TYPESCRIPT INTERVIEW QUESTIONS - COMPLETE GUIDE (ALL 79 QUESTIONS)

Table of Contents

TYPESCRIPT BASICS ANSWERS

1. What is TypeScript?

TypeScript is a **strongly typed, object-oriented** programming language developed and maintained by **Microsoft** (created by Anders Hejlsberg in 2012). It is a **superset of JavaScript** - any valid JS is valid TS. Adds **static typing**, **interfaces**, **generics**, and other features. TypeScript code **compiles (transpiles) to plain JavaScript**. Provides **compile-time type checking** to catch errors early. Improves **code quality**, **maintainability**, and **developer productivity**.

2. Why do we need TypeScript when JavaScript already exists?

- **Type Safety**: Catches type-related errors at compile time, not runtime
- **Better IDE Support**: IntelliSense, auto-completion, refactoring tools
- **Code Maintainability**: Self-documenting code through type annotations
- **Early Error Detection**: Finds bugs before code runs in production
- **Large-Scale Applications**: Better suited for enterprise applications
- **Team Collaboration**: Clearer contracts between different parts of code
- **Modern Features**: Access to latest ECMAScript features with backward compatibility

3. Is TypeScript compiled or interpreted?

TypeScript is **compiled** (more precisely, **transpiled**). TypeScript compiler (**tsc**) converts TS code to JavaScript. The resulting JavaScript is then interpreted by browser/Node.js. TypeScript itself **never runs directly** - always converted to JS first. Compilation catches type errors before code execution. Can target different JavaScript versions (ES5, ES6, ESNext, etc.).

4. What is the difference between JavaScript and TypeScript?

Feature	JavaScript	TypeScript
Typing	Dynamic (runtime)	Static (compile-time)
Type Annotations	Not supported	Supported
Compilation	Interpreted directly	Needs compilation to JS
Interfaces	Not available	Fully supported
Generics	Not available	Fully supported
Error Detection	Runtime errors	Compile-time errors
IDE Support	Limited	Excellent (IntelliSense)
Learning Curve	Easier	Slightly steeper
File Extension	`.js`	`.ts` / `.tsx`

5. What are the benefits of using TypeScript?

- **Type Safety**: Prevents type-related bugs at compile time
- **Better Tooling**: Superior autocomplete, navigation, refactoring
- **Self-Documenting Code**: Types serve as inline documentation
- **Easier Refactoring**: Safe code changes with type checking
- **Better Collaboration**: Clear contracts between team members

6. What is type safety?

Type safety means the compiler prevents type-related errors. Ensures operations are performed only on compatible types. Variables can only hold values of their declared type. Function parameters must match expected types. Prevents common bugs like calling methods on `undefined`. Provides **guarantees** about data types at compile time.

7. What is static typing?

Static typing means types are checked at **compile time** (before running). Types are declared explicitly or inferred by the compiler. Once a variable has a type, it cannot change. Opposite of **dynamic typing** where types are checked at runtime. Errors are caught during development, not in production. Enables better IDE support and documentation.

8. What are primitive types in TypeScript?

TypeScript has the following **primitive types**:

- **number**: All numbers (integers, floats, hex, binary, octal)
- **string**: Text values (single, double quotes, template literals)
- **boolean**: `true` or `false`
- **null**: Intentional absence of value
- **undefined**: Uninitialized variable
- **symbol**: Unique identifier (ES6)
- **bigint**: Large integers (ES2020)

9. What is the `any` type?

any is a special type that **disables type checking**. Variables of type `any` can hold **any value** of any type. Essentially **opts out of TypeScript's type system**. Useful for migrating JavaScript code or working with dynamic data. **Should be avoided** when possible - defeats purpose of TypeScript. Can be source of **runtime errors** that TypeScript normally prevents.

10. What is `unknown` type?

unknown is the **type-safe counterpart** to `any`. Can hold any value, but requires **type checking before use**. Must narrow the type before performing operations. Introduced in TypeScript 3.0 as a safer alternative to `any`. Forces developers to handle type verification explicitly. Ideal for values from external sources (API responses, user input).

VARIABLES AND TYPES ANSWERS

11. How do you declare variables in TypeScript?

Use **`let`**, **`const`**, or **`var`** keywords (same as JavaScript). Add **type annotation** after variable name with colon (`:`). TypeScript can also **infer types** automatically. **const** for values that won't change, **let** for variables. **Avoid var** due to function scoping issues.

12. Difference between `any` and `unknown`

Feature	`any`	`unknown`
Type Safety	No type checking	Type-safe
Assignment	Can assign any value	Can assign any value
Usage	Can use directly without checking	Must narrow type before use
Method Calls	Allowed without checks	Not allowed without narrowing
Property Access	Allowed without checks	Not allowed without narrowing
Recommended	Avoid when possible	Preferred over `any`

13. What is type inference?

Type inference is TypeScript's ability to **automatically determine types**. Compiler analyzes the value to infer the type. No need to explicitly annotate types in many cases. Makes code **cleaner** while maintaining type safety. Works with variables, function return types, and more. Best practice: Let TypeScript infer when type is obvious.

14. What is union type?

Union type allows a variable to hold **one of several types**. Defined using the **pipe (`|`)** operator between types. Variable can be **any one of the specified types** at a time. Must handle all possible types when using the value. Useful for parameters that accept multiple types.

15. What is intersection type?

Intersection type combines **multiple types into one**. Defined using the **ampersand (`&`)** operator. Resulting type has **all properties** from all combined types. Object must satisfy **all** intersected types. Commonly used to **merge interfaces** or types.

16. What is literal type?

Literal types are types that represent **exact specific values**. Variable can only hold that **exact value**, nothing else. Works with **strings**, **numbers**, and **booleans**. Often combined with **union types** for allowed values. Provides **stronger type safety** than general types.

17. What is `enum`?

Enum (enumeration) defines a set of **named constants**. Provides **readable names** for numeric or string values. **Numeric enums**: Auto-increment from 0 by default. **String enums**: Each member must be explicitly initialized. Enums exist at **runtime** as JavaScript objects. Useful for representing

fixed sets of related values.

18. Difference between `enum` and union types

Feature	Enum	Union Types
Runtime	Exists as JS object	Types only, erased at compile
Performance	Slight overhead	Zero overhead
Reverse Mapping	Available (numeric)	Not available
Iteration	Can iterate values	Cannot iterate
Bundle Size	Adds code	No extra code
Flexibility	Fixed structure	More flexible

FUNCTIONS ANSWERS

19. How to define a function in TypeScript?

Functions in TypeScript can have **typed parameters** and **return types**. Use colon (`:`) after parameter for type annotation. Use colon after parentheses for return type. TypeScript can **infer return type** automatically. Supports **arrow functions**, **function expressions**, and **declarations**.

20. Optional parameters vs default parameters

- **Optional parameters**: May or may not be provided (use `?`)
- **Default parameters**: Have a fallback value if not provided
- Optional parameters must come **after required parameters**
- Default parameters can be anywhere (but usually at end)
- Optional parameter type includes `undefined` automatically

21. What are rest parameters?

Rest parameters collect multiple arguments into an **array**. Defined using **spread operator (`...`)** before parameter name. Must be the **last parameter** in function signature. Allows functions to accept **variable number of arguments**. Type annotation applies to the array type.

22. Function return types

Return type specifies what type a function returns. Declared after parameter list with colon (`:`)). **void**: Function doesn't return a value. **never**: Function never completes (throws or infinite loop). TypeScript can **infer** return types automatically. Explicit return types improve code documentation.

23. What is function overloading?

Function overloading allows multiple function signatures for same function name. Different **parameter types** or **counts** → different behaviors. TypeScript uses **overload signatures** followed by **implementation signature**. Implementation must handle **all** overload cases. Useful when return type depends on parameter types.

24. How function overloading works in TypeScript?

- **Step 1**: Define overload signatures (just declarations, no body)
- **Step 2**: Define implementation signature (with body)
- **Step 3**: TypeScript matches call to **first matching overload**
- Implementation must be **compatible with all overloads**
- Order matters: put more specific overloads **first**
- Implementation signature is **not callable directly**

INTERFACES AND TYPES ANSWERS

25. What is an interface?

Interface defines the **structure/shape** of an object. Specifies **property names** and their **types**. Acts as a **contract** that objects must follow. Used for **type checking** at compile time. Interfaces are **erased** at runtime (no JavaScript output). Supports **optional properties**, **readonly**, and **methods**.

26. Difference between interface and type

Feature	Interface	Type
Declaration	`interface Name {}`	`type Name = {}`
Extension	`extends` keyword	`&` intersection
Merging	□ Declaration merging	□ Cannot merge
Implements	□ Classes can implement	□ Classes can implement

Feature	Interface	Type
Union/Intersection	□ Cannot create unions	□ Can create unions
Primitives	□ Only objects	□ Any type
Computed Props	□ Not supported	□ Supported

27. Can interfaces be extended?

Yes, interfaces can be **extended** using the `extends` keyword. Child interface **inherits** all properties from parent. Can extend **multiple interfaces** (comma-separated). Can **add new properties** or **override** with compatible types. Creates an **inheritance hierarchy** for object shapes.

28. Can types be extended?

Yes, types can be extended using **intersection (`&`)**. Creates a **new type** combining all properties. Cannot use `extends` keyword like interfaces. Intersection merges **all properties** from all types. Conflicting properties create intersection (may become `never`).

29. When to use interface vs type?

Use Interface when:

- Defining **object shapes** for classes to implement
- Need **declaration merging** (extending third-party types)
- Building **public APIs** or libraries
- Prefer **OOP-style** syntax with `extends`

Use Type when:

- Need **union types** or **intersection types**
- Working with **primitives**, **tuples**, or **functions**
- Need **computed/mapped types**
- Want more **flexibility**

30. What is `readonly` property?

readonly modifier makes a property **immutable after initialization**. Can only be assigned during **object creation** or **constructor**. Compile-time check only (no runtime enforcement). Useful for **configuration objects**, **IDs**, and **constants**. Can be used in interfaces, types, and classes.

31. Optional properties in interfaces?

Optional properties may or may not be present in object. Marked with **question mark (`?`)** after property name. Type automatically includes `undefined`. Useful for **partial data** or **configuration options**. Must handle `undefined` case when accessing.

CLASSES AND OOP ANSWERS

32. How do classes work in TypeScript?

TypeScript classes are **syntactic sugar** over JavaScript prototypes. Support **type annotations** for properties and methods. Include **access modifiers** (public, private, protected). Support **interfaces implementation** with `implements`. Can have **abstract classes** and **inheritance**. Compile to ES5/ES6 JavaScript classes.

33. Access modifiers: `public`, `private`, `protected`

- **public**: Accessible from **anywhere** (default)
- **private**: Accessible only **within the class** itself
- **protected**: Accessible within class and **subclasses**
- TypeScript-only enforcement (compile-time check)
- ES2022 `#` syntax for true private (runtime enforcement)

34. What is abstract class?

Abstract class is a class that **cannot be instantiated** directly. Serves as a **base class** for other classes. Can contain **abstract methods** (no implementation). Can also contain **concrete methods** (with implementation). Subclasses **must implement** all abstract methods. Used to define **common behavior** and **enforce contracts**.

35. Difference between abstract class and interface

Feature	Abstract Class	Interface
Instantiation	Cannot instantiate	Cannot instantiate
Implementation	Can have method bodies	No implementation (TS)
Properties	Can have values	Only declarations
Constructor	Can have constructor	No constructor

Feature	Abstract Class	Interface
Access Modifiers	Supported	Not supported
Multiple	Single inheritance	Multiple implementation
Runtime	Exists in JavaScript	Erased (type-only)

36. What is constructor parameter properties?

Shorthand syntax to declare and initialize class properties. Add **access modifier** or **readonly** in constructor parameters. Automatically creates and assigns the property. Reduces **boilerplate code** significantly. Works with `public`, `private`, `protected`, and `readonly`.

37. What is inheritance in TypeScript?

Inheritance allows a class to inherit properties and methods from another. Uses the **`extends`** keyword. Child class can **override** parent methods. **`super`** keyword accesses parent class constructor/methods. TypeScript supports **single inheritance** (one parent class). Can combine with **interface implementation** for multiple inheritance-like behavior.

GENERICS ANSWERS

38. What are generics?

Generics allow creating **reusable components** that work with multiple types. Use **type parameters** (placeholders) that are specified when used. Provide **type safety** while maintaining **flexibility**. Convention: Use single capital letters (`T`, `U`, `K`, `V`). Similar to generics in Java, C#, and other typed languages. Enable writing **type-safe** yet **flexible** code.

39. Why are generics used?

- **Type Safety**: Catch type errors at compile time
- **Code Reusability**: Single implementation works with multiple types
- **Avoid Duplication**: No need to write same logic for different types
- **Better IntelliSense**: IDE knows exact types being used
- **Flexibility**: Work with any type while preserving type information
- **Avoid `any`**: Type-safe alternative to using `any`

40. Generic functions

Functions with **type parameters** that work with multiple types. Type parameter declared in **angle brackets** before parameters. Can have **multiple type parameters**. Type can be **explicitly specified** or **inferred**. Can apply **constraints** to limit allowed types.

41. Generic interfaces

Interfaces with **type parameters** for flexible type definitions. Enable creating **reusable contracts** for different types. Type parameter specified when **implementing or using** interface. Common in **API responses**, **repositories**, and **collections**.

42. Generic constraints

Constraints limit the types that can be used with generics. Use **`extends`** keyword to specify constraint. Ensures type parameter has **certain properties or methods**. Can constrain to **interfaces**, **types**, or **classes**. Multiple constraints with **intersection (`&`)**

43. Real-time use cases of generics

Generics are used extensively in real-world TypeScript applications:

- **API Response Handling**: Type-safe response structures
- **State Management**: Redux/NgRx typed actions and state
- **Form Handling**: Type-safe form values
- **Data Repositories**: CRUD operations for any entity
- **Utility Functions**: Array operations, transformations
- **React/Angular Components**: Reusable typed components

ADVANCED TYPES ANSWERS

44. What is tuple?

Tuple is a **fixed-length array** with known types at each position. Each element can have a **different type**. Unlike arrays, tuples have **strict length** and **position-based types**. Useful for **returning multiple values** from functions. Supports **optional elements** and **rest elements**.

45. What is mapped type?

Mapped types create new types by **transforming properties** of existing types. Use `in keyof` syntax to iterate over properties. Can **modify**, **add**, or **remove** property modifiers. Foundation for utility types like `Partial`, `Required`, `Readonly`. Powerful for creating **derived types** from existing ones.

46. What is conditional type?

Conditional types select type based on a **condition**. Syntax: `T extends U ? X : Y`. If `T` is assignable to `U`, result is `X`, otherwise `Y`. Can be **nested** for complex type logic. Enable **type-level programming** and inference.

47. What is `keyof` operator?

keyof creates a **union type** of all property keys of a type. Returns **string literal union** of property names. Useful for creating **type-safe property access**. Often combined with **generics** and **indexed access types**. Essential for building **type-safe utilities**.

48. What is `typeof` operator in TypeScript?

typeof in TypeScript extracts the **type of a value/variable**. Different from JavaScript's runtime `typeof`. Used in **type context** to get compile-time type. Useful for getting types from **values**, **functions**, **objects**. Commonly used with **keyof** for type-safe operations.

49. What are utility types?

Utility types are **built-in generic types** provided by TypeScript. Used to **transform** existing types into new types. Save time by avoiding manual type creation. Include types for making properties optional, required, readonly, etc. Defined in TypeScript's **lib.es5.d.ts**.

50. Common utility types (`Partial`, `Required`, `Pick`, `Omit`, `Record`)

These are the **most commonly used** utility types in TypeScript:

- **Partial**: All properties become optional
- **Required**: All properties become required
- **Pick<T, K>**: Select specific properties
- **Omit<T, K>**: Remove specific properties
- **Record<K, T>**: Create object with specific key-value types

---

TYPE NARROWING ANSWERS

51. What is type narrowing?

Type narrowing is the process of **refining a type** to a more specific type. TypeScript analyzes code flow to **automatically narrow types**. Occurs within **conditional blocks** after type checks. Enables safe access to **type-specific properties and methods**. Uses various techniques: `typeof`, `instanceof`, `in`, custom guards.

52. `typeof` guards

typeof guards use JavaScript's `typeof` operator for type narrowing. Works with **primitive types**: string, number, boolean, symbol, bigint, undefined, function. **Does not work** with `null` (returns "object") or custom types. Most common and simplest form of type narrowing. TypeScript understands the check and narrows accordingly.

53. `instanceof` guards

instanceof guards check if an object is an instance of a **class**. Works with **classes** and **constructor functions**. Cannot be used with **interfaces** (they don't exist at runtime). Checks the **prototype chain** of the object. Useful for narrowing **class hierarchies** and **error types**.

54. Custom type guards

Custom type guards are functions that return a **type predicate**. Syntax: `parameter is Type` as return type. Allows creating **custom narrowing logic**. Useful for **interfaces**, **complex types**, and **validation**. TypeScript trusts the guard and narrows types accordingly.

TYPESCRIPT CONFIGURATION ANSWERS

55. What is `tsconfig.json`?

- **tsconfig.json** is the **configuration file** for TypeScript compiler
- Defines **compiler options**, **file inclusion/exclusion**, and project settings
- Located at **project root** directory
- Automatically detected by **tsc** and IDE (VS Code)
- Can **extend** other configuration files

---

56. Important compiler options

Option	Purpose
target	ECMAScript target version (ES5, ES2020)
module	Module code generation (commonjs, ES2020)
outDir	Output directory for compiled files
rootDir	Root directory of source files
strict	Enable all strict type-checking
declaration	Generate .d.ts files
noUnusedLocals	Error on unused variables
noImplicitAny	Error on implicit any

---

57. `strict` mode in TypeScript

- **strict** is a **meta flag** that enables all strict type-checking options
- Setting `"strict": true` enables **7+ strict checks**
- Recommended for **new projects**
- Enables: `noImplicitAny`, `strictNullChecks`, `strictFunctionTypes`, `strictPropertyInitialization`, etc.

---

58. What is `noImplicitAny`?

- **noImplicitAny** prevents variables from having **implicit `any` type**
- Must **explicitly type** parameters and variables
- Part of **strict** mode
- Forces **explicit type declarations**

---

59. What is `strictNullChecks`?

- Makes `null` and `undefined` **distinct types**
- Variables cannot be null/undefined unless explicitly allowed
- Must handle null/undefined before using values
- Part of **strict** mode

---

60. Difference between `target` and `module`

- **target**: The **ECMAScript version** for output JavaScript
- **module**: The **module system** for generated code
- They are **independent** but related settings
- **target** affects syntax; **module** affects imports/exports

---

TYPESCRIPT WITH JAVASCRIPT ANSWERS

61. Can TypeScript run in browser?

- **No**, TypeScript **cannot run directly** in browsers
- Browsers only understand **JavaScript**
- TypeScript must be **compiled to JavaScript** first

---

62. How is TypeScript code converted to JavaScript?

- TypeScript uses **tsc compiler** to compile code
- Process: **Parse → Type Check → Emit JavaScript**
- **Types are completely erased**
- Output JavaScript version controlled by **target**

---

63. What is declaration file (`.d.ts`)?

- **Declaration files** contain **type information only**
- No runtime code, just **type declarations**
- Allow TypeScript to understand **JavaScript libraries**
- Can be auto-generated with `declaration: true`

---

64. Why are declaration files used?

- **Type safety** for JavaScript libraries
- **IDE support** and IntelliSense
- **Compile-time checking** for third-party code
- **Documentation** of library APIs

---

65. How to use third-party JavaScript libraries in TypeScript?

Methods to use JavaScript libraries:

1. **Install @types packages** - `npm install --save-dev @types/lodash`
2. **Use bundled types** - Check package.json "types"
3. **Create custom declarations** - Create `.d.ts`
4. **Use `any` type** (not recommended)

FREQUENTLY ASKED QUESTIONS ANSWERS

66. Difference between `any` and `unknown` (real-time scenario)

unknown is safer because it **requires type checking**:

Operation	`any`	`unknown`
Call methods	□ Allowed (risky)	□ Must narrow
Access properties	□ Allowed (risky)	□ Must narrow

---

67. Why is `unknown` safer than `any`?

- **unknown** enforces **required type checking**
- Cannot call methods or access without narrowing
- Prevents **runtime errors**
- Encourages **defensive programming**

---

68. Difference between interface and type with example

Feature	Interface	Type
Merging	□ Can merge	□ Cannot
Union Types	□ No	□ Yes
Primitives	□ Objects only	□ Any

---

69. How does TypeScript help in large applications?

Benefits:

1. **Self-Documenting Code**
2. **Safe Refactoring**
3. **Contract Enforcement**
4. **Module Boundaries**
5. **Error Prevention**
6. **Team Collaboration**

---

70. What happens if TypeScript types are wrong?

- Types are **compile-time only** - erased at runtime
- Can lead to **runtime errors** if not validated
- Must use **runtime validation** for external data

---

71. Does TypeScript affect runtime performance?

- **No**, TypeScript has **zero runtime overhead**
- ALL types **completely removed** at compilation
- No type checking at runtime
- Only **build time** is affected

---

72. Can we use TypeScript without types?

- **Yes**, through **type inference**

- TypeScript **infers types** from values
- Can gradually add types
- `noImplicitAny: false` allows implicit any

---

73. How does TypeScript improve maintainability?

Benefits:

1. **Self-Documenting Code**
2. **Safe Refactoring**
3. **Better Navigation**
4. **Error Prevention**
5. **Code Reviews Easier**

VERY IMPORTANT INTERVIEW QUESTIONS ANSWERS

74. Why did you choose TypeScript in your project?

Common reasons:

1. **Type Safety** - Catch bugs at compile time
2. **Team Collaboration** - Types as documentation
3. **Developer Experience** - Better tooling
4. **Framework Support** - Angular, NestJS
5. **Maintainability** - Long-term code quality

---

75. How does TypeScript improve code quality?

1. **Catches Bugs Early** - Type checking
2. **Consistent Patterns** - Standard formats
3. **Invalid States Prevention** - Discriminated unions
4. **Better Documentation** - JSDoc + types
5. **Safe Refactoring** - Find all references

---

76. Explain `strict` mode benefits

Strict Mode Enables:

1. **noImplicitAny** - Must specify types
2. **strictNullChecks** - Handle null/undefined
3. **strictPropertyInitialization** - Initialize properties
4. **strictFunctionTypes** - Check parameters
5. **strictBindCallApply** - Check bind/call/apply

Impact: 60% fewer null/undefined runtime errors

---

77. How interfaces help in Angular projects?

Use Cases:

1. **API Response Typing** - Type-safe HTTP
2. **Component Input/Output** - Typed metadata
3. **Form Models** - Type-safe values
4. **Service Contracts** - Define interfaces
5. **Route Data** - Type route metadata

---

78. Difference between compile-time and runtime errors

Aspect	Compile-Time	Runtime
When	Development	Execution
Cost	Low	High
User Impact	None	Crashes
Prevention	Type annotations	Validation

---

79. Explain generics with a real-time example

Real-Time Example 1: API Service Layer

```
interface ApiResponse<T> {
    data: T;
    status: number;
}

class ApiService {
    async get<T>(endpoint: string): Promise<ApiResponse<T>> {
        const response = await fetch(endpoint);
        return { data: await response.json() as T, status: response.status };
    }
}
```

Real-Time Example 2: State Management

```
class Store<T> {
    private state: T;
    setState(newState: Partial<T>): void {
        this.state = { ...this.state, ...newState };
    }
}
```

Benefits:

- **Type Safety** - Catch errors at compile time
- **Code Reusability** - Works with any type
- **Better IntelliSense** - IDE knows exact types
- **Flexibility** - Preserve type information

END OF TYPESCRIPT INTERVIEW QUESTIONS - ALL 79 QUESTIONS COMPLETE

Angular Questions – Complete Prep

Beginner Level (0–2 Years)

Angular Basics

1. What is Angular?
2. Difference between AngularJS and Angular
3. What are components?
4. What is a module?
5. What is a service?
6. What is a directive?
7. What is data binding?
8. Types of data binding in Angular
9. What is interpolation?
10. What are templates?
11. What is the purpose of `@NgModule`?

Components and Templates

12. What is a component in Angular?
13. Difference between component and directive
14. How to pass data from parent to child component?
15. How to pass data from child to parent component?
16. What are input and output decorators?
17. What is `@ViewChild` and `@ContentChild`?

Project Configuration Files

angular.json

18. What is `angular.json`?
19. What is its role in an Angular project?
20. How does `angular.json` manage build and serve configurations?
21. How do you configure environments in `angular.json`?
22. How do you add assets and styles globally using `angular.json`?

package.json

23. What is `package.json`?
24. Why is `package.json` important?
25. How does `package.json` manage dependencies?
26. What are `devDependencies` and `dependencies`?
27. What are scripts in `package.json` and how are they used?
28. How to run custom npm scripts defined in `package.json`?

tsconfig.json

29. What is `tsconfig.json`?
30. What compiler options are most important?
31. What does strict mode do?
32. Difference between `target` and `module`?
33. How to enable `strictNullChecks` and `noImplicitAny`?

Webpack

34. What is Webpack?
35. How does Angular use Webpack internally?
36. What is the role of loaders and plugins in Webpack?
37. Difference between development and production build using Webpack?
38. How can you customize Webpack in Angular?

Intermediate Level (2–4 Years)

Directives and Pipes

39. What are structural directives?
40. What are attribute directives?
41. Common built-in directives (`ngIf`, `ngFor`, `ngSwitch`)
42. What is a pipe?
43. Difference between built-in and custom pipe
44. How to create a custom pipe

Services and Dependency Injection

45. What is a service in Angular?
46. How dependency injection works in Angular?
47. What is `providedIn: 'root'`?

48. Difference between service provided in root and module
49. Singleton services vs multiple instances

Routing

50. What is Angular Router?
51. Difference between `routerLink` and `href`
52. How to pass parameters in routes?
53. What are route guards?
54. What is lazy loading?

Forms

55. Difference between template-driven and reactive forms
56. How to create a reactive form?
57. Form validation (built-in and custom)
58. `FormControl`, `FormGroup`, `FormBuilder`
59. How to dynamically add form controls?

State Management

60. What is state management in Angular?
61. Difference between local component state and global application state
62. What is RxJS and how is it used in Angular?
63. What is NgRx?
64. Explain the concept of store, actions, reducers, and selectors
65. Difference between `BehaviorSubject`, `Subject`, and `ReplaySubject`
66. How to manage async state using NgRx Effects
67. How to connect components to store using `select` and `dispatch`
68. Difference between NgRx, Akita, and other state management libraries
69. How to debug state using NgRx DevTools

Advanced Level (4+ Years)

Lifecycle Hooks

70. What are lifecycle hooks?
71. Difference between `ngOnInit` and constructor
72. `ngOnChanges`, `ngDoCheck`, `ngAfterViewInit`, `ngAfterContentInit`
73. When to use each lifecycle hook

Change Detection

74. How change detection works in Angular?
75. What is `zone.js`?
76. Difference between default and `OnPush` change detection strategy
77. How to optimize change detection

RxJS and Observables

78. What is RxJS?
79. What is an Observable?
80. Difference between Promise and Observable
81. Common RxJS operators (`map`, `filter`, `switchMap`, `mergeMap`, `concatMap`, `exhaustMap`)
82. How to unsubscribe from Observables?
83. What is `Subject`, `BehaviorSubject`, `ReplaySubject`, `AsyncSubject`?
84. Difference between cold and hot Observables
85. How to handle error and retry logic in Observables

Angular Modules and Architecture

86. Difference between feature module and root module
87. Shared module vs Core module
88. What is Angular CLI and its usage?
89. What is Ahead-of-Time (AOT) compilation?
90. Difference between JIT and AOT compilation

Frequently Asked / Scenario Questions

91. Explain Angular architecture in your project
92. How did you implement routing in your project?
93. How did you handle state management?
94. Explain your usage of services and dependency injection
95. Explain lifecycle hooks you used in your project
96. How did you optimize change detection in your project?
97. How did you integrate RxJS Observables in your project?
98. Explain Angular forms implementation in your project
99. How do configuration files help in project management and build optimization?

100. Explain difference between development and production builds
101. Explain your usage of NgRx store, actions, reducers, and selectors in your project
102. How do you debug state issues using NgRx DevTools?

Micro Frontend / Micro-Component Questions

103. What is a Micro Frontend (MFE)? When would you use it?
104. Explain advantages and challenges of implementing MFEs in Angular.
105. How would you share common services (e.g., authentication, API service) across multiple MFEs?
106. How would you implement communication between two MFEs (e.g., cart and profile)?
107. How do you handle routing when multiple Angular MFEs are hosted on the same page?
108. How can you lazy load micro-components inside a larger Angular shell?
109. How to allow independent deployment of MFEs without affecting main app?
110. How do you handle versioning of shared libraries across multiple MFEs?
111. Using Web Components / Angular Elements for micro-components – how does it work?
112. How to package and distribute a micro-component for reuse?

Advanced / Missed Questions

Performance & Optimization

113. How does `trackBy` improve performance in `*ngFor`? Give an example.
114. How would you lazy load images or components to improve performance?
115. Explain `OnPush` change detection and when you would use it.
116. How can you use memoization with pipes to optimize performance?
117. How do you avoid unnecessary API calls or component re-rendering?

Advanced RxJS

118. Explain higher-order Observables (`switchMap`, `mergeMap`, `exhaustMap`) with examples.
119. How do you handle errors in Observables using `catchError`, `retry`, or `retryWhen`?
120. How do you combine multiple Observables using `forkJoin`, `combineLatest`, or `zip`?
121. How would you implement debounce or throttle for user input or search functionality?

Testing

122. How do you unit test components, services, pipes, and directives using Jasmine/Karma?
123. How do you test Observables and async operations?
124. How do you use `HttpClientTestingModule` to test API calls?
125. How do you perform end-to-end testing using Protractor or Cypress?

Security

126. How do you protect routes using Guards (`CanActivate`, `CanLoad`)?
127. How do you prevent XSS in Angular templates?
128. How do you secure APIs and tokens on the front-end?
129. How do you use Angular `DomSanitizer` to prevent security risks?

Angular CLI & Tooling

130. What are custom schematics and how do you use them?
131. How do you handle `ng update` and migration strategies?
132. How do you optimize builds using budgets, AOT, minification, and tree-shaking?

Internationalization (i18n)

133. How do you translate Angular apps using Angular i18n or `ngx-translate`?
134. How do you lazy load translations for large Angular applications?

Accessibility

135. How do you use ARIA attributes in Angular components?
136. How do you implement keyboard navigation for Angular components?
137. How do you ensure screen reader compatibility?

Server-side Rendering (Angular Universal)

138. When and why would you use server-side rendering (SSR) in Angular?
139. Difference between pre-rendering and live server rendering in Angular Universal.
140. How do you handle API calls when using Angular Universal?

State Management Patterns

141. How do you use NgRx selectors and memoization for performance?
142. How do you handle side effects for async operations in NgRx (Effects)?
143. Difference between local component state and global state in Angular.
144. How do you share state across multiple MFEs or micro-components?