Intro to Nest.JS

Jose L. Muñoz Tapia
Marc Cosgaya Capel

Outline

Outline

What is Nest.js?

Nest (NestJS) is an open-source, extensible, versatile, progressive Node.js framework for building efficient, scalable Node.js server-side applications (back-end).

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).

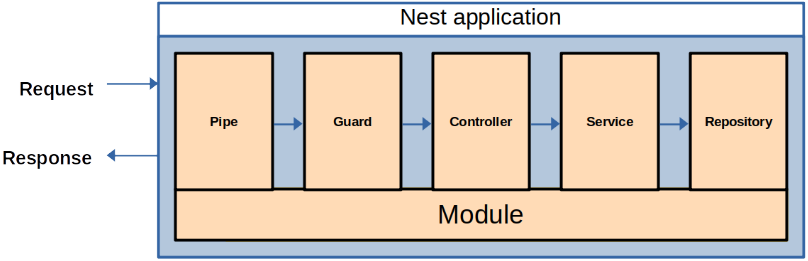
Nest.js main features

- Easy to use, learn and master.
- Powerful Command Line Interface (CLI) to boost productivity and ease of development.
- Detailed and well-maintained documentation.
- Active codebase development and maintenance.
- Support for dozens of nest-specific modules to integrate with common technologies and concepts like TypeORM, Mongoose, GraphQL, Logging, Validation, Caching, WebSockets and much more.
- Easy unit-testing applications.
- On top of NestJS you can easily build Rest API's, MVC applications, microservices, GraphQL applications, Web Sockets or CLI's and CRON jobs.

Outline

Nest application architecture

Roughly, a Nest application will take requests, process it in some way and return a response.



The requests handling logic is divided into modular blocks. Each type of block is intended for a specific purpose. Nest has tools to help us to write this blocks in a fast and efficient way. Several building blocks are packed in one module.

Nest building blocks

Nest, out of the box, provides us the following list of building blocks types:

- **Controllers:** Handles incoming requests by routing it to a particular function.
- **Services:** Handles business logic execution or access data through a repository.
- **Pipes:** Validates data contained in the requests.
- **Filters:** Handles errors that may occur during request handling.
- **Guards:** Handles authentication strategies.
- **Interceptors:** Adds extra logic to incoming requests or outgoing responses.
- **Repositories:** Handles data stored in a DB (stores or retrieves data).
- **Modules:** Groups together different building blocks.

Modularity

Nest modularity allows us to develop reusable logical parts that can be used across different types of applications.

Nest provides an out-of-the-box application architecture which allows developers and teams to create highly testable, scalable, loosely coupled, and easily maintainable applications.

Build once, use everywhere!

Outline

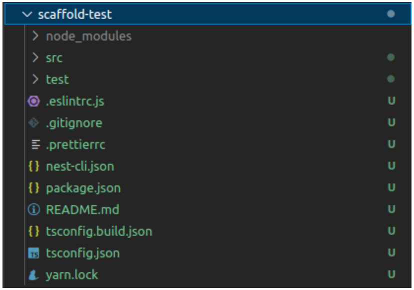
Modular file structure

The file structure of a Nest.js project follows a modular pattern in line with the logic structure of Nest.

Using nest CLI, we can generate the Nest project scaffold to examine the basic file structure of a Nest project.

```
$ nest new <project-name>
```

The command will generate a folder named as the project with the following file structure:



```
scaffold-test
  node_modules
  src
  test
  .eslintrc.js          U
  .gitignore            U
  .prettierrc           U
  nest-cli.json         U
  package.json          U
  README.md             U
  tsconfig.build.json   U
  tsconfig.json         U
  yarn.lock             U
```

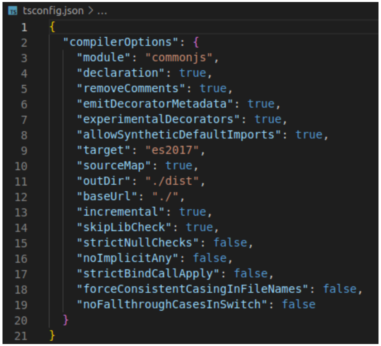
Common dependencies

We can see the nest dependencies in **package.json** file in the project root. Below is shown a list of the most important ones:

- **@nestjs/common:** Contains vast majority of functions, classes, etc, that we need from Nest. Comes with decorators such as @Controller(), @Injectable() and so on.
- **@nestjs/core:** Implements Nest's core functionality, low-level services, and utilities.
- **@nestjs/platform-express:** Nest.js Express adapter. Under the hood, Nest makes use of robust HTTP Server frameworks like Express. Also compatible with Fastify (@nestjs/platform-fastify).
- **reflect-metadata:** Used by TypeScript to design decorators, it allows for meta data to be included to a class or function; essentially it is syntax sugar.
- **typescript:** TypeScript compiler.TypeScript adds optional types to JavaScript that support tools for large-scale JavaScript applications. Not installed by default in Nest project, but highly recommended.

TypeScript compiler setup

Typescript compiler settings can be found in **tsconfig.json** file in the project root. The TypeScript compiler transpiles our code to JavaScript code, the code must be transpired before run it since node.js cannot interpret TypeScript directly.


tsconfig.json > ...

```json
{
  "compilerOptions": {
    "module": "commonjs",
    "declaration": true,
    "removeComments": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "allowSyntheticDefaultImports": true,
    "target": "es2017",
    "sourceMap": true,
    "outDir": "./dist",
    "baseUrl": "./",
    "incremental": true,
    "skipLibCheck": true,
    "strictNullChecks": false,
    "noImplicitAny": false,
    "strictBindCallApply": false,
    "forceConsistentCasingInFileNames": false,
    "noFallthroughCasesInSwitch": false
  }
}
```

experimentalDecorators and **emitDecoratorMetada** are the most relevant settings and core functionalities of Nest.

/src folder

All the logic that will be coded by us must be placed in a folder called **src** in the root of the project. Every Nest application will contain at least one module and one controller.

Below is shown the scaffold /src folder:

- **app.controller.ts:** A basic controller with a single route.
- **app.controller.spec.ts:** The unit tests for the controller.
- **app.module.ts:** The root module of the application. Note that Nest.js serves a single root module per application.
- **app.service.ts:** A basic service with a single method.
- **main.ts:** The entry file of the application which uses the core function NestFactory to create a Nest application instance. Includes an async function, which will bootstrap our application.

Naming convention

In order to make our code more scalable and maintainable, we have to use a few naming conventions when we build our Nest application.

- One single class per file.
- Class names should include the kind of thing we are creating.
- Name of class and name of file should always match up.
- Filename template: <moduleName>.<typeOfThing>.ts (example: myModule.controller.ts)
- Class name template: <moduleName><TypeOfThing> (example: myModuleController)

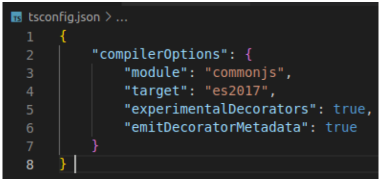
Outline

Start project

A Nest application will contain at least one module and one controller. So a minimum viable app will contain a module and a controller.

```
$ npm init -y
$ npm install @nestjs/common @nestjs/core @nestjs/platform-express reflect-metadata typescript
```

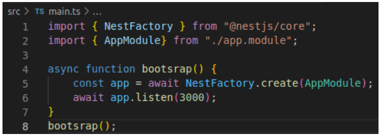
We need to define minimum TypeScript compiler configuration in tsconfig.json.



```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es2017",
        "experimentalDecorators": true,
        "emitDecoratorMetadata": true
    }
}
```

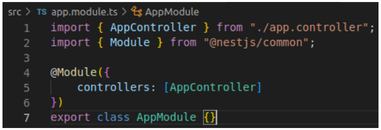
Basic files

We have to create an src folder with three files inside it: main.ts, app.module.ts and app.controller.ts.



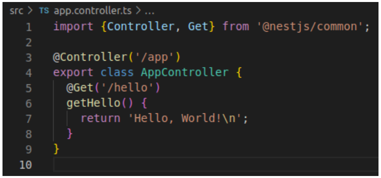
src > TS main.ts > ...

```
import { NestFactory } from "@nestjs/core";
import { AppModule} from "./app.module";

async function bootsrap() {
    const app = await NestFactory.create(AppModule);
    await app.listen(3000);
}
bootsrap();
```



src > TS app.module.ts > AppModule

```
import { AppController } from "./app.controller";
import { Module } from "@nestjs/common";

@Module({
    controllers: [AppController]
})
export class AppModule {}
```



src > TS app.controller.ts > ...

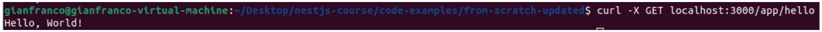
```
import {Controller, Get} from '@nestjs/common';

@Controller('/app')
export class AppController {
  @Get('/hello')
  getHello() {
    return 'Hello, World!\n';
  }
}

```

Running the application

This command is for only for running the application from scratch. Later we will use Nest CLI to run our app.

```
$ npx ts-node-dev src/main.ts\
```

The application will start to listen for http requests at port 3000 of our machine.



```
gianfranco@gianfranco-virtual-machine:~/Desktop/nestjs-course/code-examples/from-scratch-updated$ curl -X GET localhost:3000/app/hello
Hello, World!
```

Outline

Nest CLI installation

The Nest CLI is a command-line interface tool that helps you to initialize, develop, and maintain your Nest applications. It embodies best-practice architectural patterns to encourage well-structured apps.

Nest CLI is installed through npm, to install it globaly in our system we will use the following command:

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

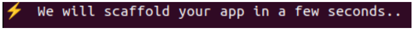
Nest CLI has 6 commands that we can use: **new**, **generate**, **info**, **build**, **start** and **add**.

nest new

The **new** command is used to generate a new Nest scaffolding project avoiding the need to build it from scratch.

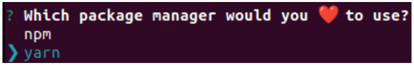
To generate a new project we will run the following command

```
$ nest new <ApplicationName>
```



```
⚡  We will scaffold your app in a few seconds..
```

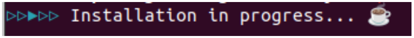
If you have multiple node package managers installed in your system, Nest CLI will ask which one you want to use.



```
? Which package manager would you ❤️ to use?
  npm
❯ yarn
```

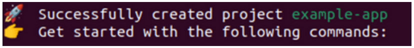
nest new

Then, the installation will begin, it will take a couple of minutes.



```
▷▷▶▷▷ Installation in progress... ☕
```

If the installation succeeded, we will get the following message.

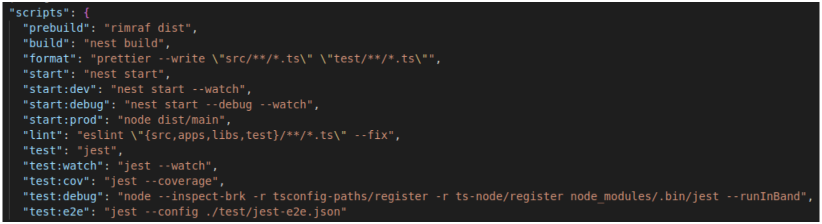


```
🚀  Successfully created project example-app
👉  Get started with the following commands:
```

Nest CLI will automatically fill our project directory with all the necessary files to run a basic Nest application.

nest new

If we check the package.json file placed in the project root, we will see all the useful scripts that Nest CLI has created for us in the new project. Any of them can be executed from project directory with npm <script name> command, or if we use yarn as packed manager, with yarn <script name> command.



```
"scripts": {
  "prebuild": "rimraf dist",
  "build": "nest build",
  "format": "prettier --write \"src/**/*.ts\" \"test/**/*.ts\"",
  "start": "nest start",
  "start:dev": "nest start --watch",
  "start:debug": "nest start --debug --watch",
  "start:prod": "node dist/main",
  "lint": "eslint \"{src,apps,libs,test}/**/*.ts\" --fix",
  "test": "jest",
  "test:watch": "jest --watch",
  "test:cov": "jest --coverage",
  "test:debug": "node --inspect-brk -r tsconfig-paths/register -r ts-node/register node_modules/.bin/jest --runInBand",
  "test:e2e": "jest --config ./test/jest-e2e.json"
```

nest generate

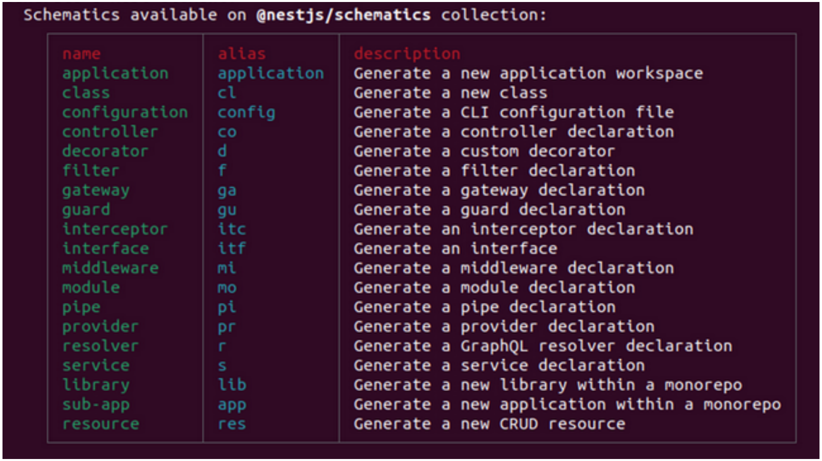
The **generate** command is used to generate Nest's functional component class files with some code inside them, instead of having to create every file manually from scratch.

```
$ nest generate <schematic> <name> [options]
```

Schematics are template code generators, and we have schematics to generate every possible block component of a Nest application.

nest generate

Using the --help option we can see all possible class files that we can create with nest generate:



```
Schematics available on @nestjs/schematics collection:
  ┌───────────────┬─────────────┬──────────────────────────────────────────────┐
  │ name          │ alias       │ description                                  │
  │ application   │ application │ Generate a new application workspace         │
  │ class         │ cl          │ Generate a new class                         │
  │ configuration │ config      │ Generate a CLI configuration file            │
  │ controller    │ co          │ Generate a controller declaration            │
  │ decorator     │ d           │ Generate a custom decorator                  │
  │ filter        │ f           │ Generate a filter declaration                │
  │ gateway       │ ga          │ Generate a gateway declaration               │
  │ guard         │ gu          │ Generate a guard declaration                 │
  │ interceptor   │ itc         │ Generate an interceptor declaration          │
  │ interface     │ itf         │ Generate an interface                        │
  │ middleware    │ mi          │ Generate a middleware declaration            │
  │ module        │ mo          │ Generate a module declaration                │
  │ pipe          │ pi          │ Generate a pipe declaration                  │
  │ provider      │ pr          │ Generate a provider declaration              │
  │ resolver      │ r           │ Generate a GraphQL resolver declaration      │
  │ service       │ s           │ Generate a service declaration               │
  │ library       │ lib         │ Generate a new library within a monorepo     │
  │ sub-app       │ app         │ Generate a new application within a monorepo │
  │ resource      │ res         │ Generate a new CRUD resource                 │
  └───────────────┴─────────────┴──────────────────────────────────────────────┘
```

nest generate

The class files generated using the generate command, will follow the Nest naming convention by default. We don't need to specify the component type in the <name> field to follow Nest naming convention, the generate command will create the class following it automatically.

Example:

- **$ nest generate module App**: generates a file /src/app.module.ts with a class AppModule inside it. **(Correct)**
- **$ nest generate module AppModule**: generates a file /src/app-module.module.ts with a class AppModuleModule inside it. **(Not correct)**

nest generate, module generation example

We can create a module called MyApp using the following command inside an existing project folder.

```
$ nest generate module MyApp
```

Nest CLI will create a file named my-app.module.ts inside src/my-app folder, with the following code inside:

```
import { Module } from '@nestjs/common';

@Module({})
export class MyAppModule {}
```

nest generate, controller generation example

We can create a controller and add it directly to an existing app using the syntax <”existing app name”/”controller name”> as controller name.

For example, if we use the following command:

```
$ nest generate controller MyApp/MyHandlingLogic --flat
```

Nest automatically will create a new class file named my-handling-logic.controller.ts to start coding the routing logic, and a test file named my-handling-logic.controller.spec.ts to start coding the test logic of the controller.

my-handling-logic.controller.ts:

```
import { Controller } from '@nestjs/common';

@Controller('my-handling-logic')
export class MyHandlingLogicController {}
```

nest generate, controller generation example

Moreover, since we have specified in the command, Nest CLI will automatically edit MyAppModule class file (my-app.module.ts) to add the new controller in the module's controllers array.

my-app.module.ts:

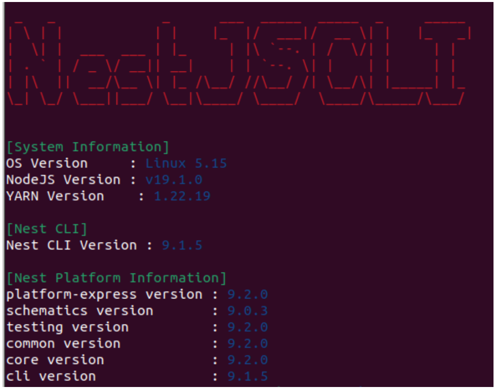
```
import { Module } from '@nestjs/common';
import { MyHandlingLogicController } from './my-handling-logic.controller';
@Module({
        controllers: [MyHandlingLogicController]
})
export class MyAppModule {}
```

If we do not specify the --flat option Nest CLI will create the new controller inside a folders controllers (src/myapp/controllers/). For large projects with lots of files, it can be useful to sort component class files by types in different folders, but for our scope, it's not strictly necessary and the Nest CLI lets us make the choice.

nest info

The **info** command displays information about nest project and other helpful system info.

```
$ nest info
```



```
 _   _             _      ___  _____  _____  _     _____
| \ | |           | |    |_  |/  ___|/  __ \| |   |_   _|
|  \| |  ___  ___ | |_     | |\ `--. | /  \/| |     | |
| . ` | / _ \/ __|| __|    | | `--. \| |    | |     | |
| |\  ||  __/\__ \| |_ /\__/ //\__/ /| \__/\| |_____| |_
\_| \_/ \___||___/ \__|\____/ \____/  \____/\_____/\___/


[System Information]
OS Version     : Linux 5.15
NodeJS Version : v19.1.0
YARN Version    : 1.22.19

[Nest CLI]
Nest CLI Version : 9.1.5

[Nest Platform Information]
platform-express version : 9.2.0
schematics version       : 9.0.3
testing version          : 9.2.0
common version           : 9.2.0
core version             : 9.2.0
cli version              : 9.1.5
```

nest build and nest start

The **build** command compiles an application or workspace into an output folder. Uses tsc compiler to transpile TypeScript to plain JavaScript.

```
$ nest build
```

The **start** command compiles and runs an application.

```
$ nest start
```

To run the app live reload mode, that its whenever changes are made to the code, live updates are done to the application (development mode)

```
$ nest start --watch
```

nest add

The **add** command imports a library that has been packaged as a nest library, running its install schematic. A Nest library is a Nest project that differs from an application in that it cannot run on its own. A library must be imported into a containing application in order for its code to execute.

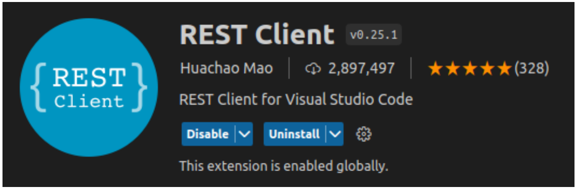
Any functionality that is suitable for re-use is a candidate for being managed as a library. Deciding what should be a library, and what should be part of an application, is an architectural design decision. Creating libraries involves more than simply copying code from an existing application to a new library. When packaged as a library, the library code must be decoupled from the application.

```
$ nest add [options] <library>
```

Outline

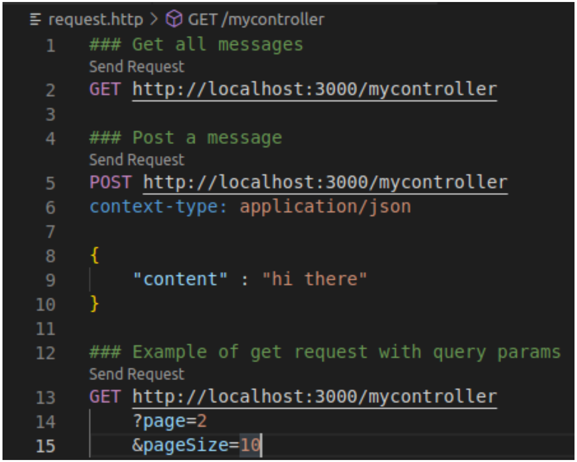
VSCode REST Client extension

To test our application in an easy way manually, we have to use an API client. Postman is a widely used client, but VSCode's REST Client extension offers a lighter option to send custom HTTP requests to our API.



.http file

Then, to send HTTP requests we have to create a file with .http extension. The extension will detect it as a known file extension and will highlight the syntax of the inner code.



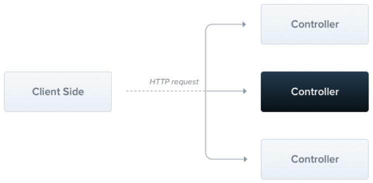
```
### Get all messages
GET http://localhost:3000/mycontroller

### Post a message
POST http://localhost:3000/mycontroller
context-type: application/json

{
    "content" : "hi there"
}

### Example of get request with query params
GET http://localhost:3000/mycontroller
    ?page=2
    &pageSize=10
```

Outline

Outline

Controllers

A controller, like in other frameworks, is a class that provides the API endpoints. Inside the class we can specify the method route and type (GET, POST...). The main goal of a controller is to interface between the client and the service.



Any object that we return from a controller, will be automatically converted into a JSON response by Nest.

Controllers

Controller methods can take any name, but need to use decorators like @Get() and @Post() that accept the route suffix pattern as parameter. The methods can be either sync or async. The returned HTTP status code is 200 for @Get() and 201 for @Post(), @Patch(), @Put(), etc.

In the controller we also have several parameter decorators for accessing data in the request:

- @Param(key?: string) extracts a parameter defined in the route decorator pattern with ':something'.
- @Query(key?: string) extracts a parameter from the query string.
- @Headers(name?: string) extracts values from the request headers.
- @Body(key?: string) extracts the body of the request.

Controllers

These decorators are applied to method arguments:

```
@Get('suffix/:someParam/moreSuffix')
methodName(@Param() params: any, @Query('optionA'): optionA: string, @Param('someParam') paramA: string) {
        ...
        this.someService.methodA(optionA);
        ...
}
```

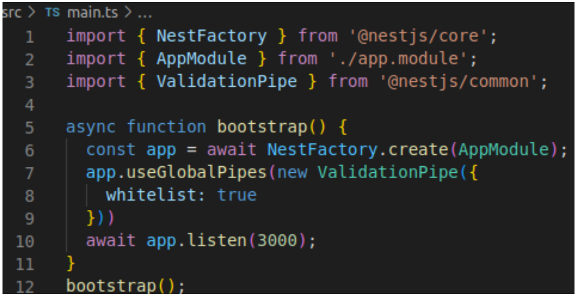
If we are using @nestjs/platform-express, we can also access the request and response themselves with @Req() and @Res(), respectively. However, if we use @Res(), then the method acts like a regular Express handler, so we need to do a res.send() or res.json() at some point.

Another important note, Nestjs will execute the first method that it finds to fit the pattern of the URL. So, more general patterns need to be put at the bottom of the class and more specific ones near the top.

Outline

Pipes

Pipes validate the data contained within the request. To add ValidationPipe, for example, we have to:


src > main.ts > ...

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe({
    whitelist: true
  }))
  await app.listen(3000);
}
bootstrap();
```

app.useGlobalPipes() will apply the pipes globally in our application.

ValidationPipe

ValidationPipe allows us to use DTOs for validating the requests to our Nest application. A Data Transfer Object, or DTO, is used to describe the properties of the body, query or route parameters of the request. The format is as follows:

```
export class MethodNameDto {
        key1: type1;
        key2: type2;
        ...
}
```

The file name should be method-name.dto.ts and placed within a dtos directory inside a module directory.

To add validation to the DTO we first need to install two new packages:

```
$ npm install class-validator class-transformer
```

DTOs

class-validator provides a full set of validators for the body.

class-transformer automates the process of converting json objects into javascript objects. Here, @Type() is converting the value into the Number type. Moreover, recall the whitelist: true from before, its use is to tell class-transformer to strip extra values from the body of the request that don't conform with the DTO structure. This adds extra security.

Now the MethodNameDto can be used as the type of the argument in the controller method.

An example of validating an integer can be:

```
import { Type } from "class-transformer"
import { IsInt } from "class-validator";

export class MethodNameDto {
        @Type(() => Number)
        @IsInt()
        key: number;
}
```

DTOs

The order of the decorators is important if we want to have relevant error messages. The decorators put closer to the attribute are applied first, while the ones put farther away are applied later.

We can also have nested DTOs. An example on how to achieve this:

```
import { Type } from "class-transformer"
import { ValidateNested } from "class-validator"
import { ChildDto } from "..."

export class ParentDto {
        @Type(() => ChildDto)
        @ValidateNested()
        childDto: ChildDto
}
```

@ValidateNested() will make Nest.js validate the nested DTO. Note that we also need to change the type.

Finally, we can make attributes optional while preserving type validation. To do so, we need to use @IsOptional() from class-validator.

ValidationPipe

The validation pipe has the following process:

1. Transform the json body into a DTO class instance using class-transformer.
2. Use the class-validator decorators to validate the DTO class instance.
3. If an error occurs, then exit immediately. Otherwise, return the body to the method argument.

Outline

Services and Repositories

Services are used for managing business logic, repositories are used for managing storage-related logic. Separating the two allows a more modular structure of our application.

A service uses one or more repositories. A service may end up having the same method signatures as the repository. Service and repository class names are of the form ModuleService and ModuleRepository respectively. The methods in a repository must be async and thus return a Promise.

Outline

Returning an error

Sometimes we want to return a different HTTP status code to the client. Nest.js has multiple exception types that can be imported from @nestjs/common, like NotFoundException. We can also use a generic HttpException that allows us to specify the status code. Nest.js returns a status code whenever an uncatched exception is thrown.

Outline

Outline

Inversion of control

One way we can connect a service to a repository is to import the latter in the service and instantiate a class attribute in the constructor. This is a bad practice for handling tests.

Instead, Nest.js uses dependency injection which follows the inversion of control principle. In the previous approach we instantiated the dependency of a service class manually. In inversion of control, the dependency is already given as a parameter to the constructor.

How Nest does DI

The best practice would be to have this parameter as an interface of a specific implementation of the dependency. This would allow us to easily replace the implementation without having to modify the other classes. However, Nest.js doesn't do this due to some limitations of TypeScript.

Nest.js uses a container, also called an injector, for listing classes and their dependencies and handling their instances as singletons. This container is created at startup and first instantiates the classes without dependencies. It then instantiates the other classes in an orderly manner. There's one container per module.

How to use DI

We can add a dependency to an injector by using the @Injectable() decorator to the class. This decorator is imported from @nestjs/common. We will also need to register the class to the providers property (array) of the @Module(). Now we can have the constructor of the parent class as follows:

```
constructor(private dependencyClass: dependencyClass) {}
```

The argument will be automatically handled by the injector.

Injection scopes

The @Injectable() decorator can accept an object as a parameter. One of its properties is ’scope’. After importing Scope from @nestjs/common, the possible values relevant to us are Scope.DEFAULT and Scope.REQUEST.

- The DEFAULT scope is the singleton previously mentioned.
- The REQUEST scope ensures that a new instance is created (and garbage collected) in each request.

Outline

Importing modules

To inject a dependency from another module, it first needs to be added to the exports property (array) of that @Module(). So, the dependency must be in both the providers and exports properties:

```
import { Module } from '@nestjs/common';
import { dependencyClass } from 'src/other/dependency.class';

@Module({
        providers: [dependencyClass],
        exports: [dependencyClass]
})
export class OtherModule {}
```

In the new module, we have to add an imports property (array) to @Module() with the first module:

```
import { Module } from '@nestjs/common';
import { OtherModule } from 'src/other/other.module';

@Module({
        imports: [OtherModule],
})
export class ThisModule {}
```

How the injector uses modules

This way the module has access to the dependencies listed in the exports property of the other one. The dependency is still accessed through the constructor methods.

Now the injector has another list for exported dependencies that can be used in other injectors. The injector that is importing the dependencies will have the previously exported classes made available. Note that children of child modules don't need to be imported as they're implicitly injected.

Outline

Outline

TypeORM

Install the required Node.js packages with:

```
$ npm i @nestjs/typeorm typeorm
```

Then we have to install the specific implementation of the database like SQLite or MongoDB. For example, the SQLite package is sqlite3.

The steps needed to use TypeORM are:

- Connect to the specific implementation of the DB in the app module.
- Impement entity classes for each repository.

Connect DB

To connect a DB to the application, we have to do one import and set some initial configuration. This configuration is also called root connection. To implement it, add this to the app module:

```
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
        imports: [TypeOrmModule.forRoot({
                type: 'sqlite',
                database: 'db.sqlite',
                entities: [],
                synchronize: true
        })],
        controllers: [AppController],
        providers: [AppService],
})
```

Type specifies the type of DBMS. Database tells TypeORM the name of the database. Entities is the array of entity classes. Synchronize tells TypeORM whether or not to automatically run migrations, which is dangerous in a production environment.

Entities

Entity class files are named name.entity.ts and the class themselves are named Name, without the Entity suffix. So, for a Users module, the entity will be named user.entity.ts and the class User.

The main decorators for entities, imported from typeorm, are:

- Entity: Specifies that the class is indeed an entity and a DB table must be associated with it.
- PrimaryGeneratedColumn: Specifies that the attribute is the primary key of the table.
- Column: Speficies that the attribute is a column of the table.

Entities

Once we have an entity class, we can use it in a module by doing:

```
import { Module } from '@nestjs/common';
import { NamesController } from './names.controller';
import { NamesService } from './names.service';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Name } from './name.entity';

@Module({
        imports: [TypeOrmModule.forFeature([Name])],
        controllers: [NamesController],
        providers: [NamesService]
})
export class NamesModule {}
```

Note that TypeOrmModule is a dependency of NamesModule.

Then we have to import the entity in the app module and add it to the entities property of the root connection. TypeORM and Nest.js will automatically generate the repository class.

Interacting with DB

A repository is not the same as an entity. The repository, on one hand, is the interface that allows us to communicate with the database implementation. The entity, on the other hand, specifies how the data is structured in the database.

A repository has several methods, all of them are explained in the documentation.

Outline

Prisma

Install the required Node.js packages with:

```
$ npm i prisma
```

Connect DB

Now we will use the prisma CLI:

```
$ npx prisma init
```

This will generate a prisma directory with a schema.prisma file inside it. It will look like this:

```
generator client {
        provider = "prisma-client-js"
}

datasource db {
        provider = "sqlite"
        url      = env("DATABASE_URL")
}
```

The datasource db part specifies what DBMS to use and what url in the .env file is used to connect. This file will be generated in the root directory and will have a DATABASE_URL variable. We set its value to ”file:./dev.db” so that the SQLite database is stored in a file. With other DBMS it will be necessary to connect to the specific URL of the database server.

Connect DB

We will need to implement an injectable service. This will extend the Prisma client to make it work with the Nest.js injectors. We need to create a file named prisma.service.ts in the src/prisma directory:

```
import { INestApplication, Injectable, OnModuleInit } from '@nestjs/common';
import { PrismaClient } from '@prisma/client';

@Injectable()
export class PrismaService extends PrismaClient implements OnModuleInit {
        async onModuleInit() {
                await this.$connect();
        }

        async enableShutdownHooks(app: INestApplication) {
                this.$on('beforeExit', async () => {
                        await app.close();
                });
        }
}
```

Connect DB

This means creating a new module named prisma with prisma.module.ts:

```
import { Module } from '@nestjs/common';
import { PrismaService } from './prisma.service';

@Module({
        providers: [PrismaService],
        exports: [PrismaService]
})
export class PrismaModule {}
```

Now the module can be imported wherever we want to use its service.

Connect DB

Sometimes we may need to regenerate the Prisma client so we have access to the changes we have made. This will update the class:

```
$ npx prisma generate
```

VSCode may complain about the class still not being updated. A quick solution is to close and run the program again.

Models

Prisma models are declared in the schema.prisma file as follows:

```
model Name {}
```

If we're using the Prisma extension for VSCode, it will tell us that a unique or id attribue is needed. In Prisma attributes define extra properties to the column. Each column follows the format:

```
column_name Type @attribute1 @attribute2 ... @attributeN
```

Examples of Type are: Int, Float, String, Date, DateTime and Boolean. Examples of attributes are @id, @default and @unique.

Attributes can also be block-based. A block is a grouping of two or more columns. Block-based attributes require an array as an argument and are declared at the end of the model.

Models

To generate the database file, in the case of SQLite, we have to run:

```
$ npx prisma migrate dev --name init
```

The command will generate the migrations and database file inside the prisma directory. In this case, a migration named init will be created and run. Also, dev indicates that we are in a development environment. In a production environment we would use deploy.

Interacting with DB

Instead of interacting with multiple repositories, Prisma uses a single service for all db-related logic. Instead of entities, Prisma uses models for representing the tables in the database. To interact with a table, each model is accessible as a lowercase attribute of the service.

The models have several methods, all of them are explained in the documentation.

Outline

TypeORM vs Prisma

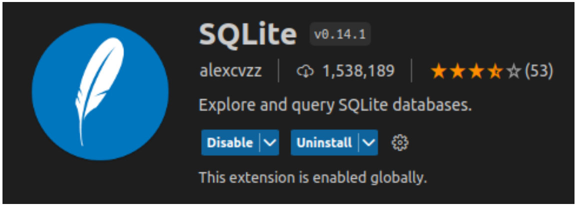
Both TypeORM and Prisma are very popular ORMs. However, TypeORM is slightly more popular than Prisma. But Prisma has better documentation than TypeORM.

Both have a CLI. Nevertheless, Prisma has more tools than TypeORM, like Prisma Studio. Another plus of Prisma is cursor-based pagination.

Outline

VSCode SQLite extension

To view the contents of a SQLite database file, we have to install the extension REST Client in our VSCode. Note that sqlite3 needs to be installed in the system using apt or rpm:


SQLite v0.14.1

alexcvzz | 1,538,189 | ★★★★☆ (53)

Explore and query SQLite databases.

This extension is enabled globally.

With Ctrl+Shift+P we can then search for “SQLite: Open Database” to start exploring the database in a new section of the explorer tab.

Outline

Outline

Import

TypeORM requires more imports.

```
import { Repository } from 'typeorm';
import { InjectRepository } from '@nestjs/typeorm';
import { Name } from './name.entity';
```

TypeORM

```
import { PrismaService } from '../prisma/prisma.service';
```

Prisma

Outline

Constructor

Prisma has a shorter syntax for the target constructor.

```
constructor(@InjectRepository(Name) private repo: Repository<Name>) {}
```

TypeORM

```
constructor(private prisma: PrismaService) {}
```

Prisma

Outline

Create

TypeORM uses hooks, which are methods called when modifying the database records.

```
const name = this.repo.create({
        key1: val1,
        key2: val2
});
// Allow hooks to execute after create.
return this.repo.save(name);
```

TypeORM

```
const name = this.prisma.name.create({
        data: {
                key1: val1,
                key2: val2,
        }
});
return name;
```

Prisma

Outline

Read

We can find one record by its primary key, or find the first one by matching properties or find many by matching properties.

```
this.repo.findOne(pk);
this.repo.findOne({
        key1: val1,
        key2: val2
});
this.repo.find({
        key1: val1,
        key2: val2
});
```

TypeORM

Read

```
this.prisma.name.findUnique({
        where: { pk }
});
this.prisma.name.findFirst({
        where: {
                key1: val1,
                key2: val2
        }
});
this.prisma.name.findMany({
        where: {
                key1: val1,
                key2: val2
        }
});
```

Prisma

Outline

Partial

This is useful for when we don't want to specify again all the properties of a model/entity. Partial is native to TypeScript. Note that in Prisma we get the entities from @prisma/client.

```
whatever: Partial<Name>;
```

TypeORM

```
import { Name } from '@prisma/client';
whatever: Partial<Name>;
```

Prisma

Outline

Update

To update a record, we can use the Partial functionality.

```
partial: Partial<Name>;
partial = some_partial_obj;
const name = await this.repo.findOne(pk);
Object.assign(name, partial);
// Allow hooks to execute after create.
return this.repo.save(name);
```

TypeORM

```
partial: Partial<Name>;
partial = some_partial_obj;
return this.prisma.name.update({
        data: partial,
        where: { pk }
})
```

Prisma

Outline

Delete

TypeORM has delete(id) and remove(Entity). The former doesn’t allow running hooks.

```
const name = await this.repo.findOne(pk);
return this.repo.remove(name);
// Allow hooks to execute after remove.
const names = await this.repo.find({
        key1: val1,
        key2: val2
});
return this.repo.remove(names);
// Allow hooks to execute after remove.
```

TypeORM

```
this.prisma.name.delete({
        where: { pk }
});
this.prisma.name.deleteMany({
        where: {
                key1: val1,
                key2: val2
        }
});
```

Prisma

Outline

NotFoundException

Prisma has special methods findFirstOrThrow and findUniqueOrThrow that will throw an exception if no record is found. We can catch the Promise to throw NotFoundException from Nest.js.

```
if(!await this.repo.findOne(pk))
        throw new NotFoundException()
```

TypeORM

```
await this.prisma.name.findFirstOrThrow({ where: { pk } })
        .catch(() => { throw new NotFoundException() })

await this.prisma.name.findUniqueOrThrow({ where: { pk } })
        .catch(() => { throw new NotFoundException() })
```

Prisma

It’s important to know that NotFoundException won’t work with non-HTTP controllers. This could be fixed by using exception filters in the other controllers.

Outline

Outline

Logger

We may want to use logging to keep track of events happening in the application. NestJS provides a default Logger class from @nestjs/common with different levels. It also accepts a context string.

```
static log(message: any, context?: string): void;
static warn(message: any, context?: string): void;
static debug(message: any, context?: string): void;
static verbose(message: any, context?: string): void;
static error(message: any, context?: string): void;
```

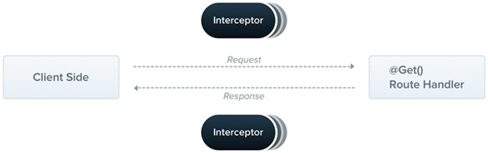
We can select the level of logging to output in the NestFactory.create() method.

```
const app = await NestFactory.create(AppModule, {
        logger: ['error', 'warn'],
});
```

Outline

Interceptors

Interceptors are like middlewares in other frameworks*. In Nestjs, interceptors are placed before each request and after each response from the controller.



They are useful for binding extra logic before and after method execution. They can also transform the returned value or the thrown exception.

* Interceptors are not middlewares. They both behave similarly.

How to

An interceptor is a class that implements NestInterceptor. This means that it needs to implement the intercept() function, which takes an execution context and call handler.

The context allows for access to the request and response properties. The handler is used to forward the request to the next interceptor. The handler also returns an RxSJ Observable, which is useful for intercepting the response.

They can be used with the @UseInterceptors() decorator in a controller or route handler. They can also be used with the app.useGlobalInterceptors() function in the bootstrap function.

Example

A simple interceptor can be as follows.

```typescript
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
        intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
                console.log('Before...');

                const now = Date.now();
                return next
                        .handle()
                        .pipe(
                                tap(() => console.log(`After... ${Date.now() - now}ms`)),
                );
        }
}
```

Outline

RequestIdInterceptor

We can build an interceptor that will modify the request with requestId. This can be helpful if we want to identify each request in the logs.

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

@Injectable()
export class ReqestIdInterceptor implements NestInterceptor {
        intercept(ctx: ExecutionContext, next: CallHandler): Observable<any> {
                const reqId = Math.floor(Math.random() * 999999);
                const reqIdStr = reqId.toString().padStart(6, '0');
                ctx.switchToHttp().getRequest().requestId = reqIdStr;

                return next.handle();
        }
}
```

RequestIdMiddleware

Alternatively, we can use a middleware. This way the requestId is added earlier.

```
import { Request, Response, NextFunction } from 'express';

export function RequestIdMiddleware(req: Request, res: Response, next: NextFunction) {
        const reqId = Math.floor(Math.random() * 999999);
        const reqIdStr = reqId.toString().padStart(6, '0');
        (req as any).requestId = reqIdStr;
        next();
}
```

Note that here we are importing from the express package and thus we are working at a lower level. The middleware should be used like in regular Express:

```
app.use(RequestIdMiddleware); // Inside bootstrap().
```

LoggerService

Finally, we use the requestId in a new CustomLoggerService.

```
import { Inject, Injectable, Logger, ConsoleLogger } from '@nestjs/common';
import { REQUEST } from '@nestjs/core';

@Injectable()
export class CustomLoggerService extends ConsoleLogger {
        constructor(@Inject(REQUEST) private readonly req: any = undefined) {
                super();
                if (this.req) this.context = this.req.requestId;
        }

        log(msg: string): void {
                Logger.log(msg, this.context ? ` [${this.context}]` : '');
        }
}
```

Extending ConsoleLogger allows us to use the string this.context. The context is set with the setContext() method from the parent class.

LoggerService

Note that in CustomLoggerService we are directly injecting the REQUEST from @nestjs/core. This will instantiate a new logger in every request (implicit request scope). This is essential if we want to avoid overwriting the context. However, controllers/services using the new service will implicitly be request-based.

Moreover, the @Inject() decorator is designed to work INSIDE a constructor. Using it in a property won't work properly. To avoid the IDE from complaining about missing arguments when calling the new service, use the "= undefined" in the constructor.

Outline

RequestIdInterceptor

Or, in the case that we are using logger directly in the interceptor, we can ditch the requestId and use logger.setContext() directly.

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

@Injectable()
export class ReqestIdInterceptor implements NestInterceptor {
        constructor(private readonly logger: CustomLoggerService = undefined) {}

        intercept(ctx: ExecutionContext, next: CallHandler): Observable<any> {
                const reqId = Math.floor(Math.random() * 999999);
                const reqIdStr = reqId.toString().padStart(6, '0');
                this.logger.setContext(reqIdStr);

                return next.handle();
        }
}
```

LoggerService

And the service can just be:

```
import { Inject, Injectable, Logger, ConsoleLogger } from '@nestjs/common';

@Injectable({ scope: Scope.REQUEST })
export class CustomLoggerService extends ConsoleLogger {
        log(msg: string): void {
                Logger.log(msg, this.context ? ` [${this.context}]` : '');
        }
}
```

Note that there is no possibility of overwriting the context.

Outline

Bare logger

Alternatively, we can set the CustomLoggerService's scope to Scope.REQUEST. This way we ensure that an instance will be created at the beginning of each request. The service can end up being:

```
import { Inject, Injectable, Logger, ConsoleLogger } from '@nestjs/common';

@Injectable({ scope: Scope.REQUEST })
export class CustomLoggerService extends ConsoleLogger {
        constructor() {
                super();
                this.logger = createWinstonLogger();
                const reqId = Math.floor(Math.random() * 999999);
                this.context = reqId.toString().padStart(6, '0');
        }
}
```

Outline

Outline

Exclude function

Sometimes we want to exclude properties from the response. For example, we don't want passwords being returned to the client. Prisma doesn't have a native way of excluding fields.

To solve this natively, they propose that we use a generic exclude function. This function accepts the instance of the model and an array of properties to exclude:

```
export class NameService {
        exclude<Name, Key extends keyof Name>(name: Name, keys: Key[]): Omit<Name, Key> {
                for (let key of keys) {
                        delete name[key]
                }
                return name
        }
}
```

Outline

Exclude with interceptor

However, another approach with Nest.js is to use a custom interceptor together with a DTO. The interceptor will allow us to add a @Serialize() decorator to a controller method or the controller itself with the DTO as an argument.

Store this class in src/interceptors:

```
import { UseInterceptors, NestInterceptor, ExecutionContext, CallHandler } from "@nestjs/common";
import { Observable } from "rxjs";
import { map } from "rxjs/operators";
import { plainToClass } from "class-transformer";

// Force decorator to use a class.
interface ClassConstructor {
        new (...args: any[]): {}
}

// Shorten decorator into @Serialize(NameDto).
export function Serialize(dto: ClassConstructor) {
        return UseInterceptors(new SerializeInterceptor(dto));
}

...
```

Exclude with interceptor

```
...

class SerializeInterceptor implements NestInterceptor {
        // Accept any dto class.
        constructor(private dto: any) {}

        // Intercept method, middleware in other frameworks.
        intercept(_: ExecutionContext, handler: CallHandler<any>): Observable<any> {
                return handler.handle().pipe(
                        map((data: any) => {
                                // After the request is handled.
                                return plainToClass(this.dto, data, {
                                        // Exclude non-@Expose() attributes in DTO.
                                        excludeExtraneousValues: true
                                });
                        })
                )
        }
}
```

The important bit is **excludeExtraneousValues**, which removes the extra properties. The DTO will need to have the @Expose() decorator in the attributes we want to be exposed. Only exposed attributes will be returned to the client after the request has been handled.

Outline

Outline

register

We will use salted hashes for storing the passwords on the database:

```
async register(email: string, password: string) {
        const users = await this.usersService.find(email);
        if (users.length > 0) throw new BadRequestException('email in use');

        const salt = randomBytes(16).toString('hex');
        const hash = (await scrypt(password, salt, 32)) as Buffer;
        const result = salt + '.' + hash.toString('hex');

        const user = await this.usersService.create(email, result);
        return user;
}
```

authenticate

To check if the password matches, use the following code:

```
async authenticate(email: string, password: string) {
        const [user] = await this.usersService.find(email);
        if (!user) throw new NotFoundException('user not found');

        const [salt, storedHash] = user.password.split('.');
        const hash = (await scrypt(password, salt, 32)) as Buffer;

        if (storedHash !== hash.toString('hex')) throw new BadRequestException('bad password');

        return user;
}
```

AuthService

Structure of the authentication class:

```
import { BadRequestException, Injectable, NotFoundException } from "@nestjs/common";
import { UsersService } from "./users.service";
import { randomBytes, scrypt as _scrypt } from "crypto";
import { promisify } from "util";
const scrypt = promisify(_scrypt);

@Injectable()
export class AuthService {
        constructor(private usersService: UsersService) {}
        async register(email: string, password: string) {...}
        async authenticate(email: string, password: string) {...}
}
```

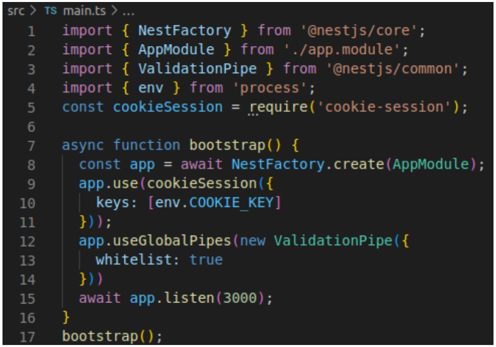
Outline

Session

We need to use cookies to have sessions. We will need to install the following npm packages:

```
$ npm install cookie-session @types/cookie-session
```

cookie-session handles the encryption and decryption of a single session cookie using keys. To add its functionality, we have to modify main.ts:



```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';
import { env } from 'process';
const cookieSession = require('cookie-session');

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(cookieSession({
    keys: [env.COOKIE_KEY]
  }));
  app.useGlobalPipes(new ValidationPipe({
    whitelist: true
  }))
  await app.listen(3000);
}
bootstrap();
```

Session

Due to Nest.js using tsconfig.json, we have to use the require syntax for cookie-session instead. If we have installed Prisma, the .env file will be located in the root directory.

Then, in the controller we will have to import Session from @nestjs/common. The session is used as an argument with:

```
@Session() session: any
```

And then we can access every property of the session cookie in the route handler. We can now assign the id of the user to the session whenever a successful login or registration is done. And we remove this id when the user signs out.

Outline

Custom decorator

We may want to add a custom param decorator with the current user in some route handlers.

We create a new src/users/decorators/current-user.decorator.ts file. Inside we want to export a const with createParamDecorator(), which will turn the variable into a param decorator:

```
import { createParamDecorator, ExecutionContext } from "@nestjs/common";

export const CurrentUser = createParamDecorator(
        (data: any, context: ExecutionContext) => {
                const request = context.switchToHttp().getRequest();
                return request.currentUser;
        }
);
```

Custom decorator

The function uses a callback whose return value is assigned to the argument of the route handler. The callback function inside createParamDecorator() has two parameters:

- The first one is the argument we put whenever we call the decorator. We can change the data type to never so it tells other developers that the decorator doesn't accept any arguments.
- The second one is the execution context. It can be of type HTTP, RPC or WebSocket.

Now we can access the userId through the HTTP request and use it to gather the User from the UserService. However, there's a caveat. We can't use the injector from the decorator to interact with UserService. For it to work we will need an interceptor.

Custom decorator

Since the decorator has access to the request, the new interceptor will query the service for the user and add it to the request object. We create a new src/users/interceptors/current-user.interceptor.ts file:

```
import { CallHandler, ExecutionContext, Injectable, NestInterceptor } from "@nestjs/common";
import { UsersService } from "../users.service";

@Injectable()
export class CurrentUserInterceptor implements NestInterceptor {
        constructor(private userService: UsersService) {}

        async intercept(context: ExecutionContext, handler: CallHandler) {
                const request = context.switchToHttp().getRequest();
                const { userId } = request.session;

                if (userId) {
                        const user = await this.userService.findOne(userId);
                        request.currentUser = user;
                }

                return handler.handle();
        }
}
```

Now the decorator has access to the current user. Note that the request, and the current user, can also be accessed from the controller, but we want to have the decorator functionality.

Custom decorator

To make it work, we have to add the @UseInterceptors(CurrentUserInterceptor) decorator to the controller and add CurrentUserInterceptor to the providers property in the module. However, we may want to avoid using the extra imports and decorator in the controller. This is done with a global-scoped interceptor. In the module we can replace CurrentUserInterceptor with:

```
{
        provide: APP_INTERCEPTOR, // APP_INTERCEPTOR is imported from @nestjs/core.
        useClass: CurrentUserInterceptor
}
```

This will make the interceptor available everywhere in our application. Note that global-scoped interceptors will always execute. So adding too many will be detrimental to the performance.

Outline

Guard

We may want to restrict some routes to signed in users only. This is done with a guard.

A guard is a class that, as its name suggests, protects a route handler. A guard adds requirements in a route. This requierement is analyzed with the canActivate() method. Guards can be global-scoped, controller-scoped or handler-scoped.

We create a src/guards/auth.guard.ts file:

```
import { CanActivate, ExecutionContext } from "@nestjs/common";

export class AuthGuard implements CanActivate {
        canActivate(context: ExecutionContext) {
                const request = context.switchToHttp().getRequest();
                return request.session.userId;
        }
}
```

We can then use the @UseGuards(AuthGuard) decorator when we want to use the guard. Not fulfilling the guard will return an HTTP 403 Forbidden response.

Outline

JWT

We may also want to use a JWT, JSON Web Token, to authenticate the client.

```
$ npm i @nestjs/jwt
```

Now we can import the JwtModule and inject JwtService from @nestjs/jwt where we need it.

```
JwtModule.register({
        global: true,
        secret: process.env.JWT_SECRET,
        signOptions: { expiresIn: '60s' },
})
```

The secret is used to sign the JWT. The signed JWT is later verified in the guard. The global property tells Nest to apply the module globally in our application.

JWT

To sign a JWT, we need to call jwtService.signAsync(token). To verify a JWT, we need to call jwtService.signAsync(jwt). If the signature is not valid, signAsync throws an error. Note that both functions are async.

Now the signing can be used in the authenticate() method and the verifying can be used in a new guard.

```
const authorizationHeader = req.get('Authorization');

if (!authorizationHeader) throw new UnauthorizedException();
if (!authorizationHeader.startsWith("Bearer")) throw new UnauthorizedException();
const parts = authorizationHeader.split(" ");
if (parts.length !== 2) throw new throw new UnauthorizedException();
const token = parts[1];
if (!token) throw new UnauthorizedException();
try {
        var decodedToken = await this.jwtService.verifyAsync(token);
}
catch (error) {
        throw new UnauthorizedException();
}
// Token should now be valid and not expired.
```

Outline

Outline

Unit testing

Unit tests allow us to check that the behaviour of functions inside a class are as expected. Unit tests are defined inside the src directory. The files follow the name.type.spec.ts nomenclature. For example, a UsersController might have a users.controller.spec.ts test file.

Mock dependencies

To avoid having to generate all the dependencies through the injector, we're going to create a testing module in the test file. Before delving into how this module is coded, it's important to see how we can load an arbitrary object faking a dependency. It's done so as follows:

```
{
        provide: LegitimateService,
        useValue: mockLegitimateService
}
```

This will trick the module into loading mockLegitimateService object as if it were a LegitimateService class. Now, fakeLegitimateService needs to implement mock methods that return arbitrary values. This will simulate the execution of methods in LegitimateService.

Mock dependencies

Nest.js uses the Jest package for testing. Jest functions and methods don't need to be imported as they are automatically added to the global scope.

However, using it with Prisma while keeping type-safety is tricky. A solution is to use arrow functions with Promise.resolve(returnVal). But we will have to specify all mocked methods as async/await to return a Promise instead of a PrismaPromise. Moreover, it is recommended to use Partial<LegitimateService> as the type of the object so we have proper type-safety.

```
const mockLegitimateService: Partial<LegitimateService> = {
        methodX: () => Promise.resolve(returnVal)
};
```

Sometimes TypeScript will complain when we resolve an object. A solution to this is to add "as Name" after the object declaration.

Running tests

To run tests:

```
$ npm run test:watch
```

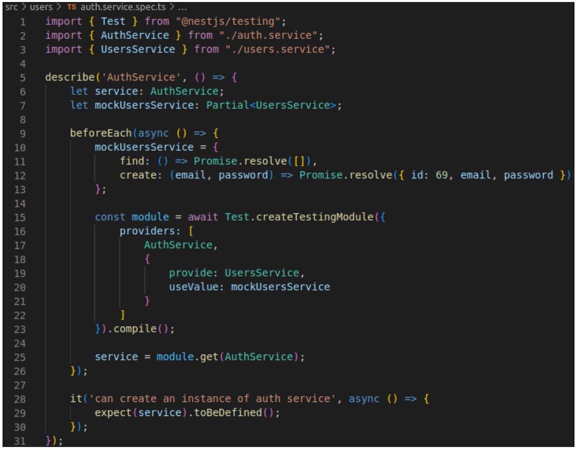
To speed up the tests, replace the test:watch line in the package.json file and change it to:

```
"test:watch": "jest --watch --maxWorkers=1",
```

maxWorkers specifies the number of threads Jest will use.

Understanding the code

The basic layout for a test, with the previous authentication service as an example, is as follows:



src > users > TS auth.service.spec.ts > ...

```typescript
import { Test } from "@nestjs/testing";
import { AuthService } from "./auth.service";
import { UsersService } from "./users.service";

describe('AuthService', () => {
    let service: AuthService;
    let mockUsersService: Partial<UsersService>;

    beforeEach(async () => {
        mockUsersService = {
            find: () => Promise.resolve([]),
            create: (email, password) => Promise.resolve({ id: 69, email, password })
        };

        const module = await Test.createTestingModule({
            providers: [
                AuthService,
                {
                    provide: UsersService,
                    useValue: mockUsersService
                }
            ]
        }).compile();

        service = module.get(AuthService);
    });

    it('can create an instance of auth service', async () => {
        expect(service).toBeDefined();
    });
});
```

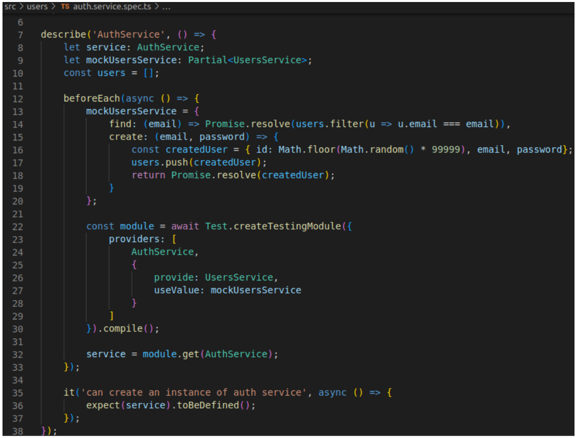
Understanding the code

The describe statement groups tests into a single category. Every it statement will try to test an aspect of our code. beforeEach will be executed before each it statement.

If we need it to, we can redefine the methods of the mock service inside the it statements. Note that the changes will be local because there's a default implementation.

Intelligent mocks

For simple methods in the mock service, we can instead use a global array for storing values. This way we can simulate the database. Note that changes are no longer local and that each it is executed in sequential order.



```typescript

describe('AuthService', () => {
    let service: AuthService;
    let mockUsersService: Partial<UsersService>;
    const users = [];

    beforeEach(async () => {
        mockUsersService = {
            find: (email) => Promise.resolve(users.filter(u => u.email === email)),
            create: (email, password) => {
                const createdUser = { id: Math.floor(Math.random() * 99999), email, password};
                users.push(createdUser);
                return Promise.resolve(createdUser);
            }
        };

        const module = await Test.createTestingModule({
            providers: [
                AuthService,
                {
                    provide: UsersService,
                    useValue: mockUsersService
                }
            ]
        }).compile();

        service = module.get(AuthService);
    });

    it('can create an instance of auth service', async () => {
        expect(service).toBeDefined();
    });
});
```

Outline

E2E testing

E2E, or end-to-end, testing allows us to check that the behaviour to the end user is as expected. Instead of having one unit test for each service/controller, only a single E2E suite (set of tests) is needed. The E2E suite is stored in the tests directory.

Here we are not testing a single service/controller, but the whole application. Hence, we don't need to mock any dependencies. However, we may want to remove some providers. To do so, we simply have to add .overrideProvider(ServiceToRemove).useValue(null) before .compile().

Running tests

To run E2E tests:

```
$ npm run test:e2e
```

An example of the Jest configuration in package.json:

```
"test:e2e": "jest --config ./test/jest-e2e.json --runInBand",
```

- `--config`: Specifies the config file for Jest.
- `--runInBand`: Makes the test run sequentially, i.e. in a single thread.

Running tests

The Jest config file is as follows:

```
{
        "moduleFileExtensions": ["js", "json", "ts"],
        "rootDir": ".",
        "testEnvironment": "node",
        "testRegex": ".e2e.spec.ts$",
        "transform": {
                "^.+\\.(t|j)s$": "ts-jest"
        },
        "moduleNameMapper": {
                "^src/(.*)$": "<rootDir>/../src/$1"
        }
}
```

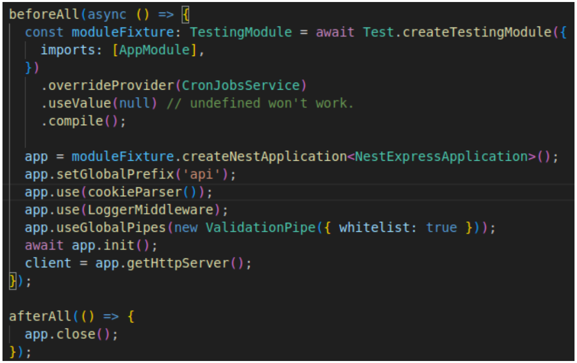
Running tests

Since we are running tests sequentially, it’s interesting to make a test fail if the previous one did. To do this we import fail from the assert package. This way we can check for a condition at the start of the ’it’ callback and call a fail if needed. This is useful if we want to avoid doing unnecessary operations.

Moreover, we may find that some of the calls may take too long and fail. By default Jest has a timeout of 5000 ms. We can change it with jest.setTimeout(), in milliseconds.

Running tests

Beffore programming the tests themselves, we need to write a beforeAll and afterAll functions:



```
beforeAll(async () => {
  const moduleFixture: TestingModule = await Test.createTestingModule({
    imports: [AppModule],
  })
    .overrideProvider(CronJobsService)
    .useValue(null) // undefined won't work.
    .compile();

  app = moduleFixture.createNestApplication<NestExpressApplication>();
  app.setGlobalPrefix('api');
  app.use(cookieParser());
  app.use(LoggerMiddleware);
  app.useGlobalPipes(new ValidationPipe({ whitelist: true }));
  await app.init();
  client = app.getHttpServer();
});

afterAll(() => {
  app.close();
});
```

Running tests

An example of a get call:

```
import * as request from 'supertest';
...
it('as X: get a Y', () => {
        if (!previousObject) fail('missing previousObject');
        request(client).get(`/api/${previousObject.id}/y`).set('Authorization', `Bearer ${jwt}`).expect(200);
});
```

An example of a post call:

```
import * as request from 'supertest';
...
it('as X: do a Z', () => {
        if (!previousObject) fail('missing previousObject');
        const body = {
                name: previousObject.name,
                type: 'animal'
        };
        request(client).post('/api/do-a-z').set('Authorization', `Bearer ${jwt}`).send(body).expect(201);
});
```

Outline

Outline

Request lifecycle

In the introduction there was a simplified diagram of how request are handled. However, the actual request lifecycle is as follows:

1. Incoming request
2. Globally bound middleware
3. Module bound middleware
4. Global guards
5. Controller guards
6. Route guards
7. Global interceptors (pre-controller)
8. Controller interceptors (pre-controller)
9. Route interceptors (pre-controller)
10. Global pipes

Request lifecycle

11. Controller pipes
12. Route pipes
13. Route parameter pipes
14. Controller (method handler)
15. Service (if exists)
16. Route interceptor (post-request)
17. Controller interceptor (post-request)
18. Global interceptor (post-request)
19. Exception filters (route, then controller, then global)
20. Server response

Outline

Error handling

We may want to use filters to handle exceptions in our Nest application. We can add app.useGlobalFilters(new FilterName()); in the bootstrap() method to handle exceptions globally. An example of a filter:

```
export class FilterName implements ExceptionFilter {
        catch(exception: any, host: ArgumentsHost) {
                const res = host.switchToHttp();
                const response = res.getResponse();

                ...handle here...
        }
}
```

Filters in Nest can be implemented on top of Express, which is achieved with the @nestjs/platform-express adapter. Because of this, we can call regular Express methods when we are handling exceptions.

Outline

API service

We can use this service fer fetching data from a remote API. It also provides validation of request and response data with DTOs.

```
import { HttpService } from '@nestjs/axios';
import { BadGatewayException, Injectable } from '@nestjs/common';
import { firstValueFrom } from 'rxjs';
import { validate } from 'class-validator';
import { ReqDto } from './dtos/req.dto';
import { ResDto } from './dtos/res.dto';

@Injectable()
export class ApiService {
        constructor(private readonly httpService: HttpService) {}

        private async _validate(object: any, dto: any) {
                const toValidate = new dto();
                Object.assign(toValidate, object);
                const errors = await validate(toValidate);
                if (errors.length > 0) {
                        const constraints = errors.map(error => error.constraints);
                        const firstError = Object.values(constraints[0])[0];
                        throw new BadGatewayException(firstError);
                }
                return toValidate;
        }
```

API service

```
        private async _call(endpoint: string, body: any = {}, reqDto: any, resDto: any = undefined): Promise<any> {
                await this._validate(body, reqDto);

                try {
                        var { data } = await firstValueFrom(this.httpService.post(endpoint, body));
                } catch (error) {
                        throw new BadGatewayException();
                }

                if (resDto) return this._validate(data, resDto);
                else return data;
        }

        async makeApiCall(body: ReqDto): Promise<ResDto> {
                return this._call('https://example.com/api/endpoint', body, ReqDto, ResDto);
        }
}
```

Outline

Cron job

Cron jobs are scheduled tasks in the operative system. They follow the format:

```
* * * * * command
| | | | |
| | | | day of week
| | | months
| | day of month
| hours
minutes
```

For example: ”45 23 * * 6 /home/oracle/scripts/export_dump.sh” will execute export_dump.sh every Sunday at 23:45.

In Nest there’s a similar approach to this with:

```
$ npm i @nestjs/schedule
```

Cron job

To start using the functionality, we only need to use the @Cron() decorator in a service method. The decorator accepts a string following the "* * * * * *" format (which also allows seconds):

```
* * * * * *
| | | | | |
| | | | | day of week
| | | | months
| | | day of month
| | hours
| minutes
seconds (optional)
```

Be wary that cron jobs fall outside of dependency injection. This means that we need to use static providers aka "regular classes". We need to instantiate each dependency manually.

Outline

Swagger

Swagger is a set of tools that help us define a standardized documentation for the endpoints. Nestjs has a package that enables us to do this.

```
$ npm i @nestjs/swagger
```

We can start using it by adding this to the bootstrap function:

```
const config = new DocumentBuilder()
        .setTitle('Some Title')
        .setVersion('1.2.3')
        .build();
const document = SwaggerModule.createDocument(app, config);
SwaggerModule.setup('some-endpoint', app, document);
```

Then we can use the browser and go to server/some-endpoint. Here we can see the Swagger page. Moreover, we can access the OpenAPI-formatted documentation in both JSON and YAML with server/some-endpoint-json and server/some-endpoint-yaml, respectively.

Swagger

We have a set of decorators at our disposal:

- @ApiTags(): Controller decorator to add tags to all its endpoints.
- @ApiOperation(): Controller method decorator to define extra information for that endpoint.
- @ApiProperty(): DTO property decorator to define extra information.
- @ApiPropertyOptional(): Same as the previous decorator but property is marked as optional.

Swagger

It might be better to allow Nest to automatically generate the DTO documentation. If we used the Nest CLI, we can use the nest-cli.json file.

```json
{
        "$schema": "https://json.schemastore.org/nest-cli",
        "collection": "@nestjs/schematics",
        "sourceRoot": "src",
        "compilerOptions": {
                "deleteOutDir": true,
                "plugins": ["@nestjs/swagger"]
        }
}
```