

Дніпровський національний університет імені Олеся Гончара  
Факультет прикладної математики та інформаційних технологій  
Кафедра інженерії програмного забезпечення та інформаційних  
технологій

Кваліфікаційна робота  
другий (магістерський) рівень вищої освіти  
спеціальність 121 Інженерія програмного забезпечення  
освітня програма Інженерія програмного забезпечення

РОЗРОБЛЕННЯ СИСТЕМИ ОБЛІКУ ВИБОРУ НАВЧАЛЬНИХ ДИСЦИПЛІН  
ЗДОБУВАЧІВ ВИЩОЇ ОСВІТИ

Виконавець  
студентка групи ПЗ-23м-2  
Лашко Євгенія Леонідівна

\_\_\_\_\_  
(підпис)

Керівник  
доцент кафедри ІПЗІТ,  
канд. техн. наук, доцент  
Антоненко Світлана Валентинівна

\_\_\_\_\_  
(підпис)

Завідувач кафедри ІПЗІТ  
д-р техн. наук, професор  
Байбуз Олег Григорович

\_\_\_\_\_  
(підпис)

Дніпро – 2024

**ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ОЛЕСЯ ГОНЧАРА**

Факультет прикладної математики та інформаційних технологій  
Кафедра Інженерії програмного забезпечення та інформаційних технологій  
Рівень вищої освіти другий (магістерський)  
Спеціальність 121 Інженерія програмного забезпечення  
Спеціалізація \_\_\_\_\_  
Освітня програма Інженерія програмного забезпечення

**ЗАТВЕРДЖУЮ**

В.о. завідувача кафедри  
Інженерії програмного  
забезпечення та інформаційних  
технологій  
\_\_\_\_\_ *Олег БАЙБУЗ*  
(підпис)  
«\_\_\_» \_\_\_\_\_ 2024 року

***З А В Д А Н Н Я***  
**НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Лашко Євгенії Леонідівні

(Прізвище, ім'я, по-батькові студента)

1. Тема роботи Розроблення системи обліку вибору навчальних дисциплін  
здобувачів вищої освіти

керівник роботи Антоненко Світлана Валентинівна, канд. техн. наук, доцент,  
(Прізвище, ім'я, по-батькові, науковий ступінь, вчене звання)

затверджені наказом по Університету від «\_\_» травня 2024 року № \_\_\_\_\_

2. Термін подання роботи 07 грудня 2024 р.

3. Вхідні дані до роботи xlsx-файл з результатами вибору дисциплін за весняний  
семестр 2024-2025 навчального року.

4. Перелік питань, які потрібно розробити \_\_\_\_\_

1. Провести огляд методів та програмних засобів для розв'язання задачі вибору  
навчальних дисциплін.

2. Розробити власне програмне забезпечення для обліку вибору навчальних  
дисциплін здобувачів вищої освіти; провести його тестування.

3. Провести практичну апробацію програмного забезпечення на реальних даних.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) \_\_\_\_\_  
Презентація у Microsoft PowerPoint.

6. Керівник, консультант з окремих (спеціальних) розділів роботи

Розділ	Ініціали прізвище та посада керівника, консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	С.В. Антоненко, доцент кафедри ІПЗІТ	14.10.2024	10.11.2024
2	С.В. Антоненко, доцент кафедри ІПЗІТ	11.11.2024	24.11.2024
3	С.В. Антоненко, доцент кафедри ІПЗІТ	25.11.2024	07.12.2024

7. Дата видачі завдання 14.10.2024

### КАЛЕНДАРНИЙ ПЛАН

№ з/п	Зміст етапів роботи	Термін виконання етапів роботи	Примітка
1	Знайомство із задачею, методами її розв'язання, існуючими програмними засобами для вибору навчальних дисциплін. Опрацювання літературних джерел.	14.10.2024 – 28.10.2024	
2	Розроблення і тестування програмного забезпечення для обліку вибору навчальних дисциплін здобувачів вищої освіти.	29.10.2024 – 17.11.2024	
3	Проведення практичної апробації програмного забезпечення на реальних даних.	18.11.2024 – 29.11.2024	
4	Оформлення роботи.	30.11.2024 – 07.12.2024	

Студент

\_\_\_\_\_  
(підпис) Євгенія ЛАШКО  
(Власне ім'я ПРІЗВИЩЕ)

Керівник роботи

\_\_\_\_\_  
(підпис) Світлана АНТОНЕНКО  
(Власне ім'я ПРІЗВИЩЕ)

## РЕФЕРАТ

**Кваліфікаційна робота:** 80 с., 38 рис., 0 табл., 12 джерел, 0 додатків.

**Об'єкт дослідження:** система вибору навчальних дисциплін здобувачами освіти.

**Мета роботи:** створення системи обліку вибору навчальних дисциплін здобувачами освіти для кафедри інженерії програмного забезпечення та інформаційних технологій Дніпровського національного університету імені Олеся Гончара.

**Методи дослідження:** методи та засоби розроблення програмного забезпечення.

**Одержані висновки та їх новизна:** під час виконання роботи було проведено аналіз існуючих систем, які забезпечують вибір навчальних дисциплін; розроблено нове програмне забезпечення для обліку вибору навчальних дисциплін здобувачами освіти для кафедри інженерії програмного забезпечення та інформаційних технологій Дніпровського національного університету імені Олеся Гончара; проведено його тестування на реальних даних.

**Реалізація:** програмне забезпечення розроблено мовою TypeScript за допомогою IDE WebStorm.

**Результати дослідження можуть бути використані** у вищих навчальних закладах та їх окремих структурних підрозділах, а саме на кафедрі інженерії програмного забезпечення та інформаційних технологій Дніпровського національного університету імені Олеся Гончара.

**Ключові слова:** ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, TYPESCRIPT, NESTJS, SEQUELIZE, КОНТРОЛЕР, СЕРВІС, МОДУЛЬ.

## ANNOTATION

The thesis of the 2nd-year student Y. Lashko (Oles Honchar Dnipro National University, Faculty of Applied Mathematics and Information Technologies, Department of Software Engineering and Information Technologies) is devoted to the development of software for managing the selection of academic disciplines by students.

The purpose of the work is to develop software for managing the selection of academic disciplines by students for the Department of Software Engineering and Information Technologies at Oles Honchar Dnipro National University.

As part of the work, the software "DNUChoice" was created to facilitate the selection and accounting of academic disciplines by students.

In the course of the research, a comprehensive application was developed. The server-side is implemented using Node.js with the NestJS framework, employing MySQL as the database managed through Sequelize ORM and Redis for caching and optimizing data access. The client-side is built with Angular, utilizing PrimeNG for a responsive and interactive user interface and RxJS for managing reactive data streams and asynchronous events. The system's architecture integrates a secure API for communication between the client and server, supporting CRUD operations, caching mechanisms, and role-based access control. It adheres to a component-oriented and modular design, ensuring scalability and ease of customization for various use cases.

The system is designed to support deployment in cloud environments, ensuring scalability, reliability, and performance. It could be deployed on AWS infrastructure, utilizing services such as Elastic Beanstalk for managing server-side deployment, RDS for MySQL database hosting, and ElastiCache for Redis-based caching. Client-side static assets could be served via Amazon S3, with CloudFront enabling efficient global content delivery.

Bibliography 12, pictures 3, tables 0.

## ЗМІСТ

ВСТУП .....	7
1 ОГЛЯДОВИЙ РОЗДІЛ .....	10
1.1 Аналіз існуючих рішень .....	10
1.2 Опис обраних технологій та засобів розроблення.....	12
1.2.1 TypeScript .....	12
1.2.2 Node.js та NestJS.....	13
1.2.3 MySQL, ORM Sequelize i Redis.....	14
1.2.4 Angular, PrimeNG, RxJS.....	16
1.3 Постановка завдання.....	17
2 ОГЛЯД РОЗРОБЛЕНОГО ПРОДУКТУ .....	19
2.1 Структура бази даних .....	19
2.2 Робота з базою даних за допомогою ORM Sequelize .....	20
2.3 Серверна частина додатку.....	25
2.3.1 Основні модулі додатку .....	28
2.3.2 Основна робота з сутностями. Module, Service, Controller.....	30
2.3.3 Система безпеки серверної частини .....	42
2.4 Клієнтська частина додатку .....	49
2.4.1 Роутинг .....	51
2.4.2 Сервіси .....	52
2.4.3 Компоненти .....	55
3 ПРАКТИЧНІ РЕЗУЛЬТАТИ.....	60
3.1 Приклади роботи продукту .....	60
3.2 Завантаження реальних даних про минулі вибори навчальних дисциплін здобувачів освіти .....	77
ВИСНОВКИ.....	79
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ .....	80

## ВСТУП

В умовах сучасного освітнього процесу важливим є створення ефективних інформаційних систем для автоматизації ключових аспектів діяльності закладів вищої освіти. Однією з важливих складових такого процесу є вибір навчальних дисциплін здобувачами вищої освіти, що забезпечує індивідуалізацію освітньої траєкторії кожного студента відповідно до його професійних інтересів та цілей. Проте, відсутність зручного інструменту для обліку й управління цим процесом часто призводить до технічних труднощів та надмірного навантаження на адміністративний персонал.

Дана кваліфікаційна робота присвячена розробленню системи обліку вибору навчальних дисциплін здобувачів вищої освіти, яка має на меті забезпечення автоматизації цього процесу, підвищення його ефективності, точності й доступності для всіх учасників освітнього процесу. Система розроблена з використанням сучасних технологій, таких як NestJS для серверної частини, Angular для клієнтської частини, а також MySQL у поєднанні з ORM-фреймворком Sequelize для роботи з базами даних. Для побудови інтерфейсів користувача використано бібліотеку компонентів PrimeNG, що дозволяє створювати зручний та сучасний інтерфейс.

Актуальність роботи визначається потребою автоматизації вибору дисциплін як складової цифровізації освітнього процесу в контексті розвитку інформаційного суспільства. Впровадження подібних систем сприяє зниженню адміністративного навантаження, мінімізації помилок при обробці інформації та підвищенню задоволеності всіх учасників освітнього процесу через зручність і ефективність процесу вибору навчальних дисциплін.

Розроблення програмного забезпечення здійснювалося для кафедри інженерії програмного забезпечення та інформаційних технологій Дніпровського національного університету імені Олеся Гончара. Основна мета створеної

системи — забезпечити інструмент для організації та обліку вибору дисциплін здобувачами освіти різних курсів і спеціальностей відповідно до навчального плану.

У роботі були застосовані такі методи й підходи:

1. Модульна архітектура для забезпечення масштабованості системи.
2. ORM-інструменти для зручного управління базою даних.
3. Реактивний підхід до побудови інтерфейсу користувача.
4. Принципи адаптивного дизайну, що дозволяють забезпечити коректну роботу системи на різних пристроях.

Основною метою роботи є створення системи обліку вибору навчальних дисциплін здобувачами освіти для кафедри інженерії програмного забезпечення та інформаційних технологій Дніпровського національного університету імені Олеся Гончара.

В ході роботи необхідно виконати наступні задачі:

- дослідити предметну область для розроблення системи обліку навчальних дисциплін здобувачами освіти;
- розробити серверну частину додатку з забезпеченням правил захисту;
- розробити клієнтську частину додатку з використанням реактивного підходу;
- розробити інструмент автоматичного завантаження інформації про попередні вибори навчальних дисциплін здобувачами освіти;
- провести практичну апробацію програмного комплексу на реальних даних.

У кваліфікаційній роботі описано наступні розділи:

1. Оглядовий розділ, в якому наведено аналіз існуючих рішень, опис обраних технологій та засобів розроблення, а також описано постановку завдання.



2. Огляд розробленого продукту, де описано структуру бази даних та роботу з нею за допомогою ORM Sequelize. Описано загальну структуру серверної частини додатку, основні функціональні та інфраструктурні модулі, систему безпеки серверної частини, роботу з основними сутностями, такими як Module, Service, Controller, Schema і DTO. Також описано клієнтську частину додатку, а саме роутинг, сервіси, компоненти, основні принципи й методи їх побудови, реактивний підхід та реактивні форми.

3. Практичні результати, де наведено приклади роботи продукту для користувачів з різними ролями і показано завантаження реальних даних про минулі вибори навчальних дисциплін здобувачами освіти.

## 1 ОГЛЯДОВИЙ РОЗДІЛ

### 1.1 Аналіз існуючих рішень

Вибір навчальних дисциплін здобувачами вищої освіти є важливим елементом освітнього процесу, спрямованого на забезпечення індивідуалізації освітніх траєкторій. Для автоматизації цього процесу розроблено низку рішень, які впроваджуються як в Україні, так і за її межами.

На сьогоднішній день у Дніпровському національному університеті імені Олеся Гончара вибір навчальних дисциплін здобувачами освіти здійснюється за допомогою опитувальників, створених на базі Microsoft Forms. Цей підхід є простим у впровадженні та використанні, а також не потребує значних фінансових затрат, оскільки університет вже має ліцензії на використання продуктів Microsoft у рамках корпоративних програм.

Опитувальники дозволяють студентам обирати дисципліни за допомогою онлайн-форми, після чого зібрані дані передаються відповідальним особам для обробки. Однак такий підхід має низку недоліків:

1. Дані, зібрані через форми, обробляються вручну, що створює ризик помилок та значно збільшує час на їх опрацювання.

2. Усунення конфліктів виконується вручну, що може призводити до затримок.

3. Microsoft Forms не дозволяє створювати комплексні моделі обліку та аналізу даних, що ускладнює роботу співробітників кафедри.

Попри ці недоліки, використання Microsoft Forms показало базову ефективність у зборі інформації про вибір дисциплін. Однак очевидно є потреба у створенні більш автоматизованої системи, яка дозволить уникнути зменшити навантаження на персонал та забезпечити швидкість прийняття рішень.

Одним із найпоширеніших рішень для автоматизації освітнього процесу є впровадження інформаційних систем управління навчальними закладами, таких як "Електронний кампус", Moodle, Canvas LMS та Blackboard.

Наприклад, Moodle широко використовується в багатьох українських університетах як система дистанційного навчання. Вона пропонує інструменти для створення курсів і тестування, однак функціонал для вибору навчальних дисциплін у ній реалізований обмежено. Основним недоліком таких систем є їхня складність у налаштуванні та адаптації до специфічних потреб кафедри, а також високі вимоги до ресурсів [1].

Blackboard та Canvas LMS, популярні за кордоном, також надають можливість адміністрування навчального процесу, але їх впровадження вимагає значних фінансових витрат. Крім того, їх функціонал часто не відповідає вимогам українських закладів освіти через різницю в нормативно-правовій базі та навчальних планах. Багато університетів розробляють власні програмні рішення для автоматизації вибору дисциплін. Наприклад, система "Деканат" використовується у деяких українських вищих навчальних закладах. Вона дозволяє вести облік студентів, розклад занять, успішність, а також частково забезпечує вибір навчальних дисциплін. Проте її обмежений функціонал та складний інтерфейс знижують ефективність використання. Крім того, розробка та підтримка таких систем потребують значних ресурсів і висококваліфікованих фахівців [2].

Останнім часом набирають популярності хмарні рішення та SaaS-платформи, які пропонують університетам готові інструменти для автоматизації процесів. До таких рішень належать, наприклад, PowerSchool та інші. PowerSchool — це провідний постачальник хмарних програмних рішень для закладів середньої освіти (K-12), що надає комплексні інструменти для управління навчальним процесом, адміністративними завданнями та комунікацією між учасниками освітнього процесу. Вони забезпечують високу

швидкість впровадження, легкість використання та технічну підтримку. Однак їхні основні недоліки полягають у відсутності гнучкості, складнощах адаптації до специфічних потреб та високій вартості ліцензій [3].

Таким чином, існуюча система, побудована на базі Microsoft Forms, є тимчасовим рішенням, яке частково вирішує проблему обліку вибору навчальних дисциплін, але не забезпечує необхідного рівня автоматизації, інтеграції та зручності використання. У поєднанні з недоліками інших наявних систем, це обґрунтовує необхідність розроблення власного програмного забезпечення, яке буде враховувати специфіку роботи кафедри та освітньої установи загалом.

## 1.2 Опис обраних технологій та засобів розроблення

Для створення системи обліку вибору навчальних дисциплін здобувачів вищої освіти обрано сучасний стек технологій, що дозволяє забезпечити високу продуктивність, масштабованість і зручність підтримки. Обрана мова програмування — TypeScript. Для реалізації клієнтської частини додатку обрано фреймворк Angular, використано бібліотеку компонентів PrimeNG та бібліотеку для реактивного програмування RxJS. Для реалізації серверної частини обрано Node.js і фреймворк NestJS. У якості бази даних обрано MySQL у поєднанні з ORM Sequelize. Для зберігання JWT-токенів обрано Redis. Обрані технології створюють комплексну екосистему для розроблення веб-додатка, що відповідає сучасним вимогам до продуктивності, зручності та масштабованості.

### 1.2.1 TypeScript

TypeScript — це мова програмування, розроблена компанією Microsoft і представлена восени 2012 року. Вона створена як інструмент для вдосконалення розробки вебзастосунків, доповнюючи можливості JavaScript. Основним

розробником TypeScript є Андерс Гейлсберг, який також є автором таких мов, як C#, Turbo Pascal і Delphi.

Код компілятора, що перетворює TypeScript у JavaScript, поширюється під ліцензією Apache, а розробка ведеться у відкритому доступі через публічний репозиторій. Специфікації мови є відкритими і опубліковані відповідно до угоди Open Web Foundation Specification Agreement (OWFa 1.0). Завдяки повній сумісності з JavaScript, код, написаний на TypeScript, після компіляції можна виконувати у будь-якому сучасному браузері або використовувати з серверною платформою Node.js.

Серед переваг TypeScript у порівнянні з JavaScript можна виділити статичну типізацію, яка дозволяє явно задавати типи, підтримку повноцінних класів і можливість використання модулів.

У проєкті TypeScript застосовувався для створення як клієнтської, так і серверної частин системи. Це забезпечило високу якість, чіткість та надійність розробленого програмного забезпечення [4].

### 1.2.2 Node.js та NestJS

Node.js — це платформа з відкритим вихідним кодом, що дозволяє створювати високопродуктивні мережеві застосунки за допомогою JavaScript. Ініціатором розробки платформи став Раян Дал (Ryan Dahl). Якщо спочатку JavaScript використовувався виключно для виконання сценаріїв у браузері, то Node.js відкрив можливість виконувати JavaScript-код на сервері та повертати клієнту результати обробки. Завдяки цьому JavaScript перетворився на мову загального призначення, яка об'єднала велику спільноту розробників.

Особливості Node.js включають асинхронну обробку запитів у однопоточному середовищі, неблокуючий ввід/вивід, систему модулів CommonJS, а також використання JavaScript-рушія Google V8.

Для управління залежностями та модулями застосовується npm (node package manager), що є стандартним інструментом у екосистемі Node.js [5].

Nest (NestJS) — це сучасний фреймворк для створення масштабованих і продуктивних серверних додатків на платформі Node.js. Він базується на JavaScript і повністю підтримує TypeScript, водночас залишаючи можливість розробки на чистому JavaScript. Nest об'єднує принципи об'єктно-орієнтованого програмування (ООП), функціонального програмування (ФП) і функціонального реактивного програмування (ФРП), створюючи потужний інструмент для різних підходів у розробці.

Фреймворк пропонує готову архітектуру, яка сприяє створенню додатків із чітко визначеними модулями, слабкими зв'язками та високим рівнем тестованості. Така структура робить програми не лише простими у розробці, але й легкими для масштабування та підтримки.

NestJS був обраний для реалізації серверної частини проєкту завдяки його відповідності сучасним стандартам розробки, зручності використання та високій продуктивності. Це забезпечило стабільну й ефективну основу для побудови серверної логіки системи [6].

### 1.2.3 MySQL, ORM Sequelize i Redis

MySQL — це система управління реляційними базами даних із відкритим вихідним кодом, розроблена компанією «ТсХ» для забезпечення високої швидкодії під час роботи з великими обсягами даних. Вона була створена як доступна альтернатива комерційним рішенням, швидко здобувши популярність завдяки своїй ефективності та гнучкості.

Спочатку MySQL мала схожість із mSQL, але з часом зазнала значних удосконалень, ставши однією з найвідоміших і найбільш використовуваних систем управління базами даних у світі. Її переваги особливо цінуються при

створенні динамічних вебсторінок, оскільки MySQL підтримує інтеграцію з багатьма мовами програмування та забезпечує стабільність роботи навіть у складних проєктах [7].

Sequelize — це ORM-бібліотека для Node.js, яка дозволяє взаємодіяти з реляційними базами даних через об'єктно-орієнтовані моделі. Вона забезпечує простоту управління даними та зменшує кількість SQL-коду, необхідного для виконання операцій.

Основні переваги Sequelize:

1. Автоматизація запитів. Моделі та асоціації дозволяють генерувати складні SQL-запити.
2. Міграції, що дозволяють вносити зміни у схемі бази даних, які будуть виконуватися автоматично.
3. Підтримка багатьох баз даних, таких як MySQL, PostgreSQL, SQLite тощо.

Sequelize забезпечує взаємодію серверної частини з MySQL у проєкті, спрощуючи розробку та підтримку коду [8].

Redis (REmote DIctionary Server) — це високопродуктивне сховище даних у пам'яті, яке використовується для кешування, управління чергами повідомлень і зберігання структур даних. Redis є одним із найшвидших інструментів у своїй категорії, забезпечуючи мінімальну затримку доступу до даних завдяки використанню оперативної пам'яті для зберігання. Redis забезпечує швидке читання й запис даних завдяки зберігання інформації у пам'яті, що зменшує час доступу до мікросекунд. У розробленій системі Redis використовується для зберігання JWT (JSON Web Token) токенів, які забезпечують механізм авторизації та автентифікації користувачів. Цей підхід дозволяє керувати терміном дії токенів і забезпечувати додатковий рівень безпеки для збереження їх актуальності [9].

### 1.2.4 Angular, PrimeNG, RxJS

Angular — це сучасний веб-фреймворк, створений компанією Google, який широко використовується для розробки динамічних односторінкових додатків (Single Page Applications, SPA). Фреймворк побудований на TypeScript, що надає додаткову строгість та типізацію, покращуючи якість і стабільність коду.

Однією з ключових переваг Angular є його компонентна архітектура. Веб-додаток у цьому фреймворку організовується як набір незалежних компонентів, кожен із яких містить свій шаблон, логіку та стилі. Це забезпечує високу модульність і спрощує масштабування й підтримку проєктів.

Особливістю Angular є двостороннє зв'язування даних, яке дозволяє автоматично синхронізувати стан між компонентами та їх візуальним відображенням. Цей підхід значно спрощує роботу з даними в реальному часі, забезпечуючи зручну інтерактивність додатків.

Фреймворк також оснащений системою ін'єкції залежностей, яка дозволяє зручно управляти зв'язками між різними частинами програми. Це не тільки полегшує розробку та підтримку коду, але й робить його легшим для тестування.

Механізм маршрутизації Angular дозволяє створювати багатосторінкові додатки, де кожен URL відповідає певному компоненту. Це забезпечує інтуїтивну навігацію для користувачів і водночас зберігає переваги SPA, такі як швидкість і безшовний перехід між сторінками.

Angular також славиться своєю розширюваністю. Його базова функціональність може бути легко доповнена за допомогою сторонніх бібліотек і модулів, що робить його універсальним інструментом для різних потреб.

Завдяки своїй потужності, структурованості та інтеграції сучасних розробницьких підходів Angular ідеально підходить для великих і складних проєктів, забезпечуючи високу продуктивність і зручність роботи з фронтом [10].



PrimeNG — це бібліотека компонентів інтерфейсу для фреймворку Angular, яка забезпечує широкий набір готових інструментів для створення сучасних веб-додатків. PrimeNG включає різноманітні компоненти, такі як таблиці, графіки, форми, календарі, діалогові вікна та багато інших, що дозволяє скоротити час розроблення й отримати естетично привабливий та функціональний інтерфейс [11].

RxJS (Reactive Extensions for JavaScript) — це бібліотека для реактивного програмування в JavaScript, яка забезпечує можливість роботи з асинхронними потоками даних. RxJS дозволяє обробляти події, запити до сервера, таймери та інші асинхронні процеси, використовуючи потужний набір операторів.

RxJS є невід'ємною частиною Angular і широко використовується для оброблення даних у реактивному стилі, що значно спрощує роботу з потоками та підвищує читабельність коду [12].

### 1.3 Постановка завдання

Розроблення системи включає створення серверної та клієнтської частин, використання бази даних для зберігання даних і реалізацію інтеграції між усіма компонентами.

Основні завдання для серверної частини додатку:

1. Реалізація REST API: створення ендпоінтів для взаємодії з клієнтською частиною для забезпечення основних функцій застосунку, таких як:

- реєстрація та авторизація;
- облік і підтримка CRUD-операцій щодо усіх даних у системі;
- оброблення вибору і перевибору навчальних дисциплін здобувачами освіти;
- завантаження даних про минулі вибори дисциплін.

2. Розроблення системи безпеки для серверної частини додатку, а також інтеграція з Passport.js та Redis.

Основні завдання для клієнтської частини додатку:

1. Реалізація зручних та інтерактивних компонентів з використанням реактивного підходу.

2. Підтримка інтерактивного пошуку, фільтрації та сортування даних.

3. Відображення повідомлень про дії користувачів.

4. Реалізація панелі управління для керування усіма даними.

5. Динамічна взаємодія з сервером.

6. Оброблення даних в асинхронному режимі.

7. Забезпечення сумісності форматів даних між клієнтом і сервером через JSON.

8. Респонсивний дизайн та адаптація вигляду і функціональності додатка до різних розмірів екранів і типів пристроїв.

Поставлена мета також передбачає проведення практичної апробації розробленого програмного застосунку і тестування на реальних даних.

## 2 ОГЛЯД РОЗРОБЛЕНОГО ПРОДУКТУ

### 2.1 Структура бази даних

EER діаграма бази даних наведена на рисунку 2.1.

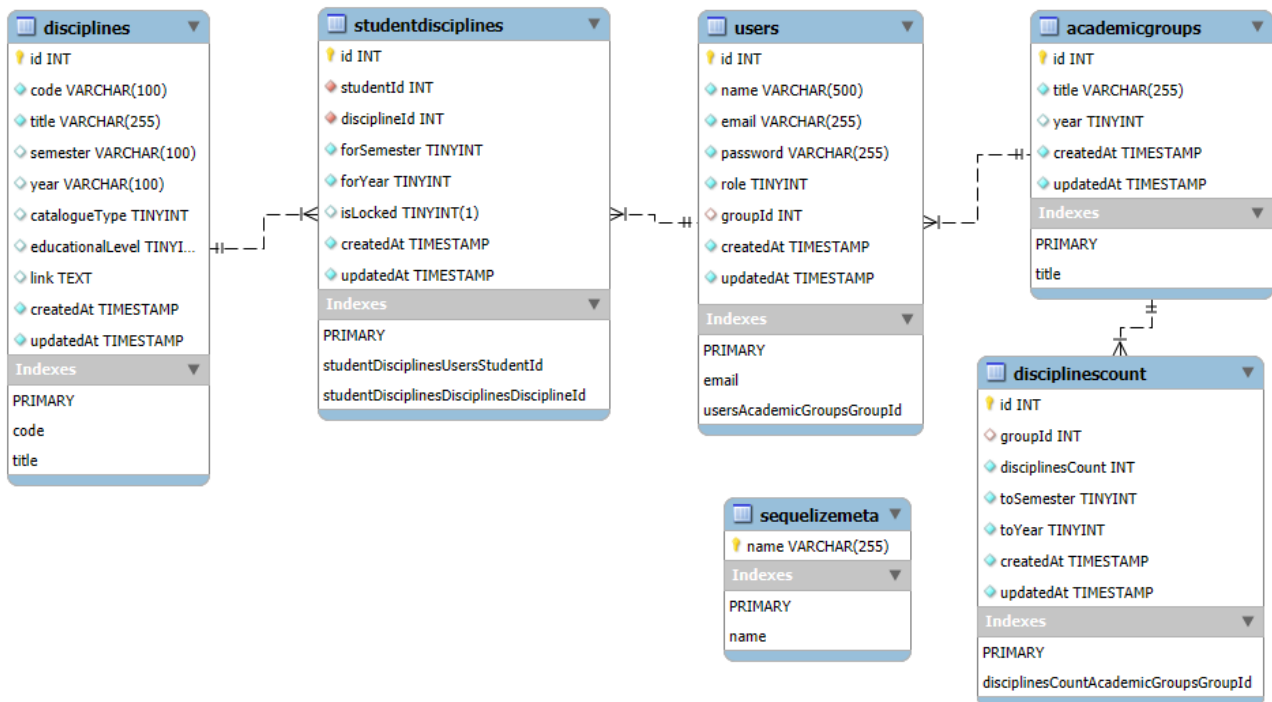


Рисунок 2.1 – EER діаграма бази даних

Таблиця `disciplines` слугує для зберігання навчальних дисциплін. Має індекси для унікальних значень на поля `code` (код дисципліни) і `title` (назва дисципліни) та зв'язок "один-до-багатьох" із таблицею `studentDisciplines` для зберігання інформації про дисципліни, які обрали певні здобувачі освіти.

Таблиця `studentDisciplines` відповідає за зберігання інформації про вибрані здобувачами освіти навчальні дисципліни. Має зв'язок "багато-до-одного" із таблицею `users` для зберігання яким саме користувачем були обрані деякі дисципліни та зв'язок "багато-до-одного" із таблицею `disciplines` для зберігання інформації про те, які саме дисципліни обрали здобувачі освіти.

Таблиця `users` слугує для зберігання інформації про користувачів додатку. Користувачі розділені по ролям в залежності від значення поля `role` (1 — адміністратор, 2 — викладач, 3 — здобувач освіти). Таблиця має індекс для унікальних значень на поле `email` (пошта користувача). Має такі зв'язки:

- зв'язок "багато-до-одного" із таблицею `academicGroups`, який відповідає за належність користувача до певної групи;
- зв'язок "один-до-багатьох" із таблицею `studentDisciplines`.

Таблиця `academicGroups` відповідає за зберігання інформації про академічні групи. Має індекс для унікальних значень на поле `title` (назва групи). Має такі зв'язки:

- зв'язок "один-до-багатьох" із таблицею `users`.
- Зв'язок "один-до-багатьох" із таблицею `disciplinesCount` для зберігання інформації про кількість дисциплін для вибору студентами з певних груп за певний семестр певного курсу.

Таблиця `disciplinesCount` містить інформацію про кількість дисциплін, доступних для вибору здобувачами освіти певної академічної групи на певний семестр певного курсу. Має зв'язок "багато-до-одного" із таблицею `academicGroups`.

Таблиця `SequelizeMeta` є службовою таблицею, яку використовує ORM `Sequelize` для управління міграціями бази даних.

## 2.2 Робота з базою даних за допомогою ORM `Sequelize`

Міграції в `Sequelize` — це інструмент, який дозволяє відстежувати зміни у структурі бази даних і автоматизувати процеси створення, зміни або видалення таблиць, стовпців, індексів тощо. Міграції особливо корисні для підтримки бази даних у складних проєктах із багатьма розробниками, оскільки вони дозволяють впроваджувати зміни у контрольованій та прозорій формі.

Sequelize відстежує, які міграції були застосовані, через службову таблицю `SequelizeMeta`. У цій таблиці зберігаються назви виконаних міграцій.

Кожний файл міграції містить опис змін, що застосовуються до бази даних. Кожна міграція складається з двох основних функцій:

- `up`: виконує зміни, наприклад, створює таблицю або додає стовпець;
- `down`: скасовує зміни, виконані функцією `up`, повертаючи базу даних до попереднього стану.

Міграції виконуються за допомогою CLI — інтерфейсу командного рядка `Sequelize`. Основні команди:

- `npx sequelize-cli migration:generate` — створення нового файлу міграції;
- `npx sequelize-cli db:migrate` — виконання всіх нових міграцій;
- `npx sequelize-cli db:migrate:undo` — скасування останньої міграції.

Приклад міграції для створення таблиці `users` наведений у лістингу 2.1.

### Лістинг 2.1 – Приклад міграції для створення 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 DEFAULT NOW(),
        updatedAt TIMESTAMP NOT NULL DEFAULT NOW(),

        CONSTRAINT usersAcademicGroupsGroupId FOREIGN KEY (groupId) REFERENCES
academicGroups(id) ON DELETE SET NULL ON UPDATE RESTRICT
      ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci`,
    );
  },

  async down(queryInterface, Sequelize) {
    await queryInterface.sequelize.query('DROP TABLE IF EXISTS users;');
  },
};
```

Для інтеграції Sequelize з додатком, розробленим на основі NestJS, необхідно виконати налаштування, яке включає підключення до бази даних, завантаження моделей і визначення параметрів взаємодії. Файл `app.module.ts` є основним модулем програми і містить конфігурацію підключення, наведений у лістингу 2.2.

### Лістинг 2.2 – 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, Discipline, StudentDiscipline, DisciplinesCount],
    }),
    UserModule,
    RedisCacheModule,
    SessionModule,
    GroupModule,
    DisciplineModule,
    StudentDisciplineModule,
    DisciplinesCountModule,
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

`SequelizeModule.forRoot` — це метод із модуля `SequelizeModule`, який використовується для підключення бази даних до додатка NestJS. Він конфігурує ORM Sequelize, дозволяючи взаємодіяти з реляційною базою даних на основі вказаних налаштувань. Метод `forRoot` приймає об'єкт, який описує параметри з'єднання та поведінки Sequelize.

Моделі у Sequelize визначають структуру таблиць, зв'язки між ними та забезпечують простий доступ до даних через методи, що абстрагують SQL-запити.

Однією з основних моделей є User, яка відповідає таблиці users у базі даних. Ця модель забезпечує зберігання інформації про користувачів системи, їх роль, групу, дисципліни та інші атрибути. Модель User представлена у лістингу 2.3.

### Лістинг 2.3 – Модель User

```
@Scopes(() => ({
  byId: (id: number) => ({ where: { id } }),
  byPage: (limit: number = null, offset: number = 0) => ({ limit, 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 } ], subQuery:
    false }),
  withGroupWithDisciplinesCount: () => ({
    include: [ { model: Group, subQuery: false, include: [ { model:
      DisciplinesCount } ] } ], subQuery: false }),
  withDisciplines: () => ({ include: [ { model: Discipline, through: {
    attributes: ['id', 'forSemester', 'forYear', 'isLocked'] } } ], subQuery: false
  }))))
@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;
@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;
@BelongsTo(() => Group)
group: Group;
@BelongsToMany(() => Discipline, () => StudentDiscipline)
disciplines: Discipline[];
}

```

Модель User описує структуру таблиці users, включаючи поля, типи даних, обмеження та інші атрибути. Щоб вказати, що модель відповідає таблиці users, використовується декоратор @Table. Поля timestamps: true автоматично додають стовпці createdAt і updatedAt.

Поля описуються за допомогою декоратора @Column. Можна вказати типи даних (DataType), обмеження (наприклад: AllowNull, Unique) та валідацію.

За допомогою декораторів визначаються зв'язки з іншими моделями, наприклад: User належить до однієї групи (BelongsTo), User пов'язаний із багатьма дисциплінами через проміжну таблицю StudentDiscipline (BelongsToMany).

Scope 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. Scopes



визначаються всередині моделі за допомогою декоратора `@Scopes()` і можуть містити умови, що стосуються вибору полів, фільтрів, обмежень, зв'язків тощо.

Для забезпечення документації API використовується декоратор `@ApiProperty`, який додає метадані про кожне поле моделі.

Кожну модель потрібно реєструвати для використання в конкретному модулі. `SequelizeModule.forFeature()` — це метод, який використовується для реєстрації моделей у межах окремого модуля в додатку NestJS. Він дозволяє зробити моделі доступними лише для цього модуля, забезпечуючи модульність і інкапсуляцію. Приклад реєстрації моделі `User` в модулі `UserModule` показаний у лістингу 2.4.

#### Лістинг 2.4 – Модуль `UserModule`

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

### 2.3 Серверна частина додатку

Серверна частина додатку побудована з акцентом на забезпечення функціональності, гнучкості та масштабованості за допомогою сучасних технологій і підходів. У її основі лежить фреймворк NestJS, який забезпечує модульну структуру, впровадження залежностей та просту інтеграцію сторонніх бібліотек. Серверна частина відповідає за виконання кількох ключових функцій:

обробка HTTP-запитів, управління бізнес-логікою, авторизація користувачів, кешування, документування API та інтеграція з клієнтською частиною.

Однією з основних характеристик серверної частини є її модульна архітектура. Кожен модуль додатку відповідає за конкретну частину функціональності, включаючи управління користувачами, академічними групами, дисциплінами та їх вибором. Наприклад, модуль UserModule містить контролери, сервіси та моделі, які забезпечують реєстрацію, авторизацію та управління користувачами. Подібним чином, модуль GroupModule обробляє дані, пов'язані з академічними групами, тоді як модуль DisciplineModule реалізує логіку управління дисциплінами.

Серверна частина побудована на принципах MVC (Model-View-Controller):

1. Моделі описують структуру даних і зв'язки між ними.
2. Контролери обробляють HTTP-запити та повертають відповіді клієнту.
3. Сервіси реалізують бізнес-логіку, працюючи з моделями та іншими ресурсами.

Особлива увага приділяється безпеці та захисту даних. Паролі користувачів зберігаються у зашифрованому вигляді з використанням сучасних алгоритмів хешування та солей, що запобігає їх компрометації у разі несанкціонованого доступу до бази даних. Авторизація в системі реалізована через стандарт JWT (JSON Web Token). Це дозволяє надійно ідентифікувати користувачів і забезпечити доступ до захищених ресурсів на основі їх ролей. Ролі користувачів визначаються на рівні моделі та перевіряються за допомогою Guard'ів у контролерах, що дозволяє забезпечити гнучке розмежування прав доступу. Наприклад, адміністраторам надається можливість управління усіма даними системи, тоді як здобувачі освіти мають доступ лише до інформації про свої дисципліни. Використання JWT-токенів у поєднанні з гвардами безпеки дозволяє контролювати доступ до маршрутів та ресурсів, забезпечуючи розмежування прав між різними ролями користувачів.

Кешування реалізовано за допомогою Redis, що дозволяє оптимізувати роботу додатку, зменшуючи навантаження на базу даних і прискорюючи виконання запитів. Redis використовується для зберігання сесій користувачів, JWT-токенів та інших часто запитуваних даних. Наприклад, під час авторизації користувача створюється токен, який зберігається у Redis і перевіряється при кожному наступному запиті до захищеного ресурсу.

Документування API забезпечується за допомогою Swagger, що дозволяє автоматично генерувати актуальну документацію для всіх маршрутів, описаних у контролерах. Це не лише полегшує тестування ендпоінтів, але й забезпечує простоту інтеграції клієнтської частини з сервером.

Серверна частина також підтримує різні середовища розробки та розгортання. Використання .env-файлів дозволяє легко налаштувати конфігурацію додатку, наприклад, змінювати параметри підключення до бази даних, секретні ключі для JWT або параметри Redis залежно від середовища (локальне, тестове чи продакшн).

Завдяки використанню NestJS додаток побудований на принципах інверсії управління (IoC), що забезпечує чітке розділення залежностей між модулями та спрощує їх тестування. Це дозволяє масштабувати додаток, додавати нові модулі чи змінювати існуючі без значних ризиків для роботи системи. Наприклад, модуль вибору дисциплін студентами використовує сервіси модулів користувачів та дисциплін для отримання необхідних даних та виконання перевірок. Це дозволяє легко замінювати або модифікувати окремі компоненти без впливу на інші частини системи.

Обробка помилок та винятків здійснюється централізовано з використанням спеціальних фільтрів та інтерцепторів. Це дозволяє надавати клієнту зрозумілі повідомлення про помилки та забезпечує логування критичних подій для подальшого аналізу та усунення проблем. Система логування також

використовується для моніторингу роботи додатку та відстеження продуктивності.

Система розроблена з урахуванням можливості горизонтального та вертикального масштабування. Завдяки використанню NestJS та модульної архітектури, додавання нових функцій або розподіл навантаження між кількома серверами не становить складності. Це забезпечує готовність системи до збільшення кількості користувачів та обсягів даних без втрати продуктивності.

### 2.3.1 Основні модулі додатку

Серверна частина додатку організована у вигляді модульної структури, кожен модуль відповідає за конкретний аспект функціональності системи.

Усі модулі розроблені відповідно до принципів модульності та інверсії управління. Вони взаємодіють один із одним через сервіси, використовуючи чітко визначені інтерфейси. Це дозволяє ізолювати функціонал кожного модуля, що спрощує підтримку, тестування та розширення системи. Структура забезпечує високу продуктивність, масштабованість і підтримуваність серверної частини додатку.

Основні функціональні модулі додатку та їх призначення:

1. `UserModule` — модуль користувачів, управляє даними користувачів системи, такими як адміністратори, викладачі та здобувачі освіти. Він обробляє реєстрацію, автентифікацію, отримання списку користувачів за ролями, отримання авторизованого користувача та управління обліковими записами. Цей модуль визначає модель `User`. Також модуль містить бізнес-логіку для перевірки унікальності email і роботи з ролями користувачів.

2. `GroupModule` — модуль академічних груп, відповідає за управління академічними групами, до яких належать студенти. Цей модуль визначає модель `Group`. Групи пов'язані з користувачами через модель `User`. У модулі реалізовано

логіку для створення, редагування та видалення груп, отримання списків студентів, які належать до конкретної групи, відомостей про вибір дисциплін для групи для певного семестру і курсу та інше.

3. `DisciplineModule` — модуль дисциплін, забезпечує управління навчальними дисциплінами. Цей модуль визначає модель `Discipline`. Модуль надає механізми для створення, редагування та видалення дисциплін, а також логіку для отримання списку дисциплін, доступних для вибору чи перевибору студентами.

4. `DisciplinesCountModule` — модуль, який реалізує обмеження на кількість дисциплін, які студенти певної групи можуть обрати на певний семестр і курс. Він містить модель `DisciplinesCount`, яка пов'язана з групами. Модуль використовується для управління цими обмеженнями, перевірки їх під час вибору чи перевибору дисциплін здобувачами освіти та забезпечення відповідності встановленим правилам.

5. `StudentDiscipline` — модуль, що відповідає за логіку вибору дисциплін студентами. Він містить проміжну модель `StudentDiscipline`, яка пов'язує студентів із дисциплінами. У цьому модулі реалізовані механізми перевірки доступності дисциплін, оброблення обмежень та збереження вибору і перевибору навчальних дисциплін здобувачами освіти.

Основні інфраструктурні модулі додатку та їх призначення:

1. `RedisModule` — модуль для інтеграції з `Redis`, використовується для кешування даних. Він оптимізує продуктивність додатку, зберігаючи тимчасові дані, такі як токени автентифікації. Реалізує конфігурацію `Redis` та продукує сервіс для доступу до `Redis`.

2. `SessionModule` — модуль сесій, управляє автентифікацією та авторизацією користувачів. Він працює з JWT-токенами та забезпечує перевірку прав доступу до захищених ресурсів. У цьому модулі визначені механізми створення, зберігання та валідації сесій користувачів.

3. `StrategiesModule` — модуль стратегій, реалізує різні підходи до авторизації користувачів. Основним компонентом цього модуля є стратегія `JwtStrategy`, яка використовується для перевірки валідності токенів та ідентифікації користувачів у захищених маршрутах.

4. `GuardsModule` — модуль, що відповідає за управління доступом до маршрутів і ресурсів. Він реалізує логіку перевірки ролей користувачів для доступу до приватних ендпоінтів.

5. `CustomValidatorsModule` — модуль, що містить спеціальні валідатори, які використовуються для перевірки даних, що надходять у запитах.

6. `ResourcesModule` — модуль ресурсів, який відповідає за управління статичними ресурсами і даними, які використовуються в системі.

### 2.3.2 Основна робота з сутностями. `Module`, `Service`, `Controller`

Схема взаємодії компонентів у типовому функціональному модулі застосунку виглядає таким чином:

1. `Controller` приймає запит, використовує `Schema` для валідації даних, передає їх у `Service`.

2. `Service` обробляє логіку, взаємодіє з `Model` (концепція моделі описана у пункті 2.2), отримує або оновлює дані у базі даних чи виконує певну бізнес-логіку.

3. Результат передається назад у `Controller`, який формує відповідь клієнту у вигляді `DTO`.

Ці компоненти є основою сучасної архітектури на базі фреймворків, таких як `NestJS`, і забезпечують структурованість, модульність та легкість у підтримці програмного забезпечення.

`Module` є фундаментальною концепцією в модульній архітектурі. У `NestJS` модуль — це контейнер для логічно пов'язаних компонентів, таких як

контролери, сервіси, провайдери та інші модулі. Модулі дозволяють ізолювати логіку певної функціональної області програми (наприклад, управління користувачами, сесіями, дисциплінами). Модулі приховують реалізацію компонентів і відкривають лише ті сервіси, які потрібні іншим модулям через `exports`. Завдяки модулям код стає більш структурованим, а проєкт — легшим у підтримці. Кожен модуль інкапсулює певний функціонал і може імпортувати інші модулі для використання їхніх сервісів або провайдерів.

Основний модуль додатку це `AppModule`, у якому зазвичай відбувається конфігурація певних частин системи, таких як взаємодія з базою даних чи налаштування модуля конфігурації, але в додатку може бути багато модулів для різних частин системи. `AppModule` наведений у лістингу 2.2 під час опису конфігурації `Sequelize`.

`UserModule` — це модуль у проєкті, який відповідає за обробку всіх операцій, пов'язаних із користувачами. Він служить контейнером для всіх компонентів, необхідних для роботи з користувачами, таких як контролери, сервіси, моделі та додаткові залежності. `UserModule` наведено у лістингу 2.5.

### Лістинг 2.5 – `UserModule`

```
import { Module } from '@nestjs/common';
import { UserController } from './user.controller';
import { UserService } from './user.service';
import { SequelizeModule } from '@nestjs/sequelize';
import { User } from './user.model';
import { IsUnique } from 'src/custom-validators/IsUnique';
import { RedisCacheModule } from 'src/redis-cache/redis-cache.module';
import { GroupModule } from '../group/group.module';
import { DisciplineModule } from '../discipline/discipline.module';
import { StudentDisciplineModule } from '../student-discipline/student-discipline.module';

@Module({
  imports: [
    SequelizeModule.forFeature([User]),
    RedisCacheModule,
    GroupModule,
    DisciplineModule,
    StudentDisciplineModule,
  ],
  controllers: [UserController],
})
```

```

    providers: [UserService, IsUnique],
    exports: [UserService],
  })
  export class UserModule {}

```

Декоратор `@Module()` повідомляє NestJS, що клас, до якого він застосований, є модулем. Без цього декоратора клас не буде розпізнано як модуль, і NestJS не зможе правильно обробити його компоненти. `@Module()` приймає об'єкт із метаданими, які описують, як компоненти модуля взаємодіють один з одним і з іншими модулями.

При описі модуля у секцію `imports` додаються модулі, що будуть використовуватись структурними елементами даного модуля, провайдери сторонніх сервісів, модулі конфігурації і т.д.

До `UserModule` імпортовано:

1. `SequelizeModule.forFeature([User])` — цей метод інтегрує модель `User` з ORM `Sequelize`, дозволяючи сервісам і контролерам виконувати операції з таблицею `users`.

2. `RedisCacheModule` — використовується для інтеграції сервісу кешування через `Redis` для отримання JWT-токенів авторизації.

3. Модулі `GroupModule`, `DisciplineModule`, `StudentDisciplineModule` підключають інші функціональні модулі, пов'язані з групами, дисциплінами та виборами дисциплін, сервіси яких використовуються при обробленні імпортованих з файлів даних про минулі вибори навчальних дисциплін.

Секція `controllers` визначає контролери модуля. У даному випадку це `UserController`. Він відповідає за обробку HTTP-запитів, пов'язаних із користувачами. Цей контролер містить маршрути для створення, оновлення, видалення користувачів, а також для отримання інформації про користувачів і імпорту даних про минулі вибори навчальних дисциплін здобувачами освіти.

Секція `providers` визначає усі сервіси і провайдери модуля. У даному випадку це `UserService` та `IsUnique`. `UserService` — це сервіс, що реалізує всю



бізнес-логіку, пов'язану з користувачами. Він відповідає за виконання операцій у базі даних, обробку імпорту даних, а також взаємодію з іншими сервісами (GroupService, DisciplineService та ін). IsUnique — це кастомний валідатор, який перевіряє, чи є поле (для даного випадку це електронна пошта) унікальним у базі даних. Його використовують у схемах для валідації даних.

Секція `exports` описує, які саме компоненти модуля можуть бути експортовані для використання в інших модулях. У даному випадку з модуля експортується `UserService`.

`Service` — це шар бізнес-логіки, який відповідає за обробку даних і реалізацію основної функціональності програми. Сервіс зазвичай не залежить від транспортного шару (HTTP чи інше), що робить його незалежним від конкретної інфраструктури.

Сервіси використовуються для:

1. Обробки запитів і даних, виконання операцій над даними (створення, оновлення, видалення, пошук і т.д.), а також для іншої бізнес-логіки.
2. Інтеграції, тобто взаємодії з іншими сервісами, модулями чи зовнішніми API.

`DisciplineService` — це сервіс, який реалізує бізнес-логіку, пов'язану з обробкою даних про дисципліни. Він дозволяє створювати, оновлювати, видаляти, шукати дисципліни та виконувати додаткові операції з ними. У лістингу 2.6 наведено `DisciplineService`.

### Лістинг 2.6 – DisciplineService

```
@Injectable()
export class DisciplineService {
  constructor(
    @InjectModel(Discipline) private disciplineModel: typeof Discipline,
  ) {}
  async findOrCreate(
    disciplineData: any,
    transaction: Transaction,
  ): Promise<Discipline> {
    const [discipline] = await this.disciplineModel.findOrCreate({
```

```

    where: { code: disciplineData.code },
    defaults: {
      code: disciplineData.code,
      title: disciplineData.title,
    },
    transaction,
  });
  return discipline;
}
async count(scopes: any[] = []): Promise<number> {
  try {
    return await this.disciplineModel.scope(scopes).count();
  } catch (error) {throw new BadRequestException(error);} }
async create(payload: CreateOrUpdateDisciplineSchema): Promise<Discipline> {
  try {
    return await this.disciplineModel.create(payload);
  } catch (error) {throw new BadRequestException(error);} }
async findAll(scopes: any[] = []): Promise<Discipline[]> {
  try {return await this.disciplineModel.scope(scopes).findAll();} catch
(error) {throw new BadRequestException(error);} }
async findOne(id: number, scopes: any[]): Promise<Discipline> {
  const discipline = await this.disciplineModel.scope(scopes).findByPk(id);
  if (!discipline) {throw new NotFoundException();}
  return discipline;}
async update(id: number, data: CreateOrUpdateDisciplineSchema):
Promise<Discipline> {
  try {
    const discipline = await this.findOne(id, []);
    return await discipline.update(data);
  } catch (error) {throw new BadRequestException(error);} }
async remove(discipline: Discipline): Promise<void> {
  await discipline.destroy();}
}

```

`@Injectable()` — це декоратор у NestJS, який маркує клас як провайдер. Провайдер — це будь-який клас, який може бути інжектований в інші класи через механізм інжекції залежностей. Клас із цим декоратором може використовуватися як залежність в інших компонентах. Без цього декоратора клас не буде визнаний провайдером, і NestJS не зможе інжектувати його в інші компоненти. Це забезпечує декларативний підхід до управління залежностями.

`@InjectModel()` — це спеціальний декоратор, який використовується для інжекції моделі Sequelize в клас. У контексті NestJS цей декоратор пов'язує модель із сервісом через ORM Sequelize. У випадку `@InjectModel(Discipline)` сервіс отримує доступ до моделі Discipline. NestJS автоматично створює

екземпляр моделі `Discipline` та інjektує його в сервіс, який використовує цей декоратор.

`DisciplineService` забезпечує такі ключові операції:

1. `findOrCreate` — метод для пошуку або створення масиву дисциплін бази даних. Пошук наявності дисципліни відбувається за її унікальним кодом. Якщо дисципліна не існує, вона створюється з переданими параметрами `code` і `title`.

Метод підтримує `Sequelize` транзакції, що забезпечує цілісність даних у межах кількох взаємозалежних операцій. Метод використовується для створення масиву дисциплін під час імпорту даних про минулі вибори навчальних дисциплін.

2. `count` — повертає кількість певних дисциплін у базі даних. Метод дозволяє застосовувати фільтри через `scopes`, які можуть враховувати специфічні умови пошуку. У разі помилки в запиті повертає `BadRequestException`.

3. `create` — створює нову дисципліну в базі даних. Метод приймає дані у форматі `CreateOrUpdateDisciplineSchema`, повертає створену дисципліну або повертає `BadRequestException`, якщо під час створення виникають помилки. Схема `CreateOrUpdateDisciplineSchema` гарантує, що дані для створення або оновлення дисциплін відповідають визначеним правилам.

4. `findAll` — повертає список певних дисциплін. Метод підтримує використання `scopes` для фільтрації результатів (наприклад, за певними критеріями або зв'язками). У разі помилки повертає `BadRequestException`.

5. `findOne` — знаходить конкретну дисципліну за її ідентифікатором. Також підтримує фільтри через `scopes`. Якщо дисципліна не знайдена, повертає `NotFoundException`.

6. `update` — оновлює дані конкретної дисципліни. Метод спочатку перевіряє існування дисципліни через `findOne`. Після успішної перевірки оновлює дані дисципліни, використовуючи метод `update`. У разі помилок оновлення повертає `BadRequestException`.

7. remove — видаляє конкретну дисципліну з бази даних. Метод використовує метод destroy моделі Discipline для видалення запису.

Вся логіка, пов'язана з дисциплінами, зібрана в одному місці, що полегшує її підтримку. Загальні операції, такі як пошук, створення або оновлення дисциплін, реалізовані у вигляді методів, які можна викликати з інших частин програми. Використання виключень забезпечує обробку помилок і зворотний зв'язок для клієнта. Підтримка фільтрів дозволяє адаптувати запити до специфічних потреб.

Controller відповідає за отримання HTTP-запитів, передачу даних у сервіси та формування відповіді для клієнта. Контролер визначає маршрути програми, отримує дані від клієнта через параметри, тіла запиту або заголовки та відправляє відповідь клієнту. Контролер не містить логіки, а делегує її виконання сервісам, фокусуючись на транспортному шарі. Через використання мідлварів, гвардів чи інтерцепторів контролери можуть захищати маршрути від несанкціонованого доступу.

Контролер StudentDisciplineController відповідає за маршрути, пов'язані з дисциплінами студентів. Він реалізує кілька ключових дій, таких як створення, видалення, оновлення і перевибір дисциплін. Усі методи контролера захищені гвардом авторизації JwtAuthGuard, а також описані за допомогою декораторів для документації та логіки обробки HTTP-запитів. Контролер представлений у лістингу 2.7.

### Лістинг 2.7 – StudentDisciplineController

```
@ApiTags('StudentDisciplines')
@Controller('/api/studentDisciplines')
export class StudentDisciplineController {
  constructor(
    private readonly studentDisciplineService: StudentDisciplineService,
  ) {}
  @ApiOperation({ summary: 'Select disciplines' })
  @ApiBody({ description: 'Data', type: CreateStudentDisciplineSchema })
  @ApiResponse({ status: HttpStatus.NO_CONTENT, description: 'Success' })
  @ApiBearerAuth()
```

```

@UseGuards(JwtAuthGuard)
@Post()
async create(@Body() body: CreateStudentDisciplineSchema) {
  await this.studentDisciplineService.bulkCreate(body.data);
  return;
}
@ApiOperation({ summary: 'Reselect disciplines' })
@ApiBody({ description: 'Data', type: ReselectSchema })
@ApiResponse({ status: HttpStatus.NO_CONTENT, description: 'Success' })
@ApiBearerAuth()
@UseGuards(JwtAuthGuard)
@Post('/reselect')
async reselect(@Body() body: ReselectSchema) {
  await this.studentDisciplineService.bulkDelete(body.toDelete);
  await this.studentDisciplineService.bulkCreate(body.reselect);
  return;
}
@ApiOperation({ summary: 'Delete chosen discipline by id' })
@ApiResponse({ status: HttpStatus.OK, description: 'Success' })
@ApiResponse({ status: HttpStatus.NOT_FOUND, description: 'Not Found' })
@ApiBearerAuth()
@UseGuards(JwtAuthGuard)
@Delete('/:id')
async remove(@Param() params: NumberIdSchema) {
  const result = await this.studentDisciplineService.findOne(params.id, []);
  if (result) { await this.studentDisciplineService.remove(result); }
}
@ApiOperation({ summary: 'Updates isLocked field for student discipline with specific id' })
@ApiBody({ description: 'Data', type: SetLockSchema })
@ApiResponse({ status: HttpStatus.OK, description: 'Success' })
@ApiResponse({ status: HttpStatus.BAD_REQUEST, description: 'Bad Request' })
@ApiResponse({ status: HttpStatus.NOT_FOUND, description: 'Not Found' })
@ApiBearerAuth()
@UseGuards(JwtAuthGuard)
@Patch('/:id')
async update(@Param() params: NumberIdSchema, @Body() body: SetLockSchema) {
  await this.studentDisciplineService.update(params.id, body);
  return;
}
}

```

Контролер `StudentDisciplineController` відповідає за маршрути, пов'язані з дисциплінами студентів. Він реалізує кілька ключових дій, таких як створення (вибір), видалення, оновлення і перевибір навчальних дисциплін здобувачами освіти. Усі методи контролера захищені гвардом авторизації `JwtAuthGuard`, а також описані за допомогою декораторів для документації та логіки обробки HTTP-запитів.

Декоратори, які використовуються в контролері:

1. `@Controller` — вказує, що клас є контролером, і задає базовий маршрут для всіх його методів (у даному випадку це `/api/studentDisciplines`).
2. `@Post`, `@Delete`, `@Patch`, `@Put`, `@Get` і т.д. — визначають HTTP-метод для кожного маршруту. Наприклад, `@Post` для створення, `@Delete` для видалення.
3. `@Body`, `@Param` — використовуються для отримання даних із тіла запиту (`@Body`) або параметрів маршруту (`@Param`).
4. `@UseGuards` і `@ApiBearerAuth()` — слугують для забезпечення приватних маршрутів. `@UseGuards` вказує, що маршрут захищений гвардом `JwtAuthGuard`, який перевіряє наявність та валідність JWT-токена, а `@ApiBearerAuth` позначає, що маршрут потребує авторизації через токен. Докладніше система захищених маршрутів описана у пункті 2.3.3.
5. `@ApiTags` — додає до контролера тег `StudentDisciplines` для групування його ендпоінтів у Swagger-документації.
6. `@ApiOperation` — описує мету кожного ендпоінту, наприклад, "Select disciplines" для методу `create`.
7. `@ApiBody` — вказує структуру тіла запиту, що очікується від клієнта. Наприклад, для методу `create` тіло повинно відповідати `CreateStudentDisciplineSchema`.
8. `@ApiResponse` — описує можливі відповіді сервера, включаючи статуси HTTP та їхній опис. Наприклад, для методу `update` вказані статуси 200 (успіх), 400 (помилка запиту) і 404 (об'єкт не знайдено).

Контролер базується на класі `StudentDisciplineService`, який реалізує бізнес-логіку для операцій із дисциплінами студентів. Конструктор інжектує цей сервіс через механізм інжекції залежностей `NestJS`, забезпечуючи доступ до методів сервісу. Методи контролера:

1. Метод `create` — відповідає за вибір дисциплін здобувачами освіти на новий семестр і курс. Він отримує вхідні дані через `@Body()` у форматі,

визначеному `CreateStudentDisciplineSchema`. Виклик сервісного методу `bulkCreate` дозволяє одночасно додати кілька дисциплін для студентів. Метод повертає статус успіху (HTTP 204) без вмісту, що означає успішне виконання операції.

2. Метод `reselect` — дозволяє переобирати дисципліни при виникненні проблем з початковим вибором. Він приймає два набори даних через тіло запиту: ті, які потрібно видалити (`toDelete`), і ті, які потрібно додати (`reselect`). Спочатку викликається метод `bulkDelete`, щоб видалити вказані дисципліни, а потім `bulkCreate` для додавання нових. Цей метод також завершується статусом HTTP 204, сигналізуючи про успішне завершення.

3. Метод `remove` — реалізує видалення конкретної дисципліни за її ідентифікатором. Ідентифікатор передається через параметри запиту, які перевіряються за допомогою `NumberIdSchema`. Метод викликає `findOne` сервісу для перевірки існування дисципліни, а якщо така знайдена, видаляє її через `remove`. У разі успіху повертається статус HTTP 200, а якщо дисципліна не знайдена — статус HTTP 404.

4. Метод `update` — оновлює поле `isLocked` для певної дисципліни студента. Ідентифікатор дисципліни передається через параметри, а дані для оновлення — через тіло запиту у форматі `SetLockSchema`. Виклик методу `update` сервісу обробляє логіку оновлення. У разі успіху метод повертає статус HTTP 200, а якщо виникають проблеми з валідністю даних чи об'єкт не знайдено, повертаються відповідні статуси HTTP 400 або 404.

`Schema` — це опис структури даних і правил їх валідації. Схеми гарантують, що дані, які надходять від клієнта, відповідають певним стандартам. Схеми перевіряють, чи відповідають дані визначеним правилам (наприклад, довжина рядка, тип даних, унікальність значень) і запобігають внесенню некоректних чи шкідливих даних у систему. Схеми часто використовуються для створення автоматичної документації API (наприклад, через Swagger). Також

схеми можуть трансформувати дані в потрібний формат (наприклад, перетворення рядка у число). Приклад схеми для створення нового користувача представлений у лістингу 2.8.

### Лістинг 2.8 – CreateUserSchema

```
export class CreateUserSchema {
  @ApiProperty({ description: 'Full name', nullable: false, example: 'John Doe',
type: 'string' })
  @IsString()
  @IsNotEmpty()
  @MaxLength(500)
  name: string;
  @ApiProperty({
    description: 'Email', nullable: false, example: 'example@example.com', type:
'string' })
  @IsString()
  @IsNotEmpty()
  @IsEmail()
  @Validate(IsUnique, ['email'])
  email: string;
  @ApiProperty({ description: 'Password', nullable: false, example: 'password',
type: 'string' })
  @IsString()
  @IsNotEmpty()
  @MaxLength(255)
  password: string;
  @ApiProperty({ description: 'Role', nullable: false, example: '3', type:
'number' })
  @Type(() => Number)
  @IsInt()
  @IsPositive()
  @IsIn(validRoles, { message: `Role must be one of the following values:
${validRoles.join(', ')} ` })
  role: number;
  @ApiProperty({ description: 'Group ID', nullable: true, example: 1, type:
'number' })
  @IsOptional()
  @Type(() => Number)
  @IsInt()
  @IsPositive()
  groupId?: number = null;
}
```

DTO (Data Transfer Object) — це об'єкт передачі даних, який використовується для формування структурованих відповідей клієнту або передачі даних між різними частинами програми. DTO забезпечує єдиний формат даних у запитах і відповідях, що спрощує обмін даними між клієнтом і сервером. DTO дозволяє приховати внутрішню реалізацію моделі чи бази даних від клієнта.



DTO може повертати лише потрібні клієнту дані, виключаючи зайві поля. DTO, як і схеми, допомагає формувати документацію API, оскільки описує структуру даних, що передаються. Приклад DTO для групи представлений у лістингу 2.9.

### Лістинг 2.9 – GroupDto

```
export class GroupDto {
  @ApiProperty({ description: 'Group identifier', nullable: false, example: 1,
    type: 'integer' })
  id: number;
  @ApiProperty({
    description: 'Title', nullable: false, example: 'PZ-23m-2', type: 'string',
  })
  title: string;
  @ApiProperty({ description: 'Year', nullable: false, example: '1', type:
'integer' })
  year: number;
  @ApiProperty({
    description: 'Created at',
    nullable: false,
    example: '2024-07-03T19:32:40.000Z',
    type: 'timestamp',
  })
  createdAt: Date;
  @ApiProperty({
    description: 'Updated at',
    nullable: false,
    example: '2024-07-03T19:32:40.000Z',
    type: 'timestamp',
  })
  updatedAt: Date;
  @ApiProperty({
    type: [UserDto],
    description: 'List of users in the group',
  })
  users: UserDto[];
  @ApiProperty({
    type: [DisciplinesCountDto],
    description: 'List of disciplines counts',
  })
  disciplinesCounts: DisciplinesCountDto[];
  constructor(group: Group) {
    this.id = group.id;
    this.title = group.title;
    this.year = group.year;
    this.createdAt = group.createdAt;
    this.updatedAt = group.updatedAt;
    this.users = group.users?.map((user: User) => new UserDto(user)) || [];
    this.disciplinesCounts =
      group.disciplinesCount?.map((item) => new DisciplinesCountDto(item)) ||
      [];
  }
}
```

### 2.3.3 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 автентифікація та авторизація в додатку. Стратегії використовуються в поєднанні з модулями безпеки, такими як Passport, для реалізації різних методів автентифікації.

Стратегія визначає, як користувачі автентифікуються. Це може включати використання токенів JWT, сесій, OAuth, OpenID Connect тощо. Кожен тип автентифікації 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 автентифікації. 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 автентифікації, що робить додаток більш безпечним і масштабованим.

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

JWT стратегія використовується для автентифікації 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 автентифікації JWT зазвичай генерується на сервері при вході користувача та використовується для підтвердження його особи.

Після успішної автентифікації (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 автентифікувати 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 автентифікації. 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 автентифікацією 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 – Контролер `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 автентифікації. `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 автентифікації.

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 наявності дійсного JWT токена в залежності від обраного `Guard`. Використання `Guard` показано у лістингу 2.17.

## Лістинг 2.17 – Використання Guard

```
// На рівні контролера

@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) {
  ...
}
```

## 2.4 Клієнтська частина додатку

Клієнтська частина додатку побудована за принципами компонентно-орієнтованої архітектури, що характерно для сучасних веб-застосунків. У ній використовується Angular, який дозволяє створювати модульну структуру, де кожен функціональний блок відокремлений у вигляді компонента, забезпечуючи чітку ієрархію та повторюваність коду. Всі компоненти взаємодіють через шаблони, стилі та файли логіки, які відповідають за рендеринг, зовнішній вигляд і поведінку відповідних частин інтерфейсу.

Одним із ключових принципів є розділення відповідальностей. Наприклад, обробка даних, що надходять із серверної частини, виконується за допомогою спеціалізованих сервісів, які ізольовані від логіки відображення. Це дозволяє зменшити залежності між компонентами та спрощує тестування й модифікацію окремих частин системи. Інтерактивність забезпечується через форми, які обробляють події на стороні клієнта, наприклад, авторизацію або заповнення профілю.

Для управління маршрутизацією використовується вбудований механізм Angular Router, що дозволяє динамічно змінювати сторінки без

перезавантаження. Кожен маршрут чітко пов'язаний з компонентом, який відповідає за певну функціональність, що робить додаток інтуїтивно зрозумілим для користувачів.

Головною ціллю стилізації є забезпечення естетичного та зручного інтерфейсу. Глобальні стилі задають основні параметри зовнішнього вигляду, а специфічні компоненти використовують локальні стилі для точного налаштування елементів. Завдяки цьому досягається як послідовність дизайну, так і гнучкість в адаптації окремих елементів.

Клієнтська частина також активно використовує типізацію даних через інтерфейси, що забезпечує надійність та передбачуваність у роботі з об'єктами. Цей підхід дозволяє спрощувати обробку складних структур даних, таких як профілі користувачів чи статистичні дані.

Вся взаємодія з сервером реалізована через асинхронні запити, що забезпечує високу швидкість роботи інтерфейсу і дозволяє користувачам отримувати зворотний зв'язок без затримок. У разі виникнення помилок передбачений механізм відображення повідомлень, які інформують користувачів про необхідні дії.

Клієнтська частина активно використовує бібліотеку компонентів PrimeNG, що надає набір готових UI-елементів, таких як форми, таблиці, модальні вікна та повідомлення. Завдяки цій бібліотеці вдається швидко створювати функціональні та привабливі інтерфейси, значно скорочуючи час розробки. Наприклад, у компоненті авторизації використовується `MessageService` з PrimeNG для відображення інтерактивних повідомлень про помилки чи успішні дії. Це дозволяє зосередитись на бізнес-логіці додатку, залишивши реалізацію елементів інтерфейсу бібліотеці.

Окрім цього, клієнтська частина дотримується реактивного підходу за допомогою RxJS — потужної бібліотеки для роботи з асинхронними потоками даних. Реактивність проявляється у роботі з формами, запитами до сервера та

обробкою подій. Наприклад, запити до API виконуються через Observable, що дозволяє ефективно управляти потоками даних і реагувати на їх зміну. Такий підхід забезпечує гнучкість та високу продуктивність при роботі з великими обсягами даних.

RxJS також використовується для обробки складних сценаріїв, таких як ланцюгові виклики, управління станом додатку чи синхронізація даних між компонентами. Це дозволяє зменшити ризик помилок та підтримувати масштабованість клієнтської частини. Реактивний підхід у поєднанні з PrimeNG створює потужний інструмент для створення сучасних веб-додатків, які є одночасно швидкими, адаптивними та зручними для користувачів.

#### 2.4.1 Роутинг

Роутинг у клієнтській частині реалізований за допомогою механізму Angular Router, що забезпечує навігацію між різними сторінками застосунку без перезавантаження браузера. Цей підхід дозволяє створювати динамічні односторінкові додатки (SPA) з чітко визначеною структурою маршрутів.

Кожен маршрут відповідає за певний компонент, який завантажується при переході на відповідну URL-адресу. Всі маршрути налаштовуються в окремому файлі `app-routing.module.ts`, що є частиною модульної архітектури Angular. Основна логіка маршрутизації включає:

1. Статичні маршрути: визначають фіксовані шляхи, такі як `/login` для авторизації, `/profile` для сторінки профілю або `/main` для головної сторінки. Ці маршрути безпосередньо прив'язуються до відповідних компонентів.

2. Захищені маршрути: для забезпечення доступу до певних сторінок лише авторизованим користувачам використовується механізм `Guard`. Цей підхід перевіряє права доступу перед завантаженням компонента, перенаправляючи неавторизованих користувачів на сторінку входу.

3. **Lazy loading** (ліниве завантаження): для оптимізації продуктивності великі модулі можуть завантажуватися динамічно лише тоді, коли вони потрібні. Це дозволяє скоротити початковий час завантаження додатку.

Angular Router також підтримує функції перенаправлення та обробки помилок. Наприклад, якщо користувач вводить некоректний URL, додаток його перенаправляє на сторінку помилки (компонент `ErrorComponent`). Це дозволяє забезпечити інтуїтивно зрозумілий досвід навігації.

Роутинг є центральною частиною логіки клієнтської частини, яка забезпечує плавний перехід між сторінками, гнучкість у відображенні контенту та безпеку доступу до ресурсів, він показаний у лістингу 2.18.

### Лістинг 2.18 – Роутинг

```
const routes: Routes = [
  { path: '', component: MainComponent },
  { path: 'registration', component: RegistrationComponent },
  { path: 'authorization', component: AuthorizationComponent },
  { path: 'teachers', component: TeacherComponent },
  { path: 'groups', component: GroupComponent },
  { path: 'students', component: StudentComponent },
  { path: 'disciplines', component: DisciplineComponent },
  { path: 'add-disciplines', component: StudentChoiceComponent },
  { path: 'profile', component: ProfileComponent },
  { path: 'reselect-disciplines', component: ReselectComponent },
  { path: 'upload', component: UploadComponent },
  { path: '**', component: ErrorComponent },
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

### 2.4.2 Сервіси

Сервіси у клієнтській частині реалізують основну бізнес-логіку додатку, забезпечуючи централізовану обробку даних, взаємодію з API та спрощуючи обмін інформацією між компонентами. Вони написані у вигляді класів, які інжектуються в компоненти через механізм Angular Dependency Injection, що

дозволяє дотримуватись принципів модульності та повторного використання коду. Сервіси є невід'ємною частиною клієнтської частини, яка забезпечує абстракцію складної логіки та взаємодії з сервером. Вони підвищують читабельність, масштабованість та підтримуваність коду, що є ключовим фактором для розвитку проєкту.

Сервіси відповідають за комунікацію між клієнтською та серверною частинами додатку через HTTP-запити, які реалізуються за допомогою модуля `HttpClientModule`. Наприклад, сервіс `UserService` має методи для авторизації, отримання профілю користувача та збереження інформації в локальному сховищі. Всі запити є асинхронними, використовують `Observable` з бібліотеки `RxJS`, що дозволяє обробляти відповіді сервера в реактивному стилі.

Сервіси зосереджують бізнес-логіку, зменшуючи навантаження на компоненти. Наприклад, `DisciplineService` обробляє всі операції, пов'язані з дисциплінами: завантаження списків, створення, редагування, видалення та ін. Це дозволяє компонентам залишатися легкими та сфокусованими на відображенні даних.

Сервіси включають механізми обробки помилок на рівні HTTP-запитів. Наприклад, якщо сервер повертає помилку, сервіс може згенерувати відповідне повідомлення для користувача через інтеграцію з `MessageService (PrimeNG)`, інформуючи про проблему в зручному вигляді.

Сервіси побудовані так, щоб бути незалежними та повторно використовуваними. Наприклад, `GroupService` обслуговує різні компоненти, пов'язані з управлінням групами, без необхідності дублювати логіку.

Приклад сервіса для роботи з дисциплінами представлений у лістингу 2.19.

### Лістинг 2.19 – `DisciplinesService`

```
@Injectable({providedIn: 'root'})
export class DisciplineService {
  constructor(
    private http: HttpClient,
```

```

    private messageService: MessageService,
    private userService: UserService,
  ) {}
  private apiUrl = 'http://localhost:3000/api/disciplines';
  getAll(): Observable<any[]> {
    const headers = this.userService.getHeaders();
    return this.http.get<any[]>(`${this.apiUrl}`, { headers }).pipe(
      catchError((err) => {
        this.messageService.add({
          severity: 'error',
          summary: 'Помилка отримання даних!',
          detail: 'Не вдалося отримати список дисциплін...',
        }); throw err; }));
  }
  create(data: Object) {
    const headers = this.userService.getHeaders();
    return this.http.post(this.apiUrl, data, { headers }).pipe(
      catchError((err) => {
        this.messageService.add({
          severity: 'error',
          summary: 'Помилка створення дисципліни!',
          detail: 'Перевірте введені дані! Або могла статись помилка на сервері...',
        }); throw err; }));
  }
  delete(disciplineId: number): Observable<void> {
    const headers = this.userService.getHeaders();
    return this.http.delete<void>(`${this.apiUrl}/${disciplineId}`, { headers
  }).pipe(
    catchError((err) => {
      this.messageService.add({
        severity: 'error',
        summary: 'Помилка видалення!',
        detail: 'Не вдалося видалити дисципліну.',
      }); throw err; }));
  }
  update(disciplineId: number, data: any): Observable<void> {
    const headers = this.userService.getHeaders();
    return this.http.put<any>(`${this.apiUrl}/${disciplineId}`, data, { headers
  }).pipe(
    catchError((err) => {
      this.messageService.add({
        severity: 'error',
        summary: 'Помилка редагування дисципліни!',
        detail: 'Перевірте введені дані! Або могла статись помилка на сервері...',
      }); throw err; }));
  }
  getBySemesterAndYear(semester: number, year: number, excludesId: any[]):
  Observable<any[]> {
    const headers = this.userService.getHeaders();
    let params = new HttpParams();
    params = params.append('semester', semester.toString());
    params = params.append('year', year.toString());
    excludesId.forEach((id: any) => {
      params = params.append('id', id.toString());
    });
    return this.http.get<any[]>(`${this.apiUrl}`, { headers, params }).pipe(
      catchError((err) => {
        this.messageService.add({
          severity: 'error',
          summary: 'Помилка отримання даних!',
          detail: 'Не вдалося отримати список дисциплін...',
        }); throw err; }));
  }

```

### 2.4.3 Компоненти

Компоненти у клієнтській частині — це основні будівельні блоки, які відповідають за рендеринг інтерфейсу, обробку подій і взаємодію з користувачем. Вони реалізують компонентно-орієнтований підхід, що дозволяє розбити додаток на окремі, незалежні частини, кожна з яких має свою специфічну роль.

Кожен компонент складається з трьох частин:

- шаблон (HTML) — визначає структуру та розташування елементів інтерфейсу;
- логіка (TypeScript) — містить основний функціонал, обробку даних і подій;
- стилі (CSS) — забезпечують вигляд компонента.

Ця ізоляція дозволяє легко тестувати, повторно використовувати та підтримувати компоненти.

Компоненти згруповані за функціональними зонами: авторизація, профіль користувача, управління дисциплінами, вибір дисциплін, сторінка помилок тощо. Це полегшує навігацію в коді та підтримку.

В додатку використовуються два види компонентів:

1. Презентаційні компоненти — фокусуються на відображенні даних, залишаючи обробку логіки на контейнерні компоненти.
2. Контейнерні компоненти — об'єднують кілька дочірніх компонентів і відповідають за бізнес-логіку та взаємодію з сервісами.

Презентаційні компоненти представлені компонентами з бібліотеки PrimeNG. PrimeNG у клієнтській частині виступає ключовою бібліотекою для створення сучасного та інтерактивного інтерфейсу. Її використання дозволяє значно скоротити час розробки завдяки готовим UI-компонентам, які легко інтегруються в архітектуру Angular. PrimeNG забезпечує доступ до широкого

спектру елементів, таких як форми, таблиці, повідомлення, модальні вікна та адаптивні меню. Це дає змогу зосередитися на логіці додатку, залишивши реалізацію базових елементів інтерфейсу бібліотеці.

Робота з формами стає значно зручнішою завдяки інтерактивним полям вводу, таким як текстові поля, списки вибору або чекбокси. У поєднанні з реактивними формами Angular ці елементи дозволяють гнучко керувати станом полів, здійснювати валідацію та динамічно реагувати на зміни даних. Наприклад, система валідації автоматично відображає повідомлення про помилки, якщо користувач вводить неправильний формат даних.

Повідомлення про дії або помилки відображаються за допомогою механізму toast, інтегрованого з PrimeNG. Це забезпечує швидкий зворотний зв'язок для користувачів, дозволяючи їм миттєво розуміти результат своїх дій. Такі повідомлення ненав'язливо зникають через певний час або можуть бути закриті вручну, що створює комфортний досвід взаємодії.

Для відображення великих обсягів даних бібліотека пропонує потужні таблиці з підтримкою сортування, фільтрації та пагінації. Це особливо корисно для управління списками користувачів, груп чи дисциплін, де потрібно швидко орієнтуватися в інформації. Таблиці адаптивні й зручні для роботи як на настільних комп'ютерах, так і на мобільних пристроях.

PrimeNG також активно використовується для створення модальних вікон, які застосовуються для підтвердження дій, редагування даних або відображення додаткової інформації. Їхнє плавне відображення та закриття, а також можливість кастомізації забезпечують естетичність і зручність.

Одним із ключових аспектів бібліотеки є її адаптивність. Більшість компонентів автоматично змінюють свій вигляд і поведінку залежно від розміру екрану. Це дозволяє створювати інтерфейс, який однаково добре виглядає як на мобільних пристроях, так і на великих екранах.



Приклад діалогового вінка на зміни паролю, побудований за допомогою компонентів PrimeNG показаний у лістингу 2.20.

### Лістинг 2.20 — Компонент діалогового вікна

```
<p-dialog
  [(visible)]="displayEditUserDialog"
  [header]=" 'Зміна пароля' "
  [closable]="true"
  [modal]="true"
  [style]="{width: '600px'}"
  [baseZIndex]="10000">
  <ng-template pTemplate="content">
    <form [formGroup]="editUserForm">
      <div class="p-fluid form-container">
        <div class="p-field fullSize">
          <p-inputGroup>
            <span class="p-inputgroup-addon">
              <i class="pi pi-key"></i>
            </span>
            <input
              id="password"
              type="password"
              pInputText
              formControlName="password"
              [minlength]="6"
              required
              placeholder="Пароль"
            />
          </p-inputGroup>
          <small *ngIf="editUserForm.get('password')?.invalid &&
editUserForm.get('password')?.touched"
            class="p-error">
            Введіть коректний пароль!
          </small>
        </div>
      </div>
    </form>
  </ng-template>
  <ng-template pTemplate="footer">
    <button pButton label="Зберегти" icon="pi pi-check"
(click)="saveEditedUser()"
      [disabled]="editUserForm.invalid"></button>
    <button pButton label="Скасувати" icon="pi pi-times"
(click)="closeEditUserDialog()"
      class="p-button-text"></button>
  </ng-template>
</p-dialog>
```

Клієнтська частина активно використовує реактивний підхід до створення форм за допомогою Angular ReactiveFormsModule. Реактивні форми

забезпечують динамічний і гнучкий спосіб управління полями вводу, їх перевіркою та взаємодією з іншими частинами додатку.

Особливості реактивних форм:

1. Форми створюються у TypeScript-кодi через об'єкти `FormGroup` та `FormControl`. Це дозволяє легко управляти станом кожного поля, включаючи значення та валідатори.

2. Реактивні форми дозволяють додавати валідатори для перевірки вводу користувача, наприклад, обов'язкові поля, правильний формат email чи мінімальну довжину пароля. Усі перевірки відображаються в реальному часі без перезавантаження сторінки.

3. Використання `RxJS` дозволяє реагувати на зміни у формі за допомогою підписок (`valueChanges`). Наприклад, зміна одного поля може динамічно оновлювати інші поля або виконувати запит до сервера.

4. Форми мають чітко визначені стани: `valid`, `invalid`, `pristine`, `dirty`. Це дозволяє будувати більш передбачуваний користувацький інтерфейс, наприклад, блокувати кнопку "Увійти", поки всі поля не заповнені правильно.

У компоненті авторизації використовується форма, що включає поля для введення email та пароля. Після заповнення форми викликається метод `onSubmit`, який передає дані на сервер через сервіс. Якщо сервер повертає помилку, компонент відображає відповідне повідомлення. Робота з реактивною формою авторизації показана у лістингу 2.21.

### Лістинг 2.21 — Реактивна форма авторизації

```
// ініціалізація форми
this.loginForm = this.fb.group({ email: ['', [Validators.required,
Validators.email]], password: ['', [Validators.required,
Validators.minLength(6)]] });

// підтвердження авторизації і валідність введених даних у форму
onSubmit() {
  if (this.loginForm.invalid) {
```

```

        this.messageService.add({ severity: 'error', summary: 'Помилка!', detail:
'Перевірте дані!' });
        return;
    }

    this.userService.authorize(this.loginForm.value).subscribe(response => {
        this.router.navigate(['/profile']);
    }, error => {
        this.messageService.add({ severity: 'error', summary: 'Помилка
авторизації', detail: 'Невірний логін або пароль' });
    });
}

```

Компоненти взаємодіють через Input/Output декоратори, сервіси та події. Наприклад, компонент Header може отримувати дані про авторизованого користувача через сервіс, а компонент Profile працювати з ними для відображення персоналізованої інформації.

Компоненти інтегруються з глобальними стилями для уніфікованого вигляду, але також використовують локальні стилі для точного налаштування. Завдяки цьому інтерфейс виглядає консистентно, незалежно від масштабу чи складності додатку.

## 3 ПРАКТИЧНІ РЕЗУЛЬТАТИ

### 3.1 Приклади роботи продукту

Згенерована Swagger документація для серверної частини додатку показана на рисунках 3.1 – 3.4.



Рисунок 3.1 – Автозгенеровані схеми і DTO

```

UserDto {
  session* {
    string
    nullable: false
    example: 65bm8u56uby95m8uy95mu9d4x98xm48f594bm45etu54m5
    Auth token
  }
  id* > [...]
  name* > [...]
  email* {
    string
    nullable: false
    example: example@example.com
    Email
  }
  role* > [...]
  groupId* > [...]
  group* GroupDto {
    id* > [...]
    title* > [...]
    year* > [...]
    createdAt* > [...]
    updatedAt* > [...]
    users* {
      List of users in the group
    }
    disciplinesCounts* {
      List of disciplines counts
    }
  }
  disciplines* {
    DisciplineDto {
      id* > [...]
      code* > [...]
      title* > [...]
      semester* > [...]
      year* > [...]
      forSemester* > [...]
      forYear* > [...]
      studentDisciplineId* > [...]
      isLocked* > [...]
      catalogueType* > [...]
      educationalLevel* > [...]
      link* > [...]
      createdAt* > [...]
      updatedAt* > [...]
      users* {
        List of students
      }
    }
  }
  studentDisciplineId* {
    number
    nullable: true
    example: 1
    Student discipline identifier
  }
  forSemester* > [...]
  forYear* > [...]
  isLocked* > [...]
  createdAt* > [...]
  updatedAt* > [...]
}

```














Рисунок 3.2 – Приклад DTO (UserDto)

## DNUChoice API 1.0 OAS 3.0












DNUChoice API on NestJS.

Authorize 

## Users

GET	/api/users	Gets all users	 
POST	/api/users	Creates new user	
GET	/api/users/me	Gets authorized user	 
GET	/api/users/{id}	Gets a user with specified id	 
PUT	/api/users/{id}	Updates a user with specified id	 
DELETE	/api/users/{id}	Delete user by id	 
POST	/api/users/import	Import students from file	 









## Groups

POST	/api/groups	Creates new group		
GET	/api/groups	Gets all groups		
GET	/api/groups/{id}	Gets a group with specified id		
PUT	/api/groups/{id}	Updates a group with specified id		
DELETE	/api/groups/{id}	Delete group by id		
PATCH	/api/groups/year	Updates a group with specified id		

## Disciplines

POST	/api/disciplines	Creates new discipline	🔒	▼
GET	/api/disciplines	Gets all disciplines	🔒	▼
GET	/api/disciplines/{id}	Gets a discipline with specified id	🔒	▼
PUT	/api/disciplines/{id}	Updates a discipline with specified id	🔒	▼
DELETE	/api/disciplines/{id}	Delete discipline by id	🔒	▼

## StudentDisciplines

POST	/api/studentDisciplines	Select disciplines	 
POST	/api/studentDisciplines/reselect	Reselect disciplines	 
DELETE	/api/studentDisciplines/{id}	Delete chosen discipline by id	 
PATCH	/api/studentDisciplines/{id}	Updates isLocked field for student discipline with specific id	 

## Sessions

**POST** /sessions Creates new session

**DELETE** /sessions Deletes the session

### DisciplinesCounts

**POST** `/api/disciplinesCounts` Creates new disciplines count

**PUT** `/api/disciplinesCounts/{id}` Updates a discipline count with specified id

Schemas ▼

Рисунок 3.3 – Згенеровані API

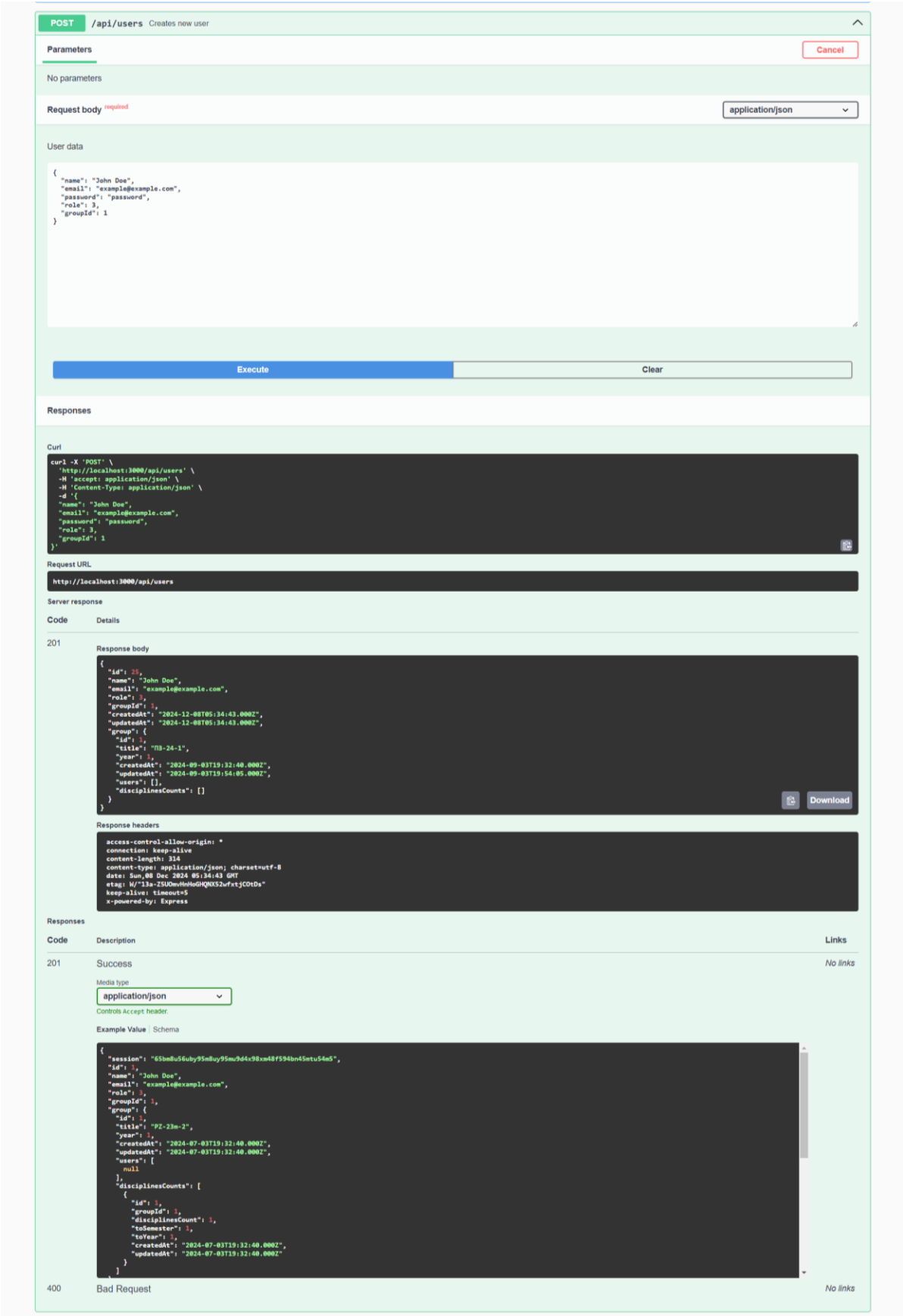


Рисунок 3.4 – Приклад роботи API (реєстрація нового користувача)

Приклади роботи із застосунком для неавторизованого користувача показані на рисунках 3.5 – 3.8.



Рисунок 3.5 – Головна сторінка для неавторизованого користувача

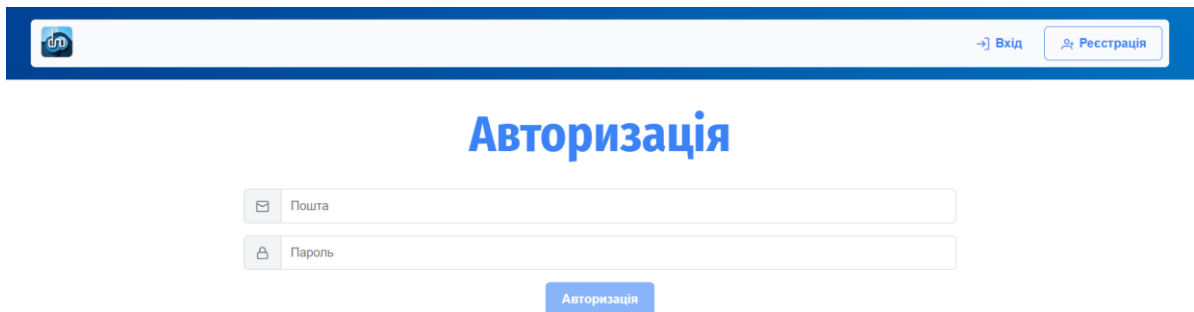


Рисунок 3.6 – Сторінка авторизації

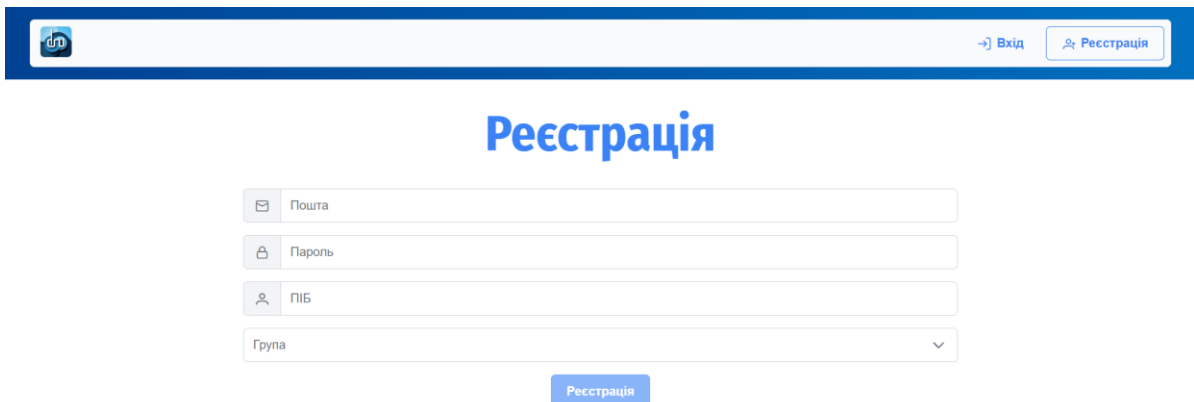
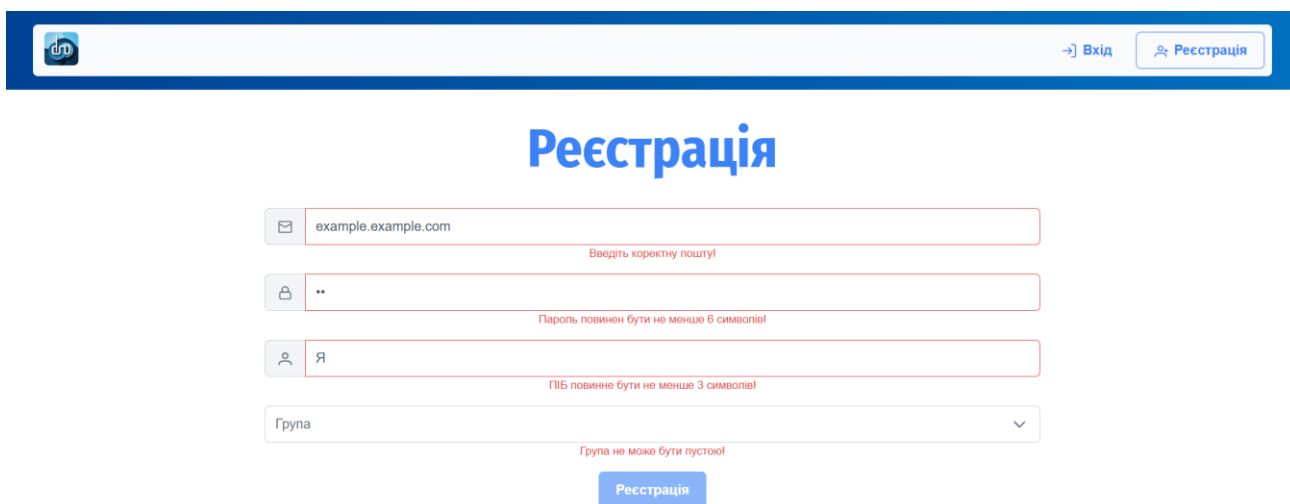


Рисунок 3.7 – Сторінка реєстрації





The registration form is titled "Реєстрація" (Registration). It contains five input fields with the following validation messages:

- Email: "example.example.com" with the message "Введіть коректну пошту!" (Enter correct email!).
- Password: "\*\*\*\*" with the message "Пароль повинен бути не менше 6 символів!" (Password must be at least 6 characters!).
- First Name: "Я" with the message "ПІБ повинне бути не менше 3 символів!" (Full name must be at least 3 characters!).
- Group: A dropdown menu with the value "Група" and the message "Група не може бути пустою!" (Group cannot be empty!).

A blue button labeled "Реєстрація" (Registration) is located below the form fields.

Рисунок 3.8 – Сторінка реєстрації з валідацією

Приклади роботи із застосунком для користувача з роллю «Адміністратор» показані на рисунках 3.9 – 3.27.

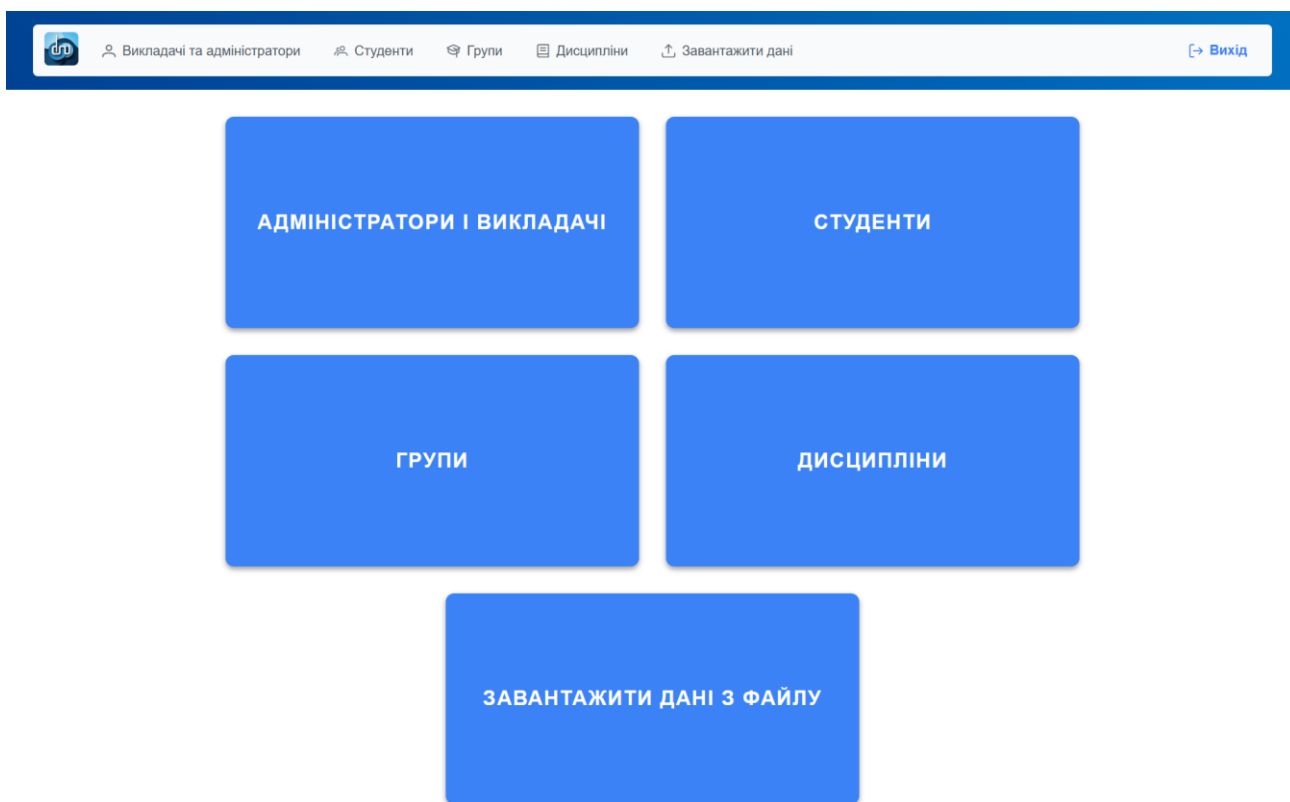


Рисунок 3.9 – Головна сторінка для адміністратора

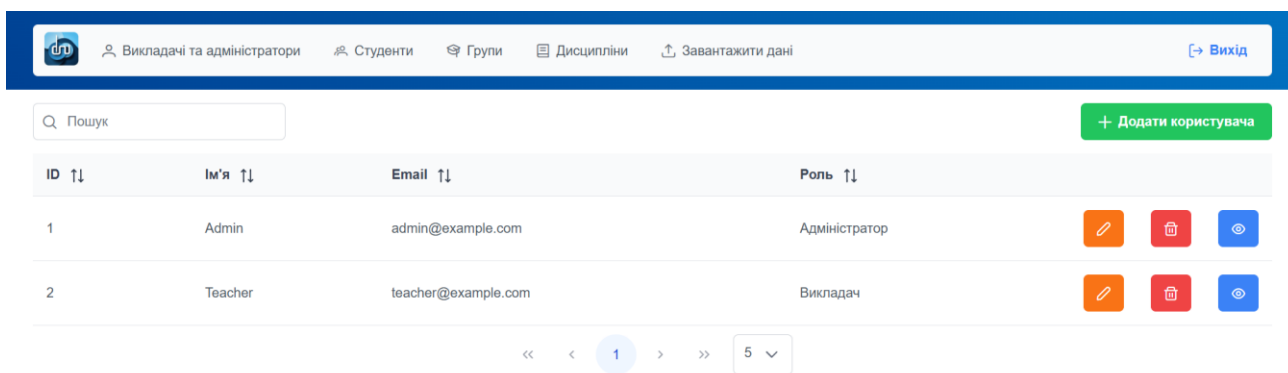


Рисунок 3.10 – Сторінка викладачів та адміністраторів

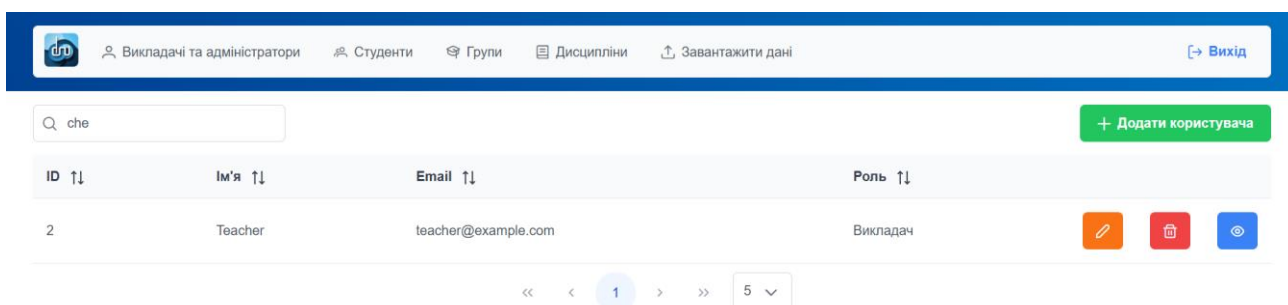


Рисунок 3.11 – Пошук

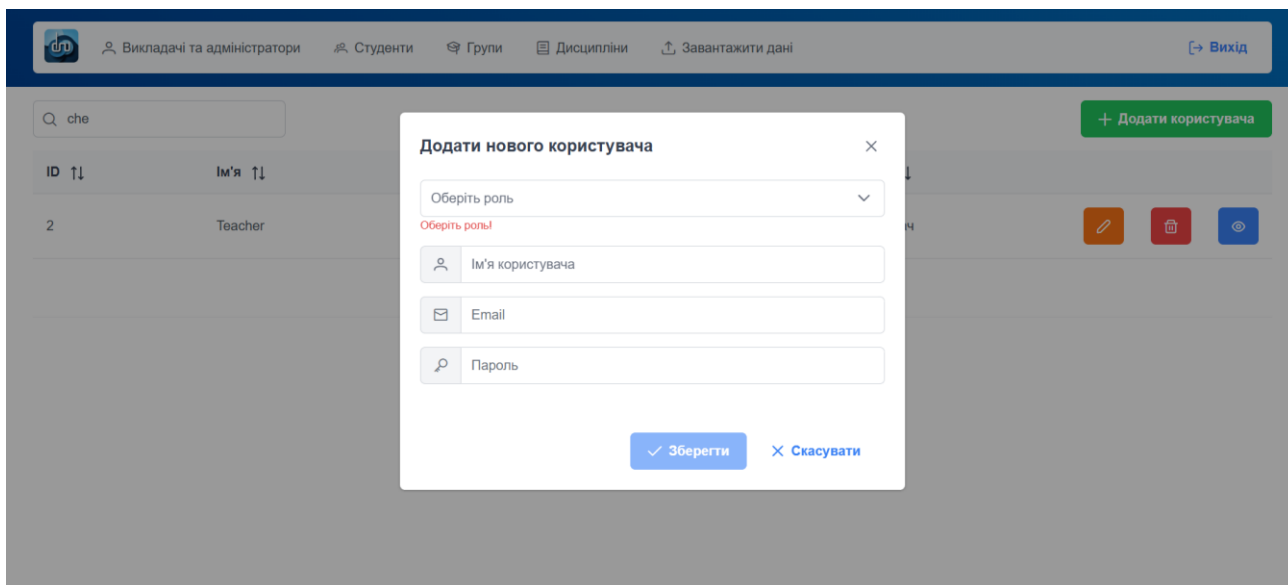


Рисунок 3.12 – Додавання нового користувача

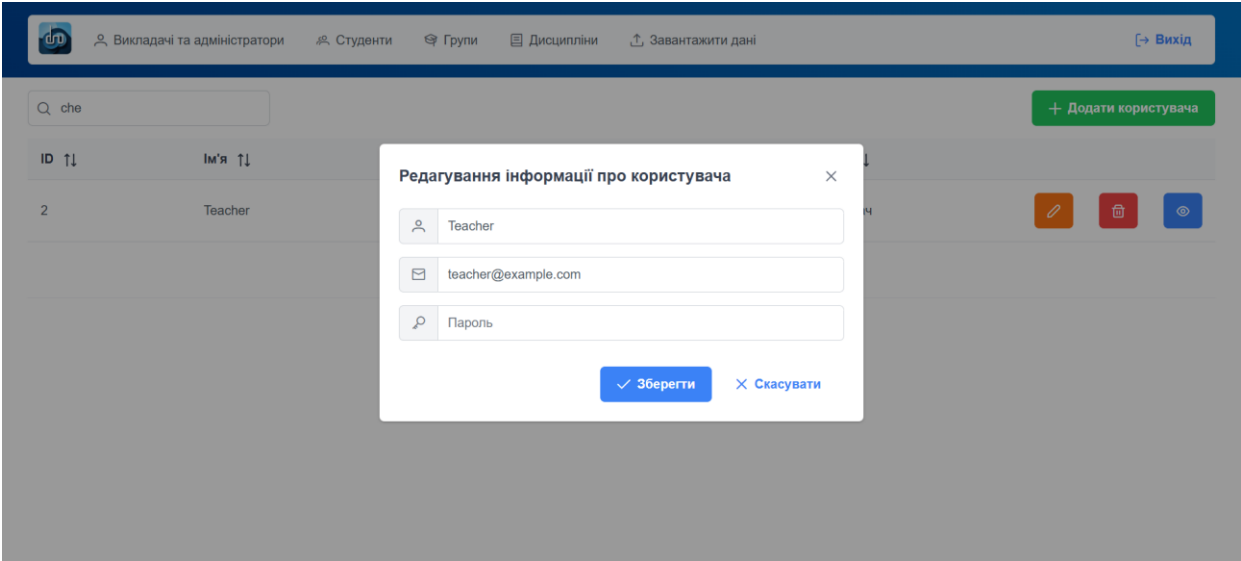


Рисунок 3.13 – Редагування інформації про викладача

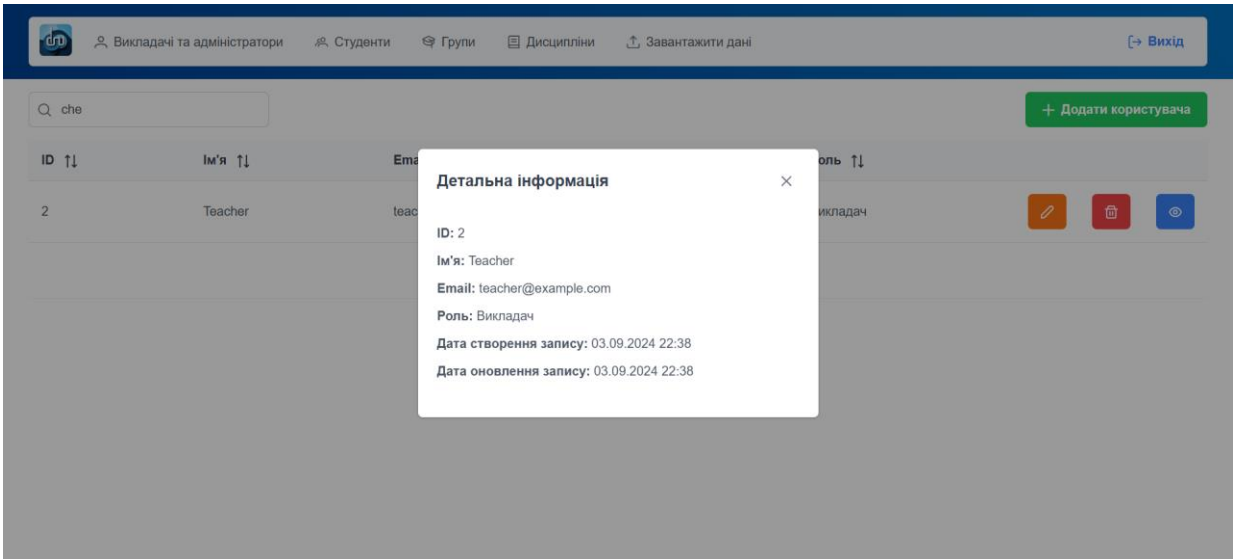


Рисунок 3.14 – Детальна інформація про користувача

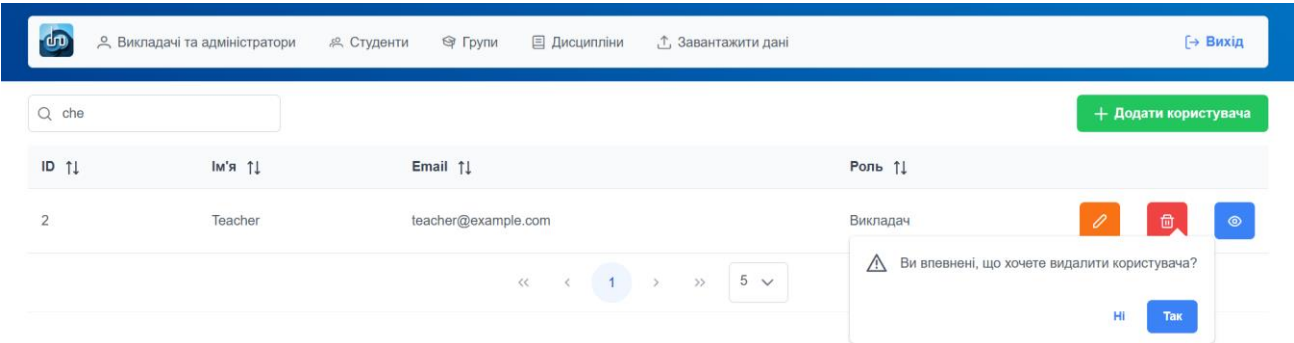


Рисунок 3.15 – Підтвердження видалення користувача

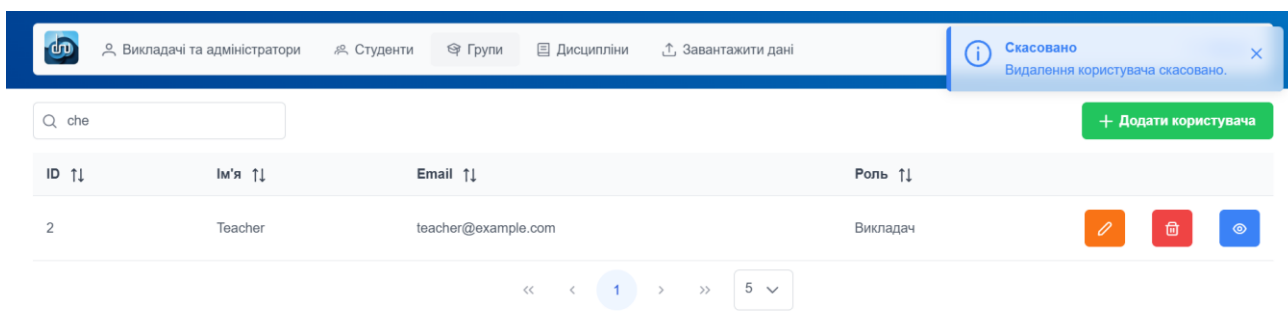


Рисунок 3.16 – Повідомлення щодо видалення користувача

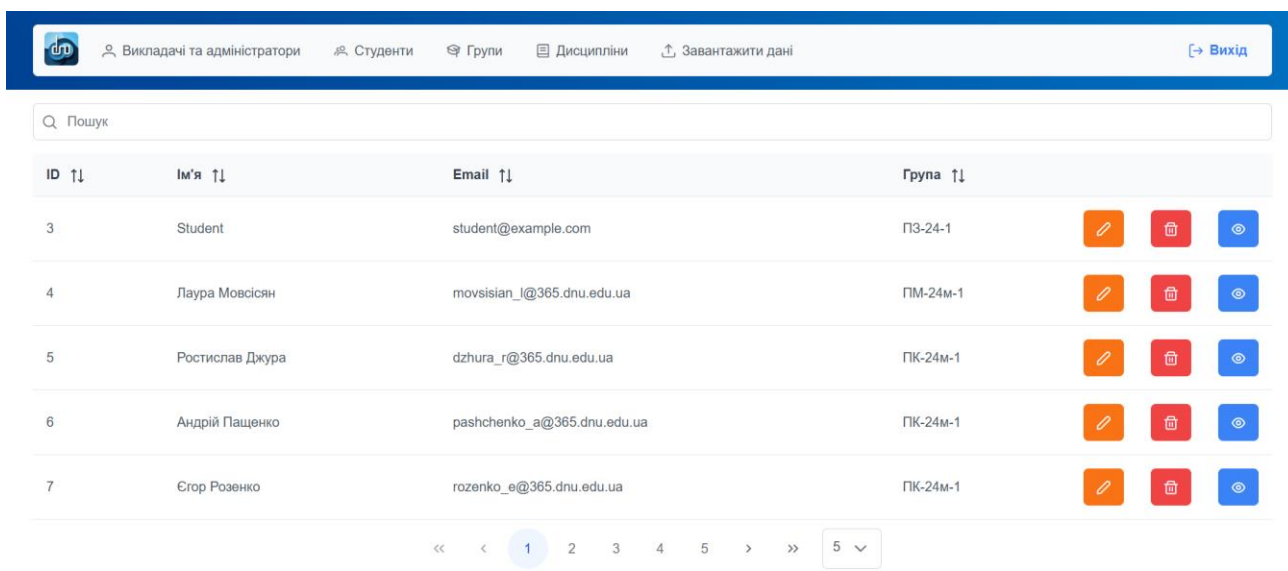


Рисунок 3.17 – Сторінка студентів

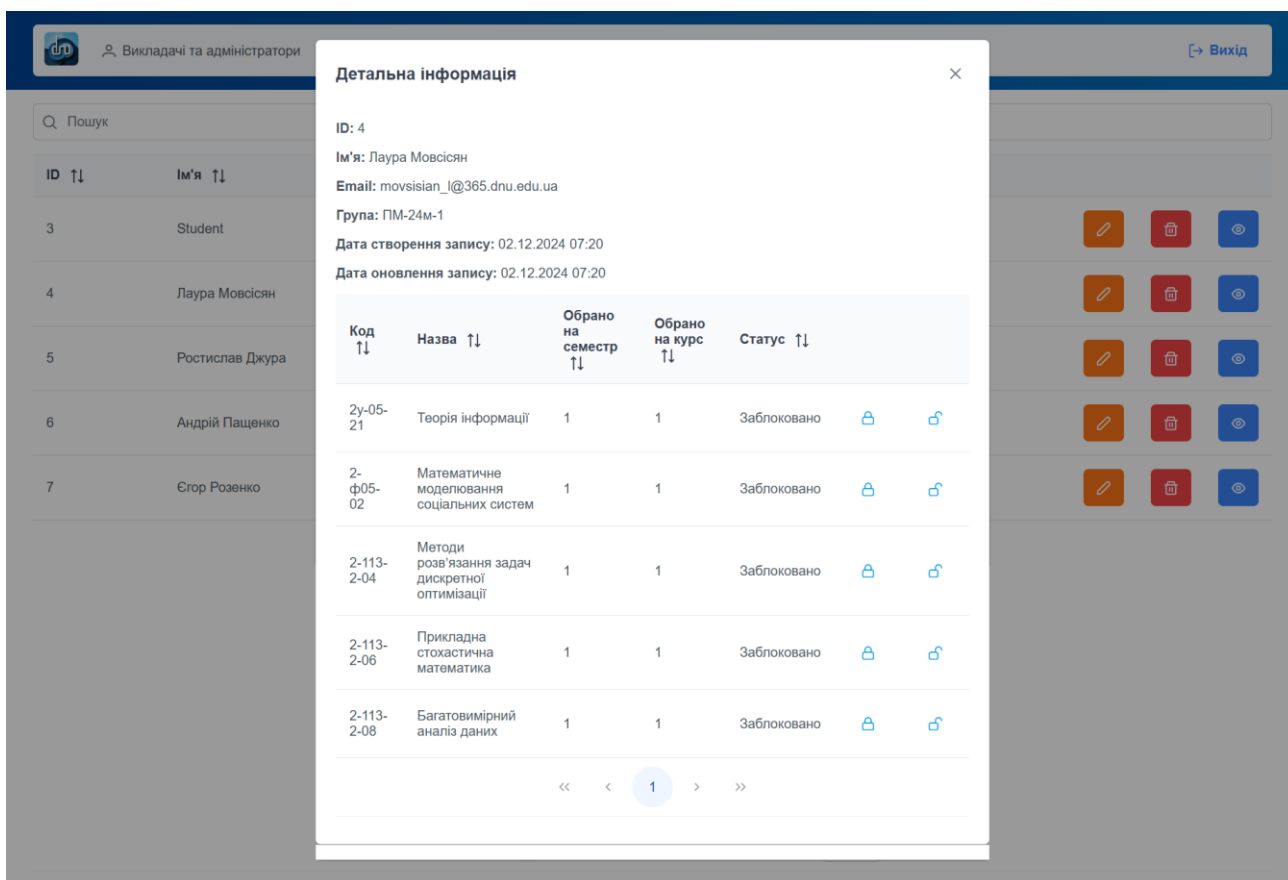


Рисунок 3.18 – Детальна інформація про студента

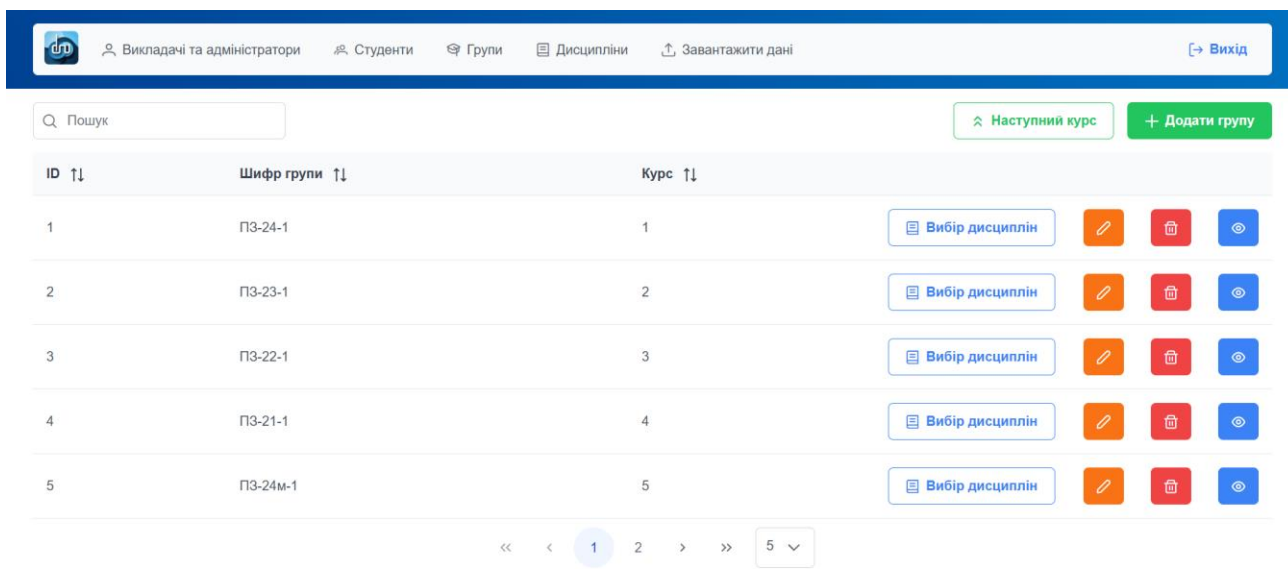


Рисунок 3.19 – Сторінка груп

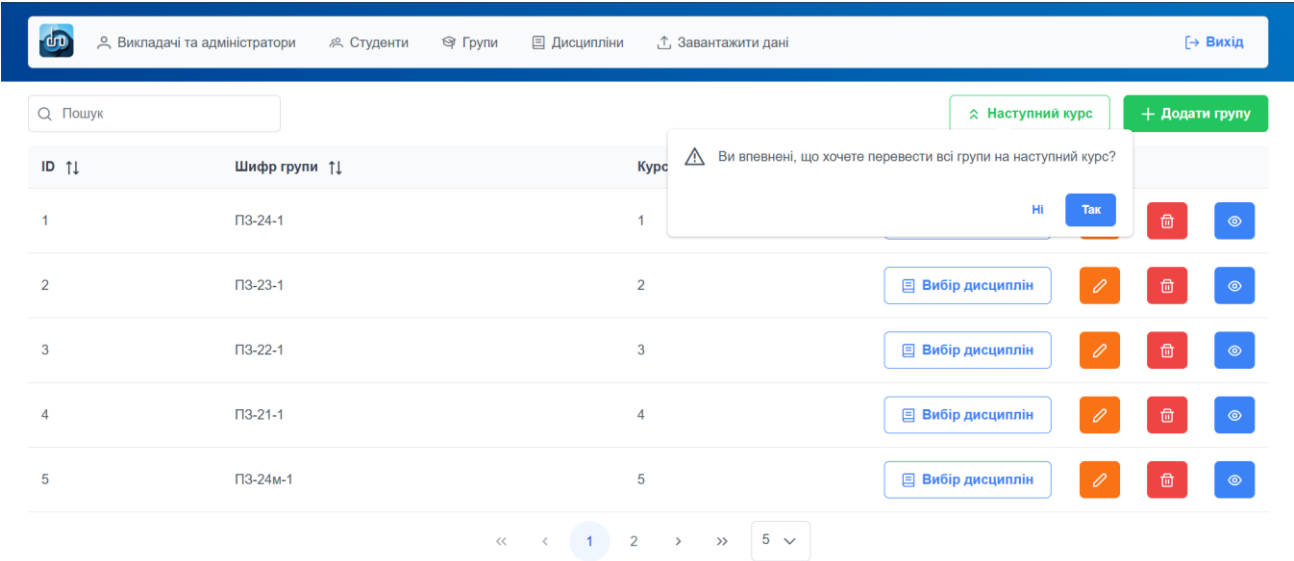


Рисунок 3.20 – Підтвердження переходу на наступний курс

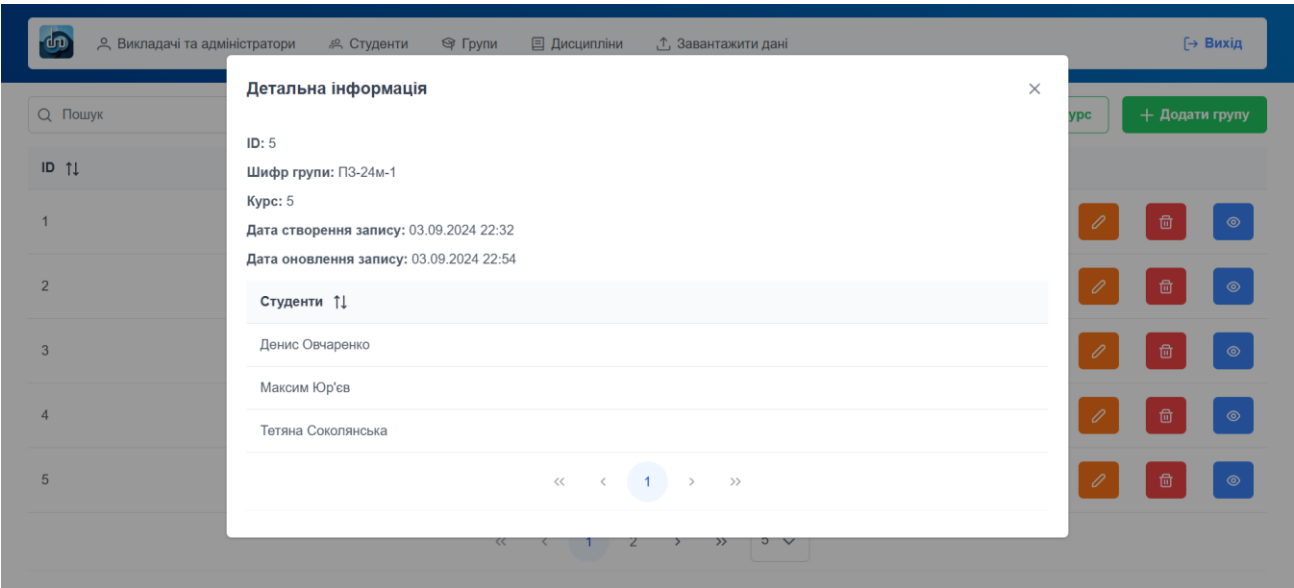


Рисунок 3.21 – Детальна інформація про групу

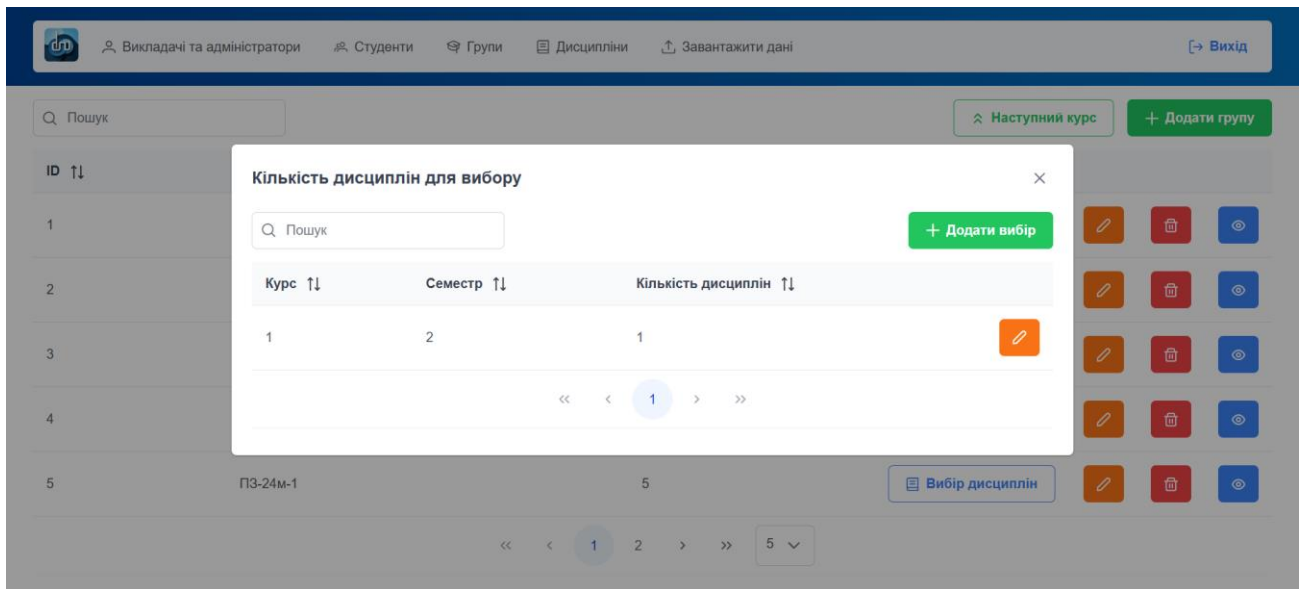


Рисунок 3.22 – Вікно кількості дисциплін для вибору

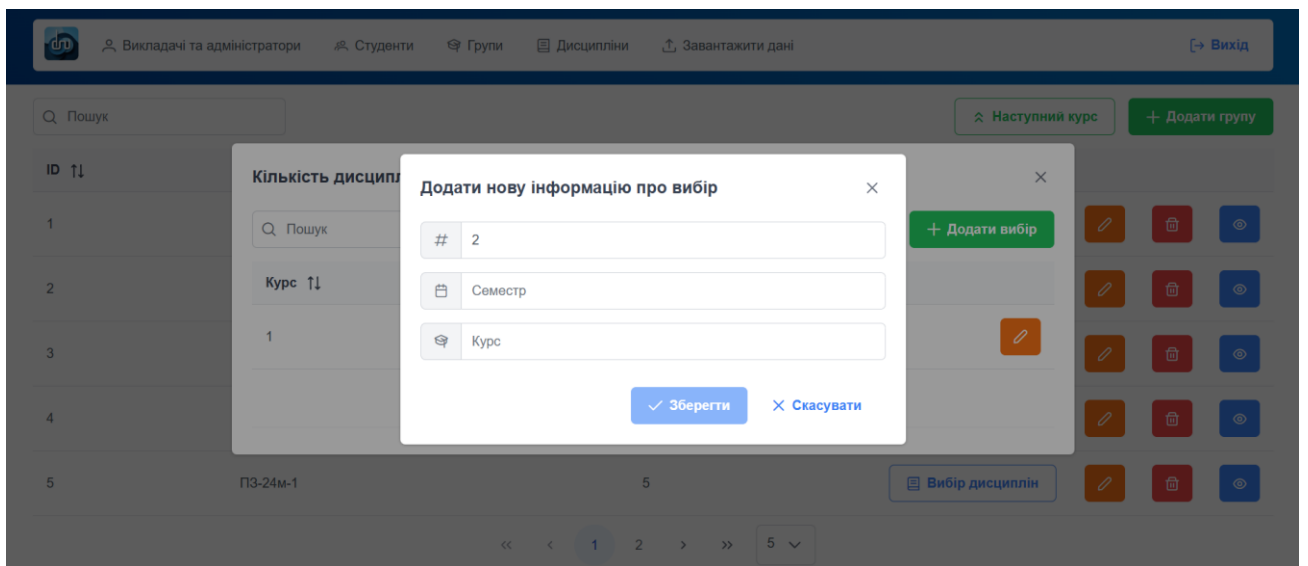


Рисунок 3.23 – Додавання нової інформації про вибір

















<div>  <span>Викладачі та адміністратори</span> <span>Студенти</span> <span>Групи</span> <span>Дисципліни</span> <span>Завантажити дані</span> <span>Вихід</span> </div>								
<div> <input type="text" value="Пошук"/> <span>+ Додати дисципліну</span> </div>								
ID ↑↓	Код ↑↓	Назва ↑↓	Семестр ↑↓	Курс ↑↓	Каталог ↑↓	Рівень ↑↓	Посилання ↑↓	
1	2y-05-21	Теорія інформації	2	1	Університетський	Бакалаврський	⋮	  
2	2-ф05-02	Математичне моделювання соціальних систем	2	1,2,3	Факультетський	Бакалаврський	⋮	  
3	2-113-2-04	Методи розв'язання задач дискретної оптимізації	1, 2	1,2,3,4	Університетський	Бакалаврський	⋮	  
4	2-113-2-06	Прикладна стохастична математика	-	-			⋮	  
5	2-113-2-08	Багатовимірний аналіз даних	-	-			⋮	  
<div> <span>&lt;&lt;</span> <span>&lt;</span> <span>1</span> <span>2</span> <span>3</span> <span>4</span> <span>5</span> <span>&gt;</span> <span>&gt;&gt;</span> <div>5 ▾</div> </div>								

Рисунок 3.24 – Сторінка дисциплін

















<div>  <span>Викладачі та адміністратори</span> <span>Студенти</span> <span>Групи</span> <span>Дисципліни</span> <span>Завантажити дані</span> <span>Вихід</span> </div>								
<div> <input type="text" value="Пошук"/> <span>+ Додати дисципліну</span> </div>								
ID ↑↓	Код ↑↓	Назва ↑↓	Семестр ↑↓	Курс ↑↓	Каталог ↑↓	Рівень ↑↓	Посилання ↑↓	
28	2-121-07	Алгоритми інтелектуального пошуку	-	-			⋮	  
22	2-124-02	Аналіз статистичної складності	-	-			⋮	  
5	2-113-2-08	Багатовимірний аналіз даних	-	-			⋮	  
14	2y-05-20	Вебаналітика	-	-			⋮	  
11	2y-05-19	Віртуальна реальність і 3D-моделювання	-	-			⋮	  
<div> <span>&lt;&lt;</span> <span>&lt;</span> <span>1</span> <span>2</span> <span>3</span> <span>4</span> <span>5</span> <span>&gt;</span> <span>&gt;&gt;</span> <div>5 ▾</div> </div>								

Рисунок 3.25 – Активне сортування



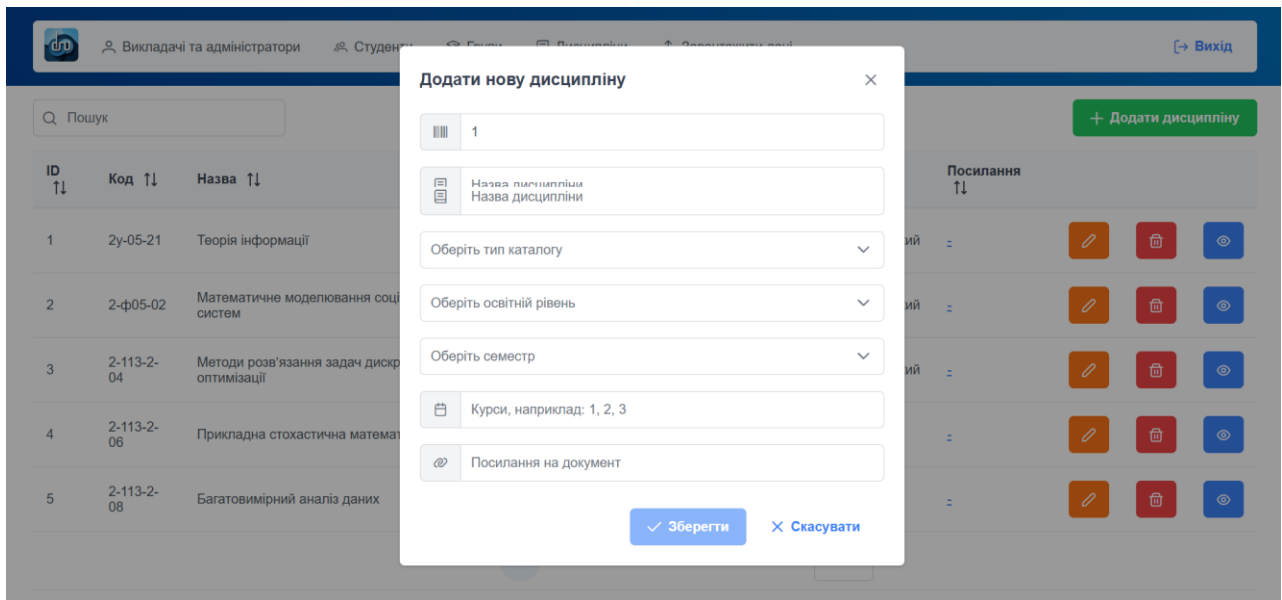


Рисунок 3.26 – Додавання нової дисципліни

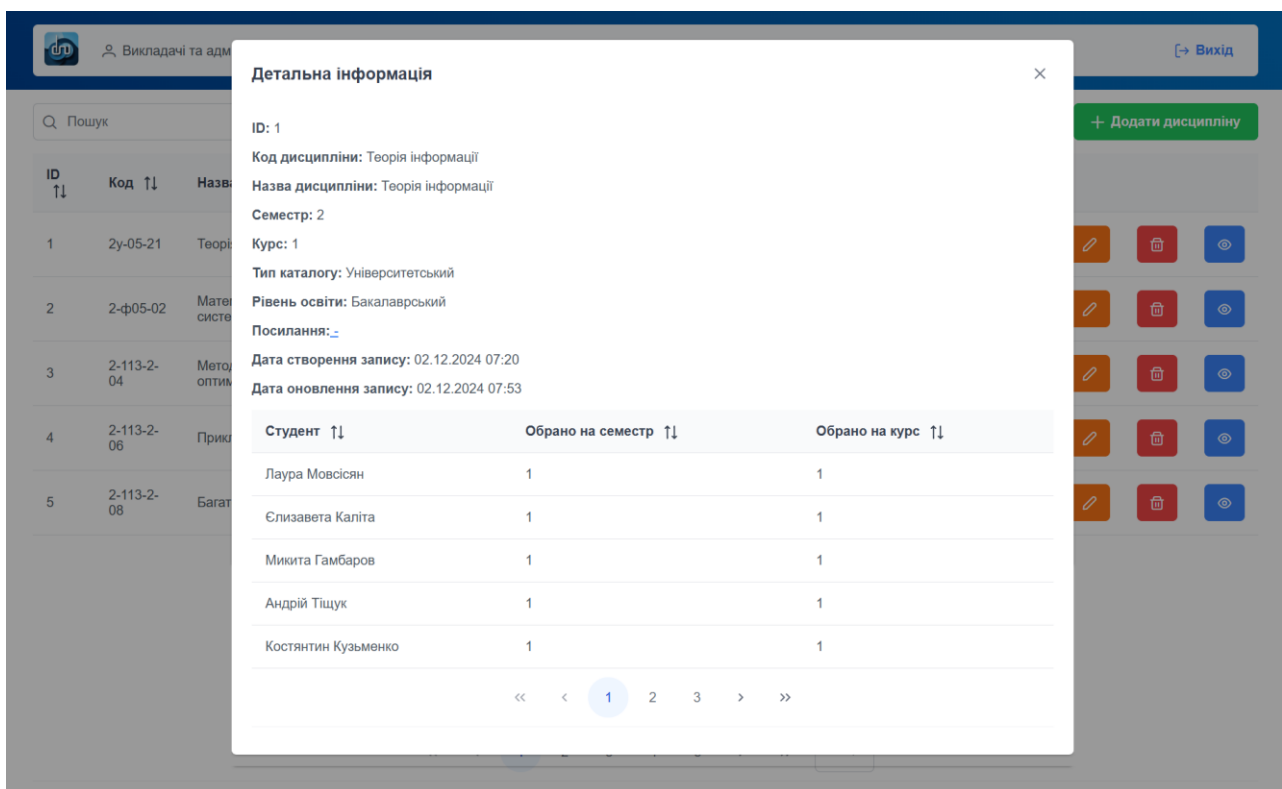


Рисунок 3.27 – Детальна інформація про дисципліну

Користувач з роллю «Викладач» має менше прав, ніж користувач з роллю «Адміністратор». Приклади роботи із застосунком для користувача з роллю «Викладач» показані на рисунках 3.28 – 3.29.

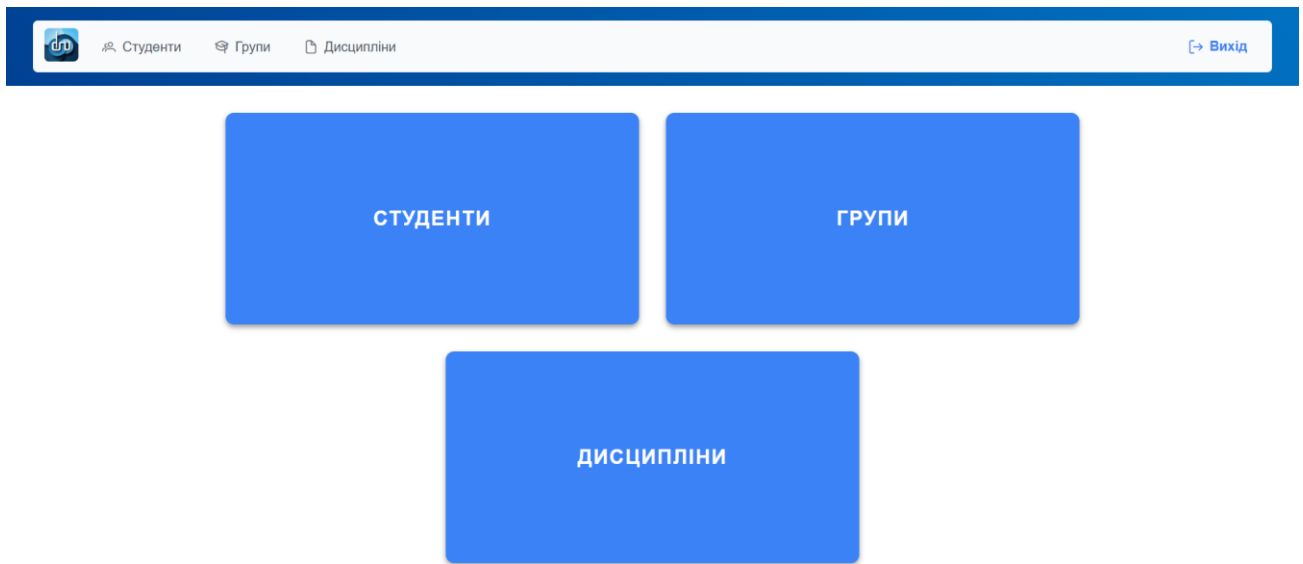


Рисунок 3.28 – Головна сторінка для викладача

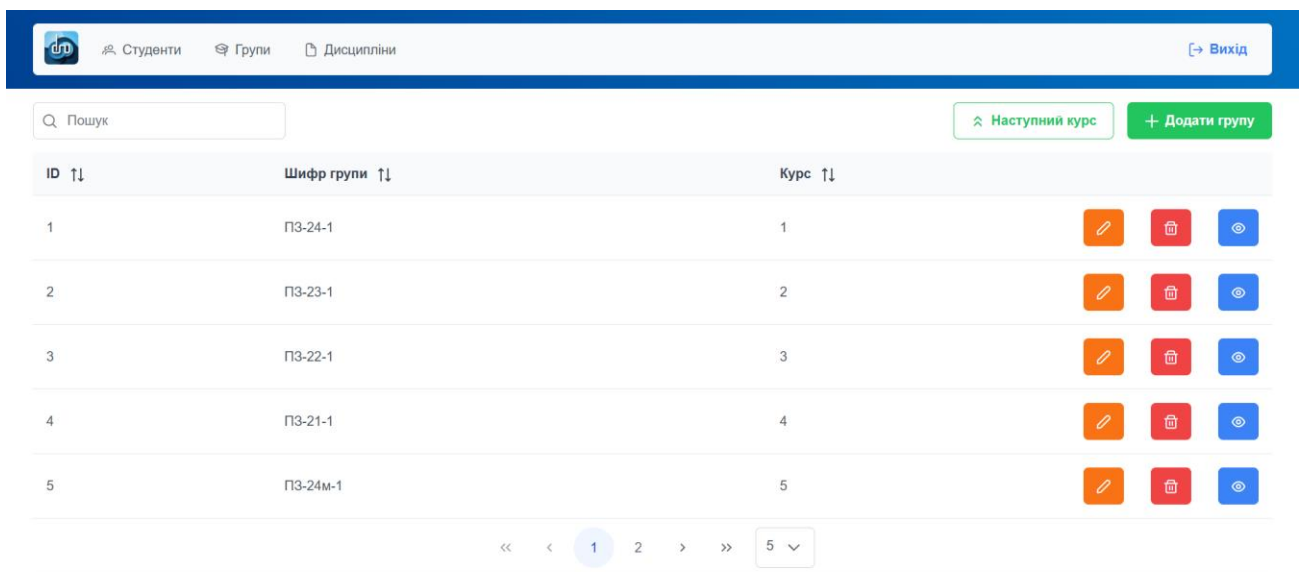


Рисунок 3.29 – Сторінка груп для викладача

Приклади роботи із застосунком для користувача з роллю «Студент» показані на рисунках 3.30 – 3.36.

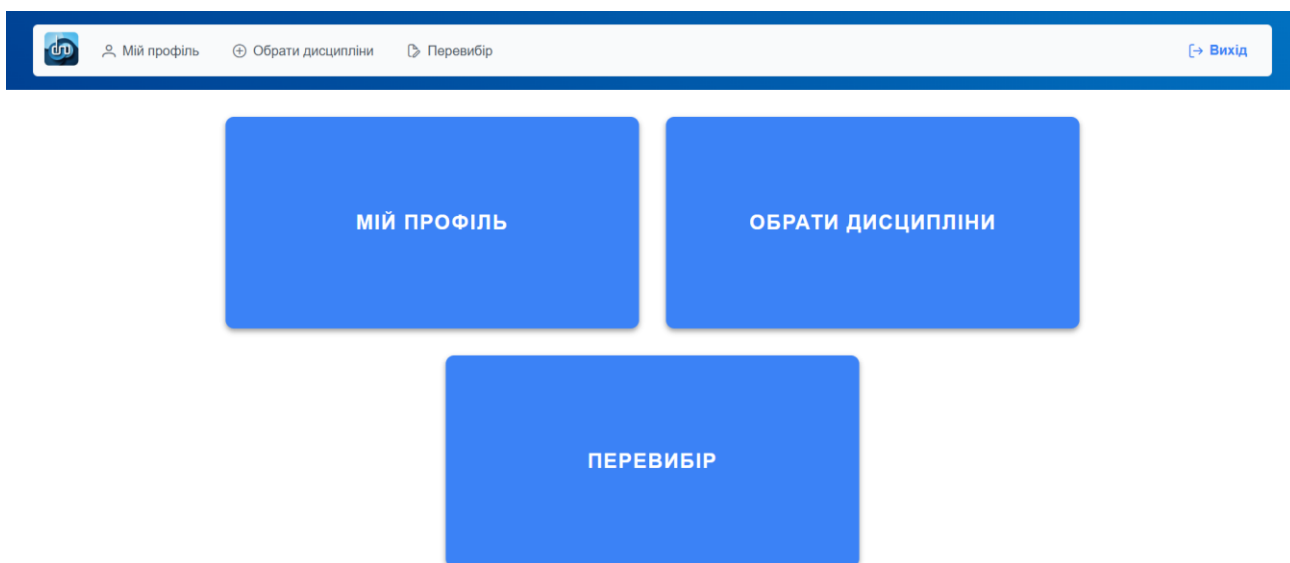


Рисунок 3.30 – Головна сторінка для студента

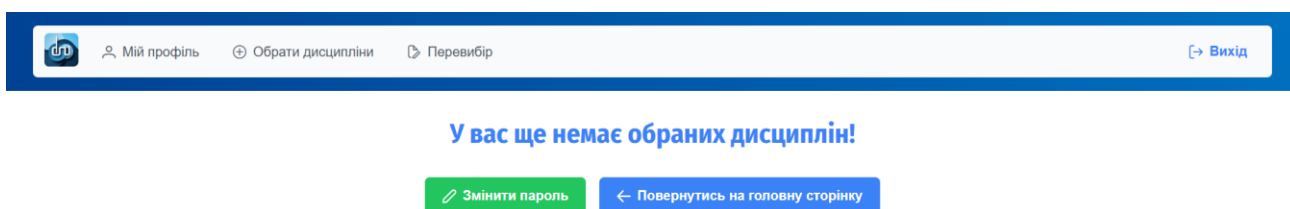


Рисунок 3.31 – Профіль студента без дисциплін

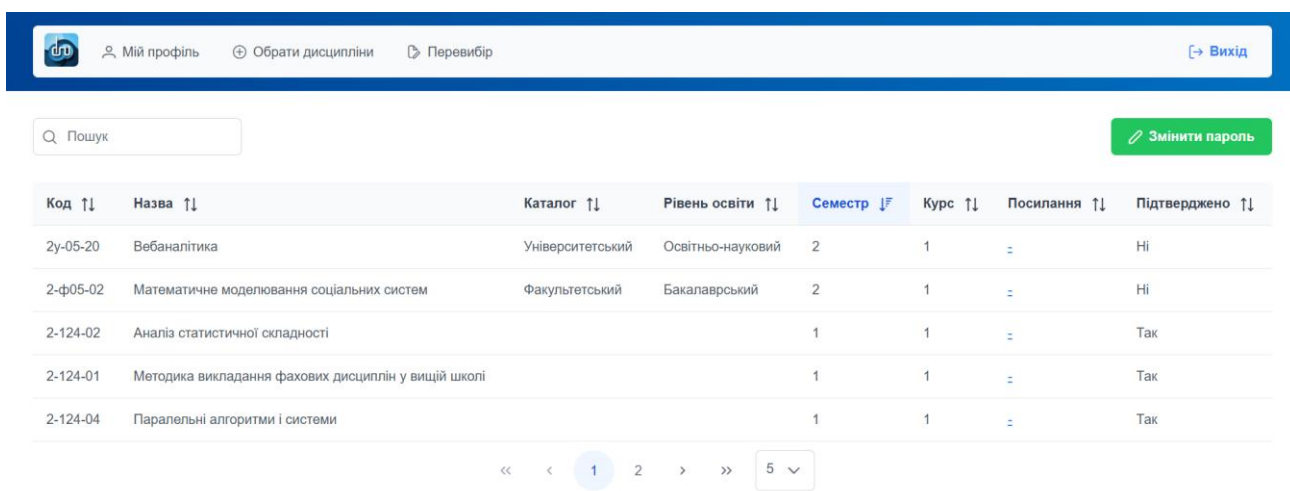


Рисунок 3.32 – Профіль студента з дисциплінами

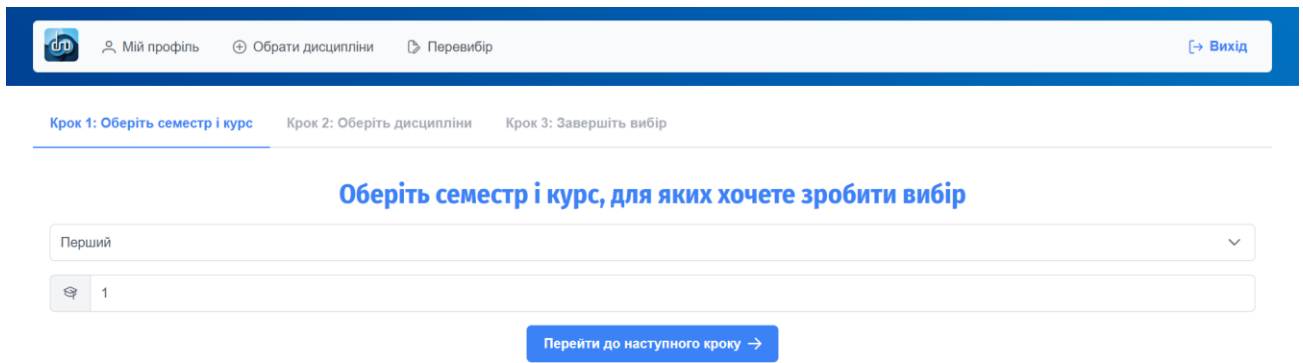


Рисунок 3.33 – Вибір дисциплін, крок 1

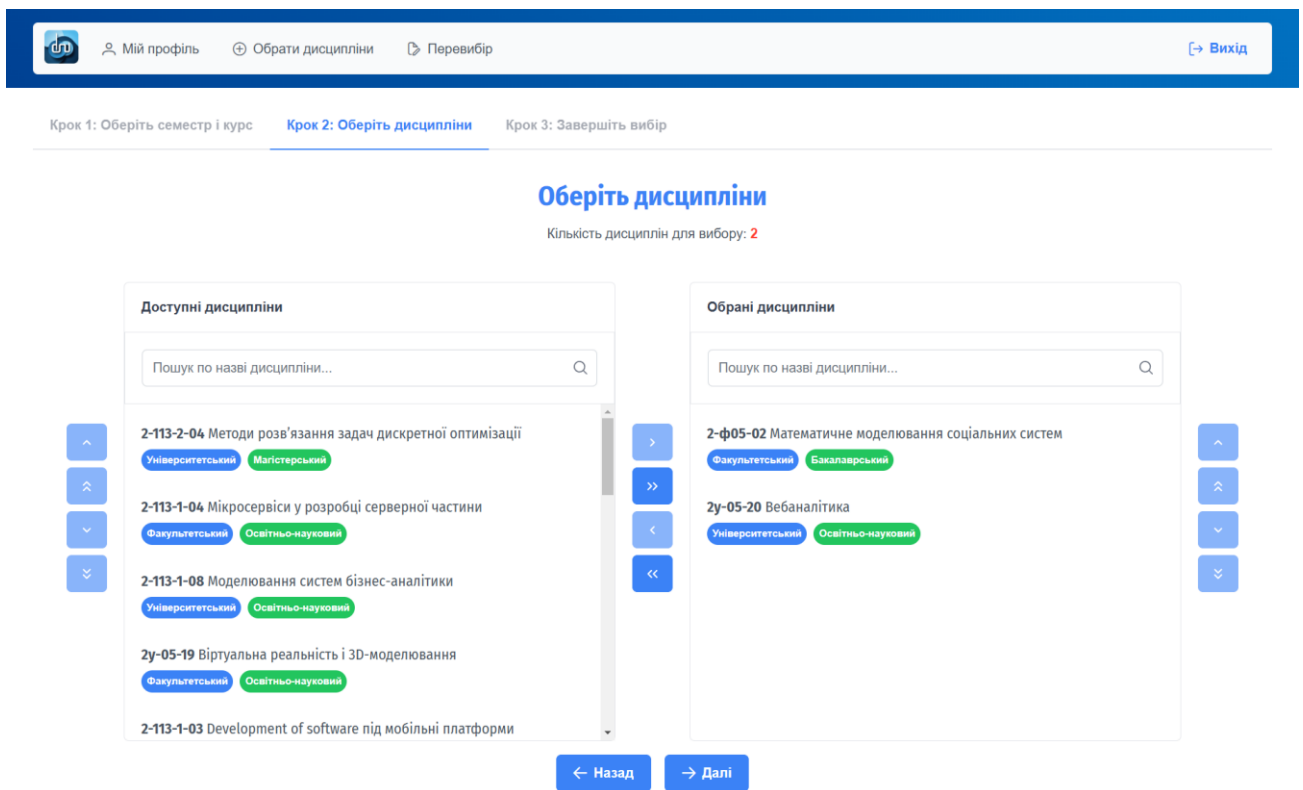


Рисунок 3.34 – Вибір дисциплін, крок 2

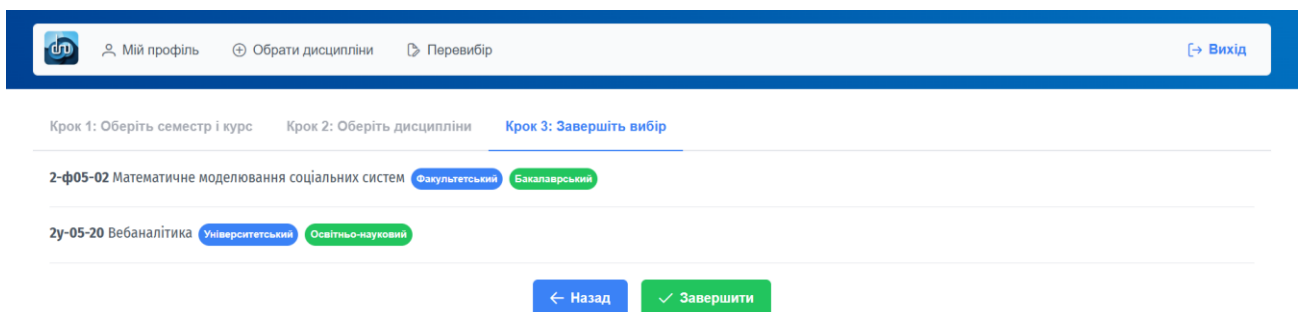


Рисунок 3.35 – Вибір дисциплін, крок 3

Мій профіль
 Обрати дисципліни
 Перевибір
 [-> Вихід]

### Дисципліни для перевибору:

Дані дисципліни будуть видалені зі списку ваших дисциплін після перевибору!

Код ↑↓	Назва ↑↓	Обрано на семестр ↑↓	Обрано на курс ↑↓	Каталог ↑↓	Рівень освіти ↑↓
2-ф05-02	Математичне моделювання соціальних систем	2	1	Факультетський	Бакалаврський
2у-05-20	Вебаналітика	2	1	Університетський	Освітньо-науковий

<< < 1 > >>

### Доступні дисципліни:

Кількість дисциплін, доступних для перевибору: 2

Пошук

Код ↑↓	Назва ↑↓	Каталог ↑↓	Рівень освіти ↑↓	Посилання ↑↓	
2-113-2-04	Методи розв'язання задач дискретної оптимізації	Університетський	Магістерський	⌵	
2-113-1-04	Мікросервіси у розробці серверної частини	Факультетський	Освітньо-науковий	⌵	
2-113-1-08	Моделювання систем бізнес-аналітики	Університетський	Освітньо-науковий	⌵	
2у-05-19	Віртуальна реальність і 3D-моделювання	Факультетський	Освітньо-науковий	⌵	
2-113-1-03	Development of software під мобільні платформи	Університетський	Освітньо-науковий	⌵	

<< < 1 2 3 4 > >> 5 ▾

### Переобрані дисципліни:

Ретельно перевірте дані!

Код ↑↓	Назва ↑↓	Каталог ↑↓	Рівень освіти ↑↓	Посилання ↑↓
2-113-1-08	Моделювання систем бізнес-аналітики	Університетський	Освітньо-науковий	⌵
2-113-1-03	Development of software під мобільні платформи	Університетський	Освітньо-науковий	⌵

<< < > >>

✓ Підтвердити вибір

Рисунок 3.36 – Перевибір дисциплін

## 3.2 Завантаження реальних даних про минулі вибори навчальних дисциплін здобувачів освіти


Завантаження реальних даних про минулі вибори навчальних дисциплін

здобувачів освіти доступне лише користувачам з роллю «Адміністратор».

Приклад роботи показаний на рисунках 3.37 – 3.38.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	email	name	group	code1	title1	code2	title2	code3	title3	code4	title4	code5	title5	
2	movsisian_l@365.dnu.edu.ua	Лаура Мовсісян	ПМ-24м-1	2у-05-21	Теорія інформації	2-ф05-02	Математичне моделювання соціальних систем	2-113-2-04	Методи розв'язання задач дискретної оптимізації	2-113-2-06	Прикладна математика	2-113-2-08	Багатовимірний аналіз даних	
3	dzhura_r@365.dnu.edu.ua	Ростислав Джура	ПК-24м-1	1у-05-043	Хмарні технології та сервіси	2-113-1-06	Методи побудови Recommendation Systems	2-113-1-01	Методи Computer modeling and simulation	2-113-1-04	Мікросервіси у розробці серверної частини	2-113-1-08	Моделювання систем бізнес-аналітики	
4	pashchenko_a@365.dnu.edu.ua	Андрій Пащенко	ПК-24м-1	2у-05-19	Віртуальна реальність і 3D-моделювання	2-113-1-01	Методи Computer modeling and simulation	2-113-1-04	Мікросервіси у розробці серверної частини	2-113-1-03	Development of software під мобільні платформи	2-ф05-06	Python для Data Scientist	
5	rozenko_e@365.dnu.edu.ua	Єгор Розенко	ПК-24м-1	2у-05-19	Віртуальна реальність і 3D-моделювання	2у-05-20	Вебаналітика	2-ф05-06	Python для Data Scientist	2-ф05-05	Digital technologies у закладах вищої освіти	2-113-1-06	Методи побудови Recommendation Systems	
6	voloshyn_d@365.dnu.edu.ua	Денис Волошин	ПМ-24м-1	1у-05-037	Технології веброботки та дизайну	1у-05-038	Основи програмування на мові JavaScript	1-ф05-07	Технології Java Backend	1-ф05-02	Сучасні середовища програмування	1-ф05-14	Unix-подібні операційні системи	
	kalita_ye@365.dnu.edu.ua	Єлизавета					Методика викладання фахових дисциплін у вищій		Аналіз статистичної		Паралельні		Системний аналіз, як світогляд	

Рисунок 3.37 – Приклад файлу


Викладачі та адміністратори
Студенти
Групи
Дисципліни
Завантажити дані
Вихід

Оберіть семестр і курс, для яких імпортуєте дані:

Другий

1

Завантажити

Пошта ↑↓	ПІБ ↑↓	Група ↑↓	Дисципліни
movsisian_l@365.dnu.edu.ua	Лаура Мовсісян	ПМ-24м-1	2у-05-21 Теорія інформації 2-ф05-02 Математичне моделювання соціальних систем 2-113-2-04 Методи розв'язання задач дискретної оптимізації 2-113-2-06 Прикладна стохастична математика 2-113-2-08 Багатовимірний аналіз даних
dzhura_r@365.dnu.edu.ua	Ростислав Джура	ПК-24м-1	1у-05-043 Хмарні технології та сервіси 2-113-1-06 Методи побудови Recommendation Systems 2-113-1-01 Методи Computer modeling and simulation 2-113-1-04 Мікросервіси у розробці серверної частини 2-113-1-08 Моделювання систем бізнес-аналітики
pashchenko_a@365.dnu.edu.ua	Андрій Пащенко	ПК-24м-1	2у-05-19 Віртуальна реальність і 3D-моделювання 2-113-1-01 Методи Computer modeling and simulation 2-113-1-04 Мікросервіси у розробці серверної частини 2-113-1-03 Development of software під мобільні платформи 2-ф05-06 Python для Data Scientist
rozenko_e@365.dnu.edu.ua	Єгор Розенко	ПК-24м-1	2у-05-19 Віртуальна реальність і 3D-моделювання 2у-05-20 Вебаналітика 2-ф05-06 Python для Data Scientist 2-ф05-05 Digital technologies у закладах вищої освіти 2-113-1-06 Методи побудови Recommendation Systems
voloshyn_d@365.dnu.edu.ua	Денис Волошин	ПМ-24м-1	1у-05-037 Технології веброботки та дизайну 1у-05-038 Основи програмування на мові JavaScript 1-ф05-07 Технології Java Backend 1-ф05-02 Сучасні середовища програмування 1-ф05-14 Unix-подібні операційні системи

<<
<
1
2
3
4
5
>
>>
5

Імпортувати дані

Рисунок 3.38 – Завантаження інформації

## ВИСНОВКИ

В ході роботи було виконано наступні задачі:

1. Розроблено базу даних, серверну та клієнтську частини додатку.
2. Створено ендпоінти для взаємодії з клієнтською частиною для забезпечення основних функцій застосунку, таких як:
  - реєстрація та авторизація;
  - облік і підтримка CRUD-операцій щодо усіх даних у системі;
  - оброблення вибору і перевибору навчальних дисциплін здобувачами освіти;
  - завантаження даних про минулі вибори дисциплін.
3. Розроблено систему безпеки для серверної частини додатку, а також інтегровано серверну частину з Passport.js та Redis.
4. Реалізовано зручні та інтерактивні компоненти з використанням реактивного підходу.
5. Реалізована підтримка інтерактивного пошуку, фільтрації та сортування даних.
6. Налаштовано відображення повідомлень про дії користувачів.
7. Реалізовано панель управління для керування усіма даними.
8. Реалізована динамічна взаємодія з сервером та оброблення даних в асинхронному режимі.
9. Реалізовано забезпечення сумісності форматів даних між клієнтом і сервером через JSON.
10. Реалізований респонсивний дизайн та адаптація вигляду і функціональності додатка до різних розмірів екранів і типів пристроїв.
11. Проведено практичну апробацію розробленого програмного застосунку і протестовано його на реальних даних.

## СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Moodle — Що таке Moodle. URL: <https://moodle.org/mod/page/view.php?id=8174> (дата звернення: 15.11.2024)
2. АС «Деканат». URL: <https://vuz.osvita.net/as-dekanat/> (дата звернення: 15.11.2024)
3. PowerSchool. URL: <https://www.powerschool.com/> (дата звернення: 15.11.2024)
4. TypeScript. URL: <https://www.typescriptlang.org/> (дата звернення: 16.11.2024)
5. Node.js. URL: <https://nodejs.org/en> (дата звернення: 16.11.2024)
6. NestJS. URL: <https://nestjs.com/> (дата звернення: 17.11.2024)
7. MySQL. URL: <https://www.mysql.com/> (дата звернення: 17.11.2024)
8. Sequelize. URL: <https://sequelize.org/> (дата звернення: 18.11.2024)
9. Redis. URL: <https://redis.io/> (дата звернення: 18.11.2024)
10. Angular. URL: <https://angular.dev/> (дата звернення: 18.11.2024)
11. PrimeNG. URL: <https://primeng.org/> (дата звернення: 19.11.2024)
12. RxJS. URL: <https://rxjs.dev/> (дата звернення: 19.11.2024)