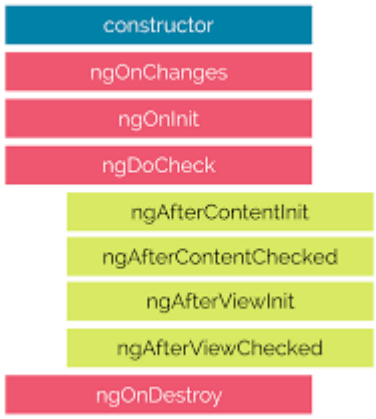
When the injector gets a request for a particular service for the first time, it creates the new instance of the service. For all the subsequent request, it will return the already created instance.

Angular Singleton Service

You can create a Singleton registration to the root injector i.e the root module (AppModule in the example).Remove the provider registration from all the components and Service by moving the Provider add it to the Providers array in the AppModule
Now the entire application will share the same instance of SharedService as shown below making the Service as Singleton

We covered the three main ways to register services in Angular, root application, component level, and lazy loaded feature level modules. By taking advantage of these techniques, we can safely isolate the responsibilities and state of large Angular applications.

What are lifecycle hooks available?



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

What is metadata?

Metadata is used to decorate a class so that it can configure the expected behavior of the class. The metadata is represented by decorators

i. **Class decorators**, e.g. @Component and @NgModule
ii. **Property decorators** Used for properties inside classes, e.g. @Input and @Output
iii. **Method decorators** Used for methods inside classes, e.g. @HostListener

```
@Component({
  selector: 'my-component',
  template: '<div>Method decorator</div>'
})
```

```
export class MyComponent {
    @HostListener('click', ['$event'])
    onHostClick(event: Event) {
        // clicked, `event` available
    }
}
```

iv. **Parameter decorators** Used for parameters inside class constructors, e.g. @Inject

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

What are the differences between Component and Directive?

Component	Directive
To register a component we use @Component meta-data annotation	To register directives we use @Directive meta-data annotation
Components are typically used to create UI widgets	Directive is used to add behavior to an existing DOM element
to break up the application into smaller components	Directive is use to design re-usable components
Only one component can be present per DOM element	Many directives can be used per DOM element
@View decorator or templateurl/template are mandatory	Directive doesn't use View

Directives

to change/modify the behavior of the HTML element in the Dom Layout

There are four types of directives in Angular,

- Components directives. - @Component
- Structural directives. – *ngIf, *ngFor
- Attribute directives. - *NgStyle, {{}}
- Custom Directive.

ng-content Directive?

<app-work>This won't work like HTML until you use ng-content Directive</app-work>.
Dynamic content. That is the simplest way to explain what ng-content provides. You use the **<ng-content></ng-content>** tag as a placeholder for that dynamic content, then when the template is parsed Angular will replace that placeholder tag with your content. Think of it like curly brace interpolation, but on a bigger scale. The technical term for this is "content projection" because you are *projecting* content from the parent component into the designated child component.
If you understand **{{myValue}}**, then you understand the basics of what ng-content does.

What is the difference between constructor and ngOnInit?

TypeScript classes has a default method called constructor which is normally used for the initialization purpose. Whereas ngOnInit method is specific to Angular, especially used to define Angular bindings. Even though constructor getting called first, it is preferred to move all of your Angular bindings to ngOnInit method. In order to use ngOnInit, you need to implement OnInit interface as below,

```
export class App implements OnInit{
  constructor(){
    //called first time before the ngOnInit()
  }

  ngOnInit(){
    //called after the constructor and called  after the first ngOnChanges()
  }
}
```

What is the purpose of async pipe?

The AsyncPipe subscribes to an observable or promise and returns the latest value it has emitted. When a new value is emitted, the pipe marks the component to be checked for changes. Let's take a time observable which continuously updates the view for every 2 seconds 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(observer =>
    setInterval(() => observer.next(new Date().toString()), 2000)
  );
}
```

What is the option to choose between inline and external template file?

You can store your component's template in one of two places. You can define it inline using the **template** property, or you can define the template in a separate HTML file and link to it in the

component metadata using the **@Component** decorator's **templateUrl** property. The choice between inline and separate HTML is a matter of taste, circumstances, and organization policy. But normally we use inline template for small portion of code and external template file for bigger views. By default, the Angular CLI generates components with a template file. But you can override that with the below command,

```
ng generate component hero -it
```

What happens if you use script tag inside template?

Angular recognizes the value as unsafe and automatically sanitizes it, which removes the **<script>** tag but keeps safe content such as the text content of the <script> tag. This way it eliminates the risk of script injection attacks. If you still use it then it will be ignored and a warning appears in the browser console. Let's take an example of innerHtml property binding which causes XSS vulnerability,

```
export class InnerHtmlBindingComponent {
  // For example, a user/attacker-controlled value from a URL.
  htmlSnippet = 'Template <script>alert("0wned")</script> <b>Syntax</b>';
}
```

What is interpolation?

Interpolation is a special syntax that Angular converts into property binding. It's a convenient alternative to property binding. It is represented by double curly braces({{}}). The text between the braces is often the name of a component property. Angular replaces that name with the string value of the corresponding component property. Let's take an example,

```
<h3>
  {{title}}
  <img src="{{url}}" style="height:30px">
</h3>
```

What are pipes?

A pipe takes in data as input and transforms it to a desired output. For example, let us take a pipe to transform a component's birthday property into a human-friendly date using **date** pipe.

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

@Component({
  selector: 'app-birthday',
  template: `<p>Birthday is {{ birthday | date }}</p>`
})
export class BirthdayComponent {
  birthday = new Date(1987, 6, 18); // June 18, 1987
}
```

What is a parameterized pipe?

A pipe can accept any number of optional parameters to fine-tune its output. The parameterized pipe can be created by declaring the pipe name with a colon (:) and then the parameter value. If the pipe accepts multiple parameters, separate the values with colons. Let's take a birthday example with a particular format(dd/mm/yyyy):

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

  @Component({
    selector: 'app-birthday',
    template: `<p>Birthday is {{ birthday | date:'dd/mm/yyyy'}}</p>` // 18/06/1987
  })
  export class BirthdayComponent {
    birthday = new Date(1987, 6, 18);
  }
```

Note: The parameter value can be any valid template expression, such as a string literal or a component property.

How do you chain pipes?

You can chain pipes together in potentially useful combinations as per the needs. Let's take a birthday property which uses date pipe(along with parameter) and uppercase pipes as below

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

      @Component({
        selector: 'app-birthday',
        template: `<p>Birthday is {{  birthday | date:'fullDate' | uppercase}} </p>` // THURSDAY, JUNE 18,
1987
      })
      export class BirthdayComponent {
        birthday = new Date(1987, 6, 18);
      }
```

What is a custom pipe?

Apart from built-inn pipes, you can write your own custom pipe with the below key characteristics,

i. A pipe is a class decorated with pipe metadata **@Pipe** decorator, which you import from the core Angular library For example,

```
  @Pipe({name: 'myCustomPipe'})
```

ii. The pipe class implements the **PipeTransform** interface's transform method that accepts an input value followed by optional parameters and returns the transformed value. The structure of pipeTransform would be as below,

```
interface PipeTransform {
  transform(value: any, ...args: any[]): any
}
```

iii. The @Pipe decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier.

```
template: `{{someInputValue | myCustomPipe: someOtherValue}}`
```

Give an example of custom pipe?

You can create custom reusable pipes for the transformation of existing value. For example, let us create a custom pipe for finding file size based on an extension,

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

@Pipe({name: 'customFileSizePipe'})
export class FileSizePipe implements PipeTransform {
  transform(size: number, extension: string = 'MB'): string {
    return (size / (1024 * 1024)).toFixed(2) + extension;
  }
}
```

Now you can use the above pipe in template expression as below,

```
template: `
    <h2>Find the size of a file</h2>
    <p>Size: {{288966 | customFileSizePipe: 'GB'}}</p>
  `
```

What is the difference between pure and impure pipe?

A pure pipe is only called when Angular detects a change in the value or the parameters passed to a pipe. For example, any changes to a primitive input value (String, Number, Boolean, Symbol) or a changed object reference (Date, Array, Function, Object). An impure pipe is called for every change detection cycle no matter whether the value or parameters changes. i.e, An impure pipe is called often, as often as every keystroke or mouse-move.

What is a bootstrapping module?

Every application has at least one Angular module, the root module that you bootstrap to launch the application is called as bootstrapping module. It is commonly known as AppModule. The default structure of AppModule generated by AngularCLI would be as follows,

```
/* JavaScript imports */
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
```

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

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

/* the AppModule class with the @NgModule decorator */
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Promises VS Observable

In Javascript, Promises are a common pattern used to handle async code elegantly. If you don't know what promises are, start there. They look something like this:

```
todoService.getTodos() // this could be any async workload
.then(todos => {
  // got returned todos
})
.catch(err => {
  // error happened
})
```

The important parts:

```
todoService.getTodos() // returns a Promise
```

Lets forget about how `getTodos()` works for now. The important thing to know is that lots of libraries support Promises and can return promises for async requests like http requests.
A Promise object implements two main methods that make it easy to handle the results of the async work. These methods are `.then()` and `.catch()`. `then` handles "successful" results and `catch` is an error handler. When the `then` handler returns data, this is called `resolving` a promise, and when it throws an error to the `catch` handler, this is called `rejecting`.

```
.then(todos => { // promise resolved with successful results })
.catch(err => { // promise rejected with an error })
```

The cool thing is, `then` and `catch` also return promises so you can **chain** them like this:

```
.then(todos => {
  return todos[0]; // get first todo
})
.then(firstTodo => {
```

```
  // got first todo!
})
```
Here's the catch: **Promises can only resolve OR reject ONCE**
This works out alright for things like http requests because http requests are fundamentally execute once and return once (success or error).

What happens when you want an elegant way to **stream** async data? Think video, audio, real-time leaderboard data, chat room messages. It would be great to be able to use promises to set up a handler that keeps accepting data as it streams in:
```
// impossible because promises only fire once!
videoService.streamVideo()
.then(videoChunk => { // new streaming chunk })
```
Welcome to the reactive pattern

In a nutshell: Promises are to async single requests, what Observables are to async streaming data.

It looks something like this:

```
videoService.getVideoStream() // returns observable, not promise
.subscribe(chunk => {  // subscribe to an observable
   // new chunk
}, err => {
  // error thrown
});
```
Looks similar to the promise pattern right? One major difference between observables and promises. Observables keep "emitting" data into the "subscription" instead of using single use `.then()` and `.catch()` handlers.
Angular's http client library returns observables by default even though you might think http fits the single use promise pattern better. But the cool thing about reactive programming (like rxJS) is that you can make observables out of other things like click events, or any arbitrary stream of events. You can then compose these streams together to do some pretty cool stuff.

For example, if you want to refresh some data every time a refresh button gets clicked AND every 60 seconds, you could set up something like this:

```
const refreshObservable = someService.refreshButtonClicked(); // observable for every time the refresh button gets clicked
const timerObservable = someService.secondsTimer(60); // observable to fire every 60 seconds

merge(
  refreshObservable,
  timerObservable
)
.pipe(
  refreshData()
)
.subscribe(data => // will keep firing with updated data! )
```
A pretty elegant way to handle a complex process! Reactive programming is a pretty big topic but this is a pretty good tool to try and visualize all the useful ways you can use observables to compose cool stuff.

Better mental model for understanding observable is consumer, publisher pattern. Consumers can asynchronously and continuely receive messages as long as subscribed to publishers. Consumers that subscribes to the Observable keeps receiving values until the Observable is completed or the consumer unsubscribes from the observable.

What is HttpClient and its benefits?

Most of the Front-end applications communicate with backend services over HTTP protocol using either XMLHttpRequest interface or the fetch() API. Angular provides a simplified client HTTP API known as **HttpClient** which is based on top of XMLHttpRequest interface. This client is avaialble from @angular/common/http package. You can import in your root module as below,

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

The major advantages of HttpClient can be listed as below,

i. Contains testability features
ii. Provides typed request and response objects
iii. Intercept request and response
iv. Supports Observalbe APIs
v. Supports streamlined error handling

How can you read full response?

The response body doesn't may not return full response data because sometimes servers also return special headers or status code which which are important for the application workflow. Inorder to get full response, you should use observe option from HttpClient,

```
getUserResponse(): Observable<HttpResponse<User>> {
  return this.http.get<User>(
    this.userUrl, { observe: 'response' });
}
```

Now HttpClient.get() method returns an Observable of typed HttpResponse rather than just the JSON data.

How do you perform Error handling?

If the request fails on the server or failed to reach the server due to network issues then HttpClient will return an error object instead of a successful reponse. In this case, you need to handle in the component by passing error object as a second callback to subscribe() method. Let's see how it can be handled in the component with an example,

```
fetchUser() {
```

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

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

What is RxJS?

RxJS is a library for composing asynchronous and callback-based code in a functional, reactive style using Observables. Many APIs such as HttpClient produce and consume RxJS Observables and also uses operators for processing observables. For example, you can import observables and operators for using HttpClient as below,

```
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators
```

What is subscribing?

An Observable instance begins publishing values only when someone subscribes to it. So you need to subscribe by calling the **subscribe()** method of the instance, passing an observer object to receive the notifications. Let's take an example of creating and subscribing to a simple observable, with an observer that logs the received message to the console.

```
Creates an observable sequence of 5 integers, starting from 1
const source = range(1, 5);

// Create observer object
const myObserver = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};

// Execute with the observer object and Prints out each item
myObservable.subscribe(myObserver);
// => Observer got a next value: 1
// => Observer got a next value: 2
// => Observer got a next value: 3
// => Observer got a next value: 4
// => Observer got a next value: 5
// => Observer got a complete notification
```

What is an observable?

An Observable is a unique Object similar to a Promise that can help manage async code. Observables are not part of the JavaScript language so we need to rely on a popular Observable library called RxJS. The observables are created using new keyword. Let see the simple example of observable,

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

const observable = new Observable(observer => {
  setTimeout(() => {
    observer.next('Hello from a Observable!');
  }, 2000);
});
```

What is an observer?

Observer is an interface for a consumer of push-based notifications delivered by an Observable. It has below structure,

```
interface Observer<T> {
  closed?: boolean;
  next: (value: T) => void;
  error: (err: any) => void;
  complete: () => void;
}
```

A handler that implements the Observer interface for receiving observable notifications will be passed as a parameter for observable as below,

```
myObservable.subscribe(myObserver);
```

Note: If you don't supply a handler for a notification type, the observer ignores notifications of that type.

What is the difference between promise and observable?

Observable	Promise
Declarative: Computation does not start until subscription so that they can be run whenever you need the result	Execute immediately on creation
Provide multiple values over time	Provide only one

Observable	Promise
Subscribe method is used for error handling which makes centralized and predictable error handling	Push errors to the child promises
Provides chaining and subscription to handle complex applications	Uses only .then() clause

What is multicasting?

Multi-casting is the practice of broadcasting to a list of multiple subscribers in a single execution. Let's demonstrate the multi-casting feature,

```
var source = Rx.Observable.from([1, 2, 3]);
var subject = new Rx.Subject();
var multicasted = source.multicast(subject);

// These are, under the hood, `subject.subscribe({...})`:
multicasted.subscribe({
  next: (v) => console.log('observerA: ' + v)
});
multicasted.subscribe({
  next: (v) => console.log('observerB: ' + v)
});

// This is, under the hood, `s
```

What are observable creation functions?

RxJS provides creation functions for the process of creating observables from things such as promises, events, timers and Ajax requests. Let us explain each of them with an example,

i. Create an observable from a promise

```
import { from } from 'rxjs'; // from function
const data = from(fetch('/api/endpoint')); //Created from Promise
data.subscribe({
 next(response) { console.log(response); },
 error(err) { console.error('Error: ' + err); },
 complete() { console.log('Completed'); }
});
```

ii. Create an observable that creates an AJAX request

```
import { ajax } from 'rxjs/ajax'; // ajax function
const apiData = ajax('/api/data'); // Created from AJAX request
// Subscribe to create the request
apiData.subscribe(res => console.log(res.status, res.response));
```

iii. Create an observable from a counter

```
import { interval } from 'rxjs'; // interval function
const secondsCounter = interval(1000); // Created from Counter value
secondsCounter.subscribe(n =>
  console.log(`Counter value: ${n}`));
```

iv. Create an observable from an event

```
import { fromEvent } from 'rxjs';
const el = document.getElementById('custom-element');
const mouseMoves = fromEvent(el, 'mousemove');
const subscription = mouseMoves.subscribe((e: MouseEvent) => {
  console.log(`Coordnitaes of mouse pointer: ${e.clientX} * ${e.clientY}`);
  });
```

What is Angular Router?

Angular Router is a mechanism in which navigation happens from one view to the next as users perform application tasks. It borrows the concepts or model of browser's application navigation.

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

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/todosList" >List of todos</a>
  <a routerLink="/completed" >Completed todos</a>
</nav>
<router-outlet></router-outlet>
```

What are active router links?

RouterLinkActive is a directive that toggles css classes for active RouterLink bindings based on the current RouterState. i.e, the Router will add CSS classes when this link is active and and remove when the link is inactive. For example, you can add them to RouterLinks as below

```
<h1>Angular Router</h1>
<nav>
 <a routerLink="/todosList" routerLinkActive="active">List of todos</a>
```

```
 <a routerLink="/completed" routerLinkActive="active">Completed todos</a>
</nav>
<router-outlet></router-outlet>
```

What are router events?

During each navigation, the Router emits navigation events through the Router.events property allowing you to track the lifecycle of the route. The sequence of router events is as below,

i. NavigationStart,
ii. RouteConfigLoadStart,
iii. RouteConfigLoadEnd,
iv. RoutesRecognized,
v. GuardsCheckStart,
vi. ChildActivationStart,
vii. ActivationStart,
viii. GuardsCheckEnd,
ix. ResolveStart,
x. ResolveEnd,
xi. ActivationEnd
xii. ChildActivationEnd
xiii. NavigationEnd,
xiv. NavigationCancel,
xv. NavigationError
xvi. Scroll

What is the purpose of Wildcard route?

If the URL doesn't match any predefined routes then it causes the router to throw an error and crash the app. In this case, you can use wildcard route. A wildcard route has a path consisting of two asterisks to match every URL. For example, you can define PageNotFoundComponent for wildcard route as below

What is Angular Universal?
Angular Universal is a server-side rendering module for Angular applications in various scenarios. This is a community driven project and available under @angular/platform-server package. Recently Angular Universal is integrated with Angular CLI.

What is JIT?

Just-in-Time (JIT) is a type of compilation that compiles your app in the browser at runtime. JIT compilation is the default when you run the ng build (build only) or ng serve (build and serve locally) CLI commands. i.e, the below commands used for JIT compilation,

```
ng build
ng serve
```

What is AOT?

Ahead-of-Time (AOT) is a type of compilation that compiles your app at build time. For AOT compilation, include the --aot option with the ng build or ng serve command as below,

```
ng build --aot
ng serve --aot
```

Note: The ng build command with the --prod meta-flag (ng build --prod) compiles with AOT by default.

Why do we need compilation process?

The Angular components and templates cannot be understood by the browser directly. Due to that Angular applications require a compilation process before they can run in a browser. For example, In AOT compilation, both Angular HTML and TypeScript code converted into efficient JavaScript code during the build phase before browser runs it.

What are the advantages with AOT?

Below are the list of AOT benefits,

i. **Faster rendering:** The browser downloads a pre-compiled version of the application. So it can render the application immediately without compiling the app.
ii. **Fewer asynchronous requests:** It inlines external HTML templates and CSS style sheets within the application javascript which eliminates separate ajax requests.
iii. **Smaller Angular framework download size:** Doesn't require downloading the Angular compiler. Hence it dramatically reduces the application payload.
iv. **Detect template errors earlier:** Detects and reports template binding errors during the build step itself
v. **Better security:** It compiles HTML templates and components into JavaScript. So there won't be any injection attacks.

What are the ways to control AOT compilation?

You can control your app compilation in two ways

i. By providing template compiler options in the tsconfig.json file
ii. By configuring Angular metadata with decorators

What are the restrictions of metadata?

In Angular, You must write metadata with the following general constraints,

i. Write expression syntax with in the supported range of javascript features
ii. The compiler can only reference symbols which are exported
iii. Only call the functions supported by the compiler
iv. Decorated and data-bound class members must be public.

What are the two phases of AOT?

The AOT compiler works in three phases,

i. **Code Analysis:** The compiler records a representation of the source
ii. **Code generation:** It handles the interpretation as well as places restrictions on what it interprets.
iii. **Validation:** In this phase, the Angular template compiler uses the TypeScript compiler to validate the binding expressions in templates.

Can I use arrow functions in AOT?

No, Arrow functions or lambda functions can't be used to assign values to the decorator properties. For example, the following snippet is invalid:

```
@Component({
  providers: [{
    provide: MyService, useFactory: () => getService()
  }]
})
```

To fix this, it has to be changed as following exported function:

```
function getService(){
  return new MyService();
}

@Component({
  providers: [{
    provide: MyService, useFactory: getService
  }]
})
```

If you still use arrow function, it generates an error node in place of the function. When the compiler later interprets this node, it reports an error to turn the arrow function into an exported function. **Note:** From Angular5 onwards, the compiler automatically performs this rewriting while emitting the .js file.

What is the purpose of metadata json files?

The metadata.json file can be treated as a diagram of the overall structure of a decorator's metadata, represented as an abstract syntax tree(AST). During the analysis phase, the AOT collector scan the metadata recorded in the Angular decorators and outputs metadata information in .metadata.json files, one per .d.ts file.

Can I use any javascript feature for expression syntax in AOT?

No, the AOT collector understands a subset of (or limited) JavaScript features. If an expression uses unsupported syntax, the collector writes an error node to the .metadata.json file. Later point of time, the compiler reports an error if it needs that piece of metadata to generate the application code.

What is metadata rewriting?

Metadata rewriting is the process in which the compiler converts the expression initializing the fields such as useClass, useValue, useFactory, and data into an exported variable, which replaces the expression. Remember that the compiler does this rewriting during the emit of the .js file but not in definition files(.d.ts file).

How do you specify angular template compiler options?

The angular template compiler options are specified as members of the **angularCompilerOptions** object in the tsconfig.json file. These options will be specified adjecent to typescript compiler options.

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
            ...
  },
  "angularCompilerOptions": {
    "fullTemplateTypeCheck": true,
    "preserveWhitespaces": true,
            ...
  }
}
```

How do you enable binding expression validation?

You can enable binding expression validation explicitly by adding the compiler option **fullTemplateTypeCheck** in the "angularCompilerOptions" of the project's tsconfig.json. It produces error messages when a type error is detected in a template binding expression. For example, consider the following component:

```
@Component({
  selector: 'my-component',
  template: '{{user.contacts.email}}'
})
class MyComponent {
  user?: User;
}
```

What is the purpose of any type cast function?

You can disable binding expression type checking using $any() type cast function(by surrounding the expression). In the following example, the error Property contacts does not exist is suppressed by casting user to the any type.

```
 template: '{{$any(user).contacts.email}}'
```

The $any() cast function also works with this to allow access to undeclared members of the component.

```
  template: '{{$any(this).contacts.email}}'
```

What is codelyzer?

Codelyzer provides set of tslint rules for static code analysis of Angular TypeScript projects. ou can run the static code analyzer over web apps, NativeScript, Ionic etc. Angular CLI has support for this and it can be use as below,

```
ng new codelyzer
ng lint
```

What is State function?

Angular's state() function is used to define different states to call at the end of each transition. This function takes two arguments: a unique name like open or closed and a style() function. For example, you can write a open state function

```
state('open', style({
  height: '300px',
  opacity: 0.5,
  backgroundColor: 'blue'
})),
```

What is Style function?

The style function is used to define a set of styles to associate with a given state name. You need to use it along with state() function to set CSS style attributes. For example, in the close state, the button has a height of 100 pixels, an opacity of 0.8, and a background color of green.

```
state('close', style({
  height: '100px',
  opacity: 0.8,
  backgroundColor: 'green'
})),
```

What is transition function?

The animation transition function is used to specify the changes that occur between one state and another over a period of time. It accepts two arguments: the first argument accepts an expression that defines the direction between two transition states, and the second argument accepts an animate() function. Let's take an example state transition from open to closed with an half second transition between states.

```
transition('open => closed', [
  animate('500ms')
]),
```

How to inject the dynamic script in angular?

Using DomSanitizer we can inject the dynamic Html,Style,Script,Url.

```
import { Component, OnInit } from '@angular/core';
import { DomSanitizer } from '@angular/platform-browser';
@Component({
  selector: 'my-app',
  template: `
      <div [innerHtml]="htmlSnippet"></div>
  `,
})
export class App {
      constructor(protected sanitizer: DomSanitizer) {}
      htmlSnippet: string = this.sanitizer.bypassSecurityTrustScript("<script>safeCode()</script>");
  }
```

What is a service worker and its role in Angular?

A service worker is a script that runs in the web browser and manages caching for an application. Starting from 5.0.0 version, Angular ships with a service worker implementation. Angular service worker is designed to optimize the end user experience of using an application over a slow or unreliable network connection, while also minimizing the risks of serving outdated content.

What are the design goals of service workers?

Below are the list of design goals of Angular's service workers,

i. It caches an application just like installing a native application
ii. A running application continues to run with the same version of all files without any incompatible files
iii. When you refresh the application, it loads the latest fully cached version
iv. When changes are published then it immediately updates in the background
v. Service workers saves the bandwidth by downloading the resources only when they changed.

hat is Angular Ivy?

Angular Ivy is a new rendering engine for Angular. You can choose to opt in a preview version of Ivy from Angular version 8.

i. You can enable ivy in a new project by using the --enable-ivy flag with the ng new command

```
ng new ivy-demo-app --enable-ivy
```

ii. You can add it to an existing project by adding enableIvy option in the angularCompilerOptions in your project's tsconfig.app.json.

```
{
  "compilerOptions": { ... },
  "angularCompilerOptions": {
    "enableIvy": true
  }
}
```

What are the features included in ivy preview?

You can expect below features with Ivy preview,

i. Generated code that is easier to read and debug at runtime
ii. Faster re-build time
iii. Improved payload size
iv. Improved template type checking

Can I use AOT compilation with Ivy?

Yes, it is a recommended configuration. Also, AOT compilation with Ivy is faster. So you need set the default build options(with in angular.json) for your project to always use AOT compilation.

```
{
  "projects": {
    "my-project": {
      "architect": {
        "build": {
          "options": {
            ...
            "aot": true,
          }
        }
      }
    }
  }
}
```

What is Angular Language Service?

The Angular Language Service is a way to get completions, errors, hints, and navigation inside your Angular templates whether they are external in an HTML file or embedded in annotations/decorators in a string. It has the ability to autodetect that you are opening an Angular file, reads your tsconfig.json file, finds all the templates you have in your application, and then provides all the language services.

How do you install angular language service in the project?

You can install Angular Language Service in your project with the following npm command

```
npm install --save-dev @angular/language-service
```

After that add the following to the "compilerOptions" section of your project's tsconfig.json

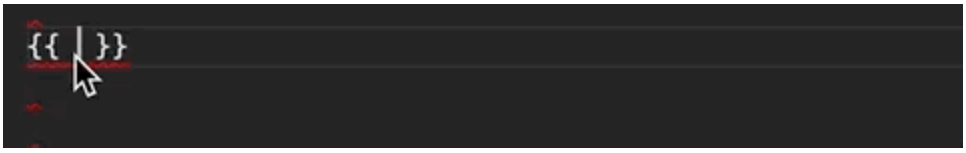
```
"plugins": [
  {"name": "@angular/language-service"}
]
```

Note: The completion and diagnostic services works for .ts files only. You need to use custom plugins for supporting HTML files.

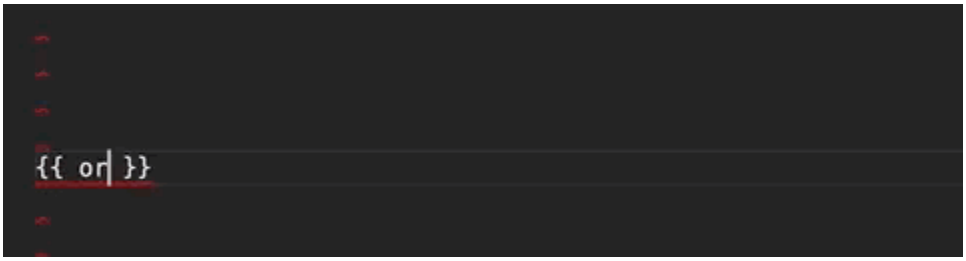
Explain the features provided by Angular Language Service?

Basically there are 3 main features provided by Angular Language Service,

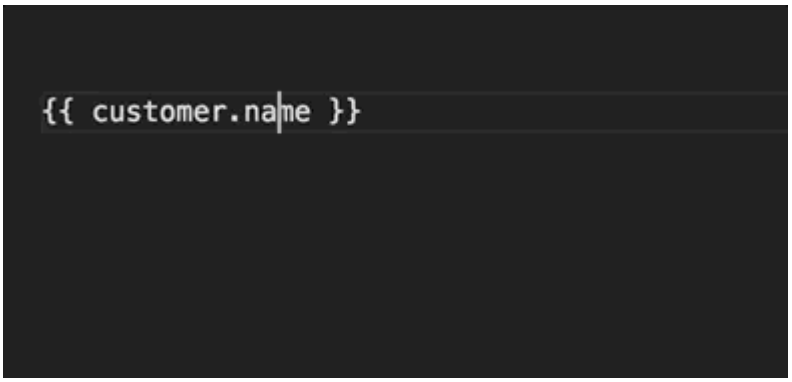
i. **Autocompletion:** Autocompletion can speed up your development time by providing you with contextual possibilities and hints as you type with in an interpolation and elements.



ii. **Error checking:** It can also warn you of mistakes in your code.



iii. **Navigation:** Navigation allows you to hover a component, directive, module and then click and press F12 to go directly to its definition.



What is a web worker? How is it gonna help me?

JavaScript is a *single-threaded* language, meaning that all your operations are run by the very same thread. If you perform compute intensive tasks on the thread, the rest of the tasks will have to way for those to be finished, leading to sloppy UI rendering and whatnot.

Web workers to the rescue.

A web worker, as defined by the W3C, is a JavaScript running on the background independently of user-interface scripts. This basically means they'll provide an execution context isolated from the user interface on which we can run anything we want.

Now, how does a web worker look like?
In the following example we'll use a web worker to calculate factorials. We won't be using any framework as of yet, but plain JavaScript and HTML.

How do you add web workers in your application?

You can add web worker anywhere in your application. For example, If the file that contains your expensive computation is src/app/app.component.ts, you can add a Web Worker using ng generate web-worker app command which will create src/app/app.worker.ts web worker file. This command will perform below actions,

i. Configure your project to use Web Workers
ii. Adds app.worker.ts to receive messages

```
addEventListener('message', ({ data }) => {
  const response = `worker response to ${data}`;
  postMessage(response);
});
```

iii. The component app.component.ts file updated with web worker file

```
if (typeof Worker !== 'undefined') {
  // Create a new
  const worker = new Worker('./app.worker', { type: 'module' });
  worker.onmessage = ({ data }) => {
    console.log('page got message: $\{data\}');
  };
  worker.postMessage('hello');
} else {
  // Web Workers are not supported in this environment.
}
```

Note: You may need to refactor your initial scaffolding web worker code for sending messages to and from.

What are the limitations with web workers?

You need to remember two important things when using Web Workers in Angular projects,

i. Some environments or platforms(like @angular/platform-server) used in Server-side Rendering, don't support Web Workers. In this case you need to provide a fallback mechanism to perform the computations to work in this environments.
ii. Running Angular in web worker using @angular/platform-webworker is not yet supported in Angular CLI.

What is Angular CLI Builder?
In Angular8, the CLI Builder API is stable and available to developers who want to customize the Angular CLI by adding or modifying commands. For example, you could supply a builder to perform an entirely new task, or to change which third-party tool is used by an existing command.

What is a builder?

A builder function ia a function that uses the Architect API to perform a complex process such as "build" or "test". The builder code is defined in an npm package. For example, BrowserBuilder runs a webpack build for a browser target and KarmaBuilder starts the Karma server and runs a webpack build for unit tests.

How do you create app shell in Angular?

An App shell is a way to render a portion of your application via a route at build time. This is useful to first paint of your application that appears quickly because the browser can render static HTML and CSS without the need to initialize JavaScript. You can achieve this using Angular CLI which generates an app shell for running server-side of your app.

```
ng generate appShell [options] (or)
ng g appShell [options]
```

What is a DI token?

A DI token is a lookup token associated with a dependency provider in dependency injection system. The injector maintains an internal token-provider map that it references when asked for a dependency and the DI token is the key to the map. Let's take example of DI Token usage,

```
const BASE_URL = new InjectionToken<string>('BaseUrl');
const injector =
  Injector.create({providers: [{provide: BASE_URL, useValue: 'http://some-domain.com'}]});
const url = injector.get(BASE_URL);
```

what is an rxjs subject in Angular

An RxJS Subject is a special type of Observable that allows values to be multicasted to many Observers. While plain Observables are unicast (each subscribed Observer owns an independent execution of the Observable), Subjects are multicast.

A Subject is like an Observable, but can multicast to many Observers. Subjects are like EventEmitters: they maintain a registry of many listeners.

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

  const subject = new Subject<number>();

  subject.subscribe({
    next: (v) => console.log(`observerA: ${v}`)
  });
  subject.subscribe({
    next: (v) => console.log(`observerB: ${v}`)
  });

  subject.next(1);
  subject.next(2);
```

What happens if I import the same module twice?

If multiple modules imports the same module then angular evaluates it only once (When it encounters the module first time). It follows this condition even the module appears at any level in a hierarchy of imported NgModules.

How do you select an element with in a component template?

You can use @ViewChild directive to access elements in the view directly. Let's take input element with a reference,

```
<input #uname>
```

and define view child directive and access it in ngAfterViewInit lifecycle hook

```
@ViewChild('uname') input;

ngAfterViewInit() {
  console.log(this.input.nativeElement.value);
}
```

How do you detect route change in Angular?

In Angular7, you can subscribe to router to detect the changes. The subscription for router events would be as below,

```
this.router.events.subscribe((event: Event) => {})
```

Let's take a simple component to detect router changes

```
import { Component } from '@angular/core';
import { Router, Event, NavigationStart, NavigationEnd, NavigationError } from '@angular/router';

@Component({
   selector: 'app-root',
```

```
  template: `<router-outlet></router-outlet>`
})
export class AppComponent {

  constructor(private router: Router) {

    this.router.events.subscribe((event: Event) => {
      if (event instanceof NavigationStart) {
        // Show loading indicator and perform an action
      }

      if (event instanceof NavigationEnd) {
        // Hide loading indicator and perform an action
      }

      if (event instanceof NavigationError) {
        // Hide loading indicator and perform an action
        console.log(event.error); // It logs an error for debugging
      }
    });
  }
}
```

Is Angular supports dynamic imports?

Yes, Angular 8 supports dynamic imports in router configuration. i.e, You can use the import statement for lazy loading the module using loadChildren method and it will be understood by the IDEs(VSCode and WebStorm), webpack, etc. Previously, you have been written as below to lazily load the feature module. By mistake, if you have typo in the module name it still accepts the string and throws an error during build time.

```
{path: 'user', loadChildren: './users/user.module#UserModulee'},
```

This problem is resolved by using dynamic imports and IDEs are able to find it during compile time itself.

```
{path: 'user', loadChildren: () => import('./users/user.module').then(m => m.UserModule)};
```

What is lazy loading?

Lazy loading is one of the most useful concepts of Angular Routing. It helps us to download the web pages in chunks instead of downloading everything in a big bundle. It is used for lazy loading by asynchronously loading the feature module for routing whenever required using the property loadChildren. Let's load both Customer and Order feature modules lazily as below,

```
const routes: Routes = [
 {
   path: 'customers',
   loadChildren: () => import('./customers/customers.module').then(module =>
module.CustomersModule)
 },
```

```
  {
    path: 'orders',
    loadChildren: () => import('./orders/orders.module').then(module => module.OrdersModule)
  },
  {
    path: '',
    redirectTo: '',
    pathMatch: 'full'
  }
];
```

What are workspace APIs?

Angular 8.0 release introduces Workspace APIs to make it easier for developers to read and modify the angular.json file instead of manually modifying it. Currently, the only supported storage3 format is the JSON-based format used by the Angular CLI. You can enable or add optimization option for build target as below,

```
import { NodeJsSyncHost } from '@angular-devkit/core/node';
import { workspaces } from '@angular-devkit/core';

async function addBuildTargetOption() {
  const host = workspaces.createWorkspaceHost(new NodeJsSyncHost());
  const workspace = await workspaces.readWorkspace('path/to/workspace/directory/', host);

  const project = workspace.projects.get('my-app');
  if (!project) {
    throw new Error('my-app does not exist');
  }

  const buildTarget = project.targets.get('build');
  if (!buildTarget) {
    throw new Error('build target does not exist');
  }

  buildTarget.options.optimization = true;

  await workspaces.writeWorkspace(workspace, host);
}

addBuildTargetOption();
```

How do you test Angular application using CLI?

Angular CLI downloads and install everything needed with the Jasmine Test framework. You just need to run ng test to see the test results. By default this command builds the app in watch mode, and launches the Karma test runner. The output of test results would be as below,

```
10% building modules 1/1 modules 0 active
```

```
...INFO [karma]: Karma v1.7.1 server started at http://0.0.0.0:9876/
...INFO [launcher]: Launching browser Chrome ...
...INFO [launcher]: Starting browser Chrome
...INFO [Chrome ...]: Connected on socket ...
Chrome ...: Executed 3 of 3 SUCCESS (0.135 secs / 0.205 secs)
```

Note: A chrome browser also opens and displays the test output in the "Jasmine HTML Reporter".

How to use polyfills in Angular application?

The Angular CLI provides support for polyfills officially. When you create a new project with the ng new command, a src/polyfills.ts configuration file is created as part of your project folder. This file includes the mandatory and many of the optional polyfills as JavaScript import statements. Let's categorize the polyfills,

i. **Mandatory polyfills:** These are installed automatically when you create your project with ng new command and the respective import statements enabled in 'src/polyfills.ts' file.
ii. **Optional polyfills:** You need to install its npm package and then create import statement in 'src/polyfills.ts' file. For example, first you need to install below npm package for adding web animations (optional) polyfill.

```
npm install --save web-animations-js
```

and create import statement in polyfill file.

```
import 'web-animations-js';
```

What are the ways to trigger change detection in Angular?

You can inject either ApplicationRef or NgZone, or ChangeDetectorRef into your component and apply below specific methods to trigger change detection in Angular. i.e, There are 3 possible ways,

i. ApplicationRef.tick(): Invoke this method to explicitly process change detection and its side-effects. It check the full component tree.
ii. NgZone.run(callback): It evaluate the callback function inside the Angular zone.
iii. ChangeDetectorRef.detectChanges(): It detects only the components and it's children.

What are the security principles in angular?

i. You should avoid direct use of the DOM APIs.
ii. You should enable Content Security Policy (CSP) and configure your web server to return appropriate CSP HTTP headers.
iii. You should Use the offline template compiler.
iv. You should Use Server Side XSS protection.
v. You should Use DOM Sanitizer.
vi. You should Preventing CSRF or XSRF attacks.

What is the reason to deprecate Web Tracing Framework?

Angular has supported the integration with the Web Tracing Framework (WTF) for the purpose of performance testing. Since it is not well maintained and failed in majority of the applications, the support is deprecated in latest releases.

What is the reason to deprecate web worker packages?

Both @angular/platform-webworker and @angular/platform-webworker-dynamic are officially deprecated, the Angular team realized it's not good practice to run the Angular application on Web worker

What are angular elements?

Angular elements are Angular components packaged as **custom elements**(a web standard for defining new HTML elements in a framework-agnostic way). Angular Elements hosts an Angular component, providing a bridge between the data and logic defined in the component and standard DOM APIs, thus, providing a way to use Angular components in non-Angular environments.

Question: **Demonstrate navigating between different routes in an Angular application.**
Answer: Following code demonstrates how to navigate between different routes in an Angular app dubbed "Some Search App":

```
const routes: Routes = [
  {
    path: '',
    loadChildren: './header/header.module#HeaderModule',
    canActivate: [AuthgaurdService]
  },
```

```
class HeaderComponent {
  constructor(private router: Router) {}
  goHome() {
    this.router.navigate(['']);
  }
  goSearch() {
    this.router.navigate(['search']);
  }
}
```

Question: **Enumerate some salient features of Angular 7.**
Answer: Unlike the previous versions of Angular, the 7th major release comes with splitting in @angular/core. This is done in order to reduce the size of the same. Typically, not each and every module

is required by an Angular developer. Therefore, in Angular 7 each split of the @angular/core will have no more than 418 modules.
Also, Angular 7 brings drag-and-drop and virtual scrolling into play. The latter enables loading as well as unloading elements from the DOM. For virtual scrolling, the latest version of Angular comes with the package. Furthermore, Angular 7 comes with a new and enhanced version of the ng-compiler.
Question: What is string interpolation in Angular?
Answer: Also referred to as moustache syntax, string interpolation in Angular refers to a special type of syntax that makes use of template expressions in order to display the component data. These template expressions are enclosed within double curly braces i.e. {{ }}.
Question: **Explain Angular Authentication and Authorization.**
Answer: The user login credentials are passed to an authenticate API, which is present on the server. Post server-side validation of the credentials, a JWT (JSON Web Token) is returned. The JWT has information or attributes regarding the current user. The user is then identified with the given JWT. This is called authentication.
Question: Can you explain the concept of scope hierarchy in Angular?
Answer: Angular organizes the $scope objects into a hierarchy that is typically used by views. This is known as the scope hierarchy in Angular. It has a root scope that can further contain one or several scopes called child scopes.
In a scope hierarchy, each view has its own $scope. Hence, the variables set by a view's view controller will remain hidden to other view controllers. Following is a typical representation of a Scope Hierarchy:

- Root $scope
    - $scope for Controller 1
    - $scope for Controller 2
    - …
    - ..
    - .
    - $scope for Controller n

Question: **How to generate a class in Angular 7 using CLI?**
Answer:

```
ng generate class Dummy [options]
```

This will generate a class named Dummy.
Question: **How do Observables differ from Promises?**
Answer: As soon as a promise is made, the execution takes place. However, this is not the case with observables because they are lazy. This means that nothing happens until a subscription is made. While promises handle a single event, observable is a stream that allows passing of more than one event. A callback is made for each event in an observable.

8. What is the difference between an Annotation and a Decorator in Angular?

Annotations in angular are "only" metadata set of the class using the Reflect Metadata library. They are used to create an "annotation" array. On the other hand, decorators are the design patterns that are used for separating decoration or modification of a class without actually altering the original source code.

15. What is a provider in Angular?

A provider is a configurable service in Angular. It is an instruction to the Dependency Injection system that provides information about the way to obtain a value for a dependency. It is an object that has a $get() method which is called to create a new instance of a service. A Provider can also contain additional methods and uses $provide in order to register new providers.

19. What is the difference between a service() and a factory()?

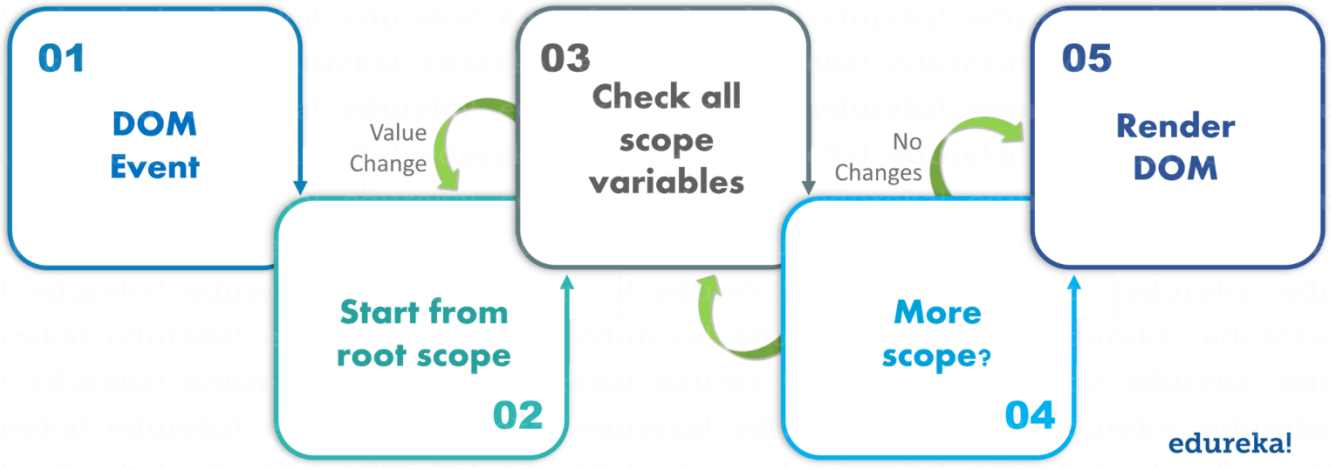
A service() in Angular is a function that is used for the business layer of the application. It operates as a constructor function and is invoked once at the runtime using the 'new' keyword. Whereas factory() is a function which works similar to the service() but is much more powerful and flexible. factory() are design patterns which help in creating Objects.

23. Explain jQLite.

jQlite is also known as **jQuery lite** is a subset of jQuery and contains all its features. It is packaged within Angular, by default. It helps Angular to manipulate the DOM in a way that is compatible cross-browser. jQLite basically implements only the most commonly needed functionality which results in having a small footprint.

24. Explain the process of digest cycle in Angular?

The digest cycle in Angular is a process of monitoring the watchlist for keeping a track of changes in the value of the watch variable. In each digest cycle, Angular compares the previous and the new version of the scope model values. Generally, this process is triggered implicitly but you can activate it manually as well by using **$apply()**.



25. What are the Angular Modules?

All the Angular apps are modular and follow a modularity system known as *NgModules*. These are the containers which hold a cohesive block of code dedicated specifically to an application domain, a workflow,

or some closely related set of capabilities. These modules generally contain components, service providers, and other code files whose scope is defined by the containing NgModule. With modules makes the code becomes more maintainable, testable, and readable. Also, all the dependencies of your applications are generally defined in modules only.

Every Angular app has at least one NgModule class, the *root module*, which is conventionally named AppModule and resides in a file named app.module.ts. You launch your app by *bootstrapping* the root NgModule.

NgModule metadata

An NgModule is defined by a class decorated with @NgModule(). The @NgModule() decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

declarations: The components, *directives*, and *pipes* that belong to this NgModule.

exports: The subset of declarations that should be visible and usable in the *component templates* of other NgModules.

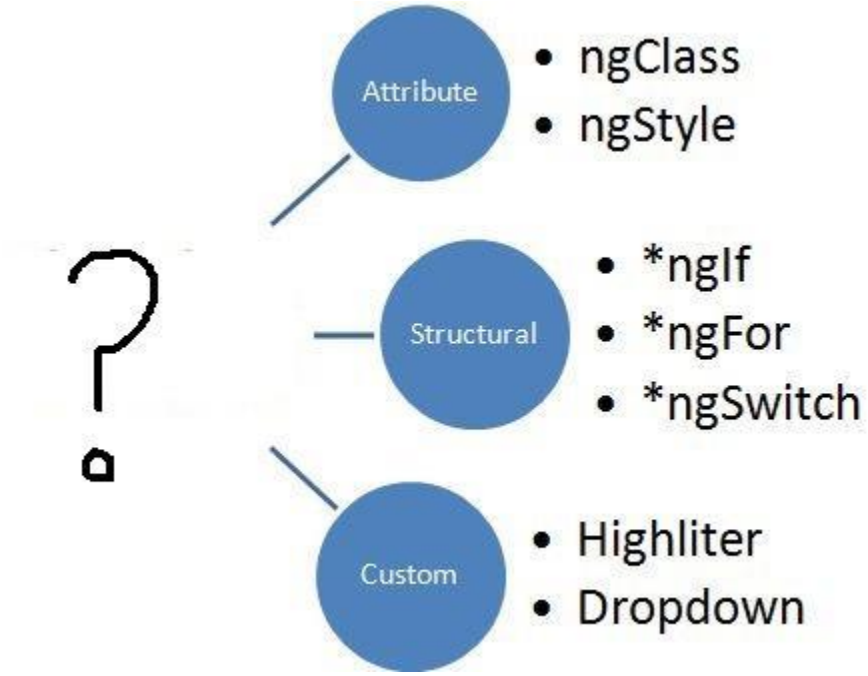
imports: Other modules whose exported classes are needed by component templates declared in *this* NgModule.

providers: Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)

bootstrap: The main application view, called the *root component*, which hosts all other app views. Only the *root NgModule* should set the bootstrap property.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Question: **Observe the following image:**



What should replace the "?"?
Answer: Directives. The image represents the types of directives in Angular; Attribute, structural, and custom.
Question: Could you explain the concept of templates in Angular?
Answer: Written with HTML, templates in Angular contains Angular-specific attributes and elements. Combined with information coming from the controller and model, templates are then further rendered to cater the user with the dynamic view.
Question: Explain the difference between an Annotation and a Decorator in Angular?
Answer: In Angular, annotations are used for creating an annotation array. They are only metadata set of the class using the Reflect Metadata library.
Decorators in Angular are design patterns used for separating decoration or modification of some class without changing the original source code.
Question: What are directives in Angular?
Answer: Directives are one of the core features of Angular. They allow an Angular developer to write new, application-specific HTML syntax. In actual, directives are functions that are executed by the Angular compiler when the same finds them in the DOM. Directives are of three types:

- Attribute Directives
- Component Directives
- Structural Directives

Question: **What are the building blocks of Angular?**
Answer: There are essentially 9 building blocks of an Angular application. These are:

1. **Components –** A component controls one or more views. Each view is some specific section of the screen. Every Angular application has at least one component, known as the root component. It is bootstrapped inside the main module, known as the root module. A component contains application logic defined inside a class. This class is responsible for interacting with the view via an API of properties and methods.
2. **Data Binding –** The mechanism by which parts of a template coordinates with parts of a component is known as data binding. In order to let Angular know how to connect both sides (template and its component), the binding markup is added to the template HTML.

3. **Dependency Injection (DI) –** Angular makes use of DI to provide required dependencies to new components. Typically, dependencies required by a component are services. A component's constructor parameters tell Angular about the services that a component requires. So, a dependency injection offers a way to supply fully-formed dependencies required by a new instance of a class.
4. **Directives –** The templates used by Angular are dynamic in nature. Directives are responsible for instructing Angular about how to transform the DOM when rendering a template. Actually, components are directives with a template. Other types of directives are attribute and structural directives.
5. **Metadata –** In order to let Angular know how to process a class, metadata is attached to the class. For doing so decorators are used.
6. **Modules –** Also known as NgModules, a module is an organized block of code with a specific set of capabilities. It has a specific application domain or a workflow. Like components, any Angular application has at least one module. This is known as the root module. Typically, an Angular application has several modules.
7. **Routing –** An Angular router is responsible for interpreting a browser URL as an instruction to navigate to a client-generated view. The router is bound to links on a page to tell Angular to navigate the application view when a user clicks on it.
8. **Services –** A very broad category, a service can be anything ranging from a value and function to a feature that is required by an Angular app. Technically, a service is a class with a well-defined purpose.
9. **Template –** Each component's view is associated with its companion template. A template in Angular is a form of HTML tags that lets Angular know that how it is meant to render the component.

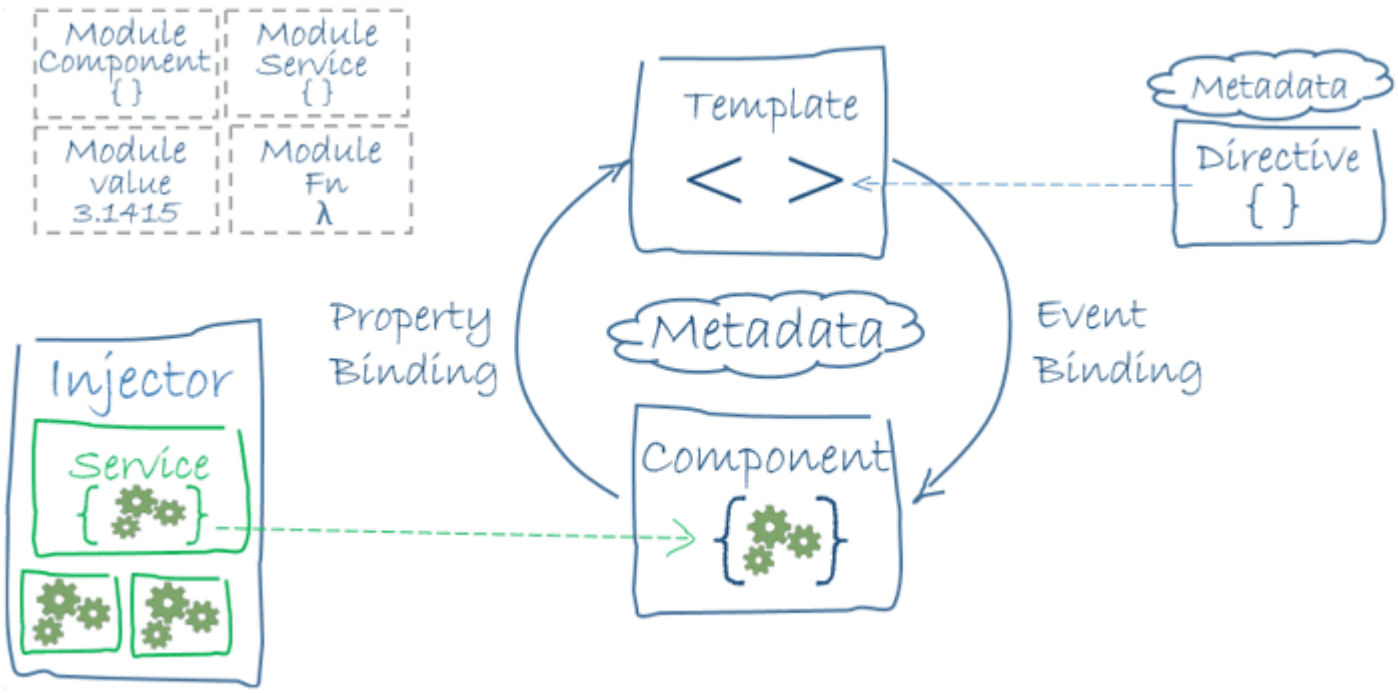
Question: Please explain the differences between Angular and jQuery?
Answer: The single biggest difference between Angular and jQuery is that while the former is a JS frontend framework, the latter is a JS library. Despite this, there are some similarities between the two, such as both features DOM manipulation and provides support for animation.
Nonetheless, notable differences between Angular and jQuery are:

- Angular has two-way data binding, jQuery does not
- Angular provides support for RESTful API while jQuery doesn't
- jQuery doesn't offer deep linking routing though Angular supports it
- There is no form validation in jQuery whereas it is present in Angular

Question: Could you explain the difference between Angular expressions and JavaScript expressions?
Answer: Although both Angular expressions and JavaScript expressions can contain literals, operators, and variables, there are some notable dissimilarities between the two. Important differences between Angular expressions and JavaScript expressions are enlisted below:

- Angular expressions support filters while JavaScript expressions do not
- It is possible to write Angular expressions inside the HTML tags. JavaScript expressions, contrarily, can't be written inside the HTML tags
- While JavaScript expressions support conditionals, exceptions, and loops, Angular expressions don't

Question: Can you give us an overview of Angular architecture?
Answer: You can draw some like this:



Here is Angular Architecture in detail: https://angular.io/guide/architecture

Question: **What is Angular Material?**

Answer: It is a UI component library. Angular Material helps in creating attractive, consistent, and fully functional web pages as well as web applications. It does so while following modern web design principles, including browser portability and graceful degradation.

Question: **What is AOT (Ahead-Of-Time) Compilation?**

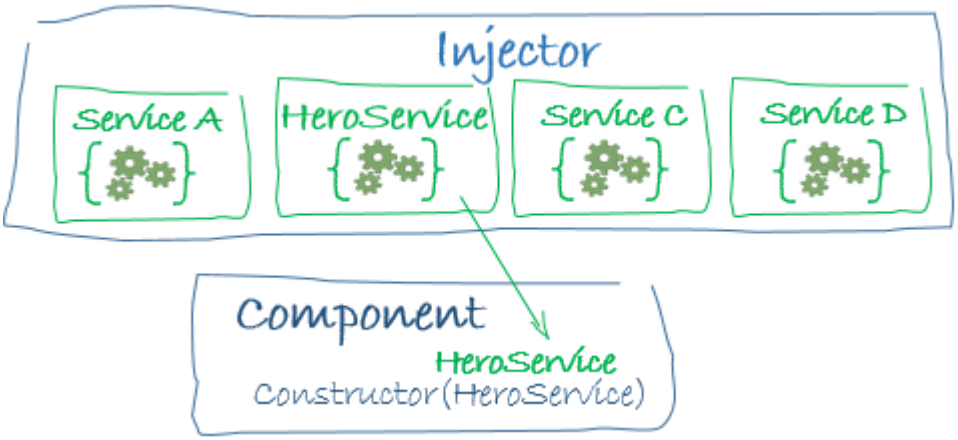
Answer: Each Angular app gets compiled internally. The Angular compiler takes in the JS code, compiles it and then produces some JS code. This happens only once per occasion per user. It is known as AOT (Ahead-Of-Time) compilation.

Question: **What is Data Binding? How many ways it can be done?**

Answer: In order to connect application data with the DOM (Data Object Model), data binding is used. It happens between the template (HTML) and component (TypeScript). There are 3 ways to achieve data binding:

1. Event Binding – Enables the application to respond to user input in the target environment
2. Property Binding – Enables interpolation of values computed from application data into the HTML
3. Two-way Binding – Changes made in the application state gets automatically reflected in the view and vice-versa. The ngModel directive is used for achieving this type of data binding.

Question: **What is demonstrated by the arrow in the following image?**



Answer: This represents a dependency injection or DI.

Question: Can you draw a comparison between the service() and the factory() functions?

Answer: Used for the business layer of the application, the *service()* function operates as a constructor function. The function is invoked at runtime using the *new* keyword.

Although the *factory()* function works in pretty much the same way as the *service()* function does, the former is more flexible and powerful. In actual, the *factory()* function are design patterns that help in creating objects.

Question: Please explain the digest cycle in Angular?

Answer: The process of monitoring the watchlist in order to track changes in the value of the watch variable is termed the digest cycle in Angular. The previous and present versions of the scope model values are compared in each digest cycle.

Although the digest cycle process gets triggered implicitly, it is possible to start it manually by using the *$apply()* function.

Question: **What is new in Angular 6?**

Answer: Here are some of the new aspects introduced in Angular 6:

- Angular Elements – It allows converting Angular components into web components and embeds the same in some non-Angular application
- Tree Shakeable Provider – Angular 6 introduces a new way of registering a provider directly inside the @Injectable() decorator. It is achieved by using the providedIn attribute
- RxJS 6 – Angular 6 makes use of RxJS 6 internally
- i18n (internationalization) – Without having to build the application once per locale, any Angular application can have "runtime i18n"

33. What is Transpiling in Angular?

Transpiling in Angular refers to the process of conversion of the source code from one programming language to another. In Angular, generally, this conversion is done from TypeScript to JavaScript. It is an implicit process and happens internally.

Question: **What is ngOnInit ()? How to define it?**

Answer: ngOnInit () is a lifecycle hook that is called after Angular has finished initializing all data-bound properties of a directive. It is defined as:

```
Interface OnInit {
 ngOnInit () : void
 }
```

Question: **What is SPA (Single Page Application) in Angular? Contrast SPA technology with traditional web technology?**

Answer: In the SPA technology, only a single page, which is index.HTML, is maintained although the URL keeps on changing. Unlike traditional web technology, SPA technology is faster and easy to develop as well. In conventional web technology, as soon as a client requests a webpage, the server sends the resource. However, when again the client requests for another page, the server responds again with sending the requested resource. The problem with this technology is that it requires a lot of time.

Question: **What is the code for creating a decorator?**

Answer: We create a decorator called Dummy:

```
function Dummy(target) {
dummy.log('This decorator is Dummy', target);
}
```

Question: **What is the process called by which TypeScript code is converted into JavaScript code?**

Answer: It is called Transpiling. Even though TypeScript is used for writing code in Angular applications, it gets internally transpiled into equivalent JavaScript.

Question: **What is ViewEncapsulation and how many ways are there do to do it in Angular?**

Answer: To put simply, ViewEncapsulation determines whether the styles defined in a particular component will affect the entire application or not. Angular supports 3 types of ViewEncapsulation:

- Emulated – Styles used in other HTML spread to the component
- Native – Styles used in other HTML doesn't spread to the component
- None – Styles defined in a component are visible to all components of the application

Question: **Why prioritize TypeScript over JavaScript in Angular?**

Answer: TypeScript is developed by Microsoft and it is a superset of JavaScript. The issue with JS is that it isn't a true OOP language. As the JS code doesn't follow the Prototype Pattern, the bigger the size of the code the messier it gets. Hence, it leads to difficulties in maintainability as well as reusability. To offset this, TypeScript follows a strict OOP approach.

39. What is a singleton pattern and where we can find it in Angular?

Singleton pattern in Angular is a great pattern which restricts a class from being used more than once. Singleton pattern in Angular is majorly implemented on dependency injection and in the services. Thus, if you use 'new Object()' without making it a singleton, then two different memory locations will be allocated for the same object. Whereas, if the object is declared as a singleton, in case it already exists in the memory then simply it will be reused.

41. What is bootstrapping in Angular?

Bootstrapping in Angular is nothing but initializing, or starting the Angular app. Angular supports automatic and manual bootstrapping.

- ***Automatic Bootstrapping:*** this is done by adding the ng-app directive to the root of the application, typically on the tag or tag if you want angular to bootstrap your application automatically. When Angular finds ng-app directive, it loads the module associated with it and then compiles the DOM.
- ***Manual Bootstrapping:*** Manual bootstrapping provides you more control on how and when to initialize your Angular application. It is useful where you want to perform any other operation before Angular wakes up and compile the page.

42. What is the difference between a link and compile in Angular?

- Compile function is used for template DOM Manipulation and to collect all the directives.

- Link function is used for registering DOM listeners as well as instance DOM manipulation and is executed once the template has been cloned.

What is Redux?

It is a library which helps us maintain the state of the application. Redux is not required in applications that are simple with the simple data flow, it is used in Single Page Applications that have complex data flow.

Explain Authentication and Authorization.

Authentication: The user login credentials are passed to an authenticate API (on the server). On the server side validation of the credentials happens and a JSON Web Token (JWT) is returned. JWT is a JSON object that has some information or attributes about the current user. Once the JWT is given to the client, the client or the user will be identified with that JWT.

Authorization: After logging in successfully, the authenticated or genuine user does not have access to everything. The user is not authorized to access someone else's data, he/she is authorized to access some data.

"Change Detection"?

First let's be on the same page, what means "change detection" anyway?

The basic mechanism of the *change detection* is to perform checks against two states, one is the current state, the other is the new state. If one of this state is different of the other, then something has changed, meaning we need to update (or re-render) the view.

Change Detection means updating the view (DOM) when the data has changed.
Tree Shaking

Angular version 6 introduced a new feature, Tree Shakeable Providers. Tree Shakeable Providers are a way to define services and other things to be used by Angular's dependency injection system in a way that can improve the performance of an Angular application.

First, let's define tree shaking before we dig too deep. Tree shaking is a step in a build process that removes unused code from a code base. Removing unused code can be thought as "tree shaking," or you can visualize the physical shaking of a tree and the remaining dead leaves falling off of the tree. By using tree shaking, we can make sure our application only includes the code that is needed for our application to run.

Angular Tree Shaking Providers

With Tree Shaking Providers (TSP) we can use a different mechanism to register our services. Using this new TSP mechanism will provide the benefits of both tree shaking performance and dependency

injection. We have a demo application with specific code to demonstrate the different performance characteristics of how we register these services. Let's take a look at what the new TSP syntax looks like.

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



@Injectable({

  providedIn: 'root'

})

export class SharedService {

  constructor() { }

}
```

In the `@Injectable` decorator we have a new property called `providedIn`. With this property we can tell Angular which module to register our service to instead of having to import the module and register it to the providers of a `NgModule`. By default, this syntax registers it to the root injector which will make our service an application wide singleton. The root provider is a reasonable default for most use cases. If you still need to control the number of instance of a Service the regular providers API is still available on Angular Modules and Components.