МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»

Кафедра інженерії програмного забезпечення

КУРСОВА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

з дисципліни: «Бази даних»
на тему:
«Розробка бази даних для інструменту керування завданнями»

студента ІІ курсу групи ВТ-21-1
спеціальності 121 «Інженерія програмного забезпечення.
Бабушка Андрія Сергійовича
(прізвище, ім'я та по-батькові)

Керівник: Кравченко С.М.

Дата захисту: "16" травня 2023 р.
Національна шкала_____
Кількість балів: _____
Оцінка: ECTS _____

Члени комісії _____ І. І. Сугоняк
(підпис) (прізвище та ініціали)
_____ О. В. Коротун
(підпис) (прізвище та ініціали)
_____ С. М. Кравченко
(підпис) (прізвище та ініціали)
_____ О.В. Чижмотря
(підпис) (прізвище та ініціали)

Житомир – 2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
Факультет інформаційно-комп'ютерних технологій
Кафедра інженерії програмного забезпечення
Освітній рівень: бакалавр
Спеціальність 121 «Інженерія програмного забезпечення»

«ЗАТВЕРДЖУЮ»
В. о. зав. кафедри
_____ А.В.Морозов
"____"_____20__ р.

ЗАВДАННЯ
НА КУРСОВИЙ ПРОЕКТ СТУДЕНТУ
Бабушку Андрію Сергійовичу

1. Тема роботи: Розробка бази даних для інструменту керування завданнями, керівник роботи: старший викладач Кравченко С.М.
2. Строк подання студентом: "_16_"__травня___ 2023р.
3. Вихідні дані до роботи: Розробити базу даних для інструменту керування завданнями.
4. Зміст розрахунково-пояснювальної записки(перелік питань. Які підлягають розробці)
   1. Постановка завдання
   2. Аналіз аналогічних розробок
   3. Алгоритми роботи програми
   4. Опис роботи програми
   5. Програмне дослідження
5. Перелік графічного матеріалу(з точним зазначенням обов'язкових креслень)
   1. Посилання на репозиторій: https://github.com/AndriiBabushko/task-management-tool
   2. Презентація до КП
6. Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посади консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1,2,3,4	Кравченко С.М.		

7. Дата видачі завдання "15" березня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів курсового проекту	Строк виконання етапів проекту	Примітки
1	Постановка задачі	01.03.2023	Виконано
2	Пошук, огляд та аналіз аналогічних розробок	13.03.2023	Виконано
3	Формулювання технічного завдання	14.04.2023	Виконано
4	Опрацювання літературних джерел	16.04.2023	Виконано
5	Проектування структури	18.05.2023	Виконано
6	Написання програмного коду	19.05.2023	Виконано
7	Відлагодження	08.06.2023	Виконано
8	Написання пояснювальної записки	14.01.2023	Виконано
9	Захист	16.01.2023	Виконано

Студент _____ Бабушко А.С.
(підпис) (прізвище та ініціали)

Керівник проекту _____ Кравченко С.М.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Завданням на курсову роботу було розробка бази даних для інструменту керування завданнями.

Пояснювальна записка до курсової роботи на тему «Розробка бази даних для інструменту керування завданнями» складається з вступу, переліку умовних скорочень, чотирьох розділів, висновків, списку використаної літератури та додатку.

Текстова частина викладена на 48 сторінках друкованого тексту.

Пояснювальна записка має 44 сторінок додатків. Список використаних джерел містить 11 найменувань і займає 1 сторінку. В роботі наведено 30 рисунків. Загальний обсяг роботи – 93 сторінки.

У першому розділі було обґрунтовано актуальність та причини створення програмного продукту.

У другому розділі проведено проектування і розробка програмного продукту.

У третьому розділі проведено тестування програмного продукту.

Висновок містить в собі результати виконаної роботи зі створення програмного продукту.

У додатку наведений лістинг розробленого програмного продукту.

Ключові слова: ООП, ДОСТУП, ЗБЕРЕЖЕННЯ ІНФОРМАЦІЇ, БАЗА ДАНИХ, ЗАВДАННЯ, NODE JS, REACT JS, MONGO DB, ІНСТРУМЕНТ, МОДЕЛЬ, КОНТРОЛЛЕР, РОУТИ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

WS – WebStorm

JS – мова програмування «JavaScript»

TS – мова програмування «Typescript»

HTML – Hyper Text Markup Language

CSS – Cascade Style Sheet

ООП – об’єктно орієнтовний підхід

ПЗ – Програмне забезпечення

БД – База даних

MongoDB – База даних «Mongo»

REST – Representational State Transfer, дослівно: передача представницького стану.

API – Application programming interface, дослівно: Інтерфейс прикладного програмування.

ВСТУП

У курсовій роботі буде наведено процес створення бази даних з використанням Mongo DB для інструменту керування завданнями.

Інструмент керування завданнями - це програмний продукт або інструмент, який допомагає управляти та організовувати завдання та проекти, встановлювати терміни та пріоритети, стежити за виконанням завдань та оцінювати продуктивність роботи.

Такі інструменти часто мають функції, такі як створення списку завдань, розподіл завдань між членами команди, спільна робота над проектом, моніторинг часу, призначення термінів, повідомлення про статус завдань та багато іншого. Вони дозволяють керівникам та командам більш ефективно організовувати свою роботу та досягати поставлених цілей.

Актуальність теми: обумовлена бурхливим розвитком IT сфери та потребою в організації завдань, термінів та пріоритетів для продуктивної роботи як звичайної людини, так і співробітників в компанії.

Предмет дослідження: робота та створення складної системи бази даних для можливості створення, перегляду, оновлення та видалення завдань та всіх пов’язаних з ними баз для зручного та приємного використання програми для покращення та спрощення життя людини.

Об’єкт дослідження: процес та реалізація бази даних Mongo DB з використанням мови програмування Node JS.

Мета роботи: розробка бази даних для інструменту керування завданнями, дослідження особливостей бази даних, проектування та налагодження.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1 Аналіз задачі, засобів та методів її вирішення

Перед тим, як почати розробляти та писати програмний код, потрібно зробити аналіз та визначити порядок виконання завдання.

Задача на курсову роботу полягає у створенні бази даних. У процесі аналізу роботи було виділено наступні умови та функції, які має містити даний програмний продукт:

- Змоделювати загальну схему додатку, схему відношень таблиць БД, продумати логіку роботи додатку та інше.
- Розробити REST API для майбутніх запитів на клієнтській частині додатку.
- Визначити, як саме користувач буде взаємодіяти з програмою.
- Розділити ролі користувачів сайту.
- Розробити можливість зміни важливих даних сайту за допомогою інтерактивного інтерфейсу для адміністраторів сайту та користувацьких даних для звичайних користувачів.
- Розробити можливість перегляду та оцінки виконаних чи поставлених задач інших користувачів системи.
- Розробити сторінки: головна, стрічка задач, категорії задач, налаштування сайту, авторизація, реєстрація, створення груп користувачів для роботи над задачами, сторінки повідомлень про помилки або успіх виконаних операцій.

Під час створення сайту пріоритети було віддано розробці основних можливостей перегляду задач, їх зміну, виконання, видалення, сортування та отримання додаткової інформації щодо них.

Для розробки додатку було обрано таку середу, як WS. Проект розроблявся за допомогою таких веб-технологій, як HTML, CSS(SCSS), Node JS(Express JS),

React JS(Typescript) та деяких додаткових бібліотек, наприклад Tailwind CSS, для полегшення процесу написання програмного коду продукту.

Для початку було вирішено розробити основу для подальшого розвитку та розширення програмного продукту. Це стосується структури проекту, його логіки, набору функціоналу, локацій певних файлів.

Далі було вирішено розробити основну сторінку сайту, хедер та футер, сторінку налаштувань користувача, авторизацію та реєстрацію для використання цієї інформації у подальшому. Всі дані, які надійшли чи надійдуть з клієнтської частини програми будуть перевірятися та потрапляти на сервер, який буде зберігати їх в БД у відповідну колекцію та діставати ці дані при потребі клієнта чи сервера.

Також, буде створено сторінку для адміністраторів сайту, які зможуть вносити нові актуальні дані, змінювати застарілі та видаляти непотрібні.

1.2 Аналіз існуючого програмного забезпечення за тематикою курсової роботи.

При аналізі все існуючого ПЗ за тематикою курсової роботи було виявлено безліч проектів. Кожен з них має індивідуальні принципи роботи та логіку.

Кожен сайт, який пропонує послуги керування виконані за допомогою різних мов програмування та мають різних зовнішній вигляд. Наприклад рисунок 1.1 та рисунок 1.2 демонструють різницю у дизайні та реалізації сайту курсів з програмування.

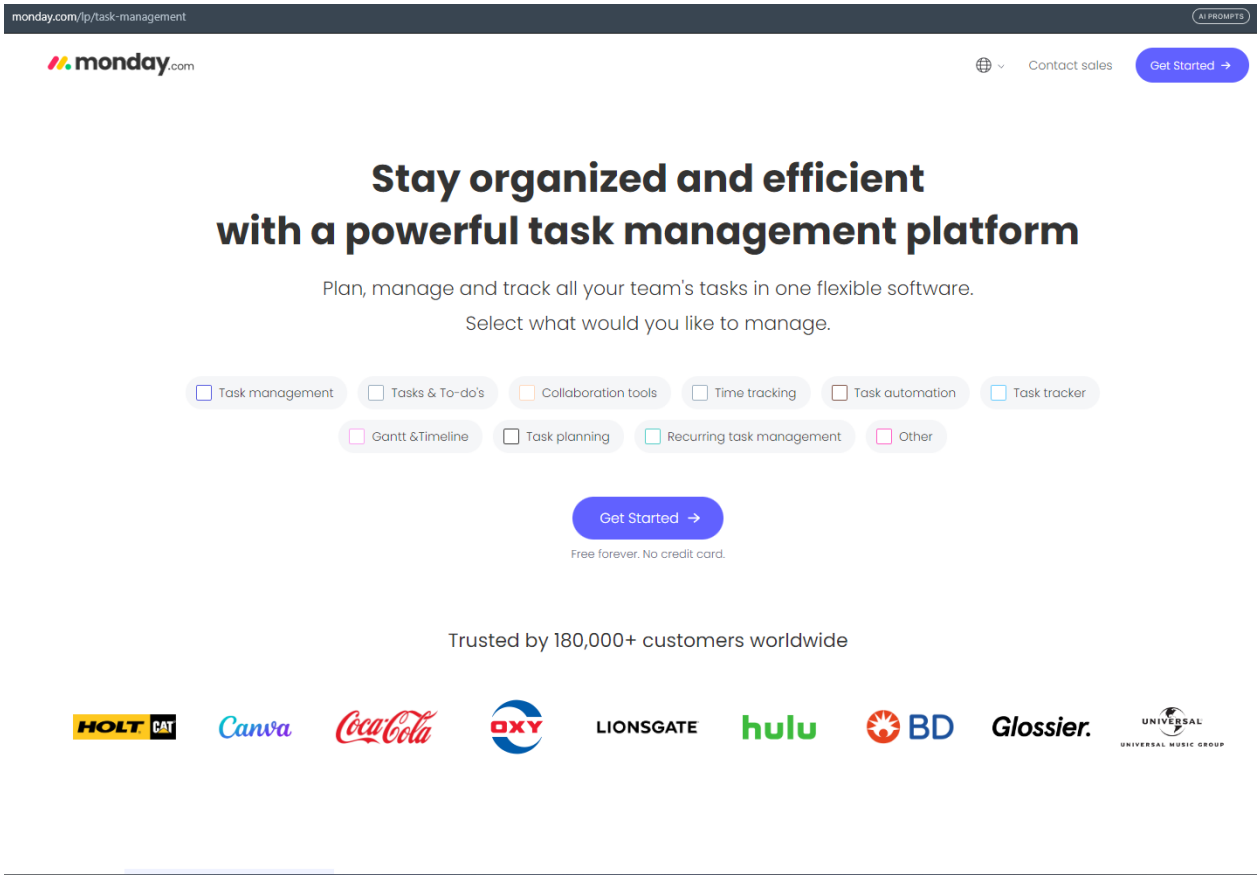


Рис. 1.1. – перший приклад реалізації сайту керування завданнями.

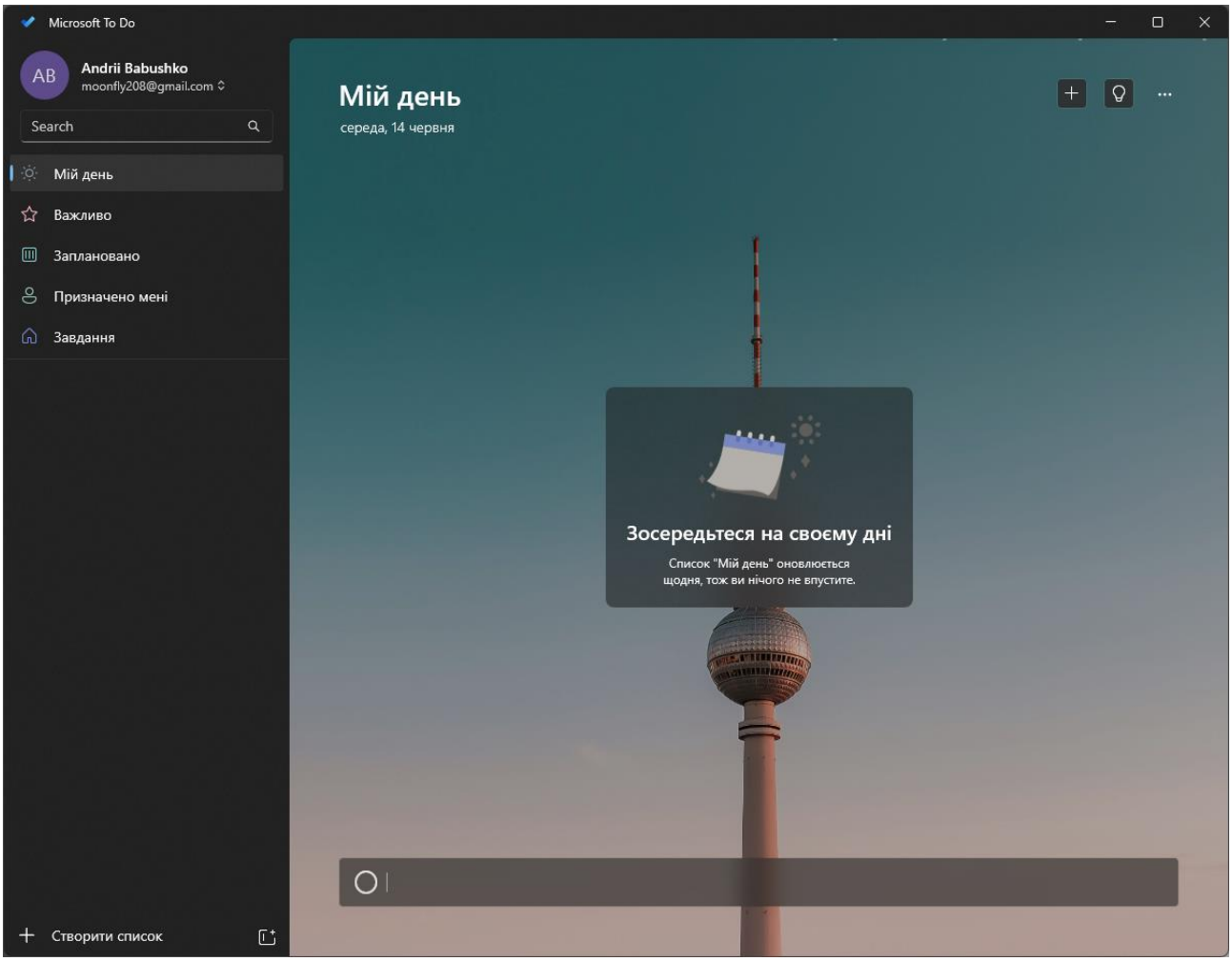


Рис. 1.2. – другий приклад реалізації сайту керування завданнями.

На цих рисунках ми можемо побачити, що кожен сайт відрізняється користувацьким інтерфейсом та має свої особливості та дизайн. На рисунку 1 як і на рисунку 2 інтерфейс виглядає повноцінним та інформаційним, але якщо заглянути глибше, то функціонал у кожного буде побудовано по різному. Але, все ж таки, у другому варіанті додаткового функціоналу більше, що дає користувачеві більше інтерактивності та можливостей вибору як саме взаємодіяти з додатком.

Таким чином, при розробці програми слід враховувати, що чим простіший та яскравіший інтерфейс має сайт, тим він стає більш привабливішим, але все ж повинен зберігати в собі основний функціонал, яким буде керуватися сайт. Тому це є ідеальним варіантом та вибором для розробки програмного продукту.

1.3 Обгрунтування вибору засобів реалізації курсового проекту.

Зараз існує досить багатий вибір технологій які можна використати для розробки програмного продукту за тематикою курсової роботи, особливо підтримки бази даних найефективнішим способом. У ході розробки програми було поставлено вибір між PostgreSQL, яка є реляційною базою даних та MongoDB яка є не реляційною. Далі наведено таблицю, яка порівнює дані бази даних за деякими параметрами.

Таблиця 1. Аналіз СУБД

Параметр	MongoDB	PostgreSQL
Тип бази даних	Документ-орієнтована	Реляційна
Мова запитів	MongoDB Query Language (MQL)	SQL (Structured Query Language)
Схема бази даних	Гнучка, не потребує фіксованої схеми	Статична, вимагає фіксованої схеми
Підтримка транзакцій	Відсутня	Присутня
Розширення	Горизонтальне масштабування, реплікація	Вертикальне та горизонтальне масштабування, реплікація
Швидкість запитів	Висока	Висока
Підтримка JOIN	Відсутня	Присутня
Формат збереження даних	BSON (Binary JSON)	Рядковий, числовий, бінарний, JSON та ін.
Підтримка індексів	Присутня	Присутня
Масштабованість	Горизонтальне та вертикальне масштабування	Вертикальне масштабування

Таким чином, було визначено, що для нашої теми курсового проєкту найоптимальнішим підійте MongoDB через гарну підтримку JSON, легке входження, швидкість та гнучкість.

Висновки з першого розділу

У ході виконання першого розділу було проаналізовано схожі варіанти реалізації інтерфейсу та баз даних інструменту керування завданнями та вирішено питання способу реалізації.

Було визначено основні інструменти для написання програми, її функціонал та варіативність. Також було визначено, що основне у складі повинен мати продукт, а що другорядне. Було визначено переваги та недоліки деяких схожих програмних продуктів, технології для написання продукту та обрано найкращий.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Проектування загального алгоритму роботи програми

За результатами попередніх досліджень було визначено необхідну структуру та архітектуру програми.

Програма реалізує:

- CRUD функціонал БД(читання, створення, оновлення, видалення даних);
- Рівні доступу до змін даних сайту(адміністратори, користувачі, власники груп);
- Адмін панель;
- Можливість перегляду як свої створених задач, так і задач створених іншими користувачами;
- Агрегації для аналізу даних які вже збережені в базі даних;
- Резервування копії бази даних та її відновлення;

Було проведено моделювання структури програми для візуалізації програмних можливостей та функціональних умов системи. На схемі 2.1 зображено загальну схему роботи програми.

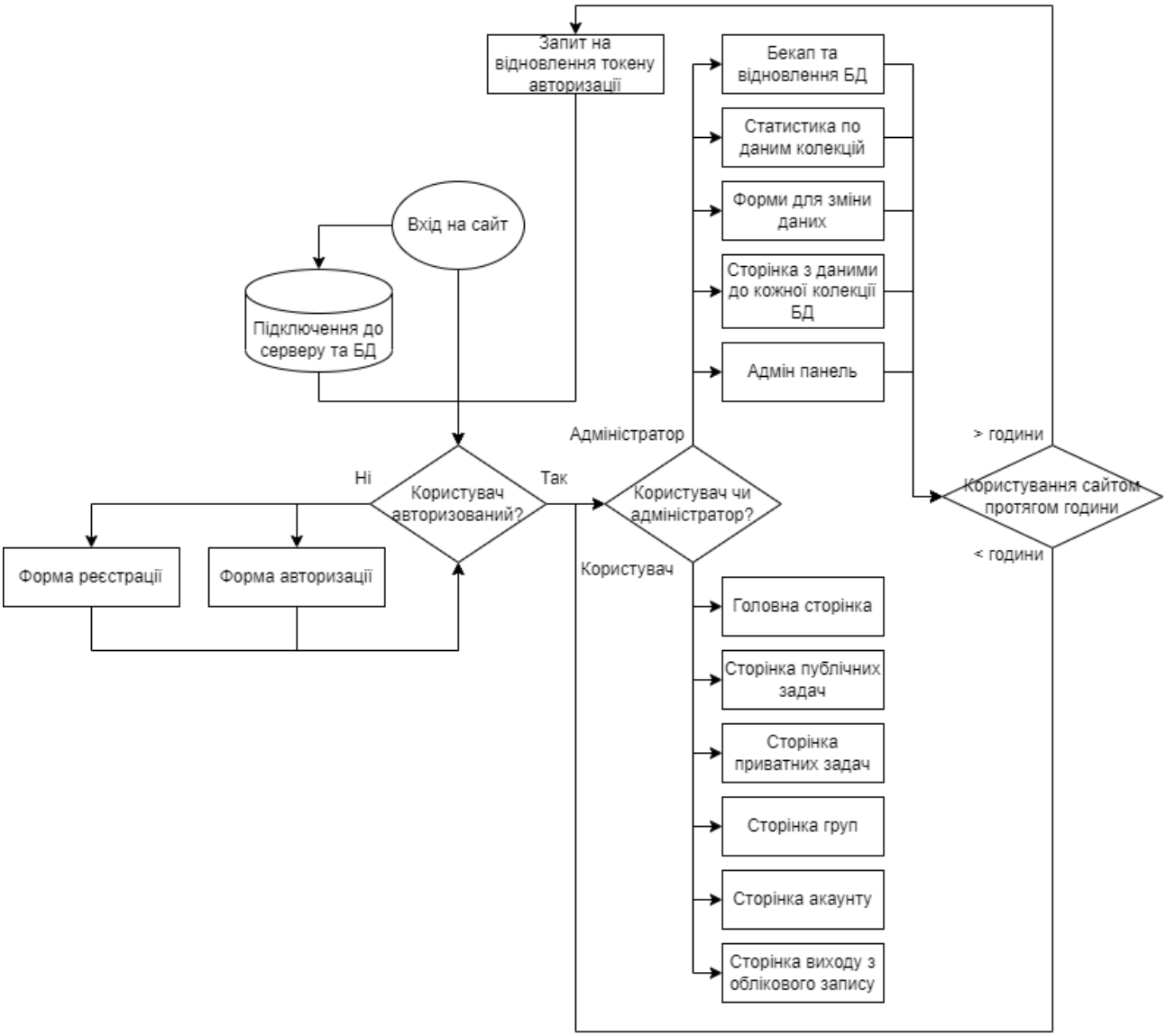


Рис. 2.1. Загальна логічна схема роботи сайту

Додатково буде наведено детальний алгоритм роботи авторизації сайту на наступному рисунку 2.2.:

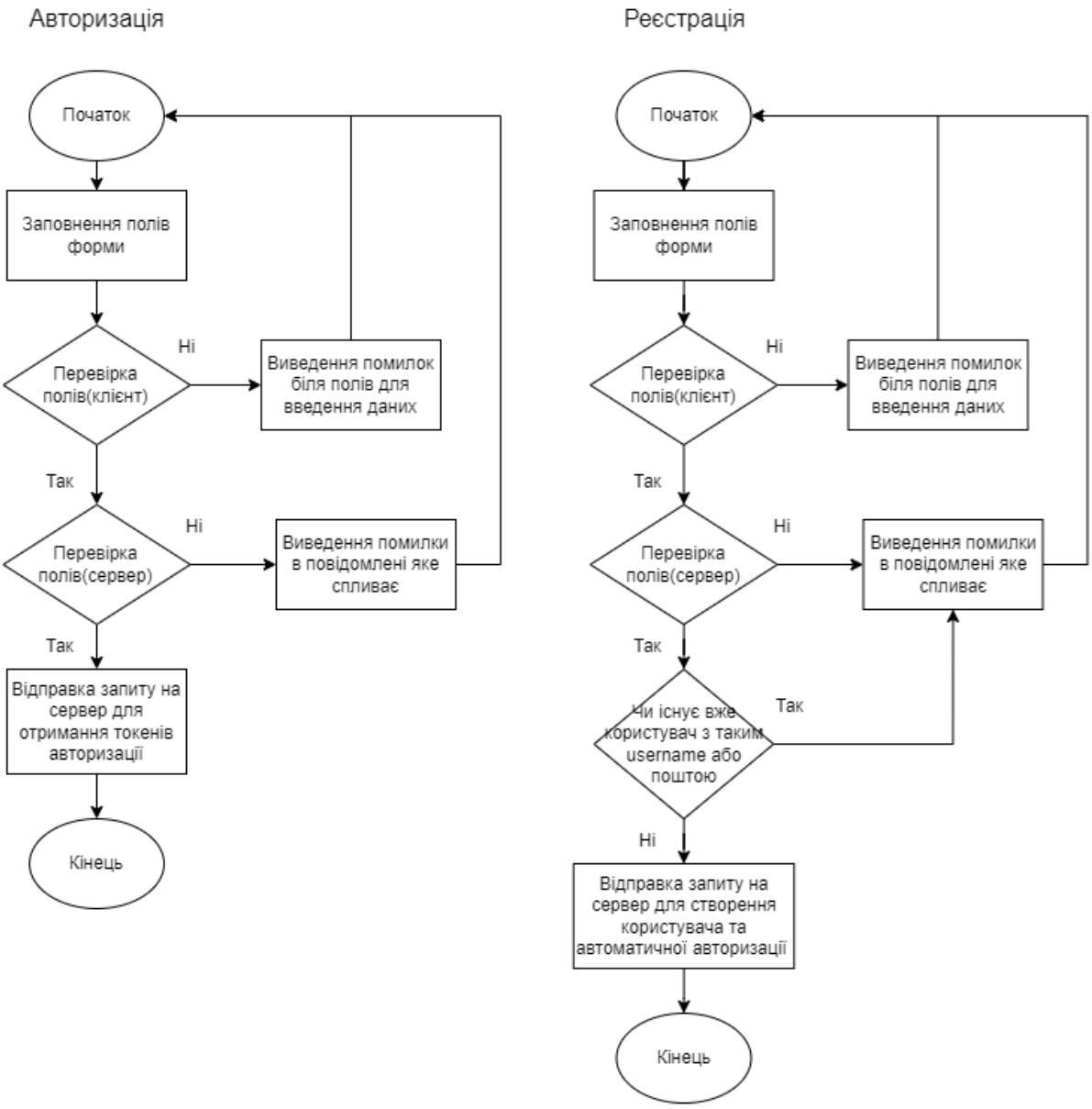


Рис. 2.2. Загальна логічна схема роботи авторизації сайту

За час який було відведено на розробку сайту було виконано лише частину функціоналу у вигляді GUI описаного вище, а саме адмін панель з читанням, редагуванням, видаленням задач, а також можливістю перегляду задачі окремо. Було додано сторінку з деякою статистикою по користувачам та задачам.

2.2 Проектування структури бази даних за напрямком курсової роботи

Важливою частиною роботи над курсовим проектом була база даних та її налаштування. Далі буде наведено рисунок зв’язків таблиць, у випадку MongoDB – колекцій у вигляді невеликої діаграми на рисунку 2.3.:

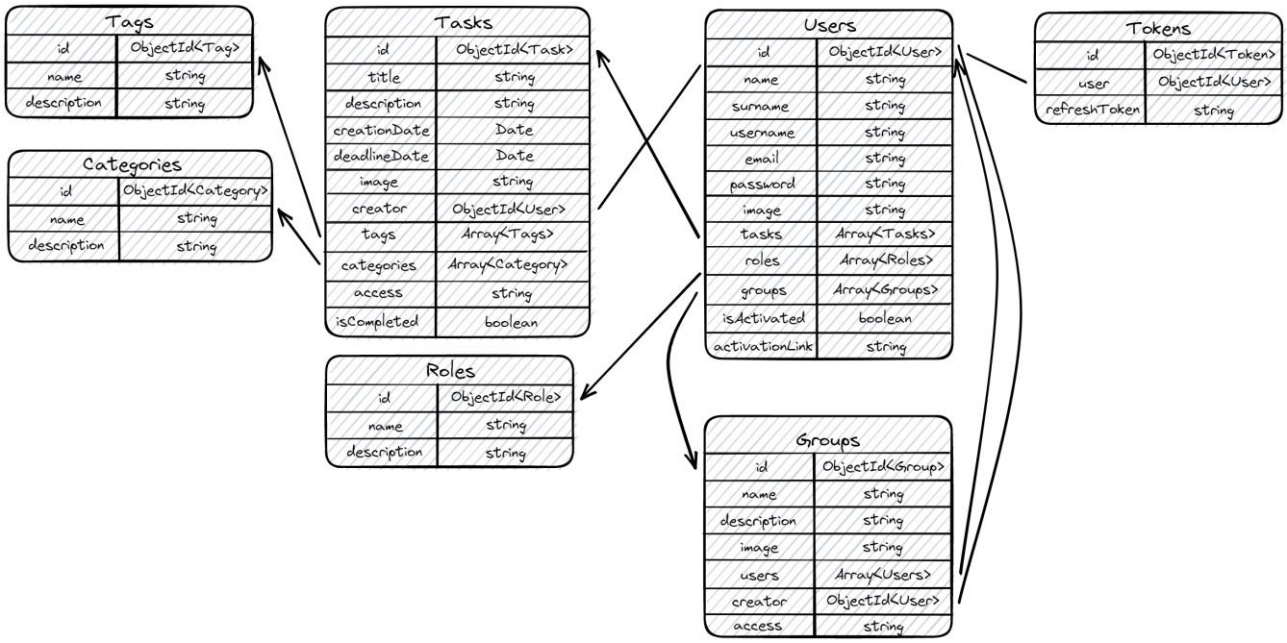


Рис. 2.3. Діаграма зв’язків між колекціями у MongoDB

Також рисунок всіх створених колекцій бази даних MongoDB до курсового проекту:

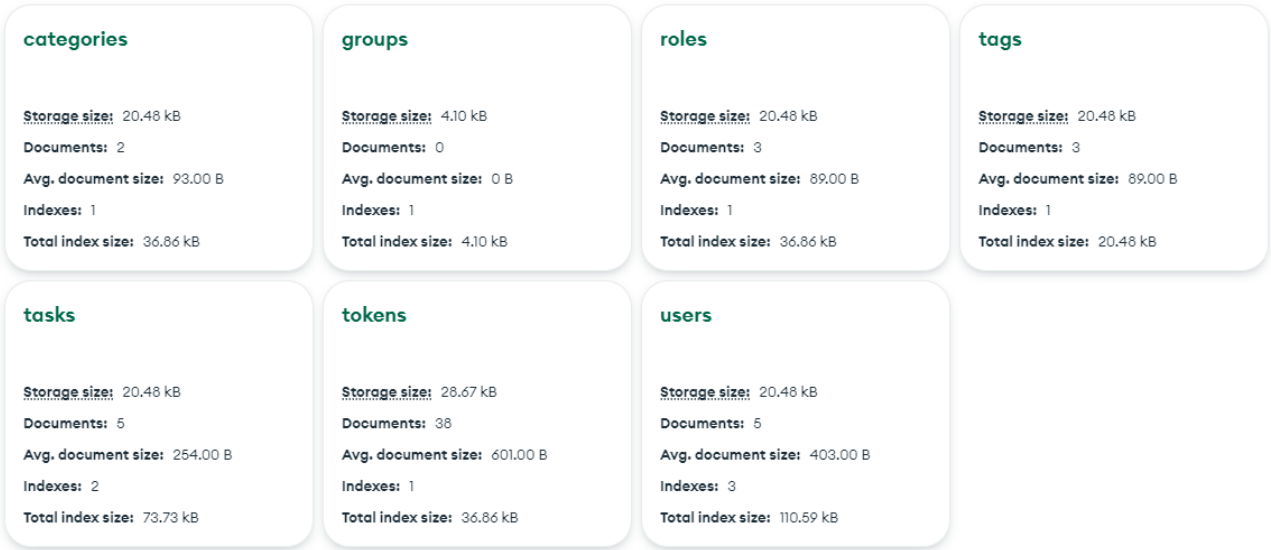


Рис. 2.4. Список колекцій MongoDB курсового проекту

У даній курсовій роботі реалізуються малі за об’ємом та швидкі для виконання методи зчитування, оновлення, видалення та запису даних з БД в окремому класі. Це дає змогу програмі швидко реагувати на зміни даних, їх оновлювати та використовувати для відображення на клієнтській стороні програми.

Висновки з другого розділу

У другому розділі було спроектовано та розроблено структуру програми за допомогою діаграм, які демонструють роботу БД, структуру самої програми, її основні можливості, які необхідні для коректної роботи програми за темою курсового проекту.

РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ

3.1 Опис роботи з програмним додатком

При першому заході на сайт користувач бачить форму для авторизації, якщо він до цього не був ні разу на сайті. На цій же сторінці можна перейти на форму реєстрації, якщо акаунт користувач ще не має. Всі дані, які були введені в формах перевіряються на правильність як на клієнті так і на сервері. Вигляд сторінки авторизації зображено на рисунку 3.1.

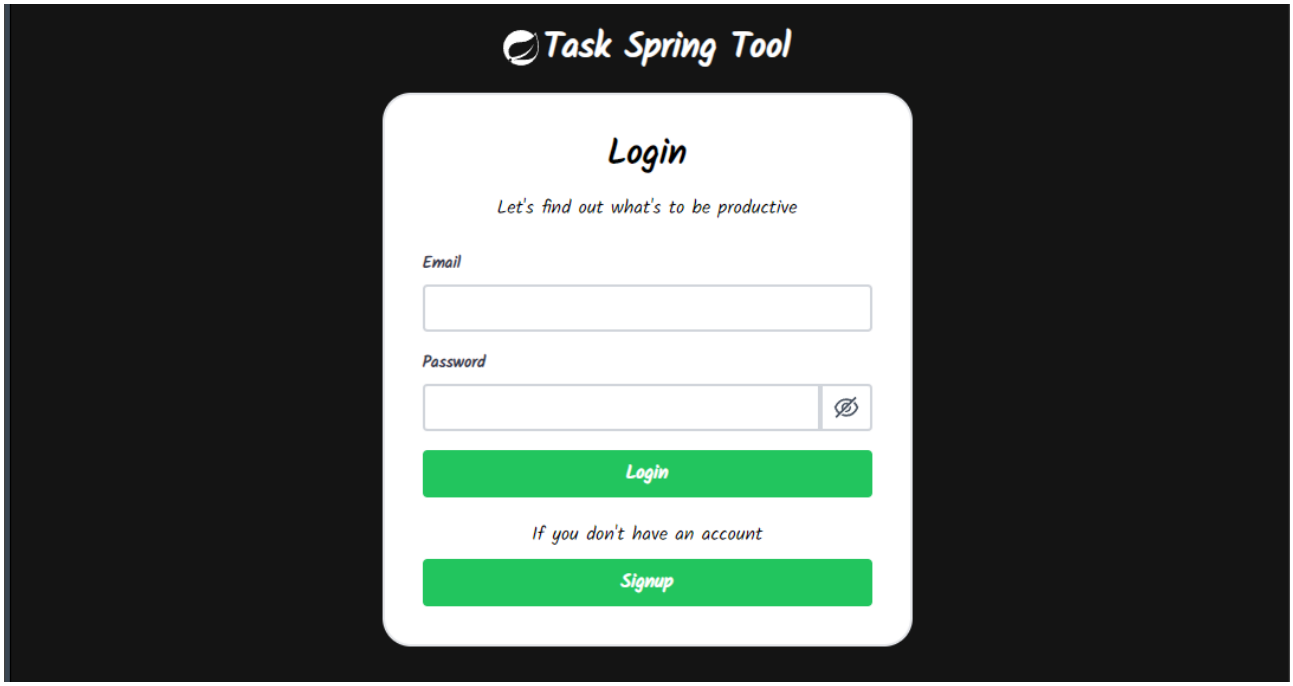


Рис. 3.1. Вигляд сторінки авторизації користувача

Також рисунок на якому зображена сторінка реєстрації користувача:

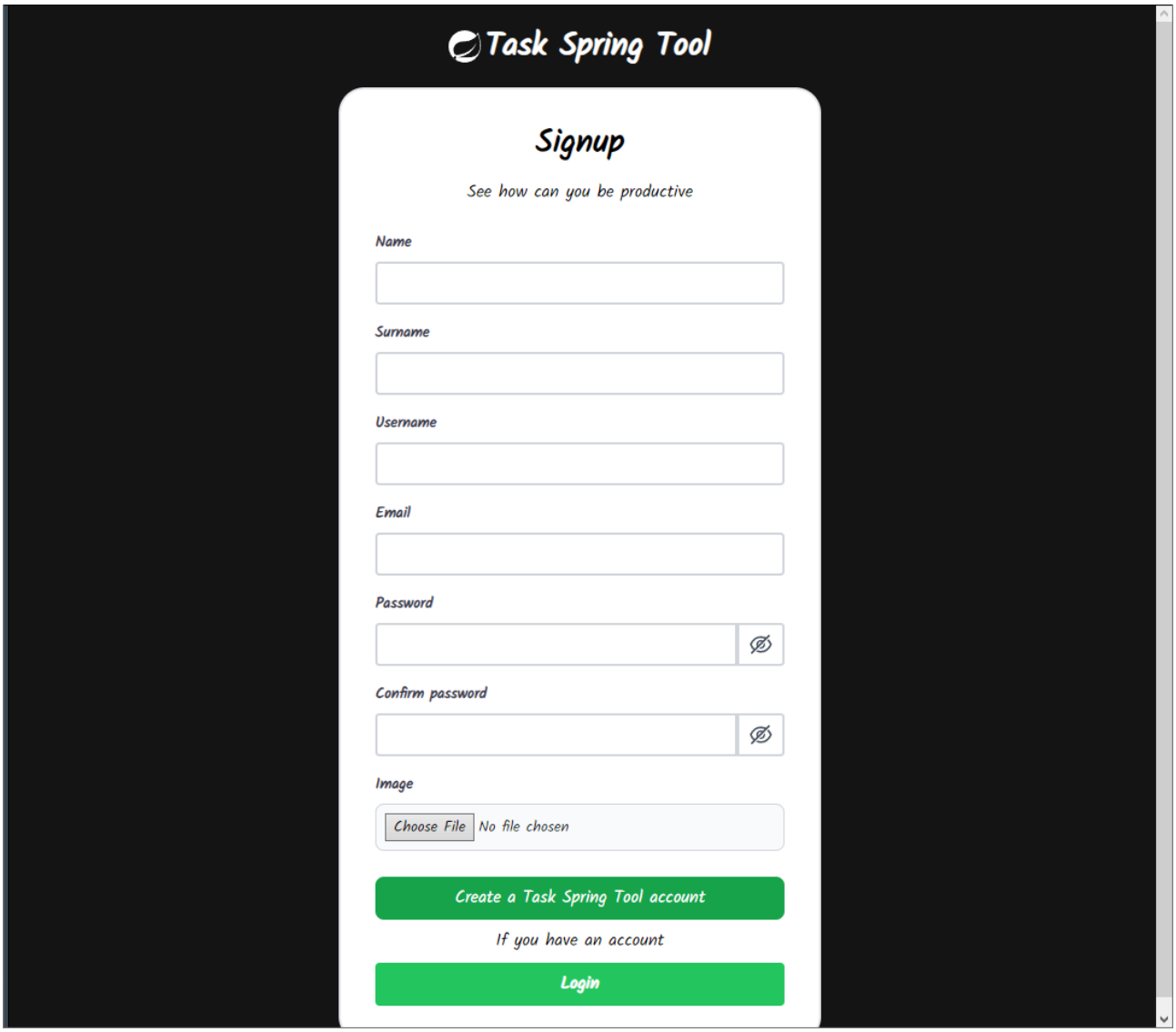


Рис. 3.2. Вигляд сторінки реєстрації користувача

Далі буде наведено приклад реєстрації користувача, дані якого не пройшли перевірку на стороні клієнта:

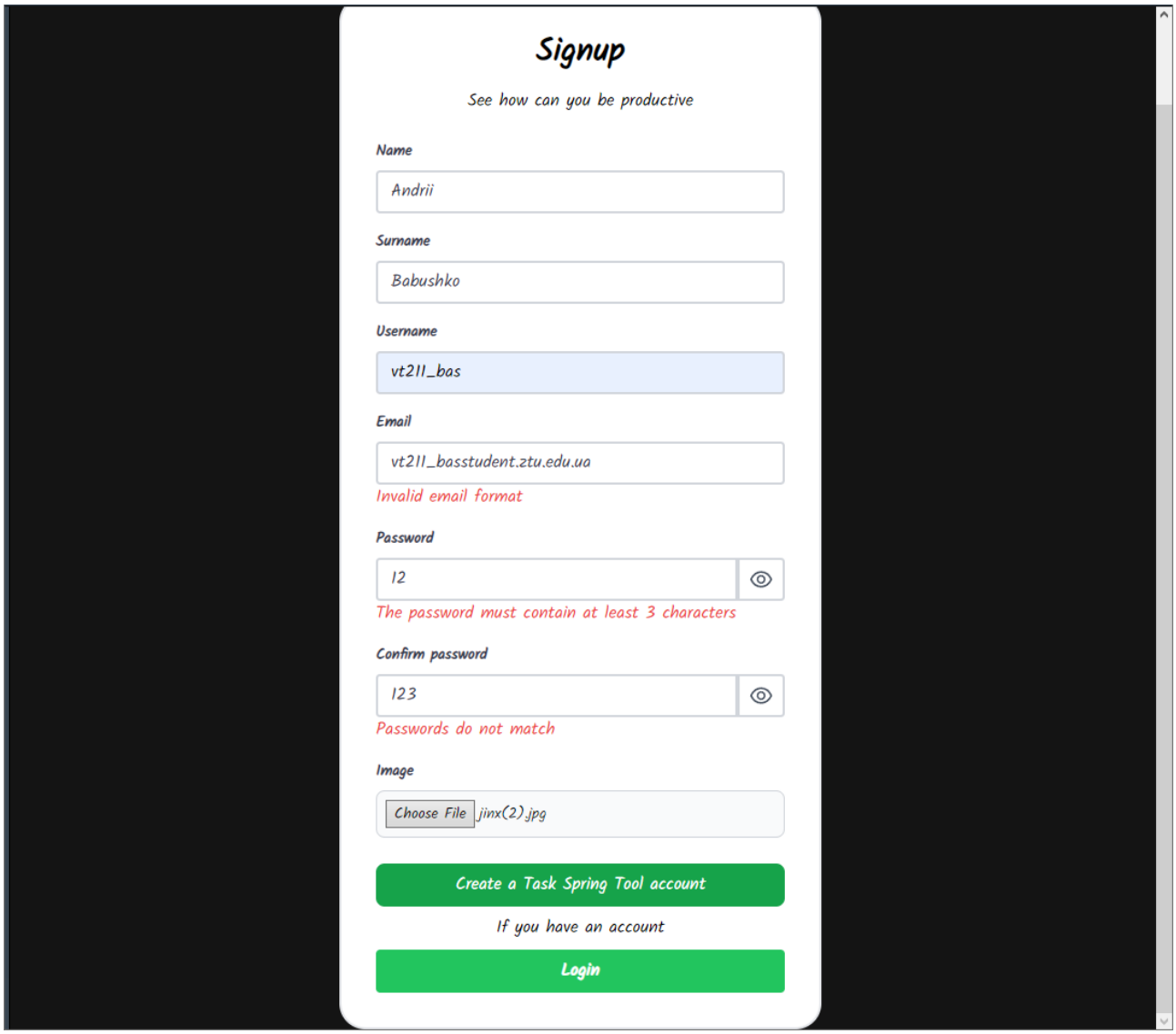


Рис. 3.3. Сторінка реєстрації користувача яка не пройшла валідацію на стороні клієнта

Також продемонструємо, що валідація працює і на стороні сервера на наступному рисунку:

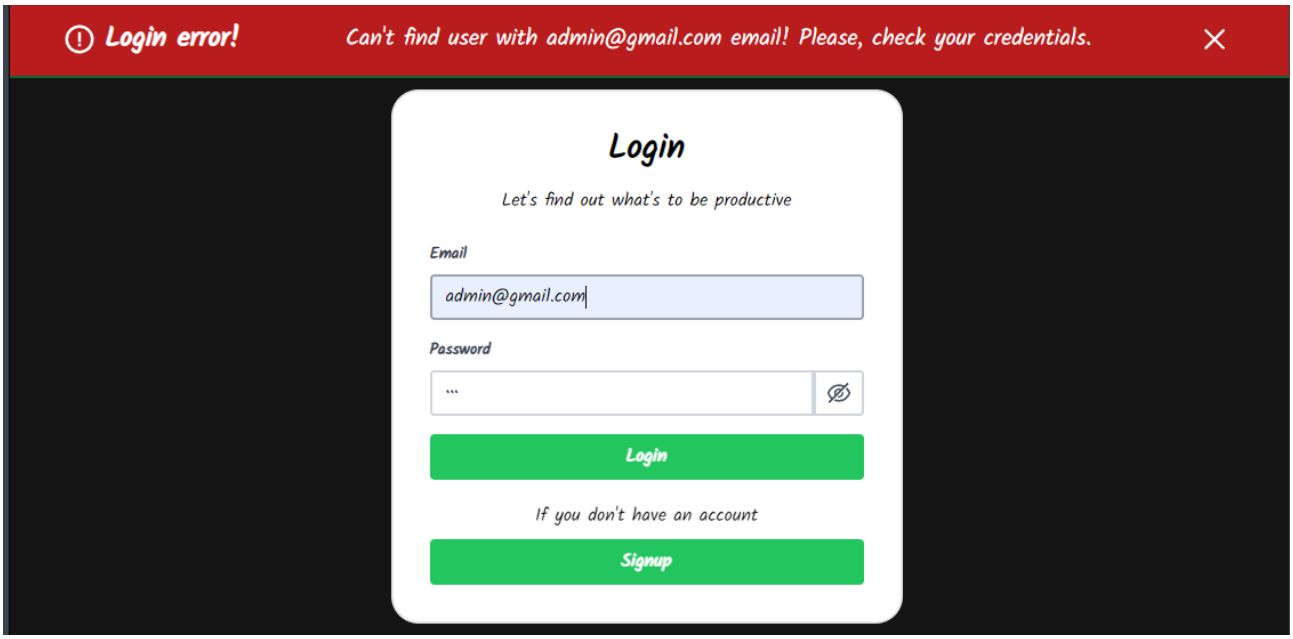


Рис. 3.4. Сторінка реєстрації користувача яка не пройшла валідацію на стороні сервера

Дане повідомлення, яке зображено на рисунку, є майже на всіх сторінках, де проводяться запити та треба щоб користувач знав чому саме він не зміг авторизуватися, зареєструватися, створити задачу, тощо. Також в функціонал повідомлення закладено те, що воно з'являється на певний період часу та після цього часу автоматично зникає. Користувач може самостійно його закрити натиснувши хрестик в правому куті блоку повідомлення, але в цьому не має потреби.

Після успішної реєстрації одразу проходить авторизація і відображається головна сторінка сайту разом з посиланнями на інші сторінки. Але для того щоб вона відобразилась треба щоб користувач активував свою пошту знайшовши відповідне повідомлення в поштовій скринці. Перейшовши за посиланням відкривається головна сторінка сайту. В залежності від того користувач це чи адміністратор, відображається відповідні посиланні на ті чи інші сторінки сайту та блокується доступ до деяких з них. Далі на рисунку зображено вигляд сторінки після реєстрації користувача з неактивованою поштою:

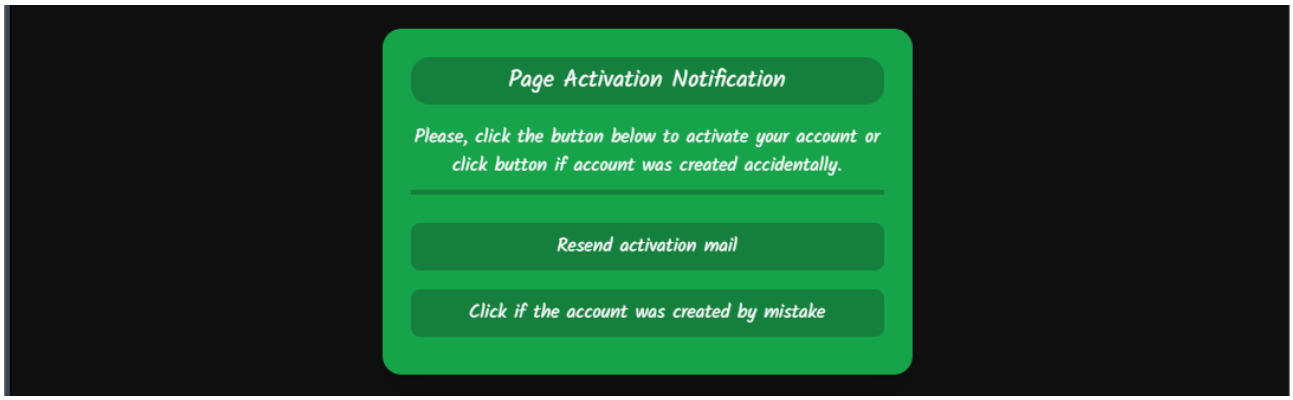


Рис. 3.5. Вигляд сторінки авторизованого користувача з неактивованою поштою

Також буде наведено рисунок листа який прийшов на пошту для активації акаунту:

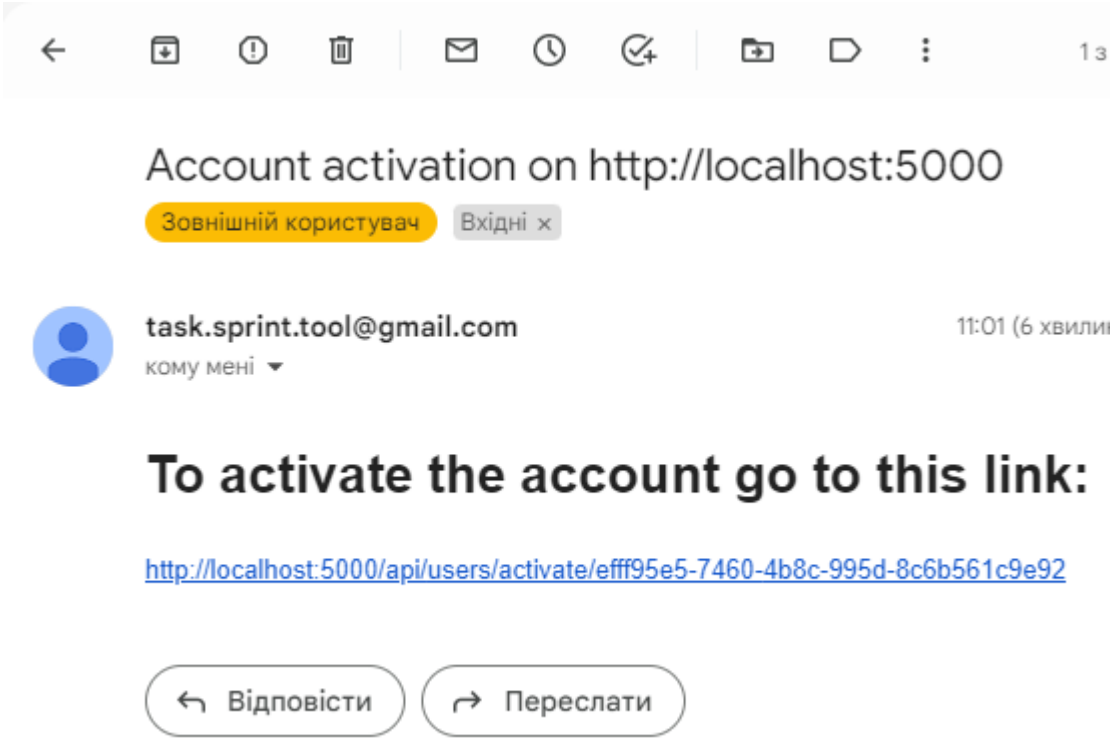


Рис. 3.6. Лист на пошту для активації акаунту

Якщо авторизувався користувач, сторінка буде мати наступний вигляд:

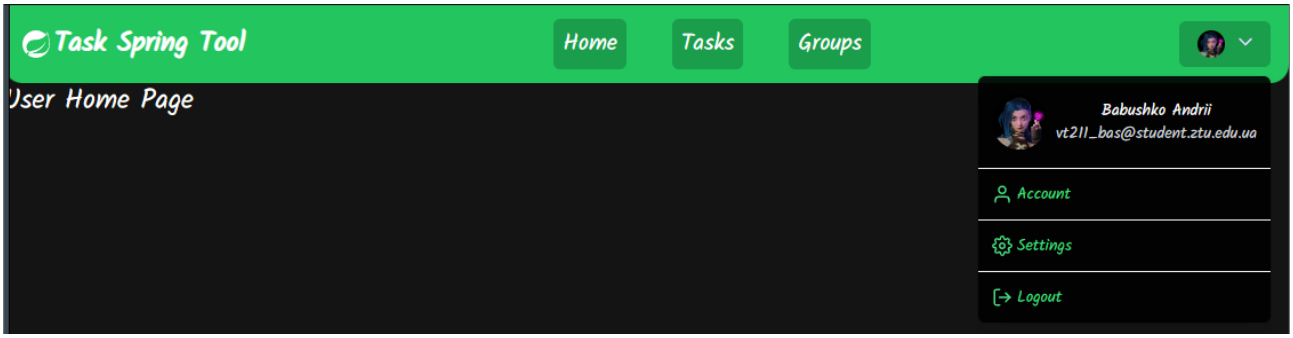


Рис. 3.7. Головна сторінка після реєстрації користувача

Кожна сторінка має хедер та футер, звідки можна перейти на інші сторінки сайту. Хедер користувача має посилання на головну сторінку сайту, де зазначена додаткова інформація про сайт, сторінка задач користувача, стрічка задач інших користувачів, групи в яких є користувач та інше. Сайт завантажується на англійській мові в темі з різними відтінками зеленого.

Спробуємо вийти зі створеного акаунту та зайди під адміністратором. Вигляд який буде мати сторінка при спробі вийти з акаунту зображена на рисунку 3.6.:

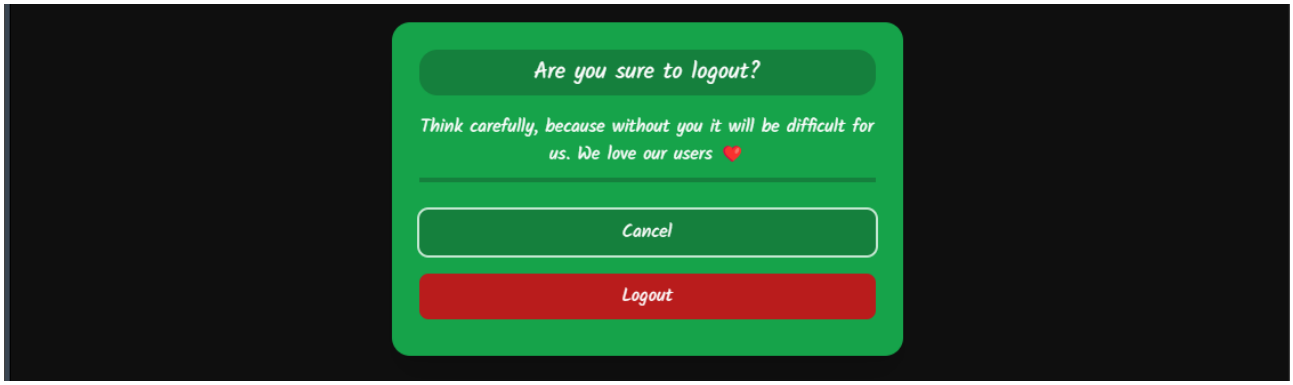


Рис. 3.8. Спроба вийти з акаунту

На даній сторінці можна повернутися на минулу сторінку натиснувши “Cancel” або вийти з акаунту натиснувши “Logout”. Після виходу з акаунту завантажиться форма авторизації. Будь-яка спроба повернутися на попередні сторінки до результату не приведе. Сторінка просто не завантажиться. Користувач зможе відвідувати лише ті сторінки, до який має доступ, який обмежено програмно.

Як було написано вище, далі буде продемонстровано вхід на сайт за допомогою акаунта, у якого є права адміністратора:

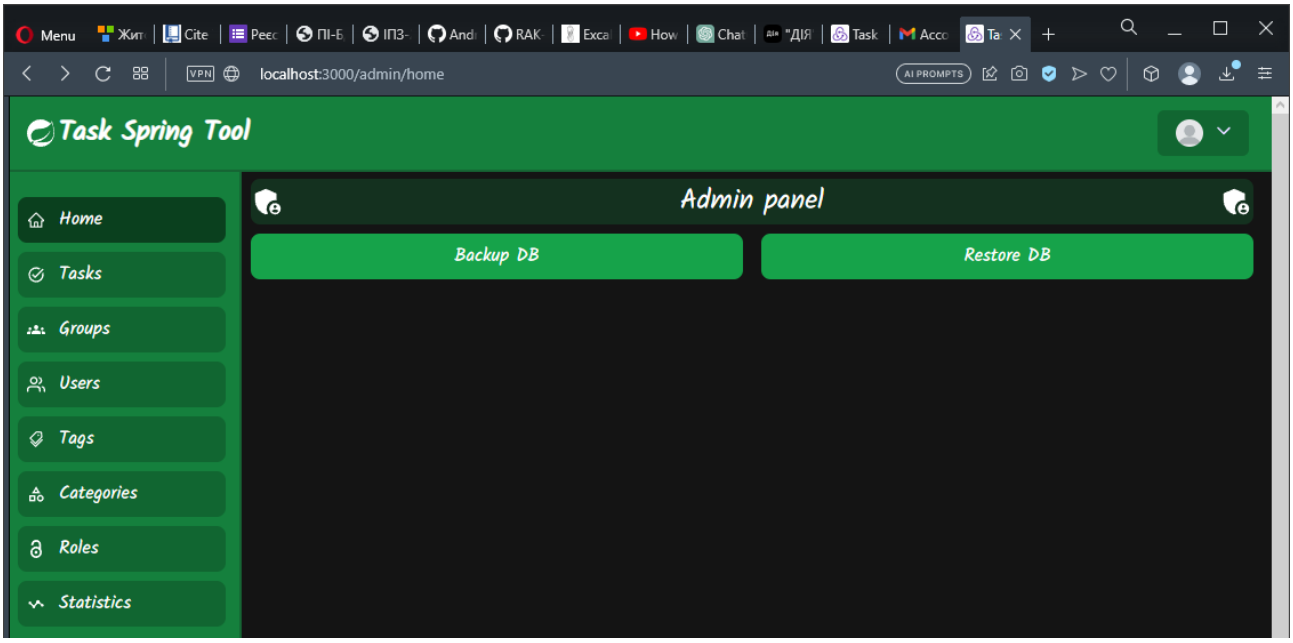


Рис. 3.9. Вигляд сайту після входу адміністратора

Повідомлення яке з’являється після успішної авторизації зображено на наступному рисунку:

Рис. 3.10. Вигляд повідомлення після успішного входу на сайт

Після завантаження сторінки адміністратора, користувач має доступ до всіх записів БД, які можна переглянути відвідавши відповідні сторінки з лівого бокового меню.

Головна сторінка має на меті розповісти про функціонал який має дана адмін панель, а також 2 кнопки, які роблять запит на сервер для збереження копії БД та відновлення її відповідно за допомогою створеної копії. Після натискання кнопки збереження копії БД(“Backup DB”) сторінка матиме вигляд:

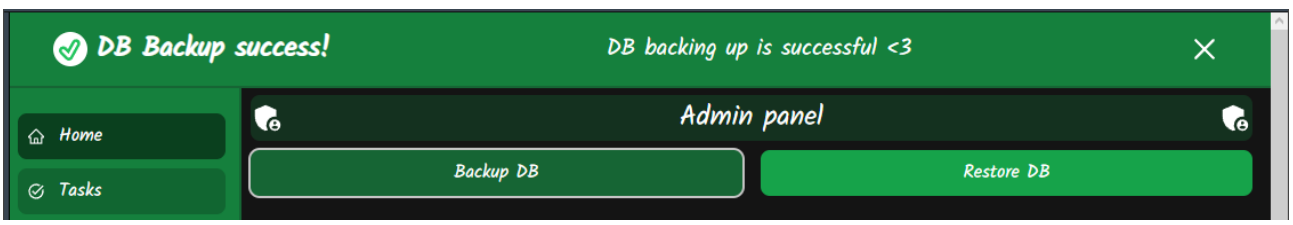


Рис. 3.11. Вигляд головної сторінки після збереження копії БД на сервері

Будь-який запит, який робиться в адмін панелі супроводжується відповідним повідомленням про успішність або невдачу запиту. Тому надалі їх демонстрації проводитись не буде.

Продемонструємо наступну сторінку. Це буде сторінка з задачами(“Tasks”). Вона зображена на рисунку 3.12 та 3.13.

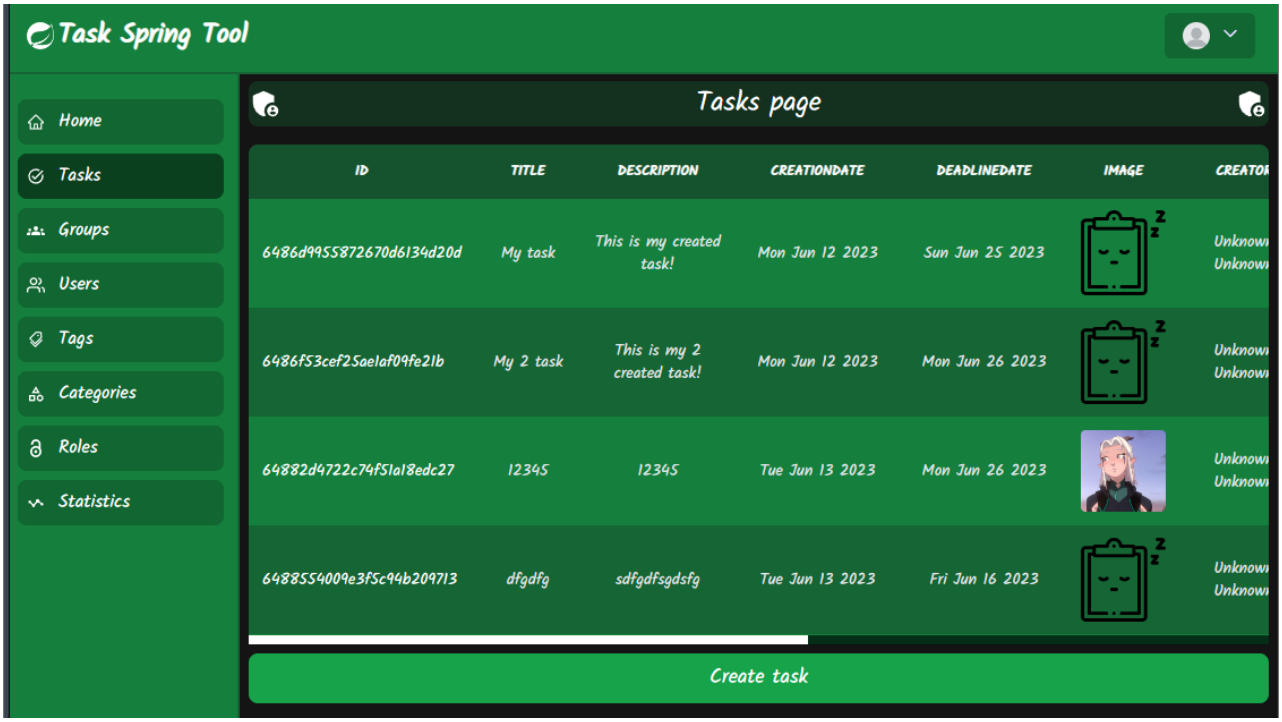


Рис. 3.12. Вигляд першої частини сторінки задач(Tasks)

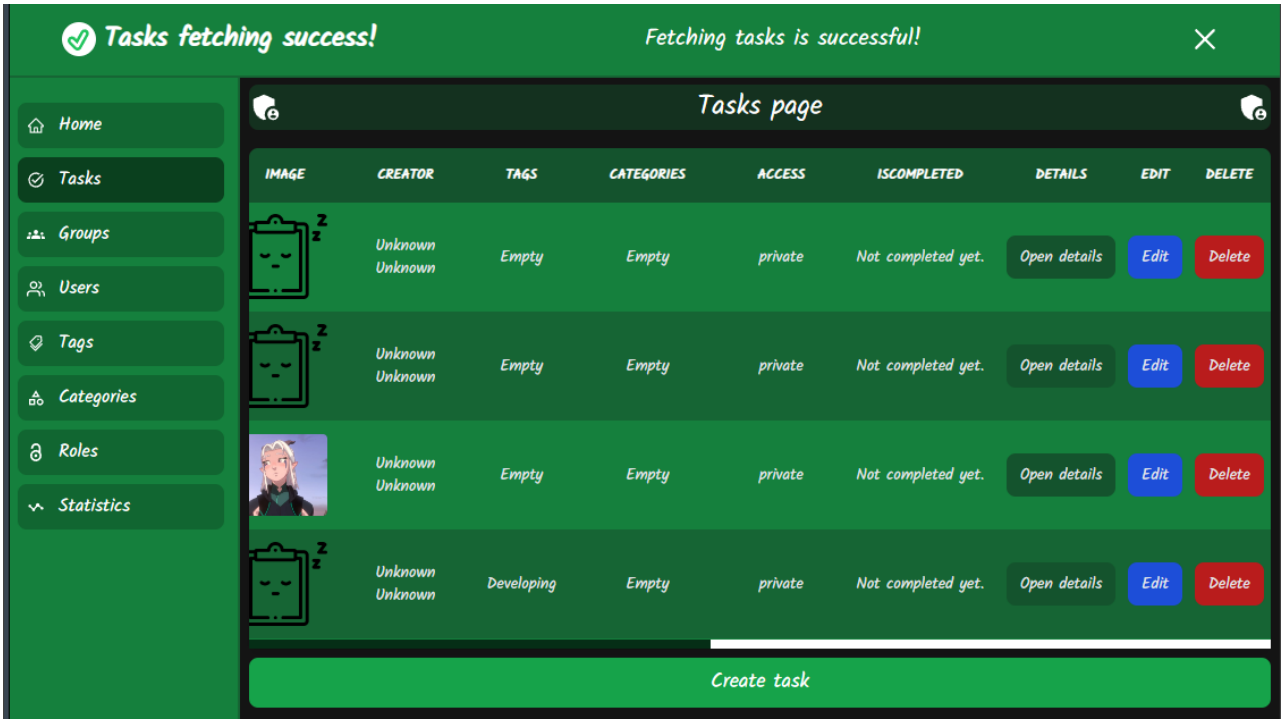


Рис. 3.13. Вигляд другої частини сторінки задач(Tasks)

Нагадаємо, що всі сторінки які демонструвалися після авторизації адміністратора стосуються лише адміністраторів.

Тут адміністратор має змогу переглянути всі записи, які існують в базі даних, створити новий, редагувати чи видалити існуючий, а також переглянути окремо інформацію про конкретну задачу. Далі буде продемонстровано вигляд кожної з цих сторінок та результат опрацювання даних.

Сторінка створення задачі буде мати наступний вигляд:

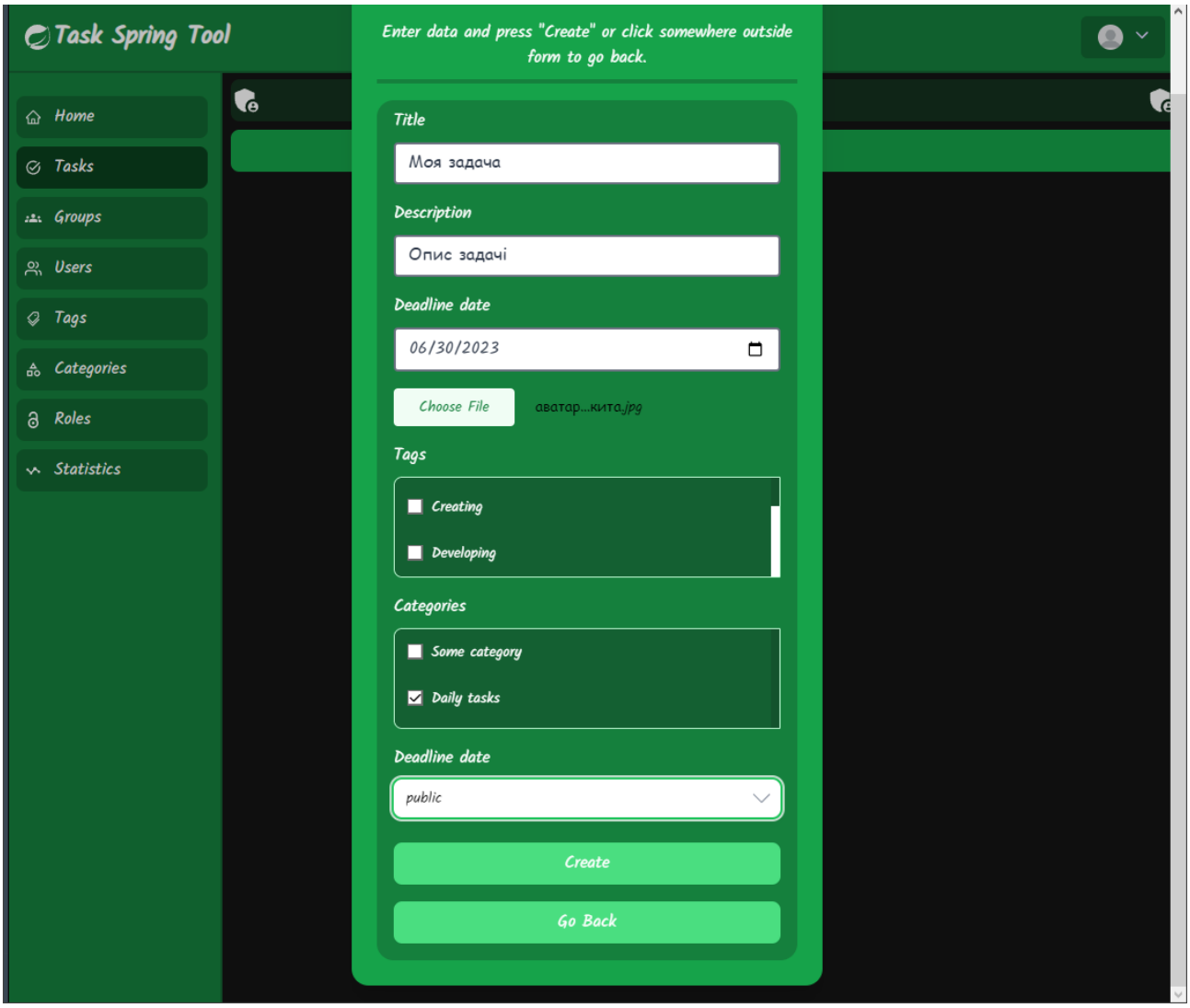


Рис. 3.14. Вигляд сторінки для створення нової задачі

Сторінка задач після створення нової буде мати наступний вигляд:

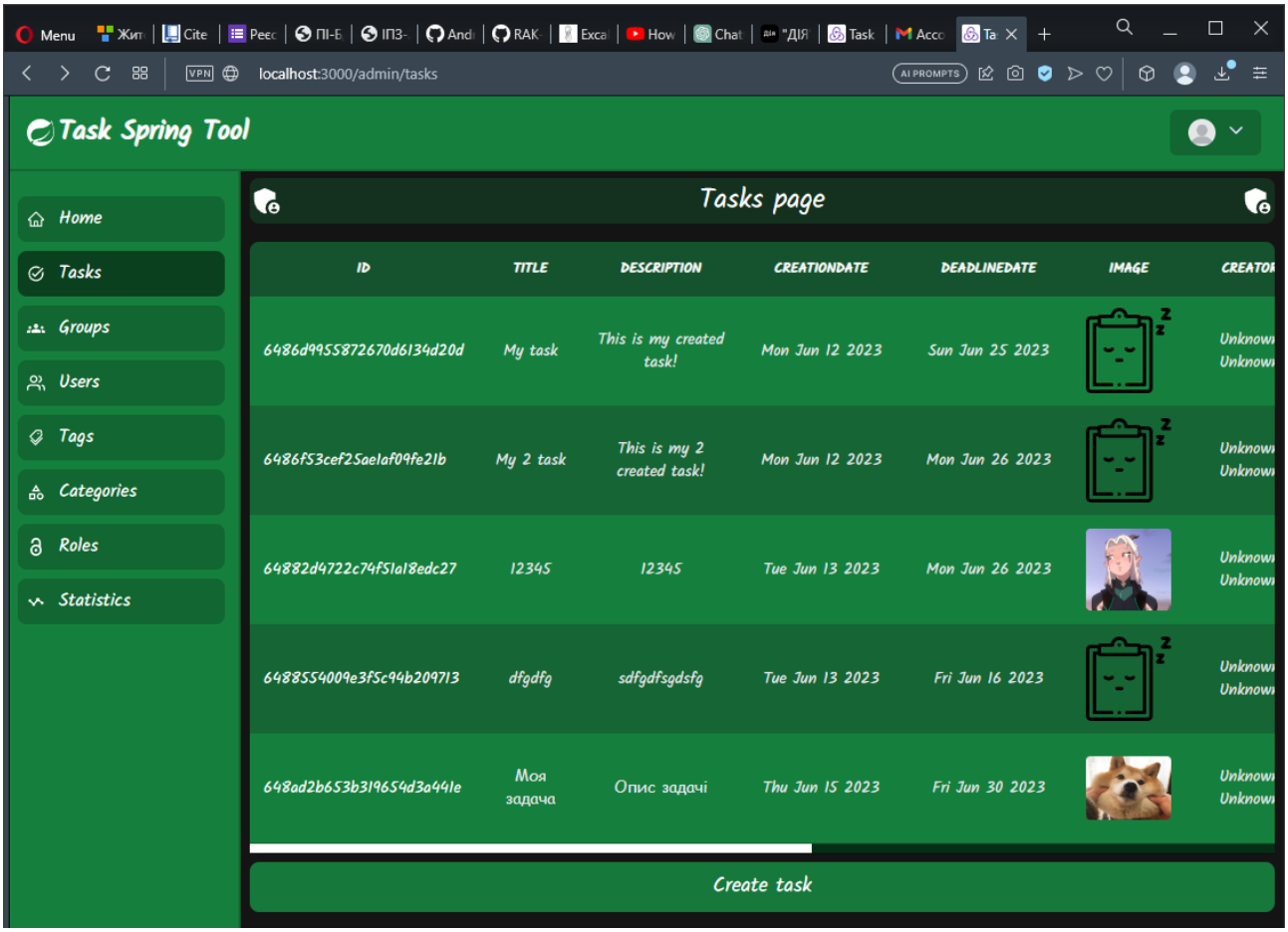


Рис. 3.15. Вигляд сторінки задач після створення нової задачі

Вигляд сторінки редагування задачі:

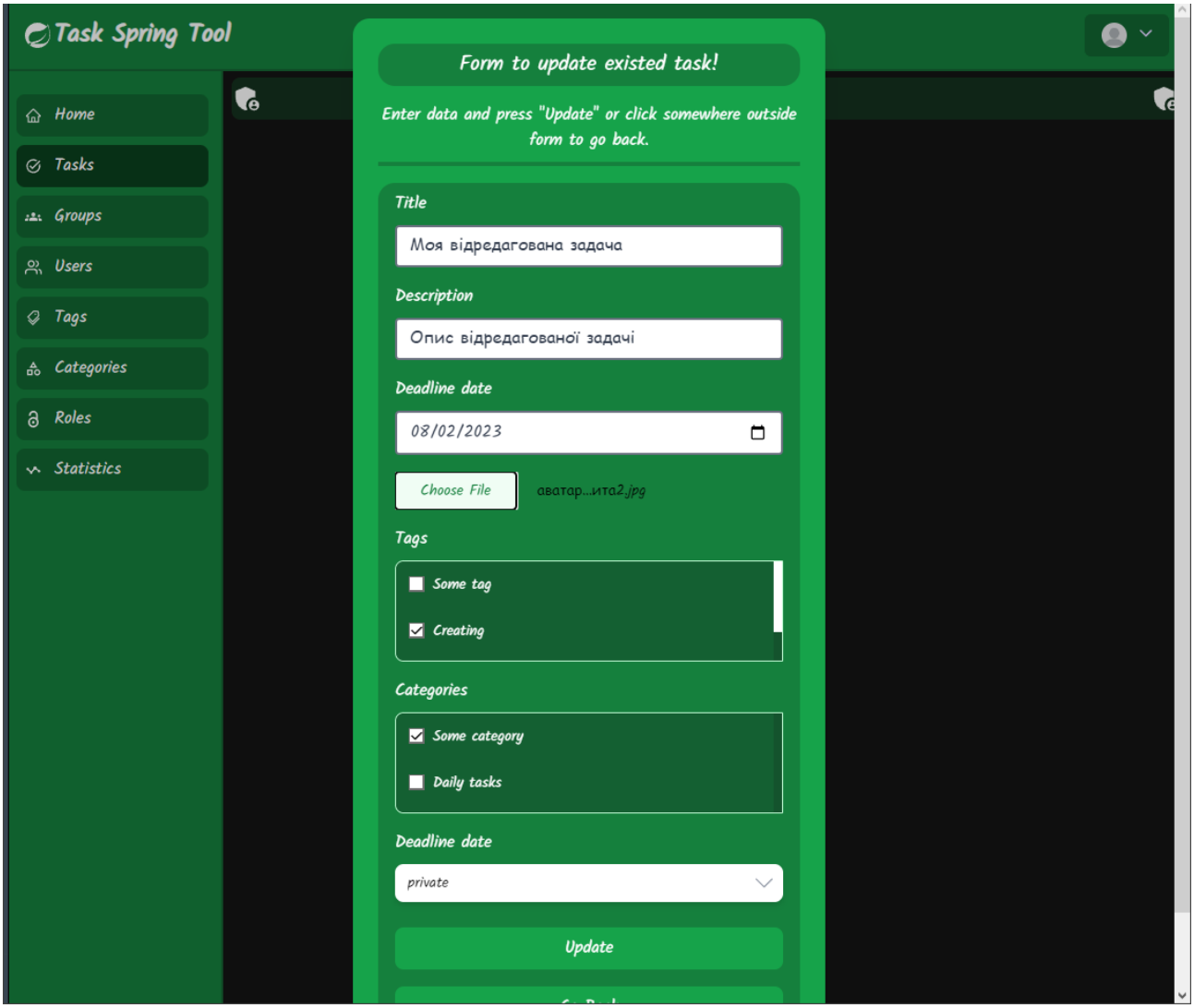


Рис. 3.16. Вигляд сторінки для редагування задачі

Сторінка задач після редагування нової створеної задачі:

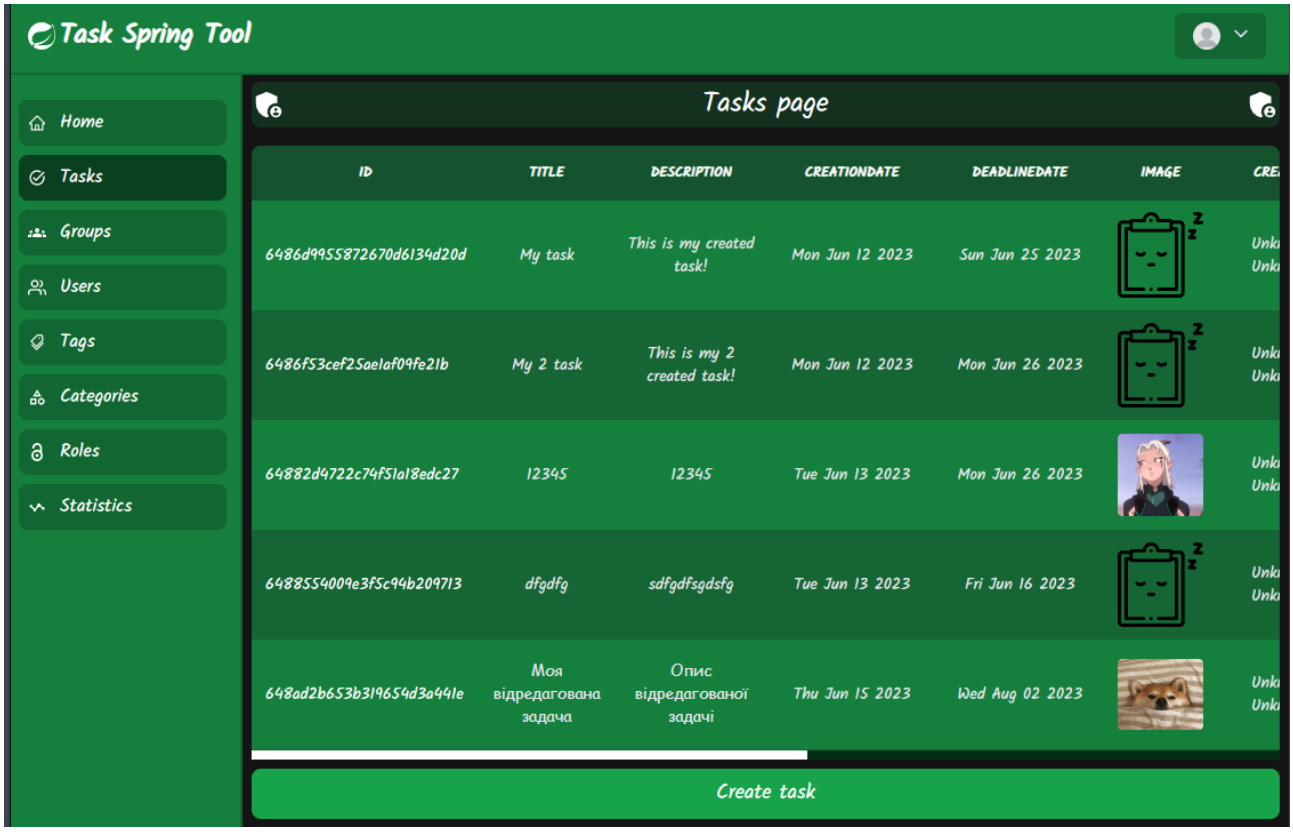


Рис. 3.17. Вигляд сторінки задач після редагування нової задачі

Вигляд окремої сторінки під створену задачу:

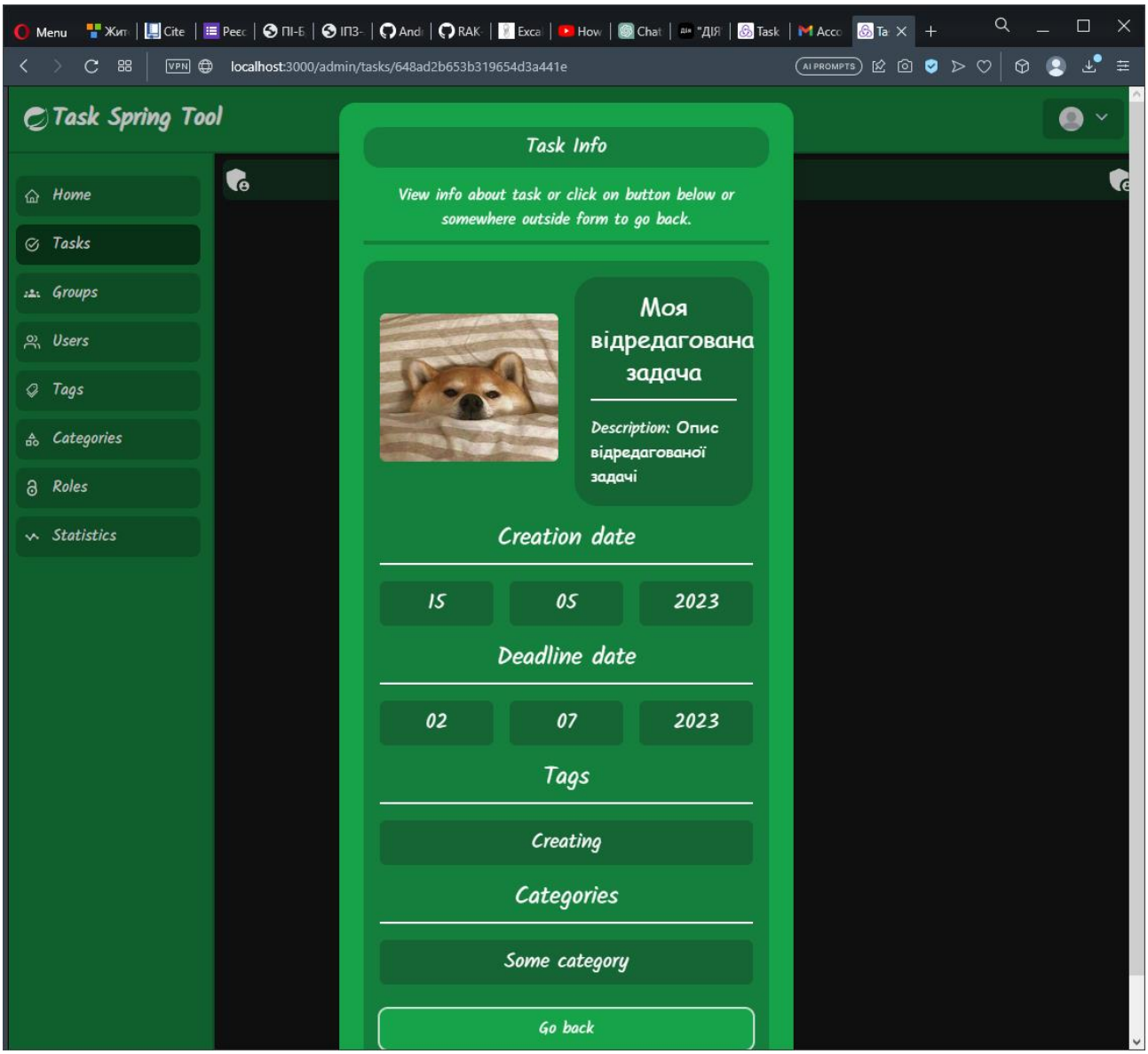


Рис. 3.18. Вигляд сторінки для окремої задачі

Вигляд сторінки видалення задачі:

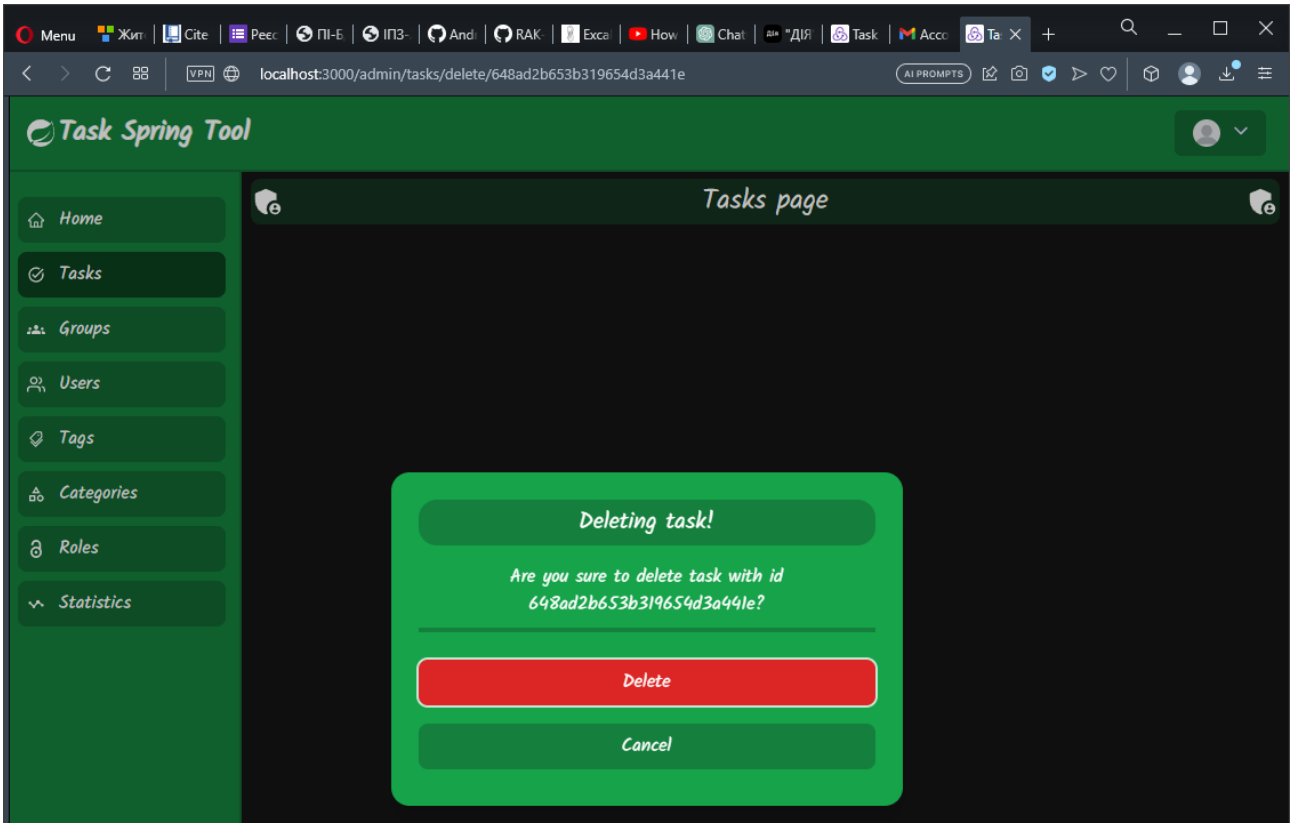


Рис. 3.18. Вигляд сторінки для видалення задачі

Результат видалення нової створеної задачі:

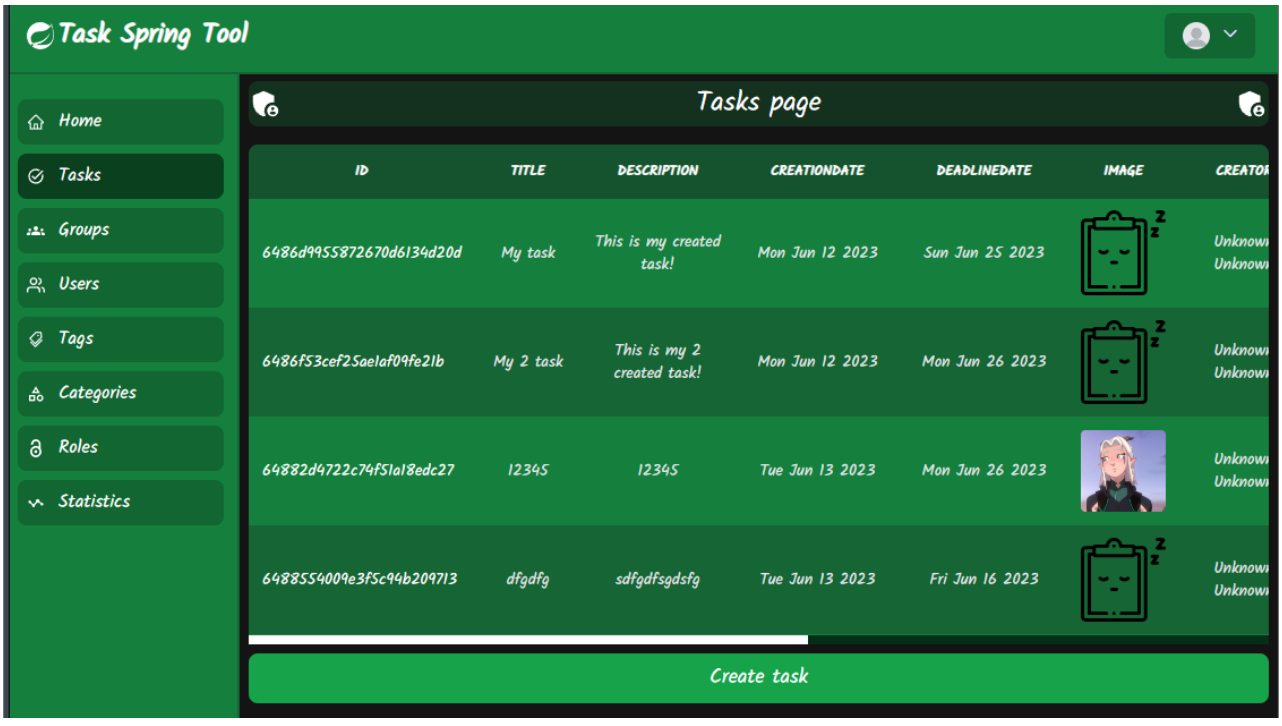


Рис. 3.19. Вигляд сторінки задач після видалення нової задачі

Такі сторінки, як перегляд, створення, редагування та видалення виглядаю майже так само і на інших сторінках для відповідних колекцій, які було описано в минулому розділі, тому демонстрації їх вигляду та роботи не буде.

На останок, це сторінка статистики. Наразі є лише деяка статистика колекцій з користувачами та задачами. На наступних рисунках 3.20, 3.21 та 3.22 продемонстровано вигляд сторінки статистики користувачів та задач:

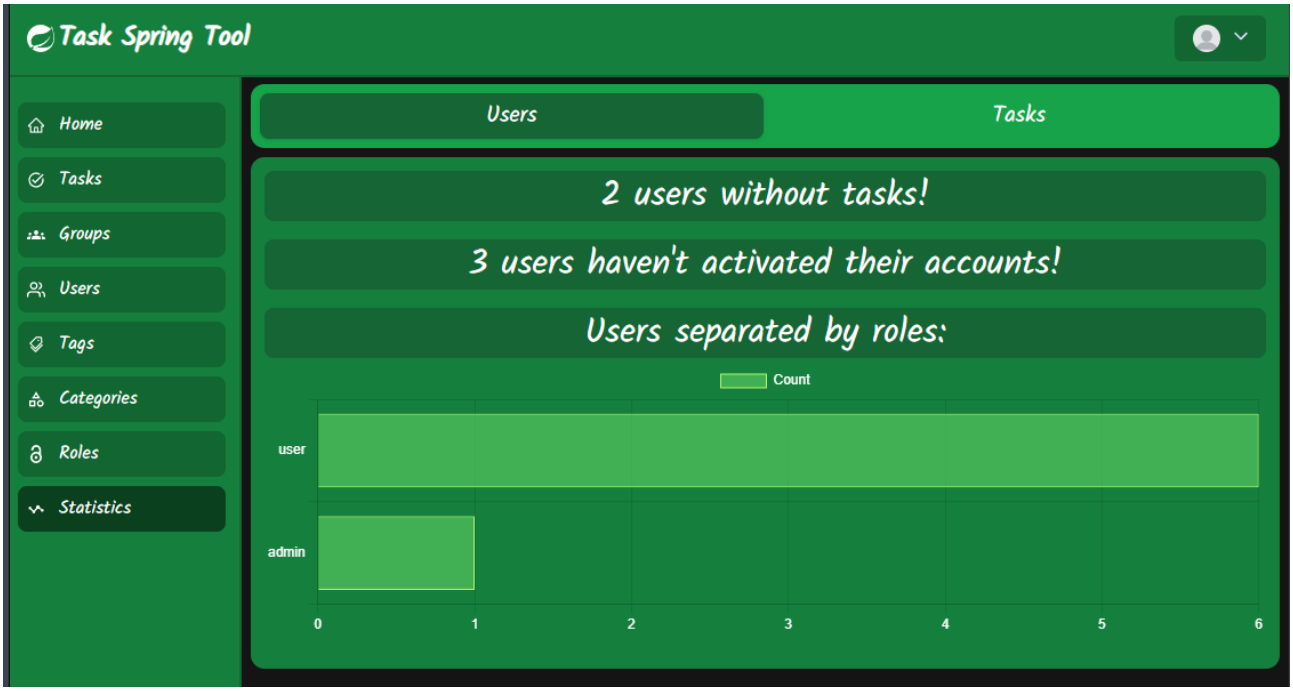


Рис. 3.20. Вигляд сторінки статистики користувачів

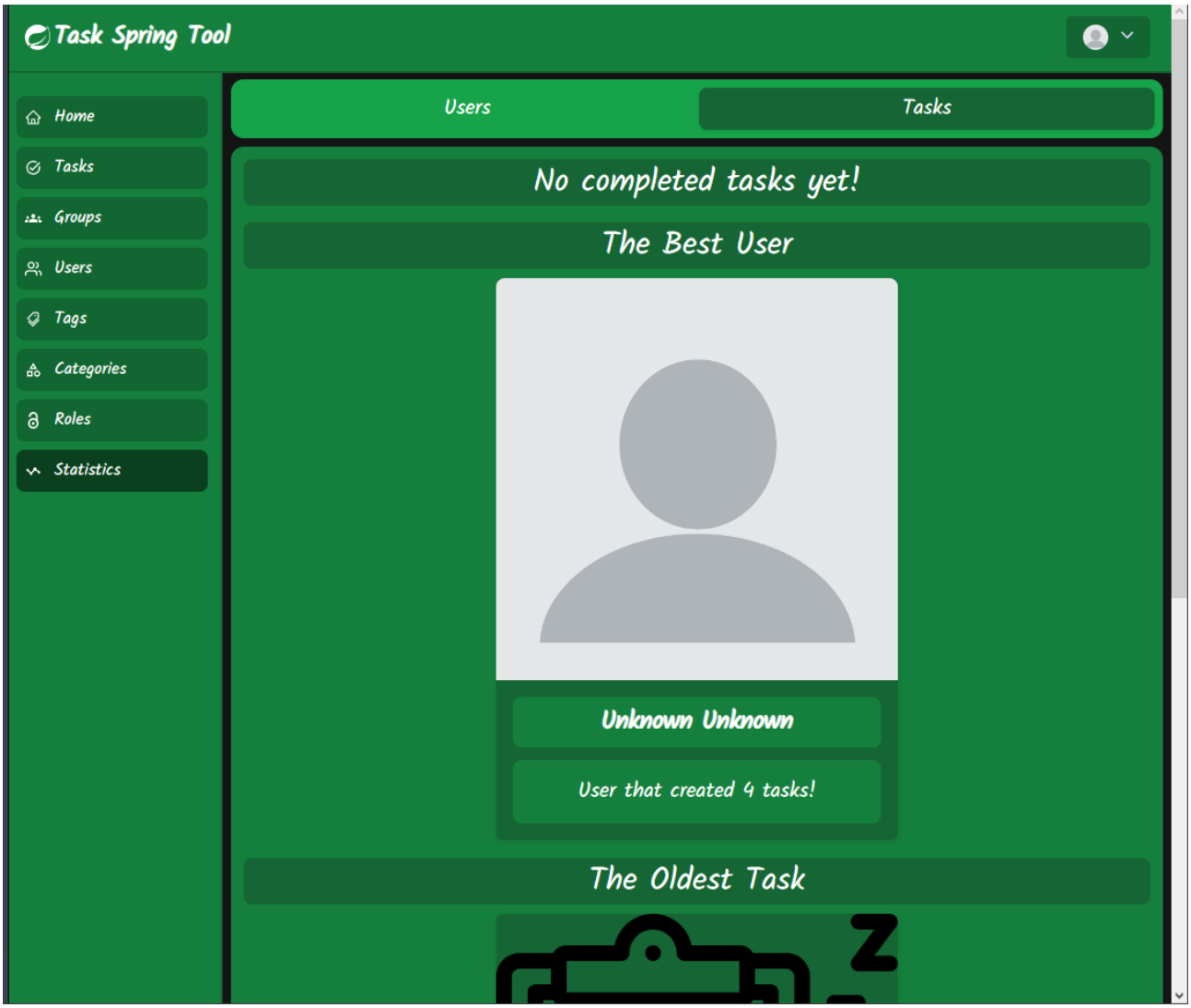


Рис. 3.21. Вигляд першої частини сторінки статистики задач

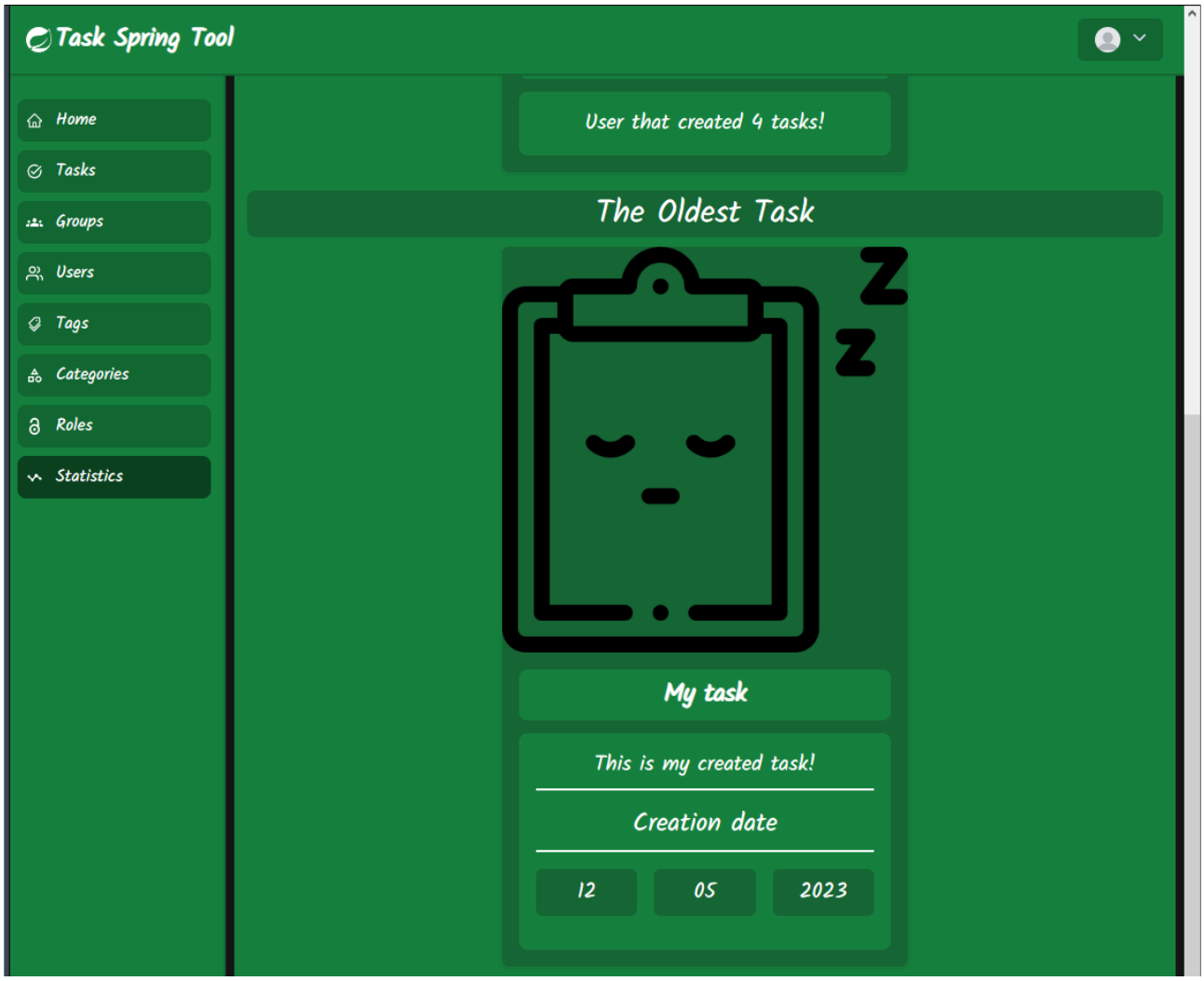


Рис. 3.22. Вигляд другої частини сторінки статистики задач

3.2 Реалізація операцій обробки даний в БД за напрямком курсової роботи

Після проектування та визначення логіки роботи застосунку необхідно розпочати саму розробку. У цьому розділі будуть розглянутий функціонал що забезпечує стабільну роботу сайту.

Для початку розглянемо основу бази даних додатку – моделі, за допомогою яких виконуються всі операції над даними. Для прикладу розглянемо модель задач:

```
import { model, Schema, Types } from 'mongoose';
import { ITaskModel } from '../ts/interfaces/ITaskModel.js';

const taskSchema = new Schema<ITaskModel>({
  title: { type: String, required: true },
  description: {
    type: String,
    required: false,
    default: 'No description yet...',
  },
  creationDate: { type: Date, required: true },
  deadlineDate: { type: Date, required: false, default: null, index: {
expires: '0s' } },
  image: { type: String, required: false, default:
'uploads/task/no_task_image.png' },
  creator: { type: Types.ObjectId, required: true, ref: 'User' },
  tags: [{ type: Types.ObjectId, required: false, ref: 'Tag' }],
  categories: [{ type: Types.ObjectId, required: false, ref: 'Category' }],
  access: { type: String, required: true, default: 'private' },
  isCompleted: { type: Boolean, required: true, default: false },
});

const mongooseModel = model<ITaskModel>('Task', taskSchema);

export { mongooseModel };
```

Основний функціонал сайту лежить в основі CRUD операції та інших маніпуляцій над даними, тому розглянемо саме його реалізацію. Далі буде представлено приклад коду, точніше методу контролера задач який обробляє запит надісланий за певним шляхом до сервера:

```
const createTask = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req.body);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Create task validation error. Please, check
your credentials.', errors.array()));
    }

    const imagePath = req.file?.filename ?
`uploads/task/${req.file.filename}` : undefined;
    const { id: creatorID } = req.userData;
```

```
    const taskData = await TaskService.createTask(creatorID, { ...req.body,
image: imagePath });

    return res.status(200).json({ ...taskData, message: 'Task is
successfully created!' });
  } catch (e) {
    next(e);
  }
};
```

Дана функція приймає дані запиту, обробляє їх в окремому методі сервісу та повертає їх якщо вони пройшли логіку, в даному випадку створення. В іншому випадку, падає помилка, що повертається сервером у форматі JSON.

Наступним йде приклад коду, який реалізує оновлення задачі за рахунок динамічного отримання ідентифікатора, за яким функція сервісу проводить пошук та оновлює ті дані, які надійшли разом з запитом.

```
const updateTaskByID = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Update task validation error. Please, check
your credentials.', errors.array()));
    }

    const taskID = req.params.taskID;
    const imagePath = req.file?.filename ?
`uploads/task/${req.file.filename}` : undefined;
    const { id: userID } = req.userData;

    const taskData = await TaskService.updateTask(taskID, userID, {
...req.body, image: imagePath });

    return res.status(200).json({ ...taskData, message: 'Task is
successfully updated!' });
  } catch (e) {
    next(e);
  }
};
```

Видалення задачі відбувається схожим чином, як оновлення, але функція додатково враховує всі можливі відношення до інших колекцій які є в колекції задач. Також присутнє видалення картинки, але тільки якщо вона не по замовчуванню.

```
const deleteTaskByID = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const taskID: string = req.params.taskID;
    const { id: userID } = req.userData;
```

```
    const taskData = await TaskService.deleteTask(taskID, userID);

    return res.status(200).json({
      ...taskData,
      message: `Successful deleted task with ${taskID} ID.`,
    });
  } catch (e) {
    next(e);
  }
};
```

Далі розглянемо функції отримання даних. Однією з них є отримання всіх наявних даних в колекції. Зауважимо, що на деякі з запитів є обмеження у вигляді необхідної аутентифікації або наявності певної ролі у користувача.

```
const getTasks = async (req: Request, res: Response, next: NextFunction) =>
{
  try {
    const tasksData = await TaskService.getTasks();

    return res.status(200).json({ ...tasksData, message: 'Fetching tasks is
successful!' });
  } catch (e) {
    next(e);
  }
};
```

Наступна функція – отримання задачі по ідентифікатору. Працює аналогічно оновленню та видаленню, тільки ми лише шукаємо за ідентифікатором та повертаємо результат, а не маніпулюємо з даними запису.

```
const getTaskById = async (req: Request, res: Response, next: NextFunction)
=> {
  try {
    const taskID: string = req.params.taskID;

    const taskData = await TaskService.getTaskByID(taskID);

    return res.status(200).json({ ...taskData, message: 'Fetching task by
task ID is successful!' });
  } catch (e) {
    next(e);
  }
};
```

Також присутня функція отримання задачі по ідентифікатору користувача, який створив задачу.

```
const getTasksByCreatorId = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const creatorID: string = req.params.creatorID;

    const tasksData = await TaskService.getTaskByCreatorID(creatorID);

    return res.status(200).json({ ...tasksData, message: 'Fetching tasks by
creator ID is successful!' });
  } catch (e) {
```

```
    next(e);
  }
};
```

Ще присутній метод на отримання певної статистики, яка, знову ж таки, збирається в окремому методі сервісу та повертається у вигляді об’єкту.

```
const taskStatistics = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const taskStatistics = await TaskService.getStatistics();

    return res.status(200).json({ ...taskStatistics, message: 'Successfully
collected task stats!' });
  } catch (e) {
    next(e);
  }
};
```

Статистика формується за допомогою агрегатів, які відбирають дані за певними умовами, виконують певні операції, сортування, та формують кінцевий масив об’єктів який зібрав даний агрегат. Далі буде наведено приклад агрегату для отримання кількості виконаних задач які наявні в базі даних.

```
export const isCompletedCountPipeline: PipelineStage[] = [
  {
    $match: {
      isCompleted: true,
    },
  },
  {
    $group: {
      _id: null,
      count: { $sum: 1 },
    },
  },
];
```

Цей агрегат працює таким чином, що $match відбирає окремі задачі по умові, що задача виконана, а наступний параметр $group вже формує кінцеву відповідь та рахує кількість виконаних задач і створює об’єкт з полем count куди присвоює нараховану кількість задач.

Інші функції контролерів для отримання даних з моделей побудовані аналогічним чином і не потребують додаткового представлення.

Далі черга дійшла до самих кінцевих точок. Це такі частини коду, які дають серверу зрозуміти, що повинно виконуватись після звернення за даним маршрутом.

```
router.get('/statistics', AuthMiddleware, tasksControllers.taskStatistics);

router.get('/:taskID', tasksControllers.getTaskById);
```

```
router.get('/user/:creatorID', tasksControllers.getTasksByCreatorId);

router.get('/', AuthMiddleware, tasksControllers.getTasks);

router.post('/', taskHandlers, AuthMiddleware, upload.single('image'),
tasksControllers.createTask);

router.patch('/:taskID', taskHandlers, AuthMiddleware,
upload.single('image'), tasksControllers.updateTaskByID);

router.delete('/:taskID', AuthMiddleware, tasksControllers.deleteTaskByID);
```

Це всі кінцеві точки, які відносяться до маніпулювання над даними колекції задач. Варто зауважити, що в коді присутній «AuthMiddleware» який перевіряє чи авторизований користувач, а також «taskHandlers». Це масив з функціями які відловлюють передані дані в запиті та перевіряють їх виконуючі інші функції. Більш детально дані запити буде розлянуто в наступному розділі.

Для збереження картинок було використано такий пакет як «multer». Даний пакет дає змогу налаштувати сховище куди будуть зберігатися відловлені з запитів картинки та з яким іменем зберігатися.

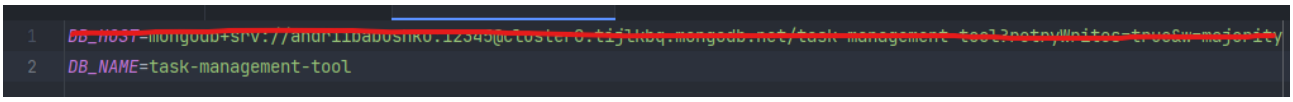
```
const storage = multer.diskStorage({
  destination:
path.join(dirname(dirname(dirname(fileURLToPath(import.meta.url)))),
'uploads/task'),
  filename: (req, file, callback) => {
    console.log(file);
    callback(null, file.fieldname + Date.now() + '.' +
file.mimetype.split('/')[1]);
  },
});
const upload = multer({ storage });
```

Варто ще продемонструвати як саме сервер підключається до бази даних.

```
const mongoURL: string | undefined = process.env.DB_HOST;
const port: string | undefined = process.env.PORT;
const corsOrigin = process.env.CLIENT_URL;

const app: Express = express();
mongoose
  .connect(mongoURL)
  .then(async () => {
    console.log('DB is connected!');
    try {
      await RoleService.createRolesIfNotExist();
    } catch (e) {
      throw e;
    }
  })
  .catch((error) => console.log(error));
```

Дані, по яким здійснюється підключення знаходяться в окремому файлі оточення(дані були замазані з міркувань безпеки).



Висновки до третього розділу

В цьому розділі було розглянуто роботу програми, зроблено повне її тестування, перевірено коректність взаємодії усіх методів роботи з базою даних, розглянуто основні можливості ПЗ, ознайомлено з основними виключеннями та похибками. Помилки у роботі програми можуть виникнути у разі неправильного використання програми, або за непередбачуваних обставин.

РОЗДІЛ 4. АДМІНІСТРУВАННЯ БАЗ ДАНИХ ТА БЕЗПЕКА

4.1 Розробка заходів захисту інформації в БД

У цьому розділі будемо зосереджуватись на адмініструванні бази даних, забезпеченні безпеки при роботі з даними та їх збереженні. Основною метою цих заходів є виявлення можливих слабких місць веб-сайту та запобігання помилкам в його роботі.

Найбільш вразливим моментом є вхідні дані, які мають бути перевірені на сервері та надіслані до бази даних у визначеному форматі. Якщо дані не відповідають вимогам, такі запити відхиляються. Для цього використовуються функції валідації, пов'язані з моделями, які перевіряють всі важливі та критичні дані. Наприклад, розглянемо процес валідації даних під час додавання задач.

```
const taskHandlers: ValidationChain[] = [
  body('title').optional().isLength({ min: 3, max: 30 }),
  body('deadlineDate').optional().isDate(),
  body('tags.*').optional().isMongoId(),
  body('categories.*').optional().isMongoId(),
  body('access').optional().isString(),
];
```

Бібліотека "express-validator" дозволяє проводити перевірку різних параметрів для вхідних полів, включаючи типи даних та розміри, а також встановлювати обов'язковість або можливість пропуску заповнення поля за допомогою методу "optional".

Після перевірки результати передаються до функції, яка відповідає за кінцеву точку запиту, де прокидується помилка у разі присутності помилок, інакше виконання функції продовжується до закінчення, звісно якщо ніяких помилок більше не буде.

```
const createTask = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req.body);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Create task validation error. Please, check
your credentials.', errors.array()));
    }
```

Це частина функції створення задачі яка відповідає за будь-які помилки отримані під час перевірки даних.

Також не менш важливою складовою забезпечення безпеки є перевірка ролі користувача додатку і визначення на основі цього доступу до певних операцій. Для цього були створені спеціальні функції перевірки, які переглядають дані користувача і загалом перевіряють, чи має користувач авторизацію.

Почнемо з перевірки авторизації, яка отримує токен і, у разі успішного отримання інформації про користувача з нього, дозволяє продовжувати виконання суміжних операцій.

```
export const AuthMiddleware = (req: IUserDataRequest, res: Response, next:
NextFunction): void => {
  try {
    const authHeader: string = req.headers.authorization;
    if (!authHeader) return next(HttpError.UnauthorizedError());

    const accessToken: string = authHeader.split(' ')[1];
    if (!accessToken) return next(HttpError.UnauthorizedError());

    const userData = TokenService.validateAccessToken(accessToken);
    if (!userData) return next(HttpError.UnauthorizedError());

    req.userData = userData;
    next();
  } catch (e) {
    return next(e);
  }
};
```

У контексті безпеки, паролі користувачів також повинні бути оброблені спеціальним чином, щоб не зберігатися у вхідному форматі. Для цього можна використовувати пакет "bcrypt", який хешує рядки та повертає їх у захешованому вигляді.

```
let hashPassword: string;

try {
  hashPassword = await bcrypt.hash(password, 4);
} catch (e) {
  throw new HttpError("Couldn't hash password. Please try again.", 500);
}
```

Тепер, для прикладу, розглянемо перевірку користувача на ролі. У цьому методі застосовується подібний алгоритм перевірки, але тепер витягується інформація про ролі зі змінної з даними користувача, які були визначені у минулій функції. Якщо ролі співпадають з визначеними у функції, користувач отримує доступ.

```
export const RoleMiddleware = async (req: IUserDataRequest, res: Response,
next: NextFunction) => {
  try {
    const userData = req.userData;

    if ([...userData.roles].some((role) => role.name === 'admin')) {
      next();
    } else {
      next(new HttpError('Access denied!', 403));
    }
  } catch (e) {
    next(e);
  }
};
```

Також варто навести матрицю доступу ролей, яка наведена у вигляді таблиці.

Матриця доступу ролей

	Матриця доступу		
	Не авторизований користувач	Авторизований користувач	Адміністратор
Users	2, 4	1, 3	5
Tasks	0	1, 2, 3, 4	5
Groups	0	1, 2, 3, 4	5
Roles	0	1	5
Categories	0	1	5
Tags	0	1	5
Tokens	0	1	5

Де 0 – немає доступу, 1 – читання, 2 – створення, 3 – редагування, 4 – видалення, 5 – повний доступ.

Також для ще більшої безпеки бази даних було розроблено окремі функції, які виконують бекап та відновлення бази даних кожен день. Далі буде наведено код функції для бекапу даних:

```
const backupMongoDB = (): Promise<{ message: string; status: number }> => {
  return new Promise((resolve, reject) => {
```

```
    const child = spawn('mongodump', [`--db=${DB_NAME}`, `--
archive=${ARCHIVE_PATH}`, '--gzip']);

    child.stdout.on('data', (data) => {
      console.log('stdout:\n' + data);
    });

    child.stderr.on('data', (data) => {
      console.log('stderr:\n' + data);
    });

    child.on('error', (err) => {
      console.log('err:\n' + err);
      reject(new HttpError('Something went wrong while backing up DB.',
500));
    });

    child.on('exit', (code, signal) => {
      if (code) {
        reject(new HttpError(`Process exit with code: ${code}`, 500));
      } else if (signal) {
        reject(new HttpError(`Process killed with signal: ${signal}`, 500));
      } else {
        resolve({ message: 'DB backing up is successful <3', status: 200 });
      }
    });
  });
};
```

Код функції для відновлення бази даних:

```
const restoreMongoDB = (): Promise<{ message: string; status: number }> => {
  return new Promise((resolve, reject) => {
    const child = spawn('mongorestore', [`--db=${DB_NAME}`, `--
archive=${ARCHIVE_PATH}`, '--gzip']);

    child.stdout.on('data', (data) => {
      console.log('stdout:\n' + data);
    });

    child.stderr.on('data', (data) => {
      console.log('stderr:\n' + data);
    });

    child.on('error', (err) => {
      console.log('err:\n' + err);
      reject(new HttpError('Something went wrong while restoring DB.',
500));
    });

    child.on('exit', (code, signal) => {
      if (code) {
        reject(new HttpError(`Process exit with code: ${code}`, 500));
      } else if (signal) {
        reject(new HttpError(`Process killed with signal: ${signal}`, 500));
      } else {
        resolve({ message: 'DB restore is successful <3', status: 200 });
      }
    });
  });
};
```

Код для виклику функцій бекапу та відновлення бази даних на кожен день:

```
cron.schedule('0 0 * * *', async () => {
  try {
    await backupMongoDB();
    await restoreMongoDB();
  } catch (error) {
    console.error('An error occurred during backup and restore:', error);
  }
});
```

Для перевірки працездатності всіх маршрутів, та доступу до маніпулювання над даними було використано таке середовище як Postman. Далі на рисунку буде наведено всі створені запити до серверу, який, в свою чергу, виконує функції які взаємодіють з базою даних.

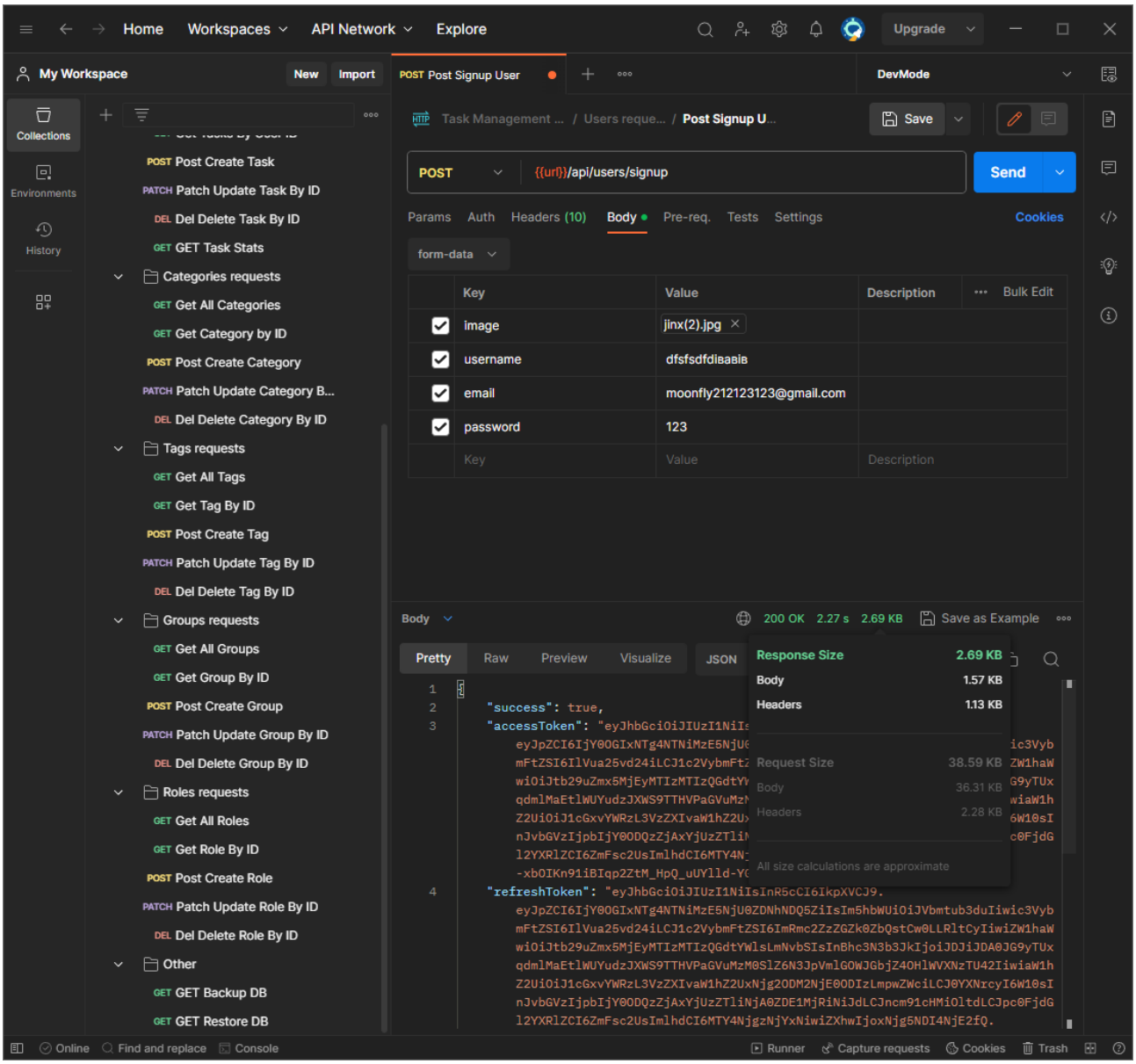


Рис. 4.1. Створені запити до бази даних в середовищі Postman

ВИСНОВКИ

Під час виконання поставленого завдання на курсову роботу були отримані практичні навички з написання програми, яка використовує базу даних MongoDB за допомогою технологій, основаних на JavaScript, а саме React JS, Node JS та прилеглих до даних технологій фреймворків та бібліотек. і продемонстровано ці навички при написанні реального сайту, який працює без нарікань та проблем.

В першому розділі було проаналізовано поставлену задачу саме які дії потрібно виконати для повної розробки програми, а також конкурентів зі схожим функціоналом та інтерфейсом. Стало зрозуміло основні потреби при розробці програми та поставлено конкретні цілі на проект. Після самостійного дослідження, всі питання та невирішені моменти були обговорені з керівником.

В другому розділі було описано логічну роботу програми, її потреби та проілюстровано за допомогою блок-схеми, діаграм класів та діаграми відносин БД. Цей етап роботи та розробки програмного забезпечення є одним з найважливіших, оскільки саме від проектування та логічної реалізації всіх функцій програмного додатку залежить надійність його роботи.

У третьому розділі ми представили інтерфейс та описали роботу застосунку, його функціонал та можливості. Зображення певних елементів сайту допомогли ілюструвати описане. Також було представлено функції та блоки коду, які були реалізовані для підтримки застосунку.

Четвертий розділ був присвячений адмініструванню бази даних та безпеці в цілому. Було проаналізовано заходи безпеки для збережених даних та перевірки нових вхідних даних. Описано роботу з користувачами та їх ролями на рівні бази даних, проведено тестування та отримано очікувані результати.

Підсумовуючи вище сказане, можем сказати, що ця робота дала нам змогу закріпити раніше вивчений матеріал, написавши для цього реальний веб-сайт. Тобто, задача на курсову роботу була успішно виконана та досягнута!

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

ЛІТЕРАТУРА

1. Редактор блок-схем - https://app.diagrams.net
2. Редактор для діаграм - https://excalidraw.com
3. Офіційна документація React: https://reactjs.org/docs/
4. Офіційна документація MobX: https://mobx.js.org/README.html
5. Офіційна документація React Router: https://reactrouter.com/en/main
6. Офіційна документація React Hook Form: https://www.react-hook-form.com
7. Офіційна документація Node.js: https://nodejs.org/en/docs/
8. Офіційна документація Express.js: https://expressjs.com/
9. Node.js – завантаження картинки на сервер: https://www.youtube.com/watch?v=wIOpe8S2Mk8&t=829s
10. Node.js – бекапи та відновлення БД в MongoDB: https://www.youtube.com/watch?v=JlM81PN9OP4
11. React, Node.JS додаток від автора UlbiTV: https://www.youtube.com/watch?v=fN25fMQZ2v0

ДОДАТКИ

ТЕХНІЧНЕ ЗАВДАННЯ

1. Загальне положення

1.1. Найменування програмного засобу

Повне найменування програмної системи: «База даних для інструменту керування завданнями». Коротка назва програмної системи – «Task Spring Tool».

1.2. Призначення розробки та область застосування

Програма призначена для керування своєю продуктивністю, ставлячи собі певні задачі та виконуючи їх разом, або окремо.

Дана програма має бути використана як веб-застосунок, що забезпечує доступ з будь-якого пристрою.

1.3. Найменування розробника та замовника

Розробник даного продукту – студент групи ВТ-21-1 Бабушко Андрій Сергійович.

Замовник продукту – кафедра інженерії програмного забезпечення Державного університету «Житомирська політехніка» в межах виконання курсової роботи з дисципліни «Бази даних».

2. Підстава для розробки

2.1. Документ на підставі якого ведеться розробка

Робота ведеться на підставі навчального плану за напрямом: 121. «Інженерія програмного забезпечення».

3. Вимоги до програми

3.1. Вимоги до функціональних характеристик

3.1.1. Загальні вимоги

Веб-застосунок має надавати:

- Постійний доступ для користувачів
- Адміністрування для адміністраторів сайту

- Організацію управління сайтом
- Можливість доступу до бази даних

3.1.2. Склад виконуваних функцій

Розробити базу даних для інструменту керування завданнями з таким функціоналом та можливостями:

- Перегляд своїх створених задач та задач інших користувачів
- Створення власних груп для виконання спільних завдань з іншими користувачами
- Перегляд інформації про конкретну задачу
- Виконання задач
- Повідомлення що мають сповіщати користувача про те що строк виконання завдання закінчився
- Авторизація, реєстрація, ролі користувачів
- Адміністрування(додавання, редагування, видалення)

3.1.3. Організація вхідних і вихідних даних

Вхідними даними є інформація про задачі(назва, опис, дата створення, дата закінчення, картинка задачі, дані про виконання задачі, тощо) та групи(назва, опис, користувачі які до неї належать, картинка групи, тощо).

Занесення інформації повинно виконуватись за допомогою діалогових вікон, побудованих на основі візуальних компонентів. Введення даних виконується на основі затверджених форм документів: анкета, заява, інформаційна довідка та в режимі online оператором зі слів користувача.

3.1.4. Часові характеристики і розмір пам’яті, необхідної для роботи програми

Максимальний час відгуку програми на дії користувача повинен бути не більше 0,5 секунди. Виконання команд меню не повинно займати більше 1 секунди. Запити на відображення масивів даних повинні завершуватися протягом 3 хвилин. База даних повинна бути доступною на рівні 90% протягом

усіх годин доби. Час підключення до бази даних не повинен перевищувати 1 хвилину. Мінімальний обсяг оперативної пам'яті, необхідний для роботи програми, становить 1 Гб. Дисковий простір, потрібний для збереження програми і файлів даних, не повинен перевищувати 300 Мб.

3.2. Вимоги до надійності

3.2.1 Вимоги до надійного функціонування

Програма повинна безперебійно працювати при неперервній роботі комп'ютера. Доступність бази даних повинна становити 90% при одночасному доступі 30 користувачів. В разі виникнення апаратних збоїв, відновлення нормального функціонування програми має бути здійснене шляхом перезавантаження операційної системи комп'ютера та запуску виконуваного файлу програми.

3.2.2 Контроль вхідної та вихідної інформації

Для забезпечення коректності вхідної інформації використовується механізм автоматичного заповнення та вибору зі списку. Некоректні дії повинні супроводжуватися повідомленнями про помилку і блокуванням операцій оновлення даних. В системі передбачений захист від загального блокування.

ДОДАТОК А

Вихідний код проекту. Контролер задач:

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

import HttpError from '../exceptions/http-error.js';
import { IUserDataRequest } from '../ts/interfaces/IUserDataRequest.js';
import TaskService from '../services/task-service.js';

const getTaskById = async (req: Request, res: Response, next: NextFunction)
=> {
  try {
    const taskID: string = req.params.taskID;

    const taskData = await TaskService.getTaskByID(taskID);

    return res.status(200).json({ ...taskData, message: 'Fetching task by
task ID is successful!' });
  } catch (e) {
    next(e);
  }
};

const getTasksByCreatorId = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const creatorID: string = req.params.creatorID;

    const tasksData = await TaskService.getTaskByCreatorID(creatorID);

    return res.status(200).json({ ...tasksData, message: 'Fetching tasks by
creator ID is successful!' });
  } catch (e) {
    next(e);
  }
};

const getTasks = async (req: Request, res: Response, next: NextFunction) =>
{
  try {
    const tasksData = await TaskService.getTasks();

    return res.status(200).json({ ...tasksData, message: 'Fetching tasks is
successful!' });
  } catch (e) {
    next(e);
  }
};

const createTask = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req.body);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Create task validation error. Please, check
your credentials.', errors.array()));
    }
```

```
    const imagePath = req.file?.filename ?
`uploads/task/${req.file.filename}` : undefined;
    const { id: creatorID } = req.userData;

    const taskData = await TaskService.createTask(creatorID, { ...req.body,
image: imagePath });

    return res.status(200).json({ ...taskData, message: 'Task is
successfully created!' });
  } catch (e) {
    next(e);
  }
};

const updateTaskByID = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Update task validation error. Please, check
your credentials.', errors.array()));
    }

    const taskID = req.params.taskID;
    const imagePath = req.file?.filename ?
`uploads/task/${req.file.filename}` : undefined;
    const { id: userID } = req.userData;

    const taskData = await TaskService.updateTask(taskID, userID, {
...req.body, image: imagePath });

    return res.status(200).json({ ...taskData, message: 'Task is
successfully updated!' });
  } catch (e) {
    next(e);
  }
};

const deleteTaskByID = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const taskID: string = req.params.taskID;
    const { id: userID } = req.userData;

    const taskData = await TaskService.deleteTask(taskID, userID);

    return res.status(200).json({
      ...taskData,
      message: `Successful deleted task with ${taskID} ID.`,
    });
  } catch (e) {
    next(e);
  }
};

const taskStatistics = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const taskStatistics = await TaskService.getStatistics();
```

```
    return res.status(200).json({ ...taskStatistics, message: 'Successfully
collected task stats!' });
  } catch (e) {
    next(e);
  }
};

export { getTaskById, getTasksByCreatorId, getTasks, createTask,
updateTaskByID, deleteTaskByID, taskStatistics };
```

Сервіс задач:

```
import { mongooseModel as UserModel } from '../models/user-model.js';
import { mongooseModel as TaskModel } from '../models/task-model.js';
import { mongooseModel as CategoryModel } from '../models/category-
model.js';
import { mongooseModel as TagModel } from '../models/tag-model.js';
import HttpError from '../exceptions/http-error.js';
import TaskDto from '../DTO/task-dto.js';
import { ITask } from '../ts/interfaces/ITask.js';
import { ClientSession, Document, startSession, Types } from 'mongoose';
import { IUser } from '../ts/interfaces/IUser.js';
import ImageService from './image-service.js';
import {
  isCompletedCountPipeline,
  theOldestTaskPipeline,
  userWithMaxTasksCreatedPipeline,
} from '../aggregates/task-aggregate.js';

class TaskService {
  async getTasks() {
    let tasks;

    try {
      tasks = await TaskModel.find().populate(['creator', 'tags',
'categories']);
    } catch (e) {
      throw new HttpError('Something went wrong while fetching tasks data
from DB.', 500);
    }

    return {
      tasks: tasks.map((t) => new TaskDto(t)),
      success: true,
    };
  }

  async getTaskByID(taskID: string) {
    let task;

    try {
      task = await TaskModel.findById(taskID).populate(['creator', 'tags',
'categories']);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
task by task ID.', 500);
    }

    if (!task) {
      throw new HttpError("Couldn't find a task for the provided task ID!",
404);
```

```
    }

    const taskDTO: TaskDto = new TaskDto(task);
    return { task: taskDTO, success: true };
  }

  async getTaskByCreatorID(creatorID: string) {
    let tasks;

    try {
      tasks = await TaskModel.find({ creator: creatorID
}).populate(['creator', 'tags', 'categories']);
    } catch (error) {
      throw new HttpError('Something went wrong while searching for some
tasks by creator ID.', 500);
    }

    if (!tasks || tasks.length == 0) {
      throw new HttpError("Couldn't find tasks for the provided user ID!",
422);
    }

    return {
      tasks: tasks.map((t) => new TaskDto(t)),
      success: true,
    };
  }

  private async checkTaskData({ deadlineDate, tags, categories }: ITask,
creatorID = null) {
    if (new Date(deadlineDate).getTime() <= new Date().getTime())
      throw new HttpError('Task deadline date is invalid!', 422);

    let identifiedCreator: Document<unknown, NonNullable<unknown>, IUser> &
        Omit<IUser & { _id: Types.ObjectId }, never>,
      identifiedTags: Awaited<void>[],
      identifiedCategories;

    if (creatorID != null) {
      try {
        identifiedCreator = await UserModel.findById(creatorID);
      } catch (e) {
        throw new HttpError("Couldn't find task creator for provided id!",
500);
      }

      if (!identifiedCreator) throw new HttpError("Couldn't find task
creator for provided id.", 404);
    }

    if (categories && categories.length > 0) {
      try {
        identifiedCategories = await Promise.all(
          categories.map(async (c) => {
            let identifiedCategory;

            try {
              identifiedCategory = await CategoryModel.findById(c);
            } catch (e) {
              throw new HttpError("Couldn't find category for provided id!",
500);
```

```
            }

            if (!identifiedCategory) throw new HttpError("Couldn't find
category for provided id.", 404);
          }),
        );
      } catch (e) {
        throw new HttpError(e.message, e.status);
      }
    }

    if (tags && Array.isArray(tags) && tags.length > 0) {
      try {
        identifiedTags = await Promise.all(
          tags.map(async (t) => {
            let identifiedTag;

            try {
              identifiedTag = await TagModel.findById(t);
            } catch (e) {
              throw new HttpError("Couldn't find tag for provided id!",
500);
            }

            if (!identifiedTag) {
              throw new HttpError("Couldn't find tag for provided id!",
404);
            }
          }),
        );
      } catch (e) {
        throw new HttpError(e.message, e.status);
      }
    }

    return {
      identifiedCreator,
      identifiedTags,
      identifiedCategories,
    };
  }

  async createTask(
    creatorID: Types.ObjectId,
    { title, description, deadlineDate, image, tags, categories, access }:
ITask,
  ) {
    let createdTask, taskCheck;

    if (tags && !Array.isArray(tags)) tags = Array(tags);
    if (categories && !Array.isArray(categories)) categories =
Array(categories);
    // if (tags && typeof tags === 'string') tags = JSON.parse(tags);
    // if (categories && typeof categories === 'string') categories =
JSON.parse(categories);

    try {
      taskCheck = await this.checkTaskData({ deadlineDate, tags, categories
}, creatorID);
    } catch (e) {
      throw e;
```

```
    }

    if (!(taskCheck instanceof HttpError)) {
      try {
        createdTask = await TaskModel.create({
          title,
          description,
          creationDate: new Date(),
          deadlineDate,
          creator: creatorID,
          image,
          tags,
          categories,
          access,
        });
      } catch (e) {
        console.log(e);
        throw new HttpError('Creating task failed!', 500);
      }

      try {
        const session: ClientSession = await startSession();
        session.startTransaction();
        await createdTask.save({ session });
        taskCheck.identifiedCreator.tasks.push(createdTask);
        await taskCheck.identifiedCreator.save({ session });
        await session.commitTransaction();
      } catch (e) {
        throw new HttpError('Creating task failed. Pls check credentials and
try again.', 500);
      }

      const taskDTO: TaskDto = new TaskDto(createdTask);

      return {
        task: taskDTO,
        success: true,
      };
    }

    throw new HttpError(taskCheck.message, taskCheck.status);
  }

  async updateTask(
    taskID: string,
    userID: Types.ObjectId,
    { title, description, deadlineDate, image, tags, categories, access,
isCompleted }: ITask,
  ) {
    let taskCheck;

    if (tags && !Array.isArray(tags)) tags = Array(tags);
    if (categories && !Array.isArray(categories)) categories =
Array(categories);

    try {
      taskCheck = await this.checkTaskData({ deadlineDate, tags, categories
});
    } catch (e) {
      throw new HttpError(e.message, 500);
    }
```

```
    if (!(taskCheck instanceof HttpError)) {
      let updatedTask;

      try {
        updatedTask = await TaskModel.findById(taskID).populate('creator');
      } catch (e) {
        throw new HttpError('Something went wrong while searching for some
task by task ID.', 500);
      }

      if (!updatedTask) {
        throw new HttpError("Couldn't find a task for the provided task
ID!", 404);
      }

      if (updatedTask.creator._id != userID) {
        throw new HttpError("No access to change task. Different user and
creator id's.", 404);
      }

      ImageService.deleteImage(updatedTask,
'uploads/task/no_task_image.png');

      console.log(tags);
      console.log(categories);

      if (title) updatedTask.title = title;
      if (description) updatedTask.description = description;
      if (deadlineDate) updatedTask.deadlineDate = deadlineDate;
      if (image) updatedTask.image = image;
      if (tags) updatedTask.tags = tags;
      else updatedTask.tags = [];
      if (categories) updatedTask.categories = categories;
      else updatedTask.categories = [];
      if (access) updatedTask.access = access;
      if (isCompleted) updatedTask.isCompleted = isCompleted;

      try {
        await updatedTask.save();
      } catch (e) {
        throw new HttpError('Something went wrong while updating task.',
500);
      }

      const taskDto: TaskDto = new TaskDto(updatedTask);

      return {
        task: taskDto,
        success: true,
      };
    }

    throw new HttpError(taskCheck.message, taskCheck.status);
  }

  async deleteTask(taskID: string, userID: Types.ObjectId) {
    let deletedTask;

    try {
      deletedTask = await TaskModel.findById(taskID).populate({
```

```
        path: 'creator',
      });
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
task by task ID.', 500);
    }

    if (!deletedTask) {
      throw new HttpError("Couldn't find a task for the provided task ID!",
404);
    }

    if (deletedTask.creator._id != userID) {
      throw new HttpError("No access to delete task. Different user and
creator id's.", 404);
    }

    ImageService.deleteImage(deletedTask, 'uploads/task/no_task_image.png');

    try {
      const session: ClientSession = await startSession();
      session.startTransaction();
      await deletedTask.deleteOne({ session });
      deletedTask.creator.tasks.pull(deletedTask);
      await deletedTask.creator.save({ session });
      await session.commitTransaction();
    } catch (e) {
      throw new HttpError('Something went wrong while deleting task.', 500);
    }

    return {
      success: true,
    };
  }

  async getStatistics() {
    let completedTasksCount, theBestUserWithMaxTasks, theOldestTask;

    try {
      completedTasksCount = await
TaskModel.aggregate(isCompletedCountPipeline);
      theBestUserWithMaxTasks = await
TaskModel.aggregate(userWithMaxTasksCreatedPipeline);
      theOldestTask = await TaskModel.aggregate(theOldestTaskPipeline);
    } catch (e) {
      console.log(e);
      throw new HttpError('Something went wrong while collecting task
stats.', 500);
    }

    return {
      success: true,
      theBestUser: theBestUserWithMaxTasks,
      completedTasksCount,
      theOldestTask,
    };
  }
}

export default new TaskService();
```

Контролер користувачів:

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

import HttpError from '../exceptions/http-error.js';
import UserService from '../services/user-service.js';
import { IUserDataRequest } from '../ts/interfaces/IUserDataRequest.js';

const getUsers = async (req: Request, res: Response, next: NextFunction) =>
{
  try {
    const users = await UserService.getUsers();

    return res.status(200).json(users);
  } catch (e) {
    next(e);
  }
};

const signup = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const errors = validationResult(req.body);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Signup validation error. Please, check
credentials.', errors.array()));
    }

    const imagePath = `uploads/user/${req.file.filename}`;

    const userData = await UserService.signup({ ...req.body, image:
imagePath });

    res.cookie('refreshToken', userData.refreshToken, {
      maxAge: 30 * 24 * 60 * 60 * 1000,
      httpOnly: true,
    });

    return res.status(200).json({ ...userData, message: 'Successfully signed
up!' });
  } catch (e) {
    return next(e);
  }
};

const login = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const errors = validationResult(req.body);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Login validation error. Please, check
credentials.', errors.array()));
    }

    const userData = await UserService.login({ ...req.body });

    res.cookie('refreshToken', userData.refreshToken, {
      maxAge: 30 * 24 * 60 * 60 * 1000,
      httpOnly: true,
    });
```

```
    return res.status(200).json({ ...userData, message: 'Successfully logged
in!' });
  } catch (e) {
    next(e);
  }
};

const logout = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const { refreshToken } = req.cookies;
    const token = await UserService.logout(refreshToken);
    res.clearCookie('refreshToken');
    return res.status(200).json({ ...token, message: 'Successfully logged
out!' });
  } catch (e) {
    return next(e);
  }
};

const activateLink = async (req: Request, res: Response, next: NextFunction)
=> {
  try {
    const activationLink: string = req.params.link;

    await UserService.activate(activationLink);

    return res.redirect(process.env.CLIENT_URL);
  } catch (e) {
    return next(e);
  }
};

const refreshLink = async (req: Request, res: Response, next: NextFunction)
=> {
  try {
    const { refreshToken } = req.cookies;

    const userData = await UserService.refresh(refreshToken);
    res.cookie('refreshToken', userData.refreshToken, {
      maxAge: 30 * 24 * 60 * 60 * 1000,
      httpOnly: true,
    });

    return res.status(200).json(userData);
  } catch (e) {
    next(e);
  }
};

const updateUser = async (req: Request, res: Response, next: NextFunction)
=> {
  try {
    const errors = validationResult(req.body);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Update user validation error. Please, check
your credentials.', errors.array()));
    }

    const imagePath = `uploads/user/${req.file.filename}`;
```

```
    const userID: string = req.params.userID;

    const userData = await UserService.updateUser(userID, { ...req.body,
image: imagePath });

    return res.status(200).json({ ...userData, message: 'User is
successfully updated!' });
  } catch (e) {
    next(e);
  }
};

const deleteUser = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const userID: string = req.params.userID;

    const userData = await UserService.deleteUser(userID);

    return res.status(200).json({
      ...userData,
      message: `Successful deleted user with ${userID} ID.`,
    });
  } catch (e) {
    next(e);
  }
};

const resendActivationMail = async (req: IUserDataRequest, res: Response,
next: NextFunction) => {
  try {
    const status = await
UserService.sendActivationMail(req.userData.activationLink,
req.userData.email);

    if (!status) {
      next(new HttpError('Send activation mail failed!', 500));
    }

    return res.status(200).json({ message: 'Successfully resend activation
mail!', success: true });
  } catch (e) {
    next(e);
  }
};

const userStatistics = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const userStatistics = await UserService.getStatistics();

    return res.status(200).json({ ...userStatistics, message: 'Successfully
collected user stats!' });
  } catch (e) {
    next(e);
  }
};

export {
  getUsers,
```

```
  login,
  signup,
  activateLink,
  refreshLink,
  logout,
  updateUser,
  deleteUser,
  resendActivationMail,
  userStatistics,
};
```

Сервіс користувачів:

```
import bcrypt from 'bcrypt';
import { v4 as uuidv4 } from 'uuid';
import { ClientSession, startSession, Types } from 'mongoose';

import { mongooseModel as UserModel } from '../models/user-model.js';
import { mongooseModel as RoleModel } from '../models/role-model.js';
import { mongooseModel as GroupModel } from '../models/group-model.js';
import { mongooseModel as TaskModel } from '../models/task-model.js';
import MailService from './mail-service.js';
import TokenService from './token-service.js';
import UserDto from '../DTO/user-dto.js';
import HttpError from '../exceptions/http-error.js';
import { IUser } from '../ts/interfaces/IUser.js';
import RoleService from './role-service.js';
import ImageService from './image-service.js';
import {
  notActivatedUsersPipeline,
  usersByRolesPipeline,
  usersWithoutTasksPipeline,
} from '../aggregates/user-aggregate.js';

class UserService {
  async signup({ name, surname, username, email, password, image }: IUser) {
    let indentifiedUser;

    try {
      indentifiedUser = await UserModel.findOne({ email: email
}).populate(['roles', 'groups']);
    } catch (e) {
      throw new HttpError('Something went wrong while singing up! Please,
try again.', 500);
    }

    if (indentifiedUser) {
      throw HttpError.BadRequest(
        `User with email address ${email} already exists! Please, choose
another email and try again.`,
      );
    }

    let hashPassword: string;

    try {
      hashPassword = await bcrypt.hash(password, 4);
    } catch (e) {
      throw new HttpError("Couldn't hash password. Please try again.", 500);
    }

    let userRole;
```

```
    try {
      userRole = await RoleModel.findOne({ name: 'user' });
    } catch (e) {
      throw e;
    }

    if (!userRole) {
      try {
        userRole = await RoleService.createUserRole();
      } catch (e) {
        throw e;
      }
    }

    const activationLink: string = uuidv4();
    const roles: Types.ObjectId[] = [userRole.id];
    let user;

    try {
      await this.sendActivationMail(activationLink, email);
    } catch (e) {
      throw e;
    }

    try {
      user = await UserModel.create({
        name,
        surname,
        username,
        email,
        password: hashPassword,
        image,
        roles,
        activationLink,
      });
    } catch (e) {
      throw new HttpError(
        'Signing up failed. Please check credentials and try again. Maybe
you are using username or email that is already used.',
        500,
      );
    }

    const userDTO: UserDto = new UserDto(user);
    const tokens = TokenService.generateTokens({ ...userDTO });
    await TokenService.saveToken(userDTO.id, tokens.refreshToken);

    try {
      await TokenService.saveToken(userDTO.id, tokens.refreshToken);
    } catch (e) {
      throw new HttpError('Signing up failed! Please try again.', 500);
    }

    return {
      success: true,
      ...tokens,
      user: userDTO,
    };
  }
```

```
  async login({ email, password }: IUser) {
    let user;

    try {
      user = await UserModel.findOne({ email: email }).populate(['roles',
'groups']);
    } catch (e) {
      throw new HttpError('Something went wrong while logging up! Please,
try again.', 500);
    }

    if (!user) {
      throw HttpError.BadRequest(`Can't find user with ${email} email!
Please, check your credentials.`);
    }

    let isValidPassword = false;
    try {
      isValidPassword = await bcrypt.compare(password, user.password);
    } catch (e) {
      throw new HttpError("Couldn't log you in! Please, check your
credentials and try again.", 500);
    }

    if (!isValidPassword) {
      throw HttpError.BadRequest('Password is incorrect! Please, try
again.');
    }

    const userDTO: UserDto = new UserDto(user);
    const tokens = TokenService.generateTokens({ ...userDTO });
    try {
      await TokenService.saveToken(userDTO.id, tokens.refreshToken);
    } catch (e) {
      throw new HttpError('Something went wrong while saving token in DB!',
500);
    }

    console.log(tokens);
    return {
      success: true,
      ...tokens,
      user: userDTO,
    };
  }

  async logout(refreshToken: string) {
    if (!refreshToken) {
      throw HttpError.UnauthorizedError();
    }

    let token;

    try {
      token = await TokenService.removeToken(refreshToken);
    } catch (e) {
      throw new HttpError("Token doesn't exist! Please, try again.", 500);
    }

    return token;
  }
```

```
  async activate(activationLink: string) {
    let user;

    try {
      user = await UserModel.findOne({ activationLink });
    } catch (e) {
      throw new HttpError('Something went wrong while searching for user to
activate account! Please, try again.', 500);
    }

    if (!user) throw new HttpError('Incorrect activation link!', 422);

    user.isActivated = true;

    try {
      await user.save();
    } catch (e) {
      throw new HttpError('Something went wrong while saving user after
activating! Please, try again.', 500);
    }
  }

  async refresh(refreshToken: string) {
    if (!refreshToken) throw HttpError.UnauthorizedError();

    const userData = TokenService.validateRefreshToken(refreshToken);
    const token = await TokenService.findToken(refreshToken);

    if (!userData || !token) {
      throw HttpError.UnauthorizedError();
    }

    const user = await UserModel.findById(userData.id).populate(['roles',
'groups']);
    const userDTO = new UserDto(user);
    const tokens = TokenService.generateTokens({ ...userDTO });

    await TokenService.saveToken(userDTO.id, tokens.refreshToken);
    return {
      ...tokens,
      user: userDTO,
    };
  }

  async getUsers() {
    let users;

    try {
      users = await UserModel.find({}, '-password');
    } catch (e) {
      throw new HttpError('Something went wrong while fetching users data.',
500);
    }

    return {
      users: users.map((u) => new UserDto(u)),
      success: true,
    };
  }
```

```
  private async checkUserData({ roles, groups }) {
    let identifiedRoles, identifiedGroups;

    if (roles) {
      try {
        identifiedRoles = await Promise.all(
          roles.map(async (r) => {
            let identifiedRole;

            try {
              identifiedRole = await RoleModel.findById(r);
            } catch (e) {
              throw new HttpError("Couldn't find role for provided id!",
500);
            }

            if (!identifiedRole) throw new HttpError("Couldn't find role for
provided id.", 404);
          }),
        );
      } catch (e) {
        throw new HttpError(e.message, e.status);
      }
    }

    if (groups) {
      try {
        identifiedGroups = await Promise.all(
          groups.map(async (g) => {
            let identifiedGroup;

            try {
              identifiedGroup = await GroupModel.findById(g);
            } catch (e) {
              throw new HttpError("Couldn't find group for provided id!",
500);
            }

            if (!identifiedGroup) throw new HttpError("Couldn't find group
for provided id.", 404);
          }),
        );
      } catch (e) {
        throw new HttpError(e.message, e.status);
      }
    }

    return {
      identifiedRoles,
      identifiedGroups,
    };
  }

  async updateUser(userID: string, { name, surname, username, password,
image, roles, groups }: IUser) {
    let userCheck;

    try {
      userCheck = await this.checkUserData({ roles, groups });
    } catch (e) {
      throw new HttpError(e.message, e.status);
```

```
    }

    if (!(userCheck instanceof HttpError)) {
      let updatedUser;

      try {
        updatedUser = await
UserModel.findById(userID).populate('tasks').populate('roles').populate('gro
ups');
      } catch (e) {
        throw new HttpError('Something went wrong while searching for some
user by user ID.', 500);
      }

      if (!updatedUser) {
        throw new HttpError("Couldn't find a user for the provided user
ID!", 404);
      }

      ImageService.deleteImage(updatedUser,
'uploads/user/no_user_image.jpg');

      if (name) updatedUser.name = name;
      if (surname) updatedUser.surname = surname;
      if (username) updatedUser.username = username;
      if (password) updatedUser.password = password;
      if (image) updatedUser.image = image;
      if (roles) {
        if (updatedUser.roles.length == 0) updatedUser.roles = roles;
        else updatedUser.roles = [...updatedUser.roles, ...roles];
      }
      if (groups) {
        if (updatedUser.groups.length == 0) updatedUser.groups = groups;
        else updatedUser.groups = [...updatedUser.groups, ...groups];
      }

      try {
        await updatedUser.save();
      } catch (e) {
        throw new HttpError('Something went wrong while updating user.',
500);
      }

      const userDTO: UserDto = new UserDto(updatedUser);

      return {
        user: userDTO,
        success: true,
      };
    }

    throw new HttpError(
      'User data validation in DB failed while updating new user! Please,
check credentials and try again.',
      500,
    );
  }

  async deleteUser(userID: string) {
    let deletedUser;
```

```
    try {
      deletedUser = await UserModel.findById(userID).populate('groups');
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
user by user ID.', 500);
    }

    if (!deletedUser) {
      throw new HttpError("Couldn't find a user for the provided user ID!",
404);
    }

    ImageService.deleteImage(deletedUser, 'uploads/user/no_user_image.jpg');

    try {
      const session: ClientSession = await startSession();
      session.startTransaction();

      await TaskModel.deleteMany({ creator: userID }, { session });
      await GroupModel.updateMany({ users: { $in: [userID] } }, { $pull: {
users: userID } }, { session });
      await GroupModel.updateMany({ creator: userID }, { creator: null }, {
session });

      await deletedUser.deleteOne({ session });
      await session.commitTransaction();
    } catch (e) {
      throw new HttpError('Something went wrong while deleting task.', 500);
    }

    return {
      success: true,
    };
  }

  async sendActivationMail(activationLink: string, email: string) {
    try {
      await MailService.sendActivationMail(email,
`${process.env.API_URL}/api/users/activate/${activationLink}`);
    } catch (e) {
      return false;
    }

    return true;
  }

  async getStatistics() {
    let notActivatedUsers, usersWithoutTasks, usersByRoles;

    try {
      notActivatedUsers = await
UserModel.aggregate(notActivatedUsersPipeline);
      usersWithoutTasks = await
UserModel.aggregate(usersWithoutTasksPipeline);
      usersByRoles = await UserModel.aggregate(usersByRolesPipeline);
    } catch (e) {
      console.log(e);
      throw new HttpError('Something went wrong while collecting user
stats.', 500);
    }
```

```
    return {
      success: true,
      usersWithoutTasks,
      notActivatedUsers,
      usersByRoles,
    };
  }
}

export default new UserService();
```

Контролер категорій:

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

import HttpError from '../exceptions/http-error.js';
import CategoryService from '../services/category-service.js';

const getCategories = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const categoriesData = await CategoryService.getCategories();

    return res.status(200).json(categoriesData);
  } catch (e) {
    next(e);
  }
};

const getCategoryById = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const categoryID: string = req.params.categoryID;

    const categoryData = await CategoryService.getCategoryByID(categoryID);

    return res.status(200).json(categoryData);
  } catch (e) {
    next(e);
  }
};

const createCategory = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Create category validation error. Please,
check your credentials.', errors.array()));
    }

    const categoryData = await CategoryService.createCategory(req.body);

    return res.status(200).json({ ...categoryData, message: 'Category is
successfully created!' });
  } catch (e) {
    next(e);
  }
};
```

```
const updateCategoryById = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Update category validation error. Please,
check your credentials.', errors.array()));
    }

    const categoryID: string = req.params.categoryID;

    const tagData = await CategoryService.updateCategory(categoryID,
req.body);

    return res.status(200).json({ ...tagData, message: 'Category is
successfully updated!' });
  } catch (e) {
    next(e);
  }
};

const deleteCategoryById = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const categoryID: string = req.params.categoryID;

    const categoryData = await CategoryService.deleteCategory(categoryID);

    return res.status(201).json({
      ...categoryData,
      message: `Successful deleted category with ${categoryID} ID.`,
    });
  } catch (e) {
    next(e);
  }
};

export { getCategories, getCategoryById, createCategory, updateCategoryById,
deleteCategoryById };
```

Сервіс категорій:

```
import { ClientSession, startSession } from 'mongoose';

import { mongooseModel as CategoryModel } from '../models/category-
model.js';
import { mongooseModel as TaskModel } from '../models/task-model.js';
import HttpError from '../exceptions/http-error.js';
import CategoryDto from '../DTO/category-dto.js';
import { ICategory } from '../ts/interfaces/ICategory.js';

class CategoryService {
  async getCategories() {
    let categories;

    try {
      categories = await CategoryModel.find();
    } catch (e) {
      throw new HttpError('Something went wrong while fetching categories
data from DB.', 500);
    }
```

```
    return {
      categories: categories.map((c) => new CategoryDto(c)),
      success: true,
    };
  }

  async getCategoryByID(categoryID: string) {
    let category;

    try {
      category = await CategoryModel.findById(categoryID);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
category by category ID.', 500);
    }

    if (!category) {
      throw new HttpError("Couldn't find a category for the provided
category ID!", 404);
    }

    const categoryDTO: CategoryDto = new CategoryDto(category);
    return { category: categoryDTO, success: true };
  }

  async createCategory({ name, description }: ICategory) {
    let createdCategory;

    try {
      createdCategory = await CategoryModel.create({
        name,
        description,
      });
    } catch (e) {
      throw new HttpError('Creating category failed!', 500);
    }

    const categoryDTO: CategoryDto = new CategoryDto(createdCategory);

    return {
      category: categoryDTO,
      success: true,
    };
  }

  async updateCategory(categoryID: string, { name, description }: ICategory)
{
    let updatedCategory;

    try {
      updatedCategory = await CategoryModel.findById(categoryID);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
category by category ID.', 500);
    }

    if (!updatedCategory) {
      throw new HttpError("Couldn't find a category for the provided
category ID!", 404);
    }
```

```
    if (name) updatedCategory.name = name;
    if (description) updatedCategory.description = description;

    try {
      await updatedCategory.save();
    } catch (e) {
      throw new HttpError('Something went wrong while updating category.',
500);
    }

    const categoryDTO: CategoryDto = new CategoryDto(updatedCategory);

    return {
      category: categoryDTO,
      success: true,
    };
  }

  async deleteCategory(categoryID: string) {
    let deletedCategory, tasks;

    try {
      tasks = await TaskModel.find().populate('tags');
      deletedCategory = await CategoryModel.findById(categoryID);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
data in DB.', 500);
    }

    if (!deletedCategory) {
      throw new HttpError("Couldn't find a category for the provided
category ID!", 404);
    }

    try {
      const session: ClientSession = await startSession();
      session.startTransaction();

      for (const task of tasks) {
        await TaskModel.updateOne({ _id: task._id }, { $pull: { categories:
deletedCategory._id } }, { session });
      }

      await deletedCategory.deleteOne({ session });
      await session.commitTransaction();
    } catch (e) {
      throw new HttpError('Something went wrong while deleting category.',
500);
    }

    return {
      success: true,
    };
  }
}

export default new CategoryService();
```

Контролер тегів:

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

import TagService from '../services/tag-service.js';
import HttpError from '../exceptions/http-error.js';

const getTags = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const tagsData = await TagService.getTags();

    return res.status(200).json(tagsData);
  } catch (e) {
    next(e);
  }
};

const getTagById = async (req: Request, res: Response, next: NextFunction)
=> {
  try {
    const tagID: string = req.params.tagID;

    const tagData = await TagService.getTagByID(tagID);

    return res.status(200).json(tagData);
  } catch (e) {
    next(e);
  }
};

const createTag = async (req: Request, res: Response, next: NextFunction) =>
{
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Create tag validation error. Please, check
your credentials.', errors.array()));
    }

    const tagData = await TagService.createTag(req.body);

    return res.status(200).json({ ...tagData, message: 'Tag is successfully
created!' });
  } catch (e) {
    next(e);
  }
};

const updateTagById = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Update tag validation error. Please, check
your credentials.', errors.array()));
    }

    const tagID: string = req.params.tagID;

    const tagData = await TagService.updateTag(tagID, req.body);
```

```
    return res.status(200).json({ ...tagData, message: 'Tag is successfully
updated!' });
  } catch (e) {
    next(e);
  }
};

const deleteTagById = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const tagID: string = req.params.tagID;

    const tagData = await TagService.deleteTag(tagID);

    return res.status(201).json({
      ...tagData,
      message: `Successful deleted tag with ${tagID} ID.`,
    });
  } catch (e) {
    next(e);
  }
};

export { getTags, getTagById, createTag, updateTagById, deleteTagById };
```

Сервіс тегів:

```
import { ClientSession, startSession } from 'mongoose';

import { mongooseModel as TagModel } from '../models/tag-model.js';
import { mongooseModel as TaskModel } from '../models/task-model.js';
import HttpError from '../exceptions/http-error.js';
import TagDto from '../DTO/tag-dto.js';
import { ITag } from '../ts/interfaces/ITag.js';

class TagService {
  async getTags() {
    let tags;

    try {
      tags = await TagModel.find();
    } catch (e) {
      throw new HttpError('Something went wrong while fetching tags data
from DB.', 500);
    }

    return {
      tags: tags.map((t) => new TagDto(t)),
      success: true,
    };
  }

  async getTagByID(tagID: string) {
    let tag;

    try {
      tag = await TagModel.findById(tagID);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some tag
by tag ID.', 500);
    }
```

```
    if (!tag) {
      throw new HttpError("Couldn't find a tag for the provided tag ID!",
404);
    }

    const tagDTO: TagDto = new TagDto(tag);

    return { tag: tagDTO, success: true };
  }

  async createTag({ name, description }: ITag) {
    let createdTag;

    try {
      createdTag = await TagModel.create({
        name,
        description,
      });
    } catch (e) {
      throw new HttpError('Creating tag failed!', 500);
    }

    const tagDTO: TagDto = new TagDto(createdTag);

    return {
      tag: tagDTO,
      success: true,
    };
  }

  async updateTag(tagID: string, { name, description }: ITag) {
    let updatedTag;

    try {
      updatedTag = await TagModel.findById(tagID);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some tag
by tag ID.', 500);
    }

    if (!updatedTag) {
      throw new HttpError("Couldn't find a tag for the provided tag ID!",
404);
    }

    if (name) updatedTag.name = name;
    if (description) updatedTag.description = description;

    try {
      await updatedTag.save();
    } catch (e) {
      throw new HttpError('Something went wrong while updating tag.', 500);
    }

    const tagDTO: TagDto = new TagDto(updatedTag);

    return {
      tag: tagDTO,
      success: true,
    };
```

```
  }

  async deleteTag(tagID: string) {
    let deletedTag, tasks;

    try {
      tasks = await TaskModel.find().populate('tags');
      deletedTag = await TagModel.findById(tagID);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
data in DB.', 500);
    }

    if (!deletedTag) {
      throw new HttpError("Couldn't find a tag for the provided tag ID!",
404);
    }

    try {
      const session: ClientSession = await startSession();
      session.startTransaction();

      for (const task of tasks) {
        await TaskModel.updateOne({ _id: task._id }, { $pull: { tags:
deletedTag._id } }, { session });
      }

      await deletedTag.deleteOne({ session });
      await session.commitTransaction();
    } catch (e) {
      throw new HttpError('Something went wrong while deleting tag.', 500);
    }

    return {
      success: true,
    };
  }
}

export default new TagService();
```

Контролер груп:

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

import GroupService from '../services/group-service.js';
import HttpError from '../exceptions/http-error.js';
import { IUserDataRequest } from '../ts/interfaces/IUserDataRequest.js';

const getGroups = async (req: Request, res: Response, next: NextFunction) =>
{
  try {
    const groupsData = await GroupService.getGroups();

    return res.status(200).json(groupsData);
  } catch (e) {
    next(e);
  }
};

const getGroupById = async (req: Request, res: Response, next: NextFunction)
```

```
=> {
  try {
    const groupID: string = req.params.groupID;

    const groupData = await GroupService.getGroupByID(groupID);

    return res.status(200).json(groupData);
  } catch (e) {
    next(e);
  }
};

const createGroup = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Create group validation error. Please,
check your credentials.', errors.array()));
    }

    const { id: creatorID } = req.userData;
    const groupData = await GroupService.createGroup(creatorID, req.body);

    return res.status(200).json({ ...groupData, message: 'Group is
successfully created!' });
  } catch (e) {
    next(e);
  }
};

const updateGroupById = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Update group validation error. Please,
check your credentials.', errors.array()));
    }

    const groupID: string = req.params.groupID;
    const { id: userID } = req.userData;
    const userData = await GroupService.updateGroup(groupID, userID,
req.body);

    return res.status(200).json({ ...userData, message: 'Group is
successfully updated!' });
  } catch (e) {
    next(e);
  }
};

const deleteGroupById = async (req: IUserDataRequest, res: Response, next:
NextFunction) => {
  try {
    const groupID: string = req.params.groupID;
    const { id: userID } = req.userData;

    const taskData = await GroupService.deleteGroup(groupID, userID);
```

```
    return res.status(200).json({
      ...taskData,
      message: `Successful deleted group with ${groupID} ID.`,
    });
  } catch (e) {
    next(e);
  }
};

export { getGroups, getGroupById, createGroup, updateGroupById,
deleteGroupById };
```

Сервіс груп:

```
import { ClientSession, startSession, Types } from 'mongoose';

import { mongooseModel as GroupModel } from '../models/group-model.js';
import { mongooseModel as UserModel } from '../models/user-model.js';
import HttpError from '../exceptions/http-error.js';
import GroupDto from '../DTO/group-dto.js';
import { IGroup } from '../ts/interfaces/IGroup.js';

class GroupService {
  async getGroups() {
    let groups;

    try {
      groups = await GroupModel.find();
    } catch (e) {
      throw new HttpError('Something went wrong while fetching groups data
from DB.', 500);
    }

    return {
      groups: groups.map((g) => new GroupDto(g)),
      success: true,
    };
  }

  async getGroupByID(groupID: string) {
    let group;

    try {
      group = await GroupModel.findById(groupID);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
group by group ID.', 500);
    }

    if (!group) {
      throw new HttpError("Couldn't find a group for the provided group
ID!", 404);
    }

    const groupDTO: GroupDto = new GroupDto(group);

    return { group: groupDTO, success: true };
  }

  private async checkGroupData({ users }, creatorID: string | null = null) {
    let identifiedUsers, identifiedCreator;
```

```
    if (creatorID != null) {
      try {
        identifiedCreator = await UserModel.findById(creatorID);
      } catch (e) {
        throw new HttpError("Couldn't find group creator for provided id!",
500);
      }

      if (!identifiedCreator) throw new HttpError("Couldn't find group
creator for provided id.", 404);
    }

    if (users) {
      try {
        identifiedUsers = await Promise.all(
          users.map(async (u) => {
            let identifiedUser;

            try {
              identifiedUser = await UserModel.findById(u);
            } catch (e) {
              throw new HttpError("Couldn't find group for provided id!",
500);
            }

            if (!identifiedUser) throw new HttpError("Couldn't find group
for provided id.", 404);
          }),
        );
      } catch (e) {
        throw new HttpError(e.message, e.status);
      }
    }

    return {
      identifiedUsers,
    };
  }

  async createGroup(creatorID: Types.ObjectId, { name, description, image,
users, access }: IGroup) {
    let groupCheck, createdGroup;

    try {
      groupCheck = await this.checkGroupData({ users },
creatorID.toString());
    } catch (e) {
      throw new HttpError(e.message, e.status);
    }

    if (!(groupCheck instanceof HttpError)) {
      if (users) users.push(creatorID);
      else users = [creatorID];

      try {
        createdGroup = await GroupModel.create({
          name,
          description,
          image,
          users,
```

```
          creator: creatorID,
          access,
        });
      } catch (e) {
        throw new HttpError('Creating group failed!', 500);
      }

      const groupDTO: GroupDto = new GroupDto(createdGroup);

      return {
        group: groupDTO,
        success: true,
      };
    }

    throw new HttpError(groupCheck.message, groupCheck.status);
  }

  async updateGroup(groupID: string, userID: Types.ObjectId, { name,
description, image, users, access }: IGroup) {
    let groupCheck;

    try {
      groupCheck = await this.checkGroupData({ users });
    } catch (e) {
      throw new HttpError(e.message, e.status);
    }

    if (!(groupCheck instanceof HttpError)) {
      let updatedGroup;

      try {
        updatedGroup = await
GroupModel.findById(groupID).populate('creator').populate('users');
      } catch (e) {
        throw new HttpError('Something went wrong while searching for some
group by group ID.', 500);
      }

      if (!updatedGroup) {
        throw new HttpError("Couldn't find a group for the provided group
ID!", 404);
      }

      if (updatedGroup.creator != userID) {
        throw new HttpError("No access to change task. Different user and
creator id's.", 404);
      }

      if (name) updatedGroup.name = name;
      if (description) updatedGroup.description = description;
      if (image) updatedGroup.image = image;
      if (users) updatedGroup.users = users;
      if (access) updatedGroup.access = access;

      try {
        await updatedGroup.save();
      } catch (e) {
        throw new HttpError('Something went wrong while updating group.',
500);
      }
```

```
      const groupDTO: GroupDto = new GroupDto(updatedGroup);

      return {
        group: groupDTO,
        success: true,
      };
    }

    throw new HttpError(groupCheck.message, groupCheck.status);
  }

  async deleteGroup(groupID: string, userID: Types.ObjectId) {
    let deletedGroup;

    try {
      deletedGroup = await GroupModel.findById(groupID).populate('users');
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
group by group ID.', 500);
    }

    if (!deletedGroup) {
      throw new HttpError("Couldn't find a group for the provided group
ID!", 404);
    }

    if (deletedGroup.creator._id != userID) {
      throw new HttpError("No access to delete group. Different user and
creator id's.", 404);
    }

    try {
      const session: ClientSession = await startSession();
      session.startTransaction();

      for (const user of deletedGroup.users) {
        await UserModel.updateOne({ _id: user._id }, { $pull: { groups:
deletedGroup._id } }, { session });
      }

      await deletedGroup.deleteOne({ session });
      await session.commitTransaction();
    } catch (e) {
      throw new HttpError('Something went wrong while deleting task.', 500);
    }

    return {
      success: true,
    };
  }
}

export default new GroupService();
```

Контролер ролей:

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

import RoleService from '../services/role-service.js';
import HttpError from '../exceptions/http-error.js';
```

```
const getRoles = async (req: Request, res: Response, next: NextFunction) =>
{
  try {
    const rolesData = await RoleService.getRoles();

    return res.status(200).json(rolesData);
  } catch (e) {
    next(e);
  }
};

const getRoleById = async (req: Request, res: Response, next: NextFunction)
=> {
  try {
    const roleID: string = req.params.roleID;

    const roleData = await RoleService.getRoleByID(roleID);

    return res.status(200).json(roleData);
  } catch (e) {
    next(e);
  }
};

const createRole = async (req: Request, res: Response, next: NextFunction)
=> {
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Create role validation error. Please, check
your credentials.', errors.array()));
    }

    const roleData = await RoleService.createRole(req.body);

    return res.status(200).json({ ...roleData, message: 'Role is
successfully created!' });
  } catch (e) {
    next(e);
  }
};

const updateRoleById = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      next(HttpError.BadRequest('Update role validation error. Please, check
your credentials.', errors.array()));
    }

    const roleID: string = req.params.roleID;

    const roleData = await RoleService.updateRole(roleID, req.body);

    return res.status(200).json({ ...roleData, message: 'Role is
successfully updated!' });
  } catch (e) {
```

```
    next(e);
  }
};

const deleteRoleById = async (req: Request, res: Response, next:
NextFunction) => {
  try {
    const roleID: string = req.params.roleID;

    const roleData = await RoleService.deleteRole(roleID);

    return res.status(201).json({
      ...roleData,
      message: `Successful deleted role with ${roleID} ID.`,
    });
  } catch (e) {
    next(e);
  }
};

export { getRoles, getRoleById, createRole, updateRoleById, deleteRoleById
};
```

Сервіс ролей:

```
import { ClientSession, startSession } from 'mongoose';

import { mongooseModel as RoleModel } from '../models/role-model.js';
import { mongooseModel as UserModel } from '../models/user-model.js';
import HttpError from '../exceptions/http-error.js';
import RoleDto from '../DTO/role-dto.js';
import { IRole } from '../ts/interfaces/IRole.js';

class RoleService {
  async getRoles() {
    let roles;

    try {
      roles = await RoleModel.find();
    } catch (e) {
      throw new HttpError('Something went wrong while fetching roles data
from DB.', 500);
    }

    return {
      roles: roles.map((r) => new RoleDto(r)),
      success: true,
    };
  }

  async getRoleByID(roleID: string) {
    let role;

    try {
      role = await RoleModel.findById(roleID);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
role by role ID.', 500);
    }

    if (!role) {
      throw new HttpError("Couldn't find a role for the provided role ID!",
```

```
404);
    }

    const roleDTO: RoleDto = new RoleDto(role);
    return { role: roleDTO, success: true };
  }

  async createRole({ name, description }: IRole) {
    let createdRole;

    try {
      createdRole = await RoleModel.create({
        name,
        description,
      });
    } catch (e) {
      throw new HttpError('Creating role failed!', 500);
    }

    const roleDTO: RoleDto = new RoleDto(createdRole);

    return {
      role: roleDTO,
      success: true,
    };
  }

  async updateRole(roleID: string, { name, description }: IRole) {
    let updatedRole;

    try {
      updatedRole = await RoleModel.findById(roleID);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
role by role ID.', 500);
    }

    if (!updatedRole) {
      throw new HttpError("Couldn't find a role for the provided role ID!",
404);
    }

    if (name) updatedRole.name = name;
    if (description) updatedRole.description = description;

    try {
      await updatedRole.save();
    } catch (e) {
      throw new HttpError('Something went wrong while updating role.', 500);
    }

    const roleDTO: RoleDto = new RoleDto(updatedRole);

    return {
      role: roleDTO,
      success: true,
    };
  }

  async deleteRole(roleID: string) {
    let deletedRole, users;
```

```
    try {
      users = await UserModel.find().populate('roles');
      deletedRole = await RoleModel.findById(roleID);
    } catch (e) {
      throw new HttpError('Something went wrong while searching for some
data in DB.', 500);
    }

    if (!deletedRole) {
      throw new HttpError("Couldn't find a role for the provided role ID!",
404);
    }

    try {
      const session: ClientSession = await startSession();
      session.startTransaction();

      for (const user of users) {
        await UserModel.updateOne({ _id: user._id }, { $pull: { roles:
deletedRole._id } }, { session });
      }

      await deletedRole.deleteOne({ session });
      await session.commitTransaction();
    } catch (e) {
      throw new HttpError('Something went wrong while deleting role.', 500);
    }

    return {
      success: true,
    };
  }

  async createUserRole() {
    let userRole;

    try {
      userRole = await RoleModel.findOne({ name: 'user' });
    } catch (e) {
      throw new HttpError('Something went wrong while searching for user
role. Please try again later.', 500);
    }

    if (!userRole) {
      try {
        userRole = await RoleModel.create({
          name: 'user',
          description: 'Just a user with default access. Nothing special',
        });

        console.log(userRole);
      } catch (e) {
        throw new HttpError('Something went wrong while creating the non-
existing user role.', 500);
      }
    }

    return userRole;
  }
```

```
  async createAdminRole() {
    let adminRole;

    try {
      adminRole = await RoleModel.findOne({ name: 'admin' });
    } catch (e) {
      throw new HttpError('Something went wrong while searching for admin
role. Please try again later.', 500);
    }

    if (!adminRole) {
      try {
        adminRole = await RoleModel.create({
          name: 'admin',
          description: 'Boss of the gym!',
        });
      } catch (e) {
        throw new HttpError('Something went wrong while creating the non-
existing admin role.', 500);
      }
    }

    return adminRole;
  }

  async createOwnerRole() {
    let ownerRole;

    try {
      ownerRole = await RoleModel.findOne({ name: 'owner' });
    } catch (e) {
      throw new HttpError('Something went wrong while searching for owner
role. Please try again later.', 500);
    }

    if (!ownerRole) {
      try {
        ownerRole = await RoleModel.create({
          name: 'owner',
          description: 'Group owner.',
        });
      } catch (e) {
        throw new HttpError('Something went wrong while creating the non-
existing owner role.', 500);
      }
    }

    return ownerRole;
  }

  async createRolesIfNotExist() {
    try {
      await this.createAdminRole();
      await this.createUserRole();
      await this.createOwnerRole();
    } catch (e) {
      throw e;
    }
  }
}
```

```
export default new RoleService();
```

Контролер іншого:

```
import { NextFunction, Request, Response } from 'express';
import { spawn } from 'child_process';
import path, { dirname } from 'path';
import cron from 'node-cron';
import { fileURLToPath } from 'url';

import HttpError from '../exceptions/http-error.js';

const DB_NAME: string | undefined = process.env.DB_NAME || 'task-management-
tool';
const ARCHIVE_PATH =
path.join(dirname(dirname(dirname(fileURLToPath(import.meta.url)))),
'public', `${DB_NAME}.gzip`);

const backupMongoDB = (): Promise<{ message: string; status: number }> => {
  return new Promise((resolve, reject) => {
    const child = spawn('mongodump', [`--db=${DB_NAME}`, `--
archive=${ARCHIVE_PATH}`, '--gzip']);

    child.stdout.on('data', (data) => {
      console.log('stdout:\n' + data);
    });

    child.stderr.on('data', (data) => {
      console.log('stderr:\n' + data);
    });

    child.on('error', (err) => {
      console.log('err:\n' + err);
      reject(new HttpError('Something went wrong while backing up DB.',
500));
    });

    child.on('exit', (code, signal) => {
      if (code) {
        reject(new HttpError(`Process exit with code: ${code}`, 500));
      } else if (signal) {
        reject(new HttpError(`Process killed with signal: ${signal}`, 500));
      } else {
        resolve({ message: 'DB backing up is successful <3', status: 200 });
      }
    });
  });
};

const restoreMongoDB = (): Promise<{ message: string; status: number }> => {
  return new Promise((resolve, reject) => {
    const child = spawn('mongorestore', [`--db=${DB_NAME}`, `--
archive=${ARCHIVE_PATH}`, '--gzip']);

    child.stdout.on('data', (data) => {
      console.log('stdout:\n' + data);
    });

    child.stderr.on('data', (data) => {
      console.log('stderr:\n' + data);
    });
```

```
    child.on('error', (err) => {
      console.log('err:\n' + err);
      reject(new HttpError('Something went wrong while restoring DB.',
500));
    });

    child.on('exit', (code, signal) => {
      if (code) {
        reject(new HttpError(`Process exit with code: ${code}`, 500));
      } else if (signal) {
        reject(new HttpError(`Process killed with signal: ${signal}`, 500));
      } else {
        resolve({ message: 'DB restore is successful <3', status: 200 });
      }
    });
  });
};

const backup = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const response = await backupMongoDB();
    console.log(response);

    return res.status(response.status).json({ message: response.message,
success: true });
  } catch (error) {
    next(error);
  }
};

const restore = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const response = await restoreMongoDB();

    return res.status(response.status).json({ message: response.message,
success: true });
  } catch (error) {
    next(error);
  }
};

cron.schedule('0 0 * * *', async () => {
  try {
    await backupMongoDB();
    await restoreMongoDB();
  } catch (error) {
    console.error('An error occurred during backup and restore:', error);
  }
});

export { backup, restore };
```

Сервіс пошти:

```
import { createTransport } from "nodemailer";

class MailService {
  transporter;

  constructService() {
    this.transporter = createTransport({
```

```
      host: process.env.SMTP_HOST,
      port: Number(process.env.SMTP_PORT),
      secure: false,
      service: "Gmail",
      auth: {
        user: process.env.SMTP_USER,
        pass: process.env.SMTP_PASSWORD,
      },
    });
  }

  async sendActivationMail(to: string, link: string) {
    this.constructService();

    await this.transporter.sendMail({
      from: process.env.SMTP_USER,
      to,
      subject: "Account activation on " + process.env.API_URL,
      text: "",
      html: `
        <div>
          <h1>To activate the account go to this link:</h1>
          <a href="${link}">${link}</a>
        </div>
            `,
    });
  }
}

export default new MailService();
```

Сервіс картинок:

```
import { existsSync, unlinkSync } from 'fs';

class ImageService {
  deleteImage(modelObject, defaultFilePath: string) {
    if (modelObject.image && modelObject.image != defaultFilePath) {
      const filePath = modelObject.image;

      if (existsSync(filePath)) unlinkSync(filePath);
      else console.log('No image in folder.');
    } else {
      console.log('No image field in model object.');
    }
  }
}

export default new ImageService();
```

Сервіс токенів:

```
import jwt from 'jsonwebtoken';

import { mongooseModel as TokenModel } from '../models/token-model.js';
import HttpError from '../exceptions/http-error.js';

class TokenService {
  generateTokens(payload) {
    const accessToken: string = jwt.sign(payload,
process.env.JWT_ACCESS_SECRET, {
      expiresIn: '1h',
    });
```

```
    const refreshToken: string = jwt.sign(payload,
process.env.JWT_REFRESH_SECRET, {
      expiresIn: '30d',
    });

    return {
      accessToken,
      refreshToken,
    };
  }

  validateRefreshToken(refreshToken: string) {
    let userData;

    try {
      userData = jwt.verify(refreshToken, process.env.JWT_REFRESH_SECRET);
    } catch (e) {
      return null;
    }

    return userData;
  }

  validateAccessToken(accessToken: string) {
    let userData;

    try {
      userData = jwt.verify(accessToken, process.env.JWT_ACCESS_SECRET);
    } catch (e) {
      return null;
    }

    return userData;
  }

  async saveToken(userID: string, refreshToken: string) {
    const tokenData = await TokenModel.findOne({ user: userID });

    if (tokenData) {
      tokenData.refreshToken = refreshToken;
      return tokenData.save();
    }

    return TokenModel.create({ user: userID, refreshToken });
  }

  async findToken(refreshToken: string) {
    let tokenData;

    try {
      tokenData = await TokenModel.findOne({ refreshToken });
    } catch (e) {
      throw new HttpError("Can't find refresh token in DB! Please, try
again.", 500);
    }

    return tokenData;
  }

  async removeToken(refreshToken: string) {
    let tokenData;
```

```
    try {
      tokenData = await TokenModel.deleteOne({ refreshToken });
    } catch (e) {
      throw new HttpError("Can't delete refresh token from DB! Please, try
again.", 500);
    }

    return tokenData;
  }
}

export default new TokenService();
```