# NestJS REST API + GraphQL Complete Cheat Sheet 🚀

## 1. BASIC CONCEPTS

### What is NestJS?

* **Node.js framework** built with TypeScript
* Uses **decorators** and **dependency injection**
* Modular architecture (like Angular for backend)

### MVC vs NestJS Pattern:

Traditional MVC: NestJS:

Model → View → Controller Module → Controller → Service → Model

Copy

Apply

### REST vs GraphQL:

REST API: GraphQL:

- Multiple endpoints - Single endpoint (/graphql)

- Fixed response - Custom response (ask what you need)

- GET/POST/PUT/DELETE - Query (read) / Mutation (write)

Copy

Apply

## 2. PROJECT STRUCTURE

src/

├── app.module.ts # Root module

├── app.controller.ts # Root controller

├── app.service.ts # Root service

├── main.ts # App bootstrap

└── task/

├── task.module.ts # Task feature module

├── task.controller.ts # REST endpoints

├── task.service.ts # Business logic

├── task.resolver.ts # GraphQL endpoints

├── schemas/

│ └── task.schema.ts # Database model

├── entities/

│ └── task.entity.ts # GraphQL types

└── dto/

├── create-task.input.ts

└── update-task.input.ts

Copy

Apply

## 3. COMPLETE CODE EXAMPLES

### App Bootstrap

// src/main.ts

import { NestFactory } from '@nestjs/core';

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

async function bootstrap() {

const app = await NestFactory.create(AppModule);

await app.listen(3000);

}

bootstrap();

Copy

Apply

### Root Module

// src/app.module.ts

import { Module } from '@nestjs/common';

import { GraphQLModule } from '@nestjs/graphql';

import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';

import { MongooseModule } from '@nestjs/mongoose';

import { TaskModule } from './task/task.module';

@Module({

imports: [

GraphQLModule.forRoot<ApolloDriverConfig>({

driver: ApolloDriver,

autoSchemaFile: true,

}),

MongooseModule.forRoot('mongodb://admin:password123@localhost:27017/tasks?authSource=admin'),

TaskModule,

],

})

export class AppModule {}

Copy

Apply

### Database Model (Shared by REST & GraphQL)

// src/task/schemas/task.schema.ts

import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';

import { Document } from 'mongoose';

@Schema()

export class Task extends Document {

@Prop({ required: true })

title: string;

@Prop()

description: string;

@Prop({ default: false })

completed: boolean;

@Prop({ default: Date.now })

createdAt: Date;

}

export const TaskSchema = SchemaFactory.createForClass(Task);

Copy

Apply

### Service (Business Logic)

// src/task/task.service.ts

import { Injectable } from '@nestjs/common';

import { InjectModel } from '@nestjs/mongoose';

import { Model } from 'mongoose';

import { Task } from './schemas/task.schema';

@Injectable()

export class TaskService {

constructor(@InjectModel(Task.name) private taskModel: Model<Task>) {}

async findAll(): Promise<Task[]> {

return this.taskModel.find().exec();

}

async findOne(id: string): Promise<Task> {

return this.taskModel.findById(id).exec();

}

async create(createTaskInput: Partial<Task>): Promise<Task> {

const createdTask = new this.taskModel(createTaskInput);

return createdTask.save();

}

async update(id: string, updateTaskInput: Partial<Task>): Promise<Task> {

return this.taskModel.findByIdAndUpdate(id, updateTaskInput, { new: true }).exec();

}

async remove(id: string): Promise<Task> {

return this.taskModel.findByIdAndDelete(id).exec();

}

async findWithFilters(completed?: boolean, limit?: number): Promise<Task[]> {

let query = this.taskModel.find();

if (completed !== undefined) query = query.where('completed').equals(completed);

if (limit) query = query.limit(limit);

return query.exec();

}

}

Copy

Apply

### REST Controller

// src/task/task.controller.ts

import { Controller, Get, Post, Body, Patch, Param, Delete } from '@nestjs/common';

import { TaskService } from './task.service';

import { Task } from './schemas/task.schema';

@Controller('tasks')

export class TaskController {

constructor(private readonly taskService: TaskService) {}

@Post() // POST /tasks

create(@Body() createTaskDto: Partial<Task>) {

return this.taskService.create(createTaskDto);

}

@Get() // GET /tasks

findAll() {

return this.taskService.findAll();

}

@Get(':id') // GET /tasks/:id

findOne(@Param('id') id: string) {

return this.taskService.findOne(id);

}

@Patch(':id') // PATCH /tasks/:id

update(@Param('id') id: string, @Body() updateTaskDto: Partial<Task>) {

return this.taskService.update(id, updateTaskDto);

}

@Delete(':id') // DELETE /tasks/:id

remove(@Param('id') id: string) {

return this.taskService.remove(id);

}

}

Copy

Apply

### GraphQL Types

// src/task/entities/task.entity.ts

import { ObjectType, Field, ID } from '@nestjs/graphql';

@ObjectType()

export class Task {

@Field(() => ID)

id: string;

@Field()

title: string;

@Field({ nullable: true })

description?: string;

@Field()

completed: boolean;

@Field()

createdAt: Date;

}

Copy

Apply

### GraphQL Input Types

// src/task/dto/create-task.input.ts

import { InputType, Field } from '@nestjs/graphql';

@InputType()

export class CreateTaskInput {

@Field()

title: string;

@Field({ nullable: true })

description?: string;

}

// src/task/dto/update-task.input.ts

import { InputType, Field, ID, PartialType } from '@nestjs/graphql';

import { CreateTaskInput } from './create-task.input';

@InputType()

export class UpdateTaskInput extends PartialType(CreateTaskInput) {

@Field(() => ID)

id: string;

@Field({ nullable: true })

completed?: boolean;

}

Copy

Apply

### GraphQL Resolver

// src/task/task.resolver.ts

import { Resolver, Query, Mutation, Args, ID, Int } from '@nestjs/graphql';

import { TaskService } from './task.service';

import { Task } from './entities/task.entity';

import { CreateTaskInput } from './dto/create-task.input';

import { UpdateTaskInput } from './dto/update-task.input';

@Resolver(() => Task)

export class TaskResolver {

constructor(private readonly taskService: TaskService) {}

@Query(() => [Task], { name: 'tasks' })

findAll(

@Args('completed', { type: () => Boolean, nullable: true }) completed?: boolean,

@Args('limit', { type: () => Int, nullable: true }) limit?: number

) {

return this.taskService.findWithFilters(completed, limit);

}

@Query(() => Task, { name: 'task' })

findOne(@Args('id', { type: () => ID }) id: string) {

return this.taskService.findOne(id);

}

@Mutation(() => Task)

createTask(@Args('input') createTaskInput: CreateTaskInput) {

return this.taskService.create(createTaskInput);

}

@Mutation(() => Task)

updateTask(@Args('input') updateTaskInput: UpdateTaskInput) {

return this.taskService.update(updateTaskInput.id, updateTaskInput);

}

@Mutation(() => Task)

removeTask(@Args('id', { type: () => ID }) id: string) {

return this.taskService.remove(id);

}

}

Copy

Apply

### Task Module

// src/task/task.module.ts

import { Module } from '@nestjs/common';

import { MongooseModule } from '@nestjs/mongoose';

import { TaskService } from './task.service';

import { TaskController } from './task.controller';

import { TaskResolver } from './task.resolver';

import { Task, TaskSchema } from './schemas/task.schema';

@Module({

imports: [MongooseModule.forFeature([{ name: Task.name, schema: TaskSchema }])],

controllers: [TaskController],

providers: [TaskService, TaskResolver],

})

export class TaskModule {}

Copy

Apply

## 4. API USAGE EXAMPLES

### REST API Calls:

# Get all tasks

curl http://localhost:3000/tasks

# Create task

curl -X POST http://localhost:3000/tasks \

-H "Content-Type: application/json" \

-d '{"title": "Learn NestJS", "description": "Study framework"}'

# Get specific task

curl http://localhost:3000/tasks/TASK\_ID

# Update task

curl -X PATCH http://localhost:3000/tasks/TASK\_ID \

-H "Content-Type: application/json" \

-d '{"completed": true}'

# Delete task

curl -X DELETE http://localhost:3000/tasks/TASK\_ID

Copy

Execute

### GraphQL Queries:

# Get all tasks (only title and completed)

query {

tasks {

id

title

completed

}

}

# Get specific task

query {

task(id: "TASK\_ID") {

id

title

description

completed

createdAt

}

}

# Get filtered tasks

query {

tasks(completed: true, limit: 5) {

title

completed

}

}

# Create task

mutation {

createTask(input: {

title: "Learn GraphQL"

description: "Study GraphQL with NestJS"

}) {

id

title

completed

}

}

# Update task

mutation {

updateTask(input: {

id: "TASK\_ID"

completed: true

}) {

id

title

completed

}

}

# Delete task

mutation {

removeTask(id: "TASK\_ID") {

id

title

}

}

Copy

Apply

## 5. KEY DECORATORS CHEAT SHEET

### NestJS Core:

* @Module() - Define a module
* @Controller() - Define REST controller
* @Injectable() - Make class injectable
* @Get(), @Post(), @Put(), @Delete() - HTTP methods
* @Param() - Route parameters
* @Body() - Request body
* @Query() - Query parameters

### GraphQL:

* @Resolver() - Define GraphQL resolver
* @Query() - GraphQL query (read)
* @Mutation() - GraphQL mutation (write)
* @Args() - GraphQL arguments
* @ObjectType() - GraphQL output type
* @InputType() - GraphQL input type
* @Field() - GraphQL field

### Database:

* @Schema() - Mongoose schema
* @Prop() - Schema property
* @InjectModel() - Inject Mongoose model

## 6. QUICK SETUP COMMANDS

# Install NestJS CLI

npm i -g @nestjs/cli

# Create new project

nest new project-name

# Generate module

nest g module task

# Generate controller

nest g controller task

# Generate service

nest g service task

# Generate resolver

nest g resolver task

# Install dependencies

npm install @nestjs/mongoose mongoose

npm install @nestjs/graphql @nestjs/apollo apollo-server-express graphql

# Start development

npm run start:dev

# Run tests

npm run test

Copy

Execute

## 7. INTERVIEW TALKING POINTS

### Architecture Benefits:

* **Modular**: Each feature is a separate module
* **Scalable**: Easy to add new features
* **Testable**: Dependency injection makes testing easy
* **Type-safe**: TypeScript throughout

### REST vs GraphQL:

* **Same backend logic**: Both use same service and database
* **Different interfaces**: REST = multiple endpoints, GraphQL = single endpoint
* **GraphQL advantages**: Flexible queries, single request for complex data
* **REST advantages**: Simple, cacheable, familiar

### Key Concepts:

* **Dependency Injection**: Services injected automatically
* **Decorators**: Clean, declarative syntax
* **Guards**: Authentication/authorization
* **Pipes**: Data validation and transformation
* **Interceptors**: Request/response manipulation

### Performance:

* **GraphQL**: Reduces over-fetching, single request
* **REST**: Simple caching, predictable responses
* **Database**: Same queries regardless of API type

## 8. COMMON INTERVIEW QUESTIONS & ANSWERS

**Q: How does NestJS differ from Express?** A: NestJS provides structure with modules, dependency injection, and decorators. Express is minimal, NestJS is opinionated and enterprise-ready.

**Q: Why use both REST and GraphQL?** A: REST for simple CRUD, GraphQL for complex queries. Same business logic, different client needs.

**Q: How does dependency injection work?** A: NestJS container manages object creation. Use @Injectable() and constructor injection.

**Q: What's the request flow?** A: Request → Controller/Resolver → Service → Database → Service → Controller/Resolver → Response

**Q: How do you handle errors?** A: Use exception filters, built-in HTTP exceptions, and GraphQL error handling.

**Good luck with your interview! 🚀**

Excellent question! This is the **core magic** of how GraphQL works. Let me break it down step by step.

## GraphQL Doesn't Use "Controllers" - It Uses "Resolvers"

First, important clarification:

* **REST** uses **Controllers** with different URLs
* **GraphQL** uses **Resolvers** with named functions

## How GraphQL Maps Queries to Functions

### 1. GraphQL Schema Generation

When your app starts, NestJS automatically generates a GraphQL schema from your resolvers:

// Your resolver code:

@Resolver(() => Task)

export class TaskResolver {

@Query(() => [Task], { name: 'tasks' }) // Creates "tasks" query in schema

findAll() {

return this.taskService.findAll();

}

@Query(() => Task, { name: 'task' }) // Creates "task" query in schema

findOne(@Args('id') id: string) {

return this.taskService.findOne(id);

}

@Mutation(() => Task) // Creates "createTask" mutation in schema

createTask(@Args('input') input: CreateTaskInput) {

return this.taskService.create(input);

}

}

Copy

Apply

### 2. Generated GraphQL Schema (Auto-created):

# This schema is automatically generated from your resolver decorators

type Query {

tasks: [Task!]! # Maps to findAll() function

task(id: ID!): Task # Maps to findOne() function

}

type Mutation {

createTask(input: CreateTaskInput!): Task! # Maps to createTask() function

}

type Task {

id: ID!

title: String!

description: String

completed: Boolean!

createdAt: DateTime!

}

## What is a Module?

**A module is an organizing box that groups related code together.**

## Why Use Modules?

1. **Organization** - Keep related stuff together
2. **Separation** - Each module has one job
3. **Reusability** - Share modules between apps
4. **Team Work** - Multiple developers can work without conflicts
5. **Testing** - Test individual features in isolation

## Module Types:

* **AppModule** = Main coordinator (building manager)
* **Feature Modules** = Specialized departments (UserModule, TaskModule, etc.)
* **Shared Modules** = Common services (EmailModule, DatabaseModule)

## Interview Answer:

**"Modules are organizing containers that group related functionality. Like folders on your computer - UserModule contains everything user-related, TaskModule contains everything task-related. AppModule is the main coordinator that imports all feature modules. Benefits: better organization, team collaboration, reusability, and easier testing. Same concept in Angular and NestJS."** 🚀

Think of it this way: **Module = Feature in a Box** 📦

Ah YES! Let me break down TypeScript for you - it's actually simpler than you think! 🚀

# What is TypeScript?

## TypeScript = JavaScript + Types + Extra Features

Think of TypeScript as **JavaScript's smarter older brother** who helps catch your mistakes before you embarrass yourself! 😄

### Simple Analogy:

JavaScript = Speaking without thinking

TypeScript = Speaking with a grammar checker that stops you from saying dumb stuff

Copy

Apply

## TypeScript vs JavaScript:

### JavaScript (Wild West - Anything Goes):

// JavaScript - no safety net

let user = "John";

user = 42; // Sure, why not? 🤷‍♂️

user = true; // Absolutely! 🤷‍♂️

user = { name: "Jane" }; // Of course! 🤷‍♂️

function add(a, b) {

return a + b; // What are a and b? Who knows! 🤷‍♂️

}

add(5, 3); // 8

add("5", 3); // "53" (string concatenation - surprise!)

add(true, null); // 1 (wat? 😵‍💫)

Copy

Apply

### TypeScript (Smart Assistant):

// TypeScript - helpful assistant

let user: string = "John";

user = 42; // ❌ Error: Type 'number' is not assignable to type 'string'

user = true; // ❌ Error: Type 'boolean' is not assignable to type 'string'

user = { name: "Jane" }; // ❌ Error: Type 'object' is not assignable to type 'string'

function add(a: number, b: number): number {

return a + b; // Clear: takes 2 numbers, returns number

}

add(5, 3); // ✅ 8

add("5", 3); // ❌ Error: Argument of type 'string' is not assignable to parameter of type 'number'

add(true, null); // ❌ Error: Invalid arguments

Copy

Apply

# How TypeScript Converts to JavaScript

## TypeScript is NOT a Runtime Language!

**Key Point:** TypeScript only exists during development. When you run your app, it's pure JavaScript!

### The Conversion Process:

1. You write TypeScript (.ts files)

2. TypeScript Compiler (tsc) converts to JavaScript (.js files)

3. Browser/Node.js runs the JavaScript

Copy

Apply

### Example Conversion:

#### TypeScript Code (what you write):

// user.ts

interface User {

name: string;

age: number;

}

class UserService {

private users: User[] = [];

addUser(user: User): void {

this.users.push(user);

}

getUsers(): User[] {

return this.users;

}

}

const userService = new UserService();

userService.addUser({ name: "John", age: 25 });

Copy

Apply

#### Compiled JavaScript (what actually runs):

// user.js (generated automatically)

class UserService {

constructor() {

this.users = [];

}

addUser(user) {

this.users.push(user);

}

getUsers() {

return this.users;

}

}

const userService = new UserService();

userService.addUser({ name: "John", age: 25 });

Copy

Apply

### Notice What Disappeared:

* ❌ interface User - gone!
* ❌ : User[] - gone!
* ❌ : void - gone!
* ❌ : User - gone!
* ✅ Only the actual logic remains

### TypeScript Compilation Commands:

# Install TypeScript compiler

npm install -g typescript

# Compile single file

tsc user.ts

# Compile and watch for changes

tsc user.ts --watch

# Compile entire project

tsc

# Compile and run immediately

npx ts-node user.ts

Copy

Execute

# DSA Questions - JavaScript vs TypeScript

## YES! You can absolutely do DSA in TypeScript! It's actually BETTER! 🎯

### 1. Array Operations:

#### JavaScript:

// JavaScript - basic array operations

let numbers = [1, 2, 3, 4, 5];

console.log(numbers);

// Add elements

numbers.push(6);

numbers.unshift(0);

// Remove elements

let last = numbers.pop();

let first = numbers.shift();

// Find elements

let found = numbers.find(x => x > 3);

let filtered = numbers.filter(x => x % 2 === 0);

// Transform

let doubled = numbers.map(x => x \* 2);

let sum = numbers.reduce((acc, curr) => acc + curr, 0);

Copy

Apply

#### TypeScript (Same Code + Type Safety):

// TypeScript - same operations but with type safety

let numbers: number[] = [1, 2, 3, 4, 5];

console.log(numbers);

// Add elements (TypeScript ensures you add numbers)

numbers.push(6); // ✅ OK

numbers.push("hello"); // ❌ Error: Argument of type 'string' is not assignable to parameter of type 'number'

// Remove elements (TypeScript knows return types)

let last: number | undefined = numbers.pop(); // TypeScript knows pop() might return undefined

let first: number | undefined = numbers.shift();

// Find elements (TypeScript knows types)

let found: number | undefined = numbers.find(x => x > 3);

let filtered: number[] = numbers.filter(x => x % 2 === 0);

// Transform (TypeScript infers return types)

let doubled: number[] = numbers.map(x => x \* 2);

let sum: number = numbers.reduce((acc, curr) => acc + curr, 0);

Copy

Apply

### 2. Common DSA Problems in TypeScript:

#### Two Sum Problem:

// TypeScript version with type safety

function twoSum(nums: number[], target: number): number[] {

const map = new Map<number, number>();

for (let i = 0; i < nums.length; i++) {

const complement = target - nums[i];

if (map.has(complement)) {

return [map.get(complement)!, i]; // ! tells TS we know it exists

}

map.set(nums[i], i);

}

return []; // TypeScript ensures we return number[]

}

// Usage with type checking

const result: number[] = twoSum([2, 7, 11, 15], 9);

console.log(result); // [0, 1]

// TypeScript catches errors

twoSum("invalid", 9); // ❌ Error: Argument of type 'string' is not assignable to parameter of type 'number[]'

twoSum([1, 2, 3], "9"); // ❌ Error: Argument of type 'string' is not assignable to parameter of type 'number'

Copy

Apply

#### Binary Search:

function binarySearch(arr: number[], target: number): number {

let left: number = 0;

let right: number = arr.length - 1;

while (left <= right) {

const mid: number = Math.floor((left + right) / 2);

if (arr[mid] === target) {

return mid;

} else if (arr[mid] < target) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1; // TypeScript ensures consistent return type

}

// Usage

const sortedArray: number[] = [1, 3, 5, 7, 9, 11];

const index: number = binarySearch(sortedArray, 7);

console.log(index); // 3

Copy

Apply

#### Linked List:

// Define types for better structure

class ListNode {

val: number;

next: ListNode | null;

constructor(val?: number, next?: ListNode | null) {

this.val = val ?? 0;

this.next = next ?? null;

}

}

function reverseLinkedList(head: ListNode | null): ListNode | null {

let prev: ListNode | null = null;

let current: ListNode | null = head;

while (current !== null) {

const nextTemp: ListNode | null = current.next;

current.next = prev;

prev = current;

current = nextTemp;

}

return prev;

}

// Usage

const head = new ListNode(1, new ListNode(2, new ListNode(3)));

const reversed = reverseLinkedList(head);

Copy

Apply

#### Stack Implementation:

class Stack<T> { // Generic type - works with any type!

private items: T[] = [];

push(item: T): void {

this.items.push(item);

}

pop(): T | undefined {

return this.items.pop();

}

peek(): T | undefined {

return this.items[this.items.length - 1];

}

isEmpty(): boolean {

return this.items.length === 0;

}

size(): number {

return this.items.length;

}

}

// Usage with different types

const numberStack = new Stack<number>();

numberStack.push(1);

numberStack.push(2);

console.log(numberStack.pop()); // 2

const stringStack = new Stack<string>();

stringStack.push("hello");

stringStack.push("world");

console.log(stringStack.pop()); // "world"

Copy

Apply

### 3. Advanced DSA with TypeScript:

#### Graph Representation:

// Type-safe graph representation

type Graph = Map<number, number[]>;

function createGraph(edges: number[][]): Graph {

const graph: Graph = new Map();

for (const [u, v] of edges) {

if (!graph.has(u)) graph.set(u, []);

if (!graph.has(v)) graph.set(v, []);

graph.get(u)!.push(v);

graph.get(v)!.push(u);

}

return graph;

}

function bfs(graph: Graph, start: number): number[] {

const visited = new Set<number>();

const queue: number[] = [start];

const result: number[] = [];

while (queue.length > 0) {

const node = queue.shift()!;

if (!visited.has(node)) {

visited.add(node);

result.push(node);

const neighbors = graph.get(node) || [];

queue.push(...neighbors.filter(n => !visited.has(n)));

}

}

return result;

}

Copy

Apply

#### Dynamic Programming:

// Fibonacci with memoization

function fibonacci(n: number, memo: Map<number, number> = new Map()): number {

if (n <= 1) return n;

if (memo.has(n)) {

return memo.get(n)!;

}

const result = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);

memo.set(n, result);

return result;

}

// Coin Change Problem

function coinChange(coins: number[], amount: number): number {

const dp: number[] = new Array(amount + 1).fill(Infinity);

dp[0] = 0;

for (let i = 1; i <= amount; i++) {

for (const coin of coins) {

if (coin <= i) {

dp[i] = Math.min(dp[i], dp[i - coin] + 1);

}

}

}

return dp[amount] === Infinity ? -1 : dp[amount];

}

Copy

Apply

# Benefits of TypeScript for DSA:

## 1. ****Catch Errors Early:****

function binarySearch(arr: number[], target: number): number {

// TypeScript catches if you pass wrong types

}

binarySearch([1, 2, 3], "hello"); // ❌ Error caught at compile time!

Copy

Apply

## 2. ****Better IDE Support:****

let arr: number[] = [1, 2, 3];

arr. // IDE shows: push, pop, map, filter, etc. with type info!

Copy

Apply

## 3. ****Self-Documenting Code:****

// Clear what function expects and returns

function quickSort(arr: number[]): number[] {

// Implementation

}

// vs JavaScript - unclear what it does

function quickSort(arr) {

// What is arr? What does it return? 🤷‍♂️

}

Copy

Apply

## 4. ****Generics for Reusable Code:****

// Works with any type!

class Queue<T> {

private items: T[] = [];

enqueue(item: T): void {

this.items.push(item);

}

dequeue(): T | undefined {

return this.items.shift();

}

}

const numberQueue = new Queue<number>();

const stringQueue = new Queue<string>();

const objectQueue = new Queue<{id: number, name: string}>();

Copy

Apply

# Running TypeScript Code:

## Option 1: Compile then Run

# Write code in file.ts

tsc file.ts # Creates file.js

node file.js # Run the JavaScript

Copy

Execute

## Option 2: Direct Execution

# Install ts-node

npm install -g ts-node

# Run TypeScript directly

ts-node file.ts

Copy

Execute

## Option 3: Online Playground

* Go to [TypeScript Playground](command:_cody.vscode.open?%22https%3A%2F%2Fwww.typescriptlang.org%2Fplay%22)
* Write TypeScript code
* See JavaScript output in real-time
* Run code immediately

# Summary:

## What is TypeScript?

**JavaScript + Type Safety + Modern Features**

## How does it work?

**TypeScript → Compiler → JavaScript → Runtime**

## Can you do DSA in TypeScript?

**Absolutely! Same algorithms, but with:**

* ✅ Type safety
* ✅ Better IDE support
* ✅ Catch errors early
* ✅ Self-documenting code
* ✅ Generics for reusable data structures

## Interview Answer:

**"TypeScript is JavaScript with static typing. It compiles to plain JavaScript - types are removed at runtime. For DSA, it's actually better than JavaScript because you get type safety, better IDE support, and catch errors early. All JavaScript algorithms work in TypeScript with added type annotations."** 🚀

**Bottom line: If you can code DSA in JavaScript, you can definitely do it in TypeScript - just with extra safety and better tooling!** 💪