Angular Interview Questions and Answers

1. What is Angular Framework?

Angular is a **TypeScript-based open-source** front-end platform that makes it easy to build applications with in web/mobile/desktop. The major features of this framework such as declarative templates, dependency injection, end to end tooling, and many more other features are used to ease the development.

2. What is the difference between AngularJS and Angular?

Angular is a completely revived component-based framework in which an application is a tree of individual components.

Some of the major difference in tabular form

AngularJS	Angular
It is based on MVC architecture	This is based on Service/Controller
This uses use JavaScript to build the application	Introduced the typescript to write the application
Based on controllers concept	This is a component based UI approach
Not a mobile friendly framework	Developed considering mobile platform
Difficulty in SEO friendly application development	Ease to create SEO friendly applications

3. What is TypeScript?

TypeScript is a typed superset of JavaScript created by Microsoft that adds optional types, classes, async/await, and many other features, and compiles to plain JavaScript. Angular built entirely in TypeScript and used as a primary language. You can install it globally as

```
npm install -g typescript
```

Let's see a simple example of TypeScript usage,

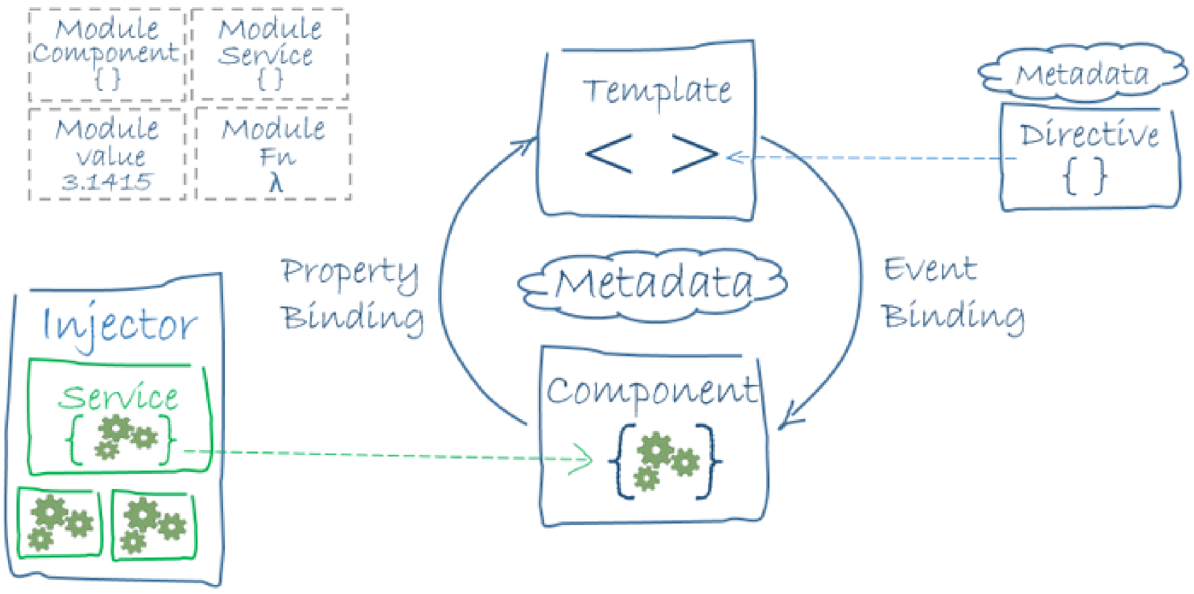
```
function greeter(person: string) {
    return "Hello, " + person;
}

let user = "Sudheer";
```

```
document.body.innerHTML = greeter(user);
```

The greeter method allows only string type as argument.

4. Write a pictorial diagram of Angular architecture?

The main building blocks of an Angular application is shown in the below diagram



5. 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:** An angular module is set of angular basic building blocks like component, directives, services etc. An application is divided into logical pieces and each piece of code is called as "module" which perform a single task.
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.

6. 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!</p>
```

7. What are components?

Components are the most basic UI building block of an Angular app which formed a tree of Angular components. These components are subset of directives. Unlike directives, components always have a template and only one component can be instantiated per an element in a template. Let's see a simple example of Angular component

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

@Component ({
   selector: 'my-app',
   template: ` <div>
      <h1>{{title}}</h1>
      <div>Learn Angular6 with examples</div>
   </div> `,
})

export class AppComponent {
   title: string = 'Welcome to Angular world';
}
```

8. What are the differences between Component and Directive?

In a short note, A component(@component) is a directive-with-a-template.

Some of the major differences are mentioned in a tabular form

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
Component is used to break up the application into smaller components	Directive is use to design re-usable components

Component	Directive
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

9. What is a template?

A template is a HTML view where you can display data by binding controls to properties of an Angular component. 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. **Using inline template with template syntax,**

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

@Component ({
   selector: 'my-app',
   template: '
      <div>
         <h1>{{title}}</h1>
         <div>Learn Angular</div>
      </div>
   '
})

export class AppComponent {
   title: string = 'Hello World';
}
```

Using separate template file such as app.component.html

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

@Component ({
   selector: 'my-app',
   templateUrl: 'app/app.component.html'
})

export class AppComponent {
   title: string = 'Hello World';
}
```

10. What is a module?

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,

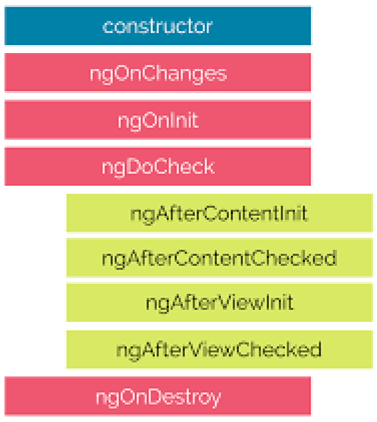
```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

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

The NgModule decorator has three options

i. The imports option is used to import other dependent modules. The BrowserModule is required by default for any web based angular application
ii. The declarations option is used to define components in the respective module
iii. The bootstrap option tells Angular which Component to bootstrap in the application

11. What are lifecycle hooks available?

Angular application goes through an entire set of processes or has a lifecycle right from its initiation to the end of the application. The representation of lifecycle in pictorial



representation as follows,

The description of each lifecycle method is as below,

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.

12. What is a data binding?

Data binding is a core concept in Angular and allows to define communication between a component and the DOM, making it very easy to define interactive applications without worrying about pushing and pulling data. There are four forms of data binding(divided as 3 categories) which differ in the way the data is flowing.

i. **From the Component to the DOM: Interpolation:** {{ value }}: Adds the value of a property from the component

```
<li>Name: {{ user.name }}</li>
<li>Address: {{ user.address }}</li>
```

Property binding: [property]="value": The value is passed from the component to the specified property or simple HTML attribute

```
<input type="email" [value]="user.email">
```

ii. **From the DOM to the Component: Event binding: (event)="function":** When a specific DOM event happens (eg.: click, change, keyup), call the specified method in the component

```
<button (click)="logout()"></button>
```

iii. **Two-way binding: Two-way data binding:** [(ngModel)]="value": Two-way data binding allows to have the data flow both ways. For example, in the below code snippet, both the email DOM input and component email property are in sync

```
<input type="email" [(ngModel)]="user.email">
```

13. 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

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

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

```
export class MyComponent {
  constructor() {
    console.log('Hey I am a component!');
  }
}

@NgModule({
  imports: [],
  declarations: [],
})
export class MyModule {
  constructor() {
    console.log('Hey I am a module!');
  }
}
```

ii. **Property decorators** Used for properties inside classes, e.g. @Input and @Output

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

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

export class MyComponent {
    @Input()
    title: string;
}
```

iii. **Method decorators** Used for methods inside classes, e.g. @HostListener

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

@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

```
import { Component, Inject } from '@angular/core';
import { MyService } from './my-service';

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

14. What is angular CLI?

Angular CLI(**Command Line Interface**) is a command line interface to scaffold and build angular apps using nodejs style (commonJs) modules. You need to install using below npm command,

```
npm install @angular/cli@latest
```

Below are the list of few commands, which will come handy while creating angular projects

i. **Creating New Project:** ng new
ii. **Generating Components, Directives & Services:** ng generate/g The different types of commands would be,
   - ng generate class my-new-class: add a class to your application
   - ng generate component my-new-component: add a component to your application
   - ng generate directive my-new-directive: add a directive to your application
   - ng generate enum my-new-enum: add an enum to your application
   - ng generate module my-new-module: add a module to your application
   - ng generate pipe my-new-pipe: add a pipe to your application
   - ng generate service my-new-service: add a service to your application

iii. **Running the Project:** ng serve

15. 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()
  }
}
```

16. 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');
  }
}
```

The above service uses Http service as a dependency.

17. What is dependency injection in Angular?

Dependency injection (DI), is an important application design pattern in which a class asks for dependencies from external sources rather than creating them itself. Angular comes with its own dependency injection framework for resolving dependencies(services or objects that a class needs to perform its function).So you can have your services depend on other services throughout your application.

18. How is Dependency Hierarchy formed?

19. 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)
  );
}
```

20. 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
```

21. What is the purpose of ngFor directive?

We use Angular ngFor directive in the template to display each item in the list. For example, here we iterate over list of users,

```
<li *ngFor="let user of users">
  {{ user }}
</li>
```

The user variable in the ngFor double-quoted instruction is a **template input variable**

22. What is the purpose of ngIf directive?

Sometimes an app needs to display a view or a portion of a view only under specific circumstances. The Angular ngIf directive inserts or removes an element based on a truthy/falsy condition. Let's take an example to display a message if the user age is more than 18,

```
<p *ngIf="user.age > 18">You are not eligible for student pass!</p>
```

Note: Angular isn't showing and hiding the message. It is adding and removing the paragraph element from the DOM. That improves performance, especially in the larger projects with many data bindings.

23. 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>';
}
```

24. 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>
```

In the example above, Angular evaluates the title and url properties and fills in the blanks, first displaying a bold application title and then a URL.

25. What are template expressions?

A template expression produces a value similar to any Javascript expression. Angular executes the expression and assigns it to a property of a binding target; the target might be an HTML element, a component, or a directive. In the property binding, a template expression appears in quotes to the right of the = symbol as in [property]="expression". In interpolation syntax, the template expression is surrounded by double curly braces. For example, in the below interpolation, the template expression is {{username}},

```
<h3>{{username}}, welcome to Angular</h3>
```

The below javascript expressions are prohibited in template expression

i. assignments (=, +=, -=, ...)
ii. new
iii. chaining expressions with ; or ,
iv. increment and decrement operators (++ and --)

26. What are template statements?

A template statement responds to an event raised by a binding target such as an element, component, or directive. The template statements appear in quotes to the right of the = symbol like **(event)="statement"**. Let's take an example of button click event's statement

```
<button (click)="editProfile()">Edit Profile</button>
```

In the above expression, editProfile is a template statement. The below JavaScript syntax expressions are not allowed.

i. new
ii. increment and decrement operators, ++ and --
iii. operator assignment, such as += and -=
iv. the bitwise operators | and &
v. the template expression operators

---

27. How do you categorize data binding types?

Binding types can be grouped into three categories distinguished by the direction of data flow. They are listed as below,

i. From the source-to-view
ii. From view-to-source
iii. View-to-source-to-view

The possible binding syntax can be tabularized as below,

Data direction	Syntax	Type
From the source-to-view(One-way)	1. {{expression}} 2. [target]="expression" 3. bind-target="expression"	Interpolation, Property, Attribute, Class, Style
From view-to-source(One-way)	1. (target)="statement" 2. on-target="statement"	Event
View-to-source-to-view(Two-way)	1. [(target)]="expression"	Two-way

Data direction	Syntax	Type
	2. bindon-target="expression"	

28. 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
}
```

29. 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.

30. 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);
        }
```

31. 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}}`
```

32. 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>
`
```

33. 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.

34. 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 { }
```

35. What are observables?

Observables are declarative which provide support for passing messages between publishers and subscribers in your application. They are mainly used for event handling, asynchronous programming, and handling multiple values. In this case, you define a function for publishing values, but it is not executed until a consumer subscribes to it. The subscribed consumer then receives notifications until the function completes, or until they unsubscribe.

36. 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

37. Explain on how to use HttpClient with an example?

Below are the steps need to be followed for the usage of HttpClient.

i. Import HttpClient into root module:

```
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  ......
  })
 export class AppModule {}
```

ii. Inject the HttpClient into the application: Let's create a userProfileService(userprofile.service.ts) as an example. It also defines get method of HttpClient

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

const userProfileUrl: string = 'assets/data/profile.json';

@Injectable()
export class UserProfileService {
  constructor(private http: HttpClient) { }

  getUserProfile() {
    return this.http.get(this.userProfileUrl);
  }
}
```

iii. Create a component for subscribing service: Let's create a component called UserProfileComponent(userprofile.component.ts) which inject UserProfileService and invokes the service method,

```
fetchUserProfile() {
  this.userProfileService.getUserProfile()
    .subscribe((data: User) => this.user = {
        id: data['userId'],
        name: data['firstName'],
        city:  data['city']
    });
}
```

Since the above service method returns an Observable which needs to be subscribed in the component.

38. 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.

39. 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.

40. 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';
```

41. 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
source.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
```

42. 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);
});
```

43. 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.

44. What is the difference between promise and observable?

Below are the list of differences 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
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

45. 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
```

46. How do you perform error handling in observables?

You can handle errors by specifying an **error callback** on the observer instead of relying on try/catch which are ineffective in asynchronous environment. For example, you can define error callback as below,

```
myObservable.subscribe({
  next(num) { console.log('Next num: ' + num)},
  error(err) { console.log('Received an errror: ' + err)}
});
```

47. What is the short hand notation for subscribe method?

The subscribe() method can accept callback function definitions in line, for next, error, and complete handlers is known as short hand notation or Subscribe method with positional arguments. For example, you can define subscribe method as below,

```
myObservable.subscribe(
  x => console.log('Observer got a next value: ' + x),
  err => console.error('Observer got an error: ' + err),
  () => console.log('Observer got a complete notification')
);
```

48. What are the utility functions provided by RxJS?

The RxJS library also provides below utility functions for creating and working with observables.

i. Converting existing code for async operations into observables
ii. Iterating through the values in a stream
iii. Mapping values to different types

iv. Filtering streams
v. Composing multiple streams

49. 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}`);
  });
```

50. What will happen if you do not supply handler for observer?

Normally an observer object can define any combination of next, error and complete notification type handlers. If you don't supply a handler for a notification type, the observer just ignores notifications of that type.

51. 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`.

52. What is the browser support of Angular Elements?

Since Angular elements are packaged as custom elements the browser support of angular elements is same as custom elements support. This feature is is currently supported natively in a number of browsers and pending for other browsers.

Browser	Angular Element Support
Chrome	Natively supported
Opera	Natively supported
Safari	Natively supported
Firefox	Natively supported from 63 version onwards. You need to enable dom.webcomponents.enabled and dom.webcomponents.customelements.enabled in older browsers
Edge	Currently it is in progress

53. What are custom elements?

Custom elements (or Web Components) are a Web Platform feature which extends HTML by allowing you to define a tag whose content is created and controlled by JavaScript code. The browser maintains a `CustomElementRegistry` of defined custom elements, which maps an instantiable JavaScript class to an HTML tag. Currently this feature is supported by Chrome, Firefox, Opera, and Safari, and available in other browsers through polyfills.

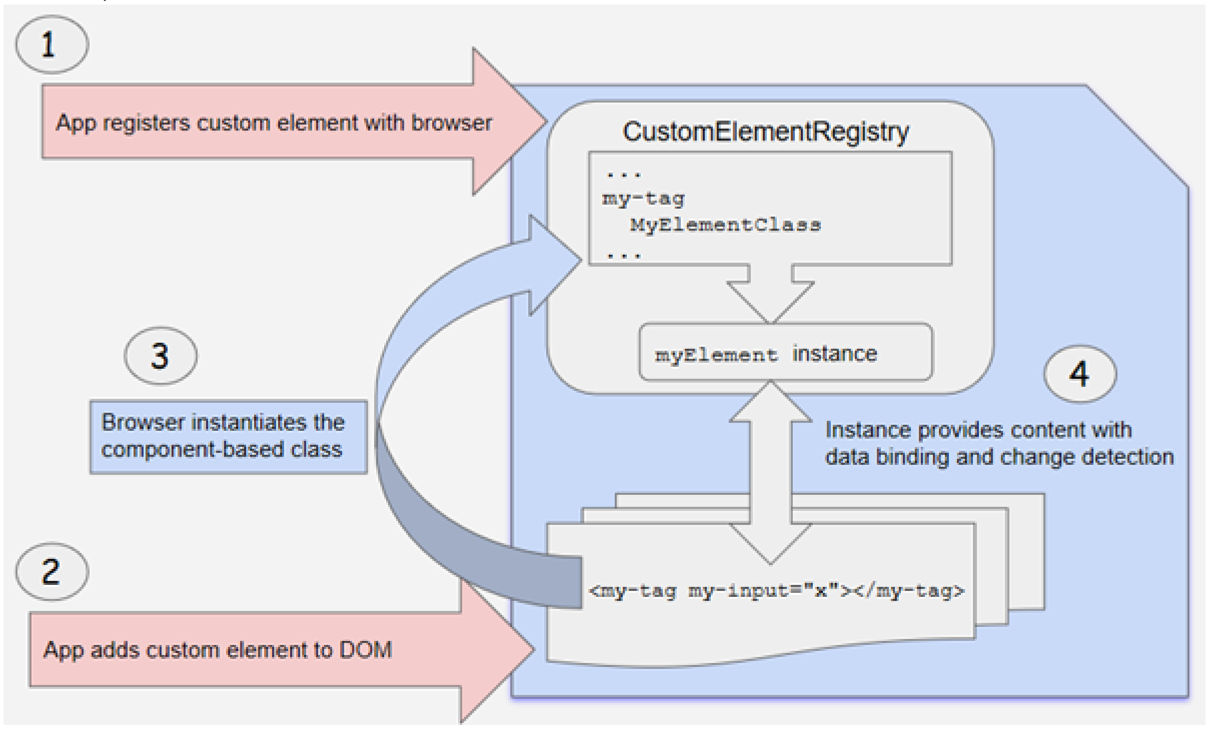
54. Do I need to bootstrap custom elements?

No, custom elements bootstrap (or start) automatically when they are added to the DOM, and are automatically destroyed when removed from the DOM. Once a custom element is added to the DOM for any page, it looks and behaves like any other HTML element, and does not require any special knowledge of Angular.

55. Explain how custom elements works internally?

Below are the steps in an order about custom elements functionality,

i. **App registers custom element with browser:** Use the createCustomElement() function to convert a component into a class that can be registered with the browser as a custom element.
ii. **App adds custom element to DOM:** Add custom element just like a built-in HTML element directly into the DOM.
iii. **Browser instantiate component based class:** Browser creates an instance of the registered class and adds it to the DOM.
iv. **Instance provides content with data binding and change detection:** The content with in template is rendered using the component and DOM data. The flow chart of the custom elements functionality would be as
follows,

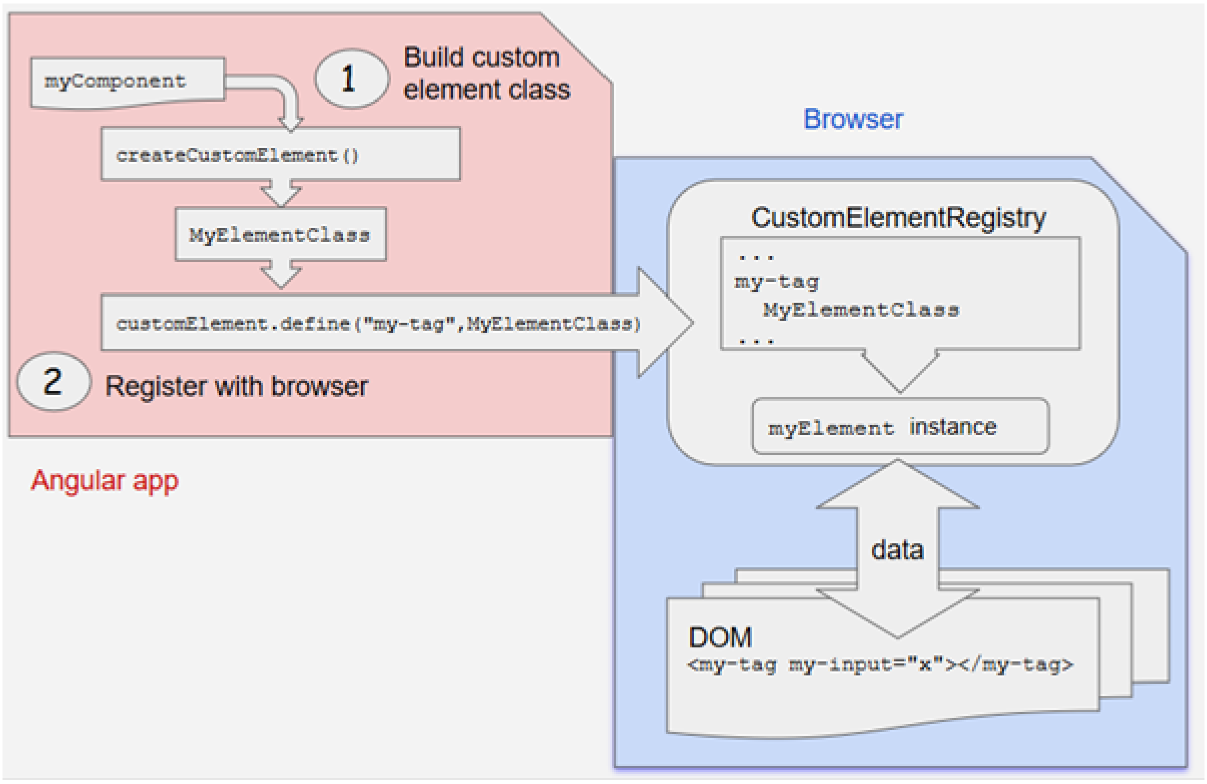

1
App registers custom element with browser
CustomElementRegistry
...
my-tag
MyElementClass
...
myElement instance
3
Browser instantiates the component-based class
4
Instance provides content with data binding and change detection
2
App adds custom element to DOM
<my-tag my-input="x"></my-tag>

56. How to transfer components to custom elements?

Transforming components to custom elements involves **two** major steps,

i. **Build custom element class:** Angular provides the `createCustomElement()` function for converting an Angular component (along with its dependencies) to a custom element. The conversion process implements `NgElementConstructor` interface, and creates a constructor class which is used to produce a self-bootstrapping instance of Angular component.

ii. **Register element class with browser:** It uses `customElements.define()` JS function, to register the configured constructor and its associated custom-element tag with the browser's `CustomElementRegistry`. When the browser encounters the tag for the registered element, it uses the constructor to create a custom-element instance. The detailed structure would be as
follows,



57. What are the mapping rules between Angular component and custom element?

The Component properties and logic maps directly into HTML attributes and the browser's event system. Let us describe them in two steps,

i. The createCustomElement() API parses the component input properties with corresponding attributes for the custom element. For example, component @Input('myInputProp') converted as custom element attribute `my-input-prop`.
ii. The Component outputs are dispatched as HTML Custom Events, with the name of the custom event matching the output name. For example, component @Output() valueChanged = new EventEmitter() converted as custom element with dispatch event as "valueChanged".

58. How do you define typings for custom elements?

You can use the `NgElement` and `WithProperties` types exported from @angular/elements. Let's see how it can be applied by comparing with Angular component, The simple container with input property would be as below,

```
@Component(...)
class MyContainer {
  @Input() message: string;
}
```

After applying types typescript validates input value and their types,

```
const container = document.createElement('my-container') as NgElement &
WithProperties<{message: string}>;
container.message = 'Welcome to Angular elements!';
container.message = true;  // <-- ERROR: TypeScript knows this should be a string.
container.greet = 'News';  // <-- ERROR: TypeScript knows there is no `greet` property
on `container`.
```

59. What are dynamic components?

Dynamic components are the components in which components location in the application is not defined at build time.i.e, They are not used in any angular template. But the component is instantiated and placed in the application at runtime.

60. What are the various kinds of directives?

There are mainly three kinds of directives.

i. **Components** — These are directives with a template.
ii. **Structural directives** — These directives change the DOM layout by adding and removing DOM elements.
iii. **Attribute directives** — These directives change the appearance or behavior of an element, component, or another directive.

61. How do you create directives using CLI?

You can use CLI command `ng generate directive` to create the directive class file. It creates the source file(src/app/components/directivename.directive.ts), the respective test file(.spec.ts) and declare the directive class file in root module.

62. Give an example for attribute directives?

Let's take simple highlighter behavior as a example directive for DOM element. You can create and apply the attribute directive using below steps,

i. Create HighlightDirective class with the file name `src/app/highlight.directive.ts`. In this file, we need to import **Directive** from core library to apply the metadata and **ElementRef** in the directive's constructor to inject a reference to the host DOM element ,

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

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

ii. Apply the attribute directive as an attribute to the host element(for example,

)

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

iii. Run the application to see the highlight behavior on paragraph element

```
ng serve
```

63. 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.

64. What is the purpose of base href tag?

The routing application should add element to the index.html as the first child in the tag inorder to indicate how to compose navigation URLs. If app folder is the application root then you can set the href value as below

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

65. What are the router imports?

The Angular Router which represents a particular component view for a given URL is not part of Angular Core. It is available in library named `@angular/router` to import required router components. For example, we import them in app module as below,

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

66. What is router outlet?

The RouterOutlet is a directive from the router library and it acts as a placeholder that marks the spot in the template where the router should display the components for that outlet. Router outlet is used like a component,

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

67. 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>
```

68. 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>
```

69. What is router state?

RouterState is a tree of activated routes. Every node in this tree knows about the "consumed" URL segments, the extracted parameters, and the resolved data. You can access the current RouterState from anywhere in the application using the `Router service` and the `routerState` property.

```
@Component({templateUrl:'template.html'})
class MyComponent {
  constructor(router: Router) {
    const state: RouterState = router.routerState;
    const root: ActivatedRoute = state.root;
    const child = root.firstChild;
    const id: Observable<string> = child.params.map(p => p.id);
    //...
  }
}
```

70. 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

71. What is activated route?

ActivatedRoute contains the information about a route associated with a component loaded in an outlet. It can also be used to traverse the router state tree. The ActivatedRoute will be injected as a router service to access the information. In the below example, you can access route path and parameters,

```
@Component({...})
class MyComponent {
  constructor(route: ActivatedRoute) {
    const id: Observable<string> = route.params.pipe(map(p => p.id));
    const url: Observable<string> = route.url.pipe(map(segments => segments.join('')));
    // route.data includes both `data` and `resolve`
    const user = route.data.pipe(map(d => d.user));
  }
}
```

72. How do you define routes?

A router must be configured with a list of route definitions. You configures the router with routes via the `RouterModule.forRoot()` method, and adds the result to the AppModule's `imports` array.

```
const appRoutes: Routes = [
 { path: 'todo/:id',      component: TodoDetailComponent },
 {
   path: 'todos',
   component: TodosListComponent,
   data: { title: 'Todos List' }
 },
 { path: '',
   redirectTo: '/todos',
   pathMatch: 'full'
 },
 { path: '**', component: PageNotFoundComponent }
];

@NgModule({
 imports: [
   RouterModule.forRoot(
     appRoutes,
     { enableTracing: true } // <-- debugging purposes only
   )
   // other imports here
 ],
 ...
})
export class AppModule { }
```

73. 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

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

74. Do I need a Routing Module always?

No, the Routing Module is a design choice. You can skip routing Module (for example, AppRoutingModule) when the configuration is simple and merge the routing configuration directly into the companion module (for example, AppModule). But it is recommended when the configuration is complex and includes specialized guard and resolver services.

75. 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.

76. What are different types of compilation in Angular?

Angular offers two ways to compile your application,

i. Just-in-Time (JIT)
ii. Ahead-of-Time (AOT)

77. 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
```

78. 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.

79. 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.

80. 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.

81. 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

82. 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.

83. 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.

84. 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.

85. 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.

86. 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.

87. What is folding?

The compiler can only resolve references to exported symbols in the metadata. Where as some of the non-exported members are folded while generating the code. i.e Folding is a process in which the collector evaluate an expression during collection and record the result in the .metadata.json instead of the original expression. For example, the compiler couldn't refer selector reference because it is not exported

```
let selector = 'app-root';
@Component({
  selector: selector
})
```

Will be folded into inline selector

```
@Component({
      selector: 'app-root'
    })
```

Remember that the compiler can’t fold everything. For example, spread operator on arrays, objects created using new keywords and function calls.

88. 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);
```

89. 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".

90. What are the differences of various versions of Angular?

There are different versions of Angular framework. Let's see the features of all the various versions,

i. Angular 1 • Angular 1 (AngularJS) is the first angular framework released in the year 2010. • AngularJS is not built for mobile devices. • It is based on controllers with MVC architecture.
ii. Angular 2 • Angular 2 was released in the year 2016. Angular 2 is a complete rewrite of Angular1 version. • The performance issues that Angular 1 version had has been addressed in Angular 2 version. • Angular 2 is built from scratch for mobile devices unlike Angular 1 version. • Angular 2 is components based.
iii. Angular 3 The following are the different package versions in Angular 2. • @angular/core v2.3.0 • @angular/compiler v2.3.0 • @angular/http v2.3.0 • @angular/router v3.3.0 The router package is already versioned 3 so to avoid confusion switched to Angular 4 version and skipped 3 version.
iv. Angular 4 • The compiler generated code file size in AOT mode is very much reduced. • With Angular 4 the production bundles size is reduced by hundreds of KB's. • Animation features are removed from angular/core and formed as a separate package. • Supports Typescript 2.1 and 2.2.
v. Angular 5 • Angular 5 makes angular faster. It improved the loading time and execution time. • Shipped with new build optimizer. • Supports Typescript 2.5.
vi. Angular 6 • It is released in May 2018. • Includes Angular Command Line Interface (CLI), Component Development KIT (CDK), Angular Material Package.
vii. Angular 7 • It is released in October 2018. • TypeScript 3.1 • RxJS 6.3 • New Angular CLI • CLI Prompts capability provide an ability to ask questions to the user before they run. It is like interactive dialog between the user and the CLI • With the improved CLI Prompts capability, it helps developers to make the decision. New ng commands ask users for routing and CSS styles types(SCSS) and ng add @angular/material asks for themes and gestures or animations.
viii. Angular 8 • It is released in October 2018 • Angular 8: Angular 8 supports TypeScript 3.4• •Angular 8 supports Web Workers • The new compiler for Angular 8 is Ivy Rendering Engine

•Angular 8 provides dynamic imports for lazy-loaded modules • Improvement of ngUpgrade

91.How do you find angular CLI version?

Angular CLI provides it's installed version using below different ways using ng command

```
ng v
ng version
ng -v
ng --version
```

and the output would be as below,

```
Angular CLI: 1.6.3
Node: 8.11.3
OS: darwin x64
Angular:
...
```

92.What is the browser support for Angular?

Angular supports most recent browsers which includes both desktop and mobile browsers.

Browser	Version
Chrome	latest
Firefox	latest
Edge	2 most recent major versions
IE	11, 10, 9 (Compatibility mode is not supported)
Safari	2 most recent major versions
IE Mobile	11
iOS	2 most recent major versions
Android	7.0, 6.0, 5.0, 5.1, 4.4

93.What is router state?

The RouteState is an interface which represents the state of the router as a tree of activated routes.

```
interface RouterState extends Tree {
  snapshot: RouterStateSnapshot
  toString(): string
}
```

You can access the current RouterState from anywhere in the Angular app using the Router service and the routerState property.

94.What is the purpose of ngSwitch directive?

NgSwitch directive is similar to JavaScript switch statement which displays one element from among several possible elements, based on a switch condition. In this case only the selected element placed into the DOM. It has been used along with `NgSwitch`, `NgSwitchCase` and `NgSwitchDefault` directives. For example, let's display the browser details based on selected browser using ngSwitch directive.

```
<div [ngSwitch]="currentBrowser.name">
  <chrome-browser    *ngSwitchCase="'chrome'"    [item]="currentBrowser"></chrome-browser>
  <firefox-browser   *ngSwitchCase="'firefox'"     [item]="currentBrowser"></firefox-browser>
  <opera-browser     *ngSwitchCase="'opera'"  [item]="currentBrowser"></opera-browser>
  <safari-browser     *ngSwitchCase="'safari'"   [item]="currentBrowser"></safari-browser>
  <ie-browser  *ngSwitchDefault           [item]="currentItem"></ie-browser>
</div>
```

95.What is a bootstrapped component?

A bootstrapped component is an entry component that Angular loads into the DOM during the bootstrap process or application launch time. Generally, this bootstrapped or root component is named as `AppComponent` in your root module using `bootstrap` property as below.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent] // bootstrapped entry component need to be declared here
})
```

96.Is it necessary for bootstrapped component to be entry component?

Yes, the bootstrapped component needs to be an entry component. This is because the bootstrapping process is an imperative process.

97.What is a routed entry component?

The components referenced in router configuration are called as routed entry components. This routed entry component defined in a route definition as below,

```
const routes: Routes = [
  {
    path: '',
    component: TodoListComponent // router entry component
  }
];
```

Since router definition requires you to add the component in two places (router and entryComponents), these components are always entry components. **Note:** The compilers are smart enough to recognize a router definition and automatically add the router component into `entryComponents`.

98. Why is not necessary to use entryComponents array every time?

Most of the time, you don't need to explicity to set entry components in entryComponents array of ngModule decorator. Because angular adds components from both @NgModule.bootstrap and route definitions to entry components automatically.

99. What is Angular compiler?

The Angular compiler is used to convert the application code into JavaScript code. It reads the template markup, combines it with the corresponding component class code, and emits component factories which creates JavaScript representation of the component along with elements of @Component metadata.

100. What is the role of ngModule metadata in compilation process?

The `@NgModule` metadata is used to tell the Angular compiler what components to be compiled for this module and how to link this module with other modules.

101. How does angular finds components, directives and pipes?

The Angular compiler finds a component or directive in a template when it can match the selector of that component or directive in that template. Whereas it finds a pipe if the pipe's name appears within the pipe syntax of the template HTML.

102. Give few examples for NgModules?

The Angular core libraries and third-party libraries are available as NgModules.

103. Angular libraries such as FormsModule, HttpClientModule, and RouterModule are NgModules.
104. Many third-party libraries such as Material Design, Ionic, and AngularFire2 are NgModules.

103. What are feature modules?

Feature modules are NgModules, which are used for the purpose of organizing code. The feature module can be created with Angular CLI using the below command in the root directory,

```
ng generate module MyCustomFeature //
```

Angular CLI creates a folder called `my-custom-feature` with a file inside called `my-custom-feature.module.ts` with the following contents

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

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class MyCustomFeature { }
```

104. What is a provider?

A provider is an instruction to the Dependency Injection system on how to obtain a value for a dependency(aka services created). The service can be provided using Angular CLI as below,

```
ng generate service my-service
```

The created service by CLI would be as below,

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

@Injectable({
  providedIn: 'root', //Angular provide the service in root injector
})
export class MyService {
}
```

105. What is the recommendation for provider scope?

You should always provide your service in the root injector unless there is a case where you want the service to be available only if you import a particular @NgModule.

106. How do you restrict provider scope to a module?

It is possible to restrict service provider scope to a specific module instead making available to entire application. There are two possible ways to do it.

a. **Using providedIn in service:**

```
import { Injectable } from '@angular/core';
import { SomeModule } from './some.module';

@Injectable({
  providedIn: SomeModule,
})
export class SomeService {
}
```

ii. **Declare provider for the service in module:**

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

import { SomeService } from './some.service';

@NgModule({
  providers: [SomeService],
})
export class SomeModule {
}
```