Nestjs

Nest is a framework for building efficient, scalable node.js server-side applications.it uses progressive JavaScript, is built with and fully Support typescript and combines elements of OOP, functional programming, and functional reactive programming.

Installation

* npm I –g @nestjs/cli
* nest new project-name

**Nodejs and Nestjs**

Development Language

Node js primarily uses JavaScript

Nestjs uses typescript which is a superset of JavaScript with static typing.

**File structure**

In nodejs application, especially smaller ones or those developed quickly developer might opt to organize their code in a simple and straightforward manner. This often involves putting all server logic into single file “index.js”.

To improve organization and maintainability, developers might then choose to separate concerns by creating a folder structure with distinct directories for different components of the application.

Controller

This directory could contain modules responsible for handling incoming HTTP requests, parsing input, and forming appropriate responses.

Services

Developers might place modules that encapsulate business logic or perform specific tasks needed by the application.

Models

Directory might have data structure, entities used by application, as well as database interaction logic if using a database.

**NestJs** application, the file structure follows a modular approach, which is designed to promote scalability, maintainability and organization.

Controllers handle incoming request and return reponses

Providers, Services that contain core business logic, interacting with databases, making api calls.

Modules, self contained units of business capabilities that group related controllers/providers.

Middleware, functions executed during the request handling pipeline

Guards, function that control access to specific endpoints

**Request handling flow**

When a request comes into a nodejs app, we manually take that request and route it across any number of modules, controllers, and services we have wired up. Any middleware needs to be explicitly inserted as well.

Nestjs handles a request using dependency injection and its built-in inversion of control container. A request first hits any global middleware then flows through a pipeline

Request ---------- guard -----------interceptor----------controller-------------service--------repository

Performance

Runtime performance and benchmark

When we talk about “runtime performance” we are comparing how fast the code runs in Node.js compared to when it is running in NestJs

Benchmark are tests that measure the performance of code. In this case, hey are measuring how many requests per seconds each can handle.

NestJs adds only a tiny bit of extra work for our computer compared to NodeJs we are talking less than 1% difference. If nodejs can handle 31,000 requests per second, NestJs might handle around 30,000

Development velocity means how quickly developers can build stuff.

Nestjs makes developers faster because it comes with lots of helpful tools and rules for how to build things.

These tools and rules help developers organize their code better and avoid mistakes. They also make it easier for new team members to understand and work on the code.

**Controllers**

Controller are responsible for handling incoming requests and returning responses to the client.

controller

controller

controller

Client side

A controller purpose is to receive specific request for the application. The routing mechanism controls which controller receives which requests. Each controller has more than one route, and different routes can perform different actions.

In order to create a basic controller, we use classes and decorators. Decorator associate classes with required metadata and enable nest to create a routing map.

**@Controller(“pathprefix”)**

“pathprefix” using in a controller decorator allows us to easily group a set of related routes, and minimize repetitive code.

cat.controller.ts

import { Controller, Get } from '@nestjs/common';

@Controller('cats')

export class CatsController {

  @Get()

  findAll(): string {

    return 'This action returns all cats';

  }

}

Request object

import { Controller, Get, Req } from "@nestjs/common";

import { Request } from "express";

@Controller("cats")

export class CatsController {

  @Get()

  findAll(@Req() request: Request): string {

    return "This action returns all cats";

  }

}

|  |  |
| --- | --- |
| @Request(), @Req() | req |
| @Respnse(), @Res()\* | res |
| @Next() | next |
| @Session() | req.session |
| @Param(key?:string) | req.params / req.params[key] |
| @Body(key?:string) | req.body / req.body[key] |
| @Query(key?:string) | req.query / req.query[key] |
| @Headers(name?:string) | req.headers / req.headers[name] |
| @Ip() | req.ip |
| @HostParam() | req.hosts |

Middleware

Middleware is a function which is called before the route handler. Middleware function have access to the request and response object, and the next() middleware function in the applications request-response cycle. The next middleware function is commonly denoted by a variable named next.

Middleware function can perform the following tasks:

execute any code.

make changes to the request and the response object.

end the request-response cycle

call the next middleware function in the stack.

If the current middleware function doesn’t end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise the request will be left hanging.

Pipes

Pipes have two typical use cases:

**Transformation**: transform input data to the desired form (e.g. from string to integer).

**Validation**: evaluate input data and if valid, simply pass through unchanged; otherwise throw a exception.

Built-in pipes

Nest comes with nine pipes available and exported from @nestjs/common

ValidationPipe

ParseIntPipe

ParseFloatPipe

ParseBoolPipe

ParseArrayPipe

ParseUUIDPipe

ParseEnumPipe

DefaultValuePipe

ParseFilePipe

Guards

Guards have a single responsibility. They determine whether a given request will be handled by the route handler or not, depending on certain conditions (like permissions, roles, ACLs, etc.) present at run-tim.

Interceptors

Inceptor have a set of useful capabilities which are inspired by the Aspect oriented programming(AOP) technique. They make it possible to

bind extra logic before / after method execution

transform the result returned from a function

transform the execution thrown from a function

extend the basic function behavior

completely override a function depending on specific conditions (e.g for caching purpose)