Дніпровський національний університет імені Олеся Гончара

Факультет прикладної математики та інформаційних технологій

Кафедра інженерії програмного забезпечення та інформаційних технологій

**ЗВІТ**

**з виробничої практики: науково-дослідної**

**Тема:** *Розроблення* *автоматизованої системи обліку вибору навчальних дисциплін кафедри інженерії програмного забезпечення та інформаційних технологій ДНУ*

**Виконала:**

студентка 2 курсу групи ПЗ-23м-2

спеціальності   
121 Інженерія програмного забезпечення

ОПП «Інженерія програмного забезпечення»

Євгенія ЛАШКО

(підпис) (Власне ім’я ПРІЗВИЩЕ)

**Керівник:**

Лілія БОЖУХА

(підпис) (Власне ім’я ПРІЗВИЩЕ)

Національна шкала .

Кількість балів: .

Члени комісії . Тетяна ЄМЕЛ’ЯНЕНКО

(Підпис) (Власне ім’я ПРІЗВИЩЕ)

. Ольга МАЦУГА

(Підпис) (Власне ім’я ПРІЗВИЩЕ)

. Вадим ПАДАЛКО

(Підпис) (Власне ім’я ПРІЗВИЩЕ)

м. Дніпро

2024 р.

**ЗМІСТ**

[ВСТУП 3](#_Toc179843725)

[ПОСТАНОВКА ЗАДАЧІ 5](#_Toc179843726)

[1 ВИБІР ОСНОВНИХ ЗАСОБІВ ДЛЯ ВИКОНАННЯ ЗАВДАННЯ 6](#_Toc179843727)

[1.1 JavaScript і TypeScript 6](#_Toc179843728)

[1.2 Node.js і NestJS 7](#_Toc179843729)

[1.3 Angular Framework 8](#_Toc179843730)

[1.4 MySQL 9](#_Toc179843731)

[2 ВИКОНАННЯ ІНДИВІДУАЛЬНОГО ЗАВДАННЯ 10](#_Toc179843732)

[2.1 Sequelize ORM 11](#_Toc179843733)

[2.2 Основна робота з сутностями. Module, Service, Controller 15](#_Toc179843734)

[2.3 Система безпеки серверної частини 28](#_Toc179843735)

[ВИСНОВКИ 36](#_Toc179843736)

[СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 37](#_Toc179843737)

**ВСТУП**

У сучасному світі інформаційних технологій автоматизація різних процесів є важливою складовою ефективного функціонування будь-якої організації. Вища освіта, зокрема, теж зазнає значного впливу автоматизації, що дозволяє покращувати організацію навчального процесу, оптимізувати роботу співробітників та надавати студентам зручніші інструменти для взаємодії з адміністрацією і викладачами. Одним з таких процесів є вибір навчальних дисциплін студентами, який з кожним роком стає дедалі більш складним та різноманітним через впровадження нових курсів і дисциплін.

На сьогодні у багатьох навчальних закладах існують системи обліку навчальних дисциплін, однак вони не завжди відповідають специфічним вимогам кожної окремої кафедри чи університету. Це спричиняє необхідність розроблення індивідуальних рішень.

Мета виробничої практики полягала у дослідженні процесу вибору навчальних дисциплін, аналізі існуючих рішень та проєктуванні й розробленні частини власної автоматизованої системи, яка забезпечить ефективне управління цим процесом. Основними завданнями було дослідження вимог до такої системи, вибір методів і засобів розроблення, проектування архітектури системи, розроблення системи безпеки для серверної частини додатку.

Забезпечення безпеки серверної частини автоматизованої системи має вирішальне значення для захисту конфіденційної інформації студентів і викладачів. Відсутність належного захисту може призвести до витоку даних, що негативно вплине на приватність та репутацію навчального закладу. У сучасному середовищі кіберзагроз, таких як атаки хакерів, стабільність і безпека системи можуть бути порушені без використання відповідних заходів.

Впровадження такої системи дозволить значно зменшити кількість помилок, пов'язаних з ручною обробкою даних, підвищить прозорість процесу вибору дисциплін, а також зробить його більш зручним і доступним для студентів та співробітників кафедри. Крім того, це сприятиме поліпшенню управлінських процесів на кафедрі, забезпечивши адміністрацію актуальною інформацією.

Важливо відзначити, що розроблення автоматизованої системи є не тільки необхідною для поліпшення організації навчального процесу, але й стає своєрідним індикатором якості освітнього середовища, що надає сучасні й ефективні інструменти для навчання і комунікації. Таким чином, впровадження подібних систем є важливим кроком до цифровізації освіти та підвищення її конкурентоспроможності у сучасних умовах.

**ПОСТАНОВКА ЗАДАЧІ**

Основною метою практики є дослідження предметної області, проєктування автоматизованої системи обліку вибору навчальних дисциплін кафедри інженерії програмного забезпечення та інформаційних технологій ДНУ та розроблення системи безпеки серверної частини додатку.

В ході роботи необхідно виконати наступні задачі:

* дослідити предметну область для розроблення системи;
* дослідити існуючі рішення для вибору дисциплін, проаналізувати дані про минулі вибори дисциплін;
* обрати основні інструменти і методи розроблення;
* спроєктувати архітектуру автоматизованої системи обліку вибору навчальних дисциплін;
* розробити систему безпеки серверної частини додатку.

В результаті роботи на практиці необхідно розробити проєкт автоматизованої системи обліку навчальних дисциплін і частину його функціоналу.

**1 ВИБІР ОСНОВНИХ ЗАСОБІВ ДЛЯ ВИКОНАННЯ ЗАВДАННЯ**

Для реалізації клієнтської частини додатку обрано фреймворк Angular, для реалізації серверної частини –– Node.js і фреймворк NestJS. Обрані мови програмування –– JavaScript і TypeScript. У якості бази даних обрано MySQL.

1.1 JavaScript і TypeScript

Основними мовами програмування обрано JavaScript і TypeScript.

JavaScript (JS) — динамічна, об'єктно-орієнтована прототипна мова програмування. Реалізація стандарту ECMAScript. Найчастіше використовується для створення сценаріїв вебсторінок, що надає можливість на боці клієнта (пристрої кінцевого користувача) взаємодіяти з користувачем, керувати браузером, асинхронно обмінюватися даними з сервером, змінювати структуру та зовнішній вигляд вебсторінки.

JavaScript класифікують як прототипну (підмножина об'єктно-орієнтованої), скриптову мову програмування з динамічною типізацією. Окрім прототипної, JavaScript також частково підтримує інші парадигми програмування (імперативну та частково функціональну) і деякі відповідні архітектурні властивості, зокрема: динамічна та слабка типізація, автоматичне керування пам'яттю, прототипне наслідування, функції як об'єкти першого класу [1].

TypeScript — мова програмування, представлена Microsoft восени 2012; позиціонується як засіб розробки вебзастосунків, що розширює можливості JavaScript.

Розробником мови TypeScript є Андерс Гейлсберг (англ. Anders Hejlsberg), який створив раніше C#, Turbo Pascal і Delphi.

Код експериментального компілятора, котрий транслює код TypeScript у представлення JavaScript, поширюється під ліцензією Apache, розробка ведеться в публічному репозиторії через сервіс CodePlex. Специфікації мови відкриті і опубліковані в межах угоди Open Web Foundation Specification Agreement (OWFa 1.0).

TypeScript є зворотньо сумісним із JavaScript. Фактично, після компіляції програму на TypeScript можна виконувати в будь-якому сучасному браузері або використовувати спільно із серверною платформою Node.js.

Переваги над JavaScript:

* можливість явного визначення типів (статична типізація);
* підтримка використання повноцінних класів (як у традиційних об'єктно-орієнтованих мовах);
* підтримка підключення модулів [2].

1.2 Node.js і NestJS

Для реалізації серверної частини додатку обрано Node.js і фреймворк NestJS.

Node.js — платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript. Засновником платформи є Раян Дал (Ryan Dahl). Якщо раніше JavaScript застосовувався для обробки даних в браузері користувача, то Node.js надав можливість виконувати JavaScript-скрипти на сервері та відправляти користувачеві результат їхнього виконання. Платформа Node.js перетворила JavaScript на мову загального використання з великою спільнотою розробників.

Node.js має наступні властивості:

* асинхронна однопоточна модель виконання запитів;
* неблокуючий ввід/вивід;
* система модулів CommonJS;
* рушій JavaScript Google V8.

Для керування модулями використовується пакетний менеджер npm (node package manager) [3].

Nest (NestJS) — це платформа для створення ефективних, масштабованих програм Node.js на стороні сервера.

Він використовує JavaScript, створений і повністю підтримує TypeScript (все ще дозволяє розробникам кодувати на чистому JavaScript) і поєднує елементи ООП (об’єктно-орієнтоване програмування), ФП (функціональне програмування) і ФРП (функціональне реактивне програмування).

Nest надає готову архітектуру додатків, яка дозволяє розробникам та командам створювати легко тестовані, масштабовані, слабозв'язані та прості в обслуговуванні додатки. [4].

## 1.3 Angular Framework

Angular –– це веб-фреймворк, розроблений компанією Google, який використовується для створення динамічних односторінкових додатків (Single Page Applications, SPA). Він дозволяє розробникам будувати високопродуктивні веб-додатки з використанням мови програмування TypeScript.

Основними особливостями Angular є:

1. Компонентна архітектура: Angular побудований на основі компонентної моделі, де веб-додаток розбивається на невеликі, самостійні компоненти. Кожен компонент містить свій шаблон, стилі і логіку, що дозволяє розробникам легко організовувати та управляти кодом.
2. Двостороннє зв’язування даних: Angular надає механізм двостороннього зв’язування даних, що дозволяє автоматично синхронізувати дані між компонентами та їх шаблонами. Це полегшує маніпулювання даними і оновлення їх в реальному часі.
3. Залежності та ін’єкція залежностей: Angular має вбудовану систему ін’єкції залежностей, що дозволяє легко управляти залежностями між компонентами. Це спрощує розробку, тестування і підтримку коду.
4. Роутинг: Angular надає механізм маршрутизації, що дозволяє створювати багатосторінкові додатки. З його допомогою можна визначити, який компонент відображати для кожного URL-шляху.
5. Розширюваність: Angular має широкий набір функціональності, яку можна розширити за допомогою сторонніх модулів. Існує велика кількість сторонніх бібліотек і модулів, які розширюють можливості фреймворка.

Angular є потужним і популярним інструментом для розробки веб-додатків, особливо для великих та складних проектів. Він надає структуровану платформу для розробки і допомагає забезпечити швидку та ефективну роботу з фронтендом [5].

## 1.4 MySQL

MySQL — вільна система керування реляційними базами даних, яка була розроблена компанією «ТсХ» для підвищення швидкодії обробки великих баз даних. Ця система керування базами даних (СКБД) з відкритим кодом була створена як альтернатива комерційним системам. MySQL з самого початку була дуже схожою на mSQL, проте з часом вона все розширювалася і зараз MySQL — одна з найпоширеніших систем керування базами даних. Вона використовується, в першу чергу, для створення динамічних вебсторінок, оскільки має чудову підтримку з боку різноманітних мов програмування [6].

**2 ВИКОНАННЯ ІНДИВІДУАЛЬНОГО ЗАВДАННЯ**

Nest є зручним фреймворком для побудови серверних застосунків насамперед тому, що має адаптовані бібліотеки для різних потреб. Наприклад, @nestjs/swagger –– зручний набір інструментів для опису, тестування і використання API. Конфігурація Swagger та інших бібліотек, які потрібні для старту застосунку, проводиться в основній стартовій функції застосунку bootstrap() і наведена у лістингу 2.1.

Лістинг 2.1 – bootstrap()

async function bootstrap() {  
 const env = *process*.env.NODE\_ENV;  
 const envFilePath = path.resolve(*\_\_dirname*, `../${env}.env`);  
 if (fs.existsSync(envFilePath)) {  
 dotenv.config({ path: envFilePath });  
 } else {  
 *console*.warn(  
 `Environment file ${envFilePath} not found, falling back to defaults.`,  
 );  
 }  
 const app = await *NestFactory*.create(AppModule);  
 app.enableCors();  
 app.useGlobalPipes(new ValidationPipe({ transform: true }));  
 const config = new DocumentBuilder()  
 .setTitle('DNUChoice API')  
 .setDescription('DNUChoice API on NestJS.')  
 .setVersion('1.0')  
 .addBearerAuth(  
 {  
 type: 'http',  
 scheme: 'bearer',  
 bearerFormat: 'JWT',  
 description:  
 'eyJhbGciOiJIUzI1NiJ9.ZXhhbXBsZmZlQGV4YW1wbGUuY29t.9AG7RC9\_qiSg284LNs\_HGZU0fMJ4Bz6v89qlkpE-OU0',  
 name: 'Authorization',  
 in: 'header',  
 },  
 'bearer',  
 )  
 .build();  
 const document = SwaggerModule.*createDocument*(app, config);  
 SwaggerModule.*setup*('docs', app, document, {  
 swaggerOptions: {  
 security: [{ bearer: [] }],  
 },});  
 useContainer(app.select(AppModule), { fallbackOnErrors: true });  
 await app.listen(3000);}

2.1 Sequelize ORM

Для зручності роботи з базою даних було прийнято рішення про використання Sequelize ORM, яка дозволяє не тільки поліпшити роботу, але і запобігає SQL-ін’єкціям і служить додатковим засобом захисту даних застосунку.

Sequelize надає можливість керувати структурою і вмістом бази даних за допомогою міграцій і сідів. Приклад міграції для створення таблиці користувалів наведено у лістингу 2.2. Приклад сіда наведено у лістингу 2.3.

Лістинг 2.2 – Міграція таблиці users

/\* eslint-disable @typescript-eslint/no-unused-vars \*/  
'use strict';  
*/\*\* @type {import('sequelize-cli').Migration} \*/  
module*.exports = {  
 async up(queryInterface, Sequelize) {  
 await queryInterface.sequelize.query(  
 `CREATE TABLE IF NOT EXISTS users (  
 id INT NOT NULL PRIMARY KEY AUTO\_INCREMENT,  
 name VARCHAR(500) NOT NULL,  
 email VARCHAR(255) UNIQUE NOT NULL,  
 password VARCHAR(255) NOT NULL,  
 role TINYINT NOT NULL COMMENT '1 - admin, 2 - teacher, 3 - student',  
 groupId INT DEFAULT NULL,  
 createdAt TIMESTAMP NOT NULL,  
 updatedAt TIMESTAMP NOT NULL,  
 CONSTRAINT usersGroupId FOREIGN KEY (groupId) REFERENCES academicGroups(id)  
 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_0900\_ai\_ci`,  
 );  
 },  
 async down(queryInterface, Sequelize) {  
 await queryInterface.sequelize.query('DROP TABLE IF EXISTS users');  
 },  
};

Лістинг 2.3 – Сід таблиці users

/\* eslint-disable @typescript-eslint/no-unused-vars \*/  
'use strict';  
*/\*\* @type {import('sequelize-cli').Migration} \*/  
module*.exports = {  
 async up(queryInterface, Sequelize) {  
 return await queryInterface.sequelize.query(  
 `INSERT INTO users VALUES   
(1,'Admin','admin@example.com','$2b$10$Cn7YKwzqh0QQpS3AG88fGenKIBjGuJouMz.BEx.bklS5Z5q6kI0KK',1,NULL,'2024-09-03 19:32:40','2024-09-03 19:54:05'), (2,'Teacher','teacher@example.com','$2b$10$eJ.XhTQhXwCZpbwcfXdMCO7mBINTC9oTtFR04ZxrIF8fvtHtx0eiC',2,NULL,'2024-09-03 19:38:37','2024-09-03 19:38:37'), (3,'Student','student@example.com','$2b$10$SaJbArA5EmNewIA1Af0bwOJxFTgL2.LkDruOoDWhPxOBOL38fNDMy',3,1,'2024-09-03 19:38:47','2024-09-03 19:38:47')`,  
 );  
 },  
 async down(queryInterface, Sequelize) {  
 return await queryInterface.sequelize.query('DELETE FROM users');  
 },  
};

Архітектура Nest-застосунків побудована на сисемі модулів, сервісів, контролерів і моделей. Основною «корінною» сутністю є сутність App, в модулі сутності App і головному файлі main потрібно проводити конфігурацію основних бібліотек, які будуть використовуватись більшістю підсутностей. Конфігурація Sequelize наведена у лістингу 2.4.

Лістинг 2.4 – Конфігурація Sequelize

@Module({  
 imports: [  
 ConfigModule.*forRoot*({  
 envFilePath: `${*process*.env.NODE\_ENV || 'dev'}.env`,  
 isGlobal: true,  
 }),  
 SequelizeModule.*forRoot*({  
 dialect: 'mysql',  
 host: *process*.env.DB\_HOST,  
 port: +*process*.env.DB\_PORT,  
 username: process.env.DB\_USERNAME,  
 password: process.env.DB\_PASSWORD,  
 database: process.env.DB\_DATABASE,  
 models: [User, Group],  
 }),  
 UserModule,  
 RedisCacheModule,  
 SessionModule,  
 GroupModule,  
 ],  
 ...

})  
export class AppModule {...}

У Sequelize модель представляє собою абстракцію таблиці бази даних. Вона визначає структуру даних, які будуть зберігатися в таблиці, та взаємодію з ними через програмний інтерфейс.

Модель в Sequelize описує поля таблиці (колонки), їхні типи даних, а також відносини між іншими таблицями (моделями). Це дозволяє працювати з даними через об'єкти в коді замість прямого написання SQL-запитів. Модель включає визначення атрибутів (поля), методів для створення, читання, оновлення та видалення (CRUD-операції), а також правила валідації та поведінки. Приклад моделі User представлений у лістингу 2.5.

Лістинг 2.5 – Модель User

@Table({  
 tableName: 'users',  
 timestamps: true,  
})  
export class User extends Model {  
 @ApiProperty({  
 description: 'User identifier',  
 nullable: false,  
 example: 1,  
 type: 'integer',  
 })  
 @PrimaryKey  
 @AutoIncrement  
 @Column(*DataType*.*INTEGER*)  
 id: number;  
 @ApiProperty({  
 description: 'Full name',  
 nullable: false,  
 example: 'John Doe',  
 type: 'string',  
 })  
 @AllowNull(false)  
 @Column(*DataType*.*STRING*(500))  
 name: string;  
 @ApiProperty({  
 description: 'Email',  
 nullable: false,  
 example: 'example@example.com',  
 type: 'string',  
 })  
 @Unique  
 @AllowNull(false)  
 @Column({  
 type: *DataType*.*STRING*(255),  
 validate: {  
 isEmail: true,  
 },  
 })  
 email: string;  
 @ApiProperty({  
 description: 'Password',  
 nullable: false,  
 example: 'password',  
 type: 'string',  
 })  
 @AllowNull(false)  
 @Column(*DataType*.*STRING*(255))  
 password: string;  
 @ApiProperty({  
 description: 'Role',  
 nullable: false,  
 example: 1,  
 type: 'integer',  
 })  
 @AllowNull(false)  
 @Column(*DataType*.*TINYINT*)  
 role: number;  
 @ApiProperty({  
 description: 'Group identifier',  
 nullable: true,  
 example: 1,  
 type: 'integer',  
 })  
 @AllowNull(true)  
 @Column(*DataType*.*INTEGER*)  
 @ForeignKey(() => Group)  
 groupId: number;  
 @BelongsTo(() => Group)  
 group: Group;  
 @ApiProperty({  
 description: 'Created at',  
 nullable: false,  
 example: '2024-07-03T19:32:40.000Z',  
 type: 'timestamp',  
 })  
 @CreatedAt  
 @AllowNull(false)  
 @Column({ type: *DataType*.*DATE* })  
 createdAt: Date;  
 @ApiProperty({  
 description: 'Updated at',  
 nullable: false,  
 example: '2024-07-03T19:32:40.000Z',  
 type: 'timestamp',  
 })  
 @UpdatedAt  
 @AllowNull(false)  
 @Column({ type: *DataType*.*DATE* })  
 updatedAt: Date;  
}

Скоупи в Sequelize — це механізм, що дозволяє визначати та повторно використовувати певні набори умов для запитів до моделі. Вони спрощують написання запитів, роблячи код чистішим і більш підтримуваним, особливо якщо одні й ті самі фільтри або параметри використовуються багаторазово. Скоупи визначаються всередині моделі і можуть містити умови, що стосуються вибору полів, фільтрів, обмежень, зв'язків тощо. Приклад деяких скоупів для моделі User наведено у лістингу 2.6.

Лістинг 2.6 – Скоупи моделі User

@Scopes(() => ({  
 byId: (id: number) => ({  
 where: { id },  
 }),  
 byPage: (offset: number) => ({  
 limit: 10,  
 offset,  
 }),  
 excludesId: (id: number) => ({  
 where: { id: { [*Op*.ne]: id } },  
 }),  
 byNameOrEmail: (value: string) => ({  
 where: {  
 [*Op*.or]: {  
 name: {  
 [*Op*.like]: `%${value}%`,  
 },  
 email: {  
 [*Op*.like]: `%${value}%`,  
 },  
 },  
 },  
 }),  
 byRole: (role: number) => ({  
 where: { role },  
 }),  
 withGroup: () => ({  
 include: [  
 {  
 model: Group,  
 subQuery: false,  
 },  
 ],  
 }),  
}))

2.2 Основна робота з сутностями. Module, Service, Controller

У фреймворку NestJS основними будівельними блоками є модулі, сервіси, контролери та DTO. Вони допомагають організувати та структурувати код, що дозволяє створювати масштабовані та підтримувані серверні додатки.

Модуль в NestJS є контейнером для пов'язаних компонентів (сервісів, контролерів, провайдерів), що дозволяє організувати код у логічні блоки. Кожен модуль інкапсулює певний функціонал і може імпортувати інші модулі для використання їхніх сервісів або провайдерів. Основний модуль додатку це AppModule, але в додатку може бути багато модулів для різних частин системи. Приклад модуля для роботи з сутністю User та ендпоінтами, що пов’язані з нею, наведено у лістингу 2.7.

Лістинг 2.7 – Модуль UserModule

@Module({  
 imports: [SequelizeModule.*forFeature*([User]), RedisCacheModule],  
 controllers: [UserController],  
 providers: [UserService, IsUnique],  
 exports: [UserService],  
})  
export class UserModule {

...

}

Модуль UserModule групує компоненти, пов'язані з користувачами: модель User, контролер UserController, і сервіс UserService. Він імпортує модуль Sequelize для роботи з базою даних і Redis для кешування, а також надає механізми перевірки унікальності полів через провайдер IsUnique. Експорт сервісу UserService дозволяє іншим модулям використовувати його функціонал.

SequelizeModule.forFeature([User]) –– це інжекція модуля Sequelize для моделі User. Це дозволяє модулю працювати з таблицею користувачів у базі даних через ORM Sequelize. Метод forFeature реєструє модель User для використання всередині цього модуля.

Сервіси в NestJS відповідають за бізнес-логіку додатку. Це клас, який можна інжектувати в інші компоненти (наприклад, контролери чи інші сервіси). Сервіси обробляють дані, виконують запити до бази даних, або здійснюють інші операції, необхідні для роботи додатку. Вони використовуються для розділення бізнес-логіки і контролерів, щоб зробити код чистішим і більш підтримуваним. Приклад сервісу UserService наведено у лістингу 2.8.

Лістинг 2.8 – Сервіс UserService

@Injectable()  
export class UserService {  
 constructor(  
 @InjectModel(User)  
 private userModel: typeof User,  
 ) {}  
 async getCount(params: GetAllUsersSchema, id: number): Promise<number> {  
 try {  
 const scopes: ScopeOptions[] = [{ method: ['excludesId', id] }];  
 if (params.query) {  
 scopes.push({ method: ['byNameOrEmail', params.query] });  
 }  
 return this.userModel.*scope*(scopes).*count*();  
 } catch (error) {  
 throw new BadRequestException(error);  
 }  
 }  
 async findAll(params: GetAllUsersSchema, id: number): Promise<User[]> {  
 try {  
 const scopes: any[] = [  
 { method: ['byPage', params.offset] },  
 { method: ['excludesId', id] },  
 'withGroup',  
 ];  
 if (params.query) {  
 scopes.push({ method: ['byNameOrEmail', params.query] });  
 }  
 return await this.userModel.*scope*(scopes).*findAll*();  
 } catch (error) {  
 throw new BadRequestException(error);  
 }  
 }  
 async findById(id: number, scopes: any[]): Promise<User> {  
 const user = await this.userModel.*scope*(scopes).*findByPk*(id);  
 if (!user) {  
 throw new NotFoundException();  
 }  
 return user;  
 }  
 async create(data: any): Promise<User> {  
 const saltOrRounds = 10;  
 const newData = {  
 ...data,  
 password: await bcrypt.hash(data.password, saltOrRounds),  
 };  
 try {  
 const user = await this.userModel.*create*(newData);  
  
 return await this.findById(user.id, ['withGroup']);  
 } catch (error) {  
 throw new BadRequestException(error);  
 }  
 }  
 async checkUniqueValues(  
 id: number,  
 property: string,  
 value: string,  
 ): Promise<User> {  
 return this.userModel.*findOne*({  
 where: { [property]: value, id: { [*Op*.ne]: id } },  
 });  
 }  
 async update(id: number, data: any): Promise<User> {  
 const saltOrRounds = 10;  
 const newData = {  
 ...data,  
 };  
 const user = await this.findById(id, []);  
 try {  
 if (await this.checkUniqueValues(id, 'email', newData.email)) {  
 throw new BadRequestException('Email must be unique!');  
 }  
 if (  
 newData.phone &&  
 (await this.checkUniqueValues(id, 'phone', newData.phone))  
 ) {  
 throw new BadRequestException('Phone number must be unique!');  
 }  
 const isMatch = await bcrypt.compare(newData.password, user.password);  
 let password = user.password;  
 if (!isMatch) {  
 password = await bcrypt.hash(newData.password, saltOrRounds);  
 }  
 newData.password = password;  
 return await user.update(newData);  
 } catch (error) {  
 throw new BadRequestException(error);  
 }  
 }  
 async findUserByEmailAndPassword(data: CreateSessionSchema): Promise<User> {  
 const user = await this.userModel.*findOne*({  
 where: {  
 email: data.email,  
 },  
 });  
 if (!user) {  
 throw new UnprocessableEntityException('Invalid credentials!');  
 }  
 const isMatch = await bcrypt.compare(data.password, user.password);  
 if (!isMatch) {  
 throw new UnprocessableEntityException('Invalid credentials!');  
 }  
 return user;  
 }  
 async delete(user: User) {  
 await user.destroy();  
 }  
}

@Injectable() позначає клас як сервіс, що можна інжектувати в інші частини додатку. @InjectModel(User) інжектує модель User, що дозволяє взаємодіяти з таблицею користувачів у базі даних.

Метод getCount(params: GetAllUsersSchema, id: number) повертає кількість користувачів, виключаючи користувача з певним id. Якщо передано параметр пошуку query, він фільтрує користувачів за ім'ям або електронною поштою.

Метод findAll(params: GetAllUsersSchema, id: number) повертає всіх користувачів з бази даних з урахуванням фільтрів. Виключає користувача з переданим id і підтримує пошук за ім'ям або email.

Метод findById(id: number, scopes: any[]) шукає користувача за унікальним id. Якщо користувача не знайдено, викидає помилку NotFoundException.

Метод create(data: any) створює нового користувача з хешованим паролем. Використовує bcrypt для хешування пароля перед збереженням. Після створення повертає дані користувача з додатковою інформацією (групою).

Метод checkUniqueValues(id: number, property: string, value: string) перевіряє, чи існують користувачі з унікальними значеннями певного поля (наприклад, email або телефон) з іншим id. Це корисно для перевірки унікальності даних під час оновлення профілю.

Метод update(id: number, data: any) оновлює дані користувача. Перед оновленням перевіряє унікальність email і телефону, а також хешує пароль, якщо він змінився. Якщо пароль не змінювався, залишає його без змін.

Метод findUserByEmailAndPassword(data: CreateSessionSchema) шукає користувача за електронною поштою та перевіряє пароль. Якщо дані не збігаються, кидає помилку UnprocessableEntityException з повідомленням "Invalid credentials!".

Метод delete(user: User) видаляє користувача з бази даних.

Для обробки помилок використовуються винятки NestJS: BadRequestException, NotFoundException, UnprocessableEntityException та ін. Вони викидаються у разі виникнення помилок, таких як невалідні дані, відсутність користувача або неправильні облікові дані.

Бібліотека bcrypt використовується для хешування паролів під час створення та оновлення користувачів, а також для порівняння паролів при вході користувачів у систему.

Контролери відповідають за обробку HTTP-запитів та повернення відповідей. Вони отримують запити від клієнтів, делегують виконання бізнес-логіки сервісам, і формують відповіді. Кожен метод контролера відповідає за обробку певного маршруту або HTTP-методу, такого як GET, POST, PUT, тощо.

DTO в NestJS є об'єктом для передачі даних. Це клас, що визначає структуру даних, які передаються між різними частинами додатку, наприклад, між клієнтом і сервером. DTO використовується для чіткої типізації та валідації вхідних даних. Приклад контролера UserController наведено у лістингу 2.9.

Лістинг 2.9 – Контроллер UserController

@ApiTags('Users')  
@Controller('/api/users')  
export class UserController {  
 constructor(private readonly userService: UserService) {}  
 @ApiOperation({ summary: 'Gets all users' })  
 @ApiQuery({  
 name: 'offset',  
 description: 'Offset',  
 example: '0',  
 type: 'integer',  
 })  
 @ApiResponse({  
 status: HttpStatus.*OK*,  
 description: 'Success',  
 type: UsersDto,  
 })  
 @ApiResponse({ status: HttpStatus.*BAD\_REQUEST*, description: 'Bad Request' })  
 @ApiBearerAuth()  
 @UseGuards(JwtAuthGuard)  
 @Get()  
 async getAll(  
 @Query() query: GetAllUsersSchema,  
 @Request() req: ExpressRequest,  
 ): Promise<UsersDto> {  
 const payload = req.user as IJwtPayload;  
 let count = 0;  
 let users = [];  
 count = await this.userService.getCount(query, payload.id);  
 if (count) {  
 users = await this.userService.findAll(query, payload.id);  
 }  
 return new UsersDto(count, users);  
 }  
 @ApiOperation({ summary: 'Gets authorized user' })  
 @ApiResponse({  
 status: HttpStatus.*OK*,  
 description: 'Success',  
 type: UserDto,  
 })  
 @ApiResponse({ status: HttpStatus.*UNAUTHORIZED*, description: 'Unauthorized' })  
 @Get('/me')  
 @ApiBearerAuth()  
 @UseGuards(JwtAuthGuard)  
 async getAuthorizedUser(@Request() req: ExpressRequest) {  
 const payload = req.user as IJwtPayload;  
 const user = await this.userService.findById(payload.id, ['withGroup']);  
 const data = {  
 token: payload.token,  
 };  
 *Object*.assign(data, user.toJSON());  
 return new UserDto(data as IUser);  
 }  
 @ApiOperation({ summary: 'Gets a user with specified id' })  
 @ApiParam({  
 name: 'id',  
 required: true,  
 description: 'User identifier',  
 type: 'integer',  
 example: 1,  
 })  
 @ApiResponse({  
 status: HttpStatus.*OK*,  
 description: 'Success',  
 type: UserDto,  
 })  
 @ApiResponse({ status: HttpStatus.*BAD\_REQUEST*, description: 'Bad Request' })  
 @ApiResponse({ status: HttpStatus.*NOT\_FOUND*, description: 'Not Found' })  
 @ApiBearerAuth()  
 @UseGuards(JwtAuthGuard)  
 @Get(':id')  
 async getById(@Param() params: NumberIdSchema): Promise<UserDto> {  
 const result = await this.userService.findById(params.id, ['withGroup']);  
 return new UserDto(result as IUser);  
 }  
 @ApiOperation({ summary: 'Creates new user' })  
 @ApiBody({  
 description: 'User data',  
 type: CreateUserSchema,  
 })  
 @ApiResponse({  
 status: HttpStatus.*CREATED*,  
 description: 'Success',  
 type: UserDto,  
 })  
 @ApiResponse({ status: HttpStatus.*BAD\_REQUEST*, description: 'Bad Request' })  
 @ApiBearerAuth()  
 @UseGuards(JwtAuthGuard)  
 @Post()  
 async addUser(@Body() body: CreateUserSchema): Promise<UserDto> {  
 const result = await this.userService.create(body);  
 return new UserDto(result);  
 }  
 @ApiOperation({ summary: 'Updates a user with specified id' })  
 @ApiParam({  
 name: 'id',  
 required: true,  
 description: 'User identifier',  
 type: 'integer',  
 example: 1,  
 })  
 @ApiBody({  
 description: 'User data',  
 type: UpdateUserSchema,  
 })  
 @ApiResponse({ status: HttpStatus.*OK*, description: 'Success', type: UserDto })  
 @ApiResponse({ status: HttpStatus.*BAD\_REQUEST*, description: 'Bad Request' })  
 @ApiResponse({ status: HttpStatus.*NOT\_FOUND*, description: 'Not Found' })  
 @ApiBearerAuth()  
 @UseGuards(JwtAuthGuard)  
 @Put(':id')  
 async updateUser(  
 @Param() params: NumberIdSchema,  
 @Body() body: UpdateUserSchema,  
 ): Promise<UserDto> {  
 await this.userService.update(params.id, body);  
 const result = await this.userService.findById(params.id, ['withGroup']);  
 return new UserDto(result);  
 }  
 @ApiOperation({ summary: 'Delete user by id' })  
 @ApiParam({  
 name: 'id',  
 required: true,  
 description: 'User identifier',  
 type: 'integer',  
 example: 1,  
 })  
 @ApiResponse({  
 status: HttpStatus.*OK*,  
 description: 'Success',  
 })  
 @ApiResponse({ status: HttpStatus.*NOT\_FOUND*, description: 'Not Found' })  
 @ApiBearerAuth()  
 @UseGuards(JwtAuthGuard)  
 @Delete(':id')  
 async deleteById(@Param() params: NumberIdSchema): Promise<void> {  
 const result = await this.userService.findById(params.id, []);  
 if (result) {  
 await this.userService.delete(result);  
 }  
 }  
}

Контролер UserController виступає як проміжний шар між HTTP-запитами та бізнес-логікою, реалізованою в сервісі UserService.

@ApiTags('Users') позначає контролер тегом Users, що використовується для документування API через Swagger. Це допомагає організувати та групувати методи контролера в API-документації.

@Controller('/api/users') задає базовий маршрут для всіх методів у цьому контролері. Усі маршрути будуть мати префікс /api/users.

Get() метод для отримання всіх користувачів. Використовує декоратори для опису операції (@ApiOperation), параметрів запиту (@ApiQuery), та відповідей (@ApiResponse). Метод обробляє запити на /api/users, отримує параметри для фільтрації користувачів та повертає об'єкт UsersDto, що містить кількість користувачів та масив користувачів.

@Get('/me') метод для отримання даних авторизованого користувача. Повертає UserDto з інформацією про користувача, включаючи токен.

@Get(':id') метод для отримання користувача за його ідентифікатором. Використовує @ApiParam для опису параметра id у маршруті. Повертає UserDto з даними користувача.

@Post() метод для створення нового користувача. Приймає дані користувача через @Body(), документує запит за допомогою @ApiBody, і повертає UserDto з даними нового користувача.

@Put(':id') метод для оновлення користувача за його ідентифікатором. Приймає оновлені дані через @Body() і повертає UserDto з оновленими даними користувача.

@Delete(':id') метод для видалення користувача за його ідентифікатором. Перевіряє, чи існує користувач, і, якщо так, видаляє його.

Кожен метод контролера документує можливі відповіді за допомогою @ApiResponse, описуючи статуси відповіді (наприклад, 200 - успішно, 201 - створено, 400 - невалідний запит, 404 - не знайдено, 401 - неавторизований).

Приклад роботи зі Swagger за даними ендпоінтами наведена на рисунках 2.1–2.4.

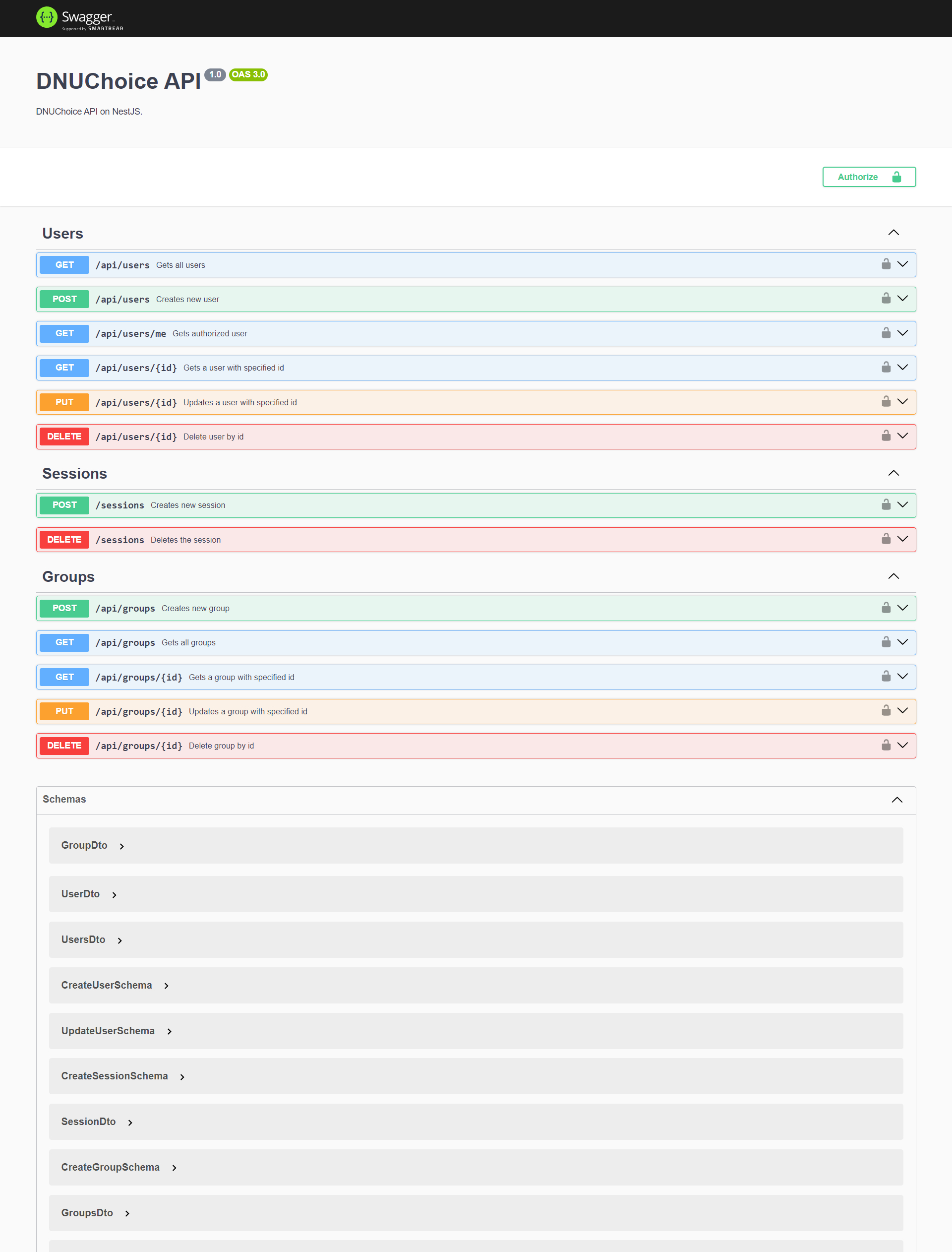
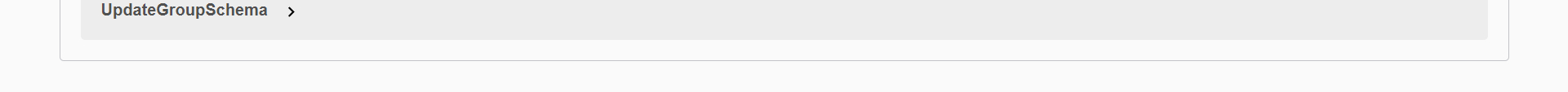
 

Рисунок 1 – Згенерована документація Swagger

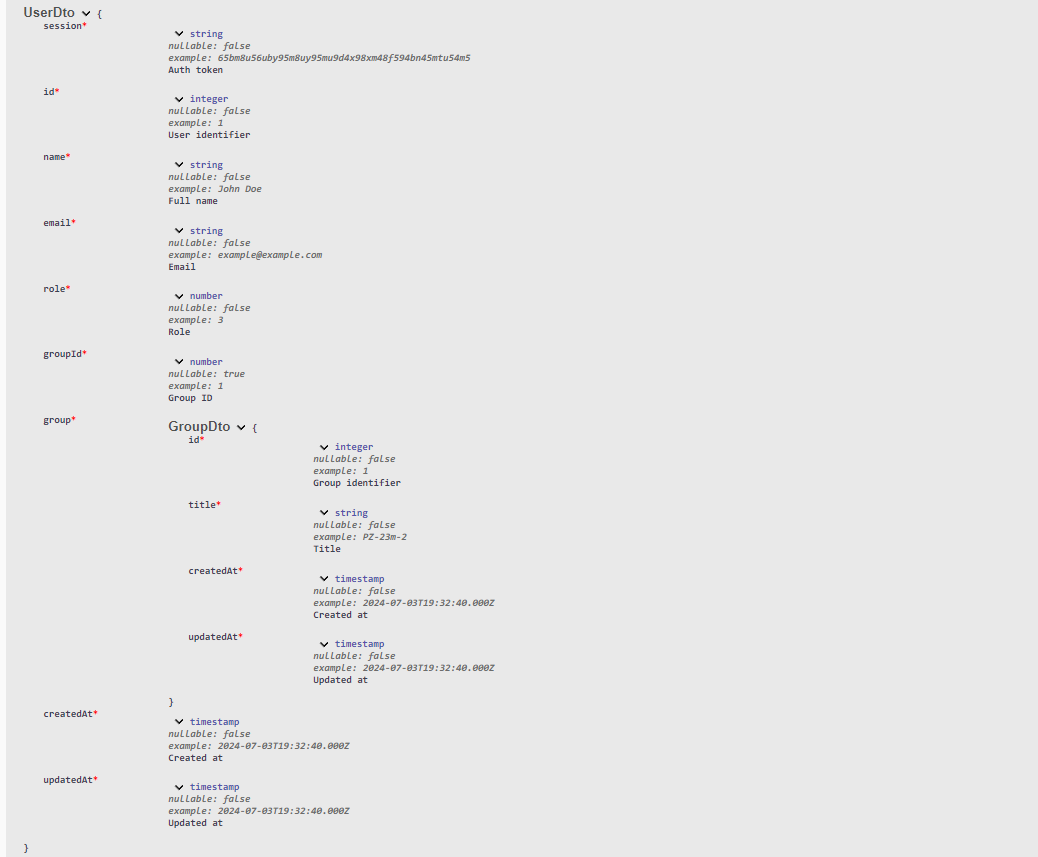
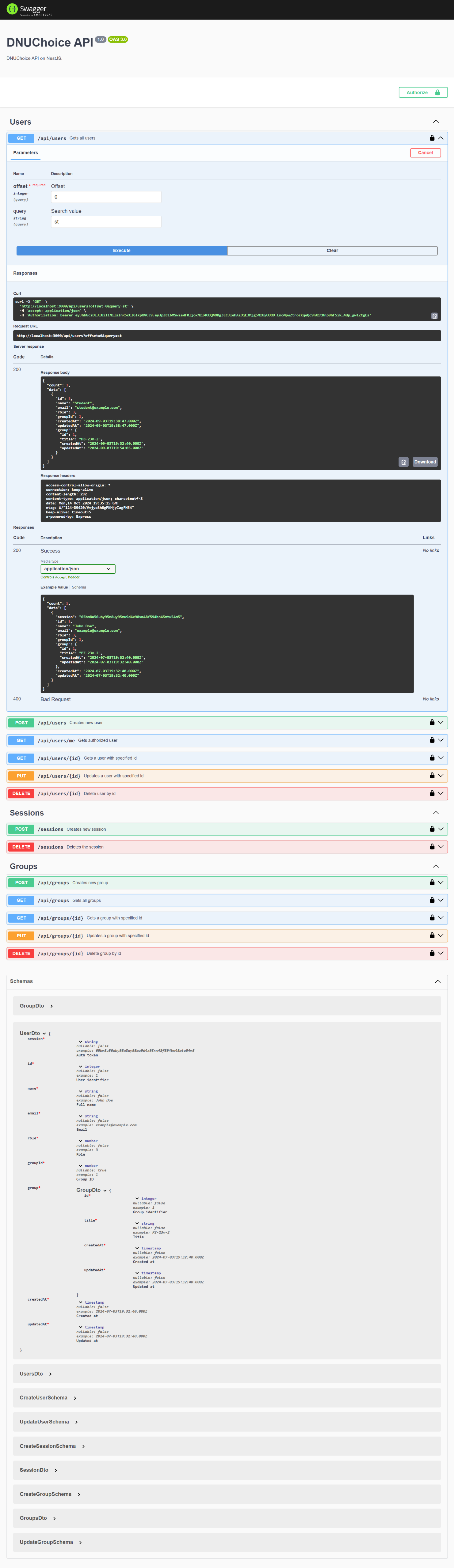
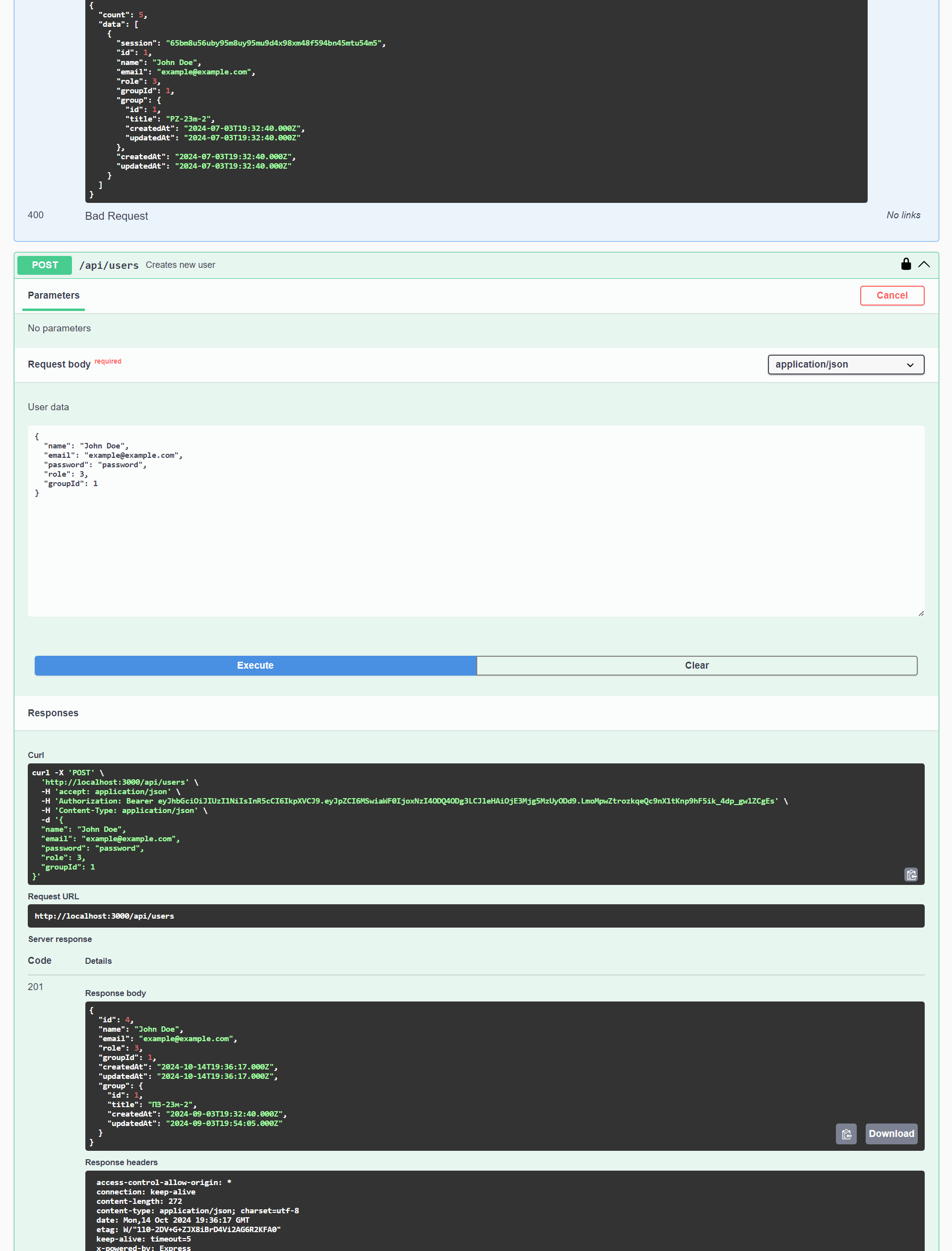


Рисунок 2 – Cхема UserDto

Рисунок 3 – Ендпоінт повернення масиву користувачів



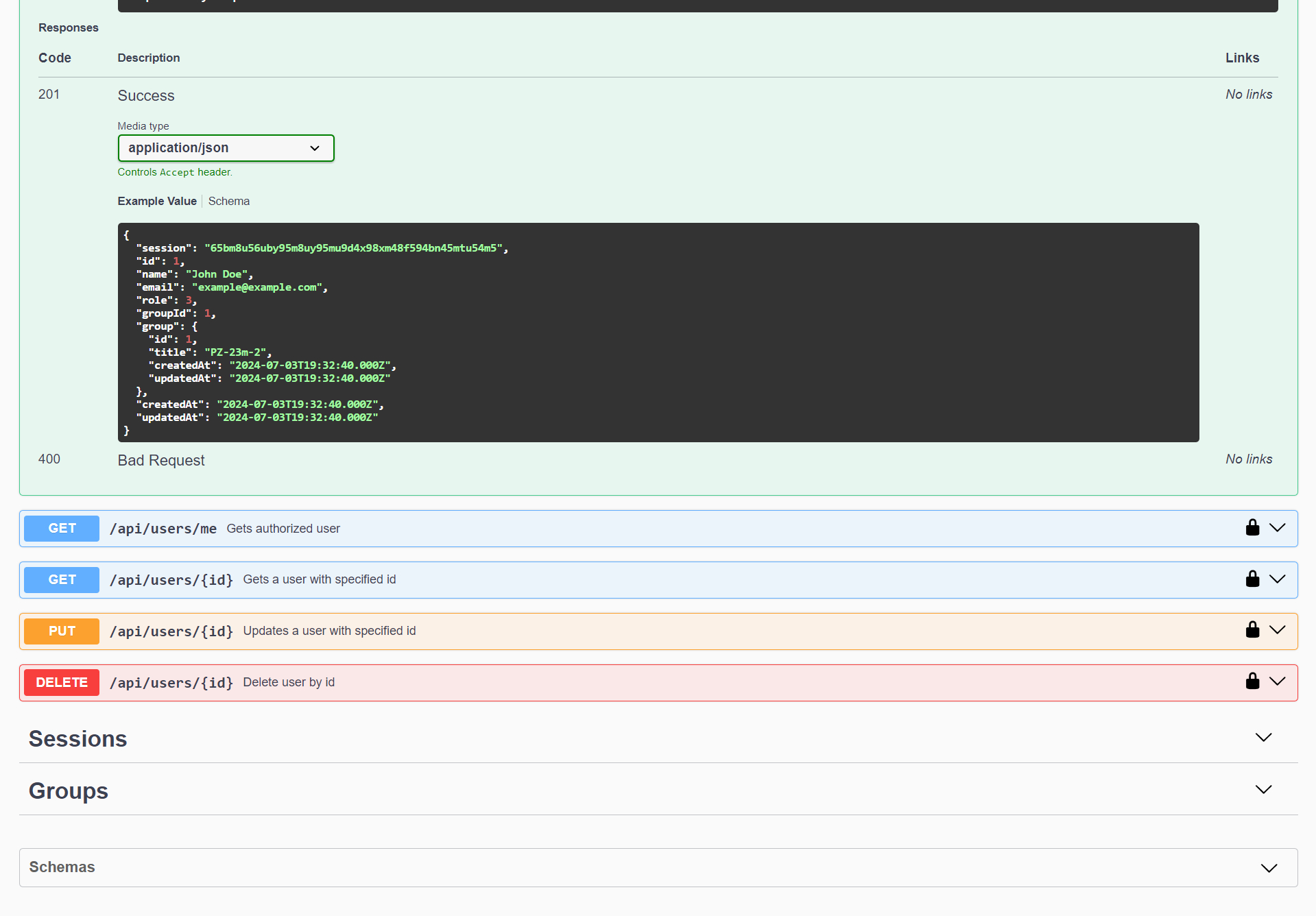


Рисунок 4 – Ендпоінт створення нового користувача

2.3 Система безпеки серверної частини

Для підтримки безпеки серверної частини додатку було обрано використання стратегій, guards, JWT-токенів, Passport.js та інтеграцію роботи з Redis для збереження токенів.

Стратегія –– це механізм, який визначає, як здійснюється аутентифікація та авторизація в додатку. Стратегії використовуються в поєднанні з модулями безпеки, такими як Passport, для реалізації різних методів аутентифікації.

Стратегія визначає, як користувачі аутентифікуються. Це може включати використання токенів JWT, сесій, OAuth, OpenID Connect тощо. Кожен тип аутентифікації має свою стратегію. Стратегії в NestJS реалізують певний інтерфейс, що дозволяє фреймворку знати, як перевіряти облікові дані користувача. Це включає методи для обробки запитів, валідації токенів та інших даних аутентифікації. Стратегії зазвичай використовуються у поєднанні з "guard" у NestJS. Guards перевіряють, чи користувач авторизований, та надають доступ до певних ендпоінтів у додатку. Використання стратегій у NestJS дозволяє реалізовувати та змінювати методи аутентифікації, що робить додаток більш безпечним і масштабованим.

Passport.js — це популярна бібліотека для Node.js, яка забезпечує просту та гнучку платформу для реалізації аутентифікації у веб-додатках. NestJS інтегрує Passport через модулі, що дозволяє легко використовувати різні стратегії аутентифікації.

JWT стратегія використовується для аутентифікації на основі токенів, що дозволяє користувачам отримувати доступ до захищених ресурсів.

JWT — це компактний, URL-безпечний спосіб передачі даних між сторонами як об'єкт JSON. У контексті аутентифікації JWT зазвичай генерується на сервері при вході користувача та використовується для підтвердження його особи.

Після успішної аутентифікації (наприклад, введення коректного логіна та пароля), сервер генерує JWT з корисними даними (payload) про користувача, наприклад, його ID. Токен повертається користувачу, який зберігає його, зазвичай у локальному сховищі браузера. При наступних запитах на сервер користувач включає токен у заголовок запиту, зазвичай в полі Authorization (Bearer Token). Сервер перевіряє дійсність токена, щоб аутентифікувати користувача. Реалізація стратегії показана у лістингу 2.10.

Лістинг 2.10 – Реалізація стратегії

import { Injectable, UnauthorizedException } from '@nestjs/common';  
import { PassportStrategy } from '@nestjs/passport';  
import { ExtractJwt, Strategy } from 'passport-jwt';  
import { RedisCacheService } from 'src/redis-cache/redis-cache.service';  
@Injectable()  
export class JwtStrategy extends PassportStrategy(Strategy) {  
 constructor(private readonly cacheService: RedisCacheService) {  
 super({  
 jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),  
 ignoreExpiration: false,  
 secretOrKey: *process*.env.SECRET.toString(),  
 });  
 }  
 async validate(payload: any) {  
 const data = {  
 token: await this.cacheService.get(`token\_${payload.id}`),  
 };  
 if (!data.token) {  
 throw new UnauthorizedException();  
 }  
 *Object*.assign(data, payload);  
 return data;  
 }  
}

Redis — це система зберігання даних у пам'яті, яка часто використовується для кешування та зберігання сесій. Використання Redis у поєднанні з JWT дозволяє підвищити безпеку аутентифікації. Токени можна зберігати у Redis для подальшої перевірки. Це дозволяє швидко отримувати інформацію про токени та сесії. Також Redis можна використовувати для реалізації механізмів витримки сесій, що дозволяє контролювати термін дії токенів. Наприклад, можна зберігати токен з терміном дії, а коли токен зберігається в Redis, його термін дії контролюється автоматично. Для роботи з Redis створено окрему компоненту. У лістингу 2.11 наведено реалізацію модуля для роботи з Redis. У лістингу 2.12 наведено реалізацію сервіса для роботи з Redis.

Лістинг 2.11 – Модуль RedisCacheModule

@Module({  
 imports: [  
 CacheModule.*register*<RedisClientOptions>({  
 // eslint-disable-next-line @typescript-eslint/ban-ts-comment  
 // @ts-expect-error  
 store: async () =>  
 await redisStore({  
 socket: {  
 host: *process*.env.REDIS\_HOST,  
 port: +*process*.env.REDIS\_PORT,  
 },  
 }),  
 }),  
 ],  
 providers: [RedisCacheService],  
 exports: [RedisCacheService],  
})  
export class RedisCacheModule {

...

}

Лістинг 2.12 – Сервіс RedisCacheService

import { Injectable } from '@nestjs/common';  
import { Cache } from '@nestjs/cache-manager';  
@Injectable()  
export class RedisCacheService {  
 constructor(private readonly cache: Cache) {}  
 async get(key: string): Promise<any> {  
 return this.cache.get(key) || 0;  
 }  
 async set(key: string, value: any, ttl?: number): Promise<void> {  
 return this.cache.set(key, value, ttl);  
 }  
 async remove(key: string): Promise<void> {  
 return this.cache.del(key);  
 }  
}

Компонента сесій відіграє важливу роль у забезпеченні захисту сервера, управлінні аутентифікацією та авторизацією користувачів. Модуль сесій об’єднує всі компоненти, пов’язані з управлінням сесіями, такими як контролери та сервіси. Він відповідає за інтеграцію залежностей і конфігурацію модулів для підтримки безпеки застосунку. Модуль сесій показаний у лістингу 2.13.

Лістинг 2.13 – Модуль SessionModule

@Module({  
 imports: [  
 JwtModule.*register*({  
 secret: *process*.env.SECRET,  
 }),  
 UserModule,  
 RedisCacheModule,  
 ],  
 controllers: [SessionController],  
 providers: [SessionService, JwtStrategy],  
 exports: [SessionService],  
})  
export class SessionModule {

...

}

Сервіс SessionService виконує функції, пов’язані з управлінням сесіями за допомогою JWT токенів та кешування через Redis. Конструктор приймає два параметри: JwtService — сервіс для роботи з JWT, який надає методи для створення та перевірки токенів; RedisCacheService — сервіс для управління кешем, що дозволяє зберігати токени у Redis для перевірки їх дійсності. Сервіс сесій показаний у лістингу 2.14.

Лістинг 2.14 – Сервіс SessionService

@Injectable()  
export class SessionService {  
 constructor(  
 private readonly jwtService: JwtService,  
 private readonly cacheService: RedisCacheService,  
 ) {}  
 createToken(id: number): Promise<string> {  
 return this.jwtService.signAsync(  
 { id },  
 {  
 secret: *process*.env.SECRET.toString(),  
 expiresIn: *process*.env.EXPIRES\_IN.toString(),  
 },  
 );  
 }

async verify(token: any) {  
 const payload = this.jwtService.verify(token, {  
 secret: *process*.env.SECRET.toString(),  
 });

const redisToken = await this.cacheService.get(`token\_${payload.id}`);  
  
 if (!redisToken) {  
 throw new WsException('Invalid token!');  
 }  
 return payload;}}

Асинхронний метод createToken створює новий JWT токен, приймаючи id користувача як параметр. signAsync генерує токен, включаючи в його корисне навантаження (payload) об’єкт, що містить id. Конфігурація токена включає: secret — секретний ключ, який зберігається в змінній середовища; expiresIn — строк дії токена, що також береться з змінної середовища.

Асинхронний метод verify перевіряє дійсність переданого токена. Метод verify декодує токен, перевіряючи його справжність за допомогою секретного ключа. Після декодування токена метод перевіряє, чи існує токен у кеші Redis, використовуючи cacheService.get. Якщо токен не знайдено у кеші, виникає помилка WsException з повідомленням "Invalid token!". Якщо токен дійсний і знайдений у кеші, метод повертає декодований payload токена.

Контролер SessionController відповідає за управління сесіями користувачів, реалізуючи можливості для створення та видалення сесій.

Лістинг 2.15 – Контроллер SessionController

@ApiBearerAuth()  
@ApiTags('Sessions')  
@Controller('/sessions')  
export class SessionController {  
 constructor(  
 private readonly sessionService: SessionService,  
 private readonly userService: UserService,  
 private readonly cacheService: RedisCacheService,  
 ) {}  
 @ApiOperation({ summary: 'Creates new session' })  
 @ApiBody({  
 description: 'User data',  
 type: CreateSessionSchema,  
 })  
 @ApiResponse({  
 status: HttpStatus.*CREATED*,  
 type: SessionDto,  
 })  
 @ApiResponse({ status: HttpStatus.*NOT\_FOUND*, description: 'Not Found' })  
 @Post()  
 async createSession(  
 @Body() body: CreateSessionSchema,  
 ): Promise<SessionDto> {  
 const user = await this.userService.findUserByEmailAndPassword(body);  
 const token = await this.sessionService.createToken(user.id);  
 await this.cacheService.set(  
 `token\_${user.id}`,  
 token,  
 +*process*.env.EXPIRES\_IN\_MS,  
 );  
  
 return new SessionDto(token);  
 }  
 @ApiOperation({ summary: 'Deletes the session' })  
 @ApiResponse({  
 status: HttpStatus.*NO\_CONTENT*,  
 description: 'Deleted',  
 })  
 @ApiResponse({  
 status: HttpStatus.*UNAUTHORIZED*,  
 description: 'Invalid auth token',  
 })  
 @HttpCode(204)  
 @UseGuards(JwtAuthGuard)  
 @Delete()  
 async deleteSession(@Request() req): Promise<void> {  
 await this.cacheService.remove(`token\_${req.user.id}`);  
 return;  
 }  
}

Guards в NestJS — це механізм, що відповідає за обробку авторизації запитів до контролерів. Вони виконуються перед обробкою запиту і можуть використовуватись для перевірки, чи має користувач право на доступ до певного ресурсу або дії. Guards можуть реалізовувати будь-яку логіку, що пов'язана з авторизацією, таку як перевірка токенів, перевірка ролей користувача та інше.

Guards повертають булеве значення (true або false), яке вказує, чи дозволено виконати запит. Якщо guard повертає false, NestJS автоматично відхиляє запит і повертає статус 403 (доступ заборонено). Guards можуть також повертати Promise<boolean> або Observable<boolean> для асинхронних операцій.

Конфігурація Guard показана у лістингу 2.16.

Лістинг 2.16 – Конфігурація Guard

import { ExecutionContext, Injectable } from '@nestjs/common';  
import { AuthGuard } from '@nestjs/passport';  
import { Observable } from 'rxjs';  
@Injectable()  
export class JwtAuthGuard extends AuthGuard('jwt') {  
 canActivate(  
 context: ExecutionContext,  
 ): boolean | Promise<boolean> | Observable<boolean> {  
 return super.canActivate(context);  
 }  
}

ExecutionContext –– це інтерфейс, що надає контекст виконання запиту, дозволяючи отримувати інформацію про запит, користувача, та інші аспекти. Базовий клас AuthGuard надається бібліотекою Passport, який реалізує логіку аутентифікації. Observable –– це клас з бібліотеки RxJS, що використовується для обробки асинхронних запитів. JwtAuthGuard наслідує від базового класу AuthGuard і передає йому стратегію 'jwt', що означає, що цей guard буде використовувати JWT для аутентифікації.

Декоратор @ApiBearerAuth() вказує, що всі запити до контролера чи ендпоінта вимагають авторизації за допомогою Bearer токена. Це підкреслює, що доступ обмежений лише автентифікованим користувачам. Декоратор @UseGuards(JwtAuthGuard) забезпечує захист ендпоінта, вимагаючи наявність дійсного JWT токена в залежності від обраного Guard. Використання Guard показано у лістингу 2.17.

Лістинг 2.17 – Використання Guard

// На рівні контроллера

@ApiBearerAuth()  
@ApiTags('Sessions')  
@Controller('/sessions')  
export class SessionController {  
 constructor(  
 private readonly sessionService: SessionService,  
 private readonly userService: UserService,  
 private readonly cacheService: RedisCacheService,  
 ) {}

...

// На рівні ендпоінтів

ApiOperation({ summary: 'Creates new group' })  
@ApiBody({  
 description: 'Group data',  
 type: CreateGroupSchema,  
})  
@ApiResponse({  
 status: HttpStatus.*CREATED*,  
 description: 'Success',  
 type: GroupDto,  
})  
@ApiResponse({ status: HttpStatus.*BAD\_REQUEST*, description: 'Bad Request' })  
@ApiBearerAuth()  
@UseGuards(JwtAuthGuard)  
@Post()  
async create(@Body() createGroupDto: CreateGroupSchema) {

...

}

**ВИСНОВКИ**

В результаті науково-дослідної практики було:

* проведено аналіз предметної області;
* досліджено існуючі рішення для вибору дисциплін;
* проаналізовано дані про минулі вибори дисциплін;
* обрано основні інструменти і методи розроблення;
* спроєктовано архітектуру автоматизованої системи обліку вибору навчальних дисциплін;
* розроблено систему безпеки серверної частини додатку.

У подальшій роботі необхідно буде використати результати практики для розроблення автоматизованої системи обліку вибору навчальних дисциплін кафедри інженерії програмного забезпечення та інформаційних технологій ДНУ.

**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ**

1. JavaScript — Wikipedia. URL: https://uk.wikipedia.org/wiki/JavaScript (дата звернення: 09.10.2024);
2. TypeScript — Wikipedia. URL: https://uk.wikipedia.org/wiki/TypeScript (дата звернення: 09.10.2024);
3. Node.js — Wikipedia. URL: https://uk.wikipedia.org/wiki/Node.js (дата звернення: 10.10.2024);
4. NestJS — Wikipedia. URL: https://uk.wikipedia.org/wiki/NestJS (дата звернення: 10.10.2024);
5. Що таке Angular? — tutorial.in.ua. URL: https://angular.tutorial.in.ua/what-is-angular/ (дата звернення: 10.10.2024);
6. MySQL — Wikipedia. URL: https://uk.wikipedia.org/wiki/MySQL (дата звернення: 10.10.2024).