Nest JS

Home Content Lab

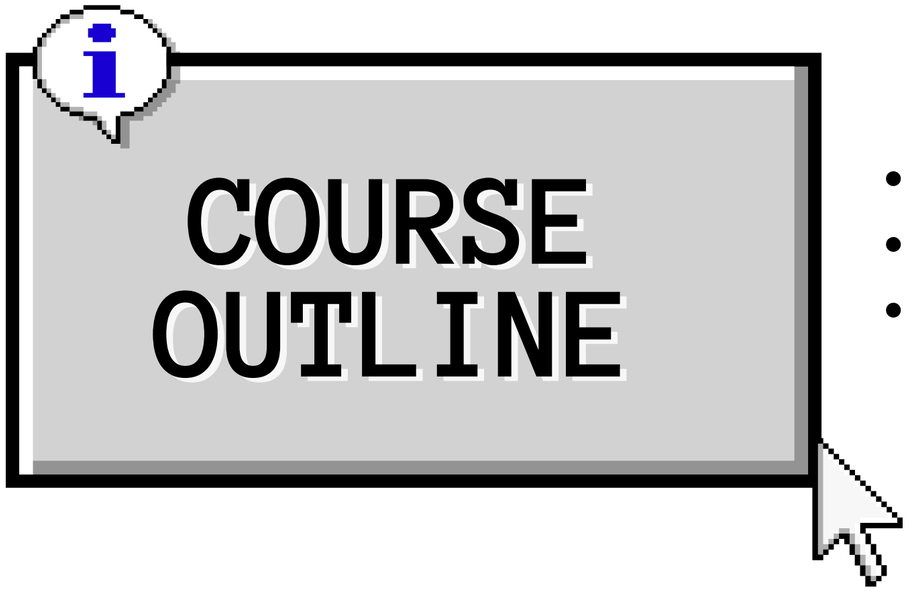
Introduction to

NEST JS

Start

Nest JS

Home **Content** Lab



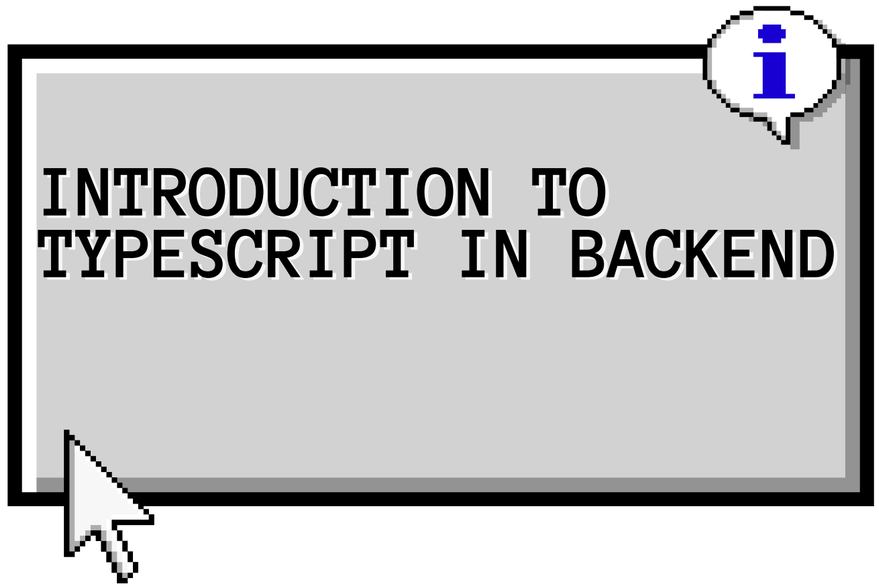
- **TypeScript, OOP & NestJS Core Concepts**
- **REST API Development in NestJS**
- **Database Integration & Advanced NestJS Concepts**

Nest JS

Home **Content** Lab

- **Why TypeScript over JavaScript?**
- **Static typing**
- **interfaces**
- **decorators**
- **generics**

Benefits in large apps: maintainability, readability, refactoring ease.

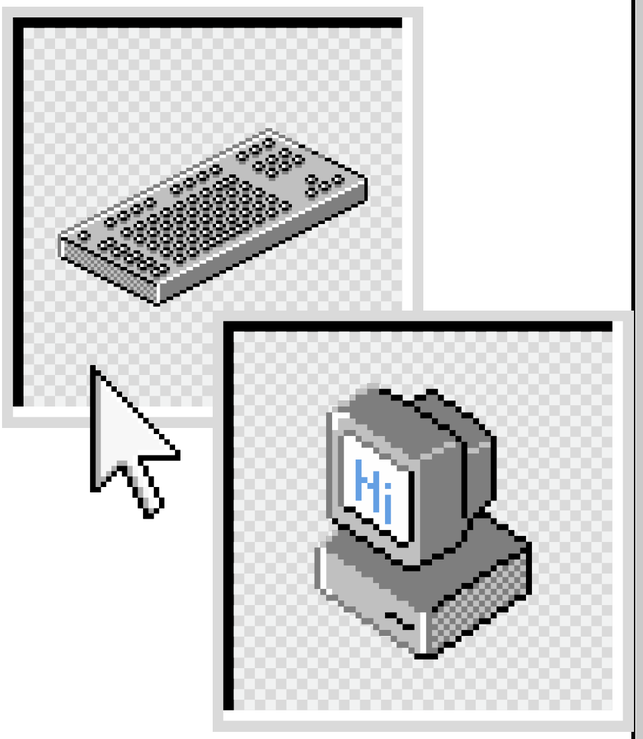


🧩 How Classes Work in TypeScript

TypeScript classes are syntactic sugar over JavaScript's prototype-based inheritance

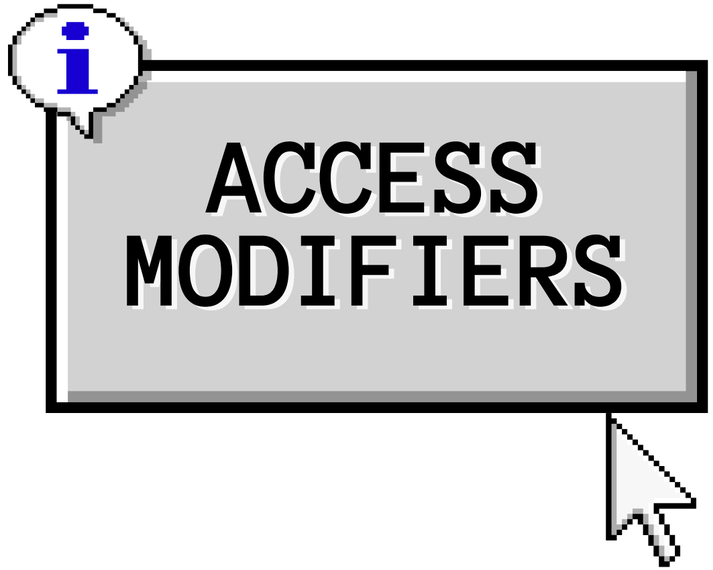
```
class User {
  constructor(public name: string, private age: number) {}

  greet() {
    return `Hello, my name is ${this.name} and I am ${this.age}`;
  }
}

const user = new User("Shady", 23);
console.log(user.greet());
```

Technically, JS does not have "true classes"; the class keyword just wraps prototypes and constructor functions.



Nest JS

Home **Content** Lab



🧩 Access Modifiers

- **public**: property accessible everywhere.
- **private**: only visible inside the class (compiled to a convention _name in JS).
- **protected**: visible in subclasses.

TypeScript enforces this at compile-time only.
The JavaScript runtime does not actually hide the properties (unless you use the #privateField syntax).

INHERITANCE

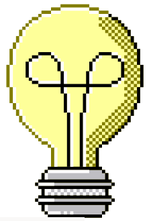
```
class Animal {
  constructor(public name: string) {}
  move(distance: number) {
    console.log(`${this.name} moved ${distance}m.`);
  }
}

class Dog extends Animal {
  bark() {
    console.log("Woof!");
  }
}

const dog = new Dog("Buddy");
dog.bark();
dog.move(10);
```

INTERFACES

```
interface IUser {
  name: string;
  email: string;
}

function printUser(user: IUser) {
  console.log(user.name, user.email);
}
```

- **Interfaces** are compile-time contracts — they do not exist in the final JS code.
- After compilation, they completely disappear.
- They’re purely for type checking, not for execution.

DECORATORS



A **decorator** is a special function that can add extra behavior to a class, method, property, or parameter — without changing its actual code.

🧠 Think of it as:
“A function that wraps something to give it extra features.”

🧩 Why Use Decorators?

- Reuse logic easily
- Make code cleaner and more declarative
- Used heavily in frameworks like NestJS, Angular, etc.

⚙️ How It Works
A decorator is just a function that receives some metadata about the item it decorates.

```
function Logger(target: Function) {
  console.log(`Class created: ${target.name}`);
}

@Logger
class User {
  constructor(public name: string) {}
}

const u = new User("Shady");
```

DEPENDENCY INJECTION

A **design pattern** where an object's dependencies are provided (injected) from the outside —
instead of the object creating them itself.

💡 The Problem Without DI
Without DI, classes depend on each other directly:

```
class UsersService {
  private db = new DatabaseService(); // hard-coded
}
```

This creates tight coupling — hard to test, replace, or reuse.

🧠 The Principle
Dependency Injection inverts control:
Instead of a class creating its dependencies, it receives them from the outside (an IoC container).

```
class EmailService {}
class UserService {
  constructor(private emailService: EmailService) {}
}
```

Nest JS

Home **Content** Lab

NEST JS

💡 What is NestJS?

NestJS is a progressive Node.js framework for building scalable, maintainable, and testable server-side applications.

It uses:

- TypeScript by default
- Express (or Fastify) under the hood
- Dependency Injection and Decorators (inspired by Angular)

🧠 Why Use NestJS

- Modular architecture (like Angular)
- Built-in Dependency Injection
- Structured and testable codebase
- First-class support for REST APIs, GraphQL, WebSockets, and Microservices
- Integrates easily with MongoDB, PostgreSQL, Redis, etc.

NEST JS

⚙️ Installation & Setup

📦 1. Install Nest CLI

```
npm i -g @nestjs/cli
```

🏗️ 2. Create a New Project

```
nest new project-name
```

Then choose your package manager (npm or yarn).

▶️ 3. Run the Application

```
npm run start
```

Or for automatic reload during development:

```
npm run start:dev
```

🧰 Common CLI Commands

Purpose	Command
Create a new module	`nest g module users`
Create a new controller	`nest g controller users`
Create a new service	`nest g service users`
Create a new middleware	`nest g middleware logger`
Create a new guard	`nest g guard auth`
Create a new interceptor	`nest g interceptor transform`
Create a new pipe	`nest g pipe validation`
Create a new filter	`nest g filter exception`
Create a new interface	`nest g interface user`

NEST JS

NestJS follows a modular, layered architecture inspired by Angular and Spring Boot.

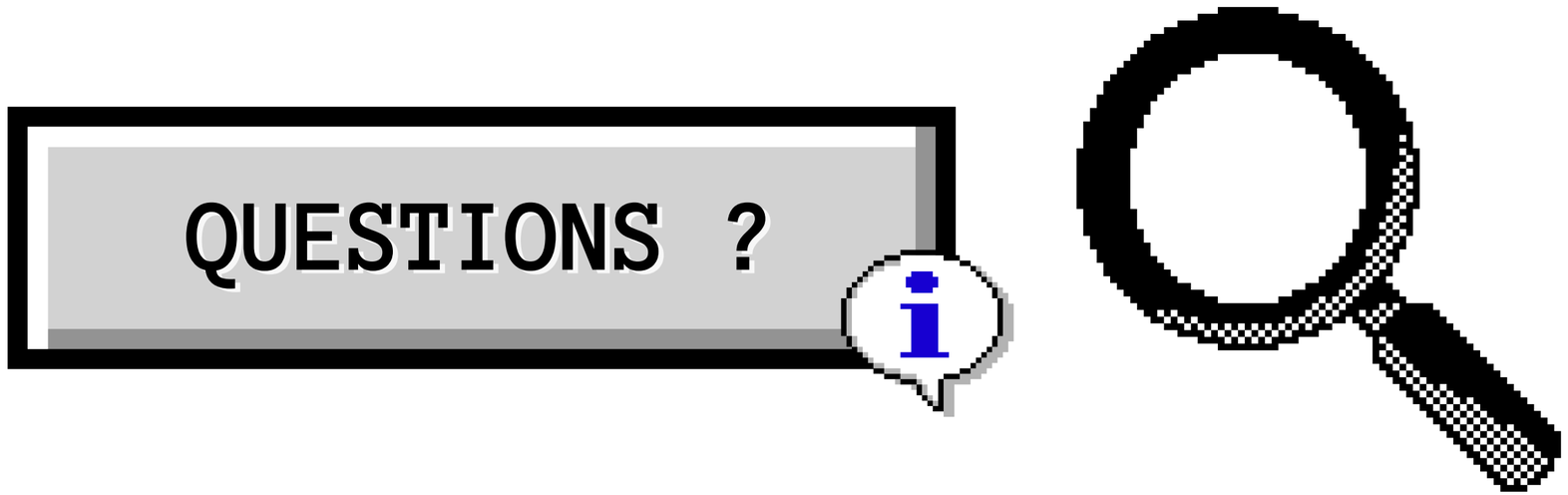
🧠 **Internal Flow (Simplified)**

HTTP Request → Router → Controller → Service → Provider → Database

1. Router Explorer (core module) scans all @Controller classes.
2. It reads metadata from @Get(), @Post() decorators.
3. It binds them to routes in Express.
4. When a request arrives:
   - The controller method is called.
   - Dependencies are resolved from the DI container.
   - Responses are sent back to the client.

Nest JS

Home **Content** Lab



Nest JS

Home Content **Lab**

LAB

- **Create a simple NestJS app with Users Module.**

-bonus: make DTO’s Validation

123-456-7890 123-456-7890 123-456-7890