

Приватний заклад вищої освіти
Одеський технологічний університет «ШАГ»
Кафедра інформаційних технологій та фундаментальної підготовки

Випускна кваліфікаційна робота бакалавра

**Інформаційна система гнучкого управління програмними проектами
(на прикладі Trello)**

на здобуття бакалаврського рівня вищої освіти
зі спеціальності «122 Комп'ютерні науки»

Виконавці проекту

	П.І.Б.	Група
1	Миронов О. М.	КН-П-211
2	Костанда Д. О.	КН-П-211
3	Жданов А. Д.	КН-П-211
4	Бордюжевич О. І.	КН-П-211
5	Голубов В. А.	КН-Д-211

Автор звіту

_____ Костанда Даниїл Олексійович

Науковий керівник	Доцент кафедри ІТ та ФП, к.т.н., доцент Данилець Є. В.
-------------------	--

АНОТАЦІЯ

УДК 004.415.2+004.415.3+004.43:005.8

К-72

Костанда Д. О.

Інформаційна система організації управління програмними проектами та завданнями: випускна кваліфікаційна робота на здобуття бакалаврського рівня вищої освіти зі спеціальності «122 Комп'ютерні науки». – Одеса: ОТУШ, 2025. – 24 с.

Робота присвячена проектуванню, розробці та випробуванню серверної частини веборієнтованої інформаційної системи управління програмними проектами та завданнями *ItRun*. Основне призначення системи — забезпечення командної взаємодії над цифровими проектами у форматі канбан-дошок із підтримкою користувачів, ролей, робочих просторів, завдань, коментарів та вкладень.

Актуальність теми зумовлена потребою малого та середнього бізнесу в адаптивних, безкоштовних та функціонально гнучких системах для управління завданнями. Існуючі рішення часто є надмірно складними або обмеженими у можливостях. Реалізована серверна частина враховує вимоги до масштабованості, захисту даних та інтеграційної сумісності з фронтендом через REST API.

У рамках дослідження обґрунтовано вибір архітектури та інструментів: NestJS (Node.js + TypeScript) як основний бекенд-фреймворк, TypeORM для взаємодії з MySQL, Multer для обробки файлів, JWT для авторизації, Swagger для документації. Забезпечено модульну архітектуру відповідно до принципів Clean Architecture, зокрема виділено окремі модулі для аутентифікації, користувачів, проектів, дошок, задач, вкладень, коментарів та міток.

Основними завданнями були:

- реалізація CRUD-функціоналу для всіх сутностей системи;
- впровадження безпечної автентифікації та розмежування прав доступу;
- побудова документації API засобами Swagger;
- забезпечення масштабованості та підтримки CI/CD через Docker і GitHub Actions;
- тестування працездатності та стабільності через REST-запити.

Результатом роботи є функціонально завершена та протестована серверна частина, що відповідає сучасним вимогам до вебсервісів і готова до інтеграції з клієнтською частиною. Робота базується на 8 використаних джерелах, у тому числі технічних довідниках, наукових публікаціях та офіційній документації фреймворків.

ЗМІСТ

ВСТУП _____	4
Технічне завдання до проєкту _____	5
РОЗДІЛ 1. Загальна характеристика системи _____	8
РОЗДІЛ 2. Реалізація серверної частини _____	10
2.1 Технології та інструменти розробки серверної частини _____	10
2.2 Архітектура серверної частини та API _____	13
РОЗДІЛ 3. Результати випробувань _____	18
ВИСНОВКИ _____	24
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ _____	25

ВСТУП

Стрімкий розвиток інформаційних технологій суттєво трансформував способи організації праці, взаємодії команд та управління цифровими проєктами. В умовах глобалізації, розподілених команд та зростання обсягу задач виникає потреба в інструментах, що дозволяють гнучко, наочно й ефективно керувати робочими процесами, розподіляти відповідальність та контролювати строки виконання. Саме тому системи управління завданнями та проєктами (так звані таск-трекери) стали невід’ємним компонентом сучасного цифрового середовища.

Незважаючи на наявність великої кількості аналогічних систем — таких як Trello, Jira, Asana — більшість із них мають обмежену гнучкість налаштувань, складний інтерфейс або потребують підключення сторонніх інтеграцій навіть для базових функцій. Крім того, значна частина таких сервісів є платними або обмеженими у функціональності для невеликих команд. Ці недоліки створюють передумови для розробки власного рішення, яке буде адаптовано до специфічних вимог користувачів, матиме відкриту архітектуру, забезпечуватиме контроль над даними та можливість подальшого розширення.

У зв’язку з цим команда розробників реалізувала інформаційну систему управління програмними проєктами та завданнями — ItRun. Основною метою проєкту є створення зручного, адаптивного та масштабованого вебзастосунку для управління завданнями у форматі канбан-дошки, з підтримкою користувачів, робочих просторів, колонок, карток і спільної роботи над проєктами. Платформа реалізована за сучасними технологічними підходами з використанням модульної архітектури, container-based deployment та засобів безперервної інтеграції.

У межах цієї дипломної роботи зосереджено увагу на проєктуванні та реалізації серверної частини інформаційної системи управління проєктами ItRun. Реалізація здійснювалась із використанням фреймворку NestJS, що ґрунтується на принципах модульної архітектури та інверсії залежностей. В роботі детально розглянуто побудову структури серверного застосунку, визначено архітектурні принципи організації модулів, контролерів, сервісів, DTO та сутностей. Для взаємодії з реляційною базою даних MySQL застосовано ORM-бібліотеку TypeORM, яка забезпечує типобезпечну роботу зі складними зв’язками між сутностями та підтримку транзакцій. Увагу приділено також валідації запитів за допомогою `class-validator`, реалізації CRUD-функціональності для основних об’єктів системи та побудові REST API для зв’язку з клієнтською частиною. Усі рішення узгоджуються з вимогами до масштабованості, безпеки й підтримуваності сучасних серверних систем.

ТЕХНІЧНЕ ЗАВДАННЯ ДО ПРОЄКТУ

Загальний опис

Програмне забезпечення «ItRun» призначене для організації ефективного управління завданнями та цифровими проєктами в команді. Система створюється у форматі вебзастосунку на базі клієнт-серверної архітектури з можливістю масштабування, контейнеризації та автоматизованого розгортання. Основною ідеєю є створення інструмента для візуального управління проєктами за допомогою канбан-дошок, що дає змогу формувати робочі простори, створювати задачі, призначати відповідальних, відслідковувати статуси виконання, коментувати й координувати дії команди.

Застосунок забезпечує повноцінну взаємодію між учасниками одного або декількох робочих просторів, підтримує контроль версій, зберігає історію дій користувачів, а також включає механізми захисту персональних даних та автентифікації через підтвердження email. Платформа передбачає розподіл ролей та прав доступу, що забезпечує гнучке адміністрування проєктів.

Призначення

Система орієнтована на використання невеликими та середніми командами розробників, стартапами, фриланс-агенціями, а також окремими користувачами, які потребують інструменту для організації власної діяльності. Особлива увага приділена простоті використання, зручному UI/UX, можливості гнучкої інтеграції, та легкому оновленню функціоналу в майбутньому без втрати даних. Застосунок має підтримувати повний цикл взаємодії: створення задач, їх планування, виконання, коментування та завершення.

Функціональні вимоги

1. Реєстрація та автентифікація

Кожен користувач повинен мати можливість створити обліковий запис у системі. Після реєстрації на електронну адресу надсилається підтвердження, яке користувач має ввести для завершення реєстрації. Після цього користувач може увійти до системи, вказавши свої облікові дані. Усі дії, які потребують доступу до персональної інформації або даних проєктів, мають бути доступні лише авторизованим користувачам. Також має бути можливість вийти із системи, в тому числі з усіх пристроїв одночасно. Деякі функції доступні тільки певним категоріям користувачів (наприклад, адміністраторам).

2. Робочі простори та дошки

Користувачі можуть створювати окремі робочі простори — умовні «команди» або «проєкти», всередині яких розміщуються канбан-дошки. У таких дошках відображається весь хід роботи над завданнями. Користувачі мають можливість

змінювати зовнішній вигляд дошки (наприклад, завантажувати зображення чи логотипи), а також запрошувати інших людей до участі у роботі над проектом. При цьому можна визначати роль кожного учасника — наприклад, дозволити редагування лише окремим особам, а іншим лише переглядати вміст.

3. Колонки та задачі

Всередині кожної дошки можна створювати колонки (етапи роботи), в які додаються окремі завдання. Ці завдання можна редагувати, переміщати між колонками, видаляти або змінювати їх порядок. Кожне завдання може мати дату завершення, пріоритетність, позначки або додані файли. Також має бути візуальне відображення того, на якому етапі виконання перебуває завдання, щоб усі учасники команди могли швидко оцінити прогрес роботи.

Технічні вимоги та рекомендації до реалізації

Клієнтська-частина реалізується у вигляді односторінкового застосунку (SPA) із застосуванням сучасного стеку технологій:

- Реалізація інтерфейсних компонентів з використанням бібліотеки React у поєднанні з мовою TypeScript
- Клієнтська навігація — забезпечується за допомогою бібліотеки React Router
- Інтерактивна функціональність drag-and-drop — реалізована з використанням відповідних інструментів, таких як react-beautiful-dnd
- Інтеграція з серверною частиною через REST API — включає обробку помилок, відображення індикаторів завантаження (loader'ів), валідацію форм введення
- UI/UX-дизайн — відповідає принципам адаптивності, інтуїтивності та доступності для користувача

Серверна-частина спроектована на основі фреймворку NestJS та побудована відповідно до концепції доменно-орієнтованого дизайну (DDD):

- Модульна структура застосунку — з поділом на auth, user, workspace, board, task, comment, history
- Робота з базою даних MySQL — реалізована через ORM-бібліотеку TypeORM
- Побудова REST API — із валідацією запитів за допомогою Data Transfer Object (DTO) та централізованою обробкою винятків
- Контроль доступу до ресурсів — реалізовано згідно з рольовою моделлю користувача (Admin / Member)
- Дотримання архітектурних патернів — що забезпечує масштабованість і підтримуваність системи

DevOps та інфраструктурний рівень охоплює засоби автоматизації розгортання та підтримки життєвого циклу застосунку:

- Контейнеризація проєкту — з використанням Docker для створення окремих контейнерів: frontend, backend та бази даних MySQL
- Опис взаємодії сервісів — через docker-compose конфігурацію
- Розгортання проєкту на VPS — із встановленим CentOS та налаштуванням зворотного проксингу через веб-сервер Nginx
- Збереження конфігурацій — у вигляді .env-файлів із секретними змінними оточення
- Реалізація CI/CD — із застосуванням GitHub Actions для автоматичного тестування, білду та розгортання системи

Менеджмент та супровід розробки включає організацію роботи команди, планування та документування проєкту:

- Постановка задач, формування user stories та створення дорожньої карти (roadmap) — реалізовано в середовищі Notion
- Використання Kanban-дошки — для відстеження статусів задач у Trello, що забезпечує прозору візуалізацію прогресу
- Проведення регулярних синків команди — з фіксацією результатів у звітах
- Централізована технічна документація — включає опис функціональності, архітектури, бізнес-логіки, API та сценаріїв використання
- Забезпечення відповідності реалізації вимогам технічного завдання — з обов'язковим контролем якості на всіх етапах розробки.

UI/UX-дизайн інтерфейсного рішення платформи ItRun базувався на сучасних підходах до цифрового продуктового дизайну з урахуванням потреб цільової аудиторії та принципів зручності використання:

- UX-дослідження — аналіз Trello, емпатійна карта, Customer Journey Map
- Структура взаємодії — побудова Site Map та User Flow
- Дизайн-система — типографіка (Poppins), кольори (Tiffany, помаранчевий), сітка 16 пікселів
- UI-компоненти — реалізація у станах (default, hover, active, disabled), auto-layout
- Прототипування — високофідельні екрани у Figma: реєстрація, онбординг, workspace, дошка
- UX-тестування — виявлення та виправлення проблем за результатами користувацьких сценаріїв
- Айдентика — створення логотипу, що поєднує IT-естетику та динаміку керування завданнями

Усі частини є незалежними, розробляються окремими учасниками команди, мають інтерфейси взаємодії та підлягають документуванню.

РОЗДІЛ 1. ЗАГАЛЬНА ХАРАКТЕРИСТИКА СИСТЕМИ

Відповідно до технічного завдання (ТЗ), командою розробників було спроектовано та реалізовано інформаційну систему управління програмними проектами за архітектурою клієнт-сервер. Система побудована за принципом SPA (Single Page Application), що забезпечує швидкий відгук користувача, мінімізацію затримок завантаження та зручність інтеграції з API. У процесі розробки було структуровано наступні функціональні підсистеми:

- «Design»
- «Frontend»
- «Backend»
- «DevOps»
- «Project Management»

Частина **Design** містить комплекс макетів та прототипів інтерфейсів користувача, які були розроблені із дотриманням сучасних принципів UI/UX-дизайну. У ході проєктування інтерфейсів було визначено кольорову палітру, типографіку, компонентну бібліотеку (UI-kit), а також реалізовано інтерактивні анімації для забезпечення комфортної взаємодії користувача з додатком. Усі графічні макети було створено в середовищі Figma відповідно до узгодженого функціоналу кожного модуля.

Підсистема **Frontend** реалізована за допомогою бібліотеки React із використанням мов програмування TypeScript та JavaScript, а також HTML/CSS для розмітки та стилізації інтерфейсу. Комунікація з сервером здійснюється через HTTP-запити за допомогою вбудованого API клієнта. Особливу увагу було приділено адаптивності та продуктивності інтерфейсу, зокрема реалізації компонентного підходу та валідації форм введення.

Серверна логіка, що складає частину **Backend**, була реалізована на платформі Node.js із використанням фреймворку NestJS. Система підтримує REST API, що забезпечує модульну структуру запитів та відповідей у форматі JSON. Для зберігання даних застосовано реляційну СУБД MySQL, взаємодія з якою відбувається через ORM-бібліотеку TypeORM. У якості механізму автентифікації впроваджено JWT-токени, що дозволяє реалізувати контроль доступу до обмежених ресурсів системи.

Адміністративна частина **DevOps** охоплює процеси автоматизації розгортання та адміністрування застосунку. Для забезпечення ізоляції компонентів було використано Docker-контейнеризацію. Розгортання інфраструктури здійснюється на базі VPS-сервера з використанням Docker Compose, що дозволяє конфігурувати та запускати пов'язані сервіси одночасно. Автоматизацію CI/CD-процесів реалізовано через GitHub Actions, що забезпечує безперервну інтеграцію та доставку оновлень до серверного середовища без необхідності ручного втручання.

У межах підсистеми **Project Management** особливу увагу було приділено управлінню знаннями проекту та організації робочого процесу за методологією Kanban. Документація всіх етапів — від постановки задач до технічних рішень — здійснювалась у Notion, що забезпечило централізований доступ до специфікацій API, архітектури, записів зустрічей і ретроспектив. Такий підхід дозволив системно керувати беклогом, user stories та приймальними критеріями, забезпечуючи прозорість і гнучкість у процесі розробки.

Таким чином, система реалізована відповідно до сучасних стандартів розробки веб-застосунків і забезпечує високу гнучкість, масштабованість та підтримку безперервної інтеграції, що відповідає запитам сучасних програмних продуктів у сфері управління проєктами.

РОЗДІЛ 2 РЕАЛІЗАЦІЯ СЕРВЕРНОЇ ЧАСТИНИ

2.1 Технології та інструменти розробки серверної частини

Серверну частину інформаційної системи управління проектами реалізовано з використанням прогресивного фреймворку NestJS — сучасного інструменту для створення масштабованих та структурованих серверних додатків на платформі Node.js. Архітектура побудована згідно з принципами модульності, що забезпечує високий рівень інкапсуляції, розділення відповідальностей та масштабованість застосунку.

У якості основної мови програмування обрано TypeScript — мову зі статичною типізацією, яка компілюється у JavaScript. Вона дозволяє виявляти помилки ще на етапі компіляції, що підвищує надійність коду. Завдяки системі типів, автодоповненню, навігації по коду та контролю відповідностей між модулями, TypeScript значно спрощує підтримку великих проєктів, пришвидшує розробку й сприяє кращій структуризації архітектури застосунку.

NestJS у поєднанні з TypeScript дозволив реалізувати чітку модульну структуру, де кожна доменна сутність винесена в окремий модуль із власними контролерами, сервісами, DTO-об'єктами та ентиті, що відповідає сучасним вимогам до надійних серверних систем. [1, с. 101]

Для роботи з базою даних у проєкті використано реляційну систему керування базами даних MySQL, яка є однією з найпопулярніших СКБД завдяки своїй швидкості, стабільності, багатоплатформеності та гнучкості. MySQL забезпечує потужний функціонал для роботи з даними: підтримує фільтрацію, сортування, пошук, транзакції та керування зв'язками між сутностями. Її здатність ефективно працювати з великими обсягами даних (до мільярда рядків у численних таблицях), обслуговувати сотні одночасних клієнтів і підтримувати реплікацію робить її придатною для масштабованих вебзастосунків і багатокористувацьких систем. Крім того, MySQL добре задокументована, має розвинене ком'юніті та підтримку, проста в адмініструванні та інтегрується з великою кількістю серверних технологій. У проєкті вона виступає як надійне ядро для зберігання інформації про користувачів, робочі простори, задачі, вкладення, коментарі тощо.

Для роботи з MySQL на рівні серверного коду було використано TypeORM — об'єктно-реляційну бібліотеку, яка забезпечує зручне зв'язування об'єктів додатку з таблицями бази даних. Вона дозволяє описувати структуру сутностей за допомогою декораторів, будувати складні зв'язки між ними (OneToMany, ManyToOne, ManyToMany), реалізовувати каскадні операції та працювати з lazy/eager-завантаженням. Це спрощує розробку, підвищує читабельність коду та уніфікує доступ до даних у межах мікросервісної архітектури. [2, с. 66]

Аутентифікацію реалізовано за допомогою JWT (JSON Web Token), що забезпечує безпечний та безстейтовий доступ до приватних маршрутів. [3]

У наукових матеріалах існує усталене визначення поняття JW токена, що повноцінно розкриває його сутність та складові частини:

JSON Web Token (JWT) — це стандарт обміну даними у форматі JSON, який використовується для створення токенів доступу. Структура JWT включає три частини, розділені крапками: заголовок (header), корисне навантаження (payload) та криптографічний підпис (signature). У заголовку зазначається тип токена (JWT) та алгоритм, за яким здійснюється підписування даних. [2, с. 208]

Приклад JSON Web Token вказано на рисунку 2.1:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpFubmEgU21pdGgiLCJpYXQiOiE1MTYyMzkwMjJ9.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Рисунок 2.1 – Структура JWT токена

Для обробки файлів, таких як аватари користувачів, обкладинки до дошок чи вкладення до карток, у проєкті використовується бібліотека **Multer** — спеціалізоване middleware для Node.js, призначене для обробки HTTP-запитів із типом **multipart/form-data**, які зазвичай містять файли. Цей інструмент побудований на основі **busboy**, що забезпечує високу продуктивність під час роботи з вхідними потоками даних. Завдяки Multer реалізовано збереження файлів у локальній файловій системі, з подальшим збереженням шляху до файлу в базі даних. Також здійснюється попередня перевірка MIME-типів та обмеження на розмір файлів, що завантажуються, відповідно до вимог безпеки та цілісності системи. Такий підхід забезпечує надійність при роботі з вкладеннями та дозволяє зручно керувати файловим контентом на боці сервера [4].

Валідація вхідних даних здійснюється за допомогою бібліотек **class-validator** та **class-transformer**, які дозволяють автоматизовано перевіряти відповідність переданих даних заданим правилам (наприклад, формат email, мінімальна довжина пароля тощо) та трансформувати їх у відповідні класи. Це забезпечує високий рівень безпеки, зменшуючи ризик невалідних або шкідливих запитів до сервера.

Для документування API використовується інтеграція **Swagger**, яка дозволяє автоматично генерувати інтерактивну документацію REST API. Розробники можуть у режимі реального часу переглядати доступні маршрути, параметри запитів, приклади відповідей та статус-коди, що значно полегшує тестування, інтеграцію та командну взаємодію. [5]

Використання цих технологій дозволило побудувати **структуровану, надійну та гнучку серверну частину**, яка відповідає сучасним вимогам до продуктивності та безпеки. Вона здатна ефективно обслуговувати велику кількість паралельних запитів, обробляти складну бізнес-логіку та масштабуватись відповідно до потреб системи.

2.2 Архітектура серверної частини та API

Проект побудований за модульною схемою. Кожна функціональна частина винесена в окремий NestJS-модуль, що включає контролери, сервіси, DTO та ентиті.

Такий підхід забезпечує чітке розділення відповідальності, спрощує супровід системи, а також дозволяє легко масштабувати функціональність без порушення існуючої архітектури. Завдяки дотриманню принципів інверсії залежностей і повторного використання компонентів, структура застосунку залишається впорядкованою навіть за умов активного розширення функцій. [7, с. 11]

Основні модулі проекту охоплюють:

1. AuthModule – реалізація авторизації та аутентифікації користувачів, генерація JWT-токенів, захист маршрутів за допомогою **JwtAuthGuard**.
UserModule – управління профілем користувача (зміна імені, пароля, email, аватарки).
2. WorkspaceModule – створення, редагування та управління робочими просторами, запрошення учасників через унікальні посилання.
3. BoardModule – функціональність для роботи з дошками (створення, редагування, видалення, обране, останні перегляди).
4. ColumnModule – реалізація CRUD-операцій для колонок, зокрема переміщення, копіювання.
5. CardModule – повна логіка роботи з картками: створення, редагування, коментарі, прикріплення користувачів, мітки, вкладення.
6. LabelModule – керування мітками (створення, редагування, видалення).
7. AttachmentModule – завантаження та обробка файлів, прив'язаних до карток.

Відповідно до принципів NestJS, основні компоненти реалізовані за наступною структурою [6]:

- Controllers – обробка вхідних HTTP-запитів та маршрутизація;
- Services – реалізація бізнес-логіки застосунку, взаємодія з базою даних через ORM;
- DTO (Data Transfer Objects) – валідація та структура запитів, перевірка через **class-validator**;
- Entities – відображення структури таблиць бази даних;
- Guards та Interceptors – для захисту маршрутів та серіалізації відповідей.

Описана архітектурна модель є результатом адаптації найкращих практик серверної розробки до потреб сучасної системи керування проектами. Вона дозволяє забезпечити одночасно і швидкодію, і гнучкість у підтримці численних користувачів та даних. [8]

У якості прикладу далі наведено приклад передавання даних методом POST, що використовується для створення дошки:

```

@Post()
@ApiOperation({ summary: 'Create board in workspace (admin only)' })
@ApiConsumes('multipart/form-data')
@ApiBody({
  description: 'Form data with board name and optional image',
  schema: {
    type: 'object',
    properties: {
      name: { type: 'string', example: 'Project A' },
      image: {
        type: 'string',
        format: 'binary',
      },
    },
    required: ['name'],
  },
})
@ApiResponse({ status: 201, description: 'Board successfully created', type: Board })
@UseInterceptors(
  FileInterceptor('image', {
    storage: diskStorage({
      destination: './uploads/board-backgrounds',
      filename: (req, file, callback) => {
        const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1e9);
        callback(null, `${uniqueSuffix}${extname(file.originalname)}`);
      },
    }),
  })
)
async create(@Param('workspaceId') workspaceId: string, @CurrentUser() user: User, @UploadedFile()
image: Express.Multer.File, @Body() dto: CreateBoardDto) {
  return this.boardService.create(workspaceId, user, dto, image);
}

```

Фрагмент методу POST використовується для створення дошки

Базу даних було спроектовано з урахуванням логіки доменної області та вимог до масштабованості системи. Її структура базується на чітко визначених сутностях та зв'язках між ними, що дозволяє забезпечити консистентність, цілісність і гнучкість при реалізації функціоналу.

Кожен Workspace (робочий простір) може містити декілька Board (дошок), кожна дошка включає Column (колонки), у які розміщуються Card (картки). Картки, у свою чергу, можуть мати пов'язану інформацію — коментарі (Comment), мітки (Label), вкладення (Attachment) та призначених користувачів (Users). Для реалізації складних

зв'язків використано додаткові проміжні таблиці-зв'язки типу *many-to-many*, що дозволяє ефективно керувати ролями, правами доступу та асоціаціями між об'єктами.

Основні таблиці бази даних вказані на Рисунку 2.2 та рисунку 2.3:

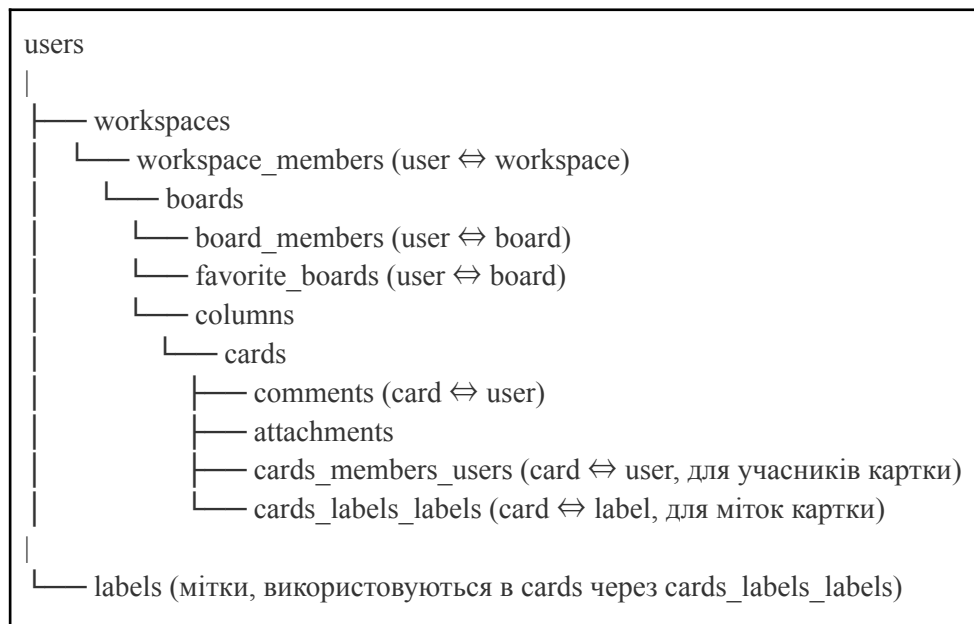


Рисунок 2.2 – Структура бази даних

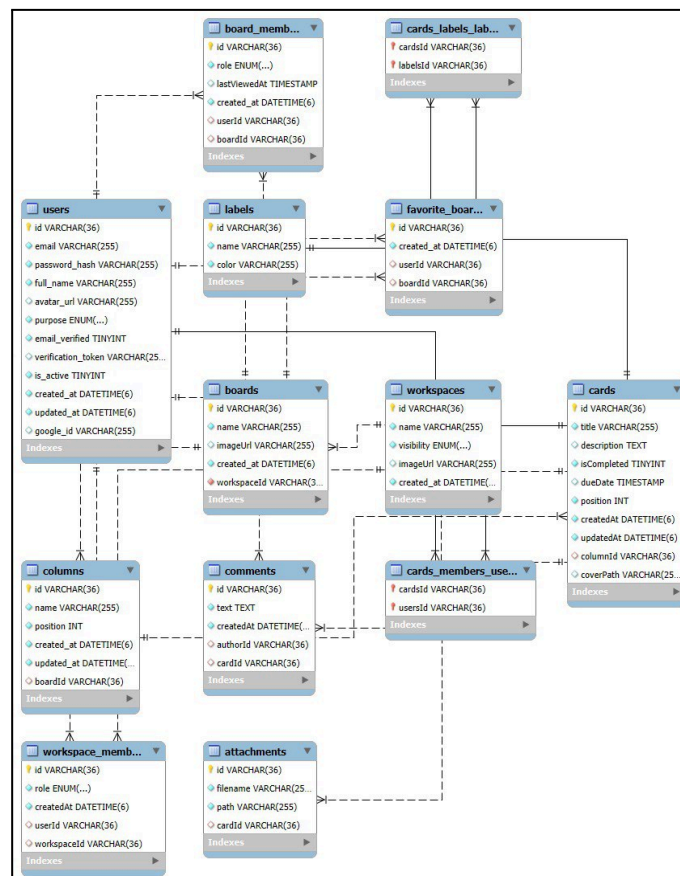


Рисунок 2.3 – Структура бази даних

Для обробки файлів, таких як аватари користувачів, вкладення до карток та обкладинки до дошок, у серверній частині системи реалізовано механізм завантаження з використанням middleware-бібліотеки **Multer** — проміжного програмного забезпечення для Node.js, що спеціалізується на обробці **multipart/form-data** і є ефективним рішенням для приймання файлів з клієнта [4].

Файли зберігаються локально у спеціально визначених директоріях, зокрема: **/uploads/avatars/**, **/uploads/attachments/**, **/uploads/covers/**. Після завантаження шляхи до файлів зберігаються у відповідних полях бази даних, що дозволяє системі динамічно повертати клієнту URL-адреси для доступу до зображень. Додатково реалізовано перевірку MIME-типів і обмеження розміру файлів відповідно до безпекових політик.

Типовий приклад застосування Multer — завантаження обкладинки для картки. Контролер приймає файл через декоратор **@UploadedFile()** і передає його у відповідний метод сервісу. У свою чергу, сервісна логіка зберігає шлях до файлу в полі **coverPath** сутності **Card**. Приклад реалізації:

```
@Post(':cardId/cover')
@UseInterceptors(FileInterceptor('file', { dest: './uploads/covers' }))
uploadCover(
  @Param('workspaceId') workspaceId: string,
  @Param('boardId') boardId: string,
  @Param('columnId') columnId: string,
  @Param('cardId') cardId: string,
  @CurrentUser() user: User,
  @UploadedFile() file: Express.Multer.File,
) {
  return this.cardService.uploadCardCover(workspaceId, boardId, columnId, cardId, file, user);
}
```

Фрагмент контролера завантаження обкладинки картки з використанням Multer.

Опис прикладного інтерфейсу програмування (API) реалізовано за допомогою Swagger — інструмента, який забезпечує автоматичну генерацію інтерактивної документації на основі анотацій у коді. Swagger дозволяє наочно відображати всі доступні маршрути, типи HTTP-запитів, параметри, тіла запитів та приклади відповідей. Доступ до документації здійснюється за шляхом **/api/docs**. [5]

У проєкті повністю охоплено CRUD-операції (створення, читання, оновлення, видалення) для всіх основних сутностей: користувачів, робочих просторів, дошок, колонок, карток, міток, коментарів тощо. Валідація параметрів запитів здійснюється на основі DTO-класів із використанням **class-validator**, що гарантує відповідність даних бізнес-вимогам і захищає систему від некоректного введення.

Завдяки чіткому розділенню логіки між модулями, контролерами, сервісами й DTO, система зберігає високий рівень масштабованості та легкість підтримки. Реалізовані архітектурні підходи відповідають принципам Clean Architecture, що дозволяє легко розширювати функціональність, підтримувати якість коду та забезпечити стабільну інтеграцію між фронтендом і бекендом.

РОЗДІЛ 3 РЕЗУЛЬТАТИ ВИПРОБУВАНЬ

Результатом компіляції програмного коду клієнтської частини є вебзастосунок, що працює в браузері користувача. Основні випробування проведені у браузерах Chrome та Firefox на Персональних Комп'ютерах з ОС Windows 11.

Початковий етап взаємодії користувача з інформаційною системою передбачає проходження процедури реєстрації. У межах цього процесу здійснюється надсилання POST-запиту до маршруту `/api/auth/register` з основними параметрами: ім'я, електронна пошта та пароль. Відповідь сервера містить сформований об'єкт нового користувача у форматі JSON, включно з хешем пароля, токеном верифікації, позначкою активності та позначкою про підтвердження електронної пошти. На рисунку 3.1 вказано результат компілювання JSON-запиту та відповідь при реєстрації користувача:

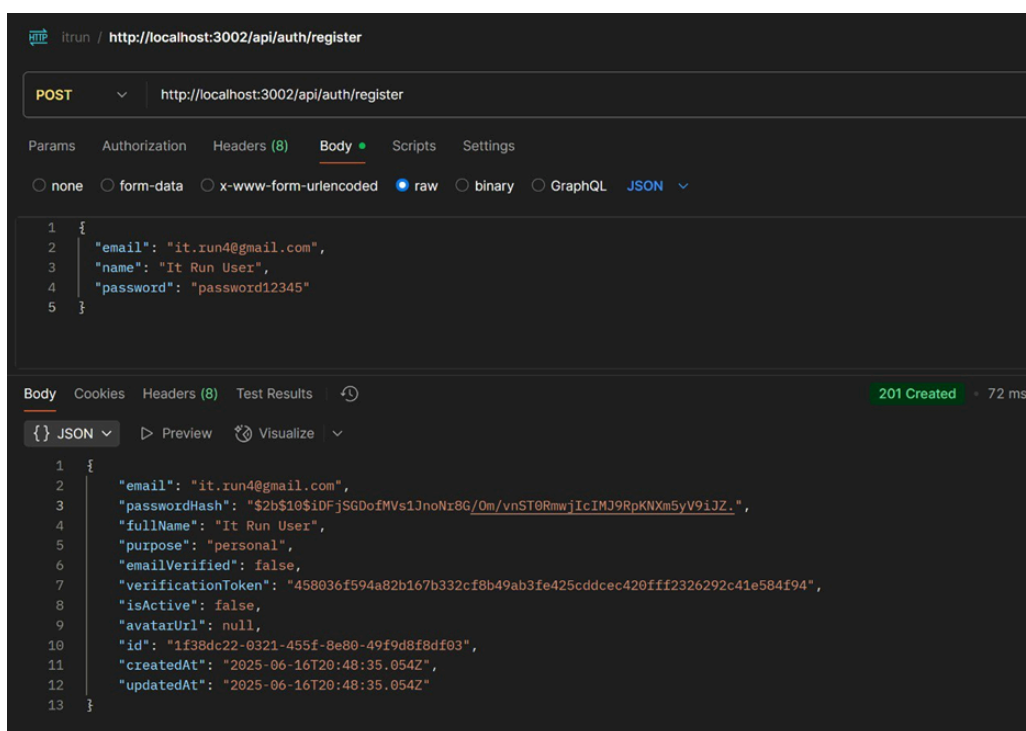


Рисунок 3.1 – Результат компілювання JSON-запиту

Наступним кроком після реєстрації є відправлення запиту на верифікацію пошти. У системі це реалізовано шляхом PATCH-запиту до маршруту `/api/auth/send-verification`, у тілі якого передається ідентифікатор користувача (`userId`). У відповідь повертається статус про успішне надсилання листа з токеном підтвердження. Результат компіляції вказано на рисунку 3.2:

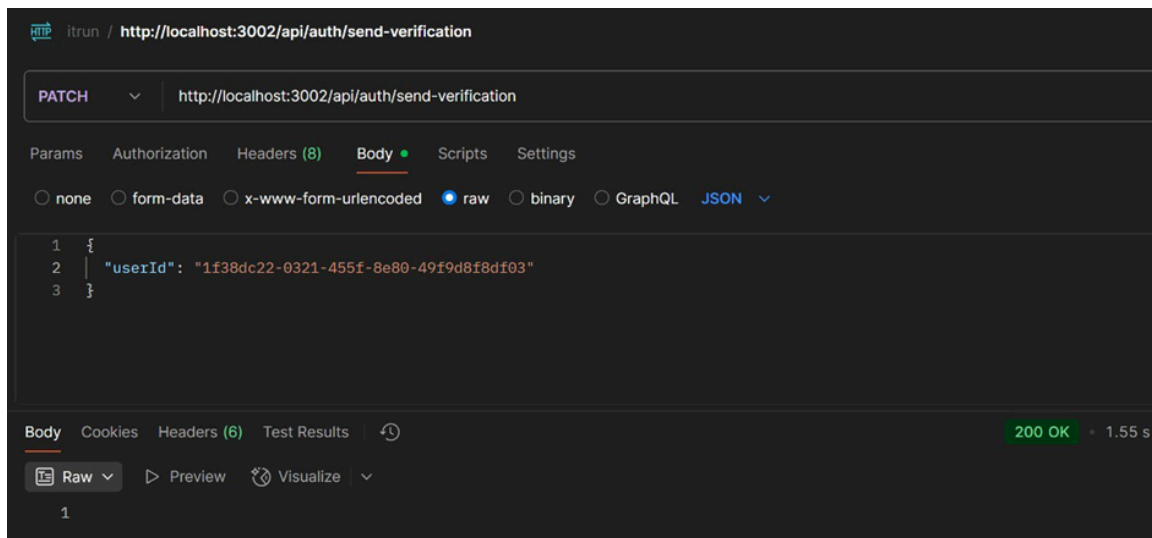


Рисунок 3.2 – Успішне надсилання листа з токеном підтвердження

На вказану при реєстрації адресу користувача надходить автоматичне повідомлення з посиланням для верифікації. У тілі листа — пряме посилання на маршрут підтвердження, який містить унікальний токен, прив'язаний до конкретного користувача. Лист, що приходить на пошту користувачу для підтвердження реєстрації вказано на рисунку 3.3:

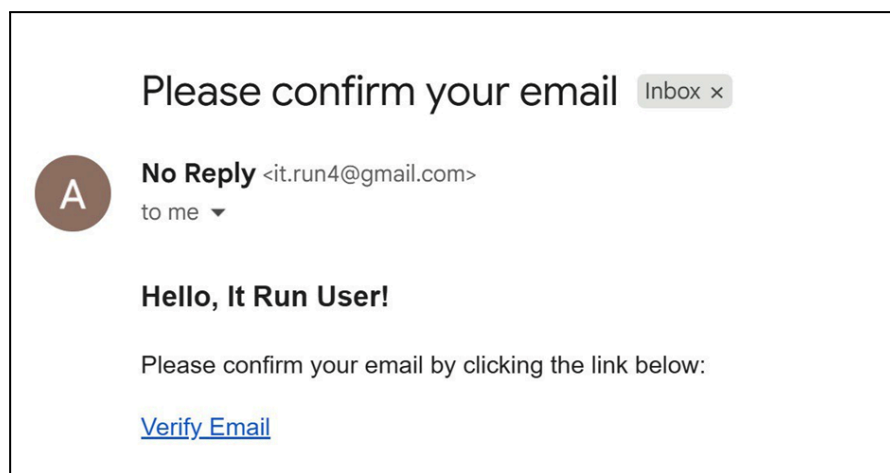


Рисунок 3.3

Після переходу за отриманим посиланням здійснюється GET-запит на маршрут `/api/auth/verify`, у параметрах якого передається токен. У разі успішної перевірки токenu сервер повертає статус `200 OK`, а в полі `emailVerified` користувача в базі встановлюється значення `true`. Перевірка токenu верифікації на рисунку 3.4:

3.11, 3.12, 3.13 :

User			^
GET	/api/user/fullName	Get user name	🔒 ▼
PATCH	/api/user/update-name	Update full name	🔒 ▼
GET	/api/user/email	Get user email	🔒 ▼
PATCH	/api/user/update-email	Update email address	🔒 ▼
PATCH	/api/user/update-password	Update password	🔒 ▼
GET	/api/user/avatar	Get user avatar URL	🔒 ▼
PATCH	/api/user/update-avatar	Update avatar URL	🔒 ▼
GET	/api/user/purpose	Get user purpose	🔒 ▼
PATCH	/api/user/update-purpose	Update user purpose	🔒 ▼
GET	/api/user/workspaces	Get user workspaces	🔒 ▼

Рисунок 3.6

Auth			^
POST	/api/auth/check-email	Check if email is already in use	▼
POST	/api/auth/register	Register new user	▼
PATCH	/api/auth/upload-avatar	Upload user avatar	▼
PATCH	/api/auth/set-purpose	Set user purpose	🔒 ▼
PATCH	/api/auth/send-verification	Send initial verification email	▼
GET	/api/auth/verify	Verify email address	▼
POST	/api/auth/resend-verification	Send verification email again	▼
POST	/api/auth/login	User login	▼
PATCH	/api/auth/change-email	Change user email on registration page	▼

Рисунок 3.7

Workspace			^
GET	/api/workspace/{id}	Get workspace by id	🔒 ▼
DELETE	/api/workspace/{id}	Delete workspace (admin only)	🔒 ▼
POST	/api/workspace	Create a new workspace	🔒 ▼
GET	/api/workspace/{id}/members	Get all members of workspace	🔒 ▼
PATCH	/api/workspace/{id}/name	Update workspace name	🔒 ▼
PATCH	/api/workspace/{id}/image	Upload or update workspace background image	🔒 ▼
PATCH	/api/workspace/{id}/visibility	Update workspace visibility	🔒 ▼
GET	/api/workspace/{id}/invite-link	Generate invite link (admin only)	🔒 ▼
POST	/api/workspace/join	Join a workspace using invite token	🔒 ▼
DELETE	/api/workspace/{workspaceId}/members/{userId}	Remove member from workspace (admin only)	🔒 ▼
DELETE	/api/workspace/{workspaceId}/leave	Leave workspace (for members)	🔒 ▼
GET	/api/workspace/own	Get workspaces where user is admin	🔒 ▼
GET	/api/workspace/guest	Get workspaces where user is guest (member)	🔒 ▼

Рисунок 3.8

Board			^
POST	/api/workspace/{workspaceId}/board	Create board in workspace (admin only)	🔒 ▼
GET	/api/workspace/{workspaceId}/board	Get all boards in a workspace (only for workspace members)	🔒 ▼
GET	/api/workspace/{workspaceId}/board/{id}	Get one board by ID (only for board members)	🔒 ▼
BoardCommon			^
PATCH	/api/boards/{id}	Update board name or image (for members)	🔒 ▼
DELETE	/api/boards/{id}	Delete board by ID (for members)	🔒 ▼
GET	/api/boards/favorites	Get user's favorite boards	🔒 ▼
POST	/api/boards/{id}/favorite	Add board to favorites	🔒 ▼
DELETE	/api/boards/{id}/favorite	Remove board from favorites	🔒 ▼
GET	/api/boards/recent	Get last 5 viewed boards by user	🔒 ▼

Рисунок 3.9

Column			^
POST	/api/workspace/{workspaceId}/board/{boardId}/column	Create a column in board	🔒 ▼
GET	/api/workspace/{workspaceId}/board/{boardId}/column	Get all columns of a board	🔒 ▼
PATCH	/api/workspace/{workspaceId}/board/{boardId}/column/{id}	Update column name	🔒 ▼
DELETE	/api/workspace/{workspaceId}/board/{boardId}/column/{id}	Delete column	🔒 ▼
POST	/api/workspace/{workspaceId}/board/{boardId}/column/{id}/move	Move column to new position	🔒 ▼
POST	/api/workspace/{workspaceId}/board/{boardId}/column/{id}/copy	Copy column with cards	🔒 ▼

Рисунок 3.10

CardBoard			^
GET	/api/workspace/{workspaceId}/board/{boardId}/cards	Get board with columns (id only) and full cards	🔒 ▼
PATCH	/api/workspace/{workspaceId}/board/{boardId}/column/{sourceColumnId}/move-all/{targetColumnId}	Move all cards from one column to another	🔒 ▼
Label			^
POST	/api/label	Create label	▼
GET	/api/label	Get all labels	▼
GET	/api/label/{id}	Get label by id	▼
PATCH	/api/label/{id}	Update label by id	▼
DELETE	/api/label/{id}	Delete label by id	▼

Рисунок 3.11

Card			^
POST	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card	Add comment to card	🔒 ▼
GET	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card	Get all comments of a card	🔒 ▼
PATCH	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card/{cardId}	Update card	🔒 ▼
DELETE	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card/{cardId}	Delete card	🔒 ▼
PATCH	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card/{id}/complete		🔒 ▼
PATCH	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card/{id}/uncomplete		🔒 ▼
POST	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card/{cardId}/move	Move card to another column or position	🔒 ▼
GET	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card/{cardId}/members	Get all members of the card	🔒 ▼
POST	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card/{cardId}/members/{userId}	Add member to the card (admin only)	🔒 ▼
DELETE	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card/{cardId}/members/{userId}	Remove member from the card (admin only)	🔒 ▼
POST	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card/{cardId}/subscribe	Subscribe current user to the card	🔒 ▼
POST	/api/workspace/{workspaceId}/board/{boardId}/column/{columnId}/card/{cardId}/unsubscribe	Unsubscribe current user from the card	🔒 ▼

Рисунок 3.12

Schemas		^
UpdateNameDto	>	
UpdateEmailDto	>	
UpdatePasswordDto	>	
UpdateAvatarDto	>	
UpdatePurposeDto	>	
EmailCheckDto	>	
CreateUserDto	>	
SetPurposeDto	>	
LoginDto	>	

Рисунок 3.13

ВИСНОВКИ

У рамках проведеного дослідження та для виконання дипломного проєкту з реалізації серверної частини інформаційної системи організації управління програмними проєктами та завданнями було здійснено всебічний аналіз сучасних архітектурних підходів, обґрунтовано вибір інструментів розробки та реалізовано функціонально повноцінний backend, що відповідає вимогам надійності, масштабованості та продуктивності.

Основою серверної частини обрано фреймворк NestJS, який у поєднанні з мовою програмування TypeScript забезпечив модульну структуру застосунку з чітким розподілом логіки за принципами Clean Architecture. Кожна функціональна область була винесена в окремий модуль з відповідними контролерами, сервісами, DTO-класами та ентиті, що сприяє розділенню відповідальності та спрощує розширення проєкту.

Для взаємодії з базою даних використано реляційну СКБД MySQL разом з ORM-бібліотекою TypeORM, яка надала змогу реалізувати складні зв'язки між сутностями, забезпечити каскадні операції, lazy/eager-завантаження та типобезпечний доступ до даних. Такий підхід дозволив забезпечити гнучкість структури даних та покращити читабельність і підтримуваність коду.

У межах реалізації серверної частини було повністю охоплено CRUD-операції для всіх ключових сутностей системи: користувачів, робочих просторів, дошок, колонок, карток, міток, коментарів тощо. Проведено валідацію вхідних даних на основі DTO та бібліотеки class-validator, що підвищило загальну надійність та захищеність застосунку.

Завдяки впровадженню архітектурних принципів, таких як інверсія залежностей, повторне використання компонентів та централізація обробки помилок, серверна частина системи вийшла гнучкою до змін, зручною для тестування і масштабованою для подальшого розвитку.

Таким чином, серверна частина інформаційної системи спроектована, реалізована та випробувана відповідно до сучасних вимог до web-розробки. Реалізоване рішення забезпечує стабільну взаємодію з клієнтською частиною, ефективну обробку бізнес-логіки, підтримку інтеграції з базою даних і створює надійну основу для подальшого розвитку системи управління програмними проєктами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ:

1. Ткаченко К. О., Мороз О. О. Типізація даних в JavaScript за допомогою TypeScript // Цифрова економіка та інформаційні технології : матеріали Міжнародної науково-практичної конференції, м. Київ, 2021 р. / М-во освіти і науки України, Державний університет інфраструктури та технологій. — Київ : Видавничий центр ДУІТ, 2021. — С. 100 - 102.
2. Андреев А. С. Перспективи використання PHP та MySQL в проєктах // Автоматизація та приладобудування (Automation and Development of Electronic Devices – ADED-2023) : зб. студент. наук. ст. / Харків. нац. ун-т радіоелектроніки ; редкол.: І. Ш. Невлюдов [та ін.]. — Харків : ХНУРЕ, 2023. — Вип. 1. — С. 66–69. — 336 с.
3. Палєга Р. В., Карпенко Н. В., Герасимов В. В. Особливості авторизації за допомогою JWT токенів // Інформаційні технології і автоматизація – 2024 : матеріали XVII міжнар. наук.-практ. конф., Одеса, 31 жовтня – 1 листопада 2024 р. — Одеса : Видавництво ОНТУ, 2024. — С. 208. — 847 с.
4. Multer [Електронний ресурс] // npm. — Режим доступу: <https://www.npmjs.com/package/multer> — Дата звернення: 19 червня 2025.
5. Swagger [Електронний ресурс]. — Офіційний сайт інструментів для генерації, візуалізації та тестування API-документації. — Режим доступу: <https://swagger.io/>
6. Mohitu. NestJS (Controller, Service, Module, Entity, DTO, Decorator, Guard, and, Routing) [Електронний ресурс] / Mohitu // *Medium*. — 2023. — 11 серпня. — Режим доступу: <https://medium.com/@mohitu531/nestjs-7c0eb5655bde>
7. Cosgaya Capel M. *Creation of a middleware for blockchain interaction based on NestJS* : Master Thesis / Advisors: Muñoz Tapia J. L., Genés Durán R. — Barcelona : Universitat Politècnica de Catalunya, Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona, 2023. — 77 p. — Режим доступу:: <https://upcommons.upc.edu/handle/2117/391622>
8. Chowdhury N. *Nest JS Fundamentals: Modules, Services, and Pipes Explained* [Electronic resource] // *DEV Community*. — Published: 10.05.2024. — Updated: 06.02.2024. — Access mode: https://dev.to/nadim_ch0wdhury/what-is-module-services-pipe-in-nest-js-325d