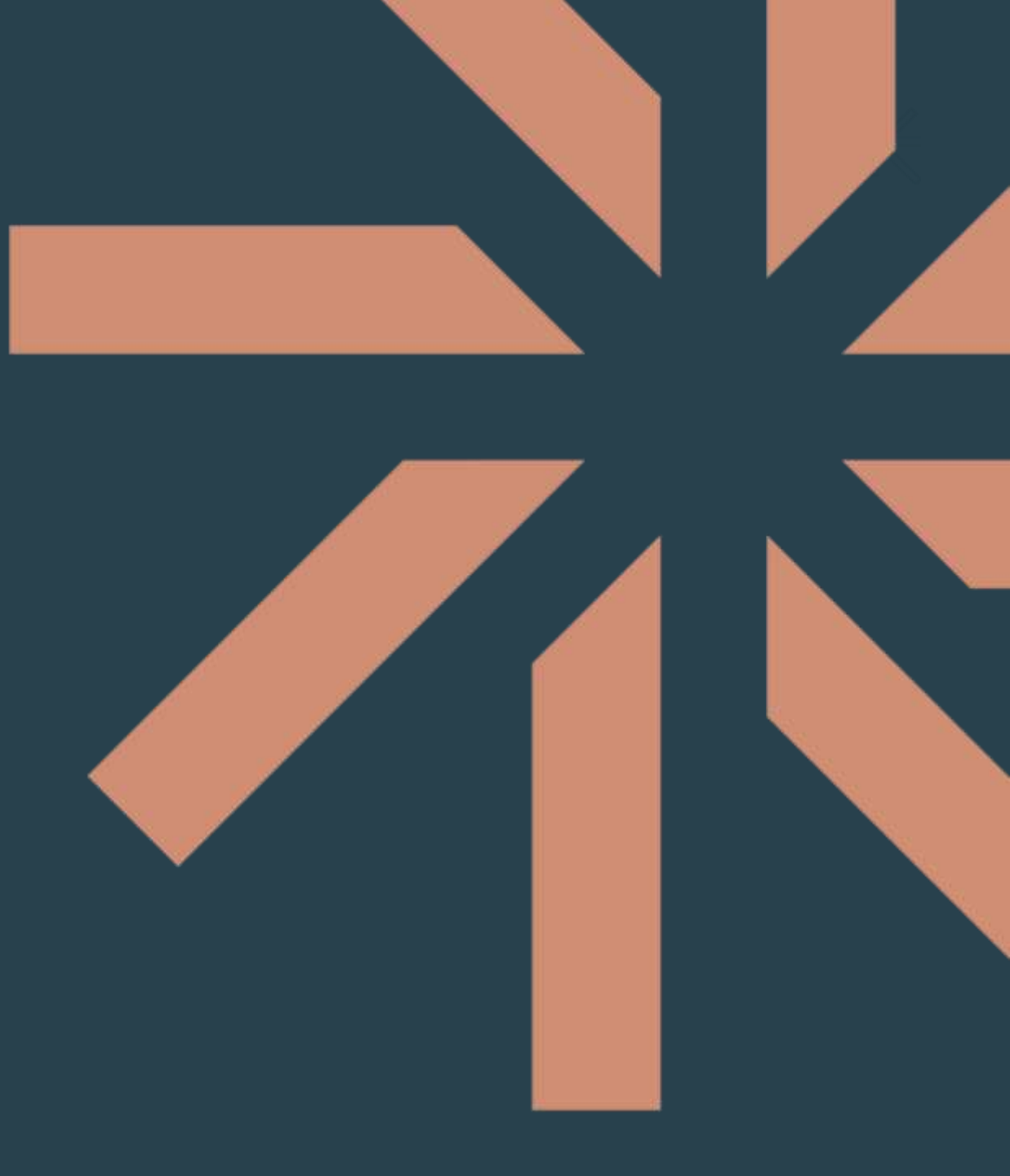
NestJS Intro & CLI

Qinshift Academy

NestJS

NestJS is a progressive Node.js framework for building efficient, scalable, and maintainable server-side applications using TypeScript and modern JavaScript, inspired by Angular’s architecture.



Nest JS CLI

The NestJS CLI is a command-line interface tool that helps developers create, develop, and manage NestJS applications more efficiently by automating common tasks like generating modules, controllers, services, and more, following the framework’s best practices and structure.

Example:

nest generate module users

What it does:

This command creates a new module called UsersModule with a properly structured file like users/users.module.ts.

Why NestJS?

- Modular Architecture
- Dependency Injection (DI)
- Decorators and Metadata
- Built-in HTTP Routing
- Support for Middleware, Guards, Interceptors, and Pipes
- Built-in Testing Utilities
- Platform Agnostic (via Adapters)
- Integration with Modern Libraries
- Command-Line Interface (CLI)
- TypeScript First (but JS compatible)

Modular Architecture

NestJS applications are organized into modules, which group related controllers, services, and providers.

- Benefit: Promotes separation of concerns, reusability, and scalability.
- Example: UsersModule, AuthModule, etc.

Dependency Injection (DI)

NestJS uses a powerful and flexible DI system built on top of TypeScript decorators.

Benefit: Simplifies managing dependencies and promotes loosely coupled code.

Example:

```
constructor(private readonly usersService: UsersService) {}
```

Decorators and Metadata

NestJS heavily uses TypeScript decorators for defining routes, modules, services, etc.

Benefit: Code is more expressive and intuitive.

Example:

```
@Controller('users')
export class UsersController {}
```

Built-in HTTP Routing

Routes are defined using decorators and are handled by controllers.

Benefit: Clean and readable syntax for RESTful APIs.

Example:

```
@Get()
findAll() { return this.usersService.findAll(); }
```

Support for Middleware, Guards, Interceptors, and Pipes

- Middleware: Code that runs before route handlers (e.g., logging, authentication).
- Guards: Used for authorization logic.
- Interceptors: Transform data before/after method execution.
- Pipes: Handle validation and transformation of input data.
- Benefit: Full control over request/response lifecycle.

Built-in Testing Utilities

Tools for writing unit and integration tests using Jest.

- Benefit: Encourages test-driven development (TDD) and code reliability.

- Example:

```
describe('UsersService', () => {
  it('should return all users', () => {
    expect(service.findAll()).toEqual([]);
  });
});
```

Platform Agnostic (via Adapters)

NestJS can run on different platforms like HTTP (Express/Fastify), WebSockets, GraphQL, etc.

Benefit: Same architecture, different protocols.

Example: @WebSocketGateway() for WebSockets.

Integration with Modern Libraries

- TypeORM / Prisma: For database interaction.
- Passport.js: For authentication.
- GraphQL: With decorators like @Resolver().
- Swagger: For auto-generated API docs.

TypeScript First (but JS compatible)

Designed for TypeScript out of the box, providing static typing and IDE support.

Benefit: Fewer runtime errors and better developer experience.

Controllers

What is a Controller in NestJS?

A Controller in NestJS is a class that handles incoming HTTP requests and sends responses back to the client.

Role in the App:

- Acts as a bridge between the client and the business logic.
- Defines routes and maps them to handler functions.

Key Features of Controllers

- Uses decorators to define routes: @Controller(), @Get(), @Post(), etc.
- Handles request routing and response returning.
- Delegates business logic to services.
- Accesses request data using parameters like @Body(), @Param(), @Query(), etc.

Common HTTP Method Decorators

Decorator	HTTP Method	Purpose
@Get()	GET	Retrieve data
@Post()	POST	Create new data
@Put()	PUT	Replace existing data
@Patch()	PATCH	Update part of the data
@Delete()	DELETE	Remove data

Best Practices for Controllers

- Keep controllers thin — don’t add business logic here.
- Always delegate work to services.
- Use DTOs (Data Transfer Objects) to validate and structure request data.
- Keep endpoints clear and RESTful.

Questions?

Trainer Name

Trainer

trainer@mail.com

Assistant Name

Assistant

asistant@mail.com

Qinshift Academy