


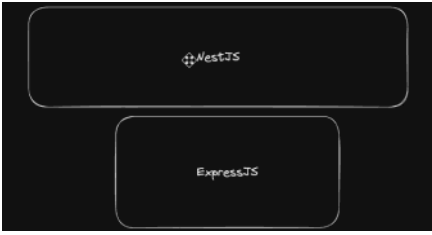
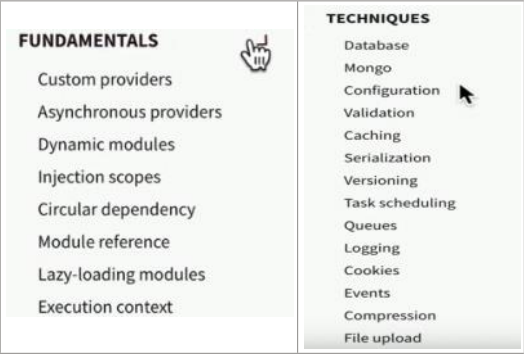

Hello, nest!

A progressive Node.js framework for building efficient, reliable and scalable server-side applications.

Nest (NestJS) is a framework for building efficient, scalable **Node.js** server-side applications. It uses progressive JavaScript, is built with and fully supports **TypeScript** (yet still enables developers to code in pure JavaScript) and combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming).

Under the hood, Nest makes use of robust HTTP Server frameworks like **Express** (the default) and optionally can be configured to use **Fastify** as well!

Why nest.js ?

	We have express but it is minimalistic web framework that provides us some concepts to let us build the apis, but for most part the overall architecture is up to us.
Express provides us docs on routing, middleware, error handling and that is all. So this architecture means that it is very flexible but as our project grows we need structure. It gets complicated when we are working with graphql, swagger etc, it is up to us how to integrate them in express.	
Nest JS sits on top of express, it is basically architecture out of the box and provides some guidelines	
Nestjs provides a lot of good documentation to do all this. 	 Big focus on use of decorators similar to other frameworks like Spring Boot.

Command to install nest cli:

```
npm install -g @nestjs/cli@latest
```

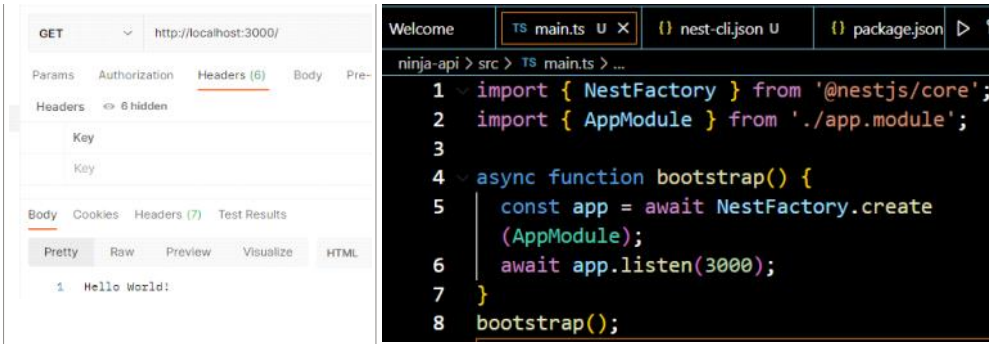
Install nestjs cli. Helps us to generate new projects and provide other commands that would help us.

Command to make new project:

```
nest new ninja-api
< We will scaffold your app in a few seconds..

? Which package manager would you like to use? (Use arrow keys)
> npm
  yarn
  pnpm
```

use command **npm run start:dev** to start in watch mode.



main.ts is the entry point. As you can see, it is listening on port 3000.

HTTP GET / --> controller --> service

Core pieces of the architecture that nest.js provides:

1. Module

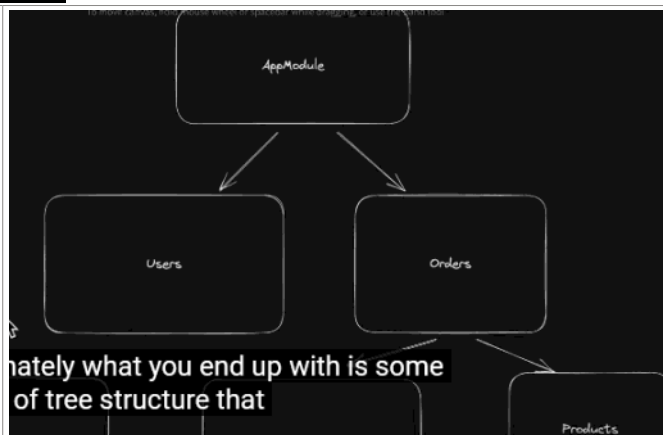
```
@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Consider this AppModule as the root of our application. We can add additional modules as well. Consider module as a way to group and encapsulate a closely related set of capabilities.

Mostly for each feature, we have a module as they have their own routes(Controllers) and their separate business logic(Services).

Eg, module for Users, Orders, Products etc.

They may be dependent on each other.



To generate a new module command is **nest g module <moduleName>**

```
PS C:\Users\Dhruv\Desktop\Nest.js\ninja-api> nest generate module ninjas
CREATE src/ninjas/ninjas.module.ts (83 bytes)
UPDATE src/app.module.ts (316 bytes)
PS C:\Users\Dhruv\Desktop\Nest.js\ninja-api>
```

It created a module in ninjas folder.

As you can see it also updated app.module.ts. It automatically imported and started making a dependency tree.

```
@Module({
  imports: [NinjasModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Remember to register the modules, controllers, services you are making to the correct module.

Command for adding a controller: **nest g controller ninjas**

Command for adding a service: **nest g service ninjas**

It automatically gets registered in the Ninjasmodule

```
1 import { Module } from '@nestjs/common';
2 import { NinjasController } from './ninjas.controller';
3 import { NinjasService } from './ninjas.service';
4
5 @Module({
6   controllers: [NinjasController],
7   providers: [NinjasService]
8 })
9 export class NinjasModule {}
10
```

To generate all resources for a module using a single command: **nest g resource users**

Use this command when you are fully aware of nest fundamentals

```
> nest g resource users
? What transport layer do you use? REST API
? Would you like to generate CRUD entry points? Yes
CREATE src/users/users.controller.spec.ts (566 bytes)
CREATE src/users/users.controller.ts (894 bytes)
CREATE src/users/users.module.ts (247 bytes)
CREATE src/users/users.service.spec.ts (453 bytes)
CREATE src/users/users.service.ts (609 bytes)
CREATE src/users/dto/create-user.dto.ts (30 bytes)
CREATE src/users/dto/update-user.dto.ts (169 bytes)
CREATE src/users/entities/user.entity.ts (21 bytes)
UPDATE package.json (1973 bytes)
UPDATE src/app.module.ts (381 bytes)
✓ Packages installed successfully.
```

Controllers:

Define the paths, the http methods for each path

```
ninja-api > src > ninjas > TS ninjas.controller.ts > NinjasController
1 import { Controller } from '@nestjs/common';
2
3 @Controller('ninjas')
4 export class NinjasController {}
5
6 // GET /ninjas -->[]
7 // GET /ninjas/:id --> {...}
8 // POST /ninjas
9 // PUT /ninjas/id -->{...}
10 // DELETE /ninjas/:id
11
```

The string we have in the decorator `Controller("ninjas")` means that every route will be prefixed by `/ninjas`.

In order to create these routes we need a class annotated with controller decorator.

```

1 import { Body, Controller, Delete, Get, Param, Post, Put } from '@nestjs/common';
2 import { CreateNinjaDto } from '../dto/create-ninja.dto';
3 import { UpdateNinjaDto } from '../dto/update-ninja.dto';
4
5 @Controller('ninjas')
6 export class NinjasController {
7
8   // GET /ninjas --> []
9   @Get()
10  getNinjas() {
11    return [];
12  }
13
14   // GET /ninjas/:id --> {...}
15   @Get('/:id')
16   getOneNinja(@Param('id') id:string) {
17     return {
18       id
19     };
20   }
21
22   // POST /ninjas
23   @Post()

```

Primary role of controllers is **defining the routes**, parsing the path parameters, query strings from the url, parse request body and then forwarding then down to services or providers that contain business logic.

```

// GET /ninjas/:id --> {...}
@Get('/:id')
getOneNinja(@Param("id") id:string){
  return {};
}

```

How to parse the path parameter in the request. For that we have `@Param` decorator.

Behind the scenes nest is parsing the url and auto injects the id in our code.

What about query parameters. Url may contain query parameters

```
// GET /ninjas?type=fast --> []
```

```

@Get()
getNinjas(@Query('type') type: string) {
  return [{ type }];
}

```

Similarly we can parse the request Body using `@Body` decorator. Remember to start the server again if you have added new routes or created new files and registered them in the module.

```

@Post()
createNinja(@Body() createNinjaDto: CreateNinjaDto) {
  return {
    name: createNinjaDto.name,
  };
}

```

We have create a dto as well for this

```

ninja-api > src > ninjas > dto > TS create-ninja.dto.ts > CreateNinjaDto > name
1 export class CreateNinjaDto {
2   name:string;
3 }

```

Similarly for update route

```

@Put('/:id')
updateNinja(@Param("id") id:string, @Body() updateNinjaDto: UpdateNinjaDto){

```

```

1 import { PartialType } from "@nestjs/mapped-types";
2 import { CreateNinjaDto } from "../create-ninja.dto";
3 export class UpdateNinjaDto extends PartialType(CreateNinjaDto){
4
5 }

```

```

@Put('/:id')
updateNinja(@Param("id") id:string, @Body() updateNinjaDto: UpdateNinjaDto){
  return {
    id,
    name:updateNinjaDto.name,
  };
}

```

```

4
5

```

Update ninja DTO that extends CreateNinjaDto

Providers and Dependency Injection:

Providers or Services are for the logic.

Providers are really just a class but they specifically have this Injectable decorator that means they can be injected into any class that depends on it.

```

src > ninjas > ninjas.service.ts > NinjasService
1 import { Injectable } from '@nestjs/common';
2
3 @Injectable()
4 export class NinjasService {}
5

```

In order to use the Provider/Service as it is a class we have to instantiate it and then use its methods.

But it would be great if this class is automatically instantiated and injected into our Ninjas Controller. (Dependency Injection)

```

// GET /ninjas?weapon=stars
@Get()
getNinjasHavingWeapon(@Query("weapon") weapon?: "Stars" | "Nunchucks"){
  const service = new NinjasService();
  return service.getNinjas(weapon);
}

```

Nest.js can do this for us in the following way.

```

3 @Injectable()
4 export class NinjasService {

```

This @injectable on service tells nest that nest is responsible for instantiating and injecting it in anything that depends on it.

```

@Controller('ninjas')
export class NinjasController {
  constructor(private readonly ninjasService: NinjasService) {}

  // GET /ninjas?weapon=fast --> []
  @Get()
  getNinjas(@Query('weapon') weapon: 'stars' | 'nunchucks') {
    //const service = new NinjasService();
    return this.ninjasService.getNinjas(weapon);
  }
}

```

We are also not instantiating the Controller class. Nest is instantiating it also behind the scenes.

We can inject multiple Services into a Controller. This is dependency injection.

```

const service = new NinjasService();
const controller = new NinjasController(service);

```

Exception Handling:

In controller:

```

// GET /ninjas/:id --> {...}
@Get('/:id')
getOneNinja(@Param("id") id:string){
  try{
    return this.ninjasService.findOne(id);
  }catch(err){
    throw new NotFoundException();
  }
}

```

In service:

```

findOne(id:string){
  const ninja = this.ninjas.find((ninja)=> id ===ninja.id);
  if(!ninja){
    throw new Error("ninja not found");
  }
  return ninja;
}

```

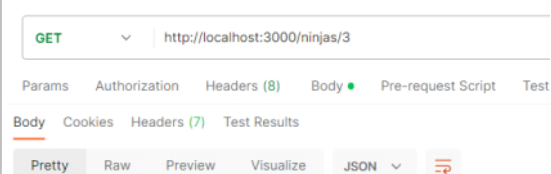
Nest has built in classes for exceptions that we can use:

Built-in HTTP exceptions

Nest provides a set of standard exceptions that inherit from the base `HttpException`. These are exposed from the `@nestjs/common` package, and represent many of the most common

HTTP exceptions:

- `BadRequestException`
- `UnauthorizedException`
- `NotFoundException`
- `ForbiddenException`
- `NotAcceptableException`



GET

http://localhost:3000/ninjas/3

Params

Authorization

Headers (8)

Body

Pre-request Script

Test

Body

Cookies

Headers (7)

Test Results

Pretty

Raw

Preview

Visualize

JSON

```

1  {
2    "statusCode": 404,
3    "message": "Not Found"
4  }

```

HTTP exceptions:

- BadRequestException
- UnauthorizedException
- NotFoundException
- ForbiddenException
- NotAcceptableException

If we want to send different exceptions and status codes depending upon the type of error.

All the built-in exceptions can also provide both an error `cause` and an error description using the `options` parameter:

```
throw new BadRequestException('Something bad happened', { cause: new Error() })
```

We can send our own message and description in the built in exception.

```

@Get(':id')
getOneNinja(@Param('id') id: string) {
  try {
    return this.ninjasService.getNinja(+id);
  } catch (err) {
    if (err instanceof DbException) {

```

We can also create custom exceptions:

In many cases, you will not need to write custom exceptions, and can use the built-in Nest HTTP exception, as described in the next section. If you do need to create customized exceptions, it's good practice to create your own **exceptions hierarchy**, where your custom exceptions inherit from the base `HttpException` class. With this approach, Nest will recognize your exceptions, and automatically take care of the error responses. Let's implement such a custom exception:

```

forbidden.exception.ts
JS

export class ForbiddenException extends HttpException {
  constructor() {
    super('Forbidden', HttpStatus.FORBIDDEN);
  }
}

```

Pipes:

Pipes

A pipe is a class annotated with the `@Injectable()` decorator, which implements the `PipeTransform` interface.



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 it through unchanged; otherwise, throw an exception

In both cases, pipes operate on the **arguments** being processed by a **controller route handler**. Nest interposes a pipe just before a method is invoked, and the pipe receives the arguments destined for the method and operates on them. Any transformation or validation operation takes place at that time, after which the route handler is invoked with any (potentially) transformed arguments.

```
@Get('/:id')
findOneNinja(@Param('id', ParseIntPipe) id: number) {
  try {
    return this.ninjasService.findOneNinja(id);
  } catch (err) {
    throw new NotFoundException();
  }
}
```

If we want to transform the id from string to number and then pass to our route handler we can make use of pipe in this way.

We can also create custom pipes.

For validation:

npm i --save class-validator class-transformer

```
npm i --save class-validator class-transformer
```

```
import { MinLength } from "class-validator";
export class CreateNinjaDto {
  @MinLength(3)
  name: string;
  weapon: string;
}
```

```
@Post()
createNinja(@Body(new ValidationPipe()) createNinjaDto: CreateNinjaDto) {
  return this.ninjasService.createNinja(createNinjaDto);
}
```

POST http://localhost:3000/ninjas

Params Authorization Headers (8) Body Pre-request Script Tests Set

none form-data x-www-form-urlencoded raw binary GraphQL

```
1 {
2   "name": "D",
3   "weapon": "Stars"
4 }
```

body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "statusCode": 400,
3   "message": [
4     "name must be longer than or equal to 3 characters"
5   ],
6   "error": "Bad Request"
7 }
```

```
import { IsEnum, MinLength } from 'class-validator';
export class CreateNinjaDto {
  @MinLength(3)
  name: string;

  @IsEnum(['stars', 'nunchucks'], { message: 'Use correct weapon!' })
  weapon: string;
}
```

class validator provides us set of decorators to validate the data.

Create your class and put some validation decorators on the properties you want to validate:

```
import {
  validate,
  validateOrReject,
  Contains,
  IsInt,
  Length,
  IsEmail,
  IsFQDN,
  IsDate,
  Min,
  Max,
} from 'class-validator';

export class Post {
  @Length(10, 20)
  title: string;

  @Contains('hello')
  text: string;

  @IsInt()
  @Min(0)
  @Max(10)
  rating: number;

  @IsEmail()
  email: string;

  @IsFQDN()
  site: string;

  @IsDate()
  createDate: Date;
}
```

POST http://localhost:3000/ninjas

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "name": "Dbafaj",
3   "weapon": "Stas"
4 }
```

body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "statusCode": 400,
3   "message": [
4     "Use correct weapon"
5   ],
6 }
```

```
import { IsEnum, MinLength } from 'class-validator';

export class CreateNinjaDto {
  @MinLength(3)
  name: string;

  @IsEnum(['stars', 'nunchucks'], { message: 'Use correct weapon!' })
  weapon: 'stars' | 'nunchucks';
}
```

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "statusCode": 400,
3   "message": [
4     "Use correct weapon"
5   ],
6   "error": "Bad Request"
7 }
```

Guards: For protecting the routes
Use cases are authorization, authentication.

Guards

A guard is a class annotated with the `@Injectable()` decorator, which implements the `CanActivate` interface.



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-time. This is often referred to as **authorization**. Authorization (and its cousin, **authentication**, with which it usually collaborates) has typically been handled by **middleware** in traditional Express applications. Middleware is a fine choice for authentication, since things like token validation and attaching properties to the `request` object are not strongly connected with a particular route context (and its metadata).

But middleware, by its nature, is dumb. It doesn't know which handler will be executed after calling the `next()` function. On the other hand, **Guards** have access to the `ExecutionContext` instance, and thus know exactly what's going to be executed next. They're designed, much like exception filters, pipes, and interceptors, to let you interpose processing logic at exactly the right point in the request/response cycle, and to do so declaratively. This helps keep your code DRY and declarative.

HINT

Guards are executed **after** all middleware, but **before** any interceptor or pipe.

If we want to provide access to specific routes to users only if they are authenticated, or are admins then we use guards.

We can attach guard to an entire controller or to an individual method of that controller. For example,

```
@Controller('ninjas')
@UseGuards(BeltGuard)
export class NinjasController {
  constructor(private readonly ninjasService: NinjasService) {}

  @Get()
  getNinjas(@Query('weapon') weapon: 'stars' | 'nunchucks') {
    return this.ninjasService.getNinjas(weapon);
  }
}
```

Or

```
// POST /ninjas
@Post()
@UseGuards(BeltGuard)
createNinja(@Body(new ValidationPipe()) createNinjaDto: CreateNinjaDto) {
  return this.ninjasService.createNinja(createNinjaDto);
}
```

Use command `nest g guard <guardName>`

```
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class BeltGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    return true;
  }
}
```

If the Boolean we are returning is false, we will get 403 status code, i.e, Forbidden resource.

So we just have to apply some logic here.

```
@Injectable()
export class BeltGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();

    // validate request
    //const hasBlackBelt = request.user.belts.includes('black');

    return hasBlackBelt;
  }
}
```