Table of Contents

[Reference 4](#_Toc146532016)

[Why we need Nest JS 4](#_Toc146532017)

[Installation using CLI 4](#_Toc146532018)

[Architecture 4](#_Toc146532019)

[Project Structure 6](#_Toc146532020)

[tsconfig.json: 6](#_Toc146532021)

[package.json/package.lock.json 6](#_Toc146532022)

[nest-cli.json 6](#_Toc146532023)

[.prettierrc 6](#_Toc146532024)

[.eslintrc 6](#_Toc146532025)

[Test folder 6](#_Toc146532026)

[src folder 6](#_Toc146532027)

[dist folder 6](#_Toc146532028)

[coverage folder 7](#_Toc146532029)

[src/main.ts 7](#_Toc146532030)

[src/app.module.ts 7](#_Toc146532031)

[Starting/Running the Project 7](#_Toc146532032)

[Running Project 7](#_Toc146532033)

[Test Application 7](#_Toc146532034)

[Start in Watch Mod 7](#_Toc146532035)

[Run Test Cases 7](#_Toc146532036)

[Run e2e (end to end) Test 7](#_Toc146532037)

[Generating Build 7](#_Toc146532038)

[Running Production Project 7](#_Toc146532039)

[Creating Controllers 7](#_Toc146532040)

[How to create controller 7](#_Toc146532041)

[Steps for implementing controller 8](#_Toc146532042)

[Creating Methods 8](#_Toc146532043)

[Handling Request 8](#_Toc146532044)

[Handling Response 8](#_Toc146532045)

[Setting Request Headers 9](#_Toc146532046)

[Redirection to Another Route with Status Code 9](#_Toc146532047)

[Dynamically redirect to Another Route with Status Code 10](#_Toc146532048)

[Accessing Rout and Query Parameter and Request Header 10](#_Toc146532049)

[Route Parameters 10](#_Toc146532050)

[Query Parameters 11](#_Toc146532051)

[Header Parameters 11](#_Toc146532052)

[Accessing Body Parameters 12](#_Toc146532053)

[Sub Domain Routing and Host and IP Details 12](#_Toc146532054)

[Restricting Route to specific domain 12](#_Toc146532055)

[Getting Dynamic Host name and domain 12](#_Toc146532056)

[Getting IP 13](#_Toc146532057)

[Dependency Injection and IOC Container 13](#_Toc146532058)

[Dependency Injection 13](#_Toc146532059)

[IOC Container 14](#_Toc146532060)

[Provider Types 14](#_Toc146532061)

[Injection Types 15](#_Toc146532062)

[Injectable Decorator and registering Providers 15](#_Toc146532063)

[Injectable Decorator 15](#_Toc146532064)

[Optional Inject Dependency 16](#_Toc146532065)

[Value Providers 16](#_Toc146532066)

[Injection Sope 17](#_Toc146532067)

[Services in Nest 18](#_Toc146532068)

[Modules in Nest 20](#_Toc146532069)

[Feature Module 20](#_Toc146532070)

[Shared Module 21](#_Toc146532071)

[Global Module 21](#_Toc146532072)

[Dynamic Module 21](#_Toc146532073)

[Router module in Nest 21](#_Toc146532074)

[Pipes 22](#_Toc146532075)

[Transformation Pipe 22](#_Toc146532076)

[ParseIntPipe 22](#_Toc146532077)

[ParseFloatPipe 23](#_Toc146532078)

[ParseBoolPipe 23](#_Toc146532079)

[new DefaultValuePipe 23](#_Toc146532080)

[Chaining Pipes 23](#_Toc146532081)

[Validation Pipes 23](#_Toc146532082)

[ParseUUIDPipe 23](#_Toc146532083)

[ParseArrayPipe 24](#_Toc146532084)

[Custom Pipes 24](#_Toc146532085)

[Data Validation (using Class formatter and transformer) 24](#_Toc146532086)

[Argument Host and ExecutionContext 24](#_Toc146532087)

[ApplicationHost 25](#_Toc146532088)

[ExecutionContext 25](#_Toc146532089)

[Types of Methods by Different Hosts 25](#_Toc146532090)

[Exception Filters | Concept & Built-in HTTP Exceptions | Throwing Exceptions 25](#_Toc146532091)

[Application Lifecycle | Lifecycle Phases, Events & Hooks 28](#_Toc146532092)

[Request Response Lifecycle | Request Flow & Order of Execution 28](#_Toc146532093)

[Middleware 29](#_Toc146532094)

[How to create and implement Functional Middleware 29](#_Toc146532095)

[Handling for Multiple routes 31](#_Toc146532096)

[Other possible ways 31](#_Toc146532097)

[Include and exclude some routes/request methods 32](#_Toc146532098)

[Apply more than one Middleware to one or more routes 32](#_Toc146532099)

[Make a global middleware in App module 32](#_Toc146532100)

[Making global with Request type 33](#_Toc146532101)

[Make global middleware in Main.ts 33](#_Toc146532102)

[Interceptors 34](#_Toc146532103)

[How to make an interceptor 34](#_Toc146532104)

[Maker global Interceptor 35](#_Toc146532105)

[Using ENV in Nest 36](#_Toc146532106)

[Getting ENV with default value 36](#_Toc146532107)

[Reading multiple files or is name of ENV file is different 37](#_Toc146532108)

[Caching the ENV file and expandvariable 37](#_Toc146532109)

[Alternate and Best way to us ENV 38](#_Toc146532110)

[HTTPS Server & Multiple Simultaneous Servers Setup | Listen on Multiple Ports 38](#_Toc146532111)

[Create HTTP and HTTPS server 39](#_Toc146532112)

[MongoDB with Nest 40](#_Toc146532113)

[Connecting to MongoDB 40](#_Toc146532114)

[Creating Scheme 40](#_Toc146532115)

[Adding References to Modal 42](#_Toc146532116)

[Creating Instances of Model Globally 43](#_Toc146532117)

**Nest JS**

# Reference

<https://youtube.com/playlist?list=PLqLR2H326bY6eRNOXJxWQkvKNlzmJQfLj&si=83cCvGNrGd6-XR8r>

# Why we need Nest JS

* Is a framework for building efficient, scalable and maintainable Node JS server-side application
* Provide application architecture, configurations and layer of abstraction above common Node JS framework with freedom of customization to create application
* Inspired by angular
* Supporting TS
* Give you well defined TS modular structure
* Uses express by default but you can use fastify and other also

# Installation using CLI

* Just install nest cli globally by command npm i -g @nestjs/cli
* Create new project by command nest new project-name
* When new project is created just install TS types as dev dependency by command npm i @types/express –save -dev

# Architecture

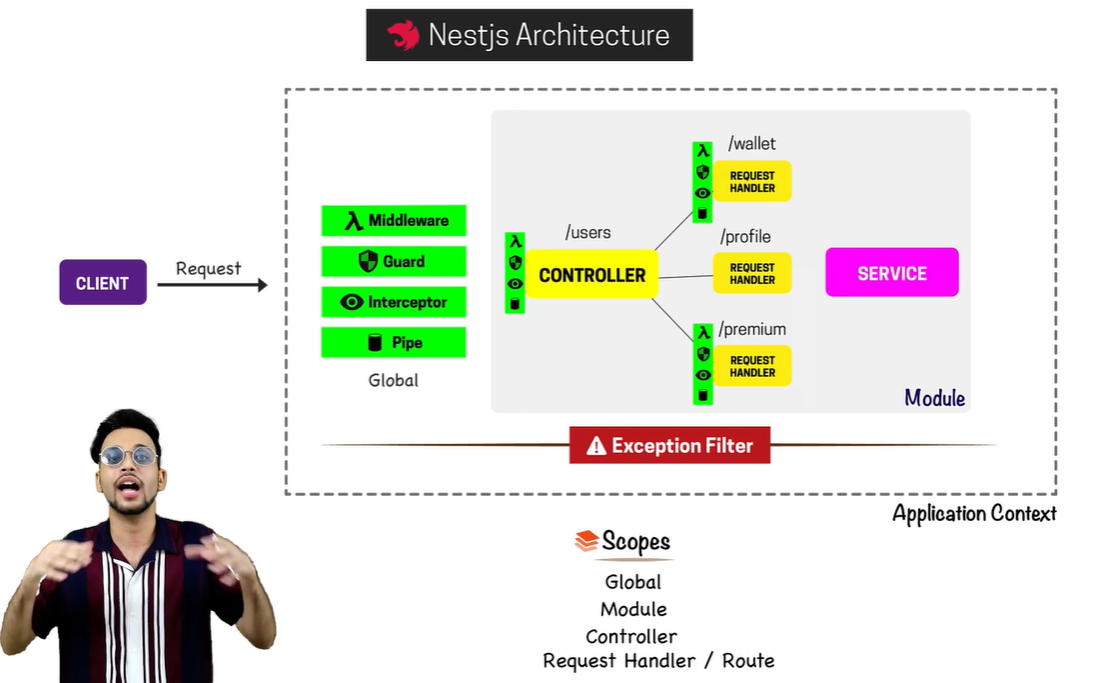
* There are 4 basic scopes

1. Global
2. Module
3. Controller
4. Request Handler/Route

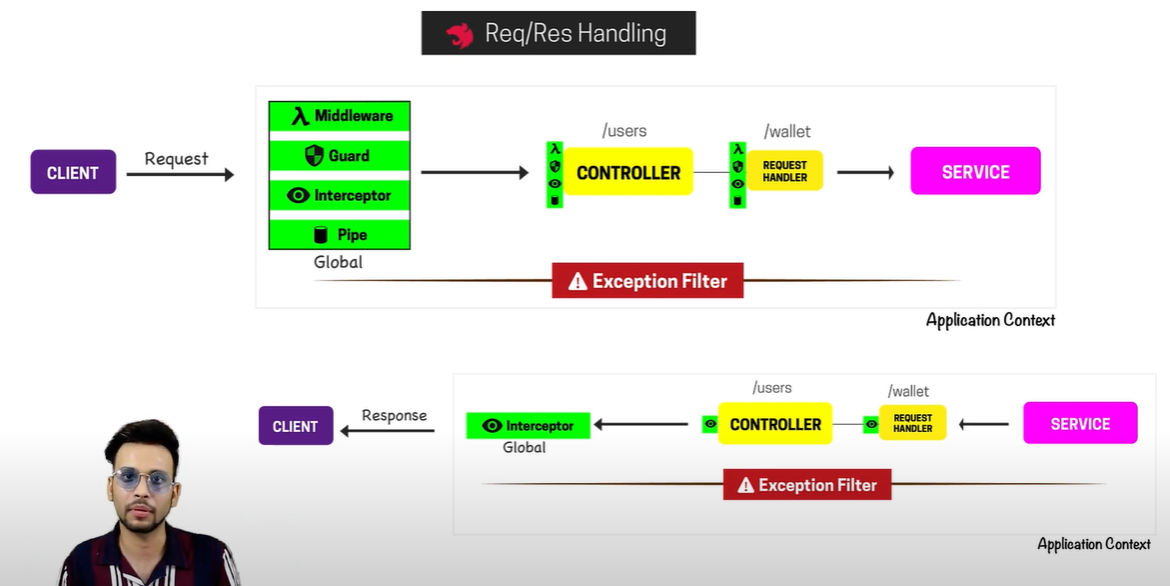
* There are 4 super powers that nest js provide us

1. **Middleware:** Execute before request move to controller (filter requests)
2. **Guard:** it tells whether to fulfil the request or not (mostly used in authentic)
3. **Interceptor:** same as middleware but it works on both times i-e on request and response time
4. **Pipe:** use to transform request

* The above 4 methods work from top to bottom in order
* You can also apply them to all 4 scopes of the project i-e global, module, controller, request handler
* We also have a very special layer called exception layer that handle any exception in the project

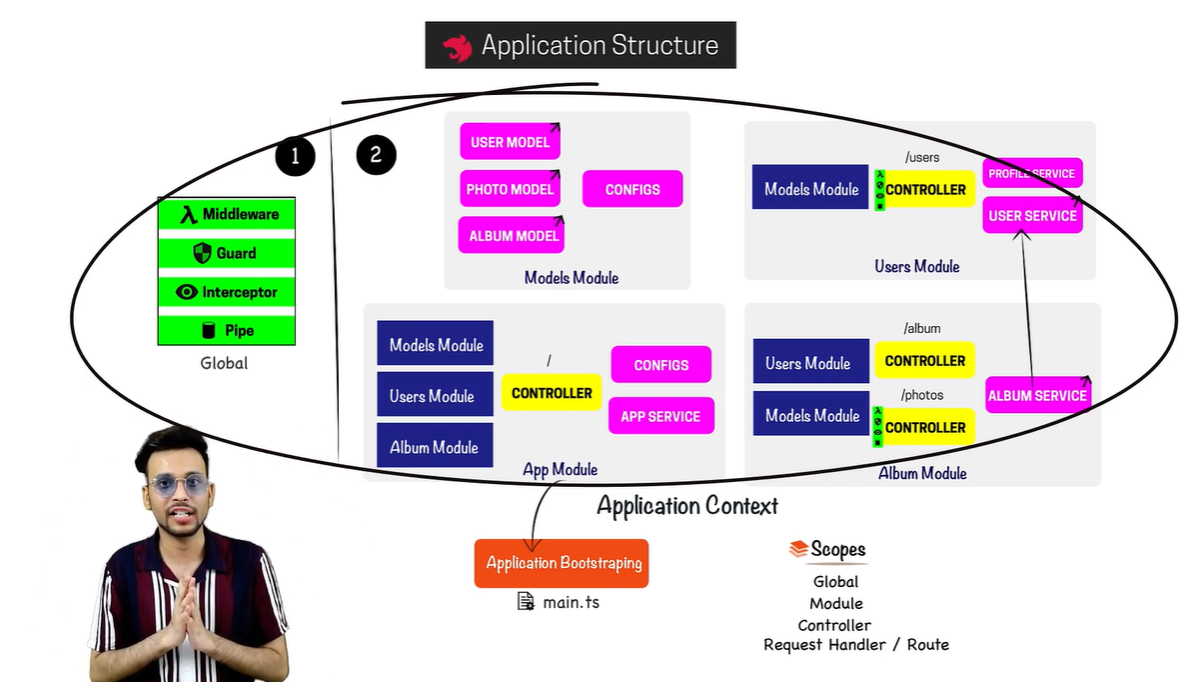


* Request and response is handling is shown below, 4 powers execute every time in request time but only interceptor executes every response time and whole project is protected by exception filter



* There are 2 types of modules in nest js

1. Module with Controller
2. Module without Controller
3. Every Module in nest js has 4 things
4. **Import**
5. **Controller**
6. **Provider**
7. **Exports**



# Project Structure

## tsconfig.json:

Handle TS build and configuration as in TS project

## package.json/package.lock.json

handle all dependencies and dev-dependencies and maintain hierarchy

## nest-cli.json

Setting regarding nest project

## .prettierrc

Use for project formatting

## .eslintrc

Use for eslint setting

## Test folder

Use to write e2e test cases

## src folder

main project folder where all controller and logic lies

## dist folder

The production code can be found here. As project in TS that’s why we get compiled JS code ready for production here

## coverage folder

Used in testing that tells how many part we have covered for testing

## src/main.ts

This is the main file which bootstrp the whole project

## src/app.module.ts

Is the main module file where we add other modules that used in project

# Starting/Running the Project

## Running Project

* Run command npm start and this will run the project on port:3000 by default

## Test Application

* While running the project run command curl <http://localhost:3000> on another terminal and it will return Hello World in the terminal that means the project has no errors

## Start in Watch Mod

* Run command npm run start:dev and project will be start running in the watch mode mean you server will restart and code will be compiled on ever file change

## Run Test Cases

* Run command npm run test and it will run all the test cases

## Run e2e (end to end) Test

* Run command npm run test:e2e and it will run all the e2e test cases

## Generating Build

* Run command npm run build and all the project is compiled into dist folder

## Running Production Project

* Run command npm run start:prod and it will run the production build that is ready to deploy

# Creating Controllers

## How to create controller

* After initializing the new nest code delete app.service.ts and app.controller.ts and also remove its presence from app.module.ts
* After that create and controller folder in src and for every controller make a same name directory there for example if you want to make a user controller than make user directory first in src/controllers/user and then create user.controller.ts file in it

## Steps for implementing controller

* Create class with Unique and descriptive name
* Import controller from nest and add the controller decorator above the class and also pass route name for that decorator
* Now export the controller and add it to the app.module.ts controller array to register that controller

## Creating Methods

* Now you can write different methods and you have to define request type for each method in the class and also you have to import that request type from the nest

import { Controller, Get } from '@nestjs/common';  
  
@Controller('user')  
export class UserController {  
 @Get('/')  
 getProfile() {  
 return 'Hello from Nest JS';  
 }  
}

user.controller.ts file

import { Module } from '@nestjs/common';  
import { UserController } from './controllers/user/user.controller';  
  
@Module({  
 imports: [],  
 controllers: [UserController],  
 providers: [],  
})  
export class AppModule {}

app.module.ts file

## Handling Request

* You can get request parameter from by importing and using the Request from nest and defining its type

import { Controller, Get, Request } from '@nestjs/common';  
  
@Controller('user')  
export class UserController {  
 @Get('/')  
 getProfile(@Request() req) {  
 console.log(req);  
 return 'Hello from Nest JS';  
 }  
}

## Handling Response

Alert: if you use response that nest will no handle response by itself using status code, and redirect etc.

* Import response from nest and use in the same way as request

import { Controller, Get, Request, Response } from '@nestjs/common';  
  
@Controller('user')  
export class UserController {  
 @Get('/')  
 getProfile(@Request() req, @Response() res) {  
 console.log(req);  
 // eslint-disable-next-line  
 return res.status(500).json({ status: 500, message: 'User controller working' });  
 }  
}

## Setting Request Headers

* You can also set the header by importing Header decorator and using it
* Well not working at my end

import { Controller, Get, Request, Response, Headers } from '@nestjs/common';  
  
@Controller('user')  
export class UserController {  
 @Get('/')  
 @Headers('Cache-Control', 'none')  
 @Headers('X-Name', 'Assassian')  
 getProfile(@Request() req, @Response() res) {  
 console.log(req);  
 return res.status(500).json({ status: 500, message: 'User controller working' });  
 }  
}

## Redirection to Another Route with Status Code

* By importing redirect from nest we can redirect our request
* You can also customize the redirect status code by passing status code after the redirect URL

import { Controller, Get, Request, Redirect } from '@nestjs/common';  
  
@Controller('user')  
export class UserController {  
 @Get('/')  
 @Redirect('/user/test', 302)  
 getProfile(@Request() req) {  
 // console.log(req);  
 // eslint-disable-next-line  
 return 'Book controller working';  
 }  
  
 @Get('/test')  
 runTest(@Request() req) {  
 // console.log(req);  
 // eslint-disable-next-line  
 return 'User redirect to test';  
 }  
}

## Dynamically redirect to Another Route with Status Code

* If you want to dynamically redirect to some route based on calculation etc. just return the status code and URL in the return parameter

import { Controller, Get, Request, Redirect } from '@nestjs/common';  
  
@Controller('user')  
export class UserController {  
 @Get('/')  
 @Redirect('/user/test', 302)  
 getProfile(@Request() req) {  
 const num = 343;  
  
 if (num % 2 == 0) {  
 return {  
 url: '/user/test1',  
 statusCode: 302,  
 message: 'Redirect to Route 1',  
 };  
 } else {  
 return {  
 url: '/user/test2',  
 statusCode: 302,  
 message: 'Redirect to Route 2',  
 };  
 }  
  
 // console.log(req);  
 // eslint-disable-next-line  
 return 'Book controller working';  
 }  
  
 @Get('/test1')  
 runTest1(@Request() req) {  
 // console.log(req);  
 // eslint-disable-next-line  
 return 'User redirect to test 1';  
 }  
  
 @Get('/test2')  
 runTest2(@Request() req) {  
 // console.log(req);  
 // eslint-disable-next-line  
 return 'User redirect to test 2';  
 }  
}

# Accessing Rout and Query Parameter and Request Header

## Route Parameters

* Import params from nest and use it to get the params while defining the params in the route

import { Controller, Post, Request, Response, Param, Get } from '@nestjs/common';  
  
@Controller('book')  
export class BookController {  
   
 @Get('/:id/:name')  
 test2(@Request() req, @Response() res, @Param() params) {  
 console.log(params.id);  
 console.log(params.name);  
 // eslint-disable-next-line  
 return res.status(500).json({ status: 500, message: 'Book Tes 2 route working successfully' });  
 }  
}

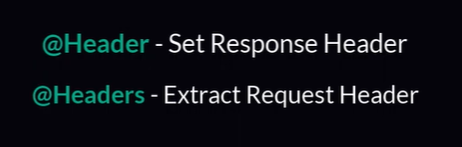
## Query Parameters

* Same like params import query parameter from the nest and use them like params

import { Controller, Post, Request, Response, Param, Get, Query } from '@nestjs/common';  
  
@Controller('book')  
export class BookController {  
 @Get('/test3')  
 test3(@Request() req, @Response() res, @Query() query) {  
 console.log(query.id);  
 console.log(query.name);  
 // eslint-disable-next-line  
 return res.status(500).json({ status: 500, message: 'Book Tes 3 route working successfully' });  
 }  
}

## Header Parameters

* There is difference between Header and Headers



* To access headers, we use Headers

import { Controller, Post, Request, Response, Param, Get, Query, Headers } from '@nestjs/common';  
  
@Controller('book')  
export class BookController {  
   
 @Get('/access-headers')  
 test4(@Request() req, @Response() res, @Headers() headers) {  
 console.log(headers);  
 // eslint-disable-next-line  
 return res.status(500).json({ status: 500, message: 'Book Tes 4 route working successfully' });  
 }  
}

# Accessing Body Parameters

* Import body from nest and use it as params to get body data

import { Controller, Get, Request, Redirect, Body } from '@nestjs/common';  
  
@Controller('user')  
export class UserController {  
 @Get('/')  
 getProfile(@Request() req, @Body() body) {  
 console.log(body.name);  
 console.log(body.age);  
 // eslint-disable-next-line  
 return 'Book controller working';  
 }  
  
}

# Sub Domain Routing and Host and IP Details

## Restricting Route to specific domain

* You can restrict the route to specific domain for that you have to pass path and host details as object in the class of controller

import { Controller, Get } from '@nestjs/common';  
  
@Controller({ path: '/specific', host: 'some.domain.com' })  
export class SpecificController {  
 @Get('/')  
 getProfile() {  
 return 'Specific Controller working';  
 }  
}

## Getting Dynamic Host name and domain

* You can also get host and domain by importing HostParam from nest and you have to place the “:” before host and before domain

import { Controller, Get, HostParam } from '@nestjs/common';  
  
@Controller({ path: '/specific', host: ':some:domain.com' })  
export class SpecificController {  
 @Get('/')  
 getProfile(@HostParam() hostParams) {  
 console.log(hostParams);  
 return 'Specific Controller working';  
 }  
}

## Getting IP

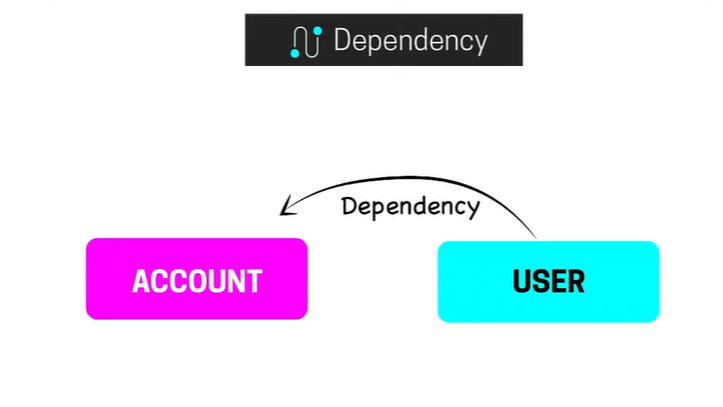
* Import Ip decorator from the nest and it will give you the ip as follow

import { Controller, Get, HostParam, Ip } from '@nestjs/common';  
  
@Controller('specific')  
export class SpecificController {  
 @Get('/')  
 getProfile(@HostParam() hostParams, @Ip() ip) {  
 console.log(hostParams);  
 console.log(ip);  
 return 'Specific Controller working';  
 }  
}

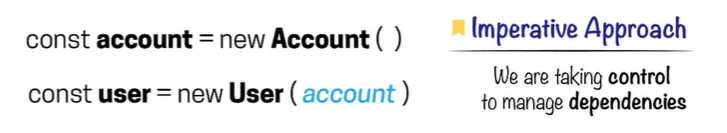
# Dependency Injection and IOC Container

## Dependency Injection

* Create and manage dependency and its hierarchy
* There are different classes in the project, when one class need other class to do some work than other class work as dependency for the using class



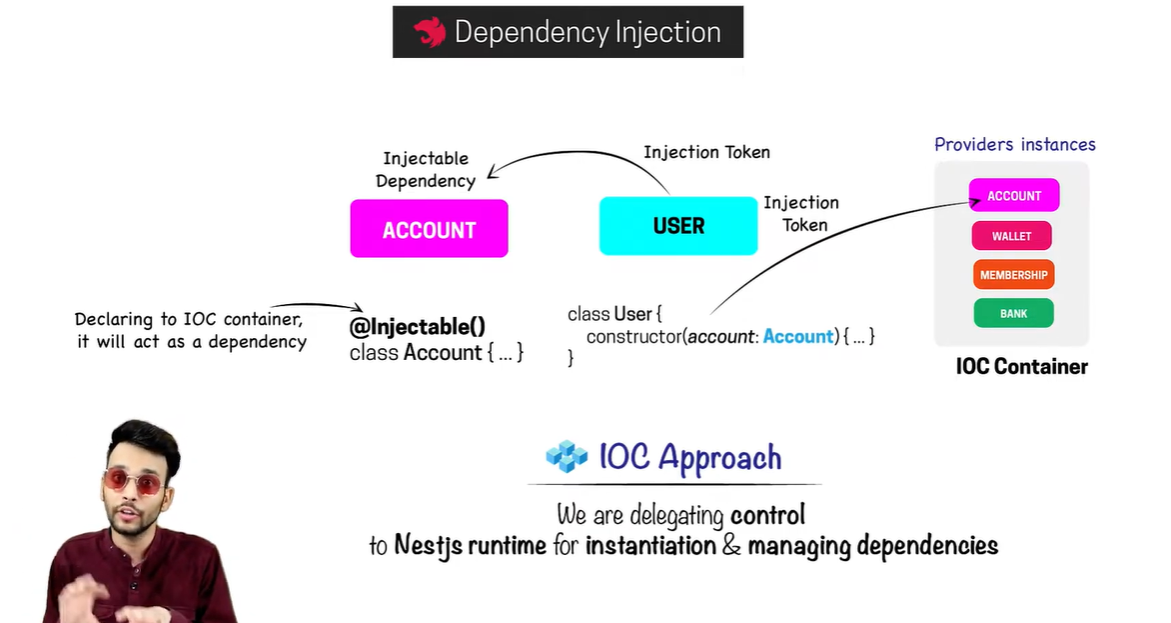
* In the case we don’t have dependency injection than we use imperative approach, we create instance of other class and pass the instance to the require dependency class



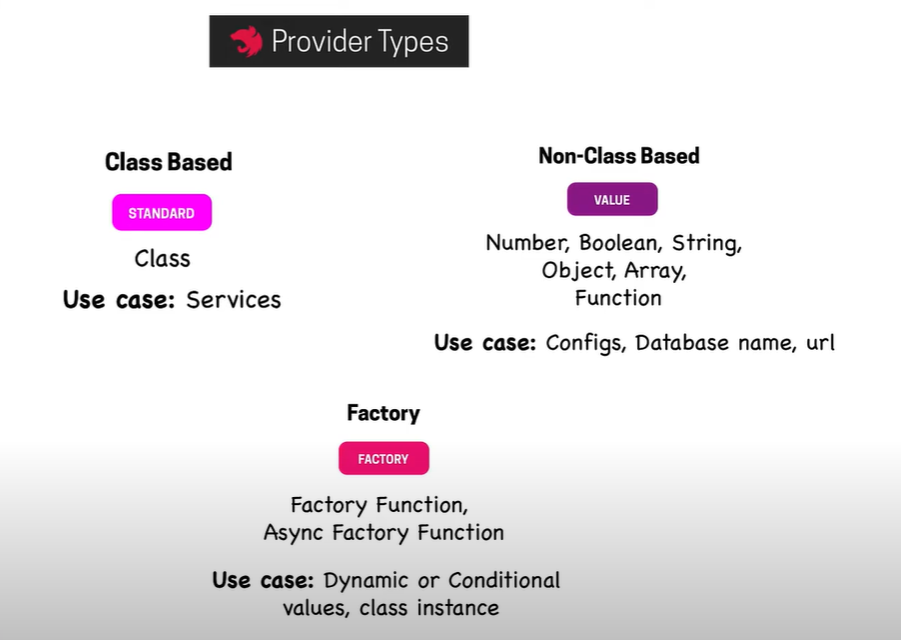
* The dependency of different classes is different and we cannot handle them using imperative approach
* Dependency Injection is built on IOC (Inversion of control) in which we gave control to manage our program to the framework

## IOC Container

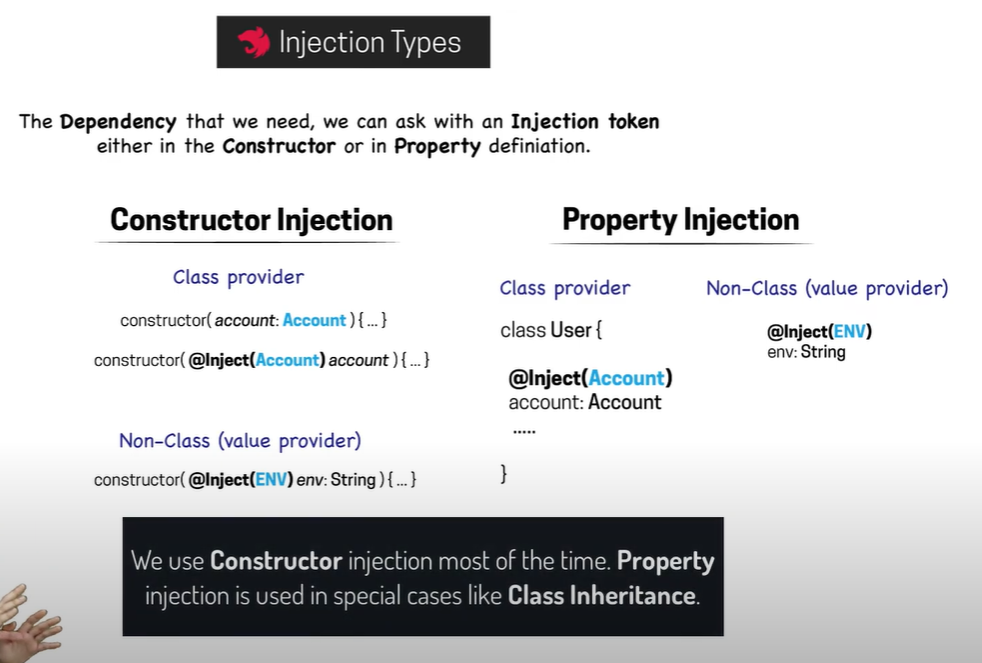
* Is a store where we place dependency instances and classes instances that can be treated as dependency
* The class that can be used as dependency we call them provider and that instance is passed in IOC container



# Provider Types



# Injection Types



# Injectable Decorator and registering Providers

## Injectable Decorator

* Create a class with function that we want to use in other class as dependency in my case I am creating “userstore” that is used by “user controller”
* Now to make a class ready for dependency we have to add Import and add injectable decorator at the top of the class

import { Controller, Get, Injectable } from '@nestjs/common';  
  
@Injectable()  
@Controller('userstore')  
export class UserStoreController {  
 @Get('/')  
 getUserStore() {  
 return 'User Store Controller working';  
 }  
}

* The instance of class cannot be made until I register it into some module, that’s why we register them in the provider array of app.modules

import { Module } from '@nestjs/common';  
import { UserController } from './controllers/user.controller';  
import { BookController } from './controllers/book.controller';  
import { SpecificController } from './controllers/specific.controller';  
import { UserStoreController } from './controllers/userstore.controller';  
  
@Module({  
 imports: [],  
 controllers: [UserController, BookController, SpecificController],  
 providers: [UserStoreController],  
})  
export class AppModule {}

* Now to use UserStore in User we have to inject in in the constructure

import {  
 Controller,  
 Inject,  
} from '@nestjs/common';  
import { UserStoreController } from './userstore.controller';  
  
@Controller('user')  
export class UserController {  
 constructor(@Inject(UserStoreController) private store: any) {  
 console.log(store);  
 }  
}

## Optional Inject Dependency

* When you don’t know if the class is used or not as dependency than we use optional dependency instead of Inject and can add or not the Injectable to provider

import {  
 Controller,  
 Get,  
 Request,  
 Redirect,  
 Body,  
 Inject, Optional,  
} from '@nestjs/common';  
import { UserStoreController } from './userstore.controller';  
  
@Controller('user')  
export class UserController {  
 constructor(@Optional() private store: any) {  
 console.log(store);  
 }  
}

## Value Providers

* These can be used setting the value as provider and getting them in the class as we do it injectables

import {  
 Controller,  
 Get,  
 Inject,  
} from '@nestjs/common';  
import { UserStoreController } from './userstore.controller';  
  
@Controller('user')  
export class UserController {  
 // constructor(@Inject(UserStoreController) private store: any) {  
 // console.log(store);  
 // }  
  
 constructor(@Inject('DB\_NAME') private dbname: any) {  
 console.log(dbname);  
 }

}

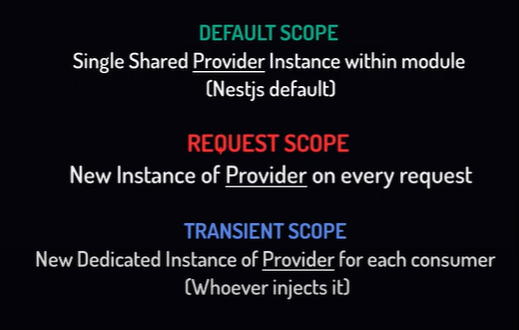
User controller

import { Module } from '@nestjs/common';  
import { UserController } from './controllers/user.controller';  
import { BookController } from './controllers/book.controller';  
import { SpecificController } from './controllers/specific.controller';  
import { UserStoreController } from './controllers/userstore.controller';  
  
@Module({  
 imports: [],  
 controllers: [UserController, BookController, SpecificController],  
 providers: [UserStoreController, { provide: 'DB\_NAME', useValue: 'Hitman' }],  
})  
export class AppModule {}

app.module

## Injection Sope

There are 3 scopes for the injection



import {Controller, Get, Injectable, Scope} from '@nestjs/common';  
  
@Injectable({ scope: Scope.DEFAULT })  
// @Injectable({ scope: Scope.REQUEST })  
// @Injectable({ scope: Scope.TRANSIENT })  
@Controller('userstore')  
export class UserStoreController {  
 @Get('/')  
 getUserStore() {  
 return 'User Store Controller working';  
 }  
}

# Services in Nest

* We can handle configuration and application-level logic using services
* We will make author crud in controller and register all the implementation in the services of author
* For registering all implementation in the services, we have to create an author service file and make it as dependable by Injection decorator and also register it in app.modules provider
* Make instance of service in user controller and add respective functions to respective routes all done.

import {Body, Controller, Delete, Get, Inject, Post, Put, Request, Response} from '@nestjs/common';  
import {AuthorService} from "../services/author.service";  
@Controller('author')  
export class AuthorController {  
  
 constructor(@Inject(AuthorService) private authorService:AuthorService) {  
 }  
  
 @Post()  
 create(@Request() req, @Response() res) {  
 this.authorService.addUser("Hitman")  
 return res.status(201).json({ status: 201, message: 'User created' });  
 }  
  
 @Get()  
 all(@Request() req, @Response() res) {  
 this.authorService.allUsers()  
 return res.status(201).json({ status: 200, message: 'All Data fetched' });  
 }  
  
 @Delete()  
 delete(@Request() req, @Response() res, @Body() body) {  
 this.authorService.deleteUser(1)  
 return res.status(201).json({ status: 200, message: 'User Deleted', passed\_id: body.id });  
 }  
  
 @Put()  
 update(@Request() req, @Response() res, @Body() body) {  
 this.authorService.updateUser(1)  
 return res.status(201).json({ status: 200, message: 'User Updated', passed\_id: body.id });  
 }  
  
  
}

author.controller.ts

import {Injectable} from "@nestjs/common";  
  
  
@Injectable()  
export class AuthorService {  
  
 private store: string[];  
  
 addUser(data){  
 // save the user against data  
 }  
  
 deleteUser(id)  
 {  
 // delete user based on id  
 }  
  
 updateUser(id)  
 {  
 // update user based on id  
 }  
  
 allUsers()  
 {  
 // return all users  
 }  
   
}

author.service.ts

import { Module } from '@nestjs/common';  
import { UserController } from './controllers/user.controller';  
import { BookController } from './controllers/book.controller';  
import { SpecificController } from './controllers/specific.controller';  
import { UserStoreController } from './controllers/userstore.controller';  
import {AuthorService} from "./services/author.service";  
  
@Module({  
 imports: [],  
 controllers: [UserController, BookController, SpecificController],  
 providers: [UserStoreController, AuthorService],  
})  
export class AppModule {}

app.module.ts

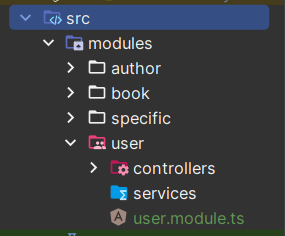
# Modules in Nest

* Primary goal is to organize our application
* 4 types of modules

1. Feature module (Made for specific feature like for user feature)
2. Shared module
3. Global module
4. Dynamic module

## Feature Module

* First create a folder in src folder with name “modules”
* For every feature like user, admin etc. we first have to create folder in modules with the name of module and then we have to create controller, service etc folder there and a moduleName.module.ts file in the root module directory as shown in pic



* Now if you want to create controller or services then make in their respective folders
* Every feature has its own module file where you have to register their imports, exports, controllers and providers,
* After registering all stuff in their feature module you have to add that module in the import of app.module to register all the things

import {Module} from "@nestjs/common";  
import {AuthorController} from "./controllers/author.controller";  
import {AuthorService} from "./services/author.service";  
  
@Module({  
 imports: [],  
 controllers: [AuthorController],  
 providers: [AuthorService],  
 exports: [AuthorService]  
})  
export class AuthorModule {}

author.module.ts

## Shared Module

* Suppose you want to use user module in job and employee module than you have to import the user module in the import of job and employee module file

## Global Module

* To make a module global that every module access it without importing it than you just import and add a global decorator above the module decorator of the feature module
* But you still need to import it once in the app.module (root module of app)

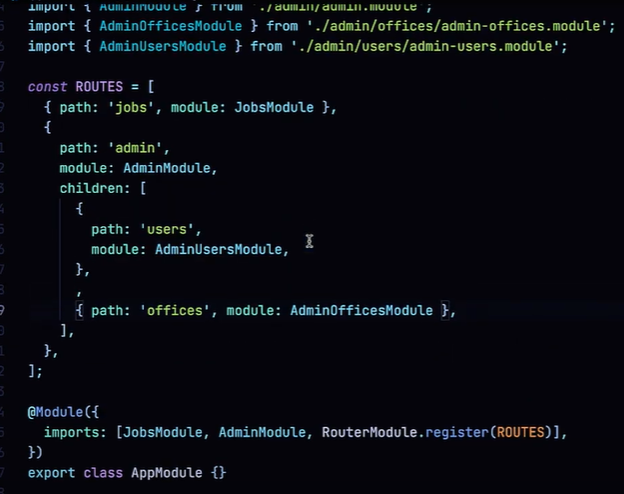
import {Global, Module} from "@nestjs/common";  
import {UserController} from "./controllers/user.controller";  
import {UserStoreController} from "./controllers/userstore.controller";  
  
@Global()  
@Module({  
 imports: [],  
 controllers: [UserController, UserStoreController],  
 providers: [UserStoreController, { provide: 'DB\_NAME', useValue: 'Hitman' }],  
 exports: [UserStoreController]  
})  
export class UserModule {}

## Dynamic Module

* Samaj nahi a rahi ☹

# Router module in Nest

* Use to add a prefix to all routes
* But not a very good approach
* We cannot register routes in feature modules. We can only configure it in root module only
* Route prefixing can turn into complex hierarchy which lead to unmaintainable routing structure over time



# Pipes

* Use to convert data in query parameter or params according to our requirement and apply validations to the data (data transformation)

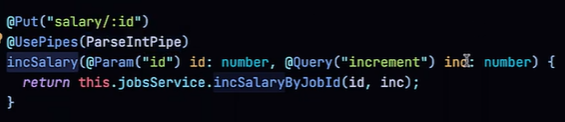
## Transformation Pipe

### ParseIntPipe

* Use to convert query/params data into integer form



For single argument



For whole route

### ParseFloatPipe

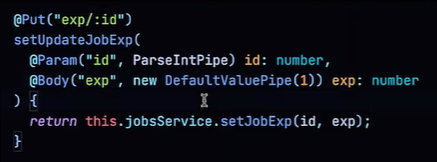
* Use to convert query/params data into float form

### ParseBoolPipe

* Use to convert query/params data into boolean form

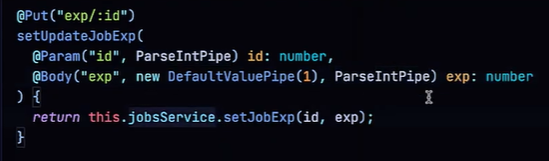
### new DefaultValuePipe

* You can set query/params default value if no value is passed
* Always called with new operator



### Chaining Pipes

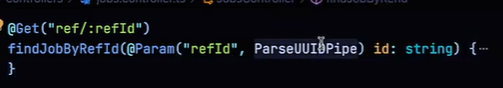
* You can chain multiple pipes also like this



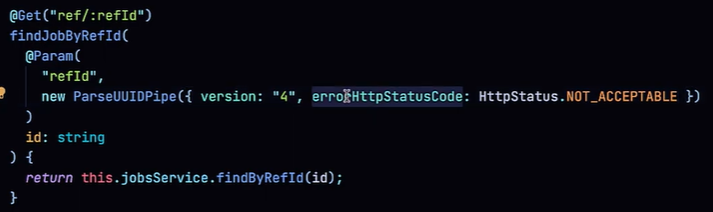
## Validation Pipes

### ParseUUIDPipe

* Validate query/params v3, v4 and v5 UUID
* You can only specify the version with new keyword and by passing parameters



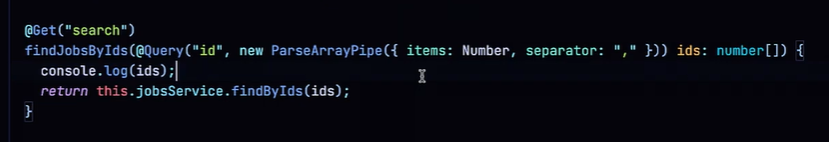
Without specific version



With specific version and error code

### ParseArrayPipe

* Use to validate array datatype



# Custom Pipes

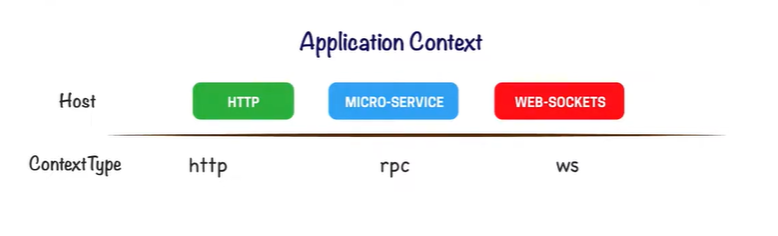
* Can be studied from the videos

# Data Validation (using Class formatter and transformer)

* Ye kam JOI se bi hoo jata ha

# Argument Host and ExecutionContext

* We have application context in Nest JS
* Depends upon which type of Application we are building on it
* We mostly build http host type application
* Different type of Host has different type of Context type



## ApplicationHost

* To work with different host, we have a class in Nest JS called ArgumentHost
* It tells us about application context and allow us to switch from one host to other
* Also, we get argument and argumentById from it

## ExecutionContext

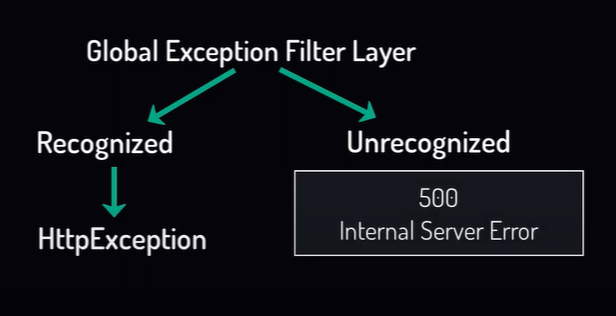
* Extends argumentHost
* Also has 2 more methods getClass (provide type of class) and getHandler (provide the request to)

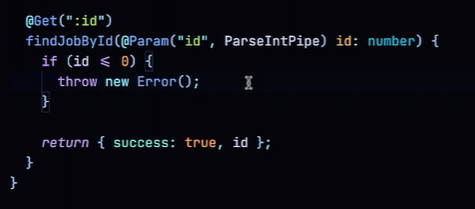
## Types of Methods by Different Hosts



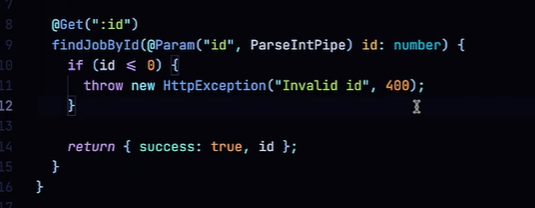
# Exception Filters | Concept & Built-in HTTP Exceptions | Throwing Exceptions

* You can throw exceptions with custom status and error code
* All the exceptions are check if they are valid or not, valid exceptions have there own status code while is any exception is not valid than by default “Internal Server Error” with status code 500 is sent to client

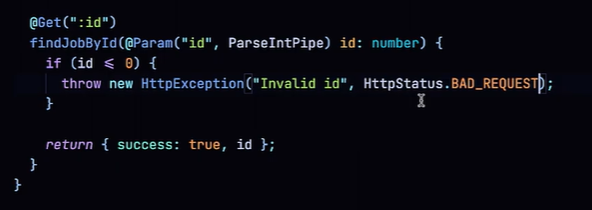




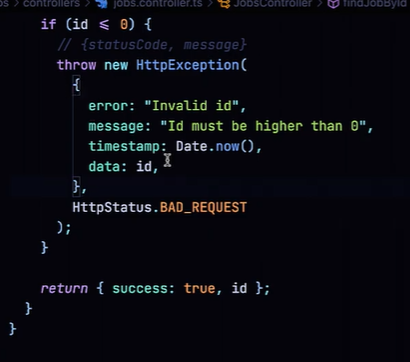
Unrecognized exception



recognized exception



recognized with enum exception



recognized with enum exception and object

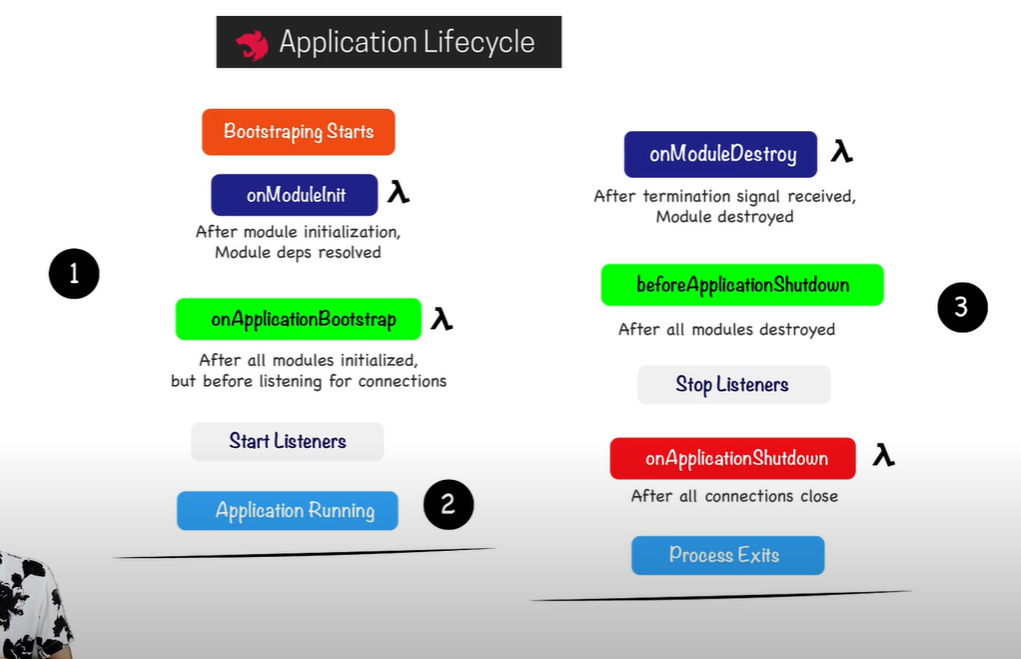


Exception with predefined exception with message and description

# Application Lifecycle | Lifecycle Phases, Events & Hooks

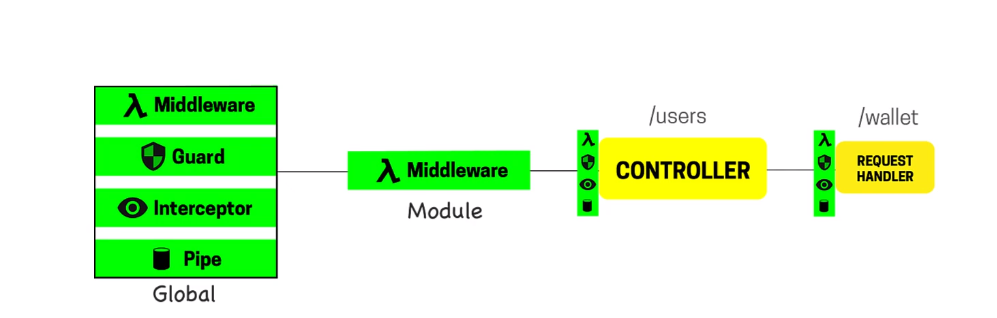
There are 3 phases of application

1. Initializing (Bootstraping)
2. Running
3. Terminating

you can use these methods in your project

# Request Response Lifecycle | Request Flow & Order of Execution

* You can apply 4 powers of nest to global, controller and request handle level but can only apply middleware to module level only



* We can also apply exception to all except module level



* Here is the flow of request and response



# Middleware

* Middleware is applied on module of a feature. It module we define which middleware runs for which route

## How to create and implement Functional Middleware

* First you need to import NestModule and then implement the module class with NestModule
* Then create a middleware function with req, res and next parameter and write the login
* Make constructor in the implemented class in module and apply the middleware to the route name
* Now middleware work for all specified routes

import {Request, Response, NextFunction} from 'express'  
import {BadRequestException} from "@nestjs/common";  
export function checkRequestPlatform(req:Request, res:Response, next:NextFunction) {  
 req["resquestPlatform"]=req.headers["user-agent"];  
 console.log(req.headers["user-agent"])  
 next();  
  
 // you can also return response directly from here  
 // for that you havr to remove next()  
 //return res.status(404).json({ status: 404, message: "Not a User" })  
  
 // OR  
  
 // throw new Error()  
  
 // OR  
  
 //throw new BadRequestException("Not a user","User is not vlaid")  
}

middleware function

import {Global, MiddlewareConsumer, Module, NestModule} from "@nestjs/common";  
import {UserController} from "./controllers/user.controller";  
import {UserStoreController} from "./controllers/userstore.controller";  
// @ts-ignore  
import {checkRequestPlatform} from "../../middlewares/check\_request\_device.middleware";  
  
@Global()  
@Module({  
 imports: [],  
 controllers: [UserController, UserStoreController],  
 providers: [UserStoreController, { provide: 'DB\_NAME', useValue: 'Hitman' }],  
 exports: [UserStoreController]  
})  
export class UserModule implements NestModule {  
 configure(consumer: MiddlewareConsumer): any {  
 consumer.apply(checkRequestPlatform).forRoutes("user")  
 }  
}

user.module.y

import {  
 Controller,  
 Get,  
 Request,  
} from '@nestjs/common';  
import { UserStoreController } from './userstore.controller';  
  
@Controller('user')  
export class UserController {  
 // constructor(@Inject(UserStoreController) private store: any) {  
 // console.log(store);  
 // }  
  
 constructor(@Inject('DB\_NAME') private dbname: any) {  
 console.log(dbname);  
 }  
  
 @Get('/')  
 getProfile(@Request() req, @Body() body) {  
 console.log(req.resquestPlatform)  
 // eslint-disable-next-line  
 return 'Book controllers working';  
 }  
  
}

user.controller.ts files

### Handling for Multiple routes

* Just add more routes after the comma

export class UserModule implements NestModule {  
 configure(consumer: MiddlewareConsumer): any {  
 consumer.apply(checkRequestPlatform).forRoutes("user","book")  
 }  
}

### Other possible ways

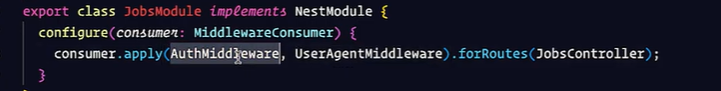
import {Global, MiddlewareConsumer, Module, NestModule, RequestMethod} from "@nestjs/common";  
import {UserController} from "./controllers/user.controller";  
import {UserStoreController} from "./controllers/userstore.controller";  
// @ts-ignore  
import {checkRequestPlatform} from "../../middlewares/check\_request\_device.middleware";  
import {SpecificController} from "../specific/controllers/specific.controller";  
import {AuthorController} from "../author/controllers/author.controller";  
  
@Global()  
@Module({  
 imports: [],  
 controllers: [UserController, UserStoreController],  
 providers: [UserStoreController, { provide: 'DB\_NAME', useValue: 'Hitman' }],  
 exports: [UserStoreController]  
})  
export class UserModule implements NestModule {  
 configure(consumer: MiddlewareConsumer): any {  
 consumer.apply(checkRequestPlatform).forRoutes("user","book")  
 // runs for test only  
 consumer.apply(checkRequestPlatform).forRoutes("user/test")  
 // runs for user and sub route with wildcard  
 consumer.apply(checkRequestPlatform).forRoutes("user/t\*st")  
 // runs if "s" is present or not at the end  
 consumer.apply(checkRequestPlatform).forRoutes("user/tests?")  
 // runs for the whole controller  
 consumer.apply(checkRequestPlatform).forRoutes(UserController)  
 // runs for specific request method  
 consumer.apply(checkRequestPlatform).forRoutes({  
 path: "user/",  
 method: RequestMethod.GET  
 })  
 }  
}

### Include and exclude some routes/request methods

import {Global, MiddlewareConsumer, Module, NestModule, RequestMethod} from "@nestjs/common";  
import {UserController} from "./controllers/user.controller";  
import {UserStoreController} from "./controllers/userstore.controller";  
// @ts-ignore  
import {checkRequestPlatform} from "../../middlewares/check\_request\_device.middleware";  
import {SpecificController} from "../specific/controllers/specific.controller";  
import {AuthorController} from "../author/controllers/author.controller";  
  
@Global()  
@Module({  
 imports: [],  
 controllers: [UserController, UserStoreController],  
 providers: [UserStoreController, { provide: 'DB\_NAME', useValue: 'Hitman' }],  
 exports: [UserStoreController]  
})  
export class UserModule implements NestModule {  
 configure(consumer: MiddlewareConsumer): any {  
 // exclude some routes  
 // @ts-ignore  
 consumer.apply({path: "user/", method: RequestMethod.GET},SpecificController).exclude(AuthorController, {path: "book", method: RequestMethod.POST})  
 }  
}

## Apply more than one Middleware to one or more routes

* Just create middleware and add as many as you want to apply by separating them with comma



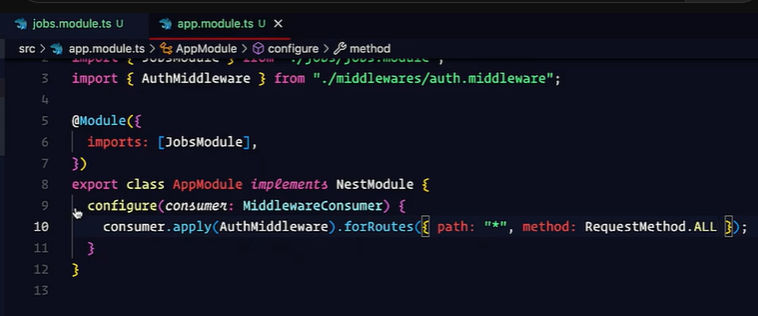
## Make a global middleware in App module

* Remove the implementation from the feature module class and goto app.module and implement the class with NestModule and define contrictor and other things there and apply the middleware to all route by passing \*



### Making global with Request type

* Just add request type there



### Make global middleware in Main.ts

* Simple make a functional component and use that middleware in main.ts as we do it express



# Interceptors

* All the features like middleware, guard, pipe and interceptor will run for the request but the interceptor is only which runs at the response time also
* Interceptors are based on AOP (Aspect Oriented Programming) like if we apply interceptor to any request than we can transform that request before input and after output
* If we got any error in function than using interceptor, we can override that exception and also can add additional behavior to that method

## How to make an interceptor

* Make a file for interceptor that implements NestInterceptor and use intercept constructor and write login in it

import {CallHandler, ExecutionContext, Injectable, NestInterceptor} from "@nestjs/common";  
import {Observable, tap} from "rxjs";  
  
Injectable()  
export class userInterceptor implements NestInterceptor{  
  
 intercept(context: ExecutionContext, next: CallHandler<any>): Observable<any> | Promise<Observable<any>> {  
  
 return next.handle().pipe(tap((result)=>{  
 console.log(result)  
 }))  
  
 }  
}

* Now got to respective controller and either decorate the whole controller or particular request with the @UseInterceptor decorator and in roud brackets pass the interceptor

import {  
 Controller,  
 Get,  
 Request,  
 UseInterceptors,  
} from '@nestjs/common';  
import {userInterceptor} from "../interceptors/user.interceptors";  
  
@Controller('user')  
@UseInterceptors(userInterceptor)  
export class UserController {  
  
 constructor(@Inject('DB\_NAME') private dbname: any) {  
 console.log(dbname);  
 }  
  
 @Get('/')  
 getProfile(@Request() req, @Body() body) {  
 console.log(req.resquestPlatform)  
 // eslint-disable-next-line  
 return 'Book controllers working';  
 }  
}

## Maker global Interceptor

* Make a interceptor as previous

import {CallHandler, ExecutionContext, Injectable, NestInterceptor} from "@nestjs/common";  
import {Observable, tap} from "rxjs";  
  
Injectable()  
export class loggingInterceptor implements NestInterceptor{  
  
 intercept(context: ExecutionContext, next: CallHandler<any>): Observable<any> | Promise<Observable<any>> {  
  
 let ctx = context.switchToHttp();  
 let req = ctx.getRequest()  
 let res = ctx.getResponse()  
  
  
 let startTime=Date.now()  
  
 return next.handle().pipe(tap((result)=>{  
 let endTime=Date.now();  
  
 let reqTime=endTime-startTime;  
  
 console.log(""+req.method+" "+req.path+" "+res.statusCode+" "+reqTime)  
 }))  
  
 }  
}

* Now in app module/feature module add the this as provider and specify the class name

import {Global, MiddlewareConsumer, Module, NestModule, RequestMethod} from "@nestjs/common";  
import {UserController} from "./controllers/user.controller";  
import {APP\_INTERCEPTOR} from "@nestjs/core";  
import {loggingInterceptor} from "./interceptors/user.interceptors";  
  
@Global()  
@Module({  
 imports: [],  
 controllers: [UserController, UserStoreController],  
 providers: [UserStoreController, { provide: 'DB\_NAME', useValue: 'Hitman' }, {provide: ***APP\_INTERCEPTOR***, useClass: loggingInterceptor}],  
 exports: [UserStoreController]  
})  
export class UserModule {}

# Using ENV in Nest

* Will be created in the root directory
* Comment is written by placing # and then the comment
* To use this file in the application we need to install npm i @nestjs/config --save
* To configure this file goto app.module.ts and import ConfigModule from the package and load it in the import of the module or in the specific module file where you want to use the env
* Data from env is returned in string format you have to convert it by yourself

import {Module} from "@nestjs/common";  
import {SpecificController} from "./controllers/specific.controller";  
import {SpecificService} from "./services/specific.service";  
import {ConfigModule} from "@nestjs/config";  
  
@Module({  
 imports: [ConfigModule.forRoot()],  
 controllers: [SpecificController],  
 providers: [SpecificService],  
 exports: [SpecificService]  
})  
export class SpecificModule {}

specific.module.ts

import {ConfigService} from "@nestjs/config";  
import {Injectable} from "@nestjs/common";  
  
@Injectable()  
export class SpecificService {  
 constructor(private configService: ConfigService) {  
 console.log(configService.get("PORT"))  
 console.log(configService.get("GMAIL"))  
 }  
}

specific.service.ts

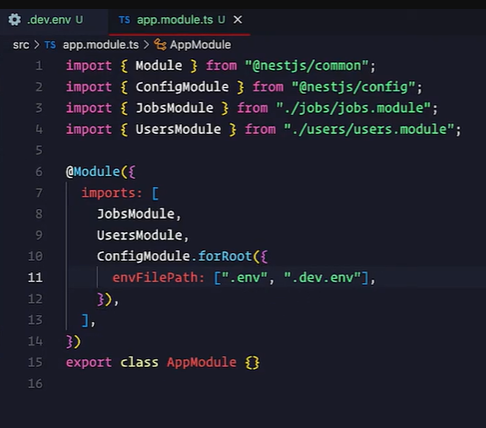
## Getting ENV with default value

* You can set to return a default value is value is not find in env by placing comma and specific the default value in the get function

import {ConfigService} from "@nestjs/config";  
import {Injectable} from "@nestjs/common";  
  
@Injectable()  
export class SpecificService {  
 constructor(private configService: ConfigService) {  
 console.log(configService.get("PORT"))  
 console.log(configService.get("SSL","INVALID\_SSL"))  
 }  
}

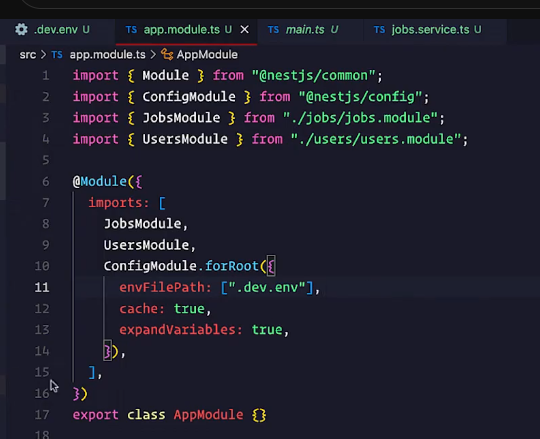
## Reading multiple files or is name of ENV file is different

* If you env filename is different or you have multiple env file to read you can pass them as below



## Caching the ENV file and expandvariable

* Just make the cache property true and your processing time is reduced greatly
* expandVariable is use to attach the env value with another env variable



Different vaiable name

HOST=wanologicalcolutions.com  
GMAIL=hamza${HOST}

expandVariable

## Alternate and Best way to us ENV

* Just import configmodule in app.module.ts file and just call process.env.ENV\_NAME to get the value

import { Module } from '@nestjs/common';  
import {UserModule} from "./modules/user/user.module";  
import {BookModule} from "./modules/book/book.module";  
import {AuthorModule} from "./modules/author/author.module";  
import {SpecificController} from "./modules/specific/controllers/specific.controller";  
import {SpecificModule} from "./modules/specific/specific.module";  
import {ConfigModule} from "@nestjs/config";  
  
@Module({  
 imports: [UserModule, BookModule, AuthorModule, SpecificModule,ConfigModule.forRoot({envFilePath: [".env",".prod.env"], cache: true, expandVariables: true, isGlobal: true})],  
 controllers: [],  
 providers: [],  
})  
export class AppModule {}

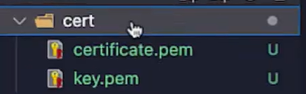
app.module.ts

import {ConfigService} from "@nestjs/config";  
import {Injectable} from "@nestjs/common";  
import \* as process from "process";  
  
@Injectable()  
export class SpecificService {  
 constructor(private configService: ConfigService) {  
 console.log(process.env.PORT)  
 console.log(process.env.GMAIL)  
 }  
}

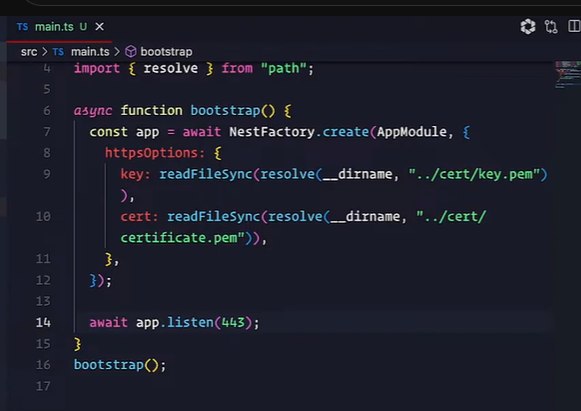
specific.service.ts

# HTTPS Server & Multiple Simultaneous Servers Setup | Listen on Multiple Ports

* Generate certificated file using <https://github.com/officialcomputerbaba/amazing-nestjs/blob/main/Fundamentals/lecture-38/cert/README.md>



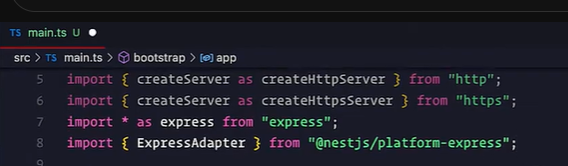
* Now goto main.ts file and in NestFactory add 2nd parameter add httpOptions with key and cert and by importing readFileSync and resolve give the file of files



* Now your ssl is applied and server run on https server

# Create HTTP and HTTPS server

* Import http and https servers
* Create server from express
* Initialize app
* Provide server and configuration for http and https servers and define different ports





# MongoDB with Nest

* Install package npm i mongoose @nestjs/mongoose

## Connecting to MongoDB

* Import the MongooseModule and in the imports of app.module add MogooseModule.forRoot() and pass connection string in it
* Now you are connected to MongoDB

import { Module } from '@nestjs/common';  
import {UserModule} from "./modules/user/user.module";  
import {BookModule} from "./modules/book/book.module";  
import {AuthorModule} from "./modules/author/author.module";  
import {SpecificModule} from "./modules/specific/specific.module";  
import {ConfigModule} from "@nestjs/config";  
import {MongooseModule} from "@nestjs/mongoose";  
import {MongooseModelModule} from "./modals/mongoose\_model.module";  
  
@Module({  
 imports: [UserModule, MongooseModule.forRoot("mongodb://127.0.0.1:27017/nestcrud"),  
 controllers: [],  
 providers: [],  
})  
export class AppModule {}

## Creating Scheme

* Create a new folder with name modals/scehema and all the modals and schema goes there
* Export the class and add @Schema() decorator on the class to specify it is schema class
* Now name all the attribute and above each attribute add Prop() decorator to apply validates on it

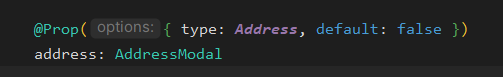
import {Prop, Schema, SchemaFactory} from "@nestjs/mongoose";  
import {Address, AddressModal} from "./address.modal";  
  
@Schema({  
 timestamps: true,  
 collection: "users" // table name in db  
})  
export class UserModal {  
  
 @Prop({ required: true })  
 name: string  
  
 @Prop({ required: true })  
 email: string  
  
 @Prop({ required: true })  
 phone: string  
  
 @Prop({ required: true, select: false }) // select false will not send password on all find queries  
 password: string  
  
 @Prop({  
 required: true,  
 enum: ["ACTIVE","DISABLE"],  
 immutable: true  
 })  
 status: string  
  
 @Prop({ default: [] })  
 social: string[]  
  
 @Prop({ default: false })  
 isVerified: boolean  
  
 @Prop({ type: ***Address***, default: false })  
 address: AddressModal  
  
}  
  
export const ***User*** =SchemaFactory.createForClass(UserModal);

user.modal.ts

* You can also use other schema to define attribute value

import {Prop, Schema, SchemaFactory} from "@nestjs/mongoose";  
  
@Schema({  
 timestamps: true,  
 collection: "addresses" // table name in db  
})  
export class AddressModal {  
  
 @Prop({ required: true })  
 city: string  
  
 @Prop({ required: true })  
 state: string  
  
 @Prop({ required: true })  
 country: string  
}  
  
export const ***Address*** =SchemaFactory.createForClass(AddressModal);

address.modal.ts



Defining address type as Address schema in User Modal

### Adding References to Modal

* You can also set foreign key by providing the reference of the other modal as

import {Prop, Schema, SchemaFactory} from "@nestjs/mongoose";  
import {Types} from "mongoose";  
import {UserModal, UserModalName} from "./user.modal";  
  
  
@Schema({  
 timestamps: true,  
 collection: "posts" // table name in db  
})  
export class PostModal {  
  
 @Prop({ type: Types.ObjectId , required: true, ref: ***UserModalName*** })  
 uid: string | UserModal | Types.ObjectId  
  
 @Prop({ required: true })  
 title: string  
  
 @Prop({ required: true })  
 desc: string  
}  
  
export const ***Post***=SchemaFactory.createForClass(PostModal)  
  
export const ***PostModalName***="PostModal"  
export const ***PostSchemaName***="Post"

## Creating Instances of Model Globally

* After crating all the modals/schema, create a new module in Models with name mongooseModelModule and make it module and add @Global decorator plus make an array of all modals and add them to MongooseModule in imports and also export it
* After that just register them in the app.module.ts file and you can create there instances in any file than

import {Global, Module} from "@nestjs/common";  
import {User, UserModalName} from "./user.modal";  
import {Address, AddressModalName} from "./address.modal";  
import {Post, PostModalName} from "./post.modal";  
import {MongooseModule} from "@nestjs/mongoose";  
  
  
const models=[  
 { name: ***UserModalName***, schema: ***User***},  
 { name: ***AddressModalName***, schema: ***Address***},  
 { name: ***PostModalName***, schema: ***Post***},  
]  
  
@Global()  
@Module({  
 imports: [MongooseModule.forFeature(models)],  
 exports: [MongooseModule],  
})  
  
export class MongooseModelModule {  
   
}

# Populate in MongoDB

* It is use to fetch result from another table based on reference modal
* Suppose in a post modal we have a uid which defines user id we create reference in post that uid is reference to user modal, now if we run populate on by providing the field name to populate we got the full result of it from user table

import {Prop, Schema, SchemaFactory} from "@nestjs/mongoose";  
import {Address, AddressModal} from "./address.modal";  
  
@Schema({  
 timestamps: true,  
 collection: "users" // table name in db  
})  
export class UserModal {  
  
 @Prop({ required: true })  
 name: string  
  
 @Prop({ required: true })  
 email: string  
  
 @Prop({ required: true })  
 phone: string  
  
 @Prop({ required: true, select: false }) // select false will not send password on all find queries  
 password: string  
  
 @Prop({  
 required: true,  
 enum: ["ACTIVE","DISABLE"],  
 immutable: true  
 })  
 status: string  
  
 @Prop({ default: [] })  
 social: string[]  
  
 @Prop({ default: false })  
 isVerified: boolean  
  
 @Prop({ type: ***Address***, default: false })  
 address: AddressModal  
  
}  
  
export const ***User*** =SchemaFactory.createForClass(UserModal);  
  
export const ***UserModalName***="UserModal"  
export const ***UserSchemaName***="User"

User Modal

import {Prop, Schema, SchemaFactory} from "@nestjs/mongoose";  
import {Types} from "mongoose";  
import {User, UserModal, UserModalName, UserSchemaName} from "./user.modal";  
import mongoose from "mongoose";  
  
  
@Schema({  
 timestamps: true,  
 collection: "posts" // table name in db  
})  
export class PostModal {  
  
 @Prop({ type: mongoose.Schema.Types.ObjectId , required: true, ref: ***UserModalName*** })  
 uid: mongoose.Schema.Types.ObjectId  
  
 @Prop({ required: true })  
 title: string  
  
 @Prop({ required: true })  
 desc: string  
}  
  
export const ***Post***=SchemaFactory.createForClass(PostModal)  
  
export const ***PostModalName***="PostModal"  
export const ***PostSchemaName***="Post"

Posr Modal

@Get('/all')  
async getAllUsers(@Request() req, @Response() res, @Body() body)  
{  
 try {  
 let data = await this.Post.find().populate("uid");  
 return res.status(200).json({status: 200, message: "Data Fetched", data:data})  
 }  
 catch (e) {  
 return res.status(500).json({status: 500, message: e.name})  
 }  
}

Query in controller

# Multiple Connection of MongoDB in Nest

* First you need to create a module for the connections and create multiple connections there with connectionName and then register them to the app.module file

import {Global, Module} from "@nestjs/common";  
import {MongooseModule} from "@nestjs/mongoose";  
  
@Global()  
@Module({  
 imports: [  
 MongooseModule.forRoot("mongodb://127.0.0.1:27017/nestcrudapp", {connectionName: "APP"}),  
 MongooseModule.forRoot("mongodb://127.0.0.1:27017/nestcrudadmin", {connectionName: "ADMIN"}),  
 ],  
 exports: [MongooseModule],  
})  
  
export class ConnectionModule {}

* Next thing is we have to define the connection name in modal module that which modal are for admin and which are for the app in my case

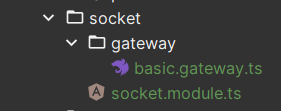
import {Global, Module} from "@nestjs/common";  
import {User, UserModalName} from "./user.modal";  
import {Address, AddressModalName} from "./address.modal";  
import {Post, PostModalName} from "./post.modal";  
import {MongooseModule} from "@nestjs/mongoose";  
  
  
const appmodels=[  
 { name: ***UserModalName***, schema: ***User***},  
 { name: ***AddressModalName***, schema: ***Address***},  
 { name: ***PostModalName***, schema: ***Post***},  
]  
  
const adminmodels=[  
 { name: ***UserModalName***, schema: ***User***},  
 { name: ***AddressModalName***, schema: ***Address***},  
 { name: ***PostModalName***, schema: ***Post***},  
]  
  
@Global()  
@Module({  
 imports: [  
 MongooseModule.forFeature(appmodels, "APP"),  
 MongooseModule.forFeature(adminmodels, "ADMIN")  
 ],  
 exports: [MongooseModule],  
})  
  
export class MongooseModelModule {  
   
}

* Now the last thing we have to define the type of connection in the modal injection as below

import {  
 Controller,  
 Get,  
 Request,  
 Response,  
 Body,  
 Post, Param, Delete,  
} from '@nestjs/common';  
  
  
import {InjectModel} from "@nestjs/mongoose";  
import {PostModalName} from "../../modals/post.modal";  
  
@Controller('post')  
export class PostController {  
  
 constructor(@InjectModel(***PostModalName***,"APP") private Post) {  
  
 }  
  
 @Post('/create')  
 async createUser(@Request() req, @Response() res, @Body() body)  
 {  
 try {  
 await this.Post.create(body);  
 return res.status(200).json({status: 200, message: "POst Created"})  
 }  
 catch (e) {  
 return res.status(500).json({status: 500, message: e.name})  
 }  
 }  
  
 @Get('/all')  
 async getAllUsers(@Request() req, @Response() res, @Body() body)  
 {  
 try {  
 let data = await this.Post.find().populate("uid");  
 return res.status(200).json({status: 200, message: "Data Fetched", data:data})  
 }  
 catch (e) {  
 return res.status(500).json({status: 500, message: e.name})  
 }  
 }  
  
  
 @Get('/by/id/:id')  
 async getUserById(@Request() req, @Response() res, @Body() body, @Param() params)  
 {  
 try {  
 let data = await this.Post.findById(params.id).populate("uid");;  
 return res.status(200).json({status: 200, message: "POst Fetched", data:data})  
 }  
 catch (e) {  
 return res.status(500).json({status: 500, message: e.name})  
 }  
 }  
}

# Sockets in Nest

* Tutorial link <https://youtu.be/7xpLYk4q0Sg?si=jzevnywjqKTZck1V>
* Here is the complete documentation <https://docs.nestjs.com/websockets/gateways>
* Install these packages npm i --save @nestjs/websockets @nestjs/platform-socket.io
* First you need to create a socket module file and a gateway file where all implementation is done
* Add the gateway file in the providers of socket module and now register the socket module in the app.module file
* Your socket are set



import {MessageBody, SubscribeMessage, WebSocketGateway, WebSocketServer} from "@nestjs/websockets";  
  
@WebSocketGateway()  
// @WebSocketGateway(80) // if you want to run on other port  
export class BasicGateway {  
  
 @WebSocketServer()  
 server;  
  
 @SubscribeMessage('newMessage')  
 handleNewMessage(@MessageBody() message){  
 this.server.emit('message', message)  
 }  
  
}

basic.gateway.ts

import {Module} from "@nestjs/common";  
import {BasicGateway} from "./gateway/basic.gateway";  
  
  
@Module({  
 imports: [],  
 controllers: [],  
 providers: [BasicGateway],  
 exports: []  
})  
export class SocketModule {}

socket.module.ts

import { Module } from '@nestjs/common';  
import {UserModule} from "./modules/user/user.module";  
import {BookModule} from "./modules/book/book.module";  
import {AuthorModule} from "./modules/author/author.module";  
import {SpecificModule} from "./modules/specific/specific.module";  
import {ConfigModule} from "@nestjs/config";  
import {MongooseModelModule} from "./modals/mongoose\_model.module";  
import {PostModule} from "./modules/post/post.module";  
import {ConnectionModule} from "./connection/connection.module";  
import {SocketModule} from "./modules/socket/socket.module";  
  
@Module({  
 imports: [UserModule, BookModule, AuthorModule, SocketModule, ConfigModule.forRoot({envFilePath: [".env",".prod.env"], cache: true, expandVariables: true, isGlobal: true})],  
 controllers: [],  
 providers: [],  
})  
export class AppModule {}

app.module.ts file

## How to use and implement event

* First you need to export the class and add @WebSocketGateway decorator to the class
* Nest you have to cerate socket.io instance by add a decorator @WebSocketServer to a variable
* Than you can listen to event triger by client by implementing the function and that function will be decorated by @SubscribeMessage('eventName') and to get the payload from the function you can use @MessageBody() decorator

import {MessageBody, SubscribeMessage, WebSocketGateway, WebSocketServer} from "@nestjs/websockets";  
  
@WebSocketGateway()  
// @WebSocketGateway(80) // if you want to run on other port  
export class BasicGateway {  
  
 @WebSocketServer()  
 server;  
  
 @SubscribeMessage('newMessage')  
 handleNewMessage(@MessageBody() message){  
 this.server.emit('message', message)  
 }  
  
}